



01076105, 01076106

Object Oriented Programming

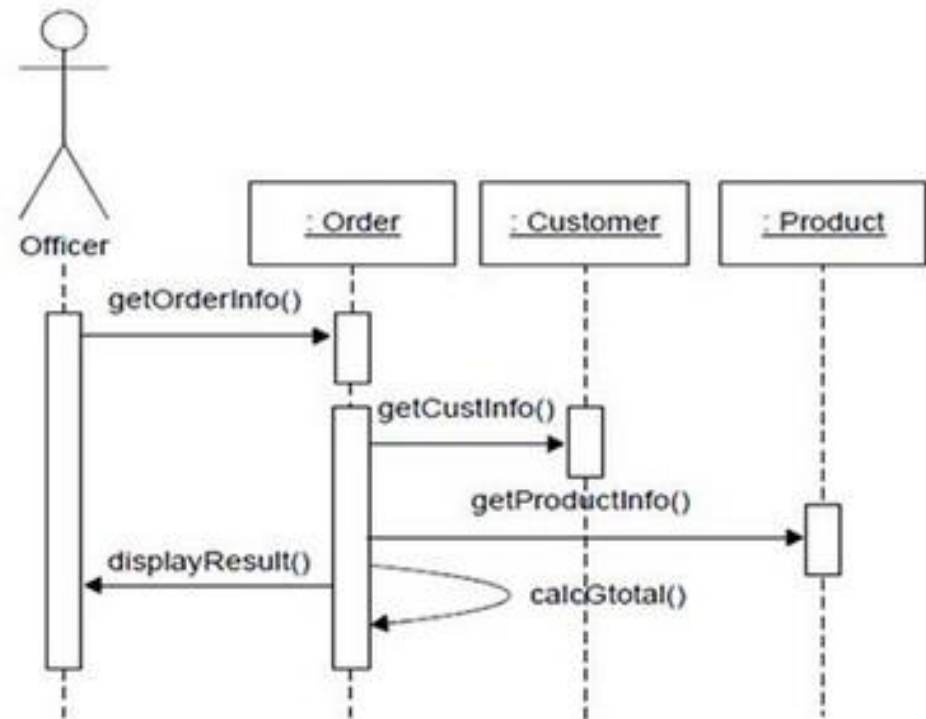
Object Oriented Programming Project

Sequence Diagram



# Sequence Diagram


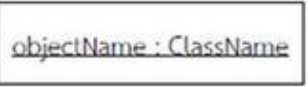


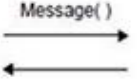

- แสดงลำดับการทำงานของระบบ  
แสดงปฏิสัมพันธ์ (interaction)  
ระหว่าง object ตามลำดับของ  
เหตุการณ์ที่เกิดขึ้น
- message ที่เกิดขึ้นระหว่าง class จะ  
สามารถนำไปสู่การสร้าง method  
ใน class ที่เกี่ยวข้องได้





# Sequence Diagram

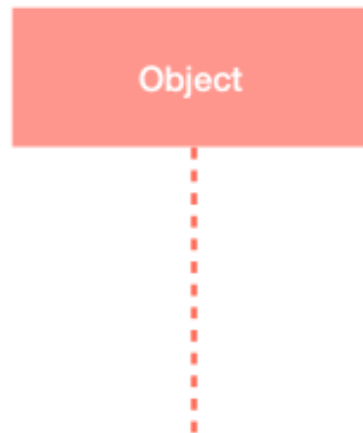
- Actor เป็น actor เดียวกับใน use case diagram
- Object ต้องเป็น class เดียวกับ class diagram
- Lifeline แทนเส้นเวลาเหตุการณ์
- Activation bar แสดงขอบเขตหรืออายุการทำงานของ event นั้นๆ
- Message แสดงชื่อ method ที่เรียกใช้ และผลลัพธ์ (ถ้ามี)
- Call back สำหรับการคืนมาใน object เดียวกัน

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)
  - คือเส้นชีวิตของวัตถุหรือ class เป็นตัวแทนของวัตถุหรือส่วนประกอบต่างๆที่มีปฏิสัมพันธ์ซึ่งกันและกันในระบบในลำดับต่างๆ
  - format การเขียนชื่อเส้นชีวิตคือ Instance Name:Class Name
  - เส้นชีวิตกับสัญลักษณ์ actor จะใช้เมื่ออย่างใดอย่างหนึ่งเป็นส่วนหนึ่งของ use case นั้น





# Sequence Diagram

- Activation Bars

- Activation bars จะวางอยู่บนเส้นชีวิตเพื่อแสดงการโต้ตอบระหว่าง object, function หรือ module ความยาวของสี่เหลี่ยมจะแสดงระยะเวลาการโต้ตอบของ object หรือ จุดเริ่มต้นและจุดสิ้นสุดของแต่ละกิจกรรมของ object นั้น
- การโต้ตอบระหว่าง 2 object เกิดเมื่อวัตถุหนึ่งส่ง message ไปให้อีก object โดย object ที่ส่งข้อความเรียกว่า message caller และ object ที่รับข้อความเรียกว่า message receiver เมื่อมีแถบ activation บนเส้นชีวิตของวัตถุ นั้นหมายความว่า object นั้นมีการทำงานในขณะที่ส่งข้อความโต้ตอบกัน





# 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

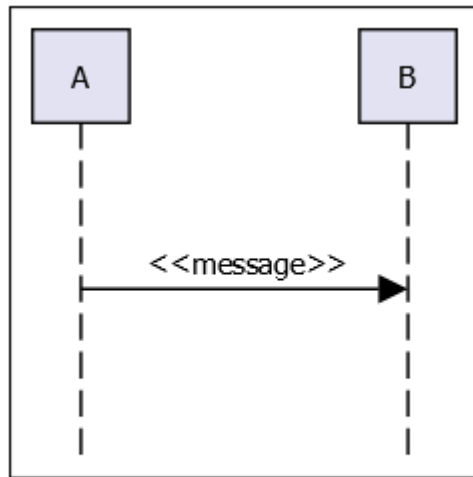
- Synchronous message จะถูกใช้เมื่อ object ที่ส่งข้อความรอให้ object ที่รับ message ประมวลผลและส่ง return กลับมา ก่อนที่จะส่ง message อันต่อไป หัวลูกศรที่ใช้จะเป็นลูกศรแบบทึบ
- Asynchronous message จะถูกใช้เมื่อ object ที่ส่ง message ไม่รอให้ object ที่รับ message ประมวลผลข้อความและส่งค่า return กลับมา แต่จะส่งข้อความต่อไปให้แก่วัตถุอื่นในระบบเลย หัวลูกศรที่แสดงในข้อความประเภทนี้เป็นหัวลูกศรเส้น



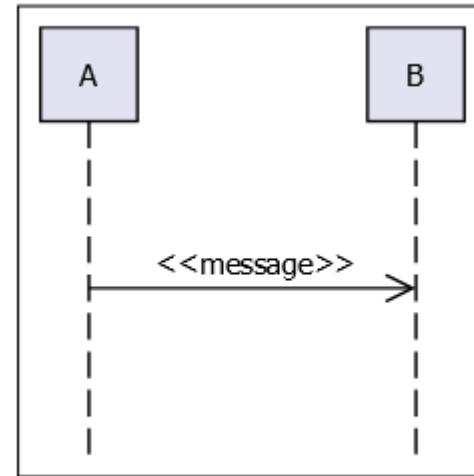


# Sequence Diagram

- ตัวอย่างของ synchronous message และ asynchronous message



A synchronous message



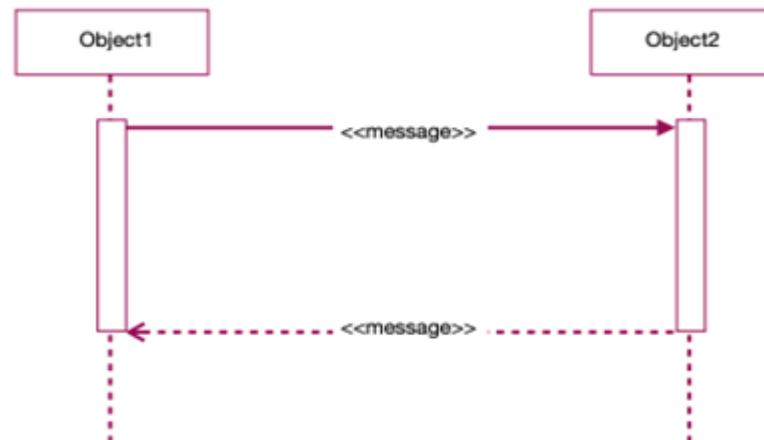
An asynchronous message



# Sequence Diagram

- Activation Bars

- return message ใช้เพื่อระบุว่า object รับ message และประมวลผล message เสร็จสิ้นแล้ว และกำลังส่งคืนการควบคุมไปยัง object ที่ทำหน้าที่ส่ง message
- return message เป็นตัวเลือก ที่จะเลือกให้มีหรือไม่มีก็ได้ สำหรับการส่ง message บนแถบ activation ด้วย synchronous message จะให้ความหมายโดยนัยว่ามี return message ด้วยแม้จะไม่ได้มีเส้น return message แสดงก็ตาม
- เราสามารถหลีกเลี่ยงการทำให้แผนภาพดูยุ่งเหยิงโดยการไม่ใช้ return message เมื่อไม่จำเป็น

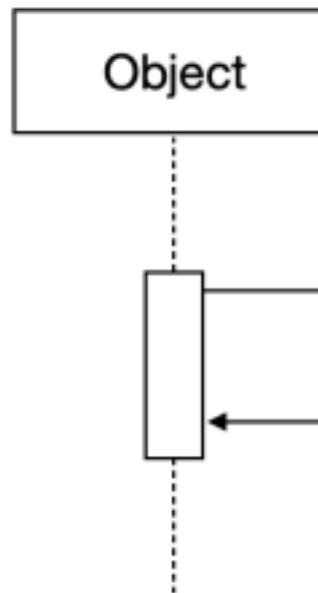






# Sequence Diagram

- Activation Bars
  - Reflexive message เมื่อ object ส่งข้อความหาตัวเอง จะเรียกว่า reflexive message แสดงข้อความประเภทนี้โดยใช้ message arrow ที่เริ่มจากจบที่เส้นชีวิตเดียวกัน อย่างตัวอย่างด้านล่างนี้



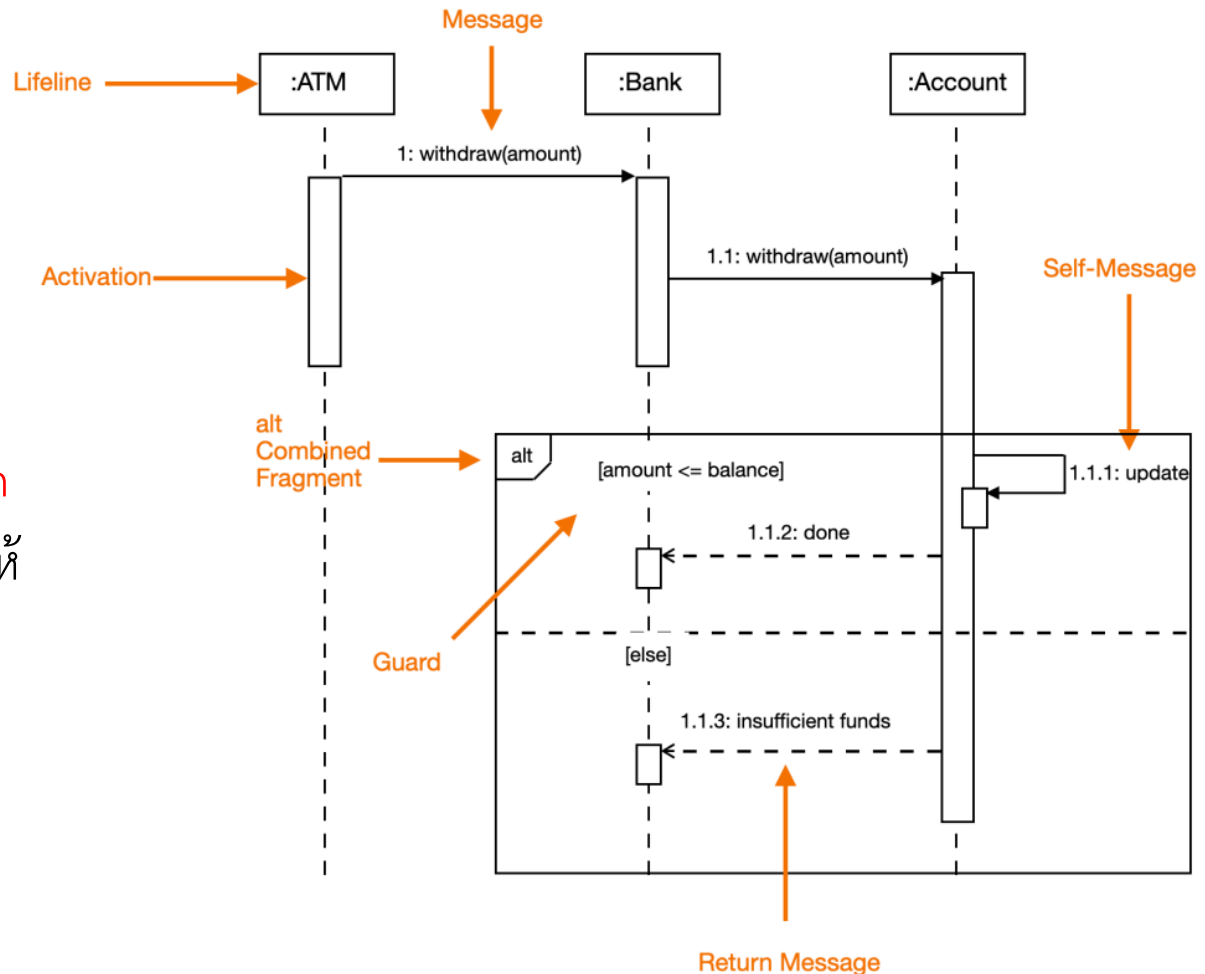


# Sequence Diagram

- Sequence Fragment

- คือกล่องที่มีเครื่องหมายแสดง section การโต้ตอบระหว่างวัตถุใน sequence diagram

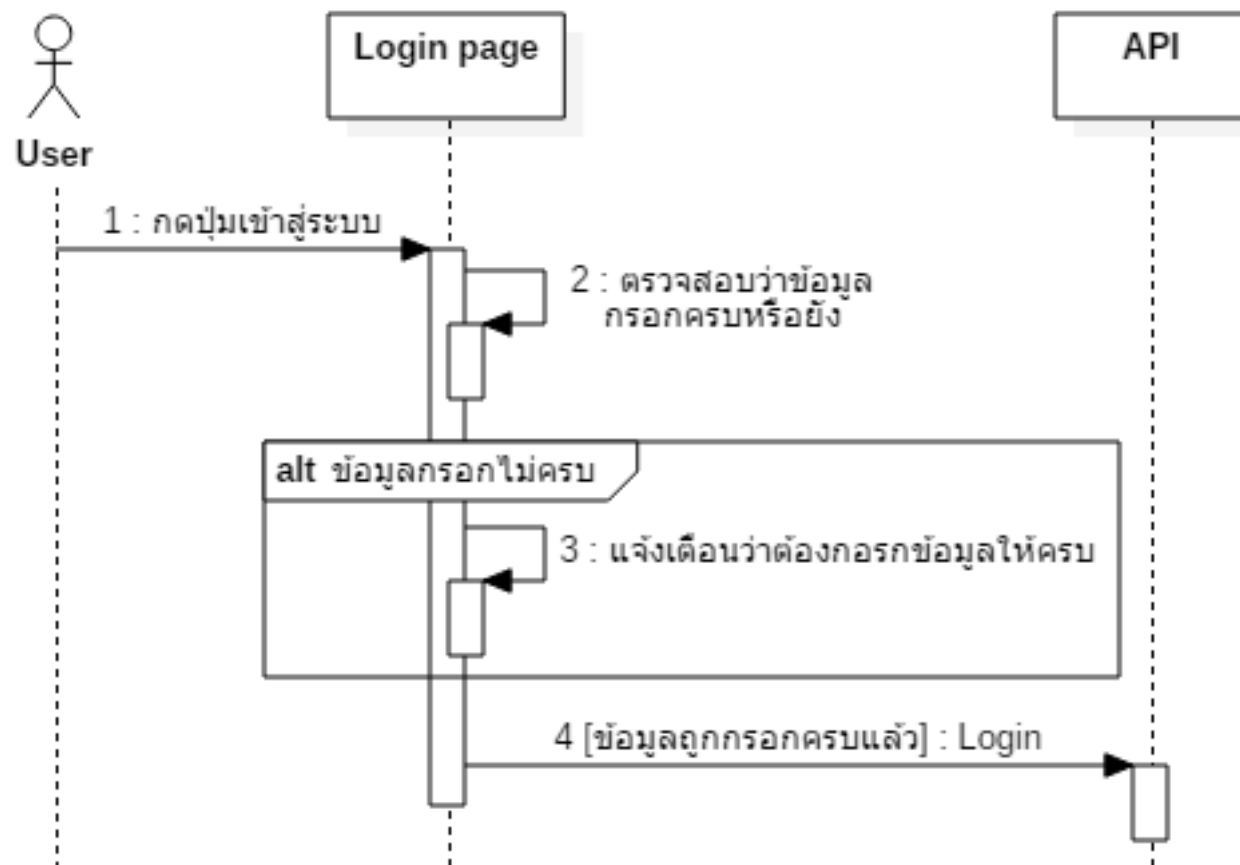
- Alternative combination fragment** ใช้เมื่อมีตัวเลือกให้เลือกตั้งแต่ 2 ตัวเลือกขึ้นไป ใช้ตรรกะแบบ “if then else”





# Sequence Diagram

- ตัวอย่าง Alternative Fragment



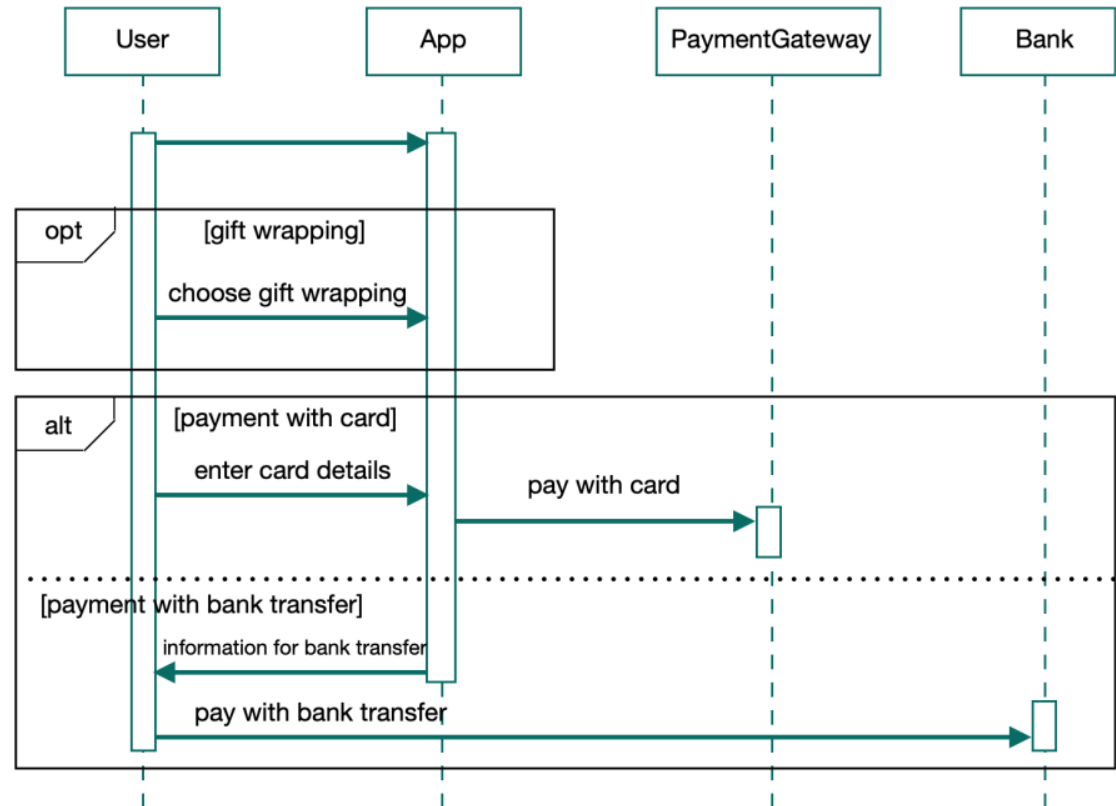


# Sequence Diagram

- Options

Combination  
fragment

- ใช้เพื่อแสดงถึงลำดับเหตุการณ์ที่เกิดขึ้นภายใต้เงื่อนไขใดเงื่อนไขหนึ่งเท่านั้น ไม่เช่นนั้นเหตุการณ์นั้นจะไม่สามารถเกิดขึ้นได้ ใช้ตรรกะแบบ 'if then'

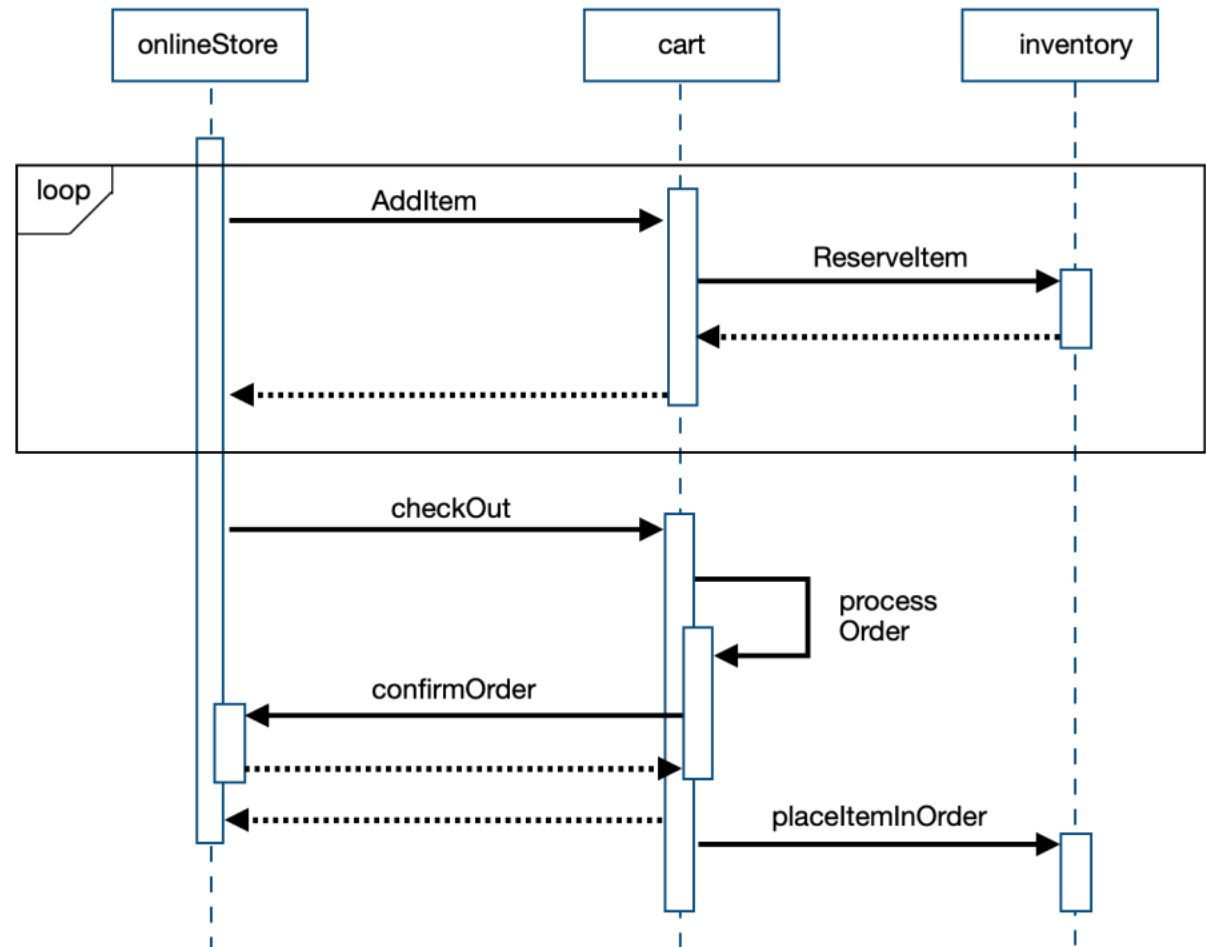




# Sequence Diagram

- **Loop fragment**

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



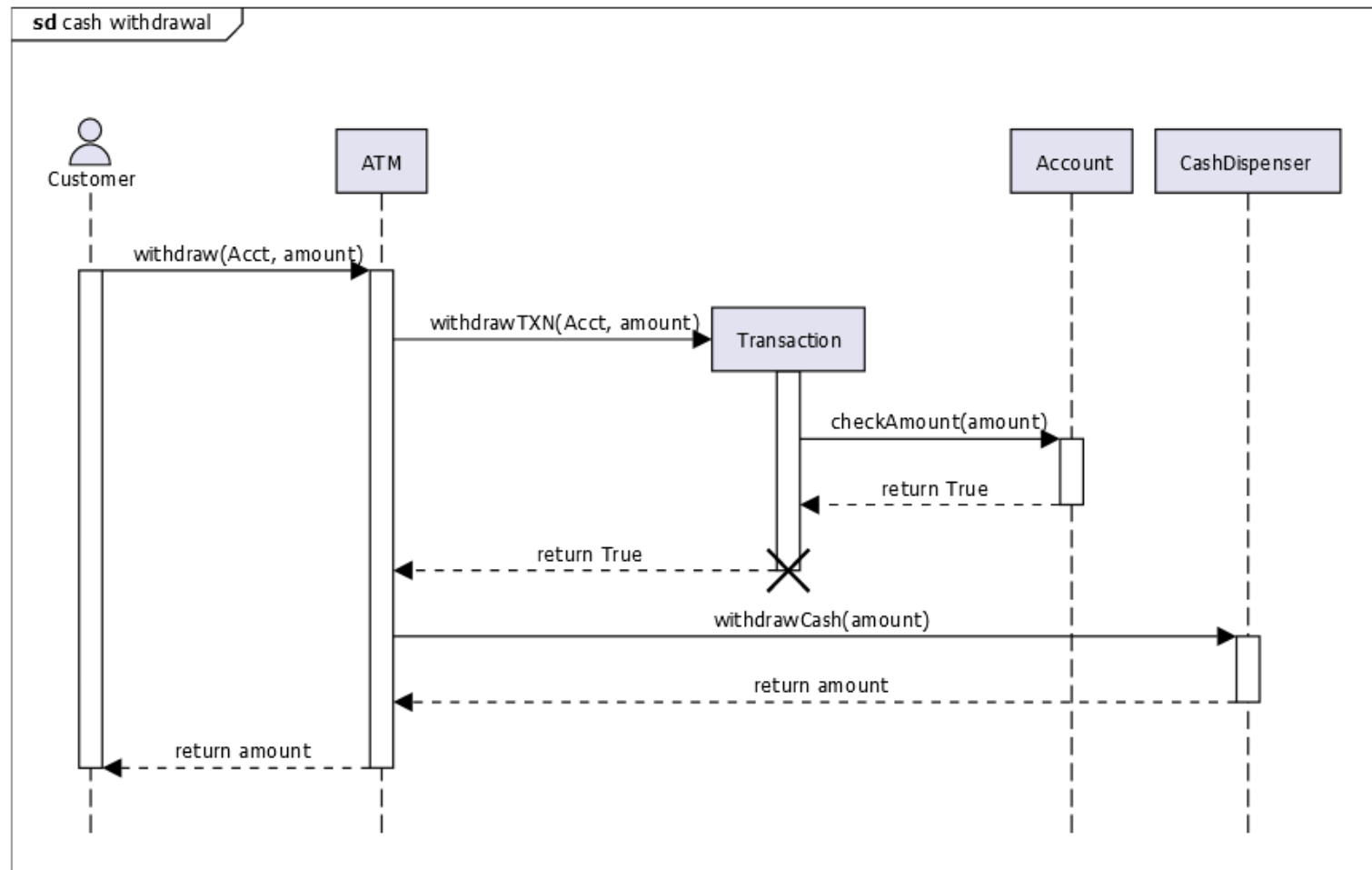


# Sequence Diagram

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

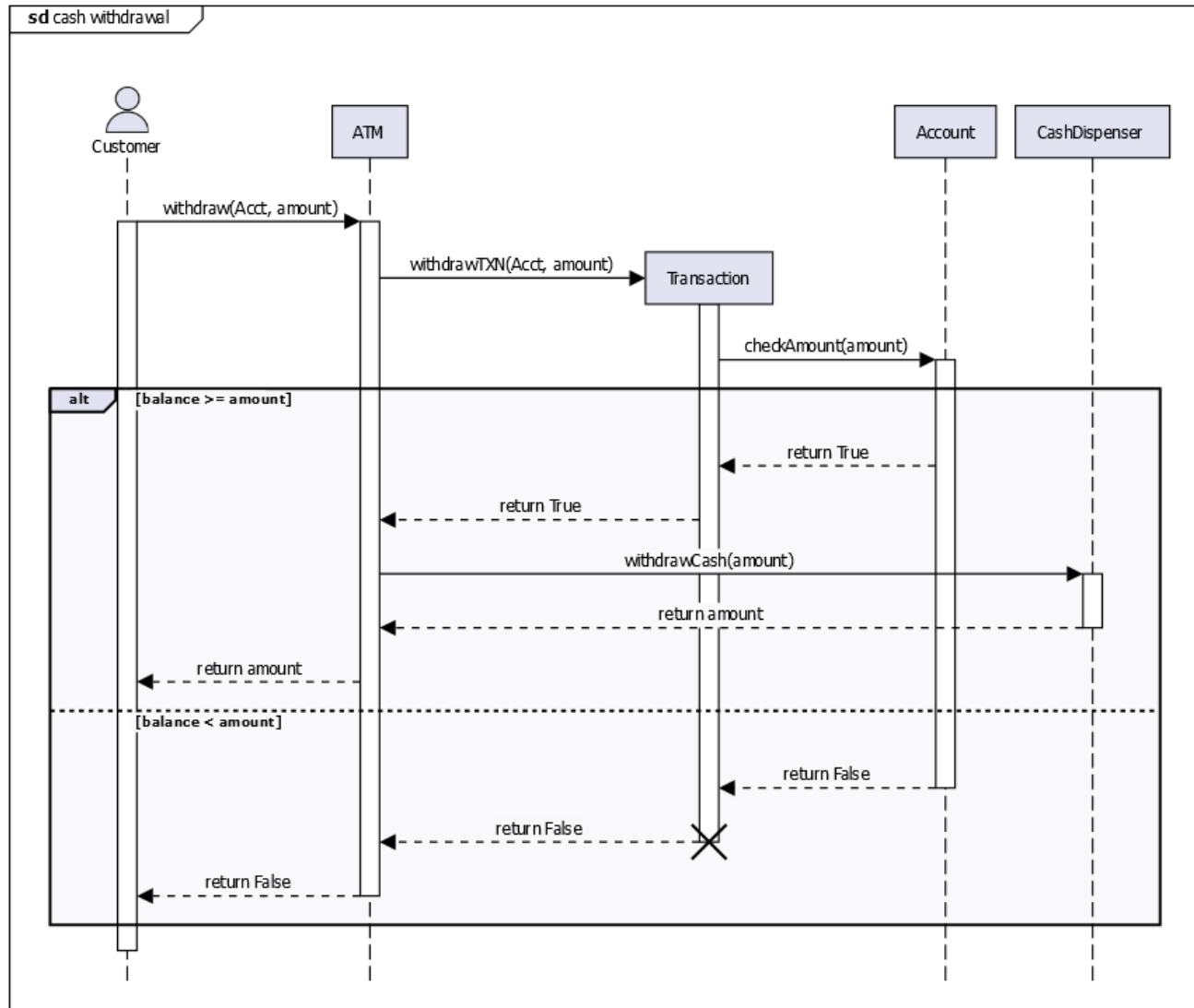


# Sequence Diagram : example





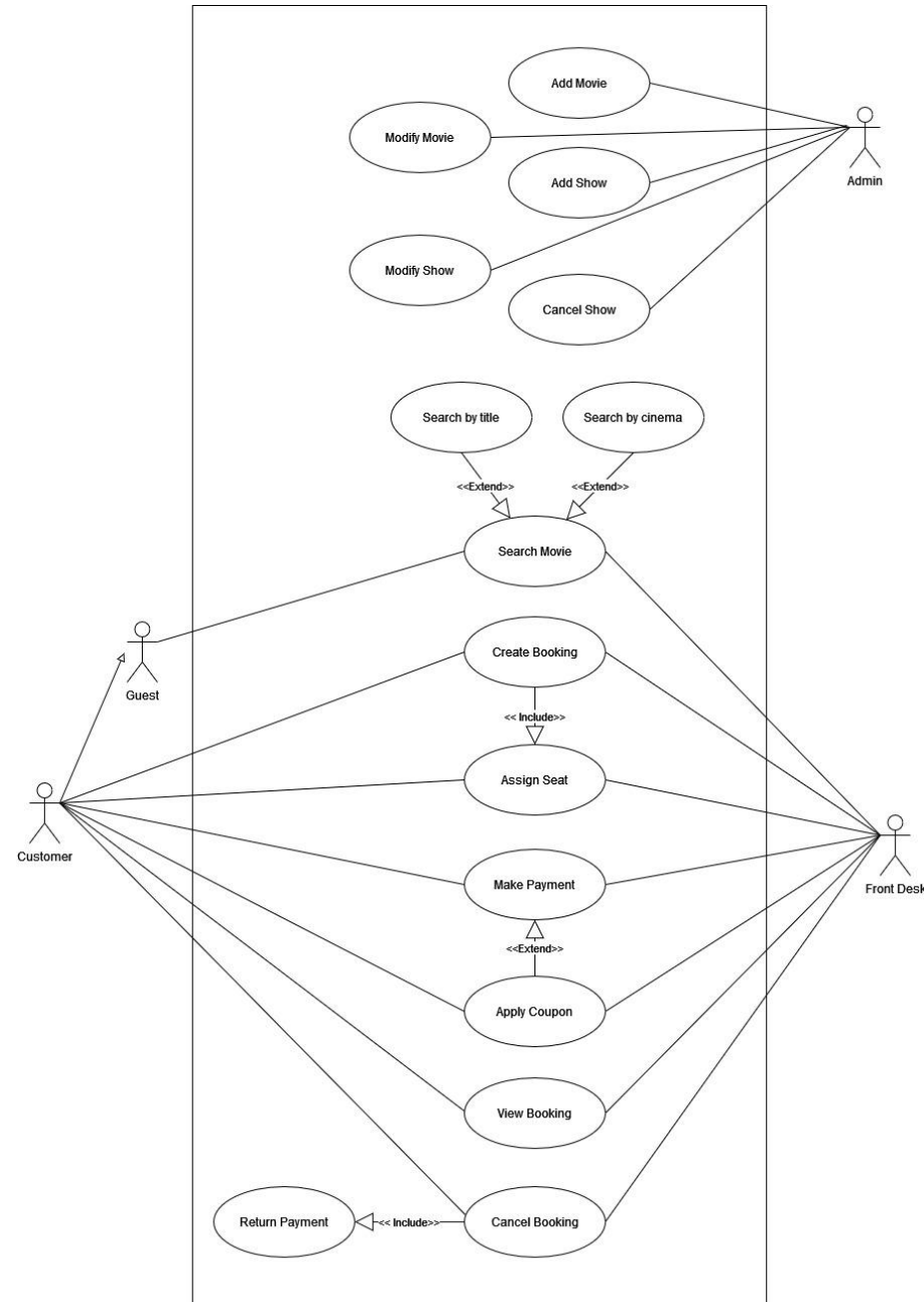
# Sequence Diagram : example





# Sequence Diagram : Movie

- ในการสร้าง sequence diagram จะต้องสร้างทุก use case
- อย่างไรก็ตาม ใน use case ที่การทำงานไม่ซับซ้อน อาจไม่ต้องเขียน sequence diagram ก็ได้





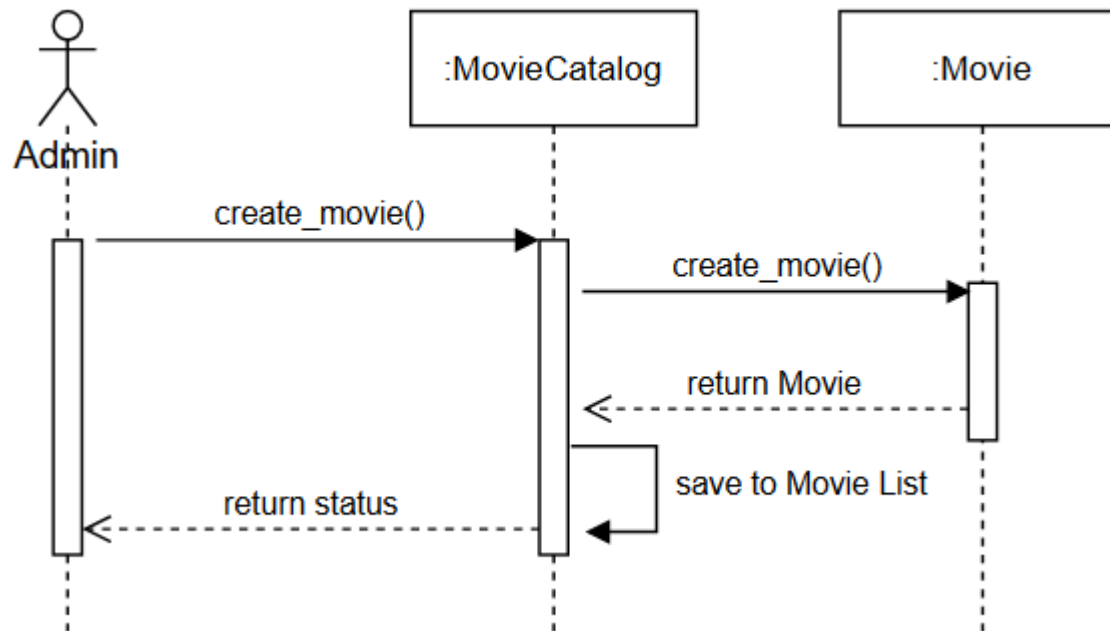
## Sequence Diagram : Movie :: Add Movie

- ระบุ Actor : Admin
- ระบุ Object : MovieCatalog, Movie
- ขั้นตอน
  1. Admin เปิดหน้าจอ ป้อนข้อมูลภาพยนตร์ ประกอบด้วย
    - ชื่อภาพยนตร์
    - หมวดหมู่ กำหนดสูงสุด 3 หมวดหมู่
    - เรทผู้ชม
    - เวลาฉาย
    - รายละเอียด
    - วันที่เข้าฉาย
  2. นำข้อมูลสร้าง object Movie และนำไปเก็บใน MovieCatalog



## Sequence Diagram : Movie :: Add Movie

- Sequence diagram ของ use case add movie
- ให้ระลึกเสมอว่า ทุก object ที่สร้างขึ้น จะต้องมีการเก็บเสมอ



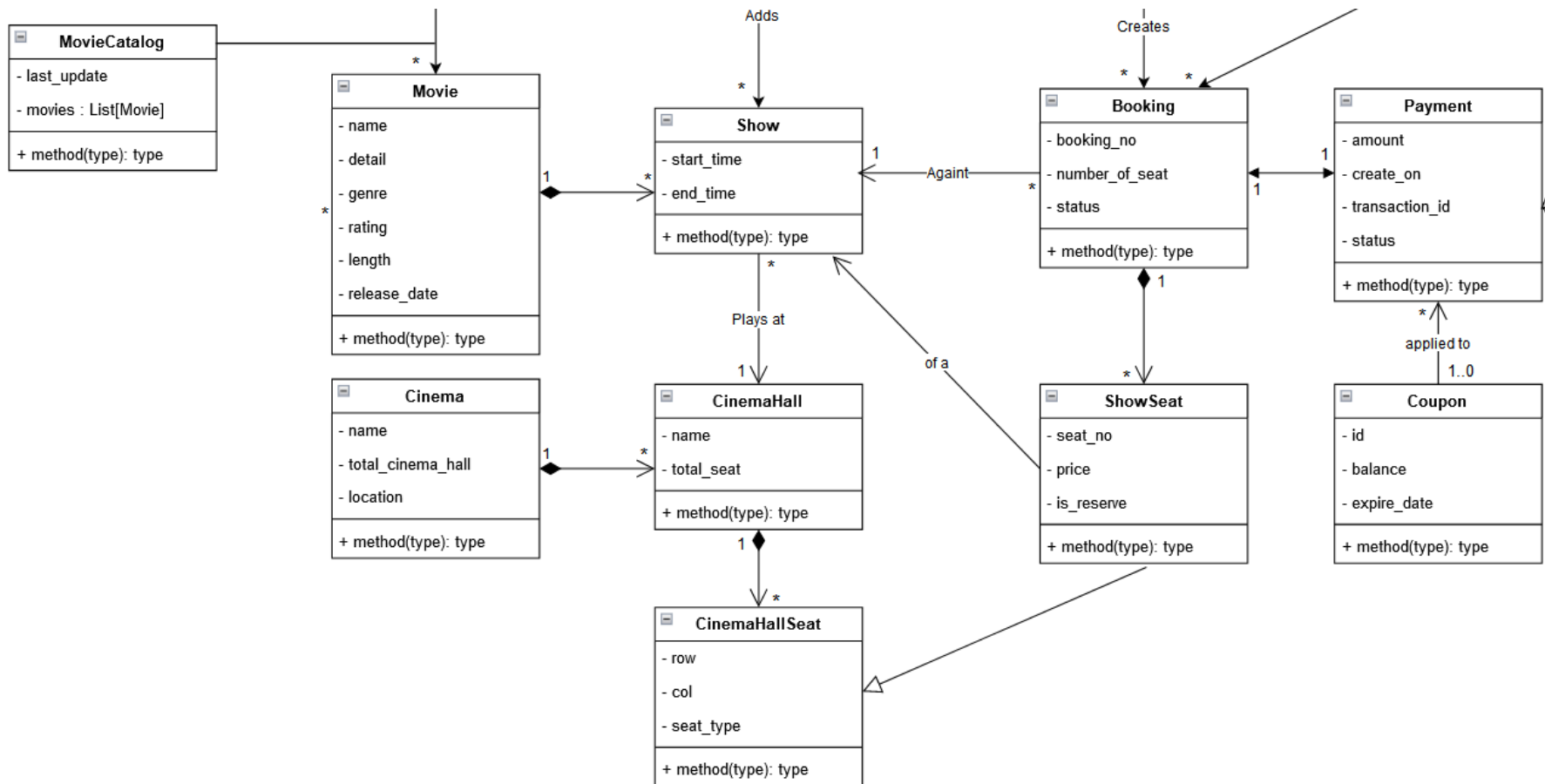


## Sequence Diagram : Movie :: Add Show

- ระบุ Actor : Admin
- ระบุ Object : Movie, CinemaHall, Show
- ขั้นตอน
  1. เลือกภาพยนตร์ เลือกสาขาโรงภาพยนตร์ และ เลือกโรงย่อย
  2. ป้อนวันฉาย เวลาเริ่มฉาย เวลาสิ้นสุดการฉาย
  3. บันทึกข้อมูลรอบฉายในคลาส Movie



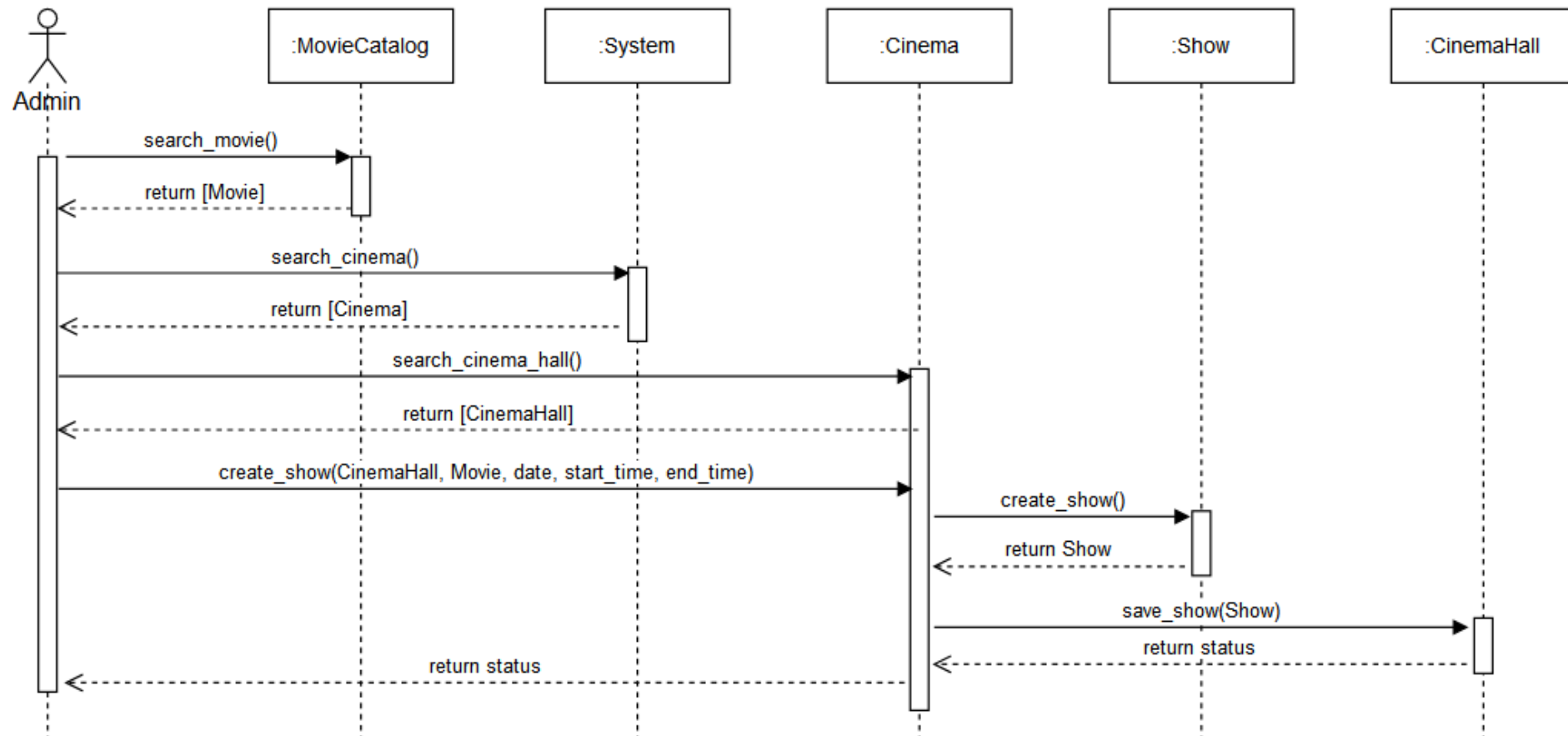
# Sequence Diagram : Movie :: Add Show





# Sequence Diagram : Movie :: Add Show

- Sequence diagram ของ use case add show





## Sequence Diagram : Movie :: Booking

- ระบุ Actor : Customer, FrontDeskOfficer
- ระบุ Object : MovieCatalog, Movie, Show
- ขั้นตอน
  1. **Precondition** : ผู้ใช้ค้นหาภาพยนตร์จาก MovieCatalog เมื่อผู้ใช้เลือกภาพยนตร์ ผู้ใช้จะทราบ object ของ Movie
  2. ผู้ใช้ request ข้อมูล ชื่อภาพยนตร์ หมวดหมู่ เรตติ้ง ระยะเวลาฉาย วันที่เริ่มฉาย จาก Object Movie (get\_movie\_detail)



**แอนท-แมน และ เดอะ วอสพ์ : อะลุ่มมิตควอนตัม**

หมวดหมู่: Action, Adventure, Comedy

เรตติ้ง: 13 | ⌚ 125 นาที

รายละเอียดภาพยนตร์



## Sequence Diagram : Movie :: Booking

- ขั้นตอน

3. จากข้อมูลวันที่เริ่มฉาย ให้แสดงวันจำนวน 6 วัน นับจากวันที่เริ่มฉาย หรือ วันปัจจุบัน

พุธ	พฤหัสบดี	ศุกร์	เสาร์	อาทิตย์	จันทร์
15 ก.พ. 2023	16 ก.พ. 2023	17 ก.พ. 2023	18 ก.พ. 2023	19 ก.พ. 2023	20 ก.พ. 2023

4. จากนั้นจะ request ข้อมูลรอบฉายของภาพยนตร์ จาก object Show โดยส่งข้อมูล ภาพยนตร์ และ วันที่ไปด้วย (get\_show\_time(movie, date)) โดยจะส่ง dictionary ของรอบฉาย เรียงตามประเภทโรงฉาย และ เวลา เพื่อมาแสดง หากมีการเลือกรอบ ก็จะเข้าสู่การทำงานของ Assign Seat หากเลือกวันอื่นก็จะดึงข้อมูลใหม่



 ENG |  | 

11:10

14:00

16:50

19:40

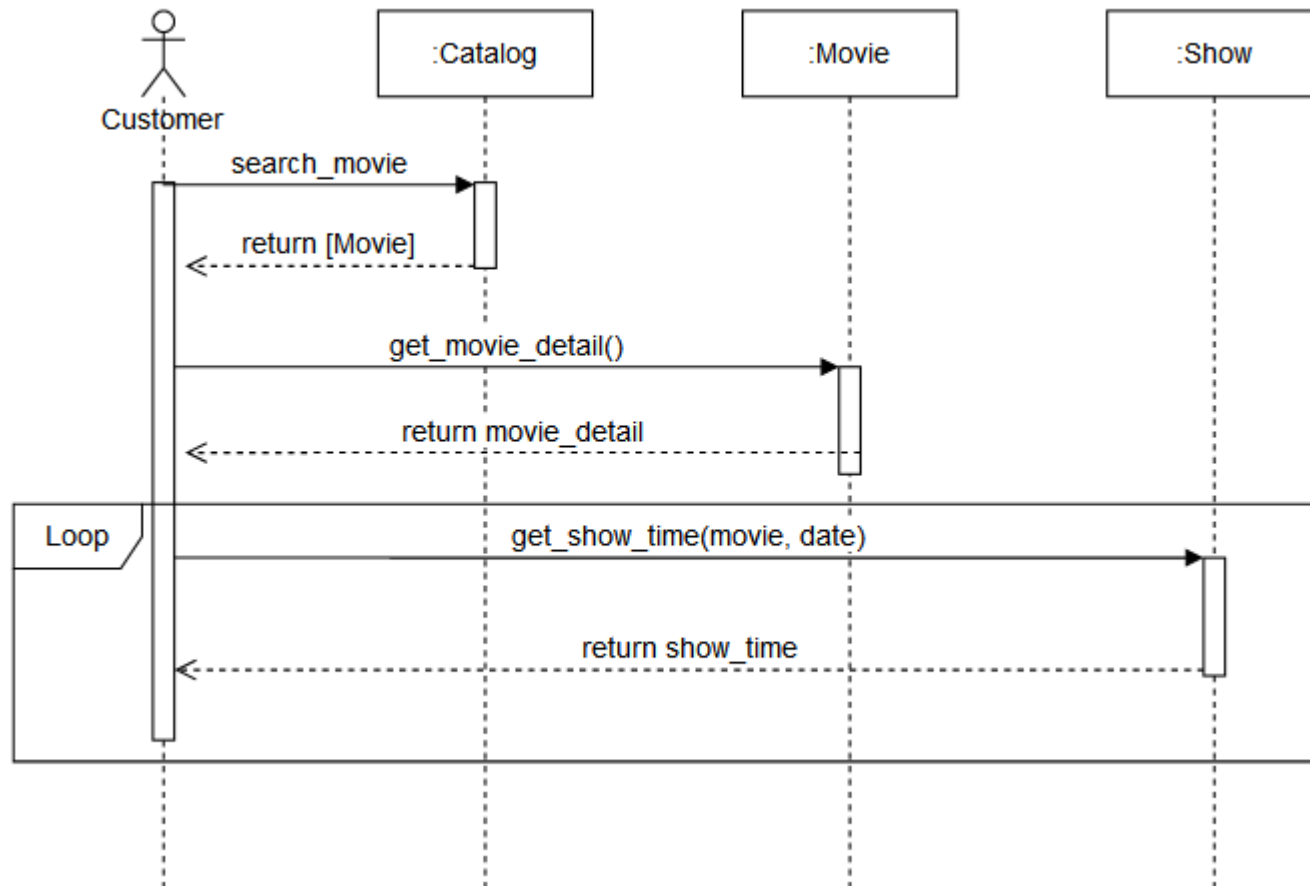
22:30





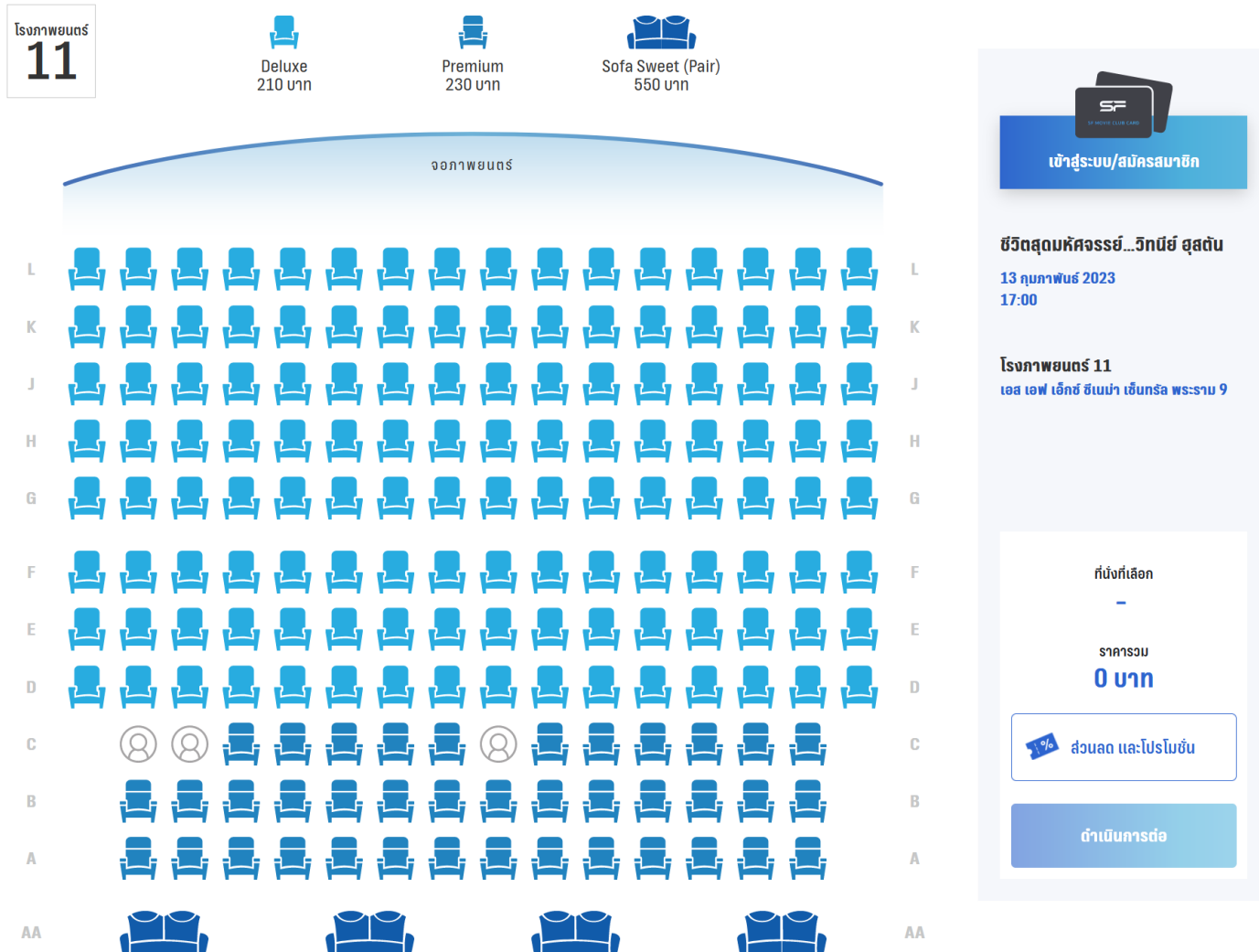
## Sequence Diagram : Movie :: Booking

- จากขั้นตอนที่ได้อธิบายมา สามารถเขียน sequence diagram ของ Booking ได้ดังนี้





# Sequence Diagram : Movie :: Assign Seat





## Sequence Diagram : Movie :: Assign Seat

- ระบุ Actor : Customer, FrontDeskOfficer
- ระบุ Object : Show, CinemaHall, CinemaHallSeat, ShowSeat
- ขั้นตอน
  1. **Precondition** : เมื่อผู้ใช้เลือกรอบภาพยนตร์ที่ต้องการดู
  2. Request ข้อมูลที่นั่งในโรง จาก object CinemaHall แต่เนื่องจาก object CinemaHall ไม่ได้เก็บที่นั่งโดยตรง แต่จะเก็บ List ของ object ที่นั่ง
  3. ดังนั้นจะ request ข้อมูลที่นั่งจากคลาส CinemaHallSeat อีกที โดยจะสร้าง method `get_seat_list` ขึ้นในคลาส CinemaHall โดยจะวนลูป `get` ค่าของที่นั่งในคลาส CinemaHallSeat มาเก็บไว้ใน dictionary

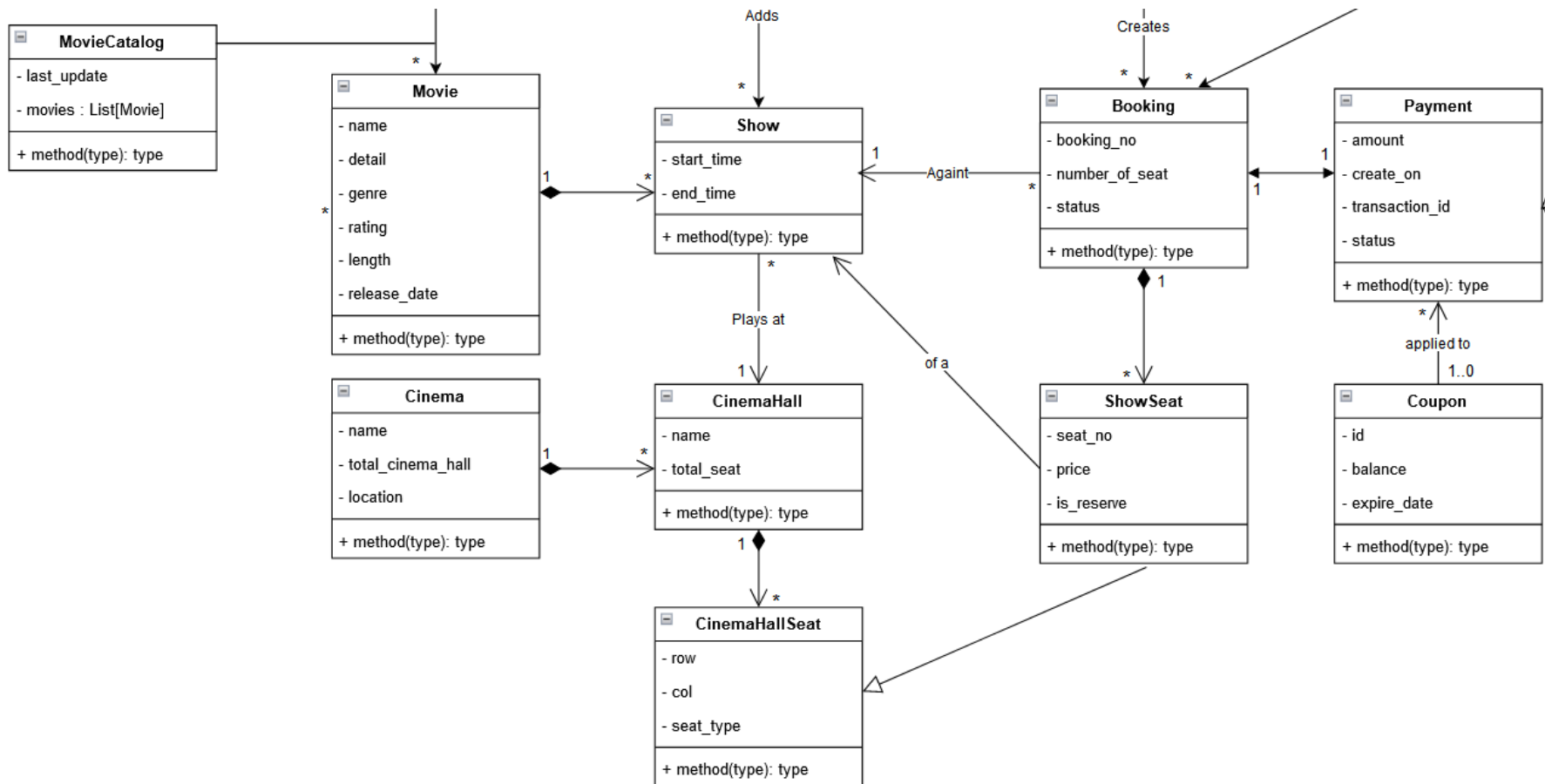


## Sequence Diagram : Movie :: Assign Seat

- ขั้นตอน
  4. เนื่องจากจะต้องแสดงด้วยว่าที่นั่งใดที่มีการจองแล้ว ซึ่งข้อมูลนั้นไม่ได้อยู่ใน object CinemaHallSeat แต่อยู่ใน object ShowSeat ดังนั้นจึงต้อง request ไปที่ Show เพื่อหาข้อมูล Booking และต่อไปที่ object ShowSeat อีกที เพื่อให้ส่งข้อมูลที่นั่งซึ่งมีการจองในรอบฉายนั้นมาให้ (get\_show\_seat) จากนั้นจึงนำไป update ใน dictionary ของที่นั่ง
  5. เมื่อได้ข้อมูลครบถ้วนก็ส่งกลับไปเพื่อแสดงผลที่นั่ง และ รอรับการเลือกที่นั่ง
  6. การเลือกที่นั่ง จะเลือกที่นั่งที่จองแล้วไม่ได้ เมื่อเลือกที่นั่งใด จะต้องแสดงการเลือกที่นั่ง และ แสดงข้อมูลราคา



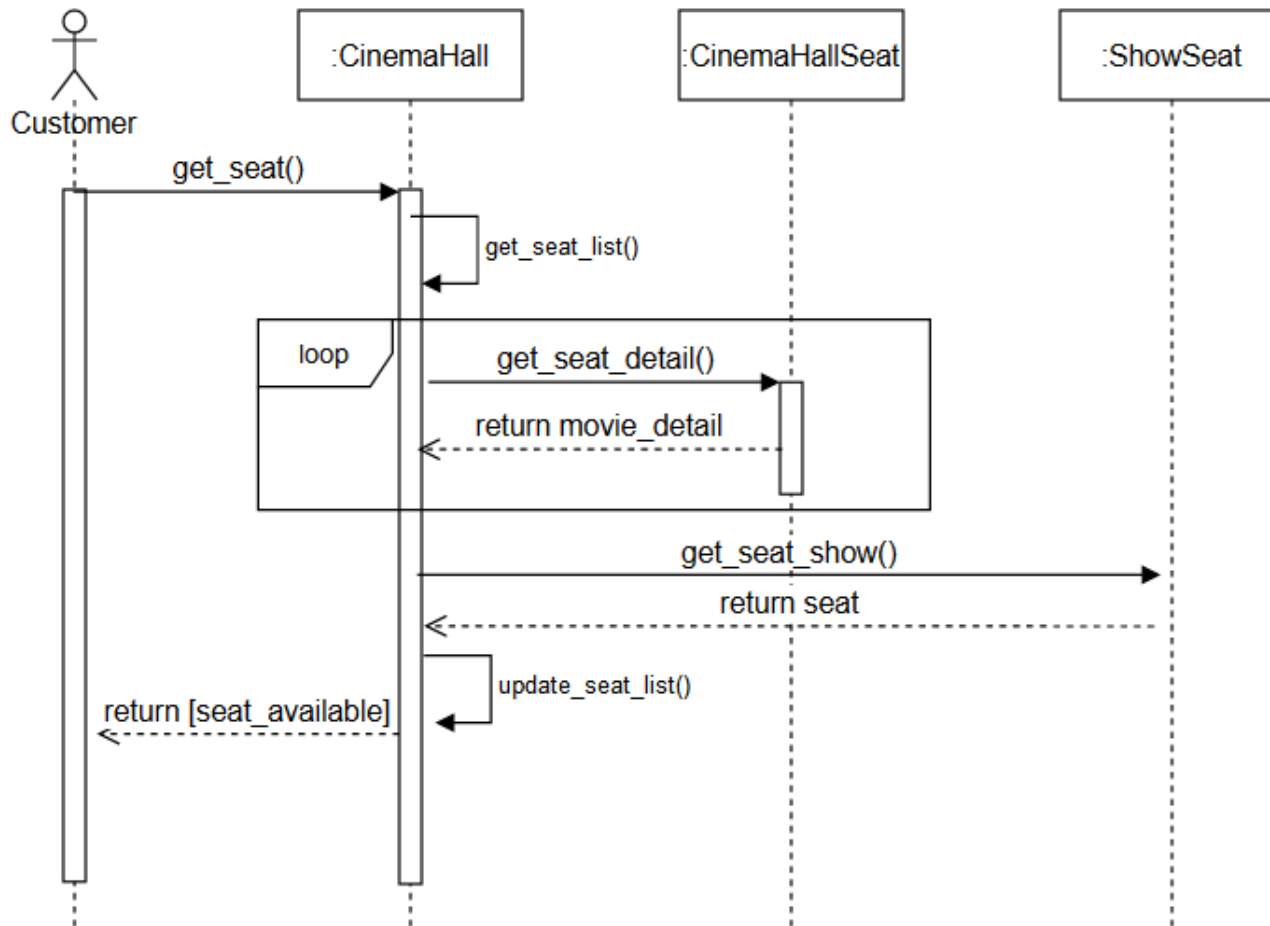
# Sequence Diagram : Movie :: Add Show





# Sequence Diagram : Movie :: Assign Seat

- sequence diagram ของ Assign Seat





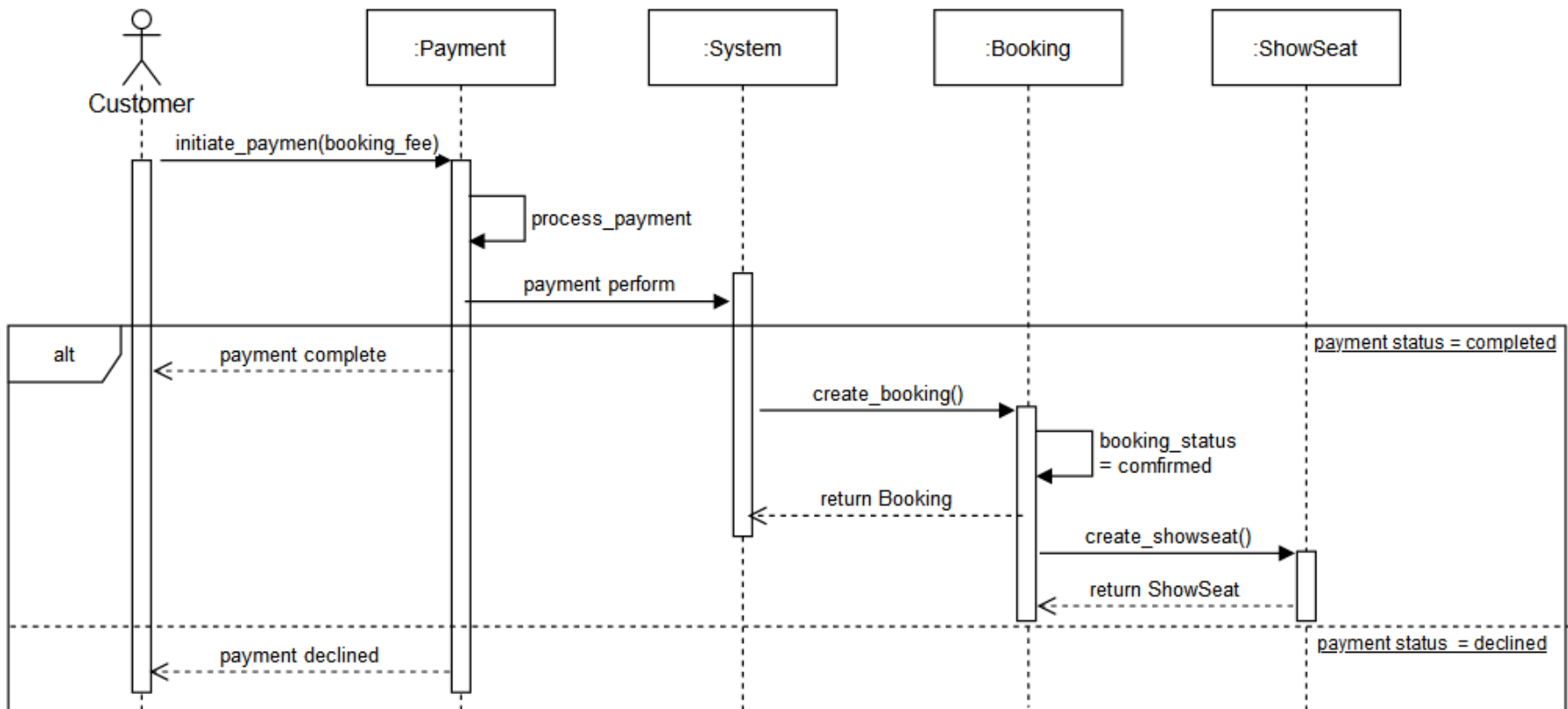
## Sequence Diagram : Movie :: Payment

- ระบุ Actor : Customer, FrontDeskOfficer
- ระบุ Object : Show, CinemaHall, CinemaHallSeat, ShowSeat
- ขั้นตอน
  1. **Precondition** : เมื่อผู้ใช้เลือกที่นั่งที่ต้องการดู
  2. ถ้าผู้ใช้อย่างไม่ได้ login จะเข้าสู่หน้า login ก่อน (diagram นี้จะถือว่า login แล้ว)
  3. ระบบจะแสดง ชื่อภาพยนตร์ วันที่ รอบฉาย โรงฉาย โรงที่ ที่นั่งและราคา
  4. เมื่อผู้ใช้ดำเนินการชำระเงินผ่านระบบที่ผู้ใช้เลือก
  5. ถ้าการชำระเงินสำเร็จ จะสร้าง object Booking และ ShowSeat และแสดงการจองตั๋วเรียบร้อยแล้ว
  6. ถ้าการชำระเงินไม่สำเร็จ แจ้งปัญหาแก่ผู้ใช้



# Sequence Diagram : Movie :: Payment

- sequence diagram ของ payment

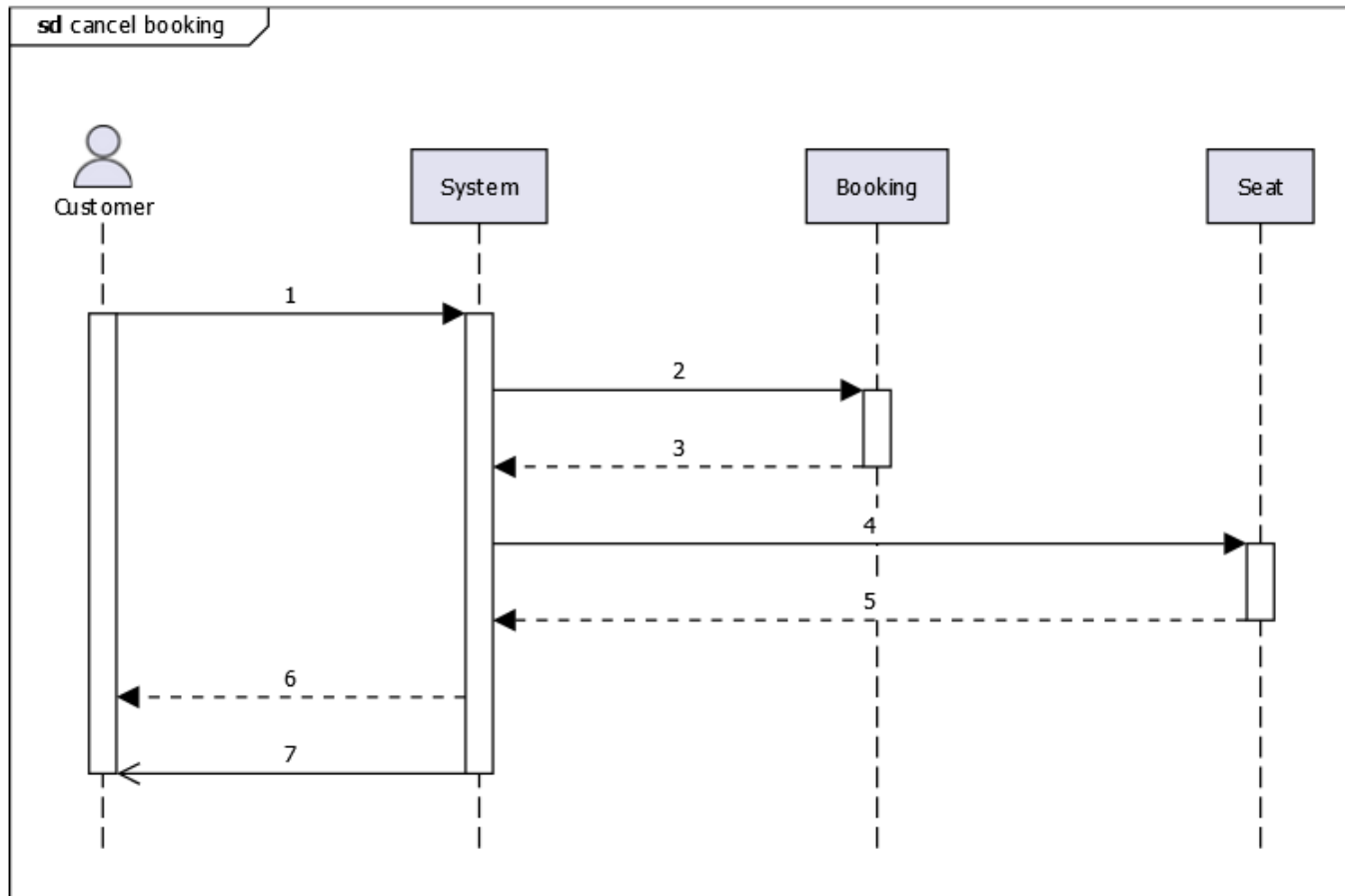






# Sequence Diagram : Movie

- **Exercise** จาก sequence diagram ต่อไปนี้ ให้บอกว่า แต่ละหมายเลขเป็นงานอะไร





## Sequence Diagram : Movie

1. `cancel_booking()`
2. `seat_status` updated
3. `update_seat_status(available)`
4. `sendNotification(cancellationDetails)`
5. `booking_status` updated
6. `Update_booking_status(canceled)`
7. `refund payment`



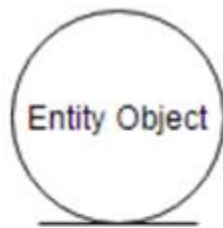
# Robustness Diagram

- Class Diagram เป็น diagram ที่แสดงระบบในมุมมอง โครงสร้างการทำงานร่วมกัน ระหว่างคลาส (Data)
- Use Case Diagram เป็น diagram ที่แสดงระบบในมุมมอง กิจกรรมระหว่าง Actor กับแต่ละส่วนของระบบ (Behavior)
- จะมีอีก diagram หนึ่งที่อยู่ระหว่าง diagram ทั้ง 2 เบื้องต้น โดยทำหน้าที่ในการเพิ่ม มุมมองเชิงพฤติกรรม (Behavior) ให้กับคลาส diagram นี้มีชื่อว่า Robustness Diagram โดยจะแบ่ง Object ออกเป็น 3 ประเภทย่อย
  - Entity Object
  - Controller Object
  - Boundary Object



# Robustness Diagram

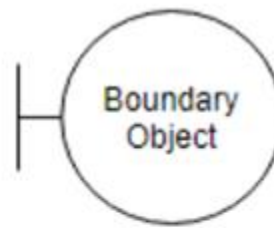
- **Boundary Object** คือ object ที่ทำหน้าที่ Interface กับ actor ซึ่งถือว่าเป็น platform แบบไหน เช่น เว็บ หรือ UI ของมือถือ
- **Entity object** คือ object ที่ทำหน้าที่เก็บข้อมูลในระบบ
- **Control objects** คือ object ที่ทำหน้าที่ประสานระหว่าง boundary กับ entity



Entity Object

Object  
representing  
stored data

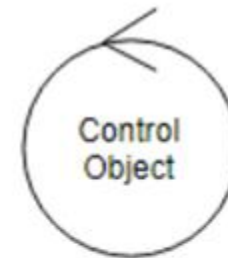
*Model*



Interface / Boundary Object

Object at the  
system  
Interface

*View*



Control Object

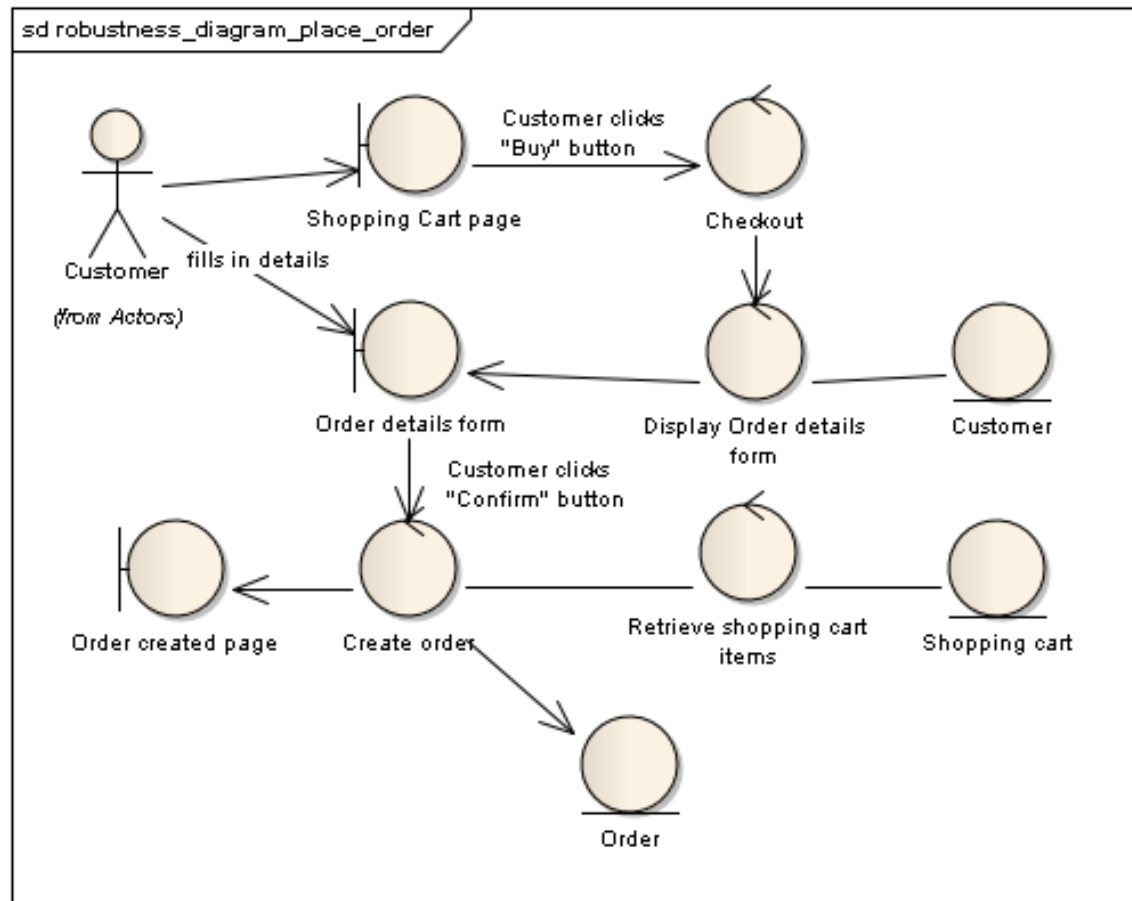
Object representing  
transfer of  
information

*Controller*



# Robustness Diagram

- ตัวอย่าง

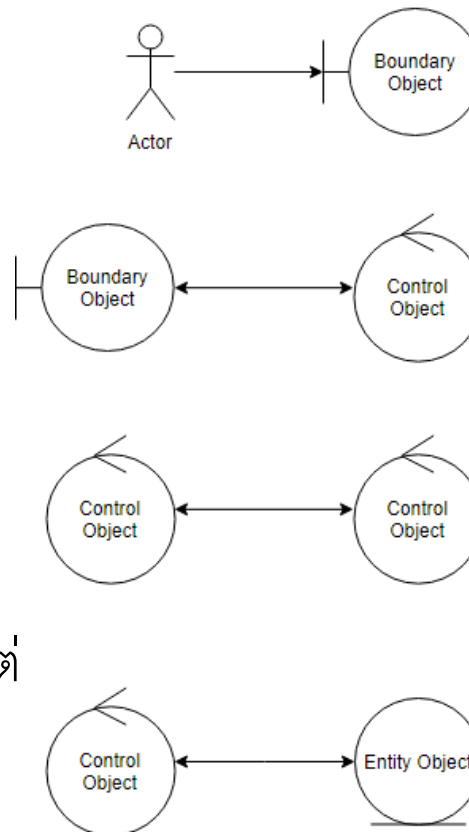




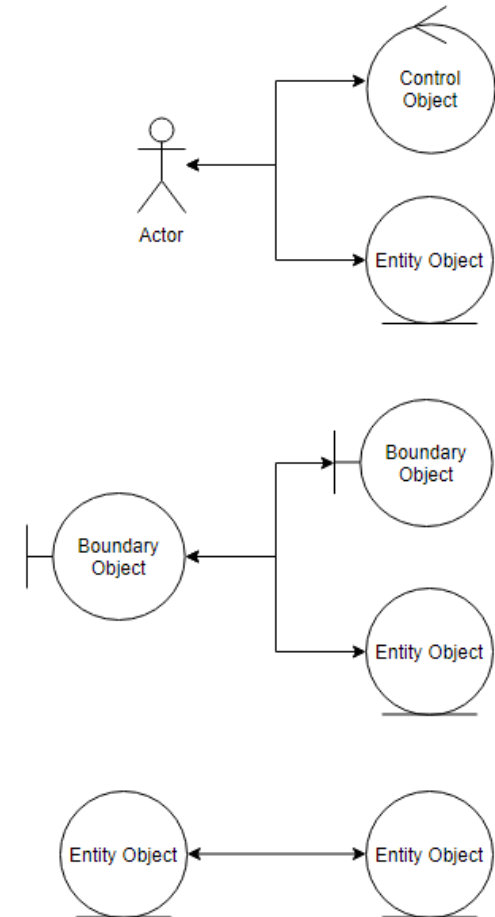
# Robustness Diagram

- Actor จะติดต่อกับ boundary object เท่านั้น
- Boundary object จะติดต่อกับ controller และ actor.
- Entity object จะติดต่อกับ controller.
- Controllers จะติดต่อกับ boundary object และ entity object และ controllers อื่น แต่จะไม่ติดต่อกับ actor

## Allowed



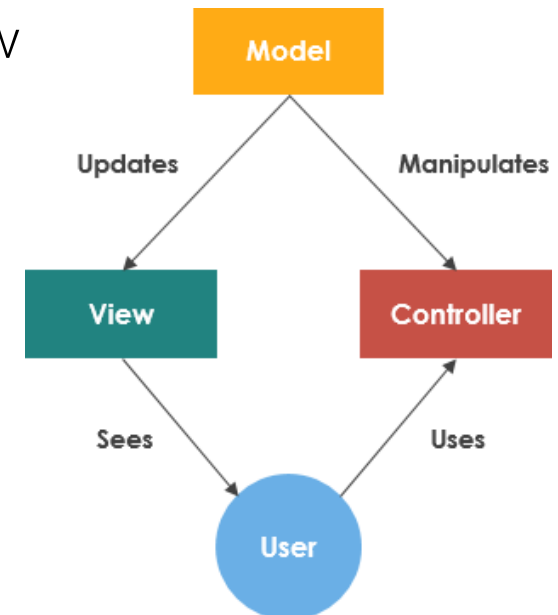
## Not Allowed





## MVC (or Model-view-controller)

- MVC (or Model-view-controller) เป็นรูปแบบ software framework ที่นิยมในปัจจุบัน
- MVC Framework จะมองแอปพลิเคชันออกเป็น 3 ระดับ presentation (UI), application logic และ resource management โดย presentation layer จะแบ่งออกอีกเป็น 2 ส่วน คือ controller and view
- ดังนั้น model จะประกอบด้วย
  - The model (core functionality and data)
  - Views display information to the user.
  - Controllers handle user input.





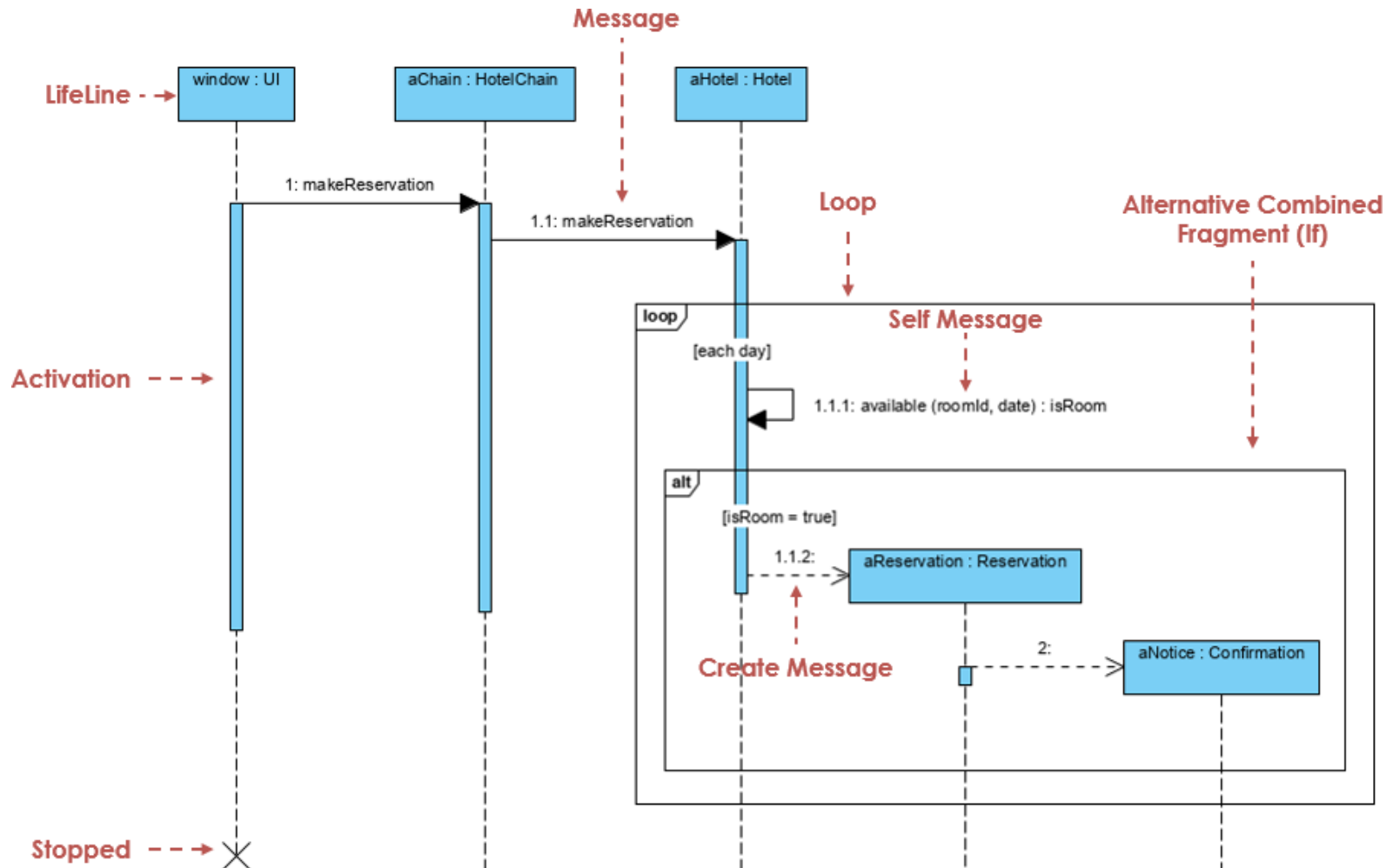
## MVC (or Model-view-controller)

- Model จะแทนส่วนข้อมูล และ logical structure ของข้อมูล โดย object model จะไม่สนใจข้อมูลเกี่ยวกับ user interface
- View เป็นคลาสที่แทนแต่ละส่วนของการแสดงผล (all of the things the user can see and respond to on the screen, such as buttons, display boxes, and so forth)
- Controller แทนคลาสที่ทำหน้าที่ประสานระหว่าง model และ view อาจมีการตรวจสอบข้อมูลเบื้องต้นจาก view และประสานงานระหว่าง model เพื่อจัดข้อมูลการแสดงผลเพื่อให้ view แสดงผล ใน controller จะไม่มีการเก็บข้อมูลไว้ในตัวเอง





# MVC Sequence Diagram





# dataclass

- ใน Object ที่เน้นการเก็บข้อมูลการเขียน Class แบบเดิมๆ จะค่อนข้างรุงรัง ดังนั้นตั้งแต่ Python 3.7 จึงได้สร้าง Library ขึ้นมาชื่อ dataclasses
- จากตัวอย่าง class แบบเดิม

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



# dataclass

- เมื่อใช้ dataclasses จะเขียนในรูปแบบนี้
- จะเห็นว่าคลาสดูง่ายขึ้น โดยจะเหมาะกับคลาสที่เก็บข้อมูลเป็นหลัก

```
from dataclasses import dataclass

@dataclass
class Person():
    name: str
    age: int
    height: float
    email: str

person = Person('Joe', 25, 1.85, 'joe@dataquest.io')
print(person.name)
```



# Object in Memory

- ทุกอย่างใน Python เป็น object

```
print(isinstance(5, object))
print(isinstance([1, 5, 2, 6], object))
print(isinstance((1, 5, 2, 6), object))
print(isinstance("Hello, World!", object))
print(isinstance({"a": 5, "b": 6}, object))
print(isinstance(False, object))
print(isinstance(True, object))
```

```
def f(x):
    return x * 2
```

```
print(isinstance(f, object))
```

```
class Movie:
    def __init__(self, title):
        self.title = title
```

```
print(isinstance(Movie, object))
```

```
True
True
True
True
True
True
True
True
```



# Object in Memory

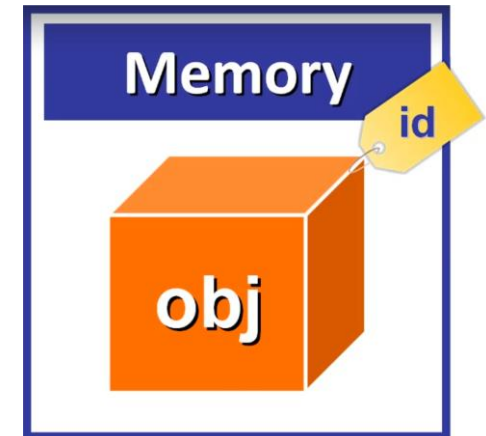
- แต่ละ object จะอยู่ในหน่วยความจำ ซึ่งสามารถระบุ object โดย id
- เราสามารถใช้ฟังก์ชัน id() ในการหา id ของ object ได้ (address)

```
print(id(15))  
print(id("Hello, World!"))  
print(id([1, 2, 3, 4]))
```

```
a = [1, 2, 3, 4, 5]  
b = [1, 2, 3, 4, 5]
```

```
print(id(a))  
print(id(b))
```

```
140000439896448  
139999665958128  
139999666404096  
139999666404096  
139999666404160
```





# Object in Memory

- Object แต่ละ instance จะอยู่ในพื้นที่หน่วยความจำคนละที่กัน

```
class Backpack:

    def __init__(self):
        self._items = []

    @property
    def items(self):
        return self._items

my_backpack = Backpack()
your_backpack = Backpack()

print(id(my_backpack))
print(id(your_backpack))
```

```
139911115521472
139911115113040
```



## “is” operator

- Operator “is” ใช้ตรวจสอบว่า 2 Object มี id เดียวกัน (memory ที่เดียวกัน) หรือไม่

```
a = [1, 6, 2, 6]
b = [1, 6, 2, 6]
print(a is b)
print(a == b)
```

```
a = [5, 2, 1, 8, 3]
b = a
print(a is b)
```

```
c = ("a", "b", "c")
d = ("e", "f")
print(c is d)
```

```
e = "Hello, World!"
f = "Hello, World!"
print(e is f)
```

```
i = 1000
j = 1000
print(i is j)
```

```
False
True
True
False
True
True
True
True
```



# Object Passing

- การผ่านค่า (Pass) ใน Programming จะมี 2 แบบ
  - Pass by value คือ การผ่านค่า โดยการ copy เฉพาะข้อมูลไป ซึ่งผลก็คือ จะไม่ทำให้ข้อมูลเดิมมีการเปลี่ยนแปลง
  - Pass by reference คือ การผ่านค่า โดยการส่งตำแหน่งที่อยู่ของข้อมูลไป (Address) ซึ่งผลก็คือ อาจทำให้ข้อมูลเดิมมีการเปลี่ยนค่าได้
  - ปกติ Programmer ต้องระลึกละระวังในการเขียนโปรแกรม ว่ากำลังใช้การผ่านค่าแบบใด
- การผ่านค่า Object ใน Python จะเป็น Pass by reference ทั้งหมด





# Object Passing

- จะเห็นว่าแม้จะมีการเปลี่ยนแปลงข้อมูล แต่ id คงเดิม (pass by reference)

```
my_list = [6, 2, 8, 2]

def multiply_by_two(seq):
    print("Inside the function:", id(my_list))
    for i in range(len(seq)):
        seq[i] *= 2

print("Outside the function:", id(my_list))
multiply_by_two(my_list)
print(my_list)
```

```
Outside the function: 140169077222144
Inside the function: 140169077222144
[12, 4, 16, 4]
```



# Aliasing in Python

- Alias ถ้าแปลแบบไทยๆ ก็อาจคล้าย ชื่อเล่น คือ เป็นอีกชื่อหนึ่งของสิ่งเดียวกัน
- Alias หมายถึง อะไรก็ตามที่ชี้ไปยัง ข้อมูลในตำแหน่งเดียวกัน (Address) กับอีกตัวแปร

```
a = [1, 2, 3, 4]
```

```
b = a
```

```
c = b
```

```
d = c
```

```
print(id(a))
```

```
print(id(b))
```

```
print(id(c))
```

```
print(id(d))
```

```
print(a is b is c is d)
```

```
140688432651008
```

```
140688432651008
```

```
140688432651008
```

```
140688432651008
```

```
True
```



# Aliasing in Python

- Object ก็สามารถสร้าง Alias ได้เช่นกัน

```
class Circle:
    def __init__(self, radius):
        self.radius = radius
```

```
my_circle = Circle(4)
your_circle = my_circle
print("Before:")
print(my_circle.radius)
print(your_circle.radius)
```

```
your_circle.radius = 18
print("After:")
print(my_circle.radius)
print(your_circle.radius)
```

Before:

4

4

After:

18

18



# Mutability and Immutability

- Mutable แปลว่า สามารถแก้ไขได้ หมายถึง Object ที่สามารถเปลี่ยนแปลงค่าได้ เช่น Lists, Set, Dictionary
- Immutable แปลว่า ไม่สามารถแก้ไขได้ หมายถึง Object ที่ไม่สามารถเปลี่ยนแปลงค่าได้ เช่น Tuple, Strings (หลังจาก assign ค่า)

```
a = [7, 3, 2, 1]
```

```
a[0] = 5
```

```
print(a)
```

```
a = (7, 3, 2, 1)
```

```
a[0] = 5 # throws an error because tuples are immutable.
```

```
a = "Hello, World!"
```

```
a[0] = "S" # throws an error because strings are immutable.
```



# Mutability and Immutability

- ข้อดีของ Object แบบ Mutable
  - ประหยัดหน่วยความจำมากกว่า เพราะสามารถใช้ข้อมูลเดิมได้ ไม่ต้องเพิ่มใหม่หากมีการแก้ไข
  - ตรงกับข้อมูลในโลกจริง ที่เปลี่ยนแปลงได้
- ข้อเสียของ Object แบบ Mutable
  - หากใช้ไม่ระวัง อาจมี Bug
  - โปรแกรมนี้มีปัญหาอย่างไร

```
def add_absolute_values(seq):  
    for i in range(len(seq)):  
        seq[i] = abs(seq[i])  
    return sum(seq)  
  
values = [-5, -6, -7, -8]  
print("Values Before:", values)  
result = add_absolute_values(values)  
print("Values After:", values)
```



# Mutability and Immutability

- โปรแกรม 2 โปรแกรมนี้ต่างกันอย่างไร

```
a = [1, 2, 3, 4]
b = a
b[0] = 15
print(a)
print(b)
```

```
a = [1, 2, 3, 4]
b = a[:]
b[0] = 15
print(a)
print(b)
```



# Mutability and Immutability

- ข้อดีของ Object แบบ Immutable
  - ไม่มี Bug
  - ง่ายต่อการทำความเข้าใจ เพราะไม่ต้องคิดเผื่อกรณีที่มีการเปลี่ยนแปลง
- ข้อเสียของ Object แบบ Immutable
  - ประสิทธิภาพต่ำกว่า เพราะเมื่อมีการเปลี่ยนแปลง ต้องสร้างข้อมูลใหม่

```
a = (1, 2, 3, 4)
print(id(a))
a = a[:2] + (7,) + a[2:]
print(a)
print(id(a))
```

```
140660086048064
(1, 2, 7, 3, 4)
140660084889376
```



# Mutability and Immutability

- Code ต่อไปนี้ จะแสดงผลอะไร ถ้าผิดพลาดจะแก้ไขอย่างไร

```
class WaitingList:

    def __init__(self, clients=[]): # The default argument is an empty list
        self.clients = clients

    def add_client(self, client):
        self.clients.append(client)

waiting_list1 = WaitingList()
waiting_list2 = WaitingList()
waiting_list1.add_client("Jake")

print(waiting_list1.clients)
print(waiting_list2.clients)
```





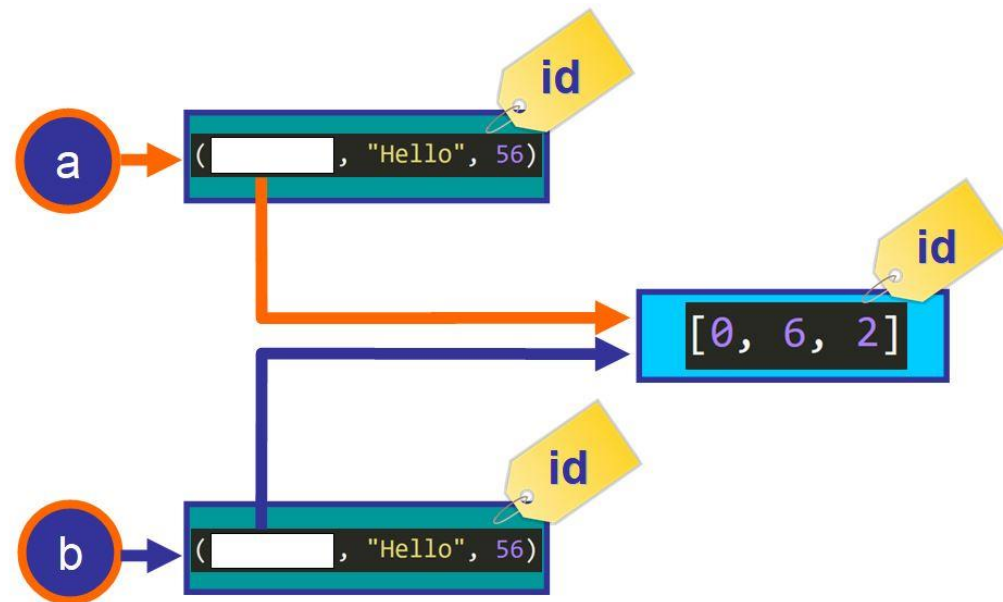
# Mutability and Immutability

- คำว่า Immutable ก็ไม่ได้หมายความว่า เปลี่ยนแปลงไม่ได้
- ดูตามตัวอย่าง

```
a = ([0, 6, 2], "hello", 56)
b = a[:]
a[0][1] = -5

print(a)
print(b)
```

```
([0, -5, 2], 'hello', 56)
([0, -5, 2], 'hello', 56)
```





# Mutability and Immutability

- วิธีแก้ไข คือ ใช้ module copy

```
import copy
```

```
a = ([0, 6, 2], "hello", 56)
```

```
b = copy.deepcopy(a)
```

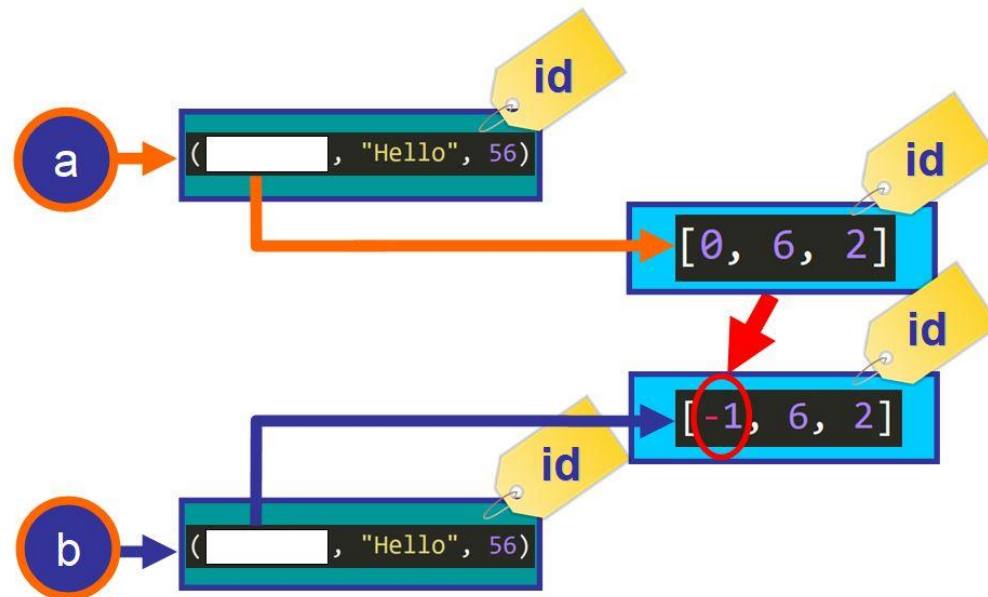
```
a[0][0] = -1
```

```
print(a)
```

```
print(b)
```

```
([-1, 6, 2], 'hello', 56)
```

```
([0, 6, 2], 'hello', 56)
```





# Mutability and Immutability

- โปรแกรมนี้ทำงานได้หรือไม่ ถ้าผิดพลาดจะแก้ไขอย่างไร

```
def remove_even_values(dictionary):  
    for key, value in dictionary.items():  
        if value % 2 == 0:  
            del dictionary[key]  
  
my_dictionary = {"a": 1, "b": 2, "c": 3, "d": 4}  
  
remove_even_values(my_dictionary) # This throws an error.
```



# Mutability and Immutability

- โปรแกรมนี้ทำงานได้หรือไม่ ถ้าผิดพลาดจะแก้ไขอย่างไร

```
def remove_even_values(dictionary):  
    for key, value in dictionary.copy().items():  
        if value % 2 == 0:  
            del dictionary[key]  
  
my_dictionary = {"a": 1, "b": 2, "c": 3, "d": 4}  
  
remove_even_values(my_dictionary)  
  
print(my_dictionary)
```



*For your attention*