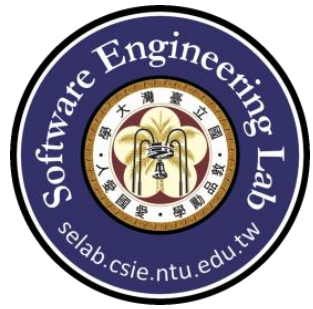


# Object-Oriented Concepts

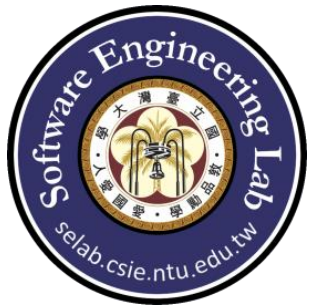
Prof. Jonathan Lee  
CSIE Department  
National Taiwan University



# Outline

---

1. Overview
2. Basic Concepts (I)
3. Basic Concepts (II)
4. Object-Oriented Concepts (III-0): Relations Overview
5. Object-Oriented Concepts (III-1): Dependency, Association, and Constraint
6. Object-Oriented Concepts (III-2): Aggregation and Composition
7. Sequence Diagrams
8. Practice and Homework



# Why Objects?

- ❑ Communication with customers through a direct mapping from real world objects to software objects

Problem  
(WHAT)



Solution  
(HOW)

Objects necessary  
for describing a  
problem  
space

Objects required  
for implementing  
a solution  
space



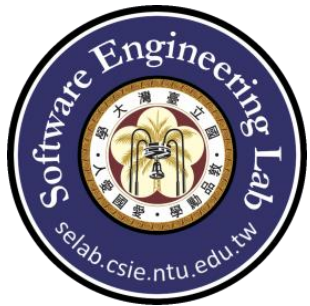
# Why Modeling First, Coding Later?

- ❑ Analyze the problems in a higher level point of view similar to natural languages. (High level perspective)
- ❑ Define the structure and behavior of a target software system in a visualized manner. (Visualization)
- ❑ Communicate with users without using implementation languages. (Communication)
- ❑ Defer the vicious cycle of debugging-fixing to a later stage until we figure out a better design model. (Changes)
- ❑ Link objects in a model to their executable code in a more systematic manner. (Systematic)



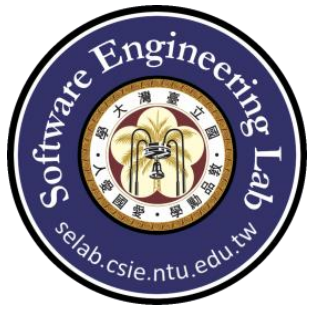
# Features of Object Orientation

- ☐ Identity
- ☐ Classification
- ☐ Inheritance
- ☐ Polymorphism
- ☐ Dispatch
- ☐ Encapsulation
- ☐ Abstraction
- ☐ Delegation
- ☐ Dependency
- ☐ Association
- ☐ Aggregation
- ☐ Composition
- ☐ Generalization



# Class Diagrams

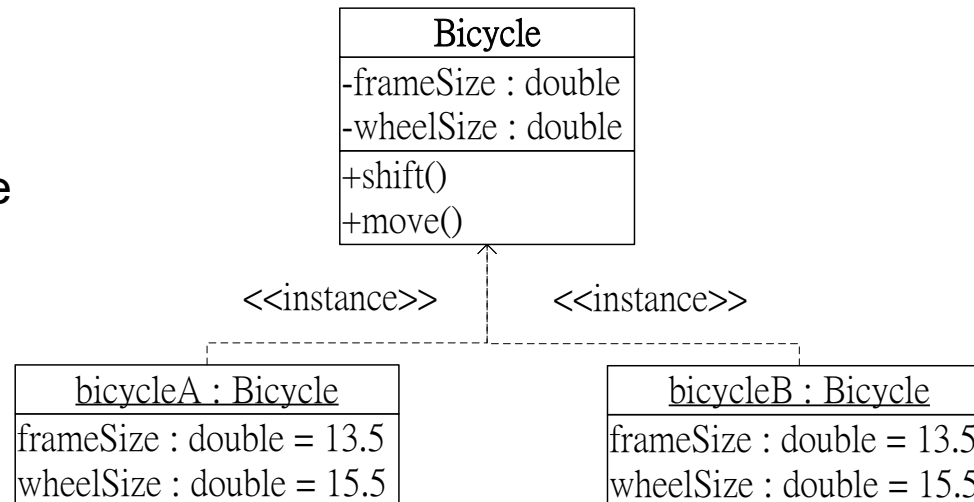
- ❑ Class is a kind of classifier.
- ❑ A Classifier represents a group of things with common properties.
- ❑ Provide a way to **capture how things are put together, and make design decisions:**
  - What classes hold reference to other classes?
  - What the interactions are among classes?
  - Which class owns some other class
  - ....



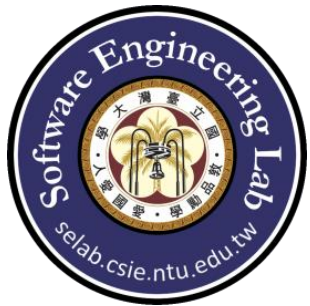
# Identity

- ❑ Each object is a discrete and distinguishable entity.
- ❑ Each real-world object is unique due to its existence.
- ❑ Each object has its own inherent identity, therefore, two objects are distinct even if all their attribute values are identical.

e.g. bicycleA and bicycleB are distinct even their attribute values are identical



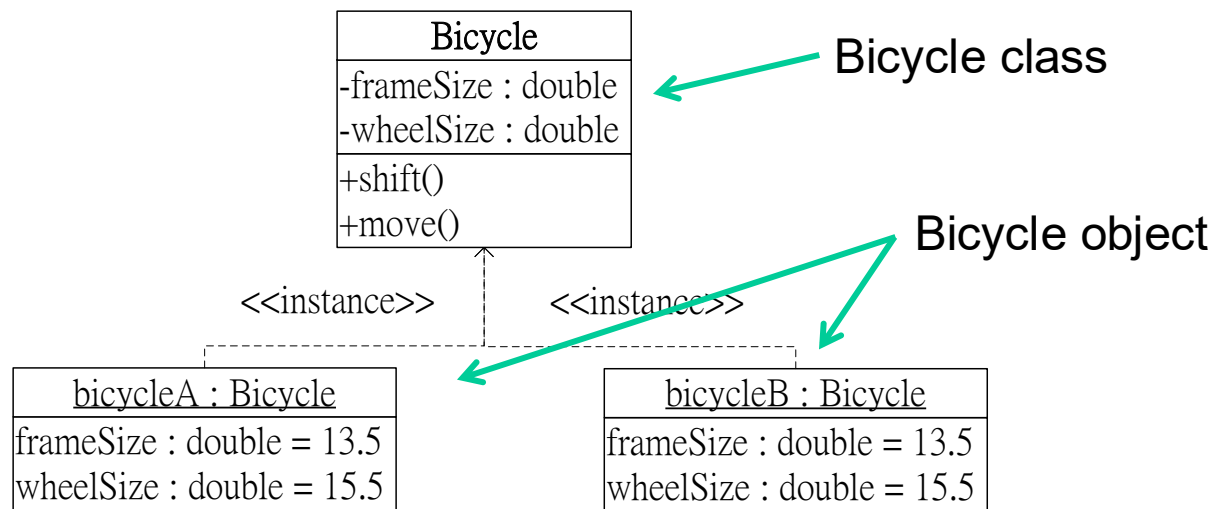




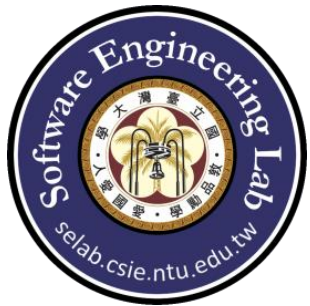
# Classification

## □ Classification: (Class & Object)

- Objects with the same attributes and operations are grouped into a class (data abstraction)
- Each object is said to be an instance of its class
- e.g. Bicycle object -----> Bicycle class

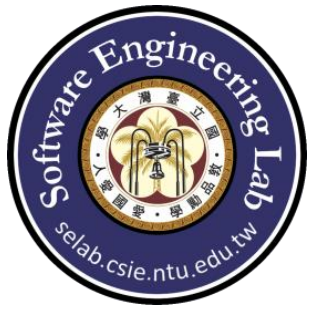






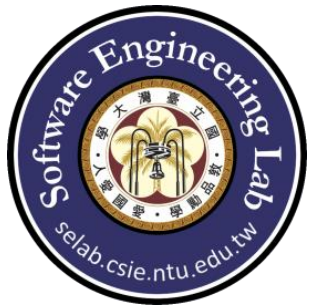
# Class

- ❑ A class is a definition of the behavior of an object, and contains a complete description of the following:
  - The data elements (variables) the object contains
  - The operations the object can do
  - The way these variables and operations can be accessed
- ❑ Objects are instances of classes
- ❑ Creating instances of a class is called instantiation.



# Class Notation

Class Name
Attribute
Operation



# Attribute

□ Visibility / name:      type      multiplicity = default

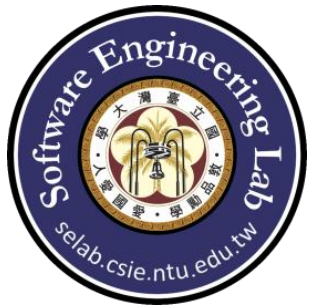
{+ | - | #}      { <sup>class</sup>  
interface  
int }      {1, \*}      default value

□ Examples:

- +wheels: wheel[4]
- -bumper stickers: sticker[ \* ]
- -passengers: person[1..5]

□ Static attributes are attributes of the class rather than of an instance of the class.

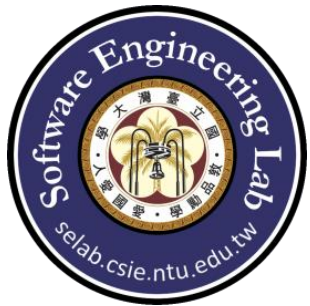
- e.g. initialize **constant values** for a class and share them between all instances of the class.



# Static Attribute Example

Color	
+BLACK:	int = 0xFF000000
+DKGRAY:	int = 0xFF444444
+GRAY:	int = 0xFF888888

```
public class Color {  
    public static final int BLACK = 0xFF000000;  
    public static final int DKGRAY = 0xFF444444;  
    public static final int GRAY = 0xFF888888;  
}
```



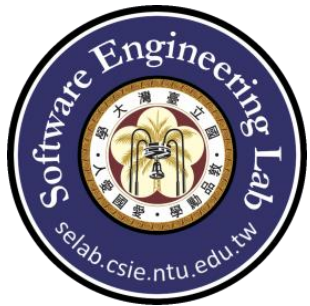
# Operation

- ❑ An operation specifies how to invoke a particular behavior. A method is an **implementation** of an operation.
- ❑ Visibility name (parameters): return-type {property}

parameter\_name: type[multiplicity] = default\_value

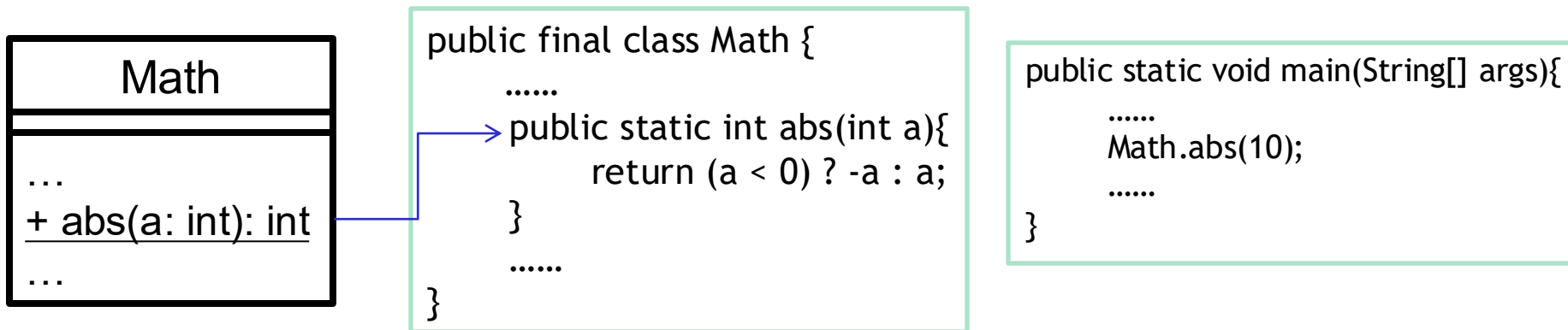
- ❑ Examples:

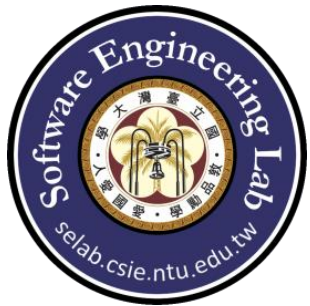
- +getSize(): Rectangle
- +setSize(name: String): void
- +getComponents(): Component [0... \* ]



# Static Operation

- ❑ Static operations specify behavior for the class itself.
  - e.g. It is common for classes to contain convenient static methods to perform common tasks.



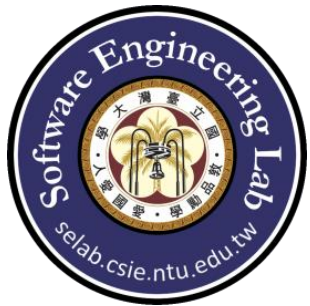


# Abstract Class

- ❑ An abstract classes provides an operation signature but no implementation.
  - e.g.

<i>Movable</i>
<i>+move(): void</i>

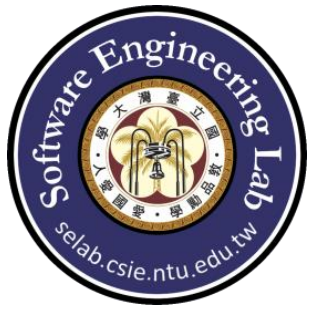




# Interface

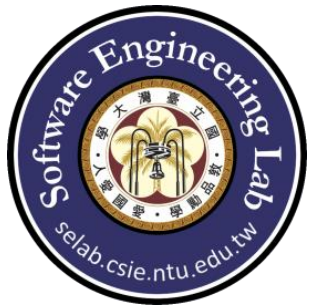
- ❑ An interface is a classifier that has declarations of properties and methods but no implementations.
- ❑ Stereotype: A model element that identifies the purpose of other model element, such as <<interface>>, <<use>>, <<create>>.
  - e.g.

<p>&lt;&lt;interface&gt;&gt; <i>Sortable</i></p>
<p><i>+comesBefore(object: Sortable): boolean</i></p>

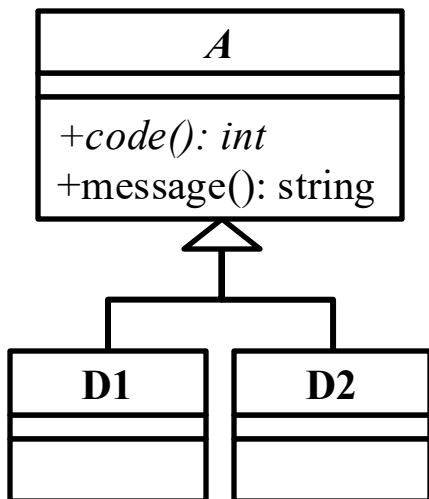


# Inheritance

- ☐ The sharing of attributes and operations among classes based on a hierarchical relationship
- ☐ Each subclass inherits all of the properties of its superclass and adds its own unique properties (called extension through override)
- ☐ Facilitate reusability

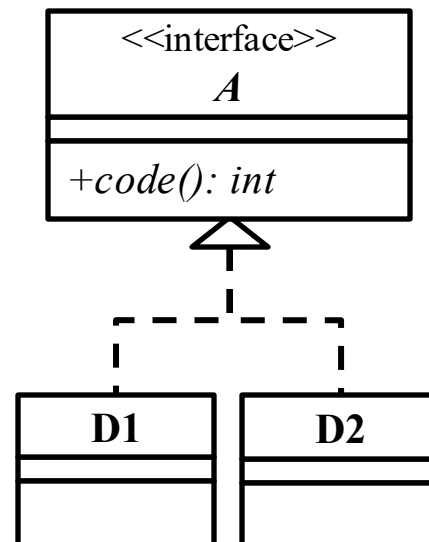


# Inheritance Examples



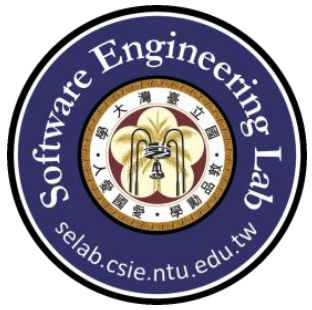
In abstract class, code in message() can be reused in class D1 & D2; While abstract operation code() needs to be implemented in D1 & D2.

```
public class D1 extends A {  
    public int code() { return 1; }  
}  
  
public class D2 extends A {  
    public int code() { return 2; }  
}
```



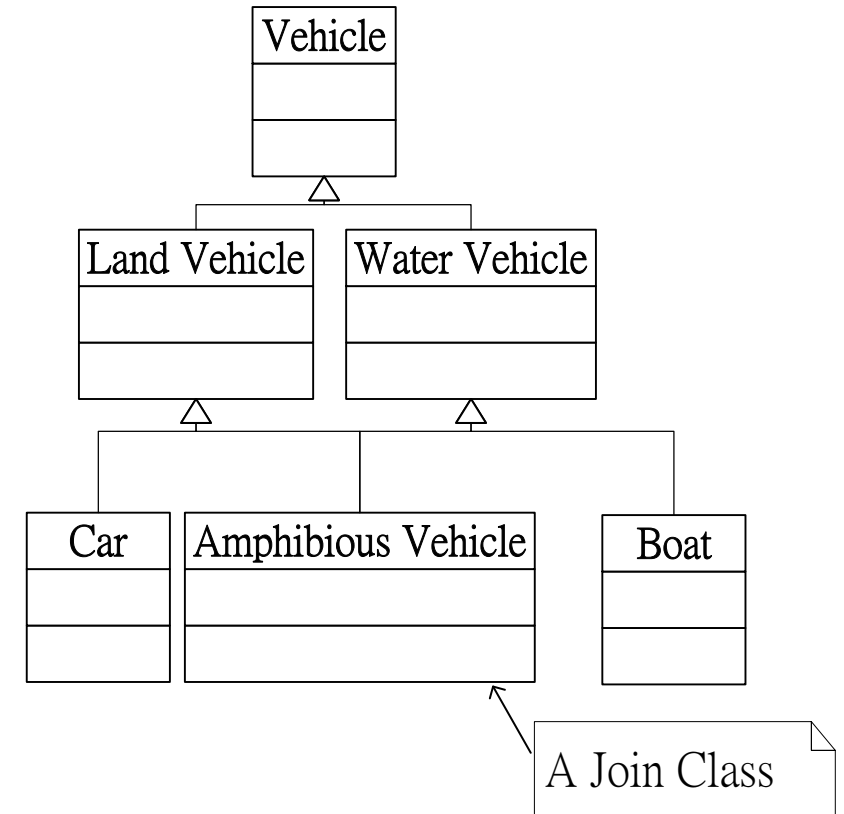
Interface, however, does not have the benefits of code reuse; D1 and D2 have to implement the abstract operation code().

```
public class D1 implements A {  
    public int code() { return 10; }  
}  
  
public class D2 implements A {  
    public int code() { return 20; }  
}
```

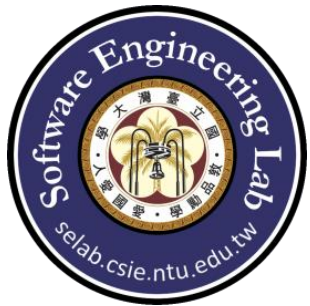


# Multiple Inheritance<sub>1</sub>

- ❑ a class has more than one superclass and inherits features from all parents (called a join class).
  - Generalization: conceptual level
  - Inheritance: implementation level. (mechanism)



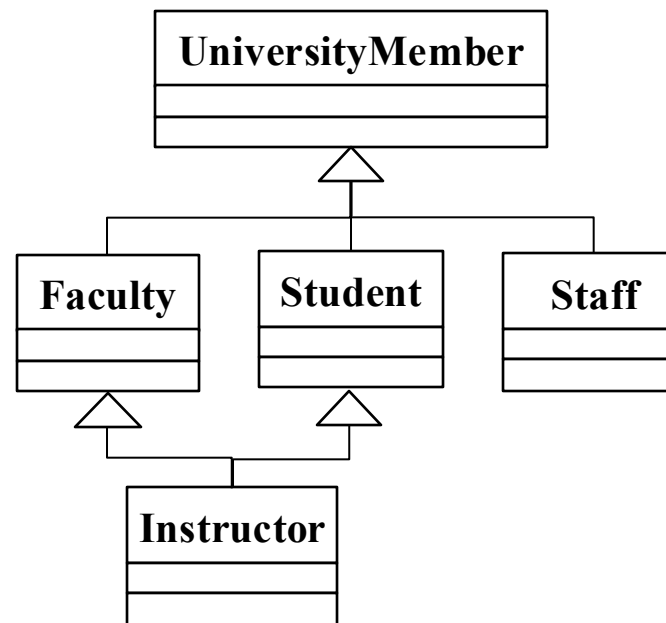
In Java, multiple inheritance is not allowed.

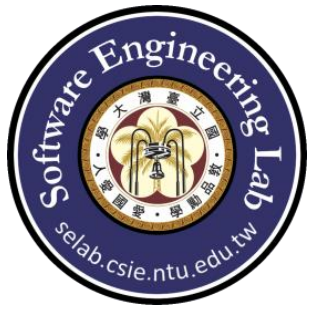


# Multiple Inheritance<sub>2</sub>

## □ A problem:

- Accidental multiple inheritance.
- That is, one instance happens to participate in two *overlapping* classes.

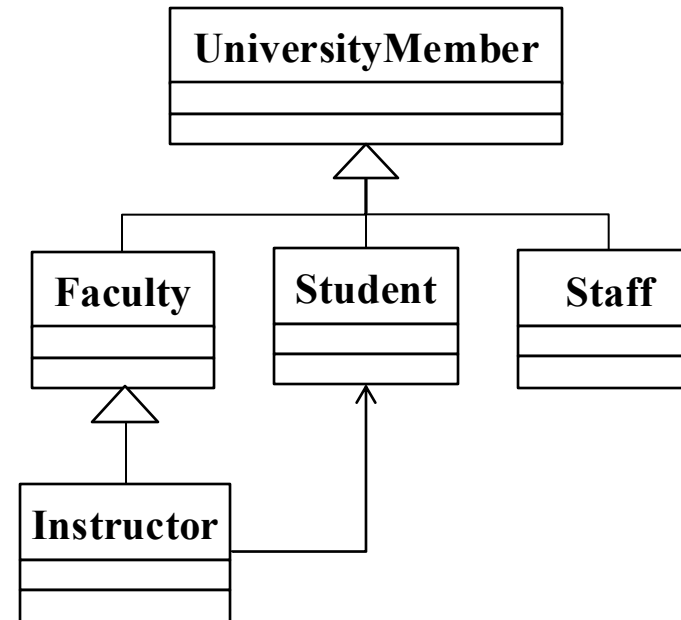




# Multiple Inheritance<sub>3</sub>

## □ Resolution:

- Delegation
- Inherit the most important features and delegate the rest.

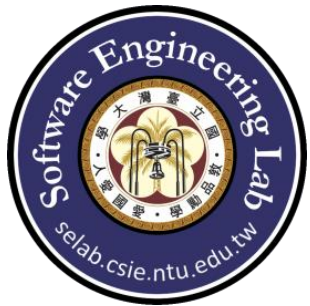




# Polymorphism

- ❑ A same operation may behave differently on different classes.
- ❑ An operation is an action or transformation that an object performs.
- ❑ Method: a specific implementation of an operation
  - a polymorphic operation is an operation that has more than one method implementing it.





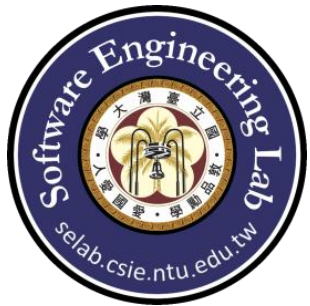
# Polymorphism (having many forms)

## ❑ Static polymorphism

- **Overloading:** an invocation can be operated on arguments of more than one type.

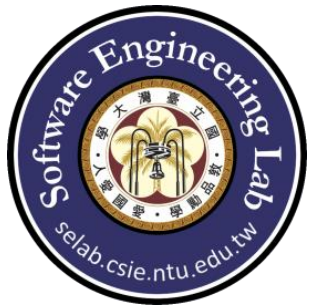
TimeOfDay
+setTime(time: char[]): void
+setTime(h: int, m: int, s: int): void

```
class TimeOfDay {  
    public void setTime(char[] time) {...};  
    public void setTime(int h, int m, int s) {...};  
}  
TimeOfDay aClock= new TimeOfDay( );  
aClock.setTime("11:55:00");  
aClock.setTime(11,55,0);
```

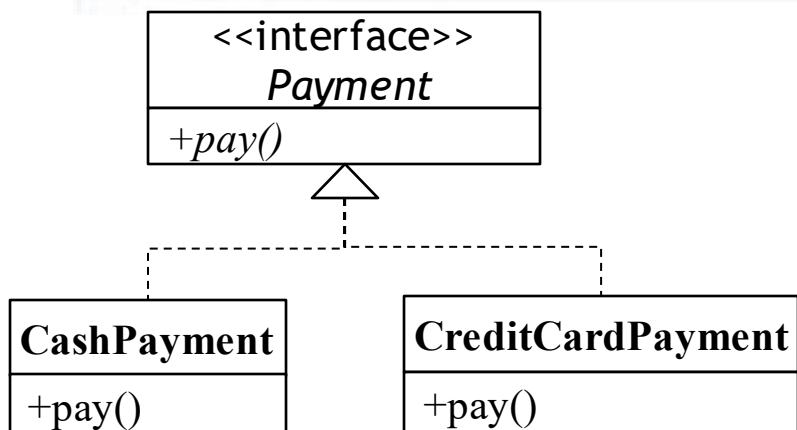


# Polymorphism

- ❑ Dynamic polymorphism: A same operation may behave differently on different classes.
  - method: a specific implementation of an operation.
  - a polymorphic operation is an operation that have more than one method implementing it.
  - Single-dispatch polymorphism is where a function or method call is dynamically dispatched based on the actual derived type of the object on which the method has been called.



# Polymorphism Example



□ `pay()` is a polymorphic operation and is implemented in **CashPayment** class and **CreditCardPayment** class.

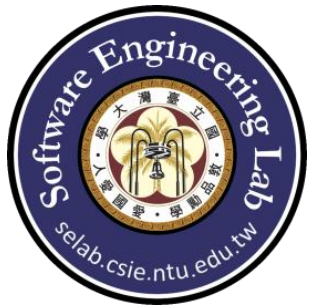
```
public interface Payment {
    public void pay();
}
```

```
public class CashPayment implements Payment {
    public void pay() {
        System.out.println("Pay in cash");
    }
}

public class CreditCardPayment implements Payment {
    public void pay() {
        System.out.println("Pay with credit card");
    }
}
```

```
Payment p = new CashPayment();
p.pay();
```

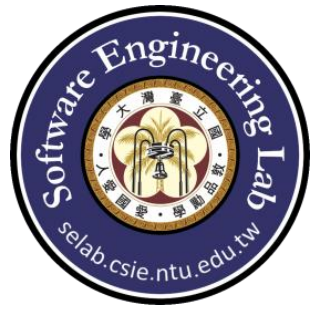
The above code outputs: Pay in cash



# Dispatch

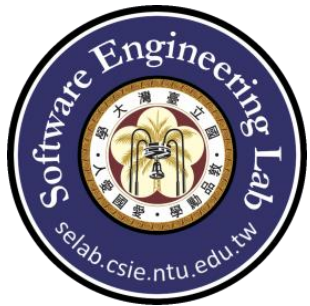
---

- ☐ The process of selecting which method to invoke.
- ☐ Static dispatch: the implementation of an operation is assigned during **compile time**.
- ☐ Dynamic dispatch: the implementation of an operation is assigned during **runtime**

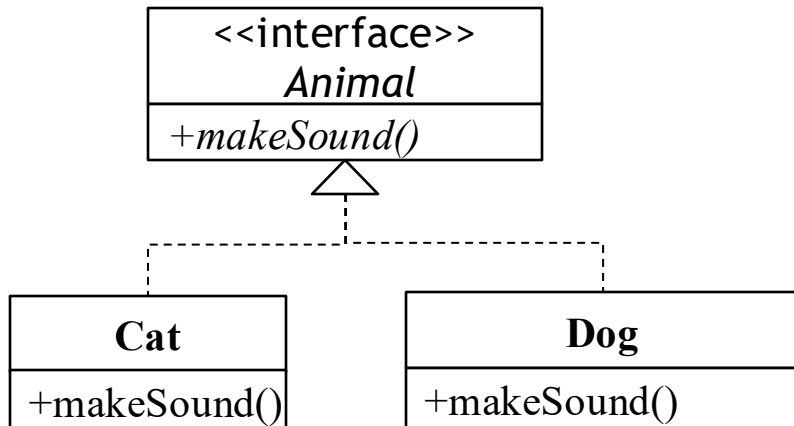


# Single Dispatch

- ❑ Single dispatch is where a method call is dynamically dispatched based on the actual derived **type** of the object on which the method has been called (namely, the concrete type of the object).



# Example of Single Dispatch



- ❑ `makeSound()` is a polymorphic operation and is implemented in **Cat** class and **Dog** class.
- ❑ The implementation of `makeSound()` is chosen based on the type of the object (at runtime).

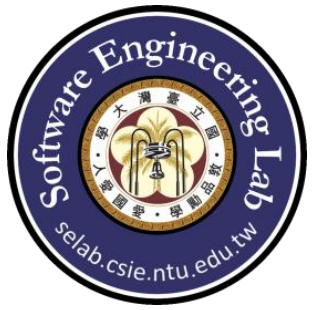
```
public interface Animal {
    public void makeSound();
}
```

```
public class Cat implements Animal {
    public void makeSound() {
        System.out.println("Meow");
    }
}
```

```
public class Dog implements Animal {
    public void makeSound() {
        System.out.println("Woof");
    }
}
```

```
Animal a = new Cat();
a.makeSound();
```

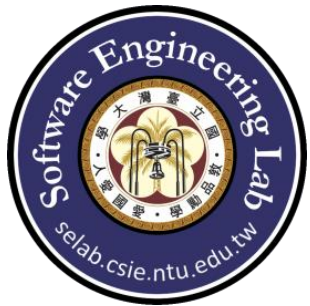
The above code outputs: Meow



# Double Dispatch










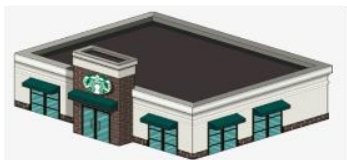
- ❑ Double dispatch is a mechanism that dispatches a function call to different concrete functions depending on the **runtime types of two objects** involved in the call.

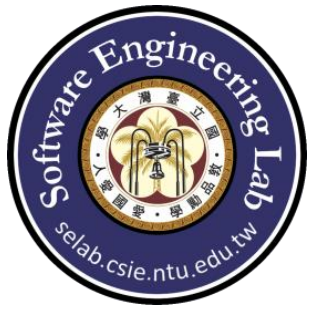




# Example 1 of Double Dispatch

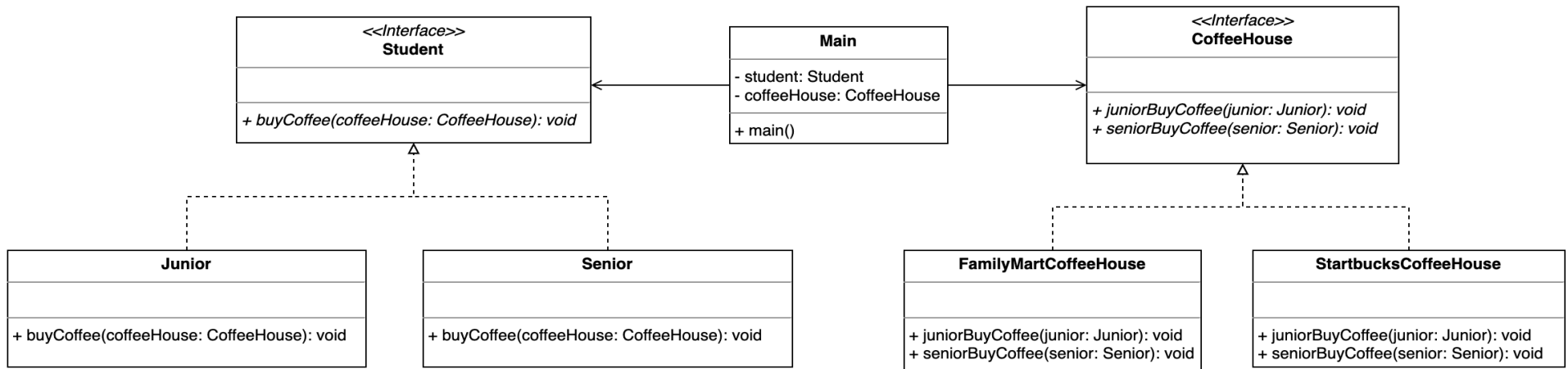
- Suppose Professor Lee wants to drink a cup of coffee. There are two groups of students to call: junior and senior, to buy coffee from Starbucks and FamilyMart.

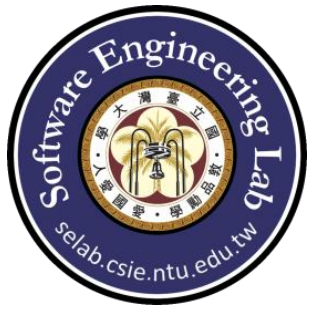
Behavior Matrix	FamilyMartCoffeeHouse	StarbucksCoffeeHouse
Junior 	 	 
Senior 	 	 



# Example 1 Class Diagram




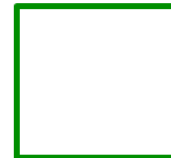
- ❑ In the following example, the double dispatch mechanism is triggered by the call of the Student's buyCoffee method, and inject the coffeehouse, that is buyCoffee(coffeehouse: CoffeeHouse).
- ❑ The buycoffee mechanism triggered by the Main is dependent on two objects - Student and a CoffeeHouse.





## Example 2 of Double Dispatch

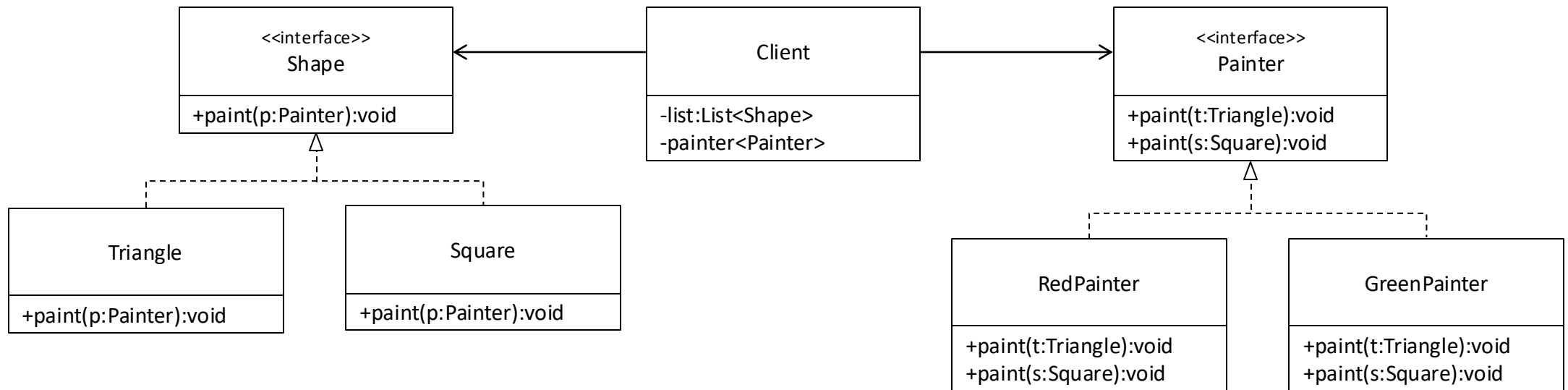
- ❑ The paint mechanism triggered by the client is dependent on two objects- Shape and a Painter. Like this the Client triggers the paint mechanism without knowing what kind of shape in what color is painted.
- ❑ If for example its painter is a GreenPainter and the Shape is a Triangle then a green triangle appears on the screen.

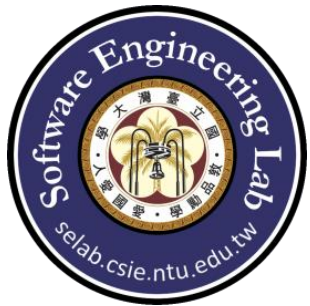
Behavior matrix	RedPainter	GreenPainter
Triangle		
Square		



## Example 2 Class Diagram

- ❑ The double dispatch mechanism is triggered by the call of the Shapes' paint method. The client calls the paint method and injects its Painter. Then the Shape passes itself as this in one of the overloaded methods of the Painter.
- ❑ The paint mechanism triggered by the client is dependent on two objects - Shape and a Painter.





# Encapsulation

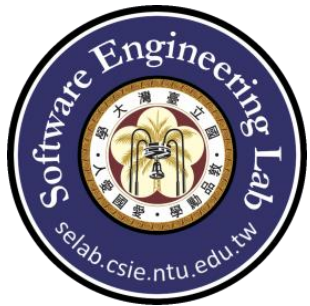
- ❑ Object orientation separates the external aspects of an object accessible to their objects from the internal implementation details of the object hidden from other objects.
- ❑ The selection of properties to be encapsulated.
  - attributes, operations, representations, algorithms...
- ❑ The determination of **visibility** of these properties, that is, a well-defined interface.



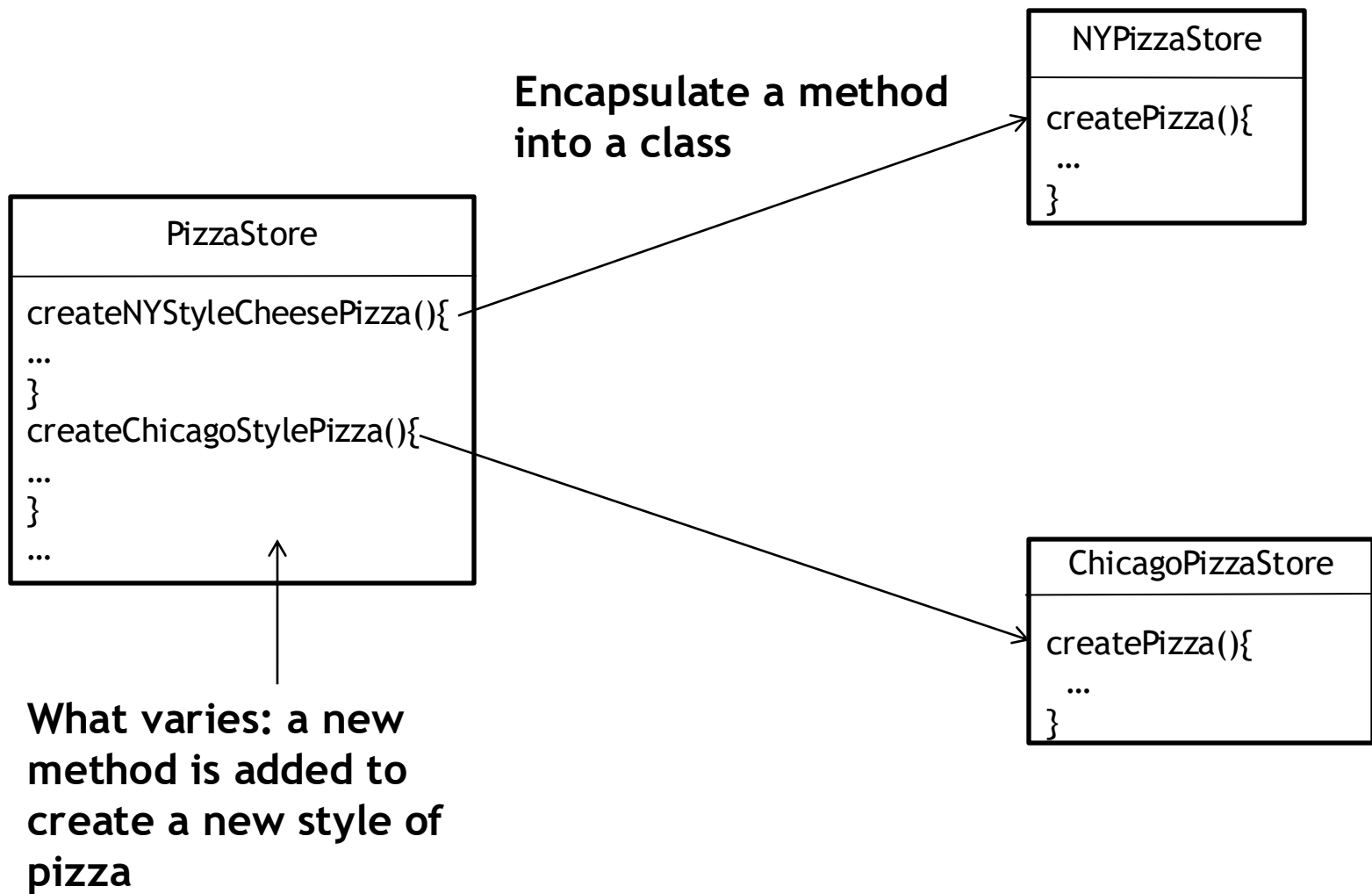
Only shift() & move() are *visible* to users. Further, the implementation of the two methods are *encapsulated*

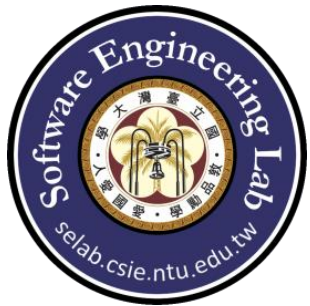


Bicycle
-frameSize : double
-wheelSize : double
+shift()
+move()



# Encapsulate What *Varies* (*Design Principle*)

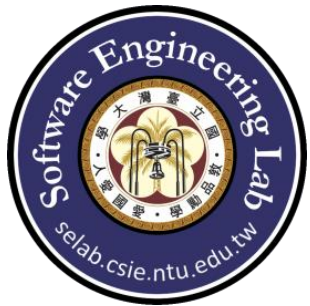




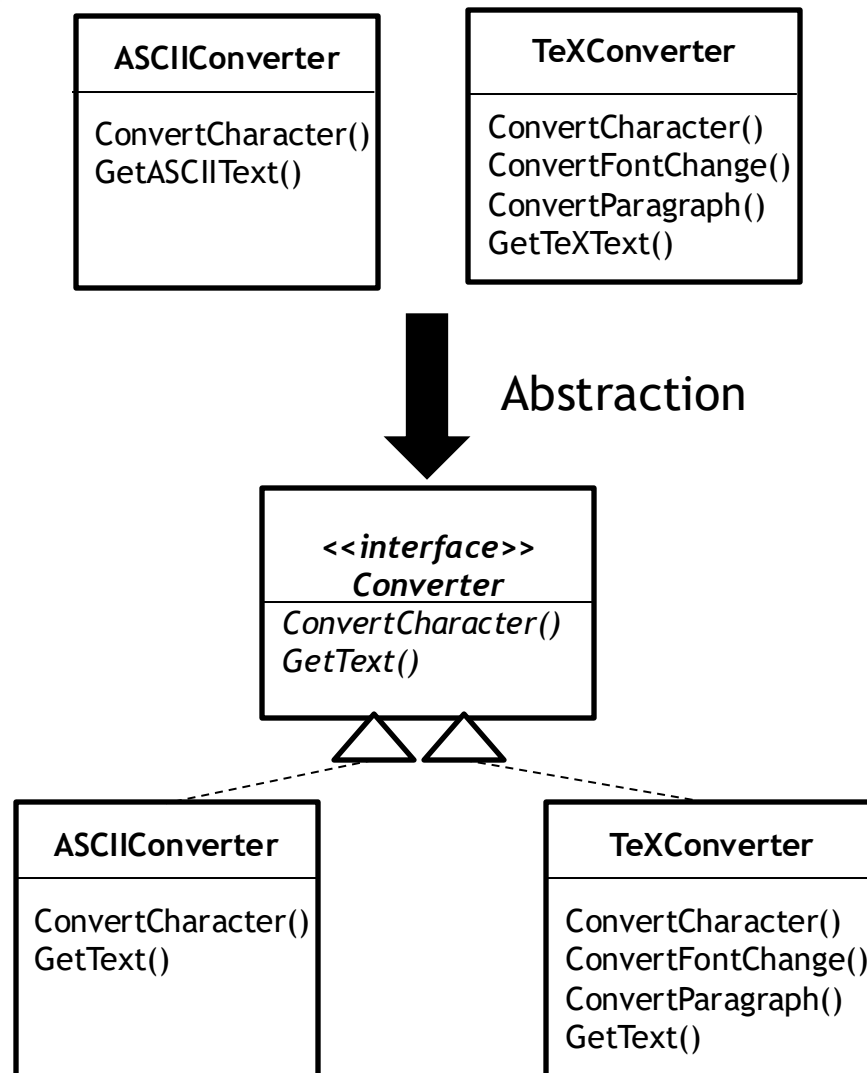
# Abstraction

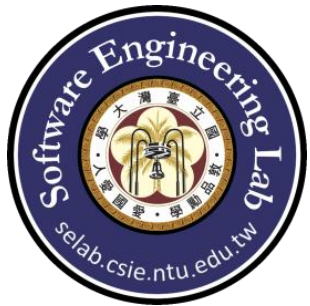
- ❑ Focus on the essential aspects of an entity and ignore others.
  - Use of abstraction during analysis: deciding only with application-domain concepts, not making detailed design and implementation decisions.
- ❑ In Bicycle class, only frame size, wheel size, shift and move are considered. Others, like color, weight, and etc. are ignored.





# Abstract Common Features





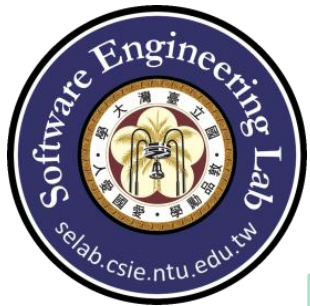
# Delegation

- ❑ Delegation: An object **forwards an operation** to another object that is part of or related to the first object for execution.
  - This mechanism is sometimes referred to as aggregation, consultation or forwarding.



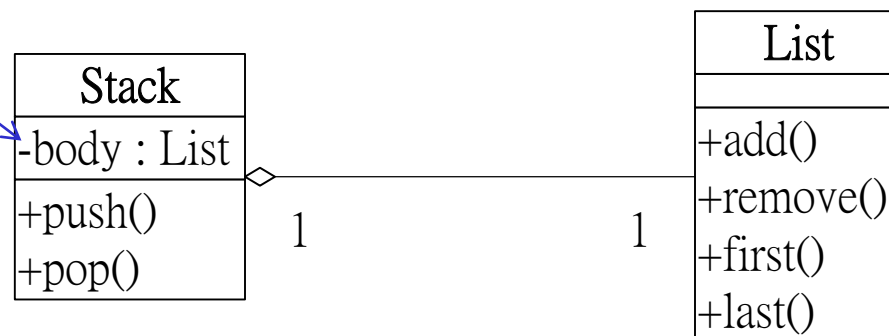
Delegate **add** for **push**

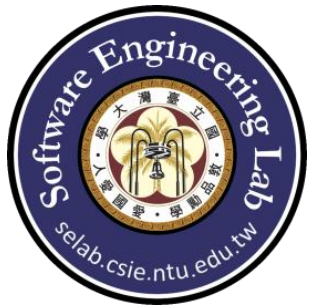
Delegate **last** and **remove** for **pop**



# Stack Implementation Example

```
public class Stack {  
    private List body;  
  
    public void push(Object item) {  
        body.add();  
    }  
  
    public Object pop() {  
        Object o = null;  
        if (body.last() != null) {  
            o = body.last();  
            body.remove();  
        }  
        return o;  
    }  
}
```





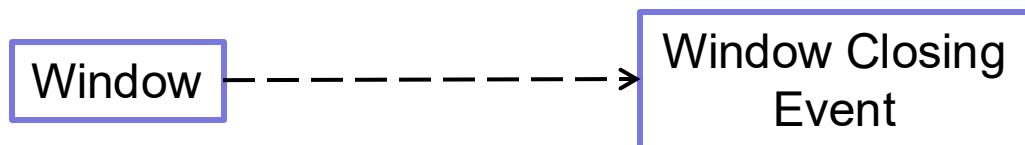
# Relationship<sub>1</sub>

❑ Dependency is the weakest relationship between classes.

➤ Uses-a

➤ A transient relationship, that is, it doesn't retain a relationship for any real length of time

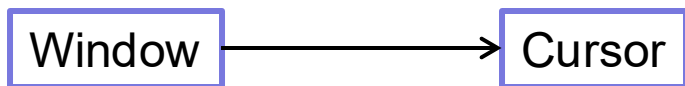
➤ A dependent class **briefly** interacts with the target class

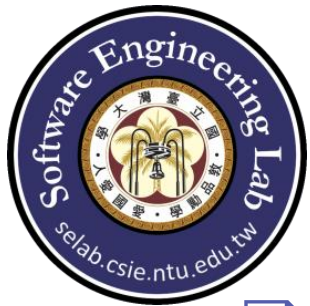


❑ Association is stronger than dependency.

➤ One class retains a relationship to another class **over an extended period of time**

➤ Has-a

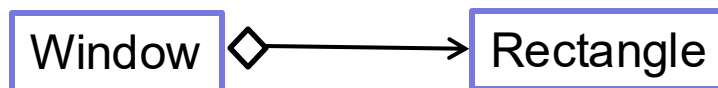




# Relationship<sub>2</sub>

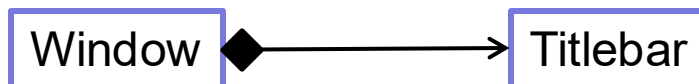
- ❑ Aggregation is a stronger version of association.

- Implies ownership
- Owns-a
- 



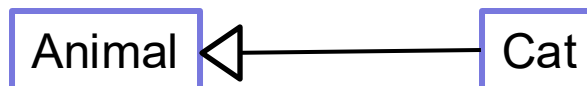
- ❑ Composition represents a **very strong relationship** between classes to the point of containment.

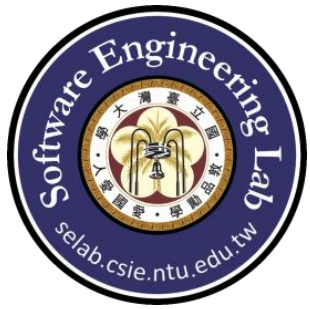
- A whole-part relationship
- Is-part-of
- 



- ❑ Generalization is **the strongest relationship**

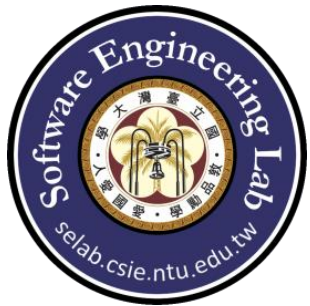
- Is-a
- 



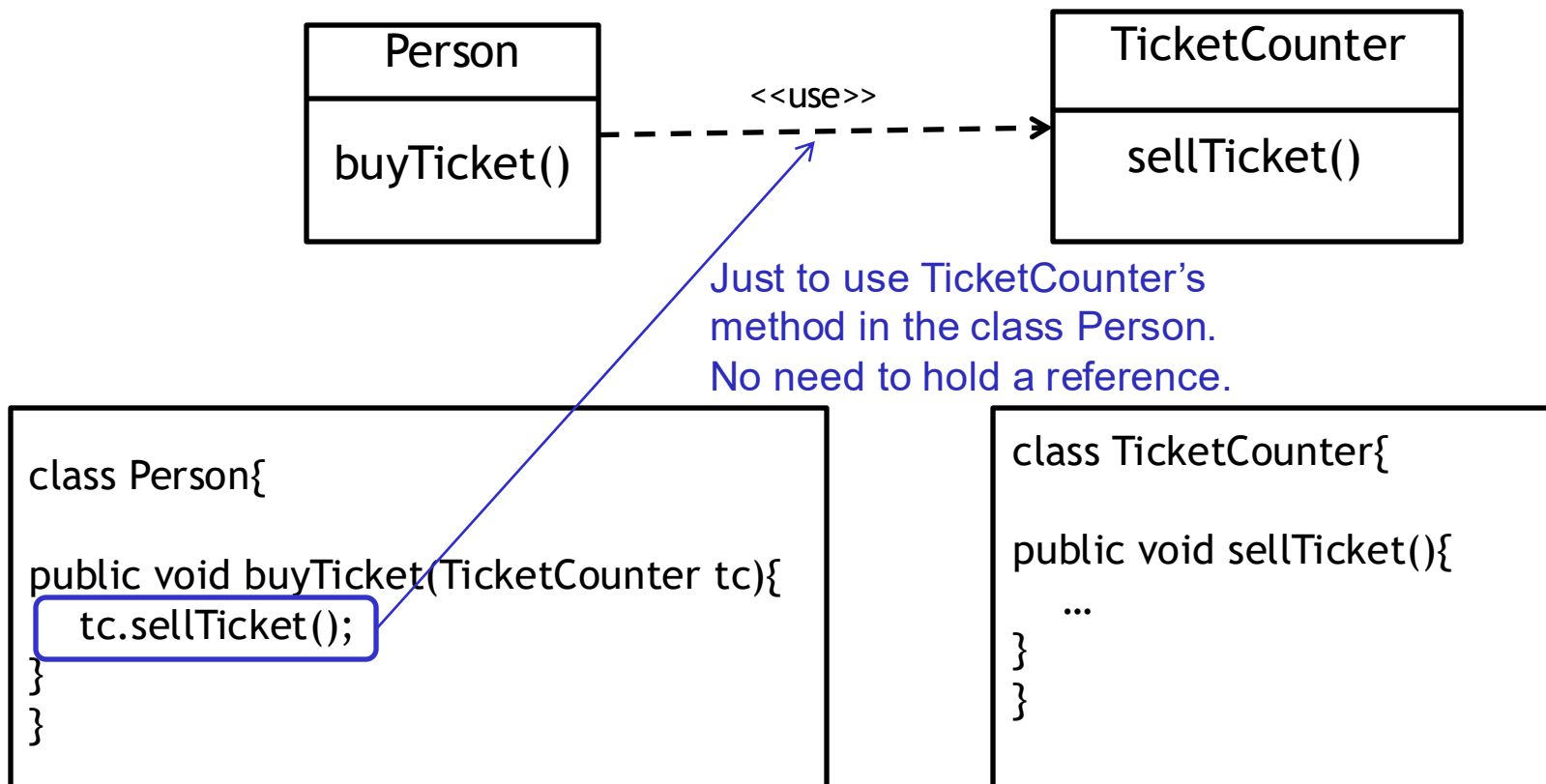


# Dependency<sub>1</sub>

- ❑ A dependency states that the implementation or functioning of one or more elements requires the presence of one or more other elements.
- ❑ Types of stereotype that can be associated with dependency.
  - <<use>>
    - A usage dependency is one in which the client requires the presence of the supplier for its correct functioning or implementation.
  - <<create>>
    - A create dependency signifies that the source class creates one or more instances of the target class.



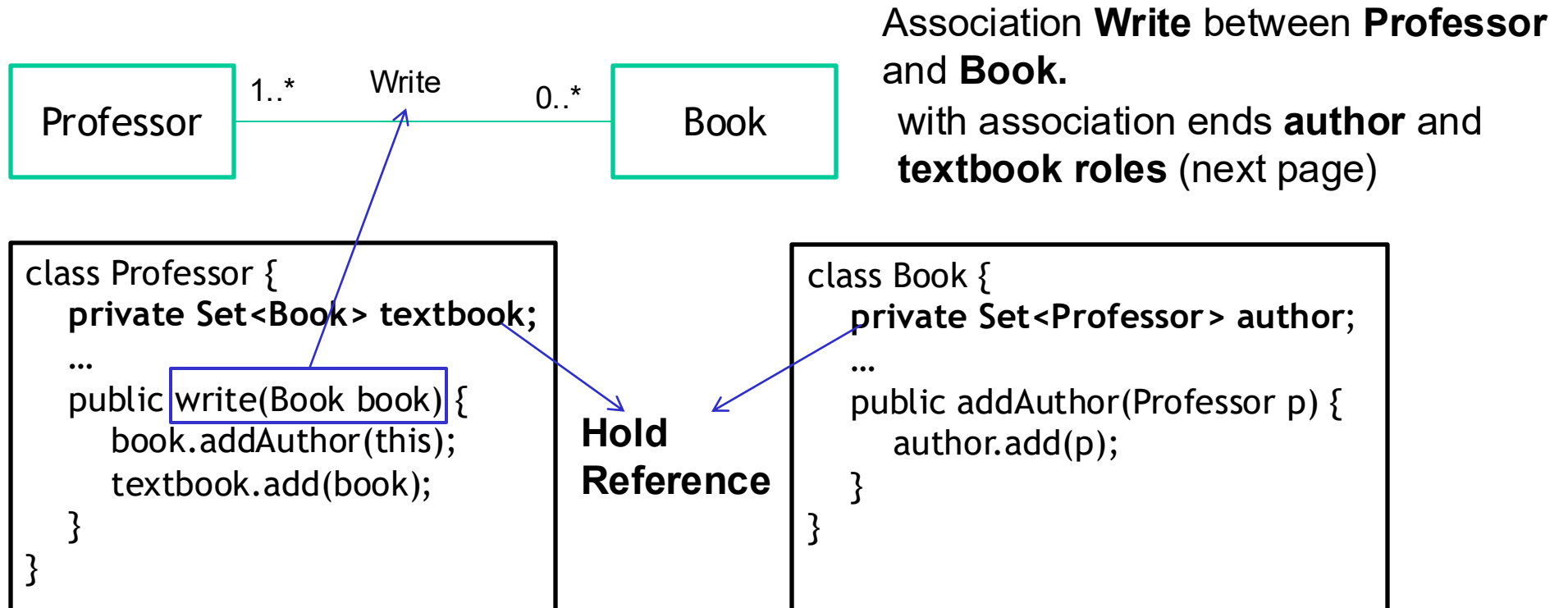
# Dependency<sub>2</sub>



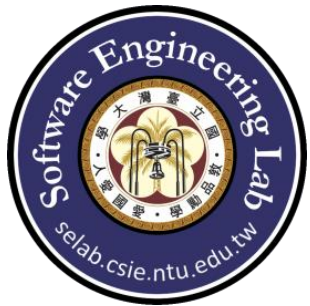


# Association

- Association is a relationship between classes used to show that instances of classes could be either **linked** to each other or combined logically or physically into some **aggregation**.

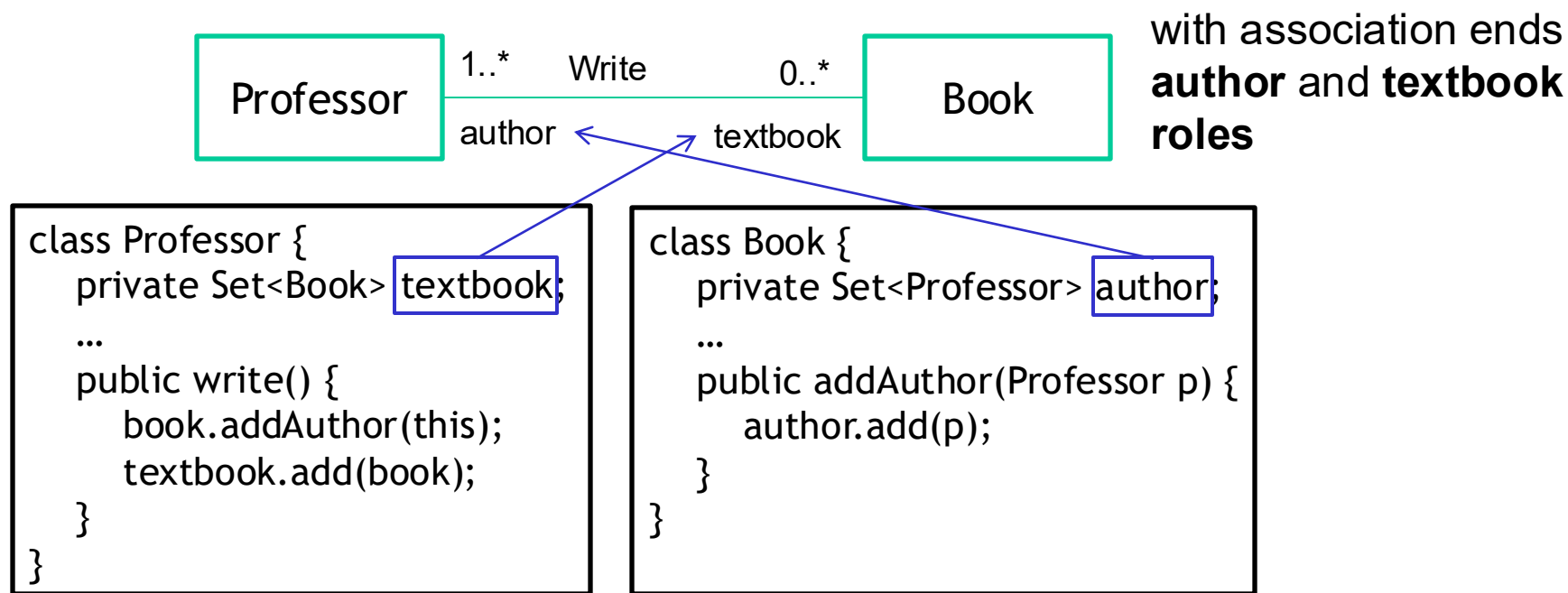


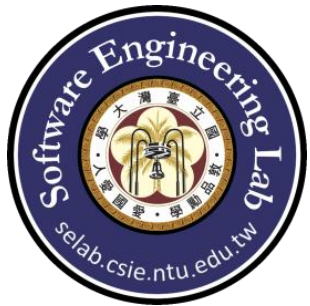




# Role

- Association end is a connection between the line depicting an association and the icon depicting the connected class.
  - The association end name is commonly referred to as role name.
  - The role name is optional.





# Arity

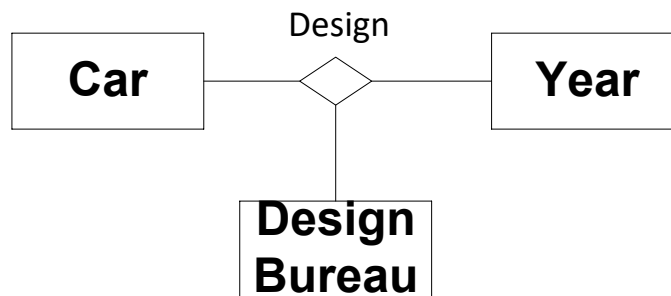
- ❑ Each association has specific arity as it could relate two or more items.

- Binary Association



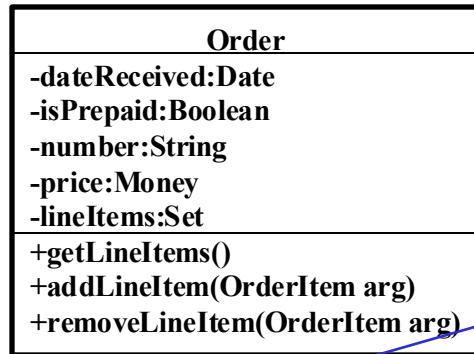
Study and Year classifiers are associated

- N-ary Association

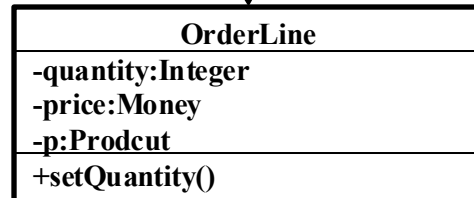


Ternary association Design relates three classifiers

# Association Example<sub>1</sub>

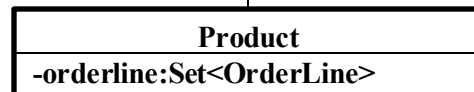


(multiplicity) 1..\* ↓ lineItems (role)



\* (multiplicity)

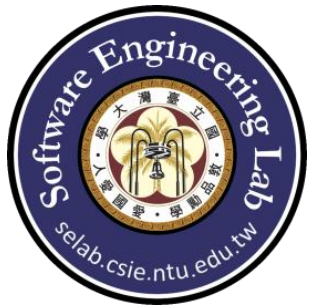
1



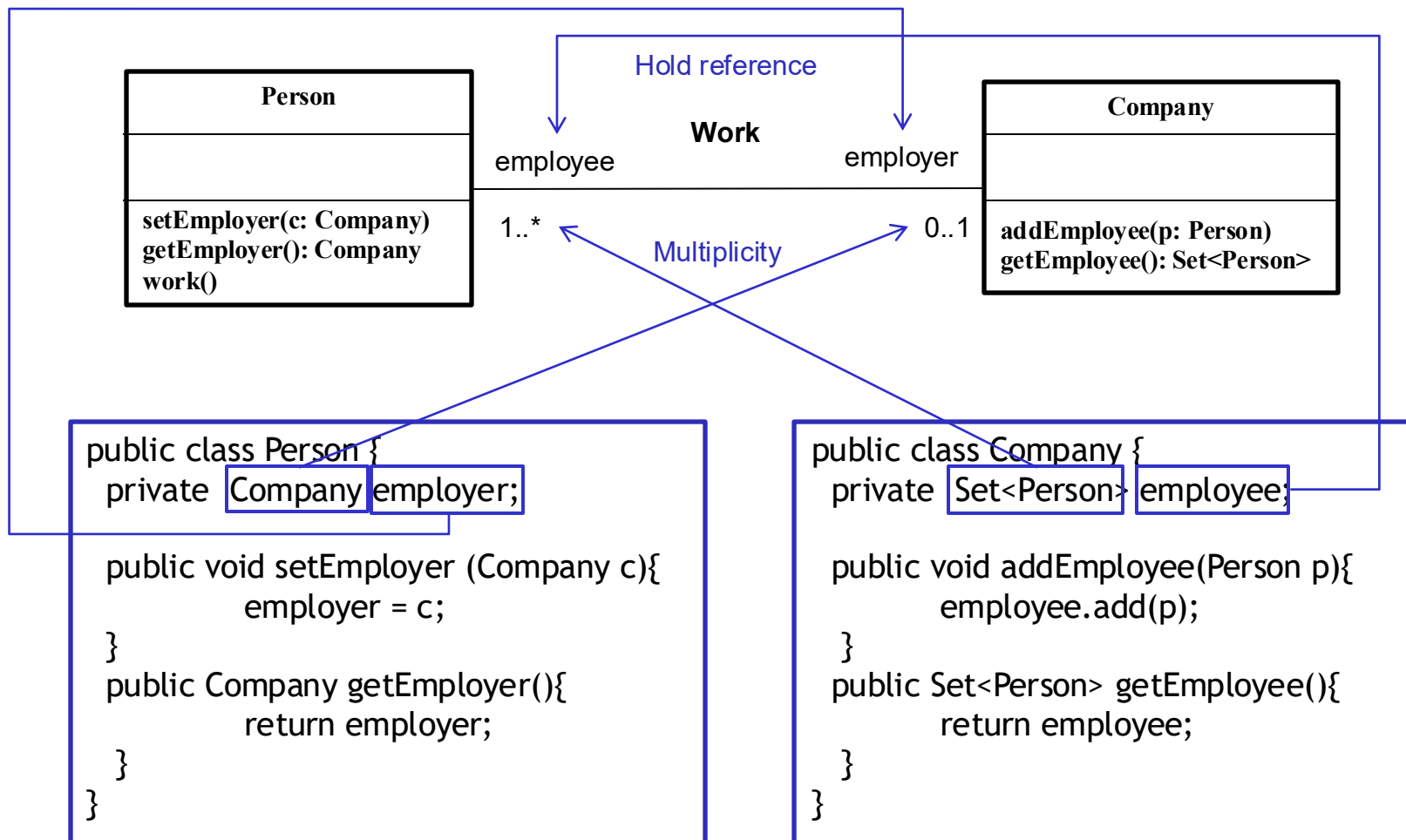
```
public class Order {
    private Date dateReceived;
    private Boolean isPrepaid;
    private String number;
    private Money price;
    private Set<OrderLine> lineItems = new HashSet();
    ...
    public Set getLineItems() { return Collections.unmodifiableSet(lineItems); }
    public void addLineItem (OrderItem arg) { lineItems.add(arg); }
    public void removeLineItem (OrderItem arg) { lineItems.remove(arg); }
}
```

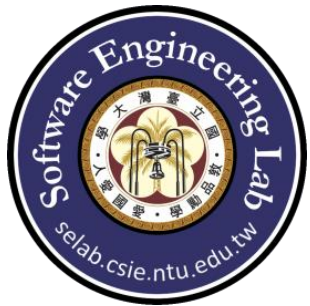
```
public class OrderLine ...
    private int quantity;
    private Money price;
    private Product p;
    ...
    public void setQuantity(int quantity) { this.quantity = quantity; }
    ...
}
```

```
public class Product {
    private Set<OrderLine> orderline;
    ...
}
```



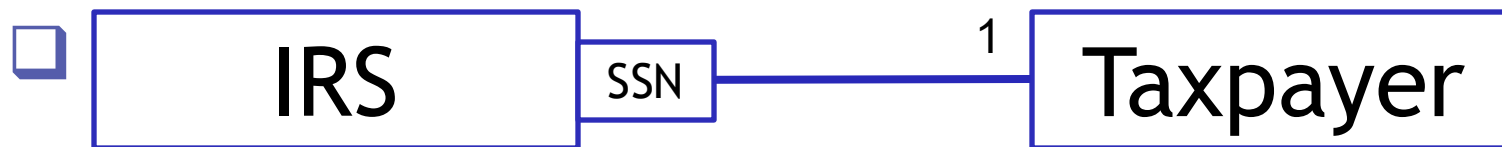
# Association Example<sub>2</sub>





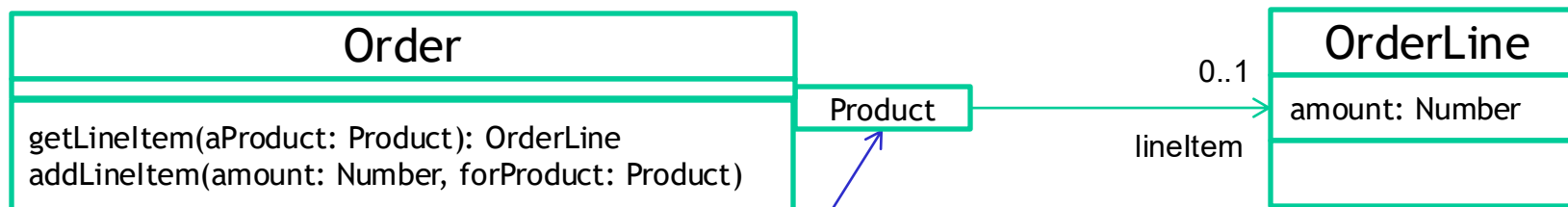
# Association Qualifier

- ❑ Association Qualifiers to capture such information as relationships between elements are often **keyed or indexed** by some other value.



# Association Qualifier Example

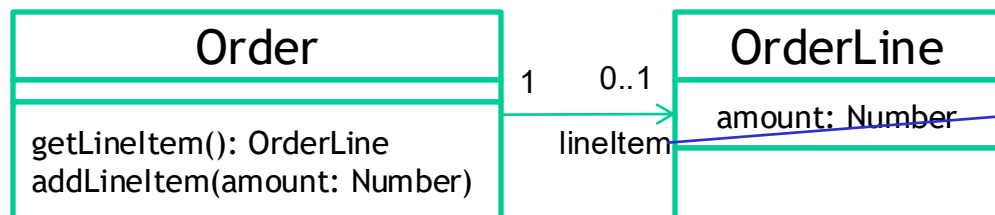
- The qualifier says that in the connection with an Order, there may be one Order Line for each instance of Product.



```

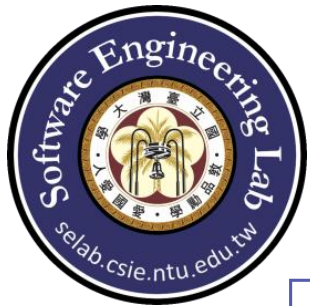
class Order{
    Hashtable<Product,OrderLine> qualifier;
    public OrderLine getLineItem(Product aProduct) { qualifier.get(aProduct); }
    public void addLineItem(Number amount, Product forProduct) {...}
}
    
```

Without qualified association



```

class Order {
    private OrderLine lineItem;
    public OrderLine getLineItem() {...}
    public void addLineItem(Number amount) {...}
}
    
```



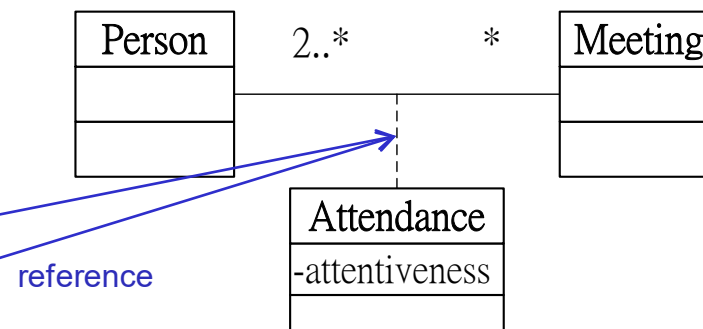
# Association Class

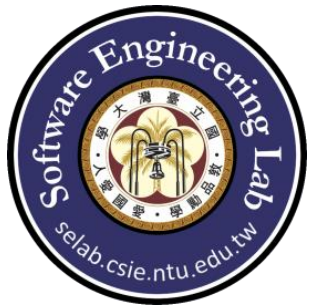
- ❑ An association has attributes associated with the association itself (not just the participating objects)
- ❑ Implementation
  - Each participating object contains a reference to the association class object
  - The association class object contains references to each of the related objects

```
class Person {  
    ...  
    (some sort of collection of references  
    to Attendance objects) attendance;  
    ...  
}
```

```
class Meeting {  
    ...  
    (some sort of collection of references  
    to Attendance objects) attendance;  
    ...  
}
```

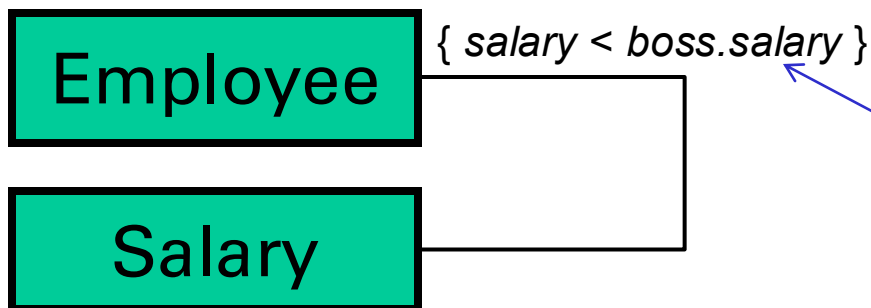
```
class Attendance {  
    ...  
    Person person;  
    Meeting meeting;  
    int attentiveness;  
    ...  
}
```





# Constraints<sub>1</sub>

- ❑ A constraint restricts the values that entities can assume, which is defined as functional relationships between entities of an object model.
  - entity: objects, classes, attributes, links, and associations.
  - e.g. No employee's salary can exceed the salary of the employee's boss.



```
public class Employee {
    private Salary salary;
    private Boss boss;

    public void setSalary(Salary salary)
        throws SalaryExceededException {
        if (salary.getValue() >= boss.salary.getValue())
            throw new SalaryExceededException();
        this.salary = salary;
    }
}
```

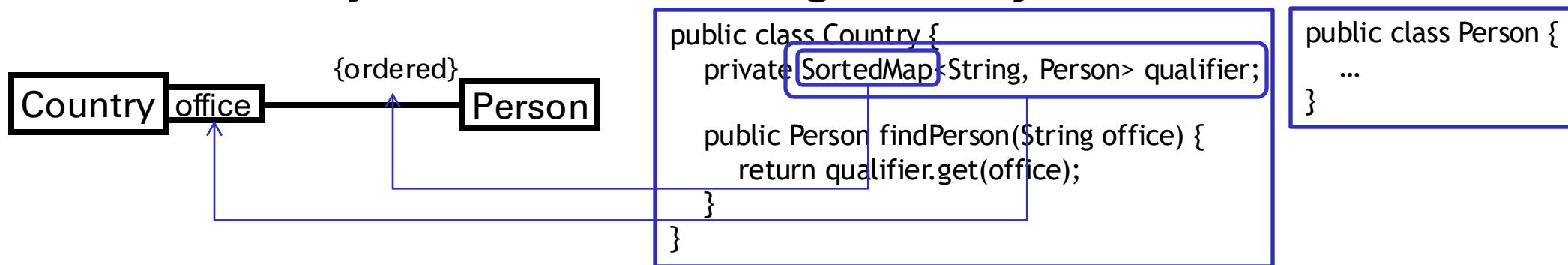
```
public class Salary {
    private double salary;

    public double getValue() {
        return salary;
    }
}
```



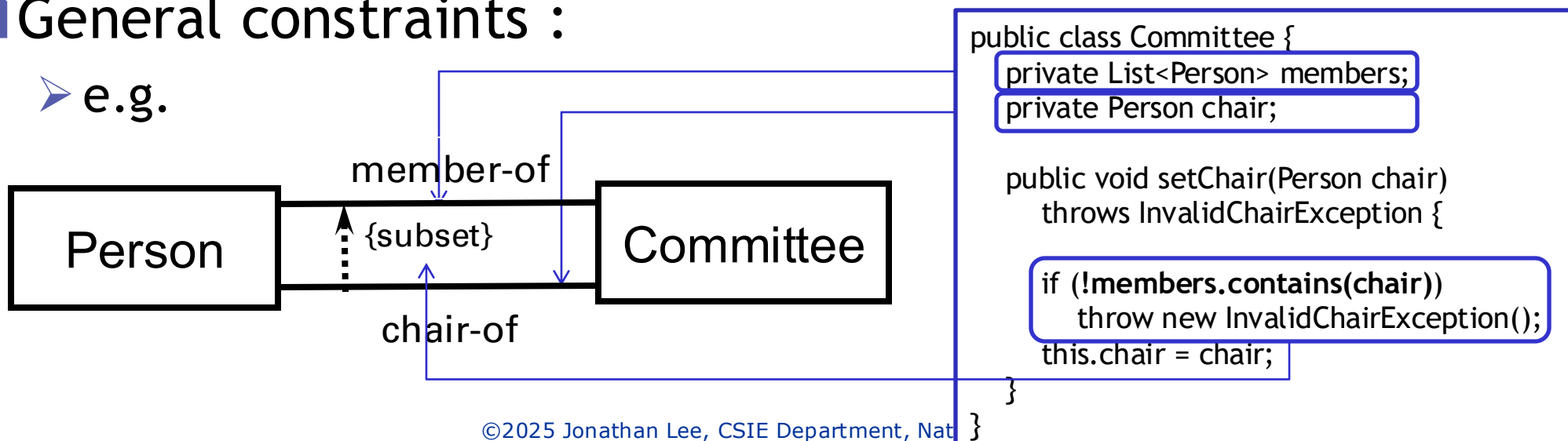
# Constraints<sub>2</sub>

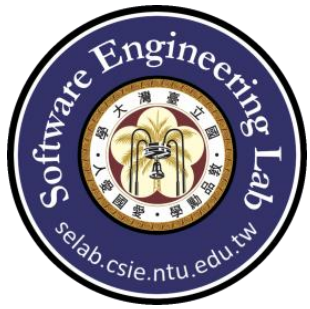
- Multiplicity constrains an association. That is, it restricts the number of objects related to a given object.



- General constraints :

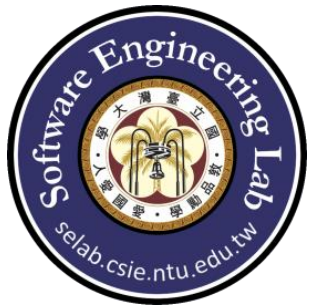
➤ e.g.





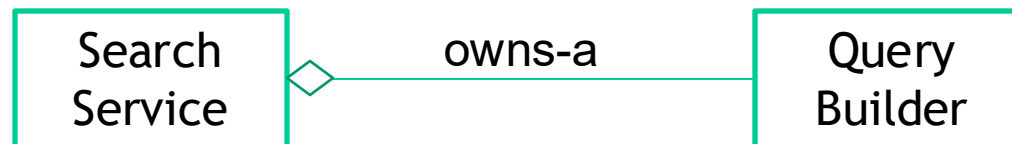
# Aggregation Type

- ❑ An association may represent a composite aggregation (i.e., composition or a whole/part relationship).
  - Composite aggregation is a strong form of aggregation that requires a **part instance be included in at most one composite** at a time. (Composition)
  - If a composite is deleted, all of its parts are normally deleted with it.
- ❑ Aggregation type could be:
  - Shared aggregation (aggregation)
  - Composite aggregation (composition)



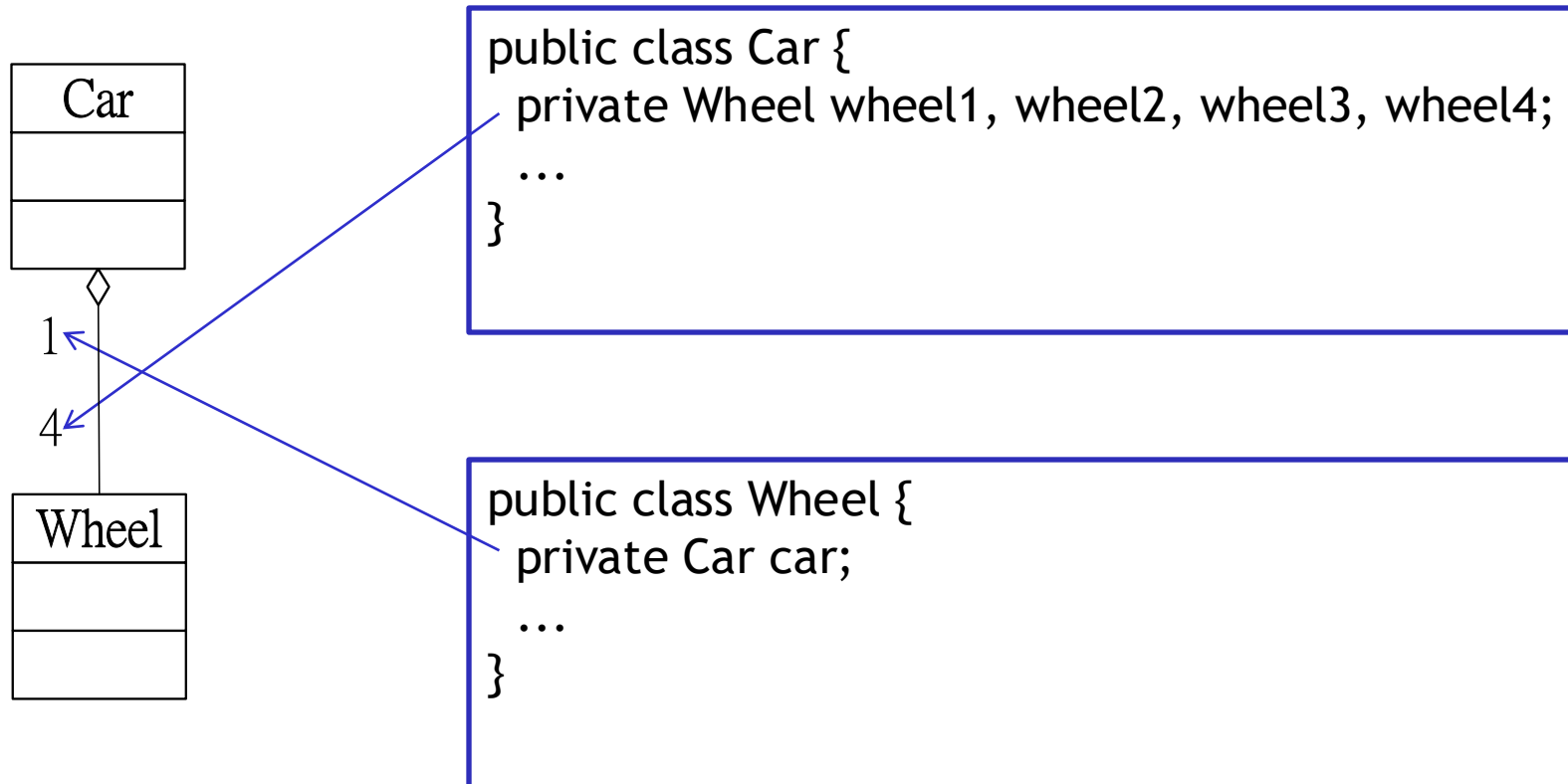
# Aggregation

- ❑ Aggregation is a “weak” form of aggregation when **part instance is independent of the composite**:
  - The same (shared) part could be included in several composites, and
  - If composite is delete, shard parts may still exist.



**Search Service** has a **Query Builder** using shared aggregation

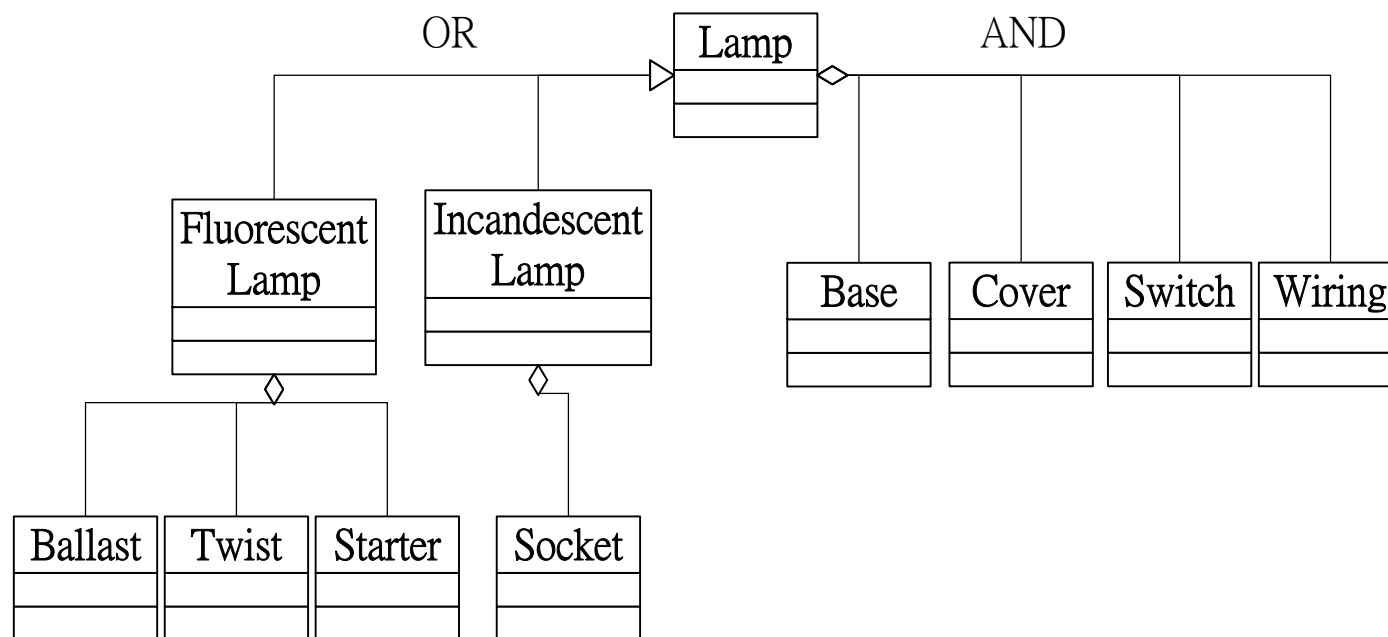
# Aggregation Example

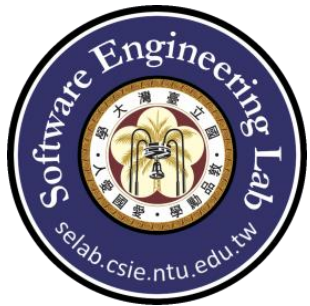


# Aggregation

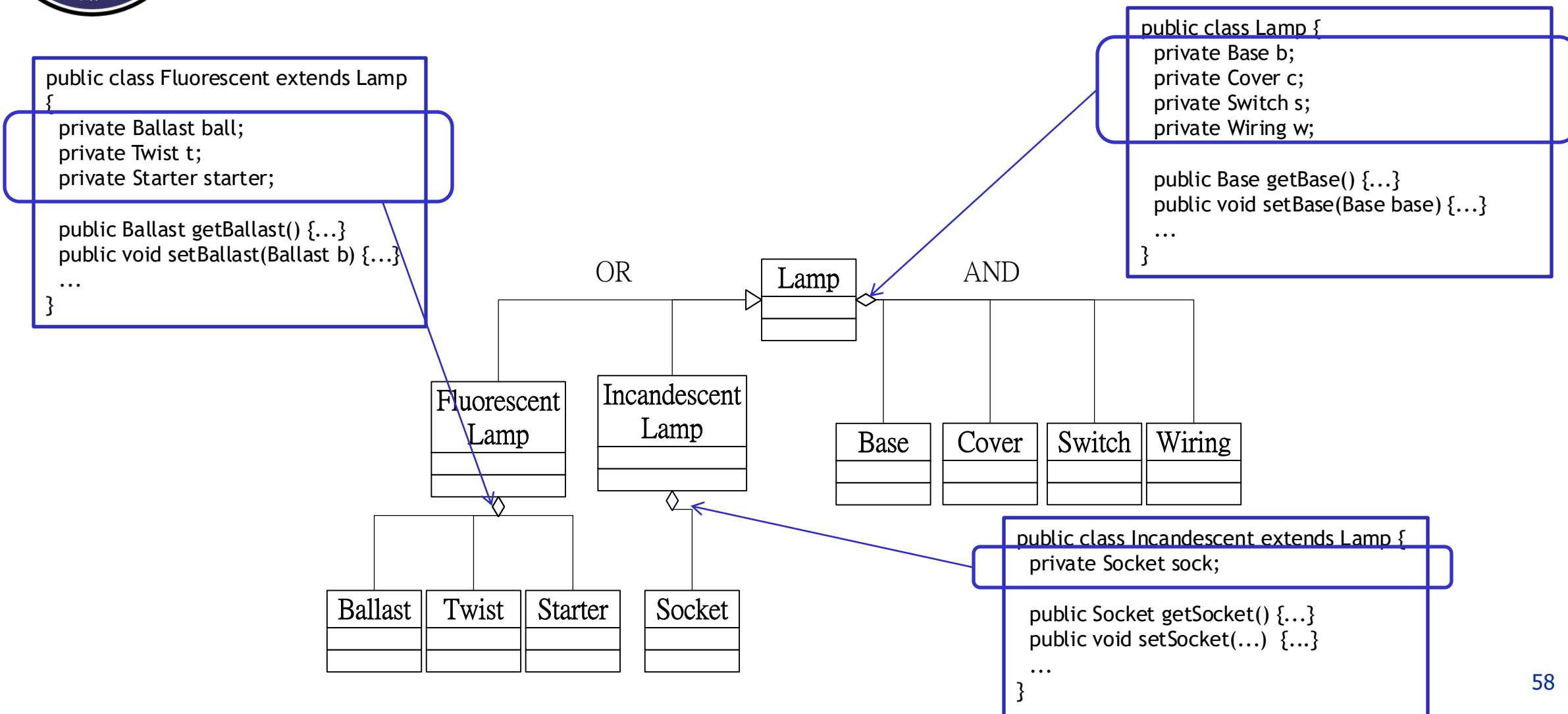
## □ Aggregation or generalization

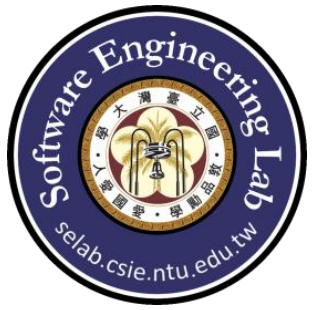
- an aggregation tree is composed of object instances that are all parts of a composite object.
- a generalization tree is composed of classes.





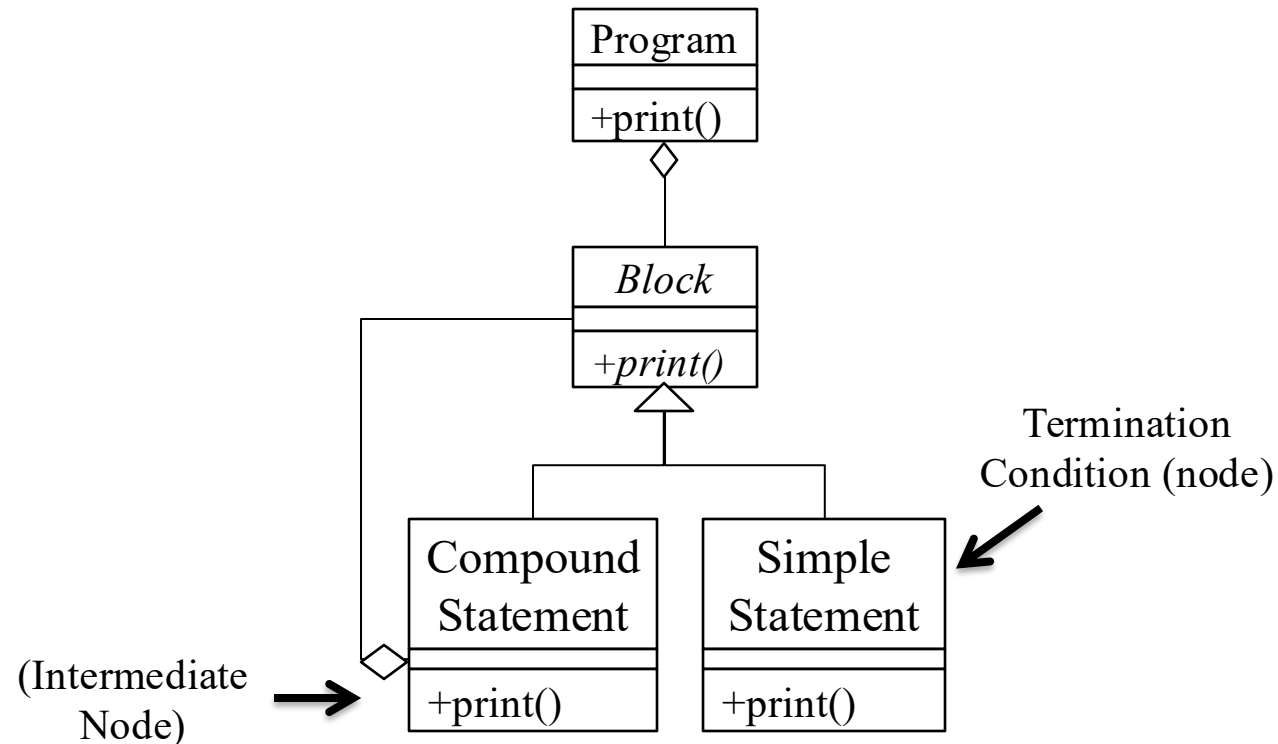
# Aggregation: Java Code

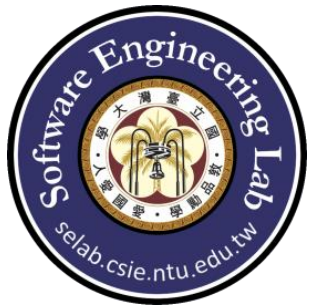




# Recursive Aggregation

- ❑ Recursive: contains an instance of the same kind of aggregate, the number of potential levels is unlimited.





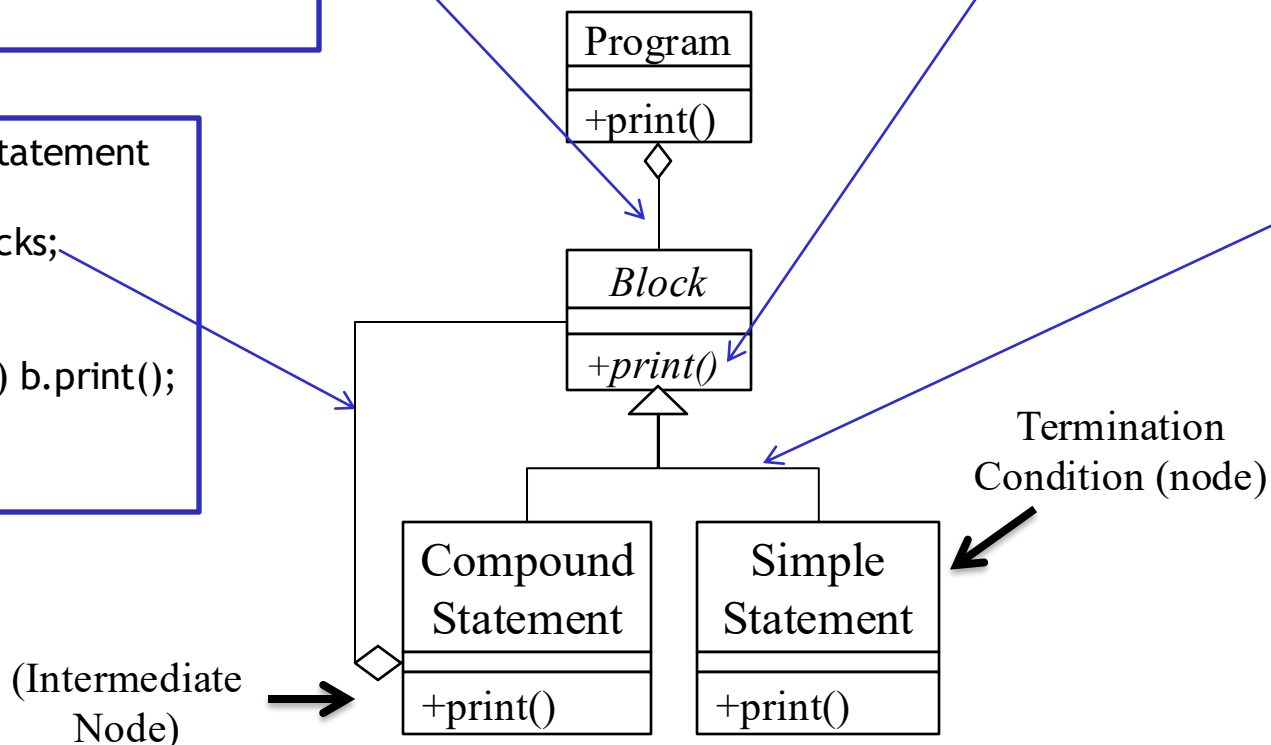
# Recursive Aggregation: Java Code

```
public class Program {  
    private Set<Block> blocks;  
    ...  
    public void print() {  
        for (Block b in blocks)  
            b.print();  
    }  
}
```

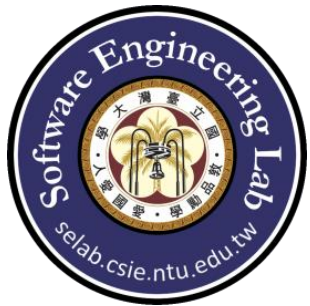
```
public abstract class Block {  
    ...  
    public abstract void print();  
}
```

```
public class CompoundStatement  
extends Block {  
    private Set<Block> blocks;  
    ...  
    public void print() {  
        for (Block b in blocks) b.print();  
    }  
}
```

```
public class SimpleStatement  
extends Block {  
    ...  
    public void print() {  
        // print statement  
    }  
}
```







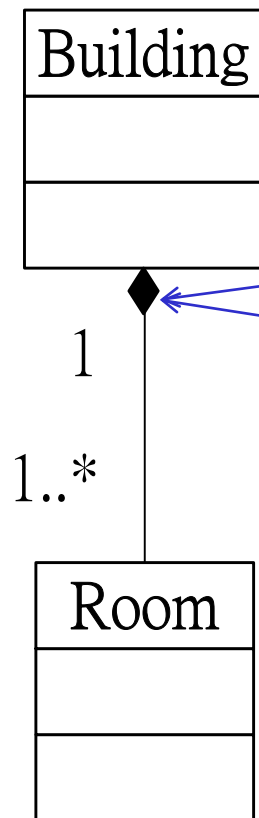
# Composition

- ❑ Composition is a “strong” form of aggregation where the whole and parts have coincident lifetimes.
  - It is a whole/part relationship,
  - It is binary association,
  - Part could be included in at most one composite (whole) at a time,
  - If a composite (whole) is deleted, all of its composite parts are “normally” deleted with it.
- ❑ **A Composition adds a lifetime responsibility to *Aggregation***



**Folder** could contain many **files**, while each **File** has exactly one **Folder** parent.  
If **Folder** is deleted, all contained Files are deleted as well.

# Composition Example<sub>1</sub>



```
public class Building {
    private Set<Room> rooms;
```

```
    public Building() {
```

Create Room objects

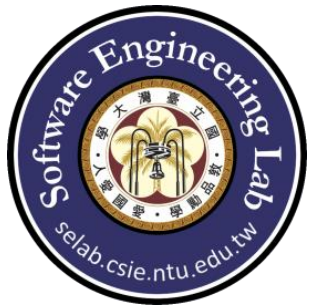
```
        rooms = new Set<Room> ();
    }
```

```
    protected void finalize() {
```

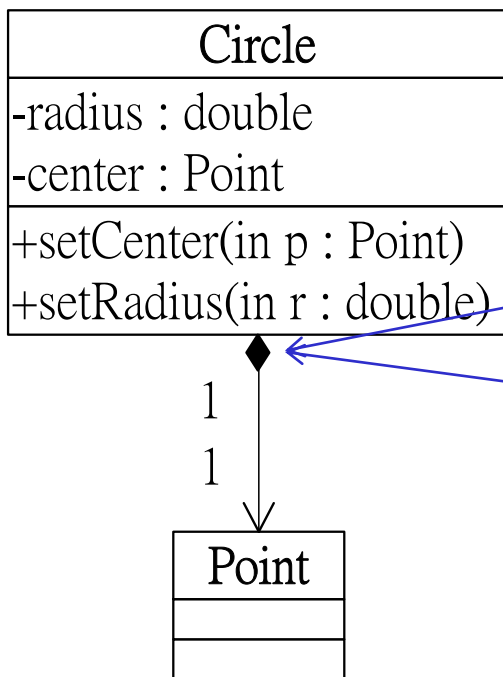
Destroy Room objects

```
        rooms = null;
    }
```

```
public class Room {
    private ...;
    private ...;
    ...
}
```



# Composition Example<sub>2</sub>



```
public class Circle {
    double radius;
    Point center;

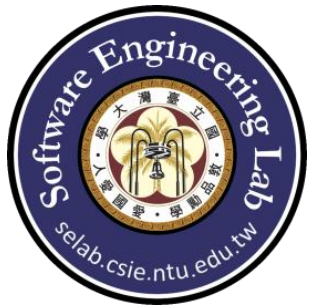
    public Circle(Point c, double r) {
        this.radius = r;
        this.center = c;
    }

    protected void finalize() {
        this.radius = 0;
        this.center = null;
    }
    ...
}

public class Point {
    private ...;
    private ...;
    ...
}
```

Create Point object

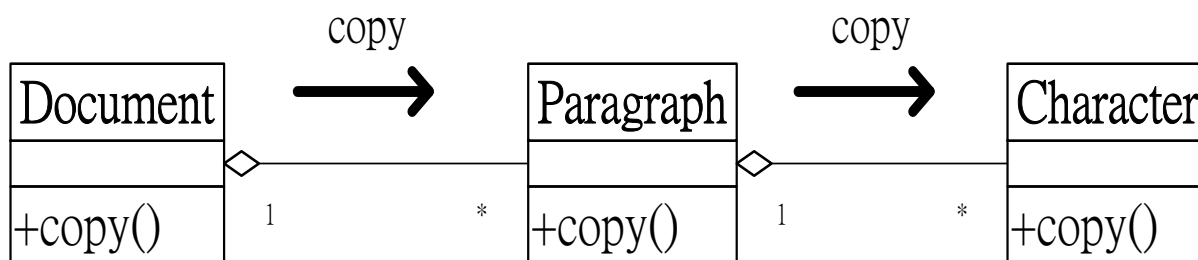
Destroy Point object



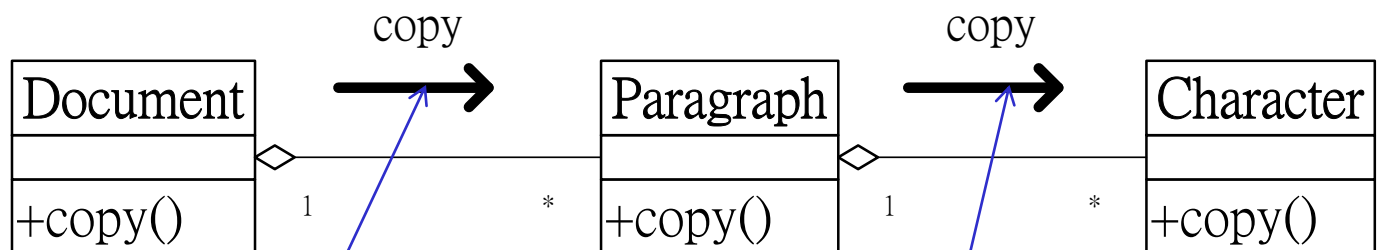
# Propagation of Operations

## □ Propagation of operations

- automatically applying an operation to a network of objects.
- an operation can be thought of as starting at some initial object and flowing from object to object through links.



# Propagation Java Code



```

public class Document {
    private Set<Paragraph> paragraph;
    public Document copy() {
        Document d = new Document();
        for(Paragraph p in paragraph)
            d.add(p.copy());
        return d;
    }
}
  
```

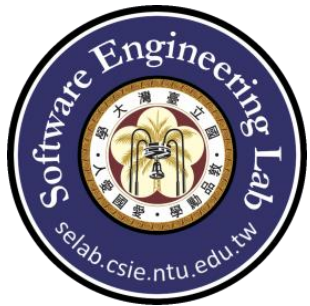
```

public class Paragraph {
    private Set<Character> character;
    public Paragraph copy() {
        Paragraph p = new Paragraph();
        for(Character c in character)
            p.add(c.copy());
    }
}
  
```

```

public class Character {
    public Character copy()
    {
        return this.clone();
    }
}
  
```

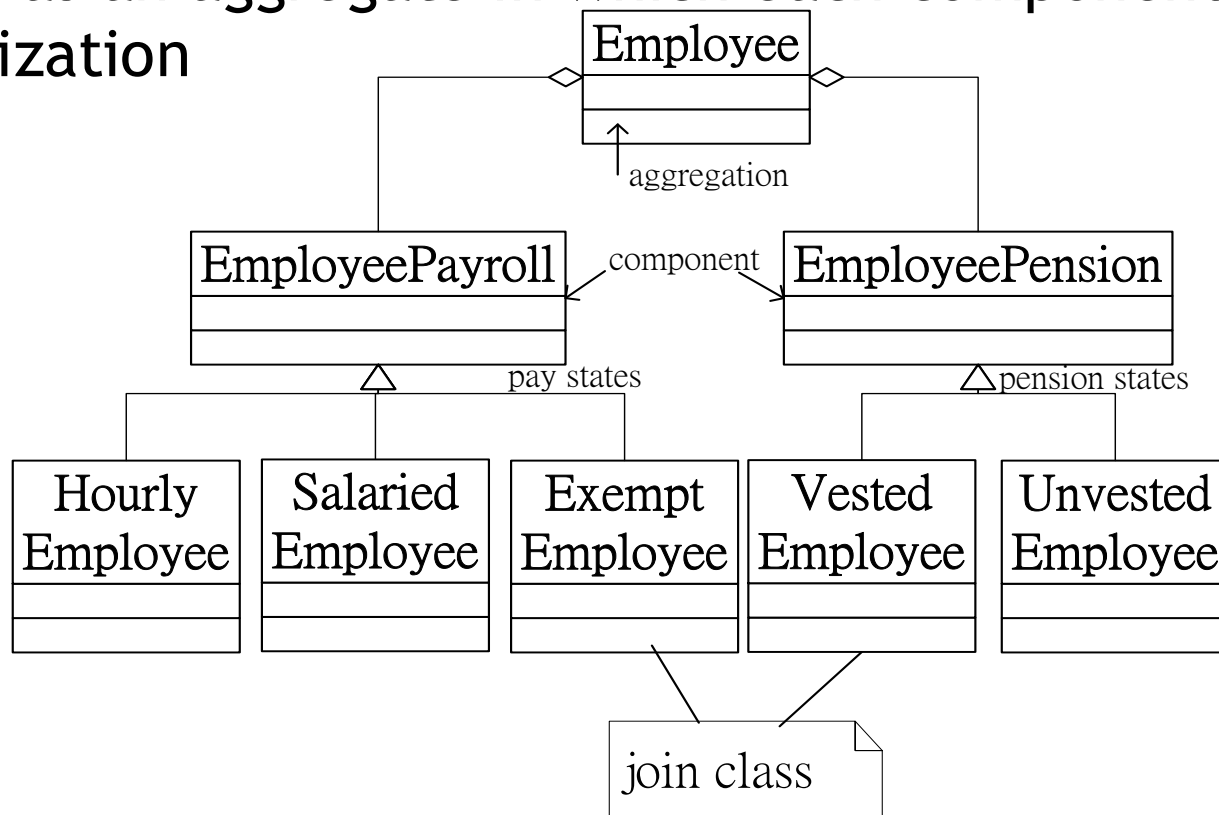
This method creates a new instance with the same values of the attributes of the Character object.

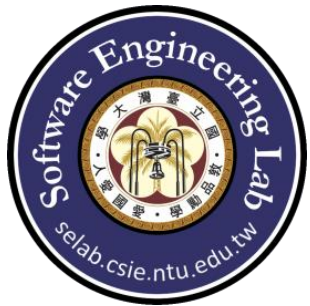


# Resolution of Multiple Inheritance<sub>1</sub>

## ❑ Delegation using aggregation

- A super class with multiple independent generalizations can be recast as an aggregate in which each component replaces a generalization





```
public class EmployeePayroll {  
    public void getPayroll() {  
        System.out.println("Not  
        decided yet!");  
    }  
    ...  
}
```

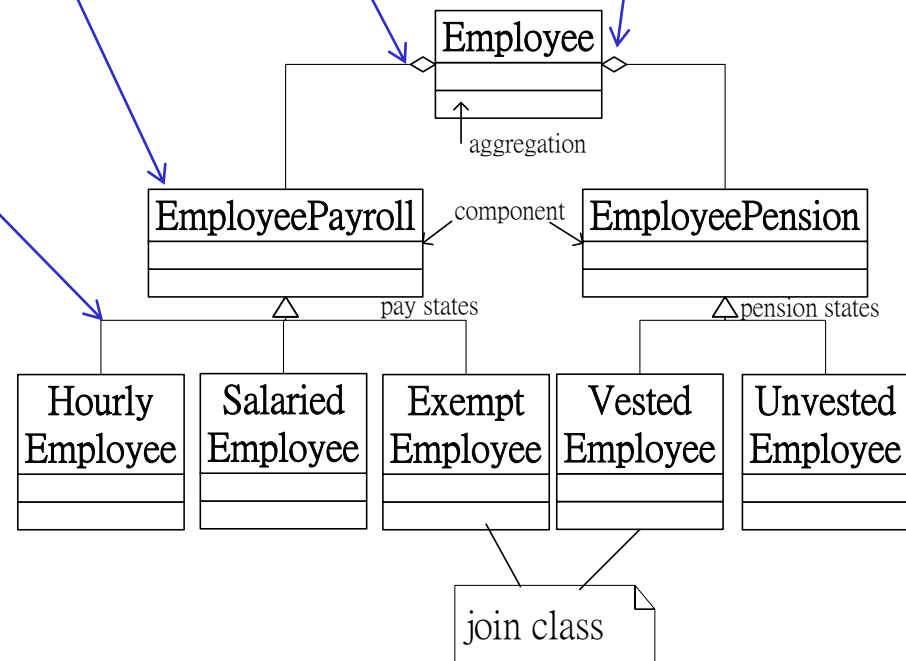
```
public class HourlyEmployee  
    extends EmployeePayroll {
```

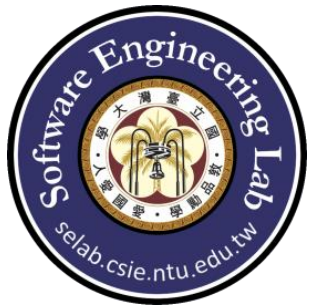
```
    public void getPayroll() {  
        System.out.println("Hourly  
        Employee Payroll");  
    }  
    ...  
}
```

```
Employee employee = new Employee();  
employee.setPayroll(new HourlyEmployee());  
employee.getPayroll();
```

↓ outputs  
Hourly Employee Payroll

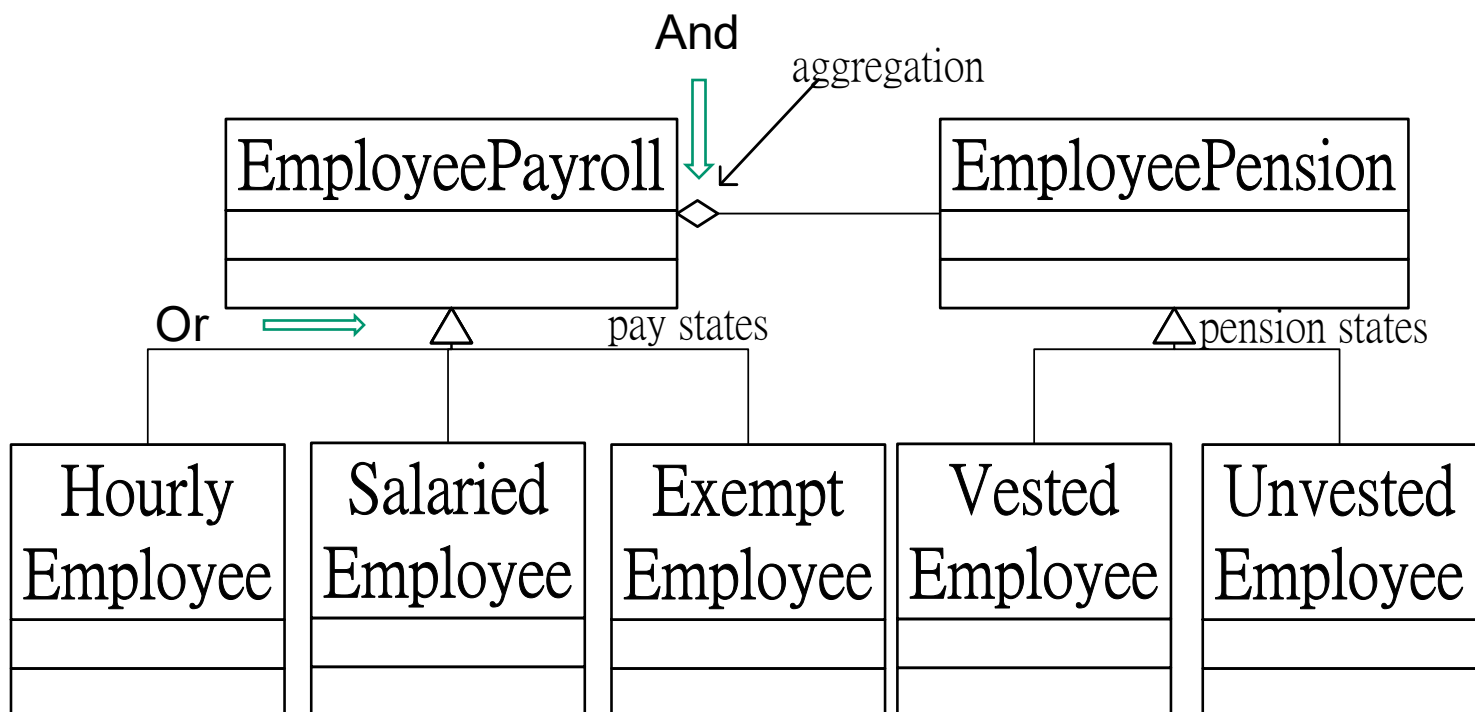
```
public class Employee {  
    private EmployeePayroll ePay;  
    private EmployeePension ePension;  
  
    public void setPayroll(EmployeePayroll ep) {  
        this.ePay = ep;  
    }  
    public void getPayroll() {  
        ePay.getPayroll();  
    }  
    ...  
}
```



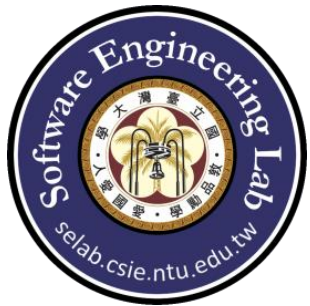


# Resolution of Multiple Inheritance<sub>2</sub>

- ❑ Inherit the most important class and delegate the rest:
  - to make a join class a subclass of its most important superclass.





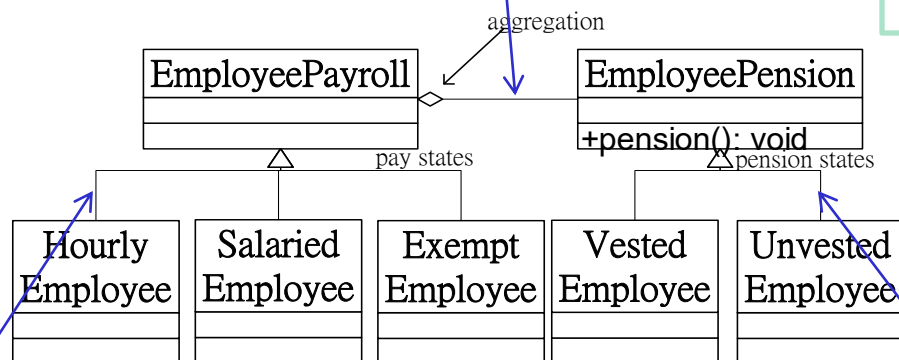


```
public class EmployeePayroll {  
    private EmployeePension ePension;  
    ...  
    public void ...(...) {  
        ePension.pension();  
    }  
}
```

delegate

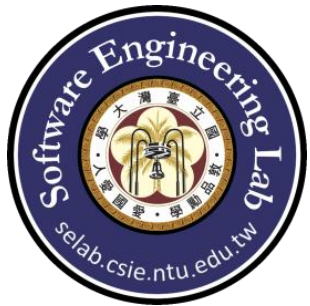
through

```
public class EmployeePension  
{  
    public void pension() {...}  
}
```



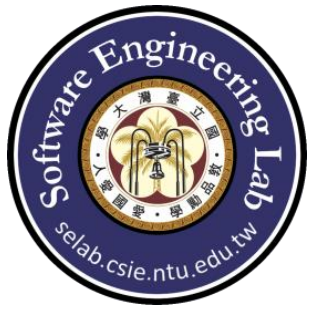
```
public class HourlyEmployee  
extends EmployeePayroll {  
    ...  
}
```

```
public class VestedEmployee extends  
EmployeePension {  
    public void pension() {...}  
}
```



# Delegation vs Composition

- ❑ Use Delegation (or Aggregation) when you have a task to perform, but you don't want to do it by yourself. Dispatch the task (via a request) to a class that encapsulates a functionality for performing the task.
- ❑ Use Composition when you view another class or classes as a part that belongs only to you. You can use the functionality provided by the class/classes via the instantiation.
- ❑ Composition is a much stronger relationship than Delegation.



# Sharpen your Skill

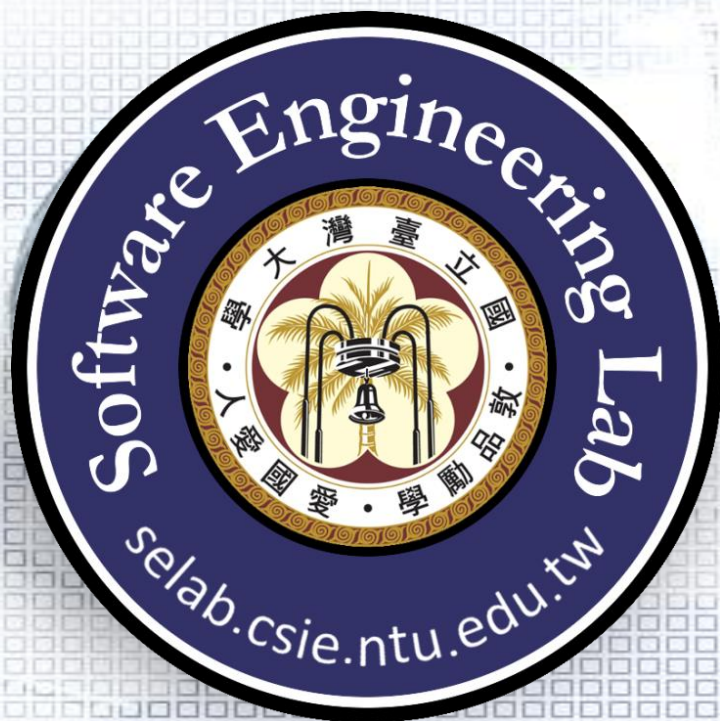
---

- A polygon is composed of an ordered set of points.



# Homework: Problem Statement

- ❑ A person has a name, address, and social security number. A person **may** charge time to projects and earn a salary. A company has a name, address, phone number, and primary product. A company hires and fires persons. Person and Company have a many-to-many relationship.
- ❑ There are two types of persons: workers and managers. Each worker works on many projects; each manager is responsible for many projects. A project is staffed by many workers and exactly one manager. Each project has a name, budget, and internal priority for securing resources.
- ❑ A company is composed of multiple departments; each department within a company is **uniquely identified by its name**. A department usually, but not always, has a manager. Most managers manage a department; a few managers are not assigned to any department. Each department manufactures many products; while each product is made by exactly one department. A product has a name, cost, and weight.



# Homework: Sharpen Your Skill

Prof. Jonathan Lee (李允中)

Department of Computer Science and  
Information Engineering  
National Taiwan University



# Sharpen Your Skill<sub>1</sub>

---

- ☐ A country has a capital city.
- ☐ A dining philosopher is using a fork.
- ☐ Files contain records.
- ☐ A drawing object is text, a geometrical object, or a group.
- ☐ A polygon is composed of an ordered set of points.

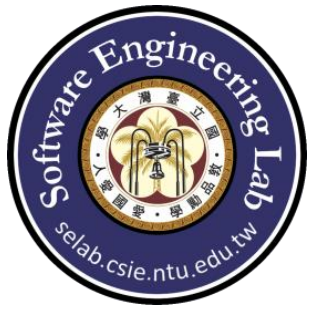




# Sharpen Your Skill<sub>2</sub>

---

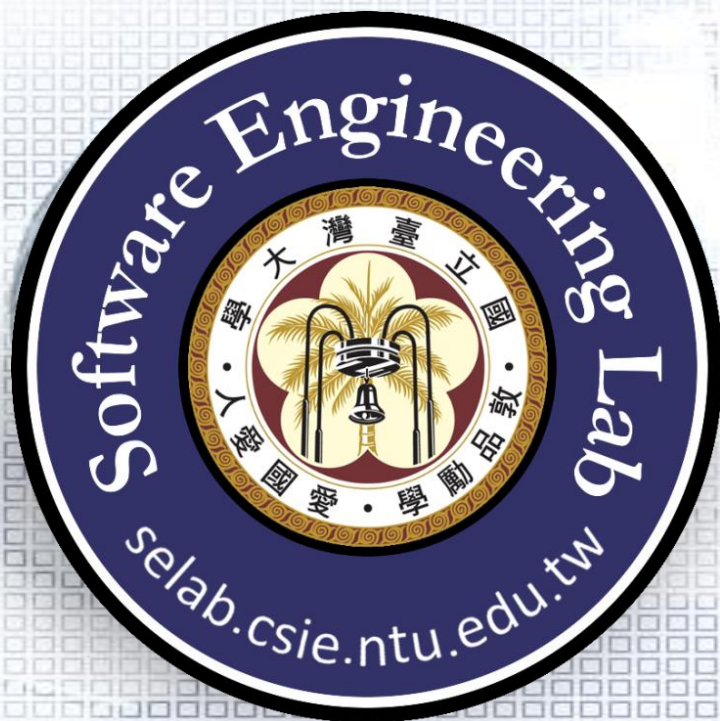
- ☐ A person uses a computer language on a project.
- ☐ Modems and keyboards are input/output devices.
- ☐ Object classes may have several attributes.
- ☐ A person plays for a team in a certain year.
- ☐ A route connects two cities.
- ☐ A student takes a course from a professor.



# Sharpen Your Skill<sub>3</sub>

- ❑ Create an application to simulate the Windows/UNIX file system. The file system consists mainly of two types of components – directories and files.
  - Directories can be made up of other directories or files, whereas files cannot contain any other file system component.
  - In this aspect, directories act as nonterminal nodes and files act as terminal nodes of a tree structure.

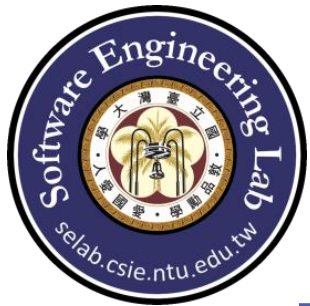




# Homework: A Hospital System

Prof. Jonathan Lee (李允中)

Department of Computer Science and  
Information Engineering  
National Taiwan University



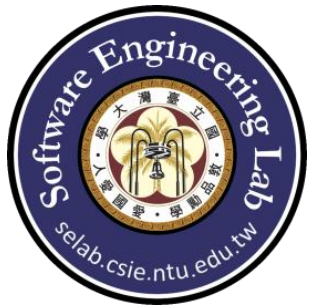
# Requirements Statements

- ☐ A hospital has a large number of registered physicians.
- ☐ Patients are admitted to the hospital by physicians.
- ☐ Any patient who is admitted must have exactly one admitting physician.
- ☐ A physician may optionally admit any number of patients. Once admitted, a given patient must be treated by at least one physician.
- ☐ A particular physician may treat any number of patients, or he or she may not treat any patients.
- ☐ Whenever a patient is treated by a physician, the hospital wishes to record the details of the treatment by including the date, time and results of the treatment.







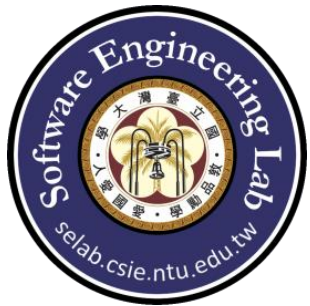
# Sequence Diagram Basic Notation

- ❑ Synchronous Messages are represented by solid line with solid arrow →
- ❑ Return messages are in dashed line with arrow -->
- ❑ Asynchronous message in solid line with arrow can be used for one of the three things: →
  - Create a new thread to link to the top of an activation
  - Create a new object
  - Communicate with a thread that is already running
- ❑ Self-delegation is a message that an object sends to itself. ↩

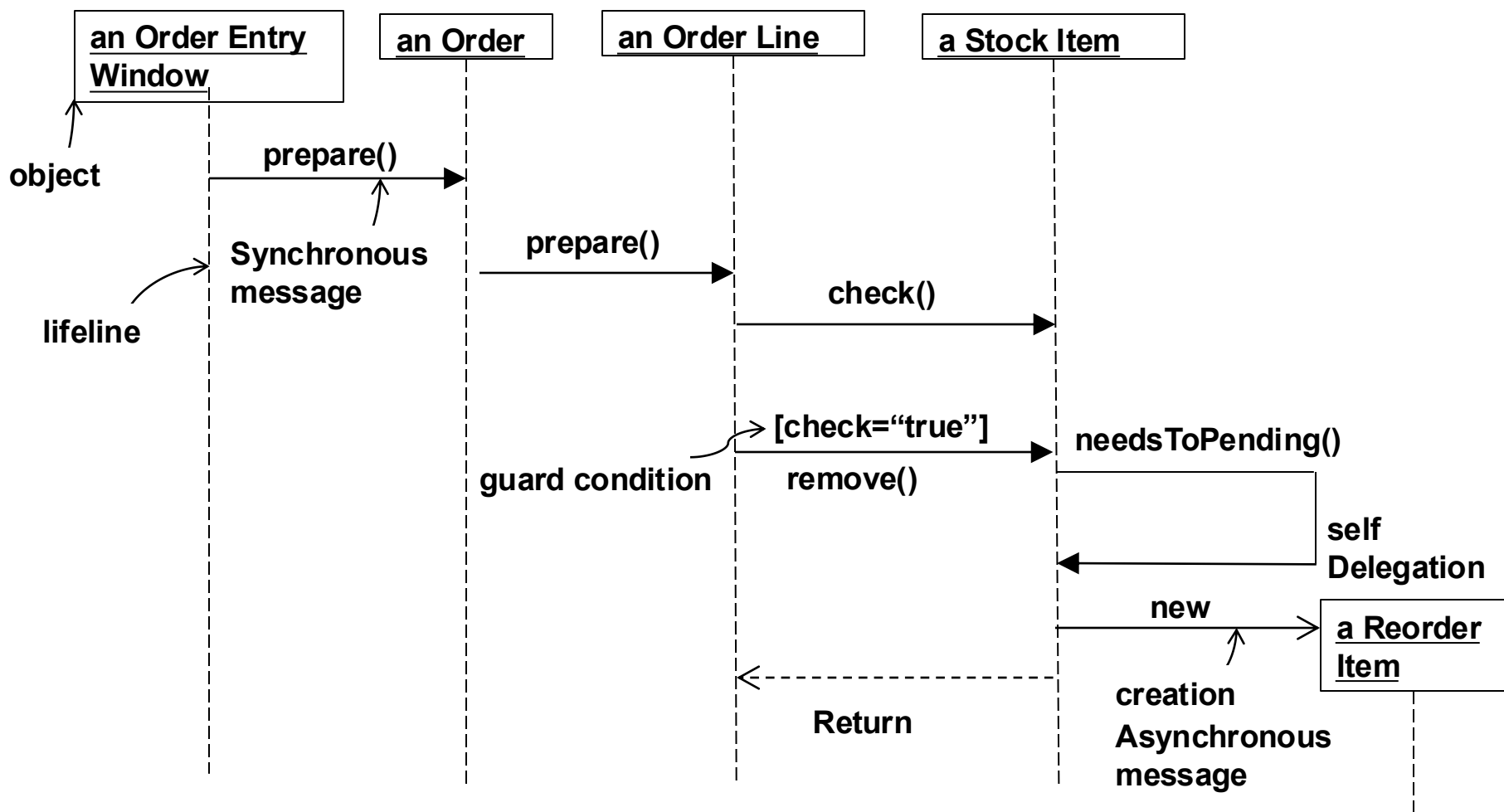


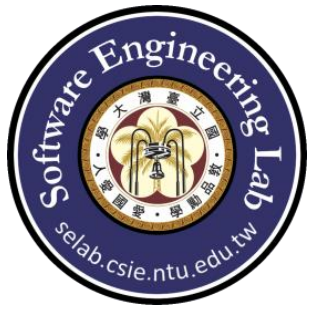
# Basic Notation

- ❑ Object instances are represented by a rectangle box 
- ❑ Lifelines are represented by vertical dashed lines 
- ❑ Activations to indicate a method is active because it is either executing or waiting for a subroutine to return. 
- ❑ Object deletion for objects to self-destruct or to be destroyed by another message 



# Sequence Diagram Example A



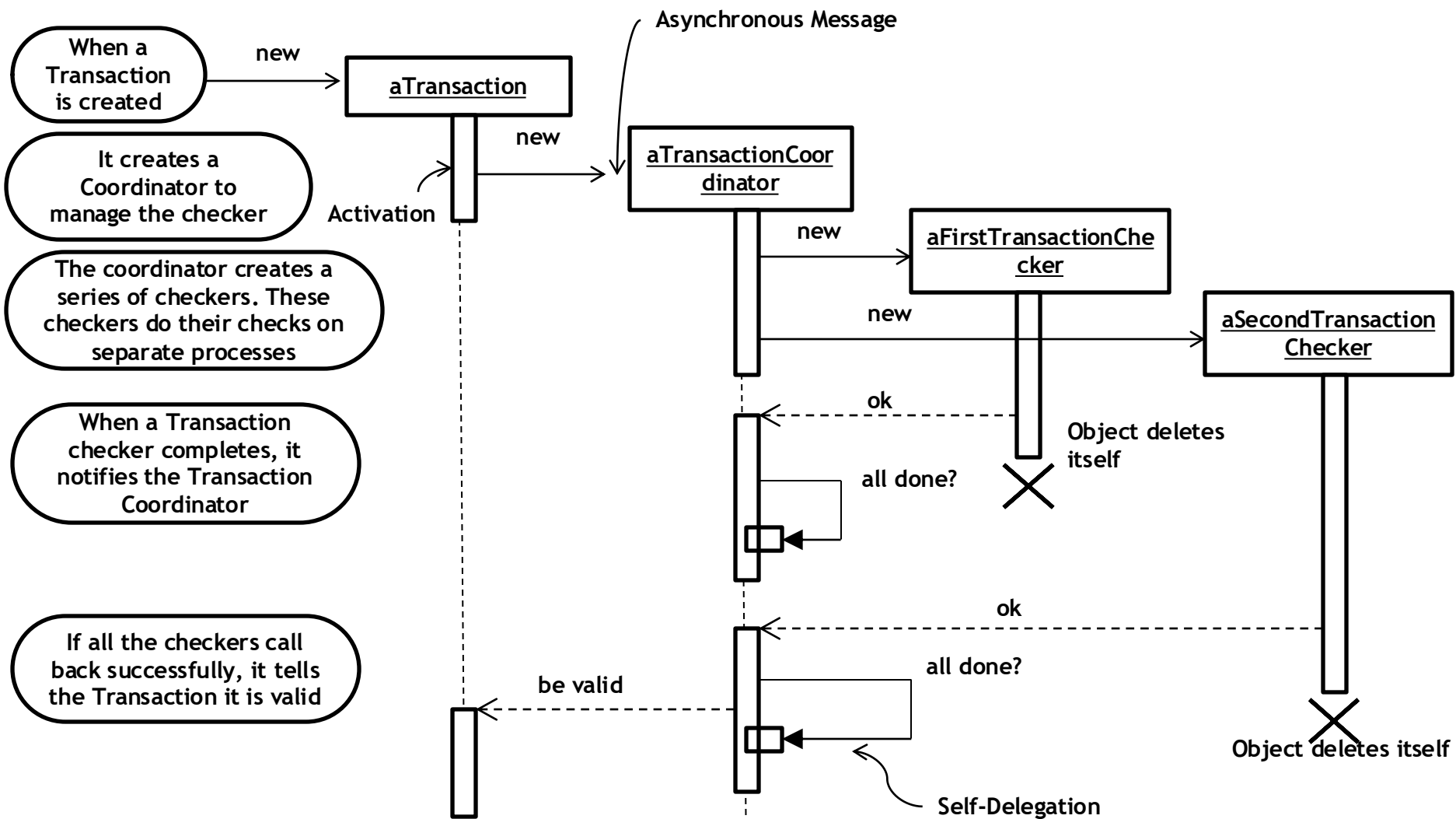


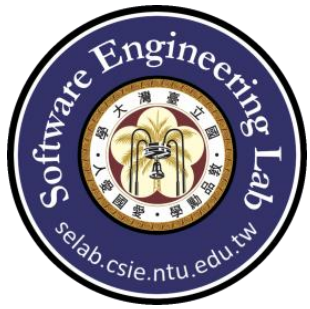
# Sequence Diagram Example B<sub>1</sub>

- ❑ When a Transaction is created, it creates a Transaction Coordinator to coordinate the checking of the Transaction.
- ❑ This coordinator creates a sequence of Transaction Checkers objects, each of which is responsible for a particular check. Each checker is called asynchronously and proceeds in parallel.
- ❑ When a Transaction Checked completes, it notifies the Transaction Coordinator. The coordinator looks to see if all the checkers called back. If not, the coordinator does nothing. If they have, and all of them are successful, then the coordinator notifies the Transaction that all are valid.



# Sequence Diagram Example B2



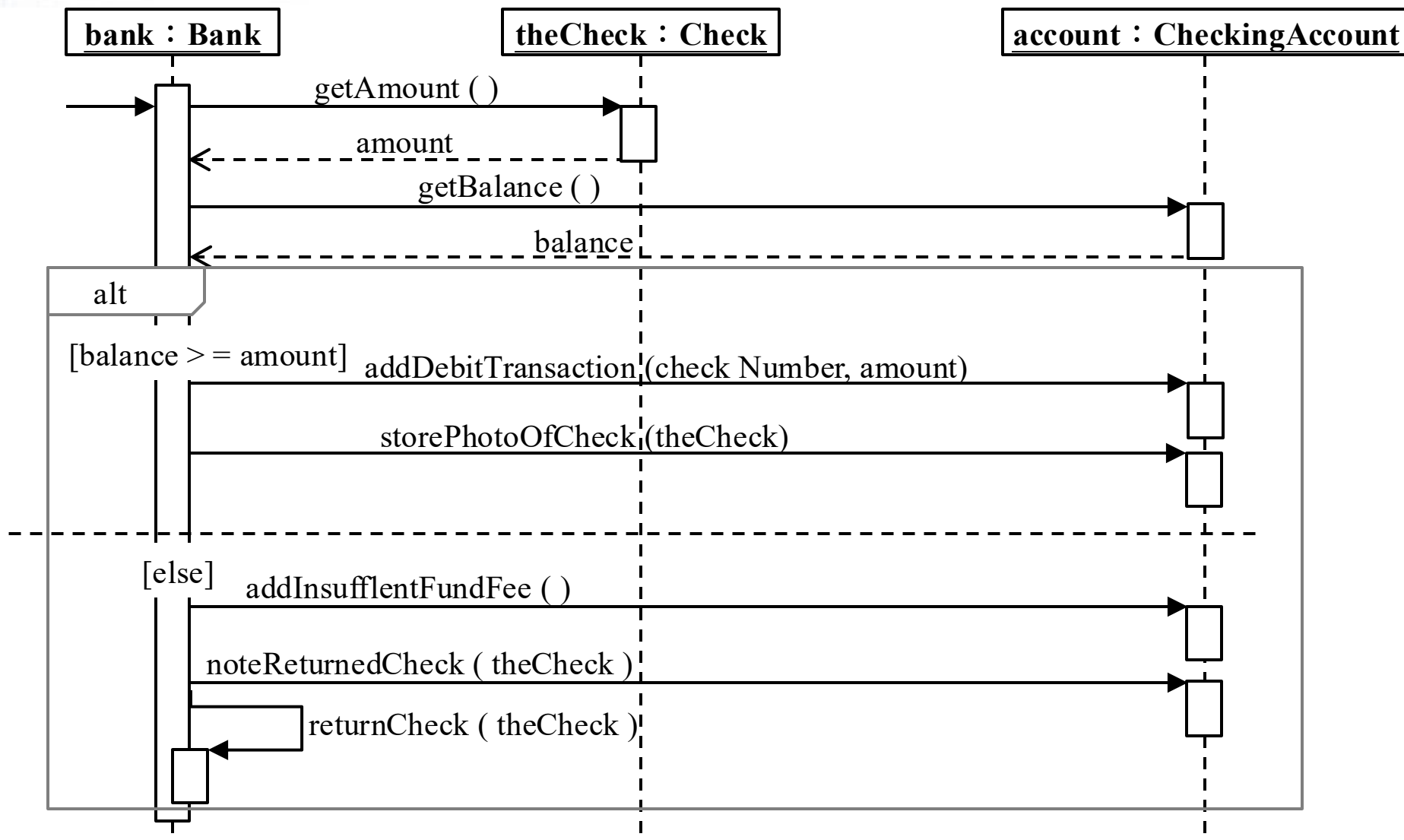


# Combined Fragment

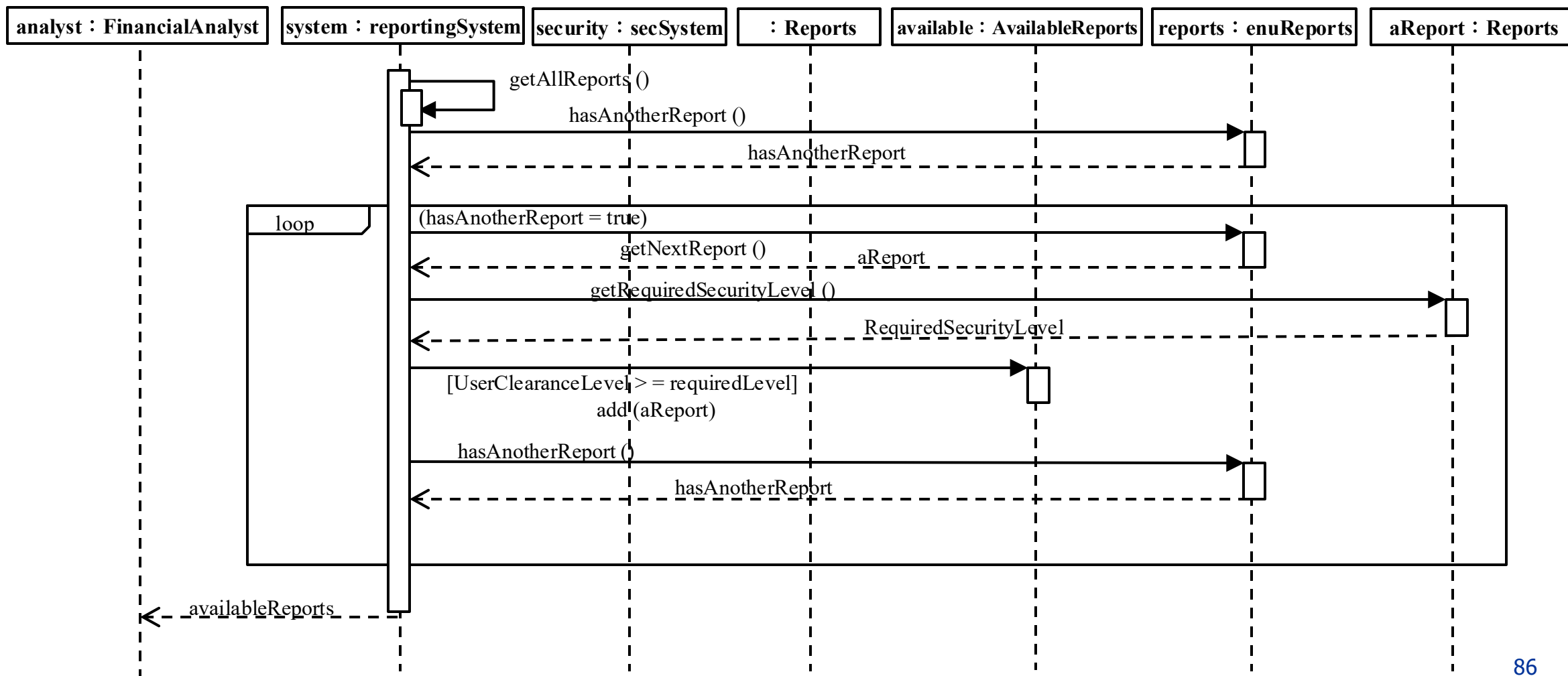
- ❑ A combined fragment is represented as a box that frames a section of interactions between objects in a sequence diagram.
- ❑ Three types of fragments:
  - Alternative fragment (if then else): fragment operator "alt".
  - Loop fragment: fragment operator "loop".
  - Option fragment (if then): fragment operator "opt".
- ❑ Reference fragment: reuse existing sequence diagrams in a sequence diagram, two ways: **ref** fragment overlays on an activation, **gate** by passing messages to a ref fragment.



# Alternative Fragment



# Loop Fragment



# Option and Reference Fragment

