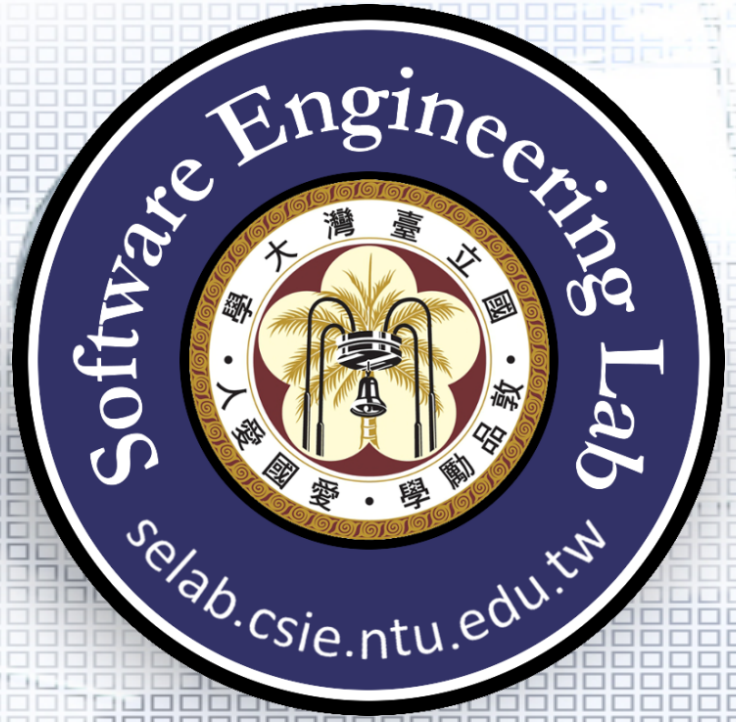


Object-Oriented Concepts

Prof. Jonathan Lee
CSIE Department
National Taiwan University

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 executable code in a more systematic manner. (Systematic)

Why Objects?

- Communications 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

Features of Object Orientation

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

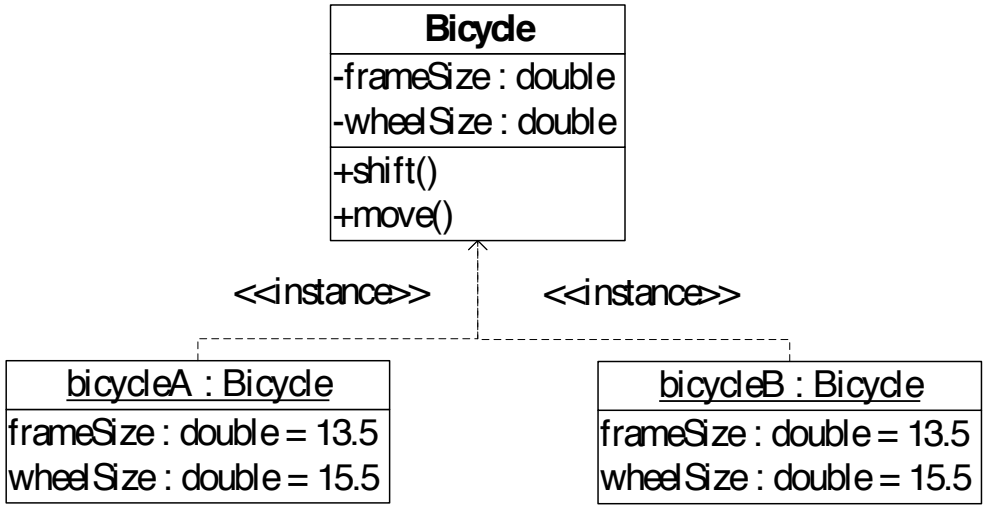
Class Diagrams

- 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

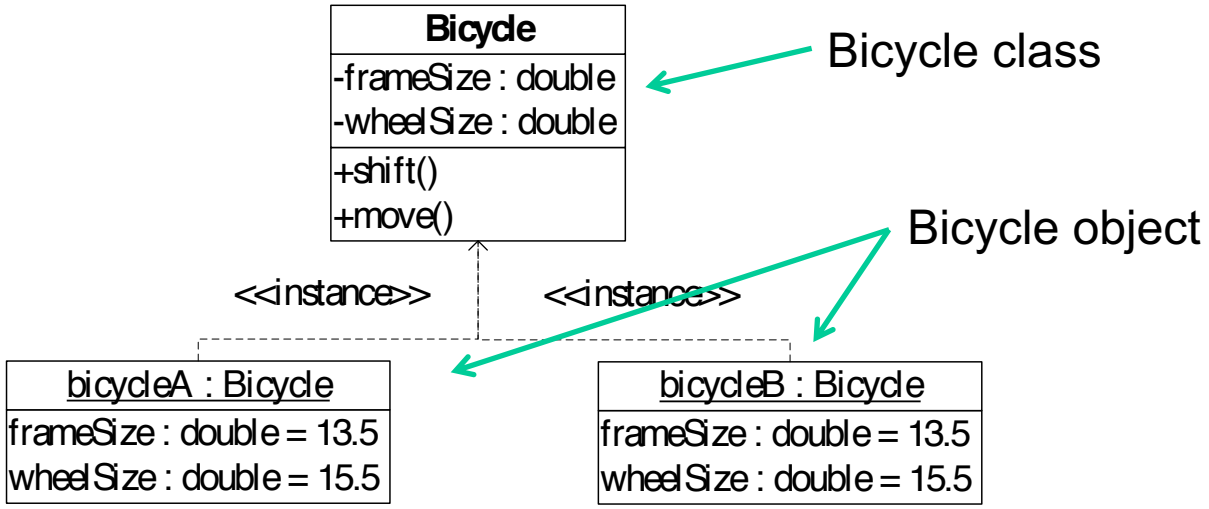
- Each object is a discrete and distinguishable entity.
- Each real-world object is unique due to its existence.
- Each object has its own inherent identity, therefore, two objects are distinct even if all their attribute values are identical.

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



Classification

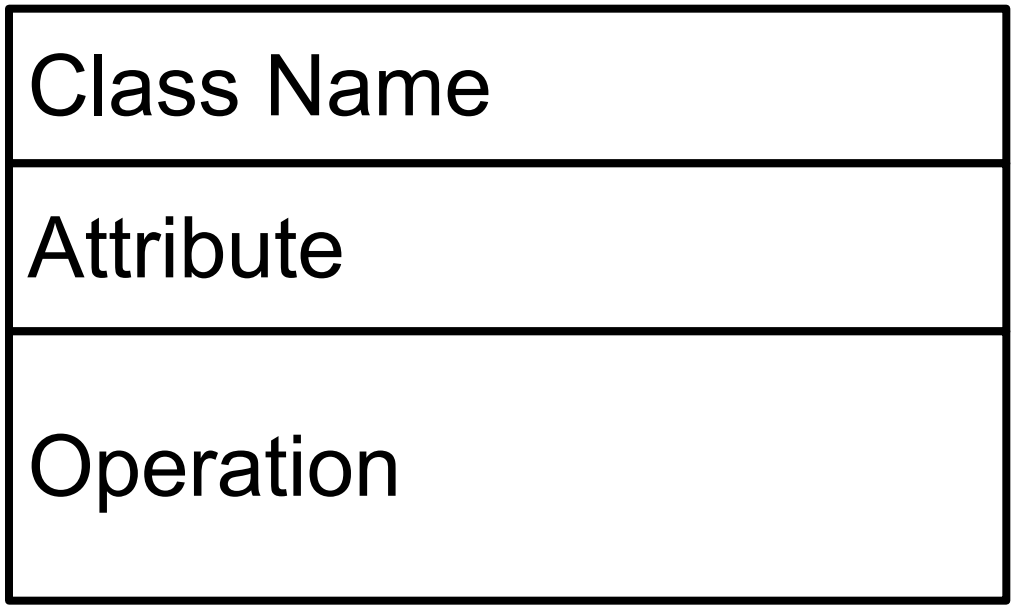
- Classification: (Class & Object)
  - Objects with the same attributes and operations are grouped into a class (data abstraction)
  - Each object is said to be an instance of its class
  - e.g. Bicycle object -------> Bicycle class



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



Attribute

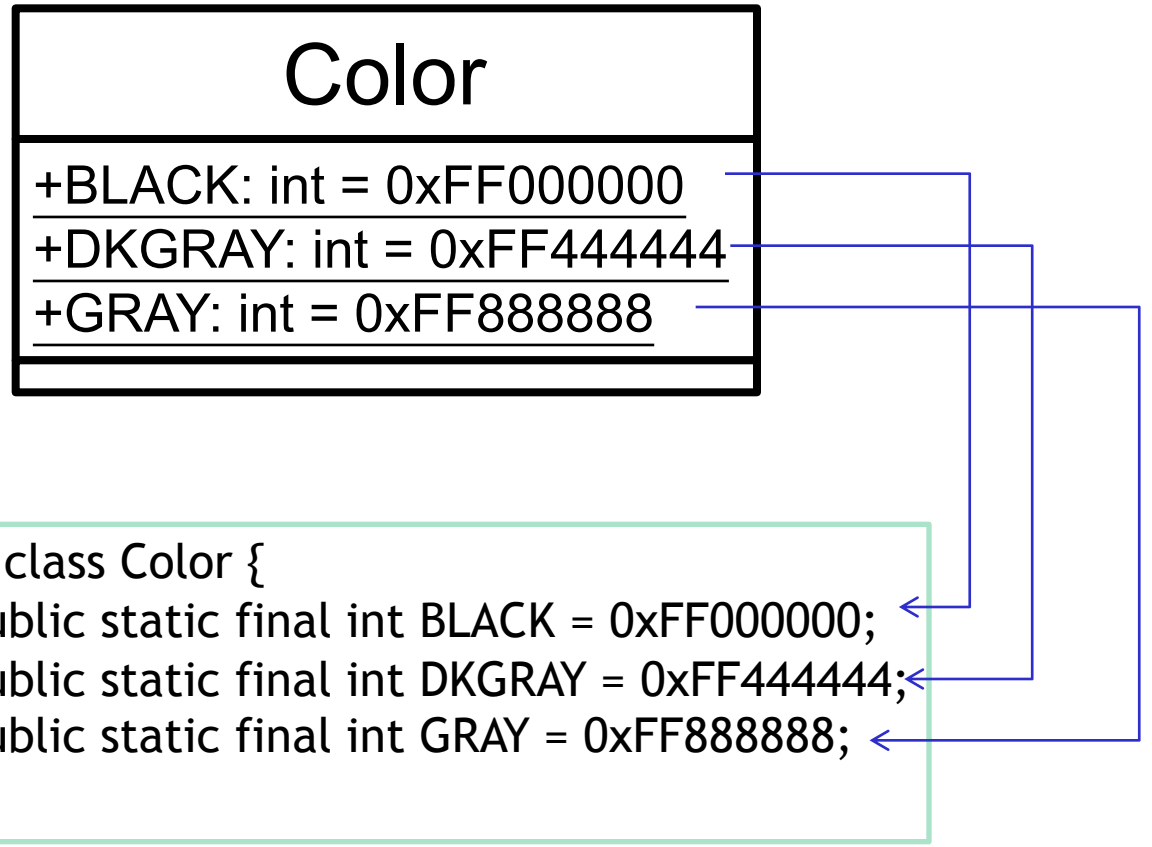
- Visibility / name: type multiplicity = default

  {+ | - | #} { class / 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



Operation

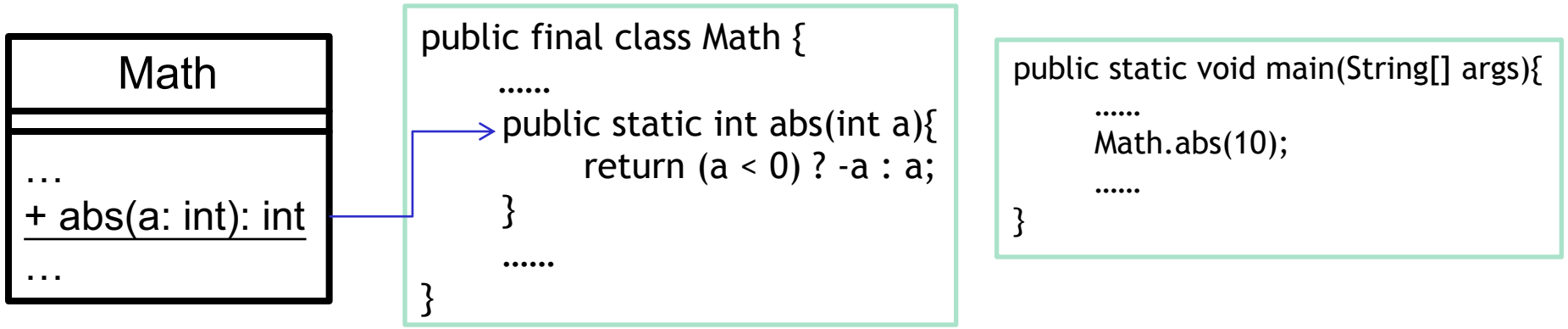
- An operation specifies how to invoke a particular behavior. A method is an **implementation** of an operation.
- Visibility name (parameters): return-type {property}

parameter_name: type[multiplicity] = default_value

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

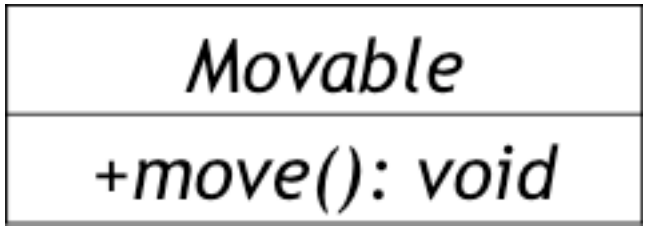
Static Operation

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



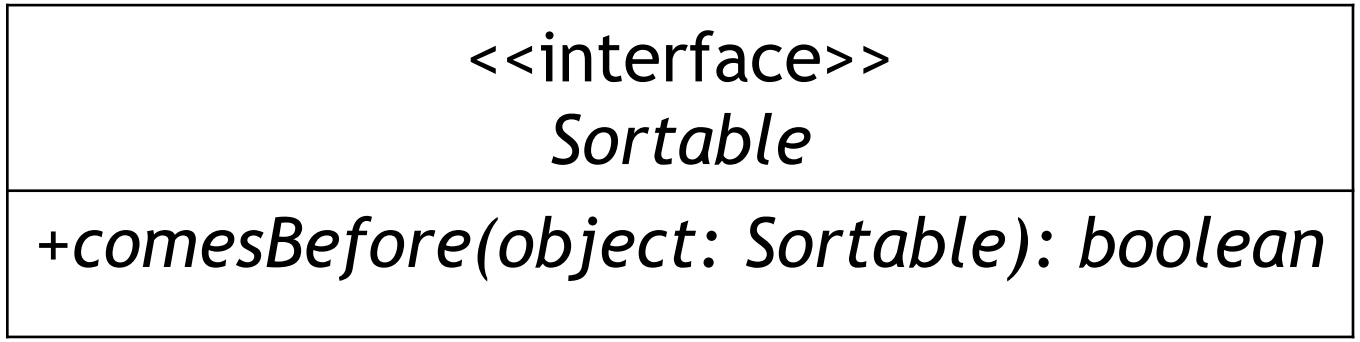
Abstract Class

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



Interface

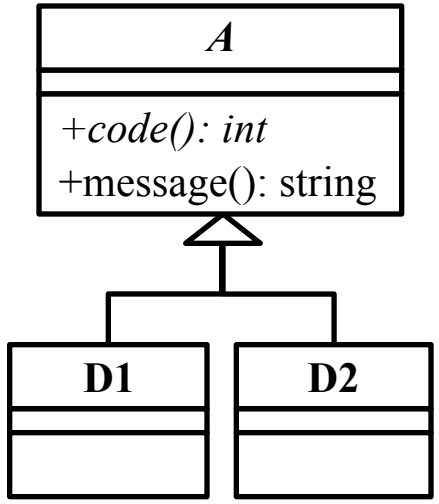
- An interface is a classifier that has declarations of properties and methods but no implementations.
  - e.g.



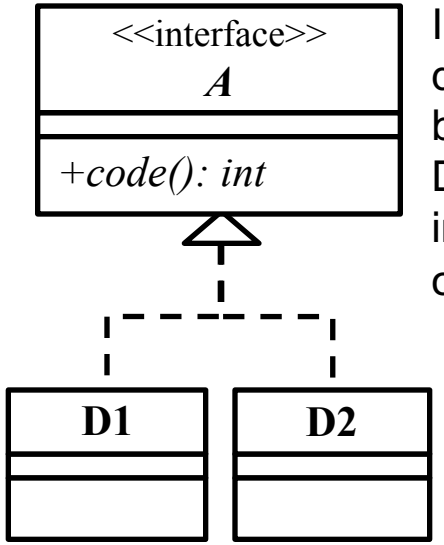
Inheritance

- 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 overide)
- 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.



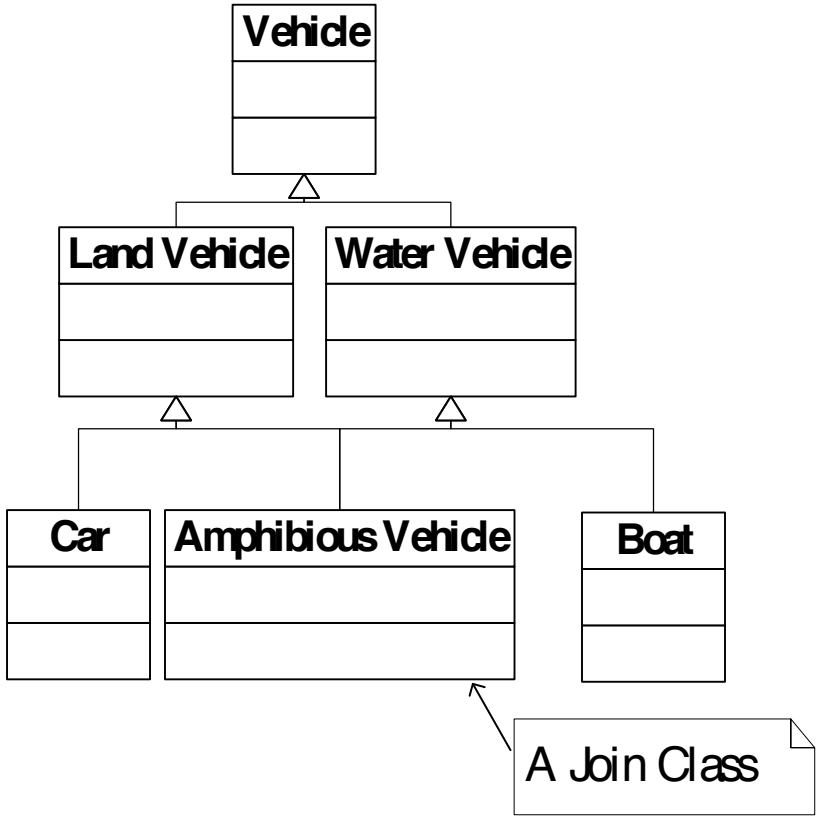
Interface, however, does not have the benefits of code reuse; D1 and D2 have to implement the abstract operation code().

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

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

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

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

Multiple Inheritance$_1$

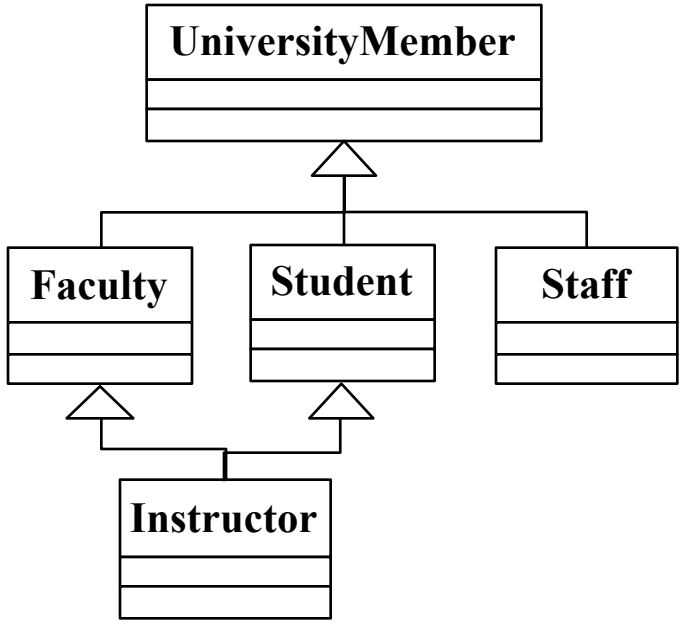
- a class has more than one superclass and inherits features from all parents (called a join class).
  - Generalization: conceptual level
  - Inheritance: implementation level. (mechanism)



In Java, multiple inheritance is not allowed.

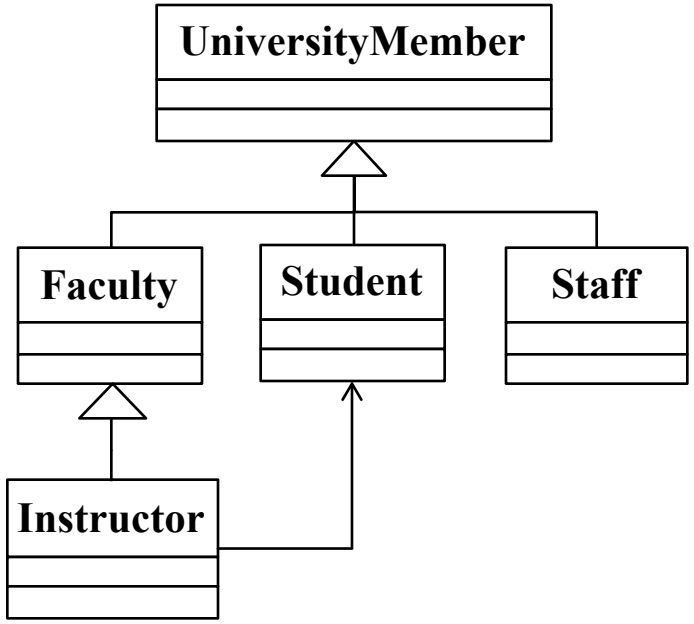
Multiple Inheritance$_2$

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



Multiple Inheritance$_3$

- 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.

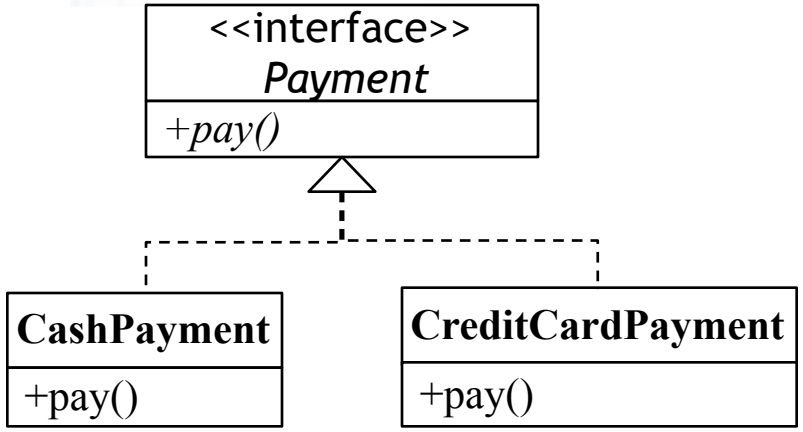
TimeOfDay
+setTime(time: char[]): void **+setTime(h: int, m: int , s: int): void**

```
class TimeOfDay {
  public void setTime(char[] time) {…};
  public void setTme(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




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

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

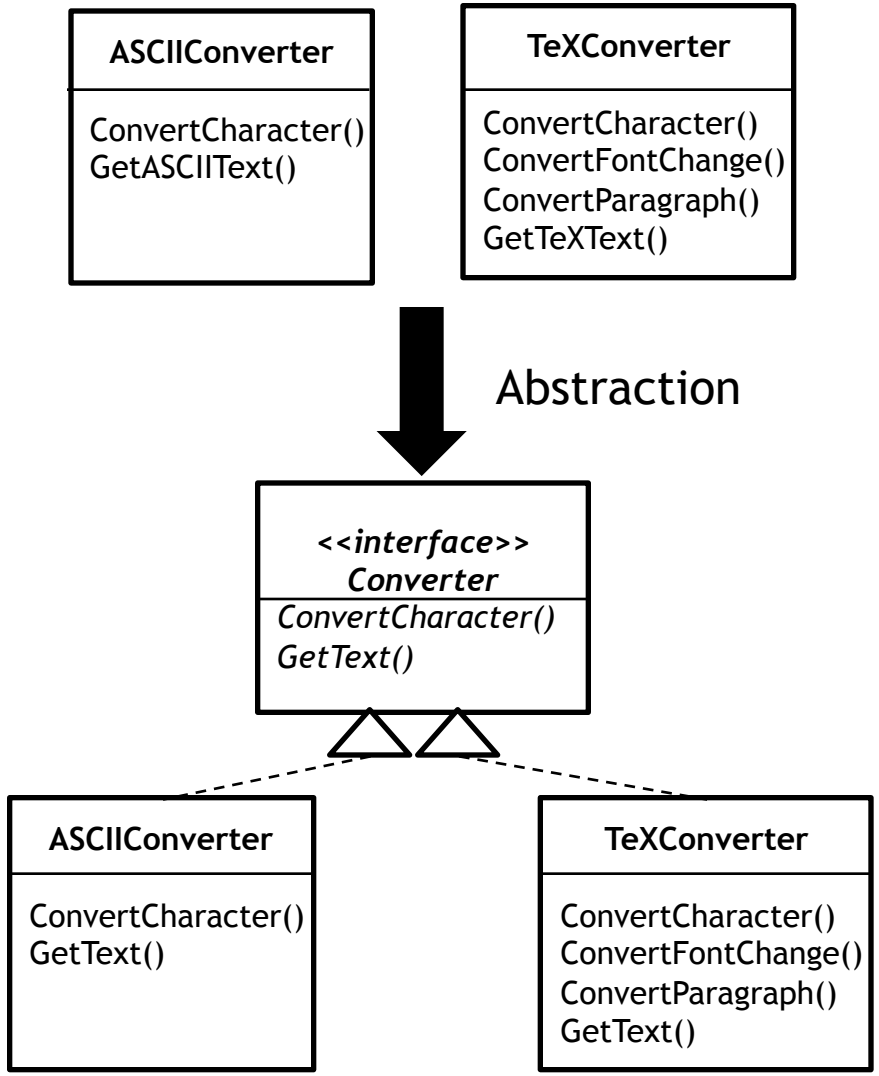
```
public class CashPayment implements Payment {
  public void pay() {
    System.out.println("Pay in cash");
  }
}

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
```

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



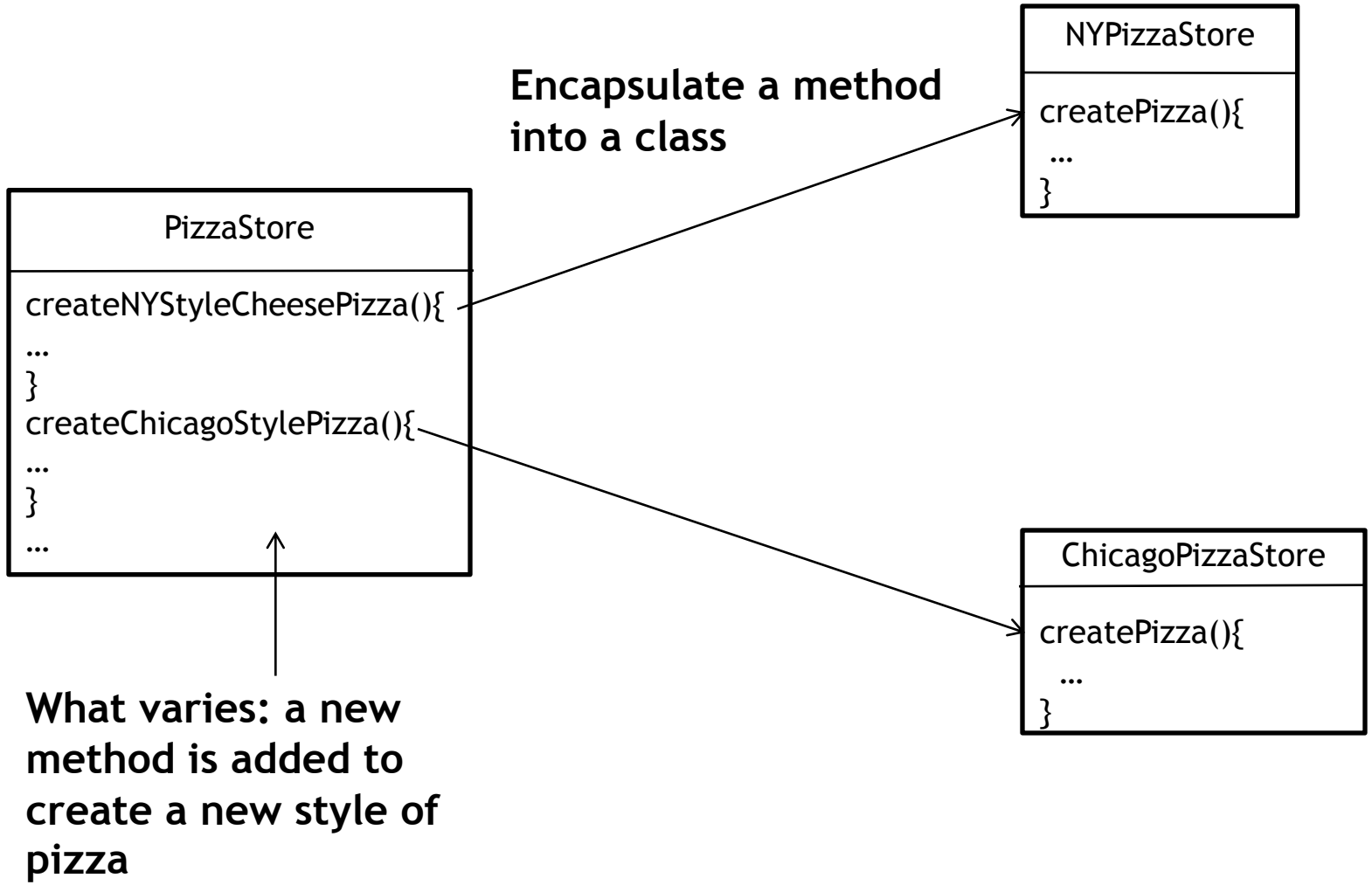
Encapsulation

- 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.

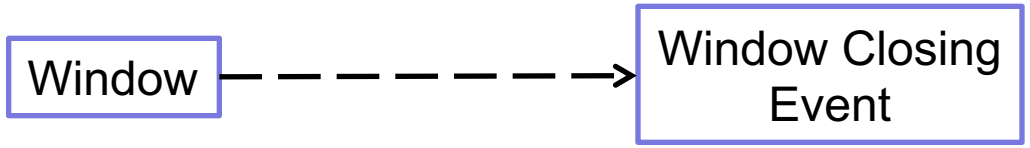
Only shift() & move() are *visible* to users. Further, the implementation of the two methods are *encapsulated*

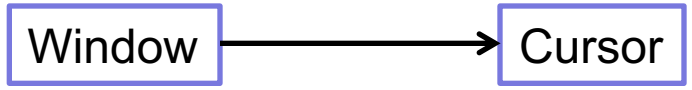
Bicycle
-frameSize : double -wheelSize : double
+shift() +move()

Encapsulate What *Varies*



Relationship$_1$

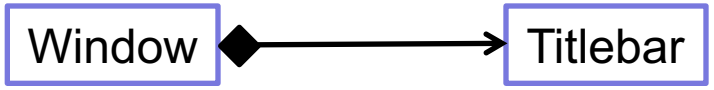
- Dependency is the weakest relationship between classes.
  - Uses-a
  - A transient relationship, that is, it doesn't retain a relationship for any real length of time
  - A dependent class **briefly** interacts with the target class
  - 

- Association is stronger than dependency.
  - One class retains a relationship to another class **over an extended period of time**
  - Has-a
  - 

Relationship$_2$

- 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
  -

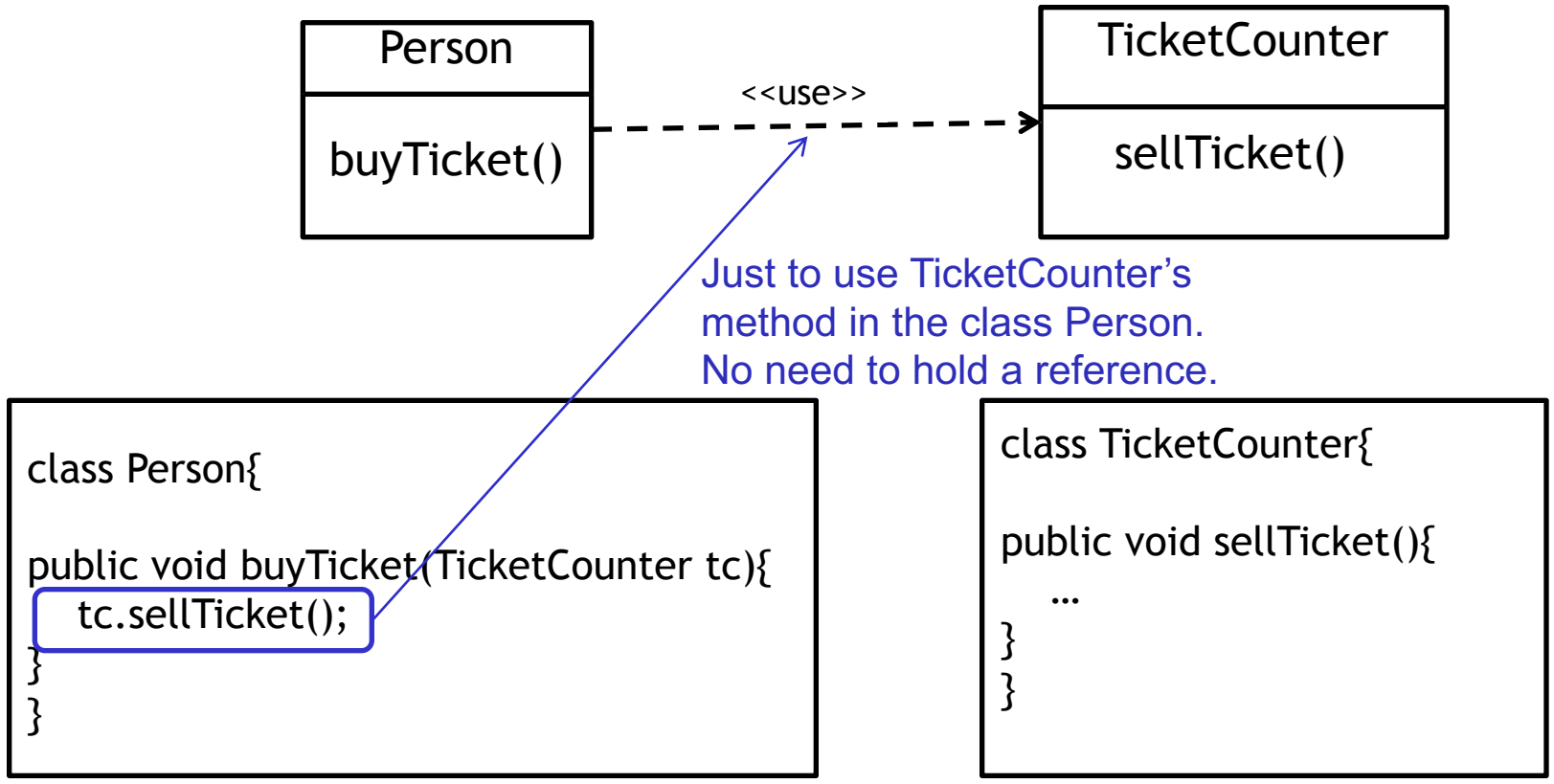


- Generalization is **the strongest relationship**
  - Is-a
  -

Dependency$_1$

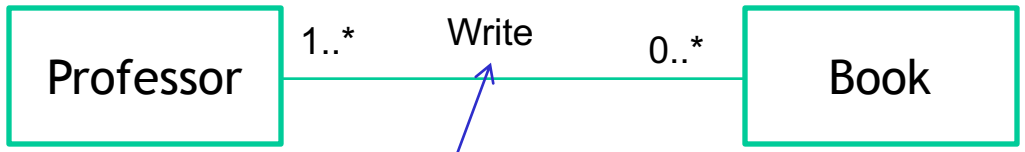
- A dependency states that the implementation or functioning of one or more elements requires the presence of one or more other elements.
- Types of stereotype that can be associated with dependency.
  - <<use>>
    - A usage dependency is one in which the client requires the presence of the supplier 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$_2$



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**.



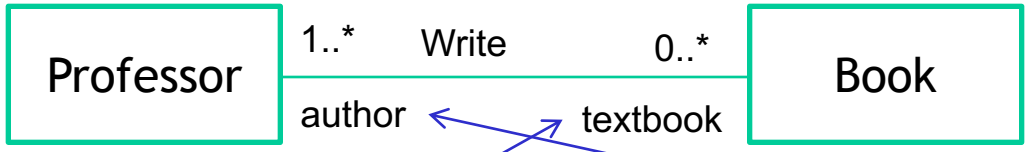
Association **Write** between **Professor** and **Book** with association ends **author** and **textbook**

```
class Professor {
    private Set<Book> textbook;
    …
    public write(Book book) {
        book.addAuthor(this);
        textbook.add(book);
    }
}
```

```
class Book {
    private Set<Professor> author;
    …
    public addAuthor(Professor p) {
        author.add(p);
    }
}
```

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. (e.g. author, textbook in the example below)
  - The role name is optional.




```
class Professor {
    private Set<Book> textbook;
    …
    public write() {
        book.addAuthor(this);
        textbook.add(book);
    }
}
```

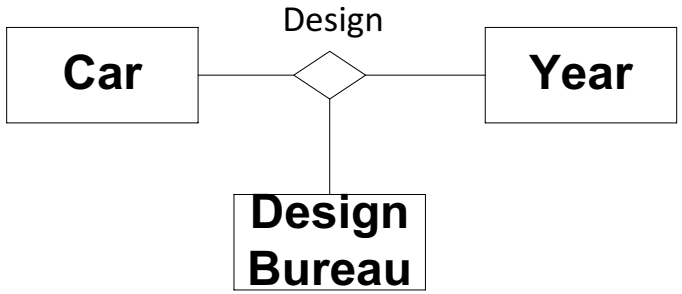
```
class Book {
    private Set<Professor> author;
    …
    public addAuthor(Professor p) {
        author.add(p);
    }
}
```

Arity

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

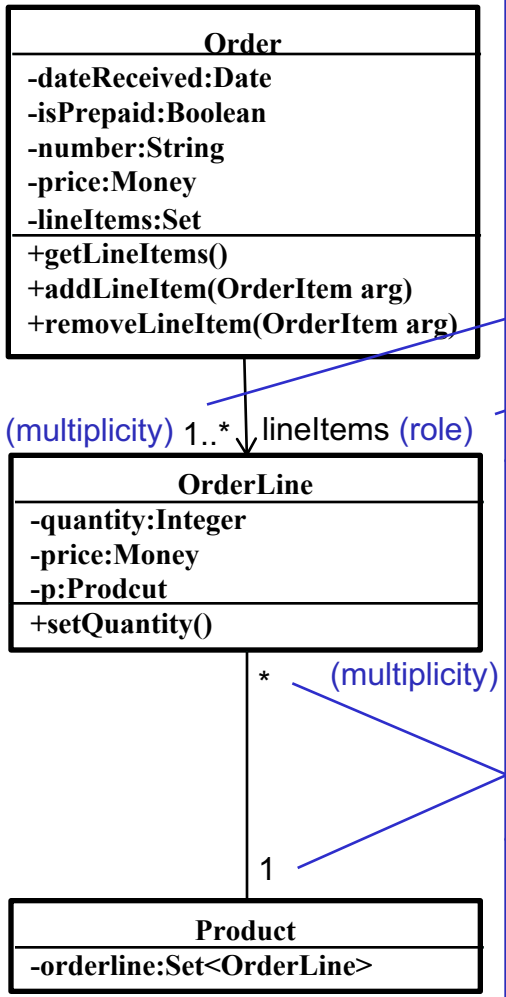
Study and Year classifiers are associated

  - N-ary Association



Ternary association Design relates three classifiers

Association Example$_1$

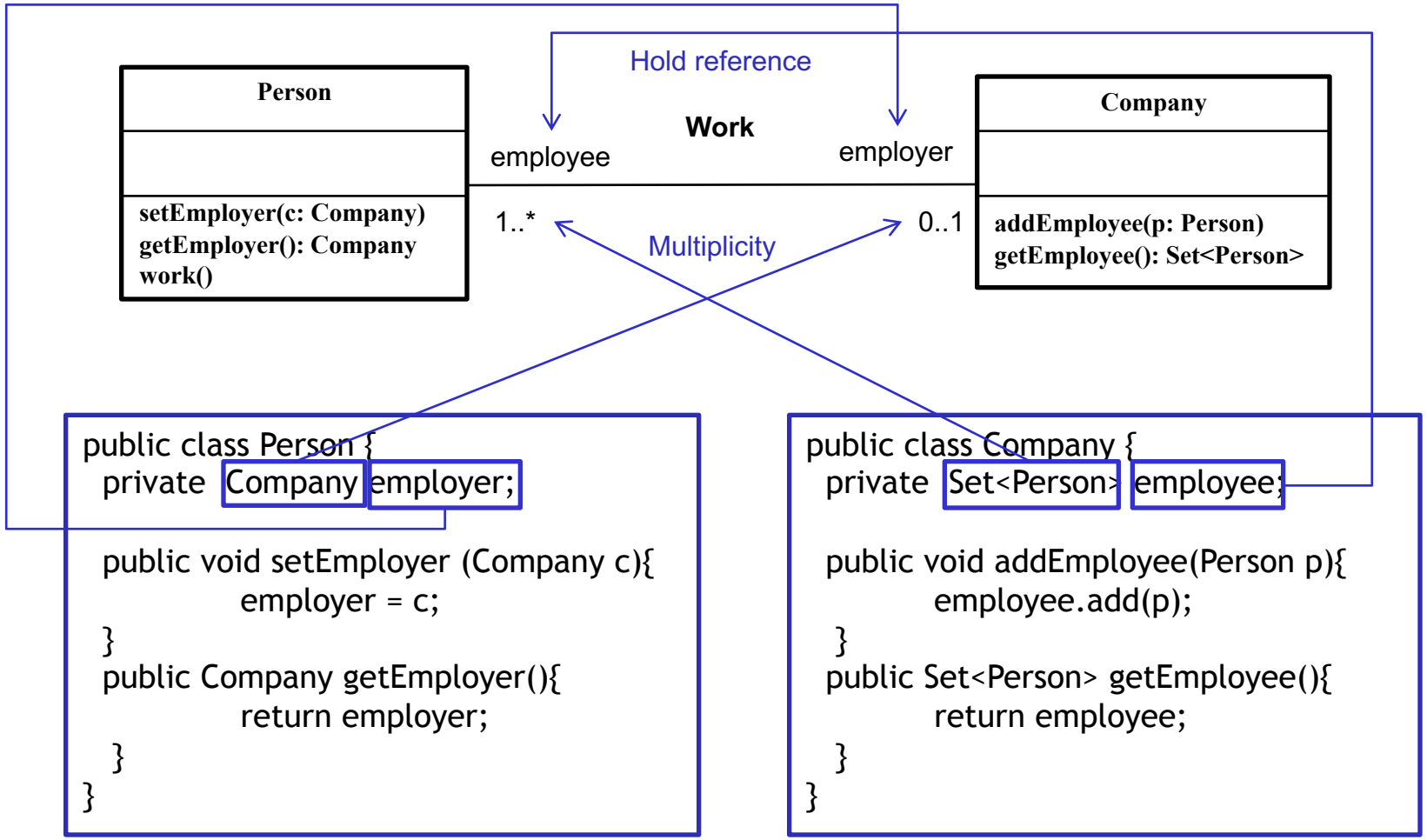



```
public class Order {
  private Date dateReceived;
  private Boolean isPrepaid;
  private String number;
  private Money price;
  private Set<OrderLine> lineItems = new HashSet();
  ...
  public Set getLineItems() {  return Collections.unmodifiableSet(lineItems);  }
  public void addLineItem (OrderItem 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$_2$

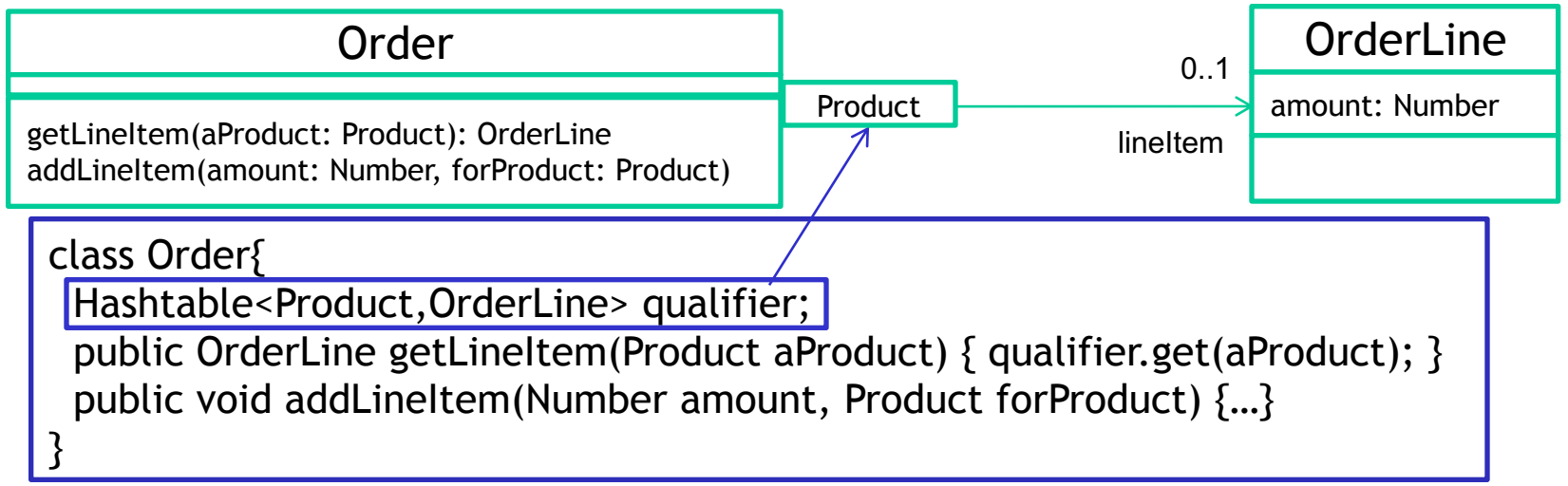


Association Qualifier

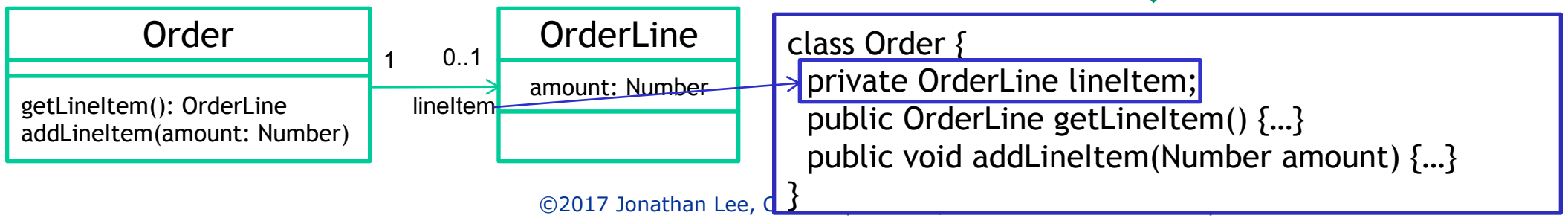
- Association Qualifiers to capture such information as relationships between elements are often **keyed or indexed** by some other value.

Association Qualifier Example

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



Without qualified association



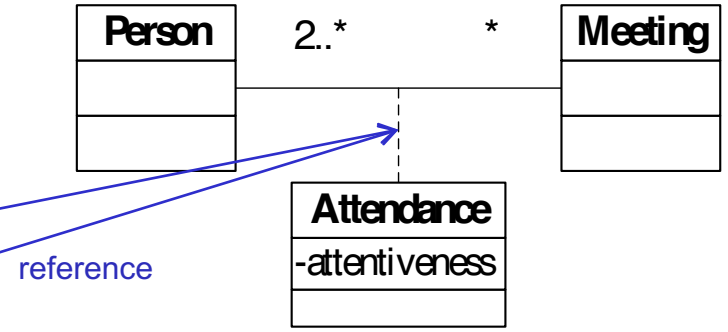
Association Class

- 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;
  …
}
```



Aggregation Type

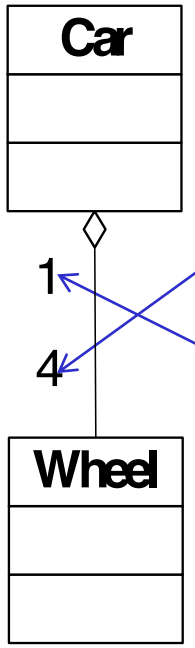
- An association may represent a composite aggregation (i.e., composition or a whole/part relationship).
  - Composite aggregation is a strong form of aggregation that requires **a part instance be included in at most one composite** at a time. (Composition)
  - If a composite is deleted, all of its 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

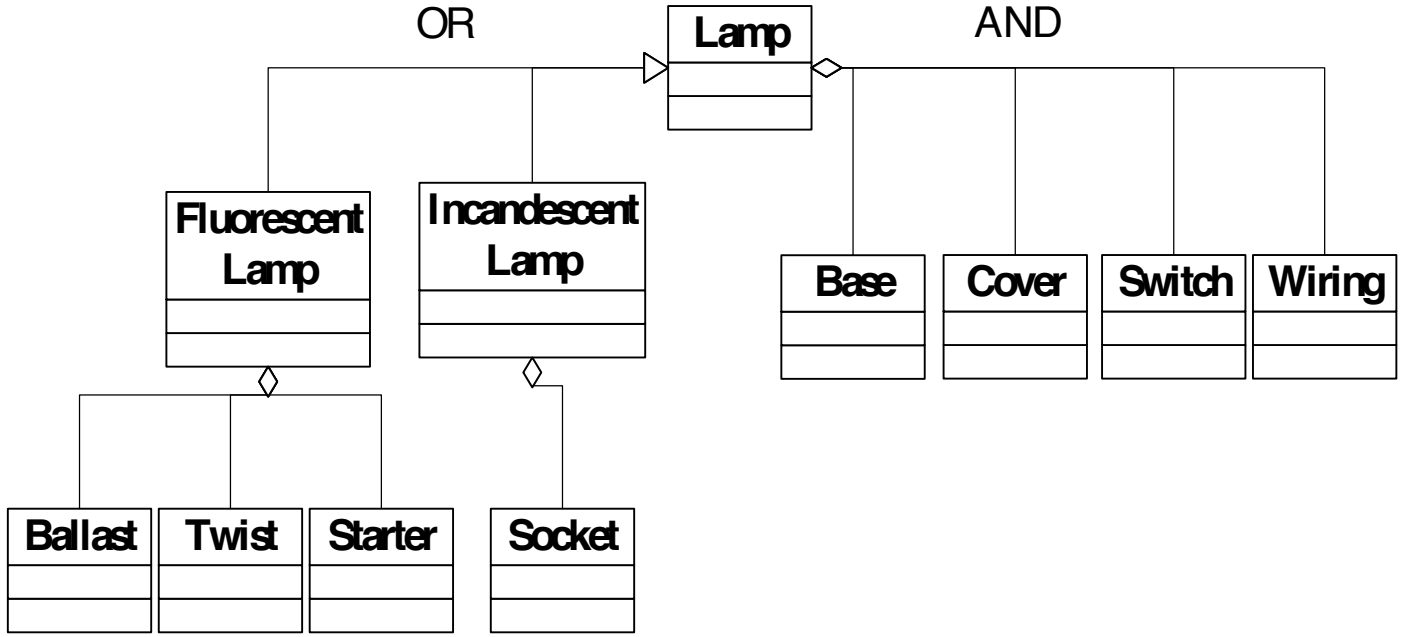



```
public class Car {
  private Wheel wheel1, wheel2, wheel3, wheel4;
  …
}
```

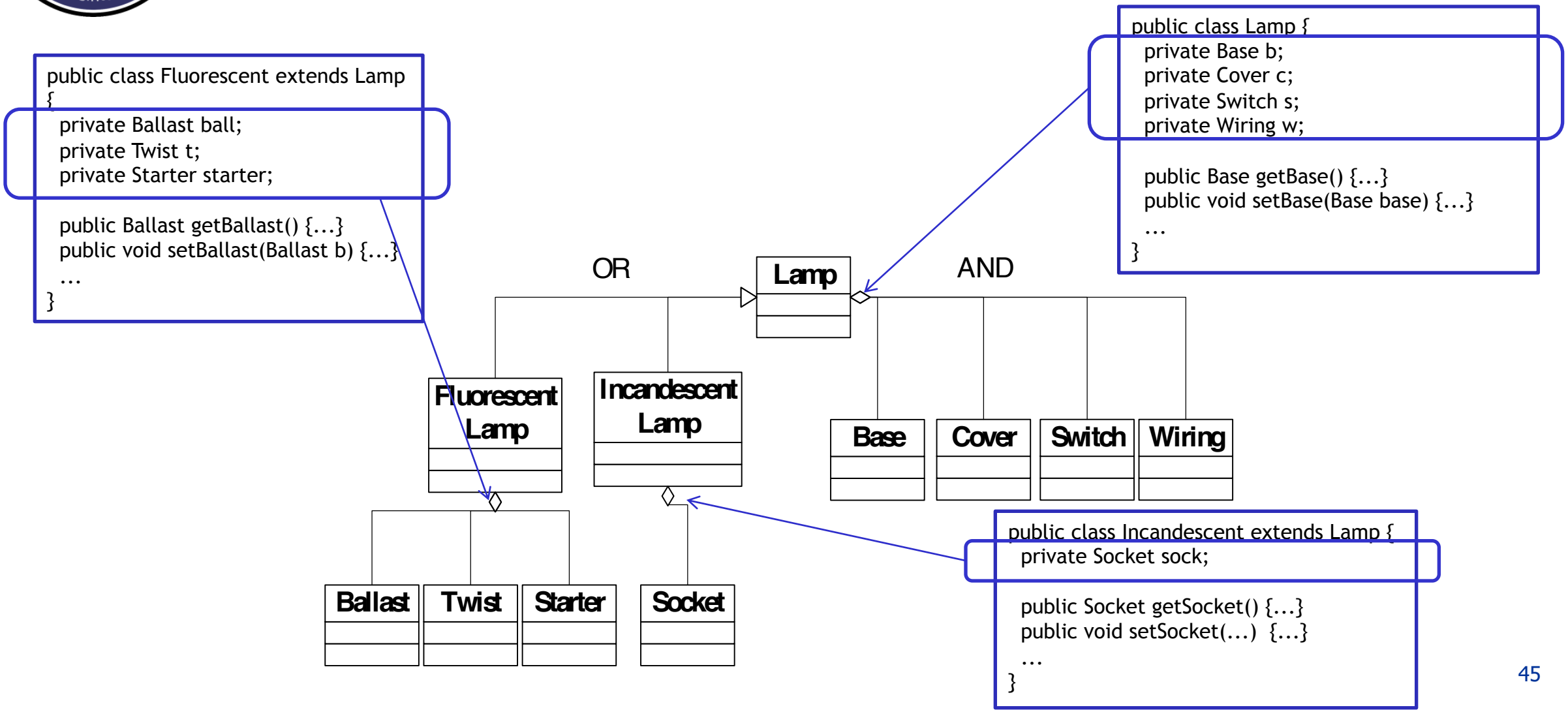
```
public class Wheel {
  private Car car;
  …
}
```

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



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.

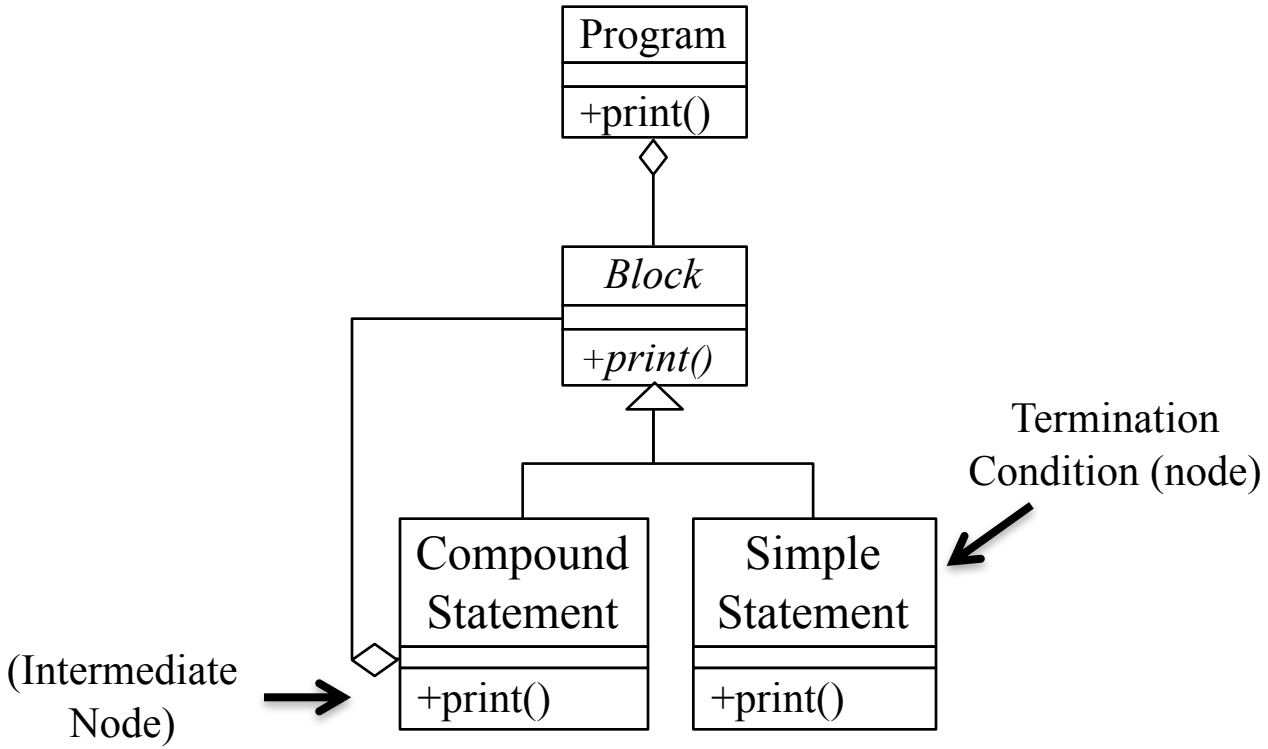
Delegate **add** for **push**
Delegate **last** and **remove** for **pop**

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;
  }
}
```

Recursive Aggregation

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



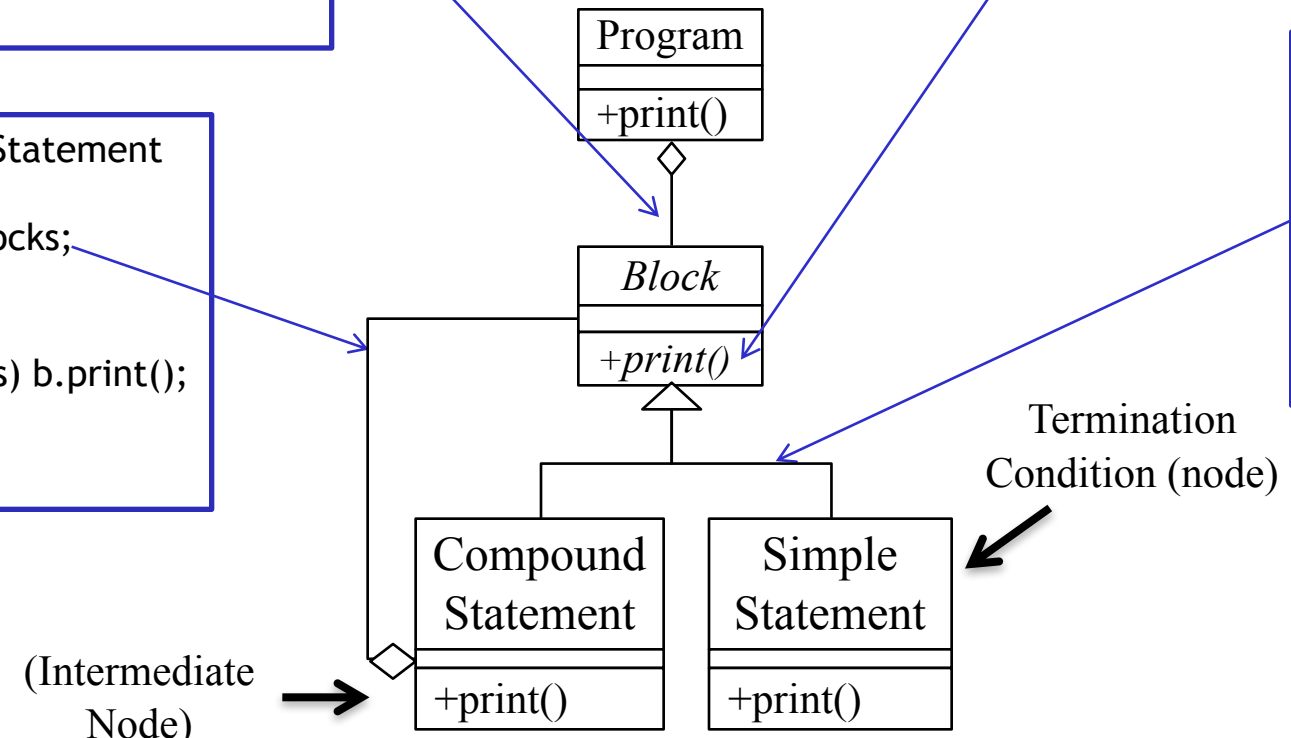
Recursive Aggregation: Java Code

```
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();
}
```

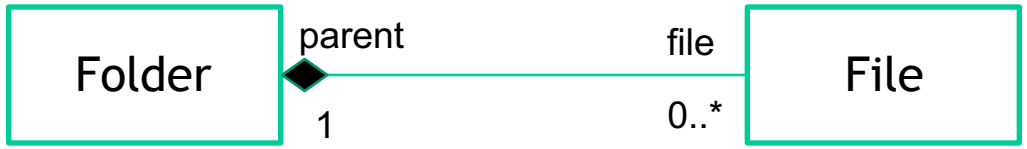
```
public class CompoundStatement
extends Block {
  private Set<Block> blocks;
  ...
  public void print() {
    for (Block b in blocks) b.print();
  }
}
```

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



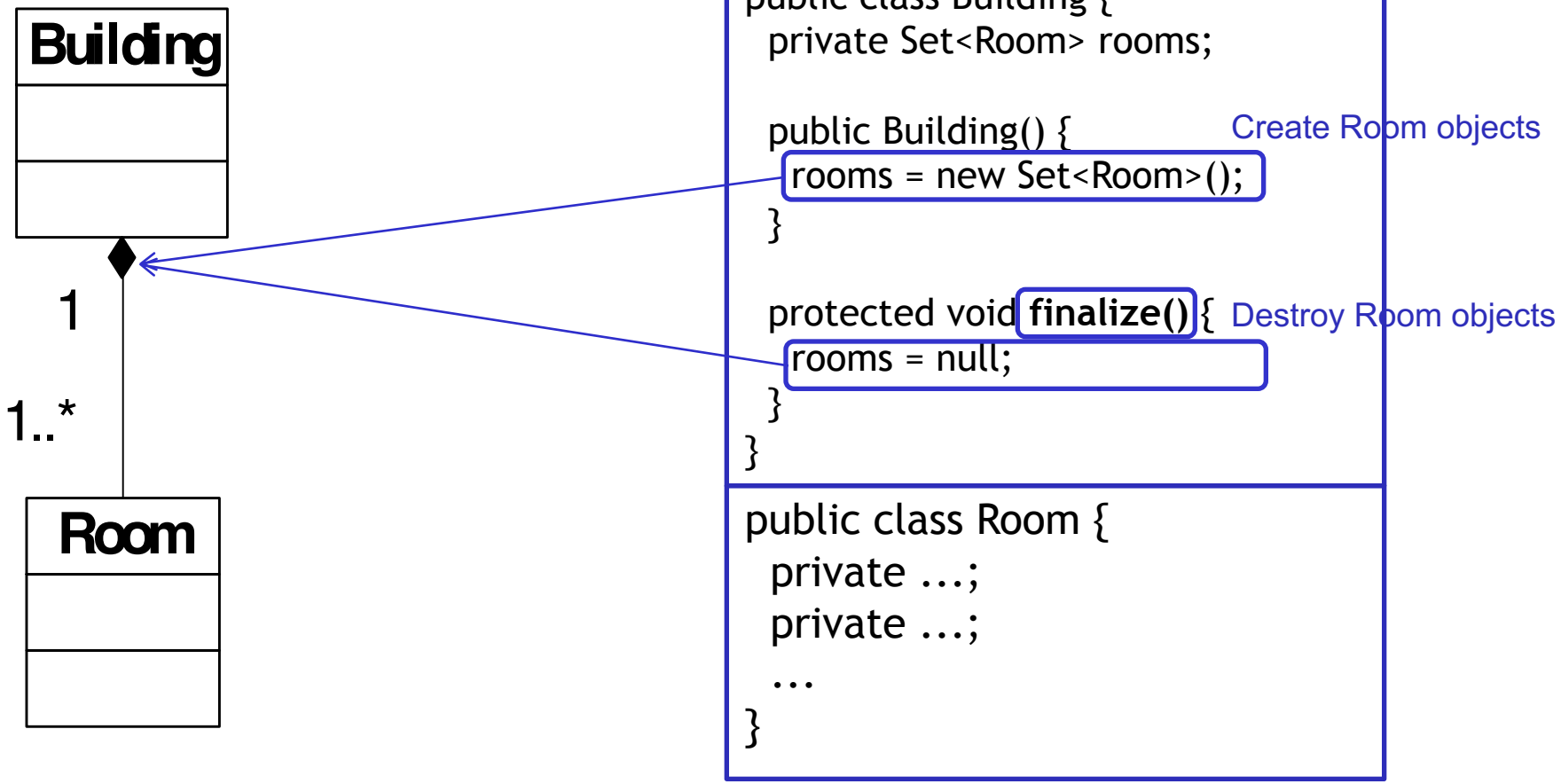
Composition

- 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*

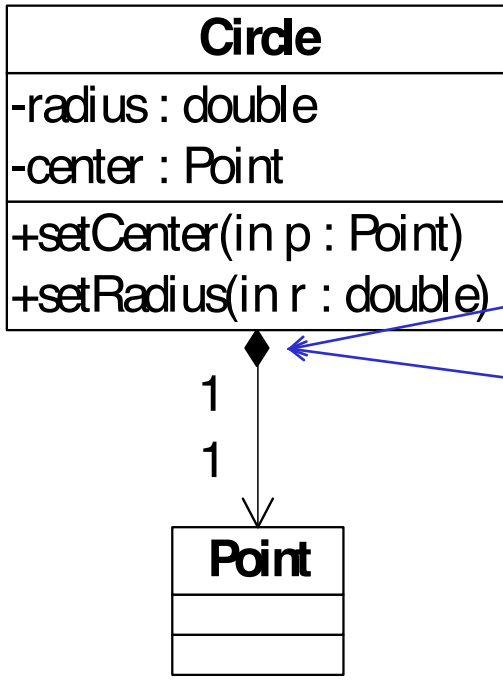


Folder could contain many **files**, while each **File** has exactly one **Folder** parent.
If **Folder** is deleted, all contained Files are deleted as well.

Composition Example$_1$



Composition Example$_2$




```
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;
  }
  …
}
```

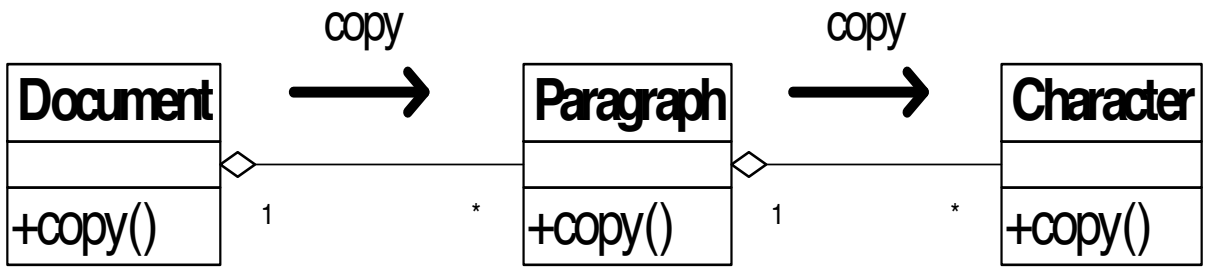
Create Point object

Destroy Point object

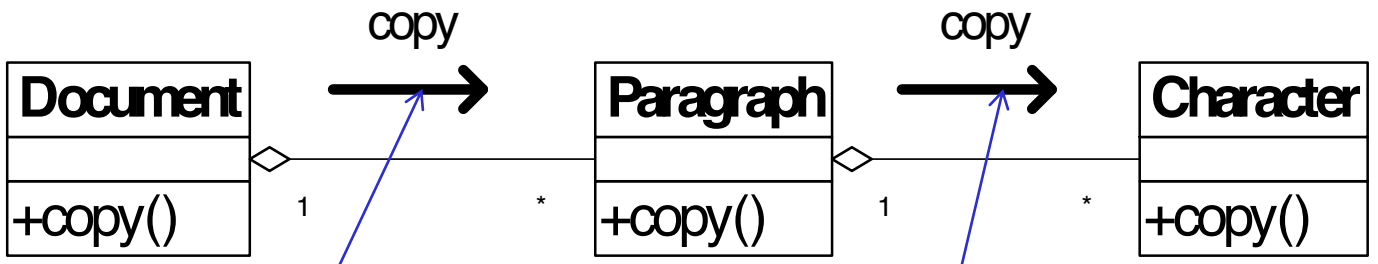
```
public class Point {
  private ...;
  private ...;
  ...
}
```

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;  }
}
```

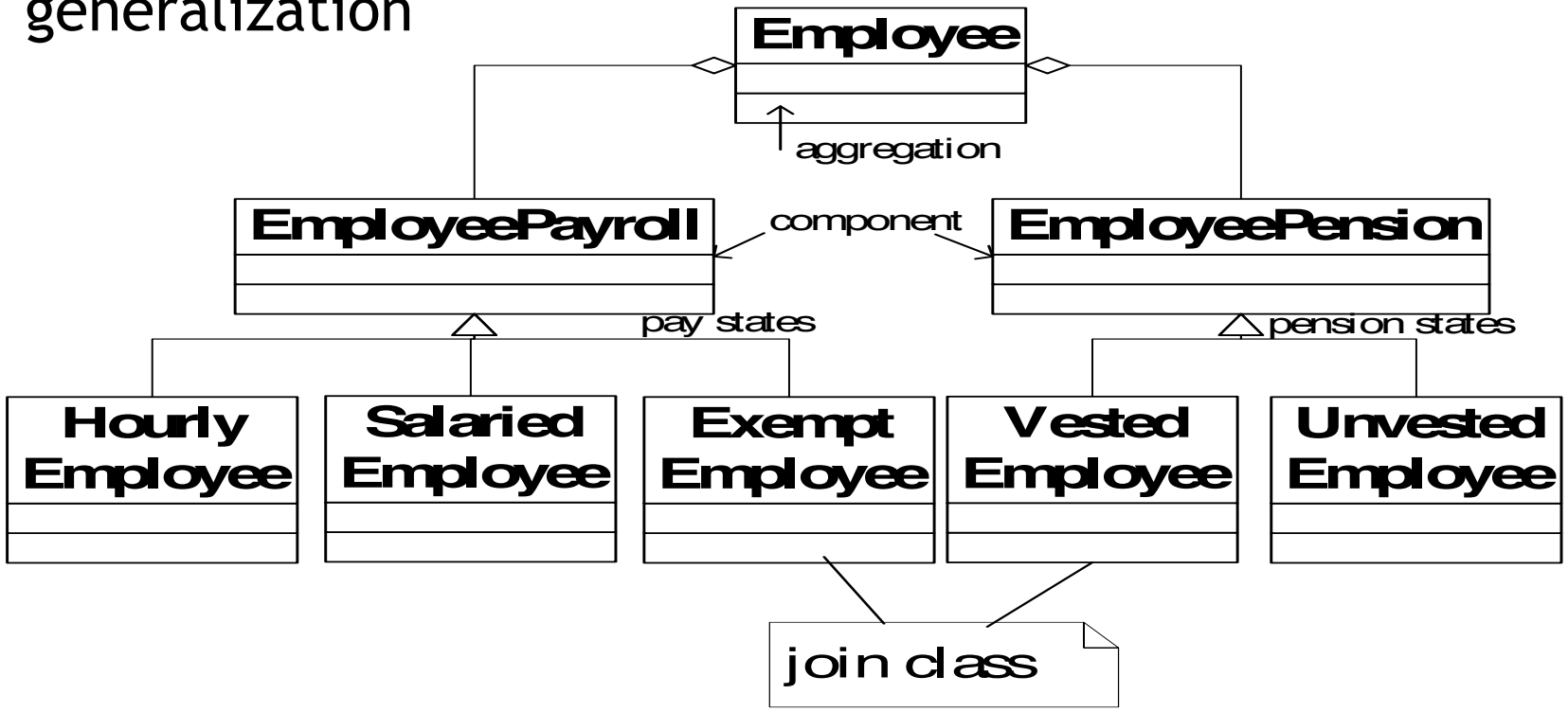
```
public class Paragraph {
  private Set<Character> character;
  public Paragraph copy() {
    Paragraph p = new Paragraph();
    for(Character c in character)
      p.add(c.copy());
  }
}
```

```
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$_1$

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




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

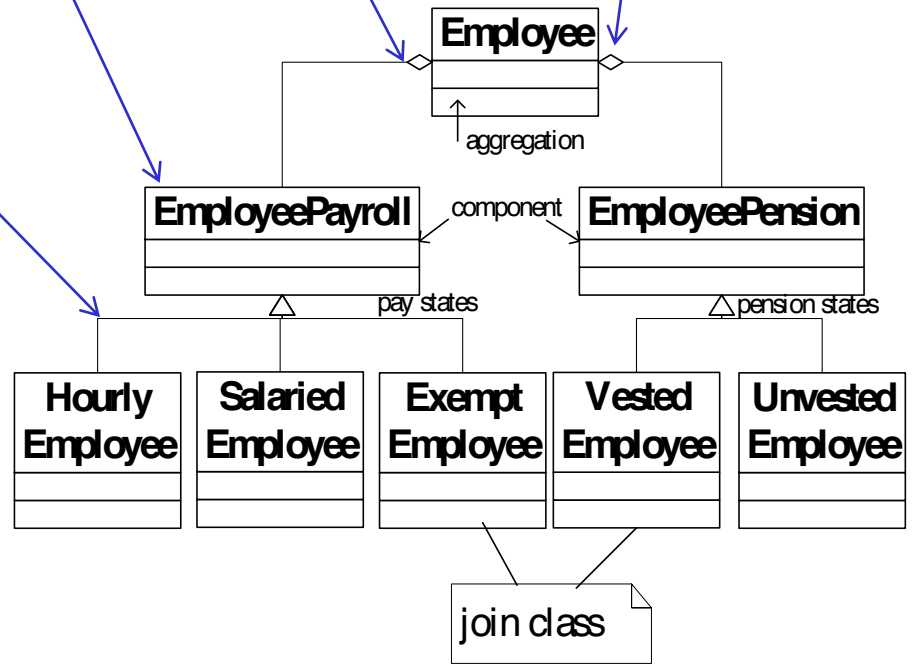
```
public class Employee {
  private EmployeePayroll ePay;
  private EmployeePension ePension;

  public void setPayroll(EmployeePayroll ep) {
    this.ePay = ep;
  }
  public void getPayroll() {
    ePay.getPayroll();
  }
  ...
}
```

```
public class HourlyEmployee
extends EmployeePayroll {

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

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

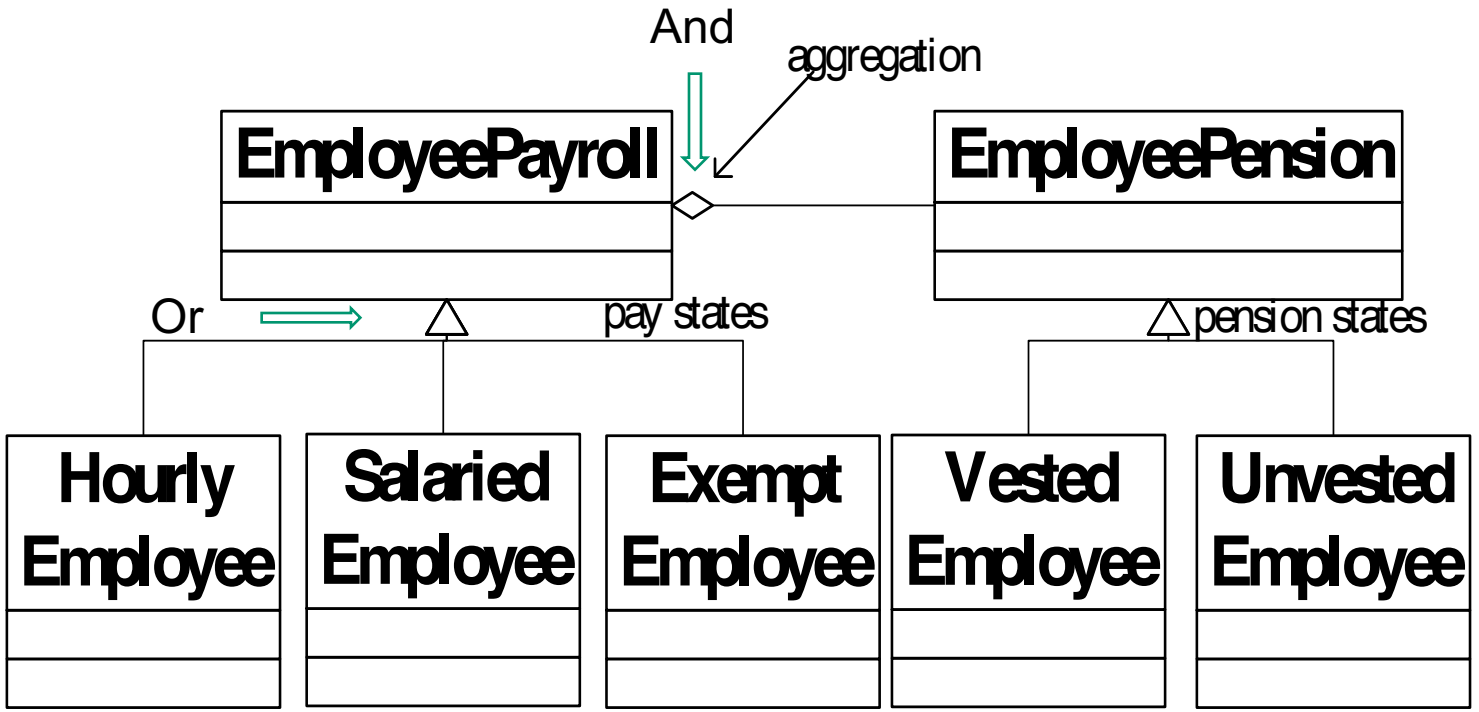
⇩ outputs

Hourly Employee Payroll



Resolution of Multiple Inheritance$_2$

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

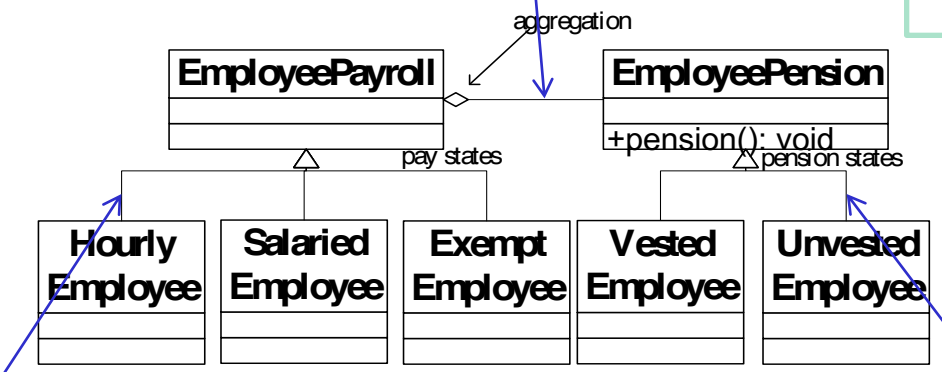



```
public class EmployeePayroll {
  private EmployeePension ePension;
  ...
  public void ...(...) {
    ePension.pension();
  }
}
```

delegate

through

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

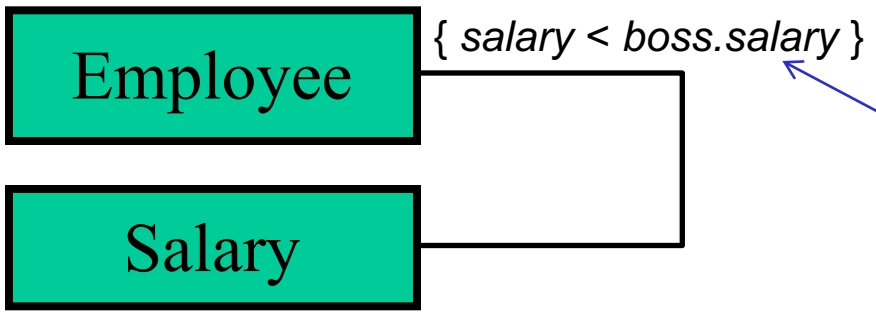



```
public class HourlyEmployee
extends EmployeePayroll {
  ...
}
```

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

Constraints$_1$

- 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.

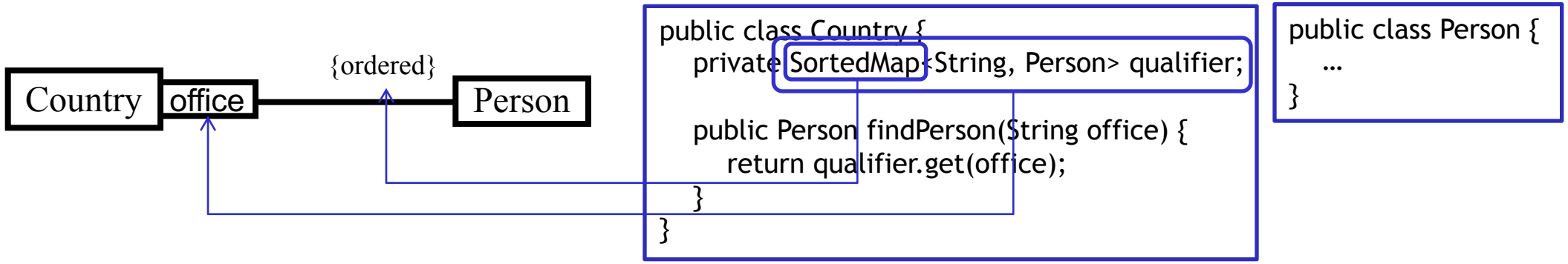



```
public class Employee {
    private Salary salary;
    private Boss boss;

    public void setSalary(Salary salary)
        throws SalaryExceededException {

        if (salary.getValue() >= boss.salary.getValue())
            throw new SalaryExceededException();
        this.salary = salary;
    }
}
```

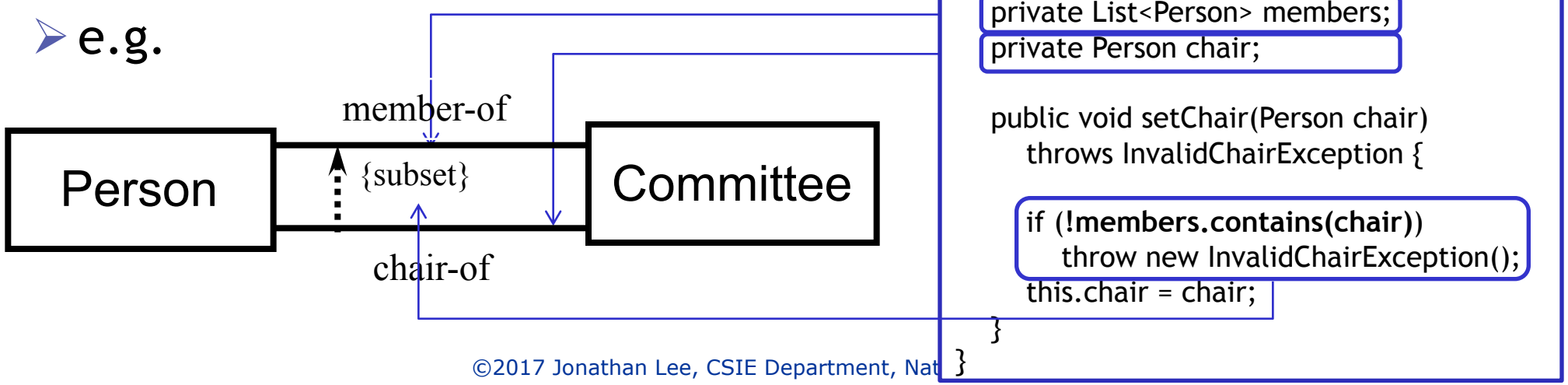
```
public class Salary {
    private double salary;

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

Constraints$_2$

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



- General constraints :
  - e.g.

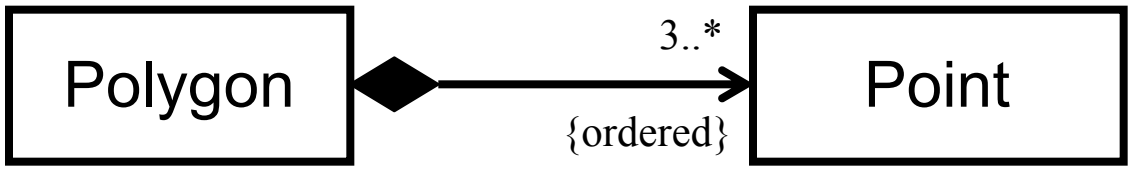


Delegation vs Composition

- 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

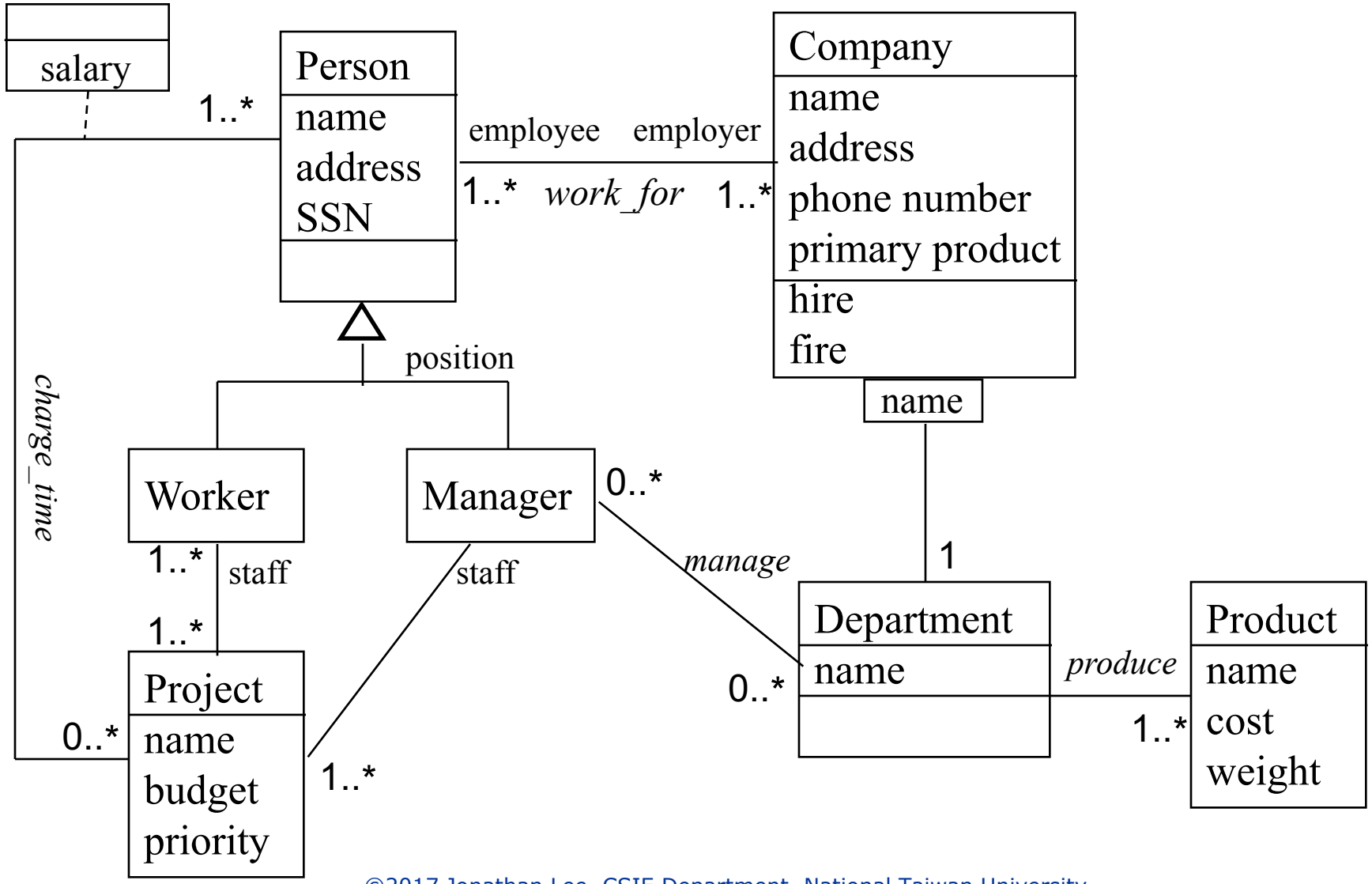
- A polygon is composed of an ordered set of points.

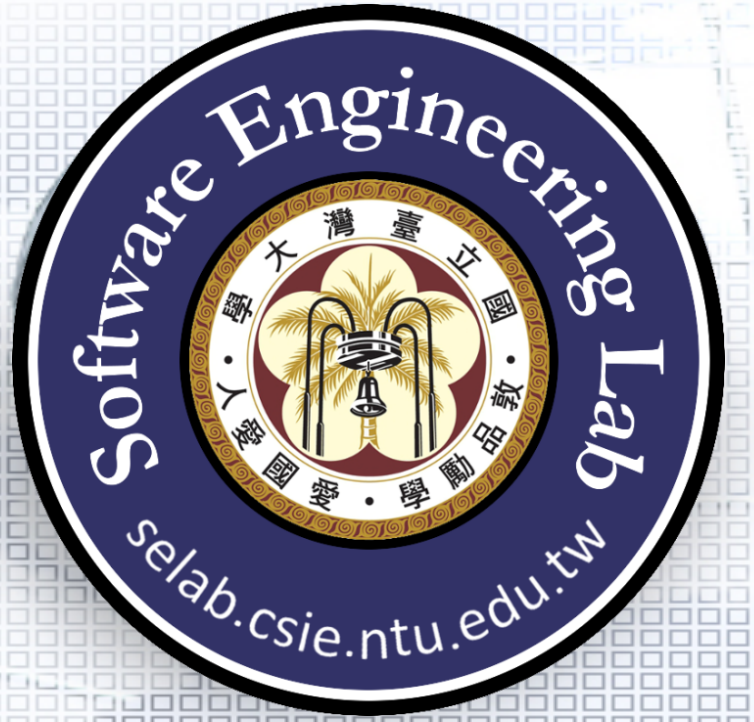


Problem Statement

- 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.

Class Diagram





A Hospital System

Prof. Jonathan Lee (李允中)

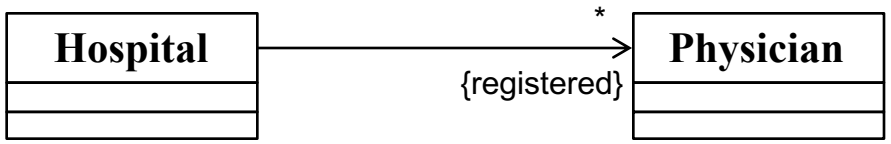
Department of Computer Science and Information Engineering
National Taiwan University

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.

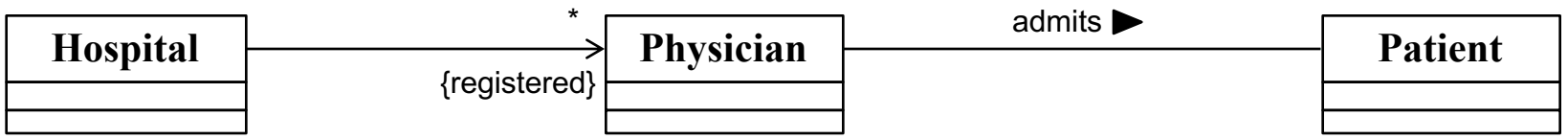
Requirements Statement$_1$

- A hospital has a large number of registered physicians.



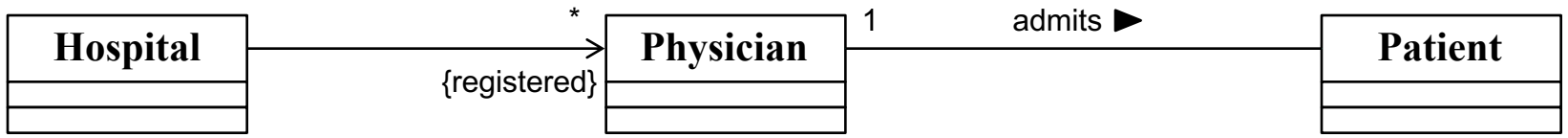
Requirements Statement$_2$

- Patients are admitted to the hospital by physicians.



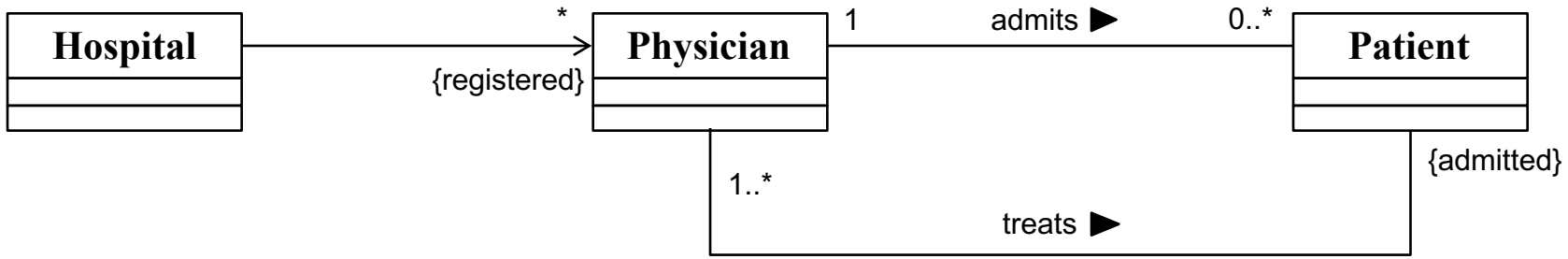
Requirements Statement$_3$

- Any patient who is admitted must have exactly one admitting physician.



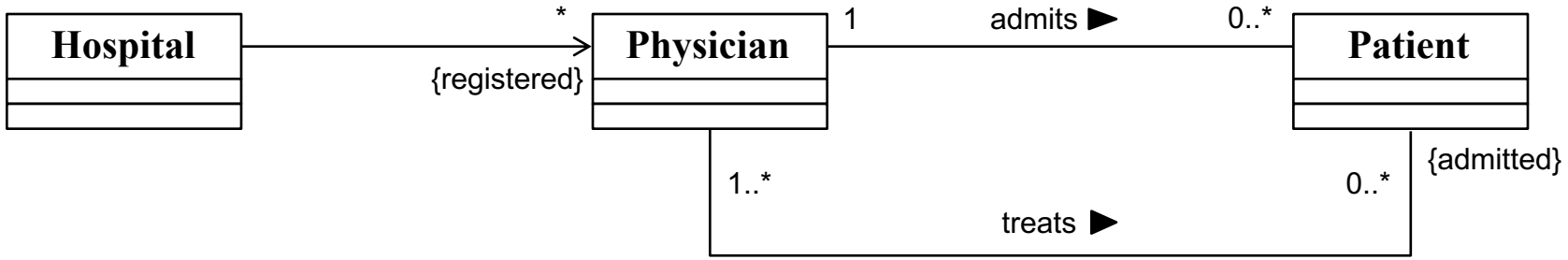
Requirements Statement$_4$

- A physician may optionally admit any number of patients. Once admitted, a given patient must be treated by at least one physician.



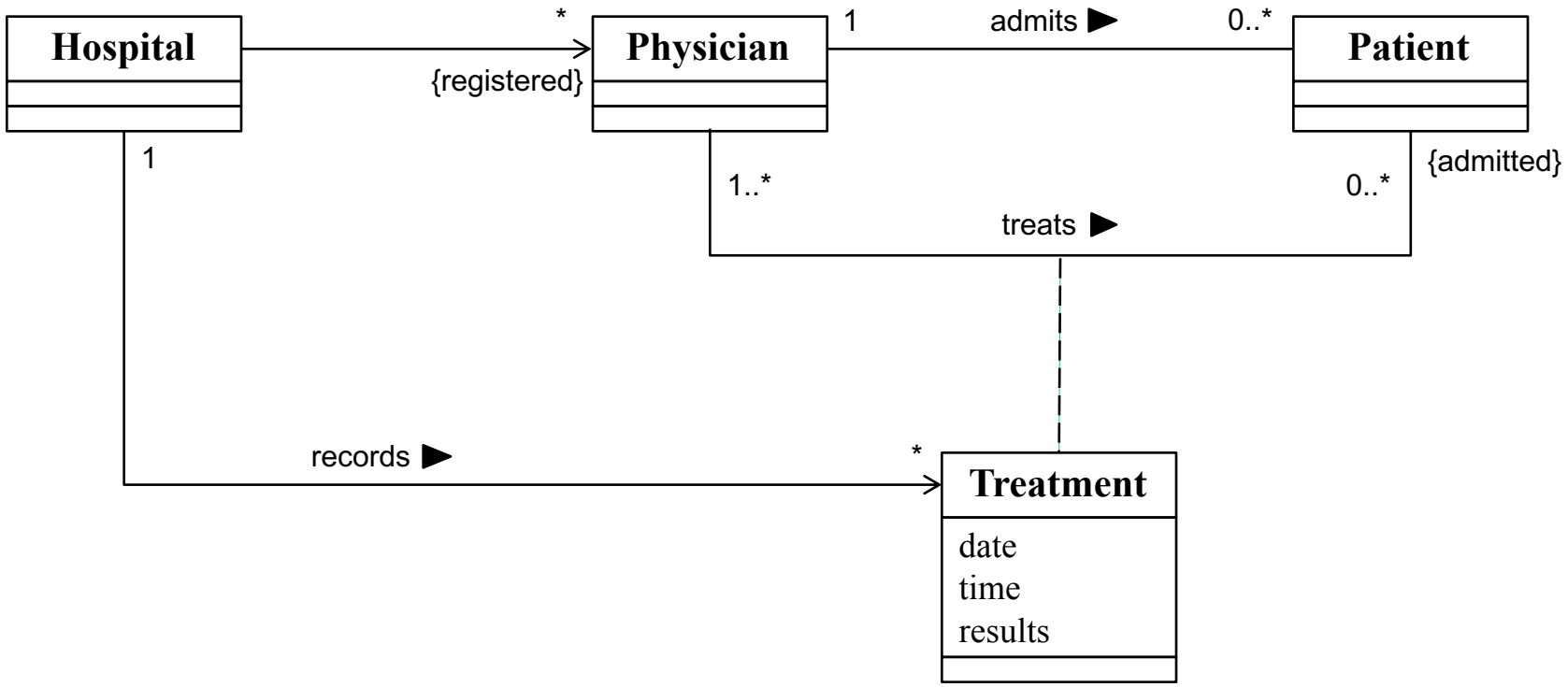
Requirements Statement$_5$

- A particular physician may treat any number of patients, or he or she may not treat any patients.

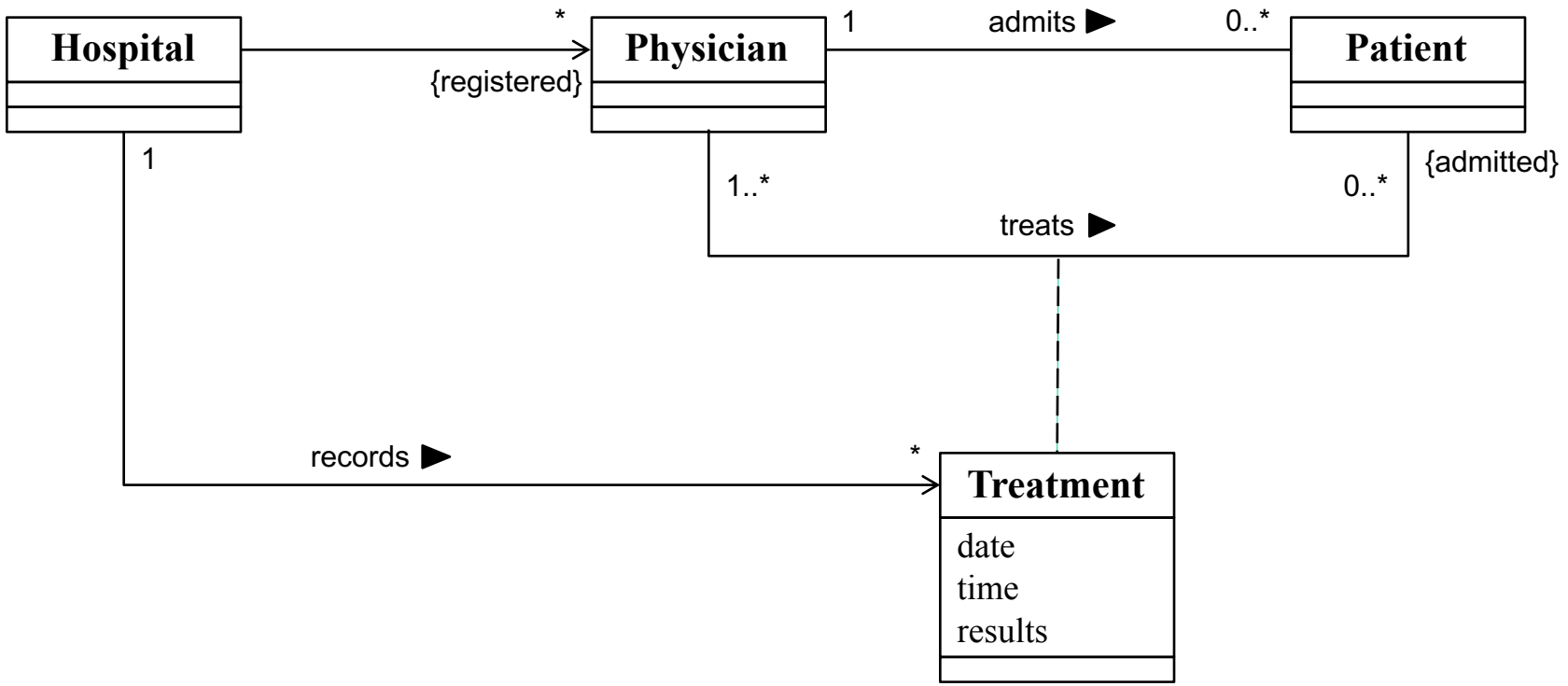


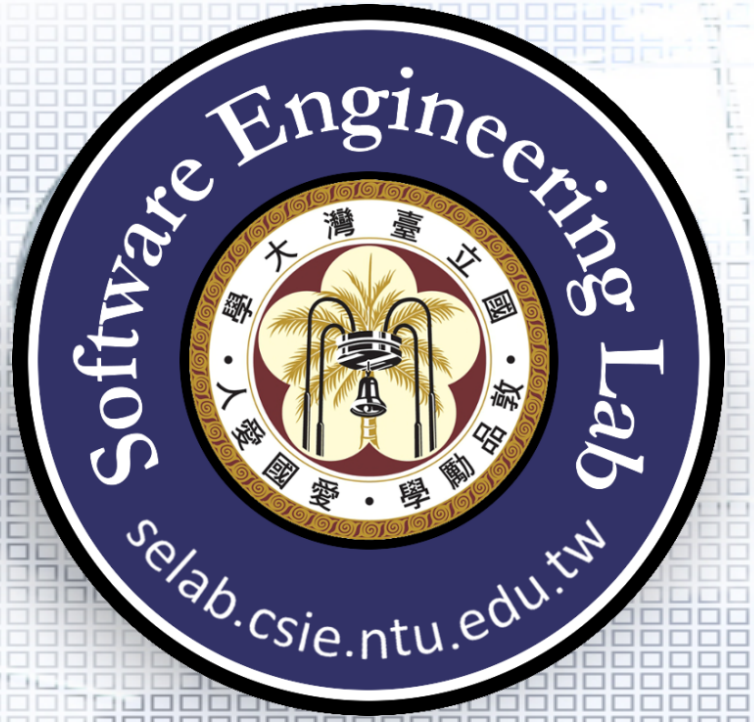
Requirements Statement$_6$

- 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.



Class Diagram Design

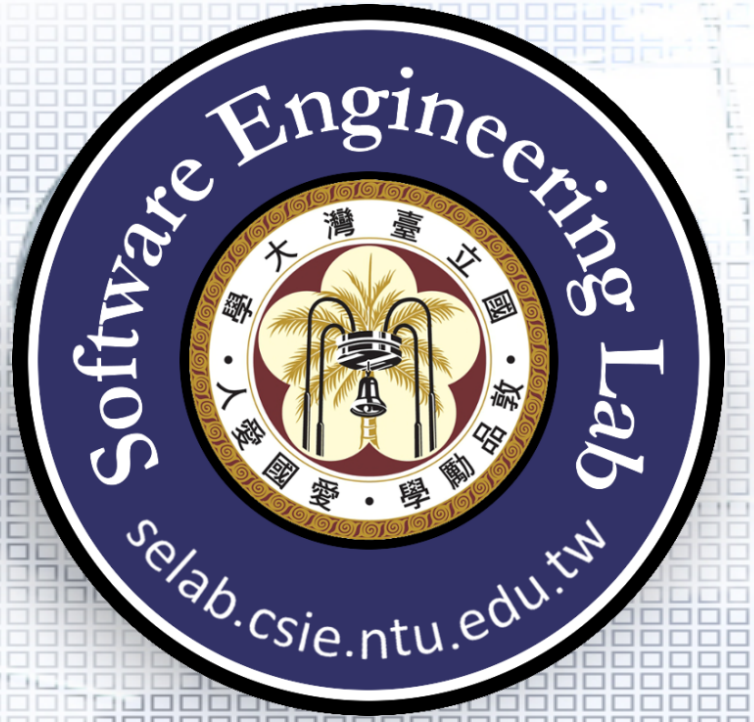




Homework

Prof. Jonathan Lee (李允中)

Department of Computer Science and Information Engineering
National Taiwan University



Sharpen Your Skill

Prof. Jonathan Lee (李允中)

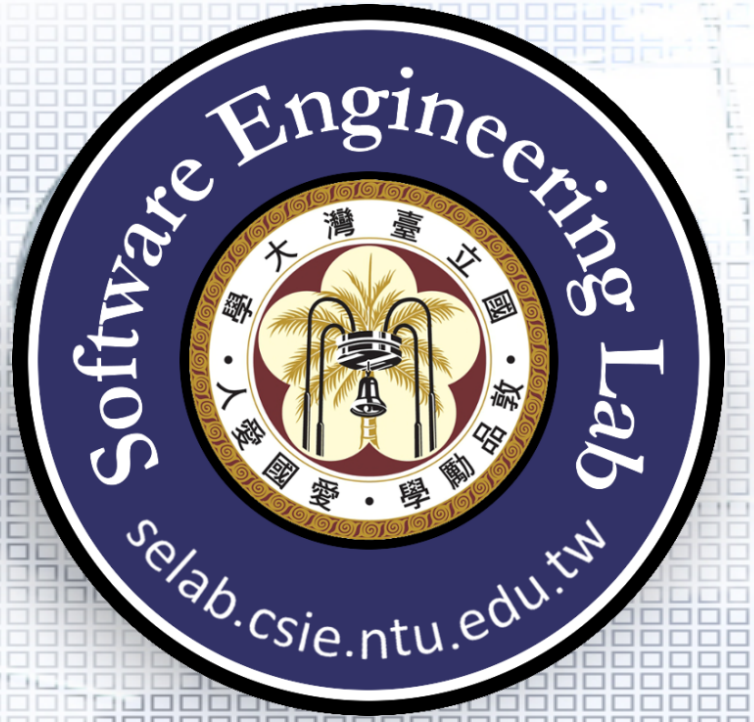
Department of Computer Science and Information Engineering
National Taiwan University

Sharpen Your Skill$_1$

- A country has a capital city.
- A dining philosopher is using a fork.
- A file is an ordinary file or a directory file.
- Files contain records.
- A drawing object is text, a geometrical object, or a group.

Sharpen Your Skill$_2$

- 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.



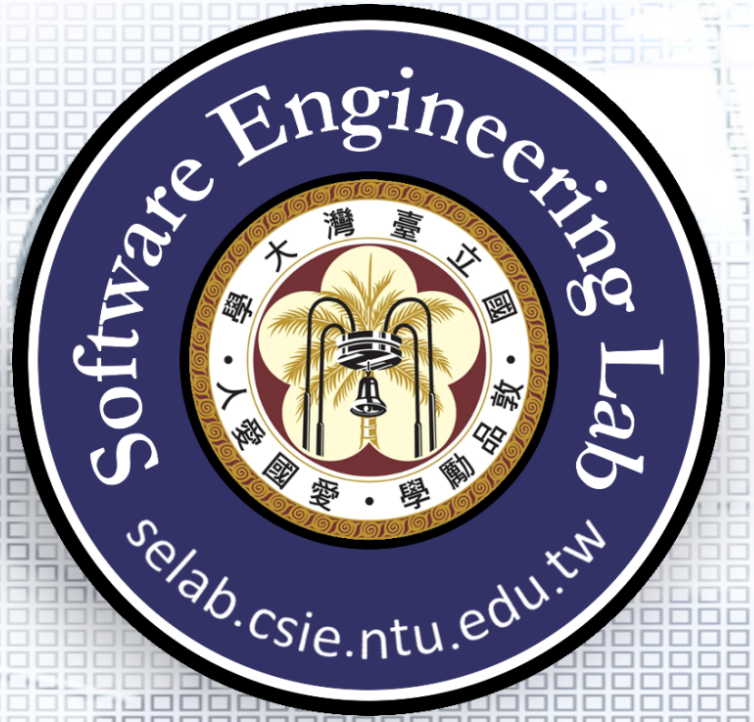
An Order Handling System

Prof. Jonathan Lee (李允中)

Department of Computer Science and Information Engineering
National Taiwan University

Requirements Statement

- Design the order handling functionality for a different type of an online shopping site that transmits orders to different order fulfilling companies based on the type of the goods ordered.
- Suppose that the group of order processing companies can be classified into three categories based on the format of the order information they expect to receive.
- These formats include comma-separated value (CSV), XML and a custom object. When the order information is transformed into one of these formats, appropriate header and footer information that is specific to a format needs to be added to the order data. The series of steps required for the creation of an Order object can be summarized as follows:
  - Create the header specific to the format
  - Add the order data
  - Create the footer specific to the format



A Sales Reporting Application

Prof. Jonathan Lee (李允中)

Department of Computer Science and Information Engineering
National Taiwan University

Requirements Statements

- Build a sales reporting application for the management of a store with multiple departments.
- Users should be able to select a specific department they are interested in.
- Upon selecting a department, two types of reports are to be displayed:
  - Monthly report - A list of all transactions for the current month for the selected department.
  - YTD sales chart - A chart showing the year-to-date sales for the selected department by month.
- Whenever a different department is selected, both of the reports should be refreshed with the data for the currently selected department.