Object-Oriented Design

Shin-Jie Lee (李信杰)

Assistant Professor

Computer and Network Center

Department of CSIE

National Cheng Kung University

National Cheng Kung University

1

Why Modeling First, Coding Later?

- ❑ Analyze the problems in a higher level point of view similar to natural languages

- ❑ Define the structure and behavior of a target software system in a visualized manner

- ❑ Communicate with users without using implementation languages

- ❑ Defer the vicious cycle of debugging-Fixing to a later stage until we figure out the problem space and its corresponding solution space

- ❑ Link objects in a model to executable code in a more systematic manner

Why Objects?

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

Problem Domain (WHAT) ⟶ Solution Domain (HOW)

Objects necessary for describing a problem space

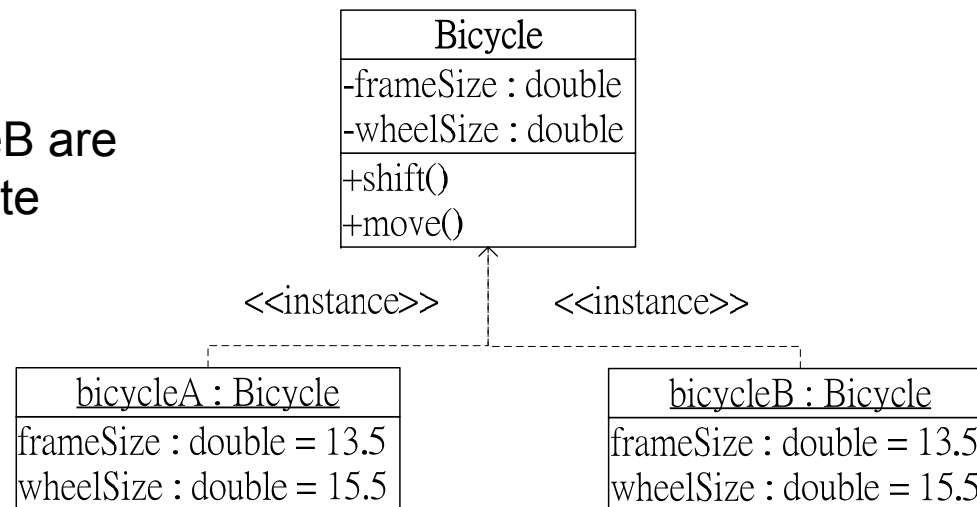Objects required for implementing a solution space

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.

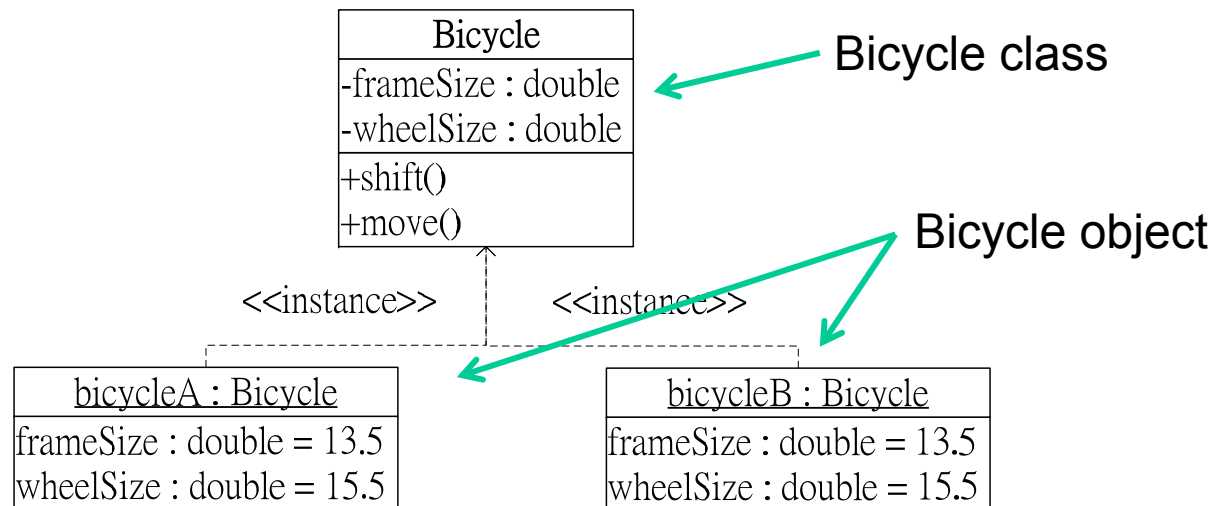e.g. bicycleA and bicycleB are distinct even their attribute values are identical

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

<<instance>>   <<instance>>

bicycleA : Bicycle
frameSize : double = 13.5
wheelSize : double = 15.5

bicycleB : Bicycle
frameSize : double = 13.5
wheelSize : double = 15.5

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

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

Bicycle class

Bicycle object

<<instance>>   <<instance>>

bicycleA : Bicycle
frameSize : double = 13.5
wheelSize : double = 15.5

bicycleB : Bicycle
frameSize : double = 13.5
wheelSize : double = 15.5

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

{+ | - | #} derived $\left\{ \begin{array}{l} \text{class} \\ \text{interface} \\ \text{int} \end{array} \right\}$ {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.
  ➢ -instance: **Singleton**

Operation

❑ Operations specify how to invoke a particular behavior. A method is an **implementation** of an operation.

❑ Visibility name (parameters): return-type {property}

❑ Examples:

➢ +getSize(): Rectangle

➢ +setSize(name: String): void

➢ +getComponents(): Component [0…＊]

Static Operation

❑ Static operations specify behavior for the class itself and are frequently used as utility operations that don't need to use the attributes of the owning class.

➢ Example: +fileExists(path: String): boolean

Abstract Class

❑ Abstract classes provide an operation signature but no implementation.

➢ e.g.

Movable
+move(): void

Interface

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

➢ e.g.

| <<interface>>
Sortable |
|---|
| *+comesBefore(object: Sortable): boolean* |

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)

❑ Facilitate reusability

Inheritance Example

<>A

+*code() : int*
+message() : string

D1

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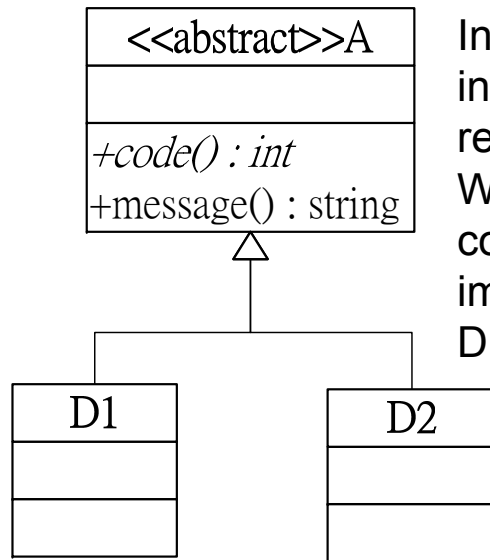
In abstract class, code in message() can be reused in class D1 & D; While abstract operation code() needs to be implemented in D1 & D2.

<<interface>>
A

+*code() : int*

D1

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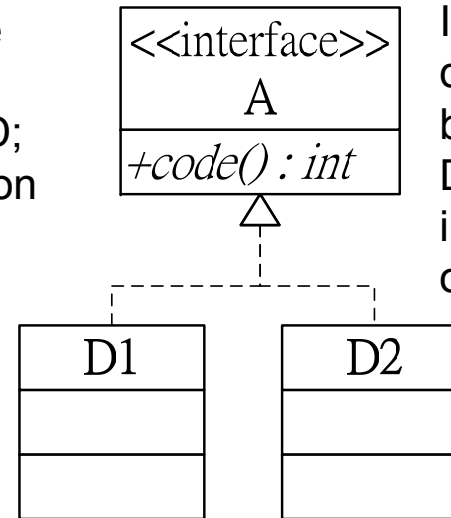
Interface, however, does not have the benefits of code reuse; D1 and D2 have to implement the abstract operation code().

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

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 (compile time)

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

TimeOfDay
+setTime(in time[8] : char) +setTime(in h : int, in m : int, in s : int)

```
class TimeOfDay {
    public void setTime(char[8] time) {…};
    public void setTme(int h, int m, int s) {…};
}
TimeOfDay aClock= new TimeOfDay( );
aClock.setTime(17,1,0);
aClock.setTime("11:55:00");
```
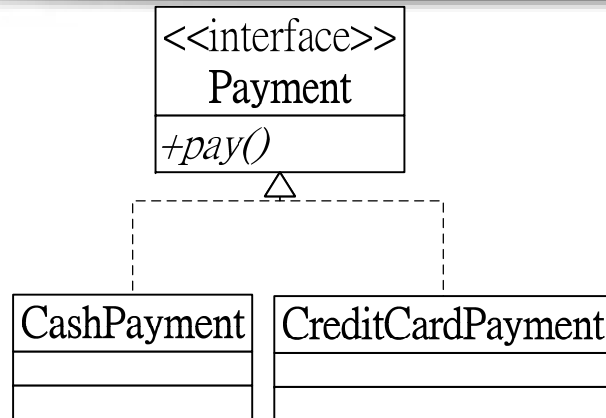
Polymorphism

❑ Dynamic polymorphism (run time): 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.

Polymorphism Example

```
<<interface>>
Payment
+pay()
```

```
CashPayment    CreditCardPayment
```

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

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

```java
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");
  }
}
```

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

The above code outputs: Pay in cash
```

Abstraction

❑ Focus on the essential, inherent aspects of an entity and ignore its accidental properties.

➤ Use of abstraction during analysis: deciding only with application-domain concepts, not making design and implementation decisions.

❑ In Bicycle class, only frame size, wheel size, shift function and move function are considered. Others, like color, weight, and etc. are ignored.

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. (i.e. a well-defined interface)

Only shift() & move() are *visible* to users. Further, the implementation of the two methods are *encapsulated*
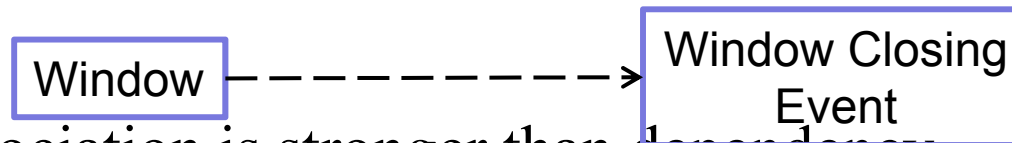
| Bicycle |
| --- |
| -frameSize : double |
| -wheelSize : double |
| +shift() |
| +move() |

Relationship[1]

❑ Dependency is the weakest relationship between classes.

➢ Uses-a

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

➢ A dependent class briefly interacts with the target class

➢

```
┌──────────┐                      ┌────────────────┐
│  Window  │ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ▸│ Window Closing │
└──────────┘                      │     Event      │
                                  └────────────────┘
```

❑ Association is stronger than dependency.

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

➢ Has-a

➢

```
┌──────────┐                ┌──────────┐
│  Window  │ ──────────────▸│  Cursor  │
└──────────┘                └──────────┘
```
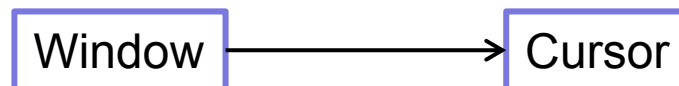
Relationship₂

❑ Aggregation is a stronger version of association.
- ➢ **Implies** ownership
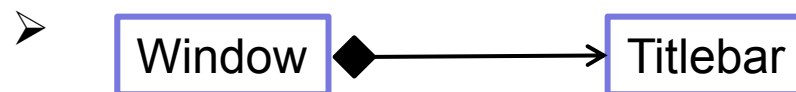- ➢ Owns-a
- ➢

```
[ Window ]◇──────────▶[ Rectangle ]
```

❑ Composition represents a very strong relationship between classes to the point of containment.
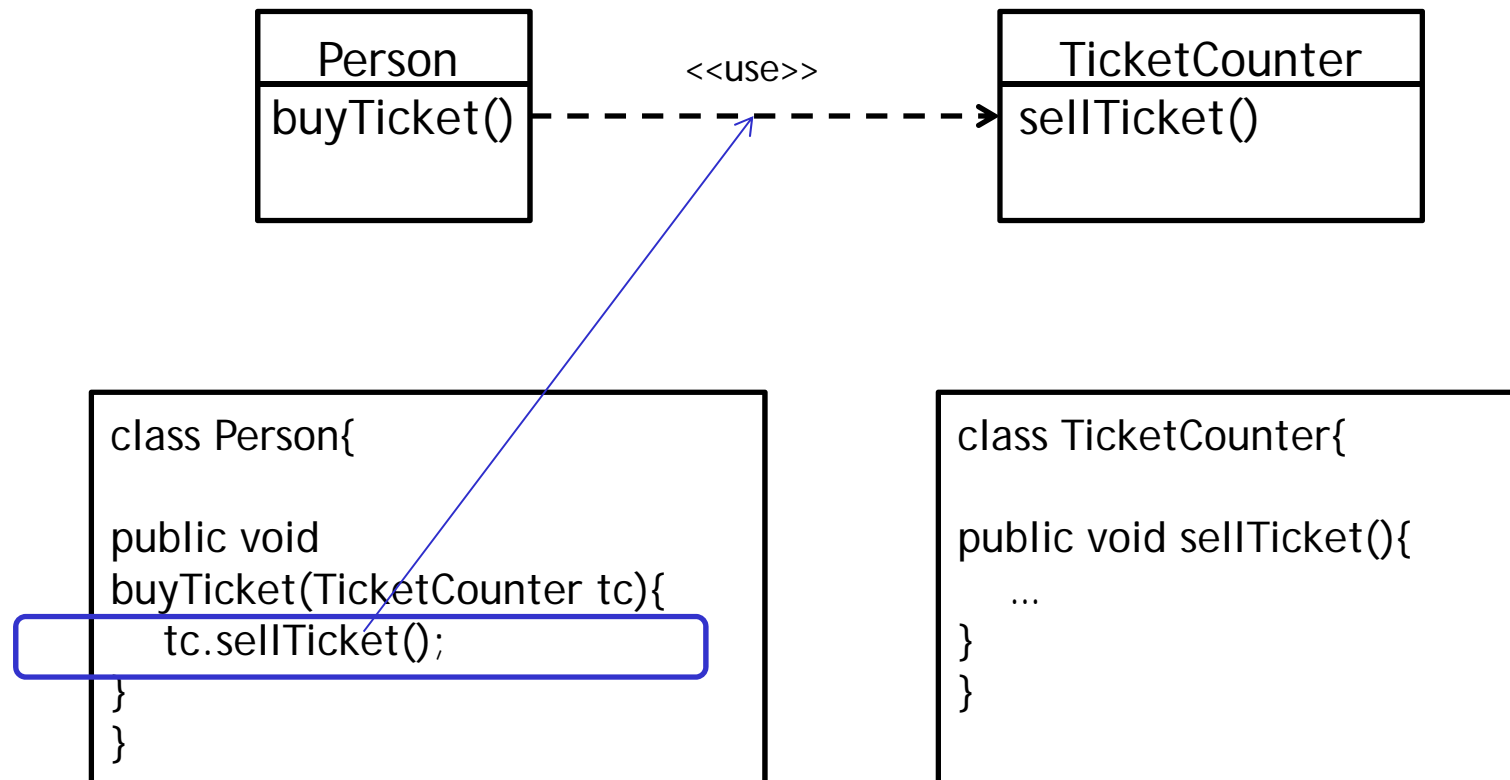- ➢ A whole-part relationship
- ➢ Is-part-of
- ➢

```
[ Window ]◆──────────▶[ Titlebar ]
```

❑ Generalization
- ➢ Is-a
- ➢

```
[ Animal ]◁──────────[ Cat ]
```

Dependency

❑ A dependency states that the implementation or functioning of one or more elements requires the presence of one or more other elements.

Person
buyTicket()

<<use>>

TicketCounter
sellTicket()

```
class Person{

public void
buyTicket(TicketCounter tc){
    tc.sellTicket();
}
}
```

```
class TicketCounter{

public void sellTicket(){
    ...
}
}
```

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.
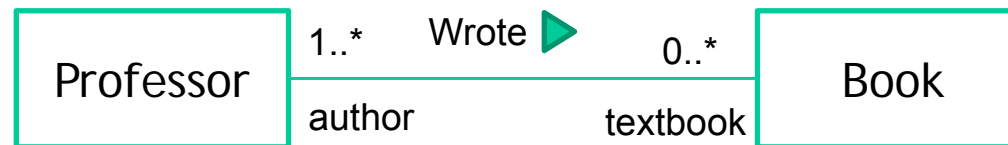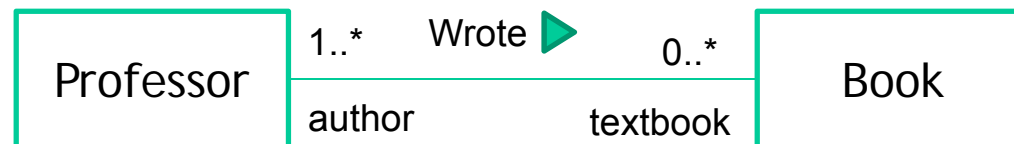

Association **Wrote** between **Professor** and **Book**
with association ends **author** and **textbook**

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 figure below)

➢ The role name is optional.

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

Navigability

❑ Navigability means instances participating in links at runtime (instances of an association) can be accessed efficiently from instances participating in links at the other ends of the association.

- If an end is not navigable, access from the other ends may or may not be possible, and if it is, it might not be efficient.
- An open arrowhead on the end of an association indicates the end is navigable

```
┌─────────┐         ┌─────────┐
│    A    │────×────→│    B    │
└─────────┘         └─────────┘
```
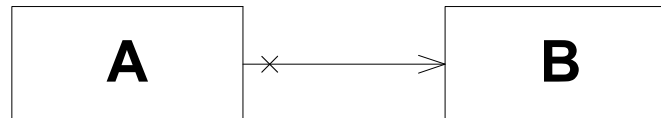
A is not navigable from B while B is navigable from A
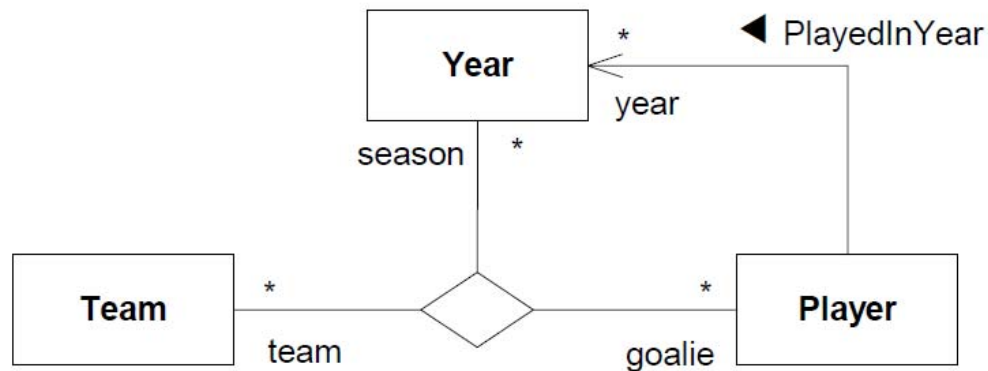
Arity

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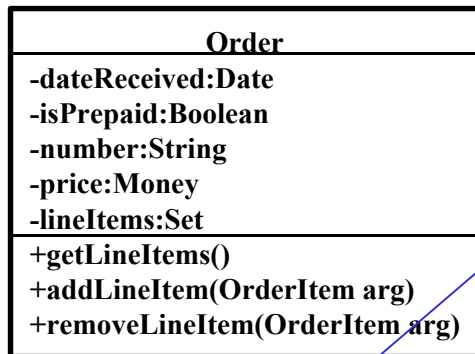
➢ Binary Association

| Study | — | Year |

Study and Year classifiers are associated

➢ N-ary Association

Association Example[1]

Order
- -dateReceived:Date
- -isPrepaid:Boolean
- -number:String
- -price:Money
- -lineItems:Set
- +getLineItems()
- +addLineItem(OrderItem arg)
- +removeLineItem(OrderItem arg)

1

* lineItems

OrderLine
- -quantity:Integer
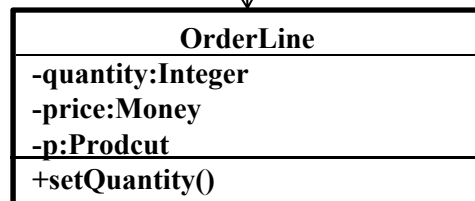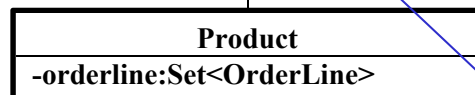- -price:Money
- -p:Prodcut
- +setQuantity()

*

1

Product
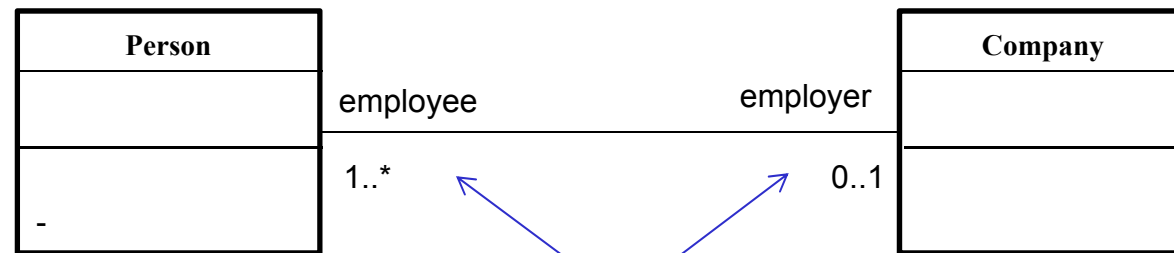- -orderline:Set<OrderLine>

```
public class Order {
  private Date dateReceived;
  private Boolean isPrepaid;
  private String number;
  private Money price;
  private Set 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$_2$

```
         Person                                    Company
┌──────────────────┐  employee        employer  ┌──────────────────┐
│                  │───────────────────────────│                  │
│                  │   1..*              0..1    │                  │
│  -               │                             │                  │
└──────────────────┘                             └──────────────────┘
```

```
public class Person {
  private  Company employer;

  public void setEmployer (Company c){
          employer = c;
  }
  public Company getEmployer(){
          return employer;
  }
}
```

```
public class Company {
  private  Set<Person> employee;

  public void addEmployee(Person p){
          employee.add(p);
  }
  public Set<Person> getEmployee(){
          return employee;
  }
}
```
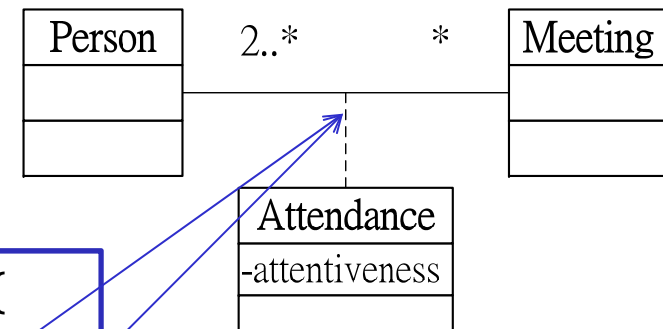
Association Class

❑ 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 ~~ated 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;
…
}
```

Person	2..*	*	Meeting

Attendance
-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)
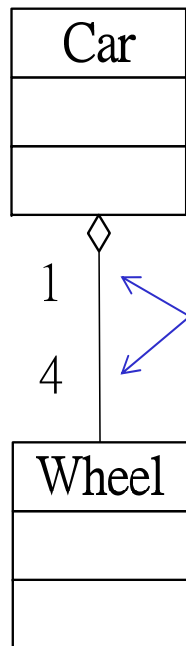 ➢ Composite aggregation (composition)

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

Car

Wheel

1

4

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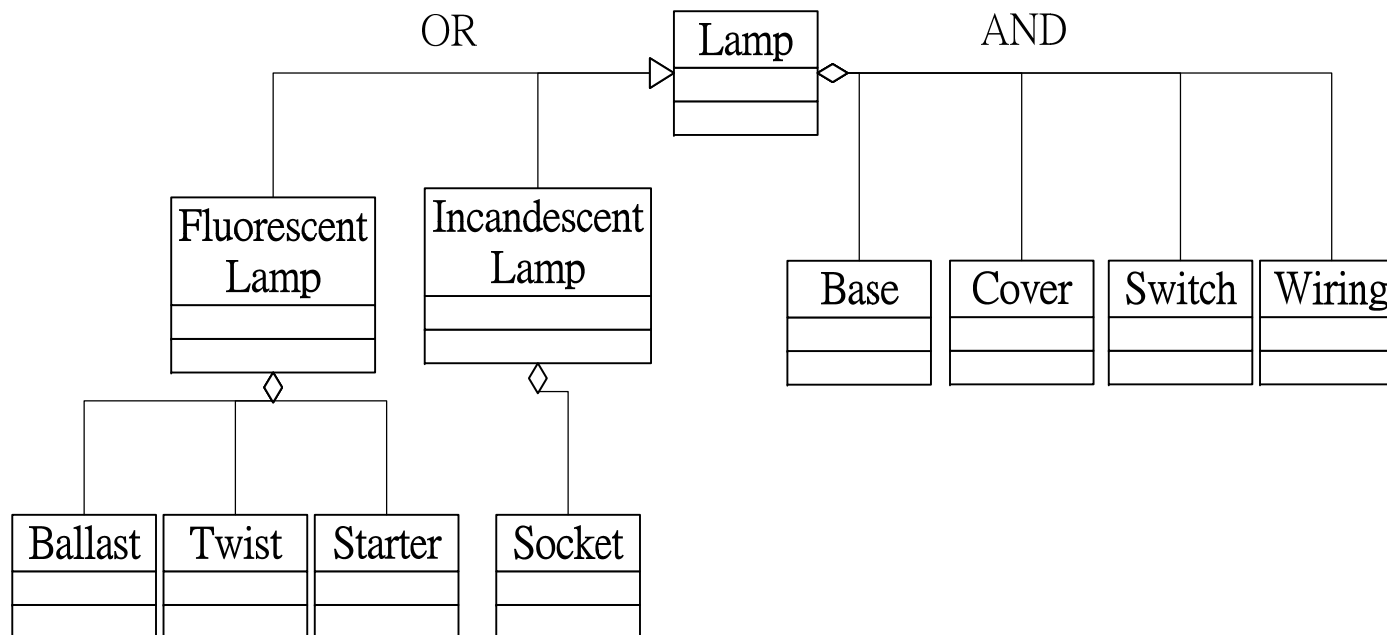
```
public class Wheel {
  private Car car;
  …
}
```

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.

OR   Lamp   AND

Fluorescent Lamp   Incandescent Lamp   Base   Cover   Switch   Wiring

Ballast   Twist   Starter   Socket

Delegation

❑ Delegation: An object forwards an operation to another object that is part of or related to the first object for execution.

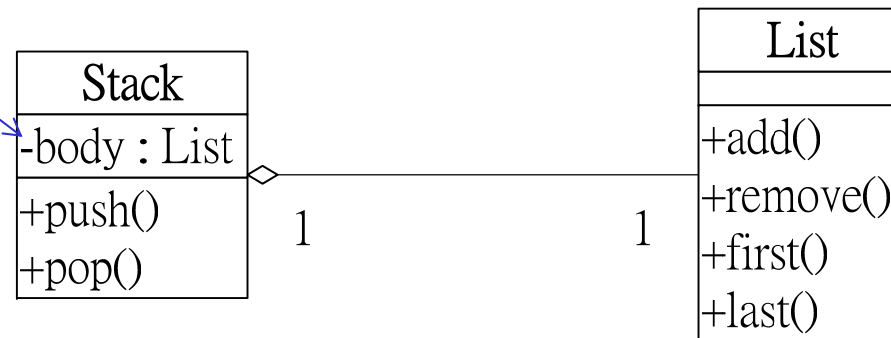➢ This mechanism is sometimes referred to as aggregation, consultation or forwarding.

Stack
-body : List
+push() +pop()

1   1

List
+add() +remove() +first() +last()
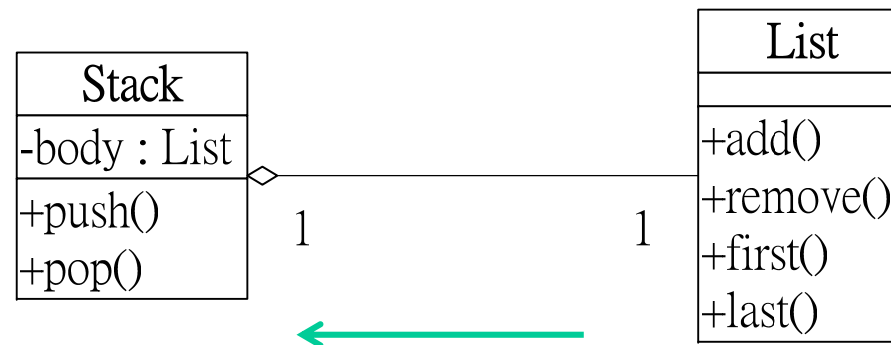
Delegate **add** for **push**
Delegate **last** and **remove** for **pop**
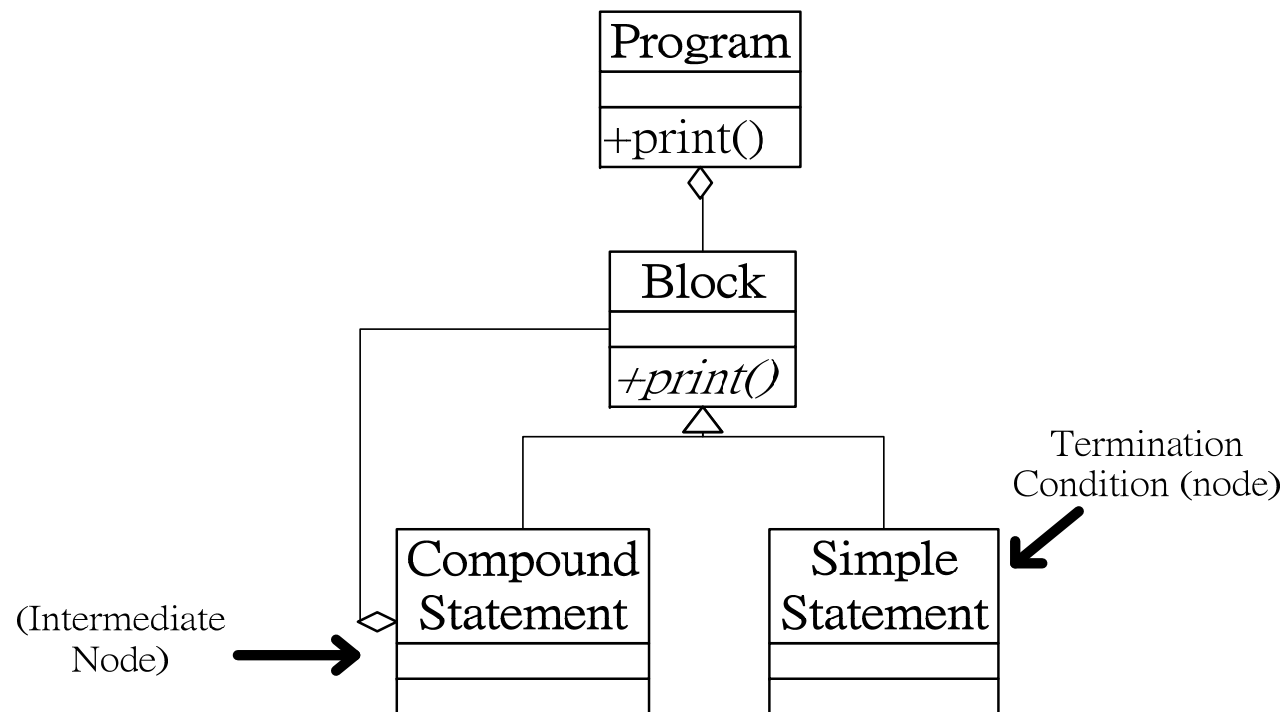
```
public class Stack {
  private List body;

  public void push(Object item) {
    body.add();
  }

  public Object pop() {
    Object o = null;
    if (body.last() != null) {
      o = body.last();
      body.remove();
    }
    return o;
  }
}
```

Stack
-body : List
+push()
+pop()

List
+add()
+remove()
+first()
+last()

1   1

Recursive Aggregation

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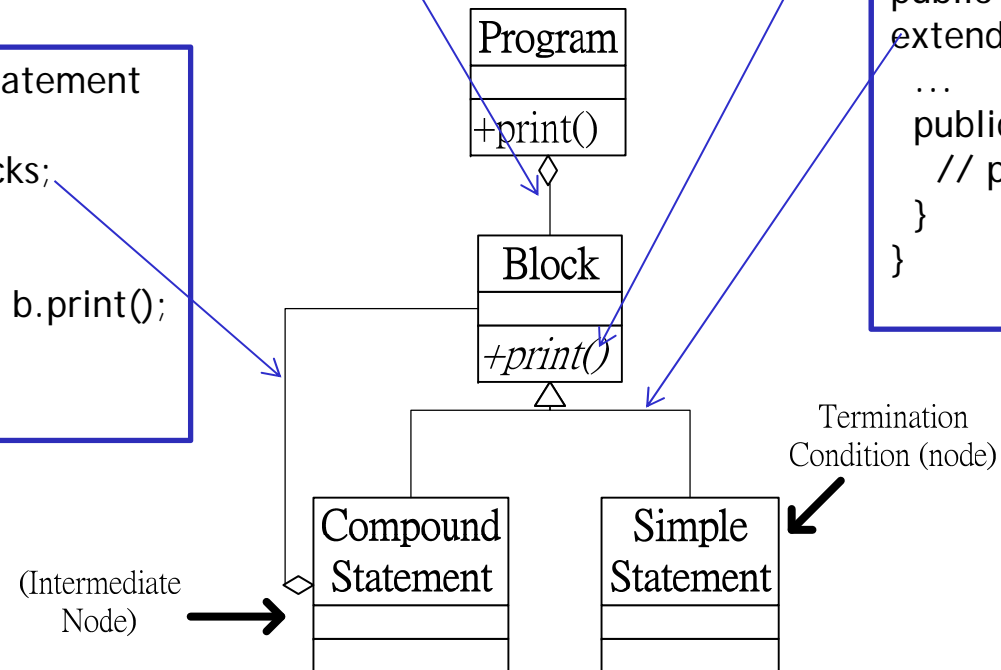
Recursive Aggregation: Java Code

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

```
public 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
   }
}
```

Program

+print()

Block

+*print()*

Compound Statement

Simple Statement

(Intermediate Node)

Termination Condition (node)

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	parent	file	File
	1	*	

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

Building

1

1..*
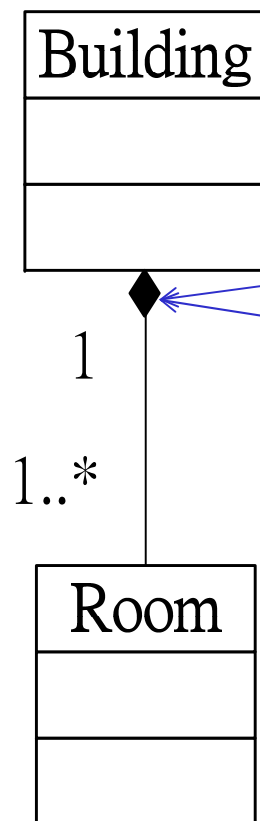
Room

```
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 …;
  …
}
```

41

Composition Example

```
Circle
-------------------------
-radius : double
-center : Point
-------------------------
+setCenter(in p : Point)
+setRadius(in r : double)
```

1
1

```
Point
-------------------------

-------------------------

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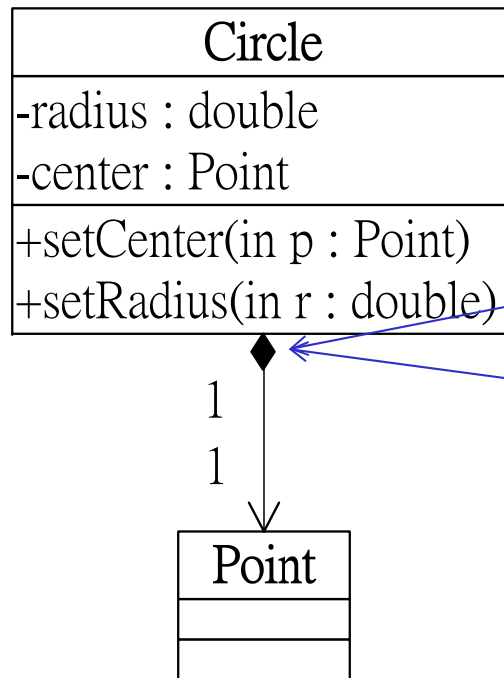
```
public class Circle {
  double radius;
  Point center;

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

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

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

42

Sharpen Your Modeling 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 polygon is composed of an ordered set of points.

❑ A drawing object is text, a geometrical object, or a group.

Sharpen Your Modeling 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.