Object-Oriented Software Design

Content

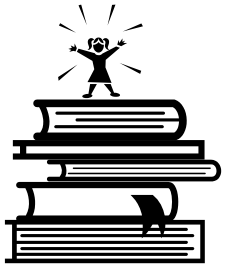
- Object-oriented concepts
- Object modelling using Unified Modelling Language (UML)
- Object-oriented software development and patterns
- CASE tools
- Summary

Introduction

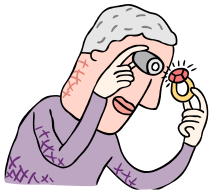
- **Object-oriented design (OOD)** techniques are now extremely popular:
  - Inception in early 1980's.
  - Widespread acceptance in industry and academics.
  - **Unified Modelling Language (UML) became an ISO standard (ISO/IEC 19501) in 2004.**

Objects

- A system is designed as a set of interacting objects.
- **Objects are often real-world entities:**
  - **Examples: an employee, a book etc.**

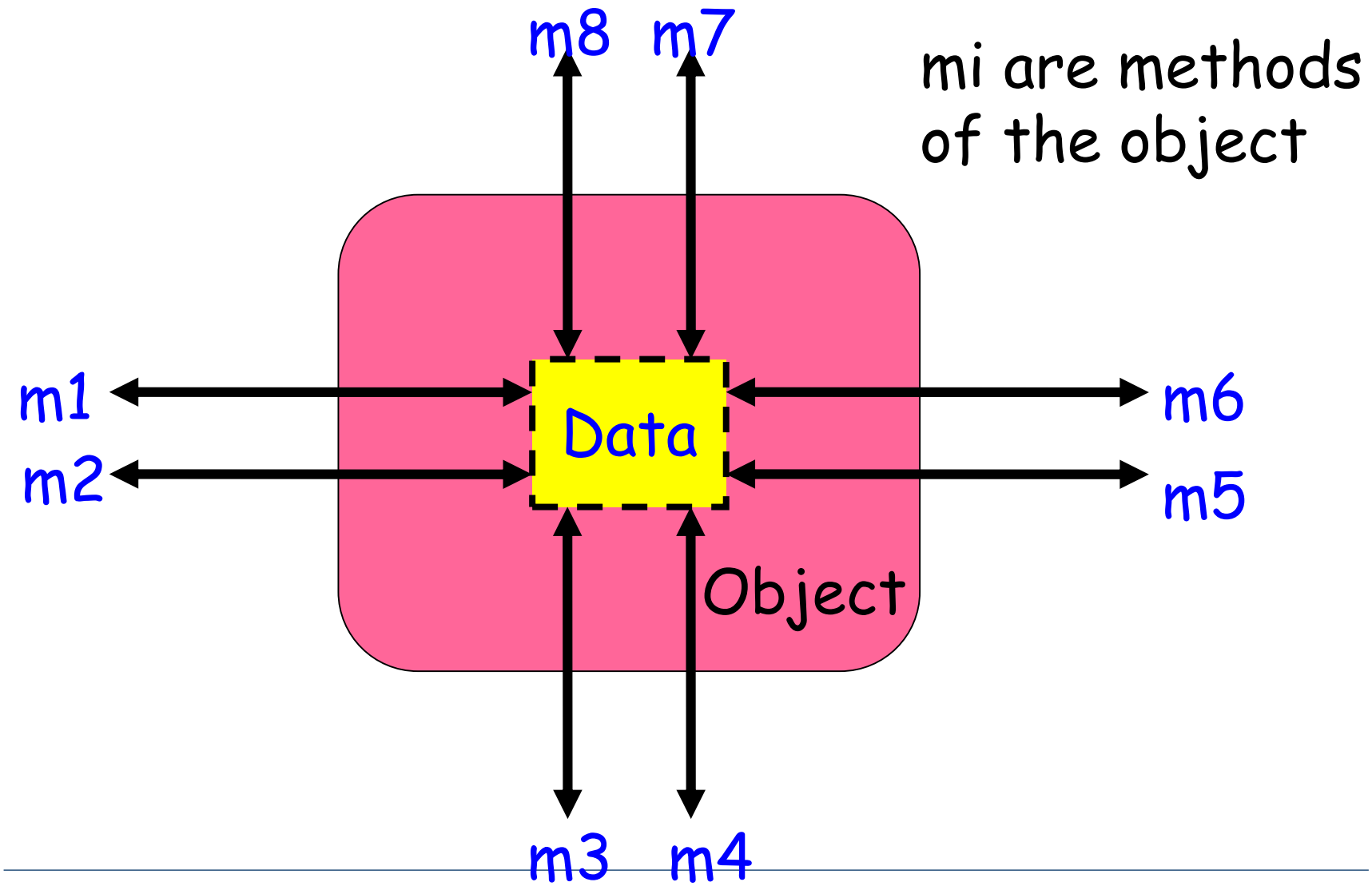


  - **Can also be conceptual objects :**
    - **Controller, etc.**



- An object consists of **data** (attributes) and **functions** (methods) that operate on data.
  - **Encapsulation.**

Model of an object



Encapsulation

- James Rumbaugh *et al*., 1991
  - “A mechanism by which **external aspects of a class** that are visible to or accessible by other objects are **separated from the internal** or implementation details of these aspects.”

Information Hiding

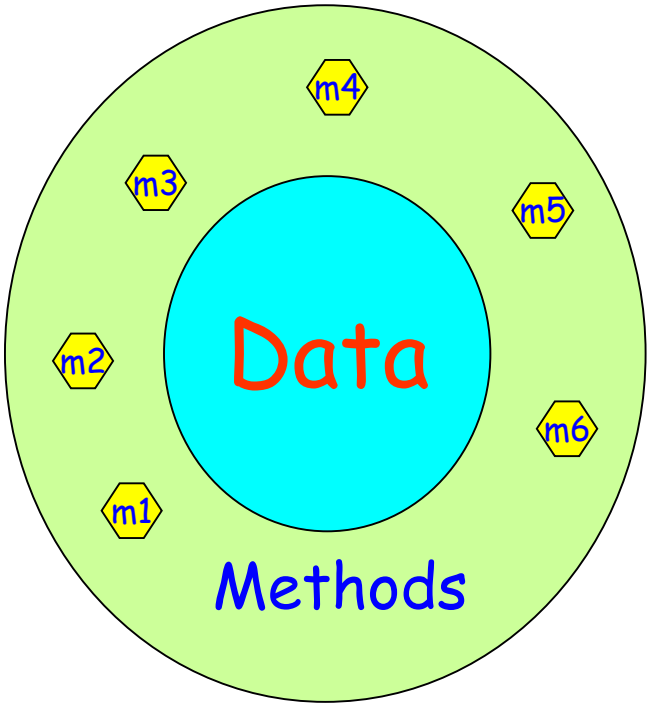
"…encapsulation is most often achieved through ***information hiding***, which is the process of hiding all of the secrets of object that do not contribute to its essential characteristics."

Grady Booch

What is Encapsulation?

- An Object communicates with other objects through messages:

—Data of an object encapsulated within its methods and can be accessed only through its methods.



Class

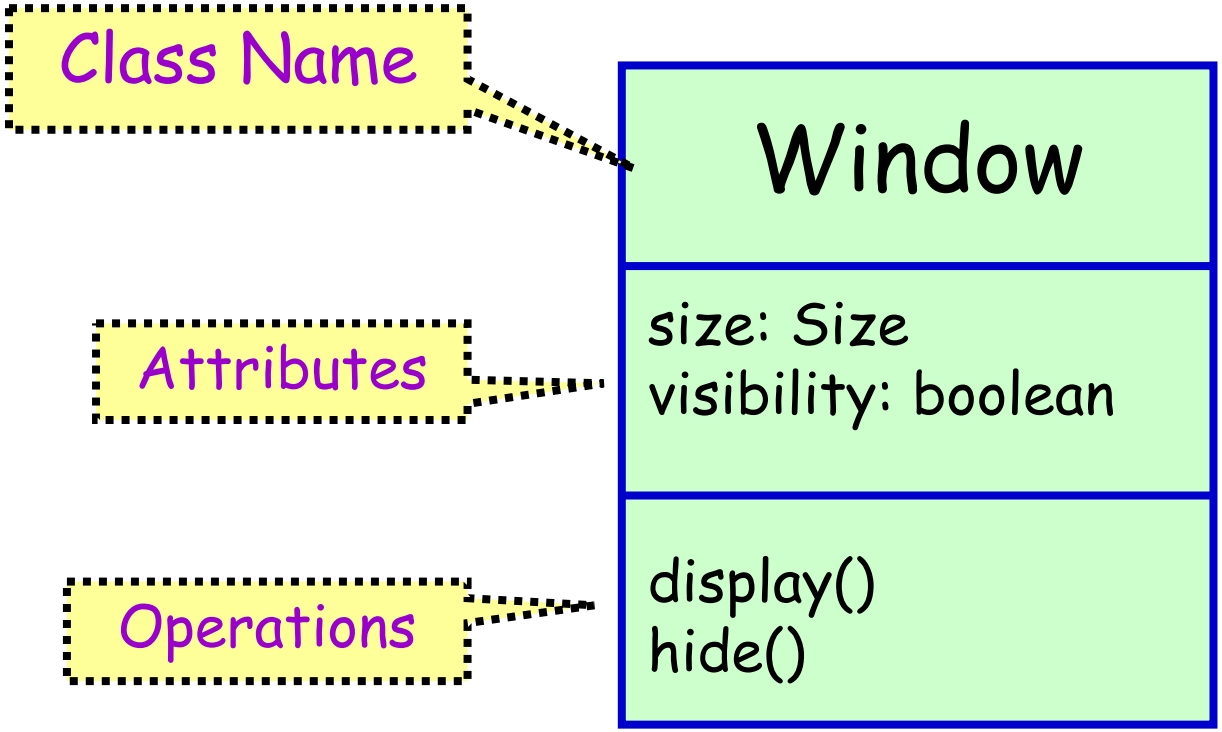
- **Template for object creation**:
  - Instantiated into objects
  - An abstract data type (ADT)
- Examples: Employees, Books, etc.

OO Concepts..

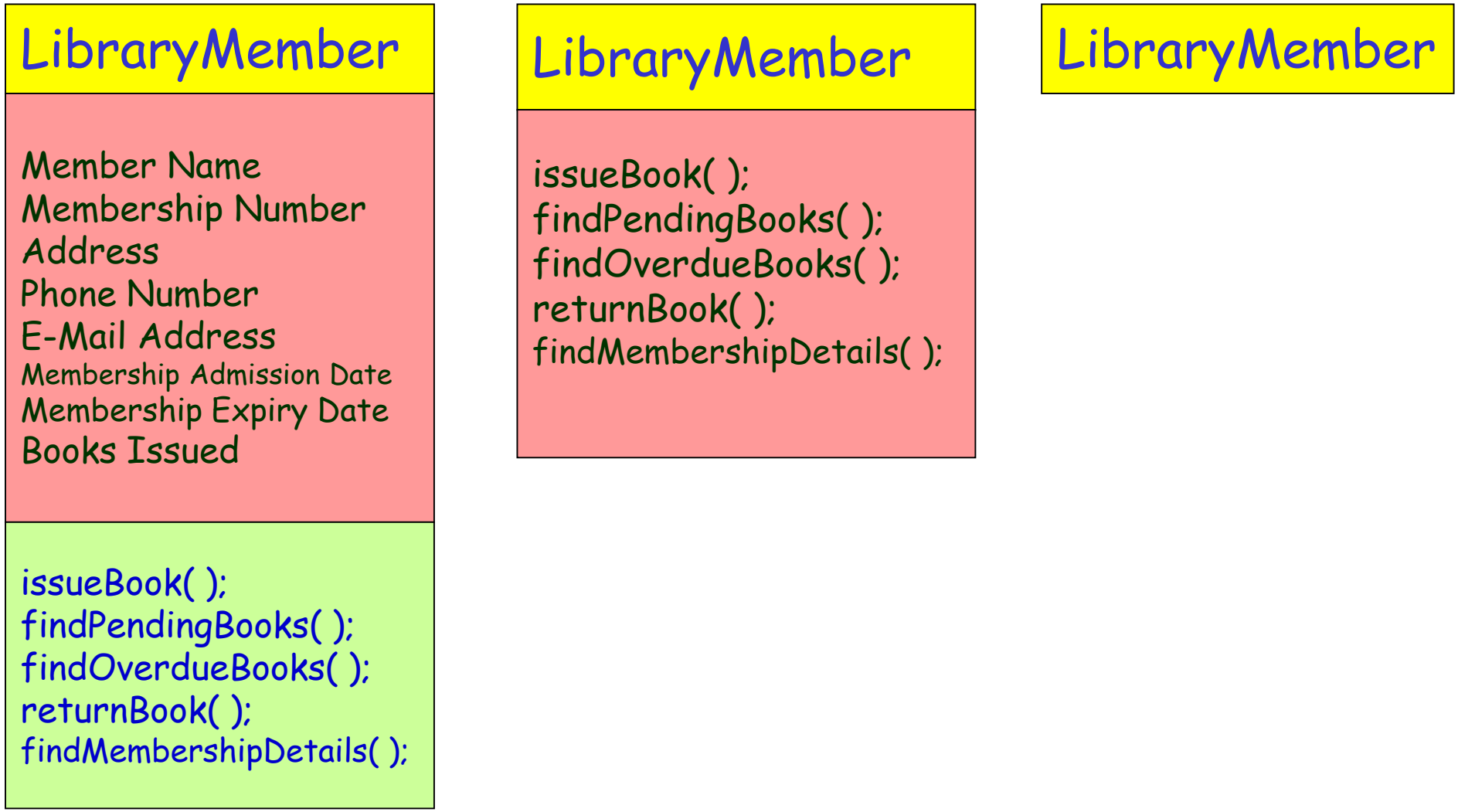
- **Classes** – a class is a stencil from which objects are created; defines the ***structure*** and ***services***. A class has
  - An **interface** which defines which parts of an object can be accessed from outside
  - Body that implements the operations
  - Instance variables to hold object state
- Objects and classes are different; **class** is a **type**, **object** is an **instance**
- **State** and **identity** is of **objects**

UML Class Representation

- A class represents a set of objects having similar attributes, operations, relationships and behavior.



Example UML Classes



Different representations of the LibraryMember class

Class Attribute Examples

Java Syntax	UML Syntax
Date birthday	birthday:Date
Public int duration = 100	+duration:int = 100
Private Student students[0..MAX_Size]	-students[0..MAX_Size]:Student

Visibility Syntax in UML

Visibilty	Java Syntax	UML Syntax
public	public	+
protected	protected	#
package		~
private	private	-

Methods vs. Messages?

- Methods are the operations supported by an object:
  - Means for manipulating the data of an object.
  - Invoked by sending a message (method call).
  - Examples: calculate_salary, issue-book, member_details, etc.

Method Examples

Java Syntax	UML Syntax
void move(int dx, int dy)	~move(int dx,int dy)
public int getSize()	+int getSize()

Are Methods and Operations Synonyms?

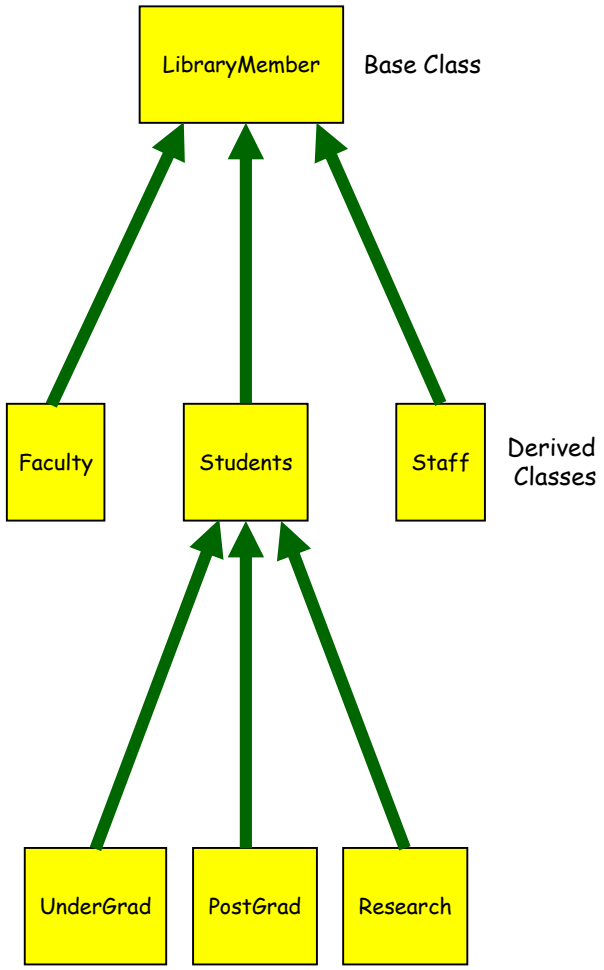
- No
- An operation can be implemented by multiple methods.
  - Known as **polymorphism**
  - In the absence of polymorphism-–the two terms are used as synonyms.

What are the Different Types of Relationships Among Classes?

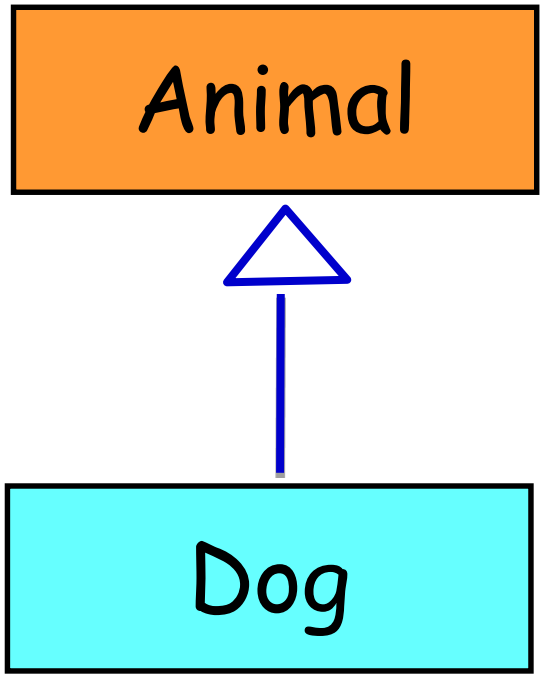
- Four types of relationships:
  - **Inheritance**
  - **Association**
  - **Aggregation/Composition**
  - **Dependency**

Inheritance

- Allows to define a new class (**derived class**) by **extending** an existing class (**base class**).
  - Represents generalization-specialization relationship.
  - Allows redefinition of the existing methods (method overriding).



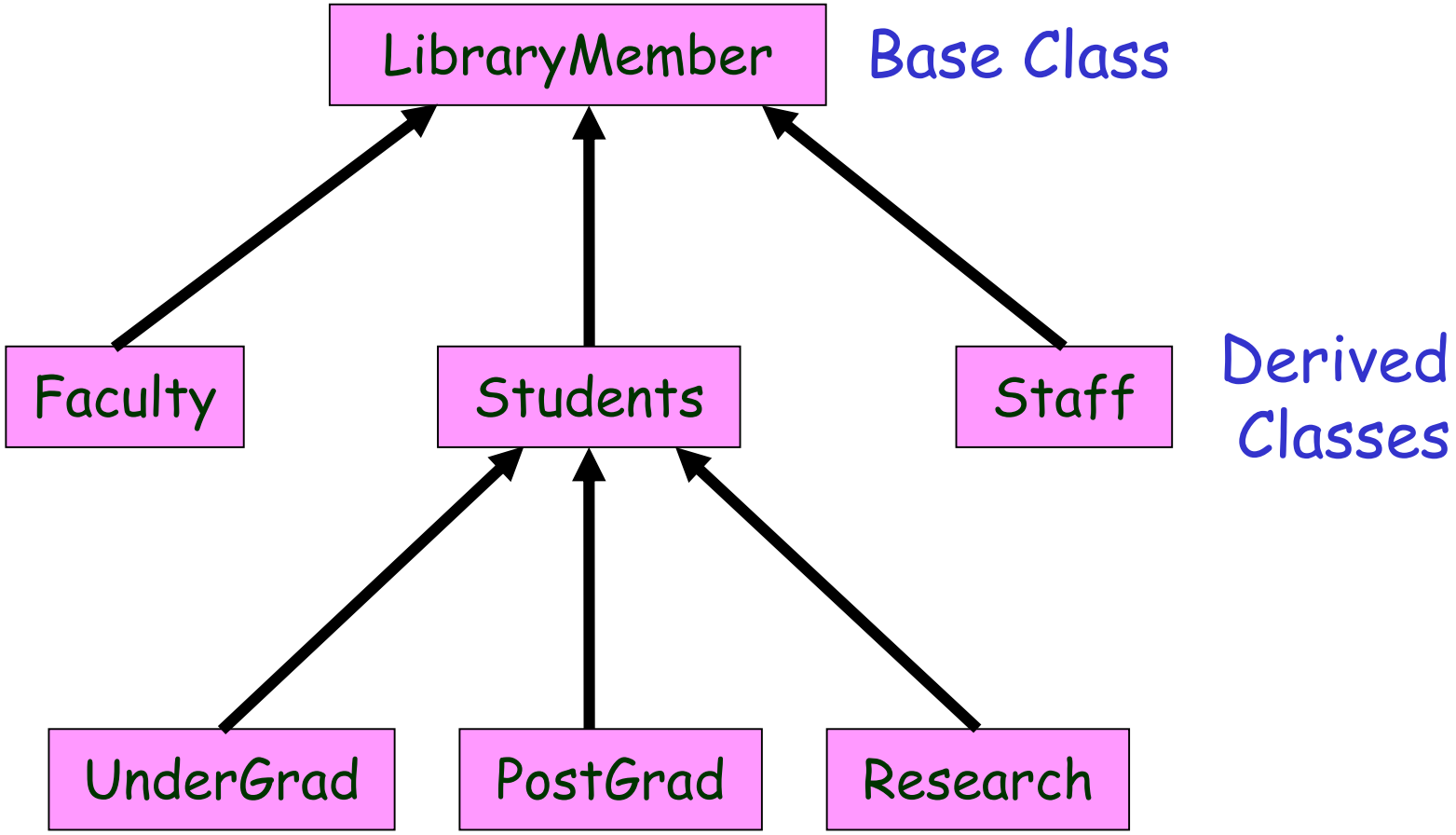
Inheritance Example



"A Dog ISA Animal"

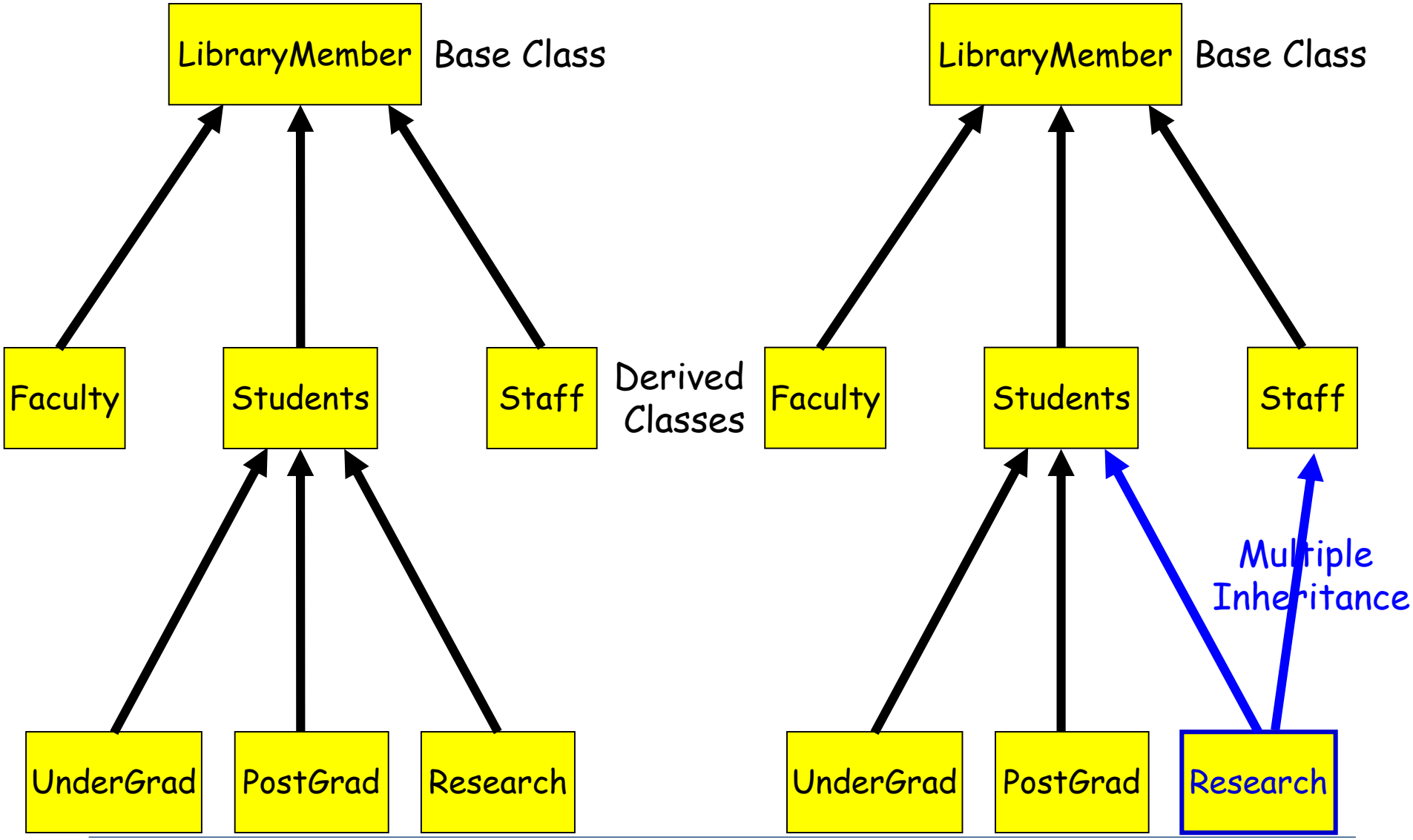
Inheritance

- Lets a subclass **inherit attributes** and **methods** from a base class.



Multiple Inheritance

cont…

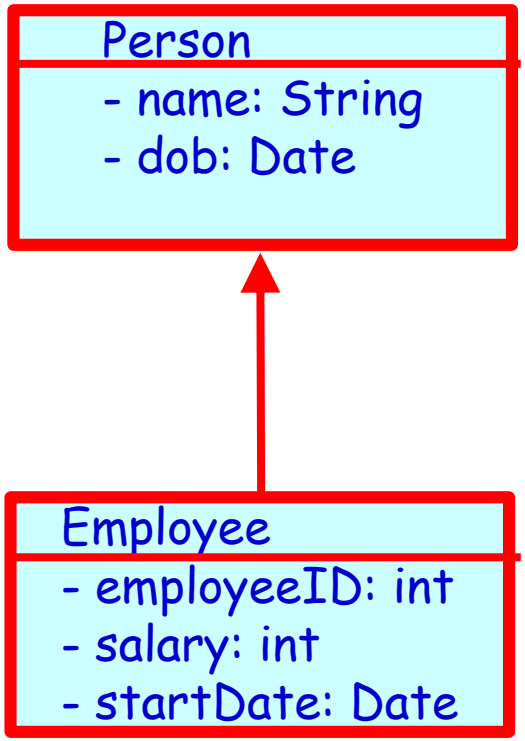


Inheritance Implementation in Java

- Inheritance is declared using the "extends" keyword

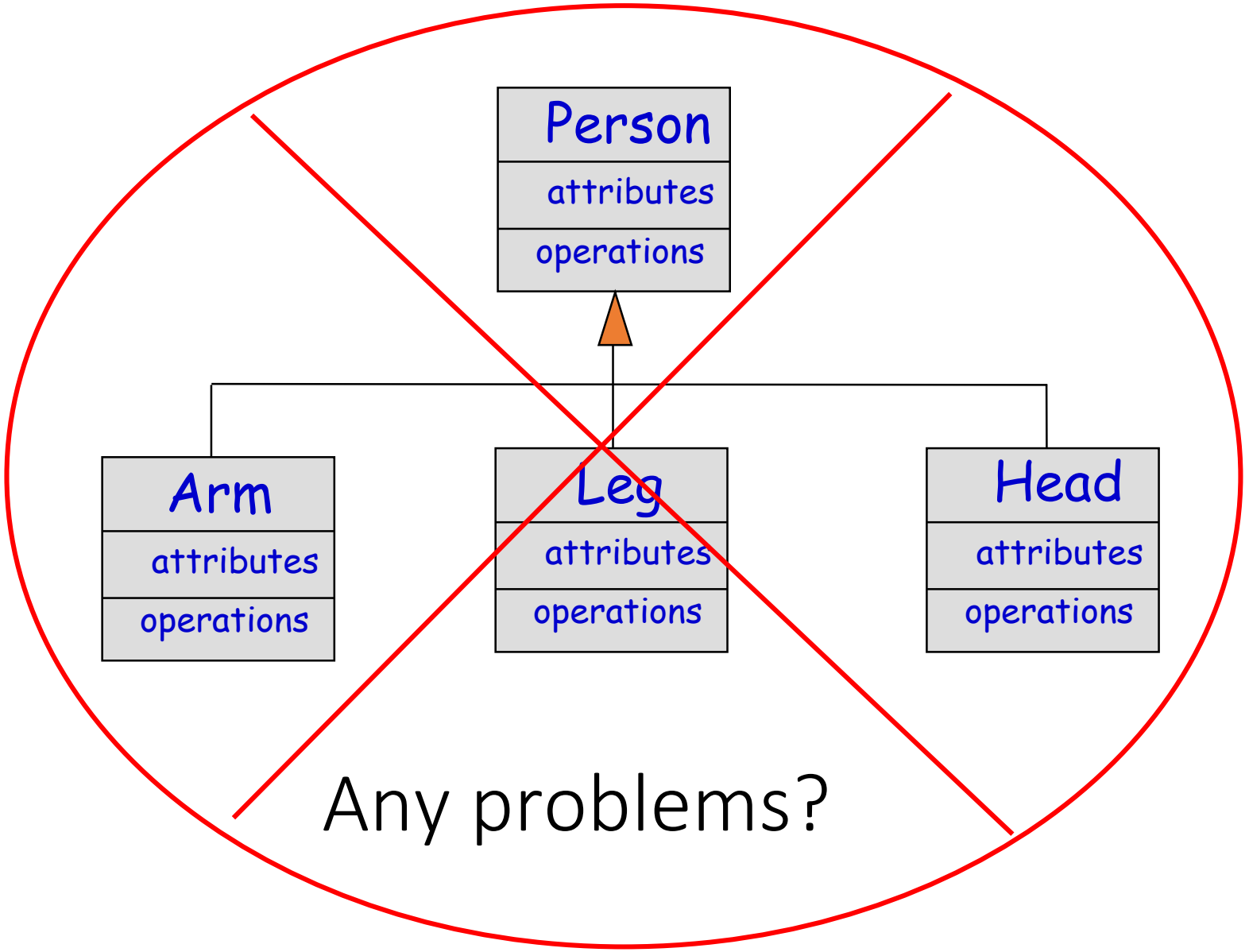
```
class Person{
  private String name;
  private Date dob;
  ...
}
```

```
class Employee extends Person{
  private int employeeID;
  private int salary;
  private Date startDate;
  ...
}
```

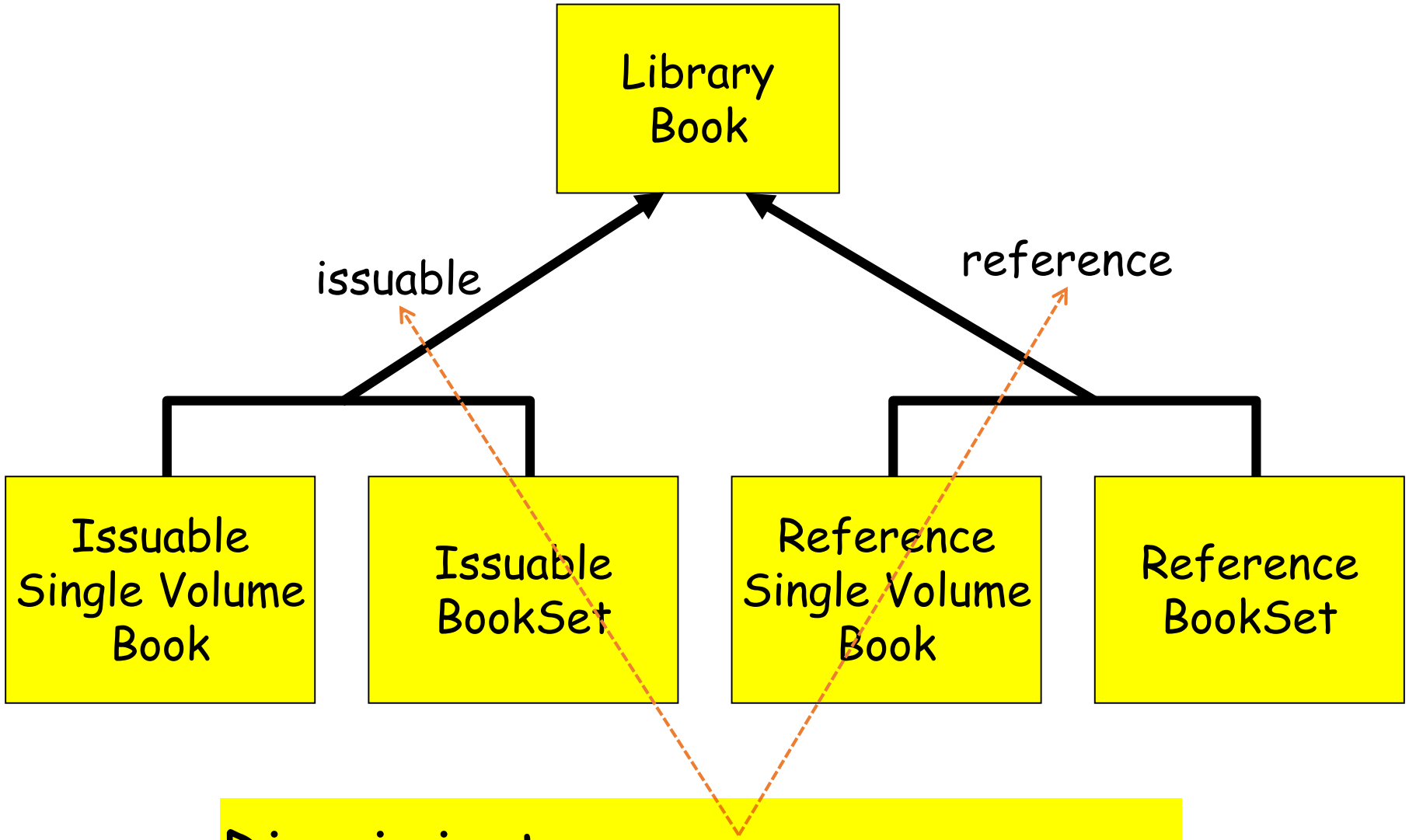



```
Employee anEmployee = new Employee();
```

Poor Generalization Example
(violates "is a" or "is a kind of" heuristic)



Inheritance Example



Discriminator: allows one to **group subclasses** into clusters that correspond to a semantic category.

Inheritance Pitfalls

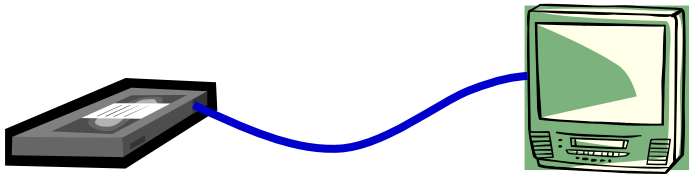
- Inheritance certainly promotes reuse.
- **Indiscriminate use can result in poor quality programs.**
- Base class attributes and methods visible in derived class…

Association Relationship

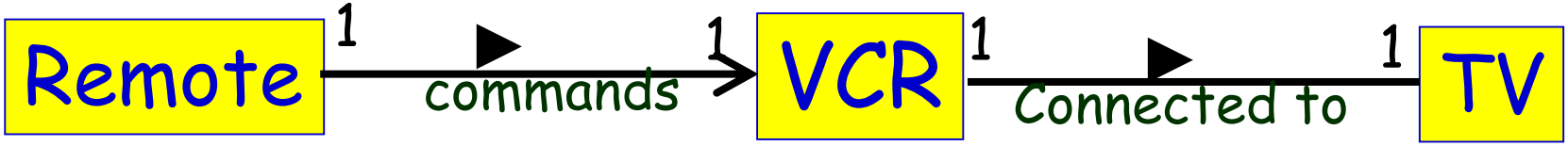
- Enables objects to communicate with each other:

- Usually binary:
  - But in general can be n-ary.

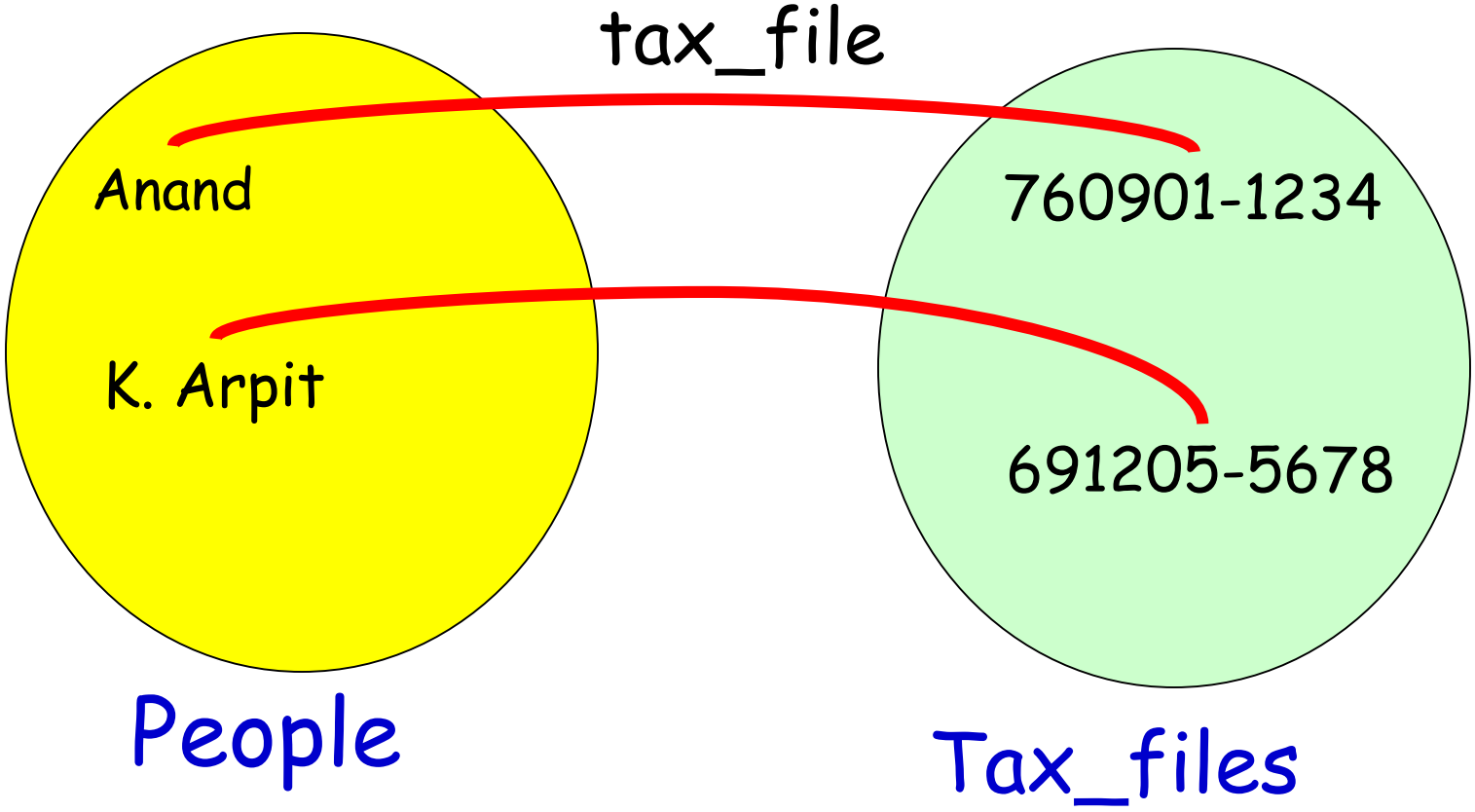
Association – example



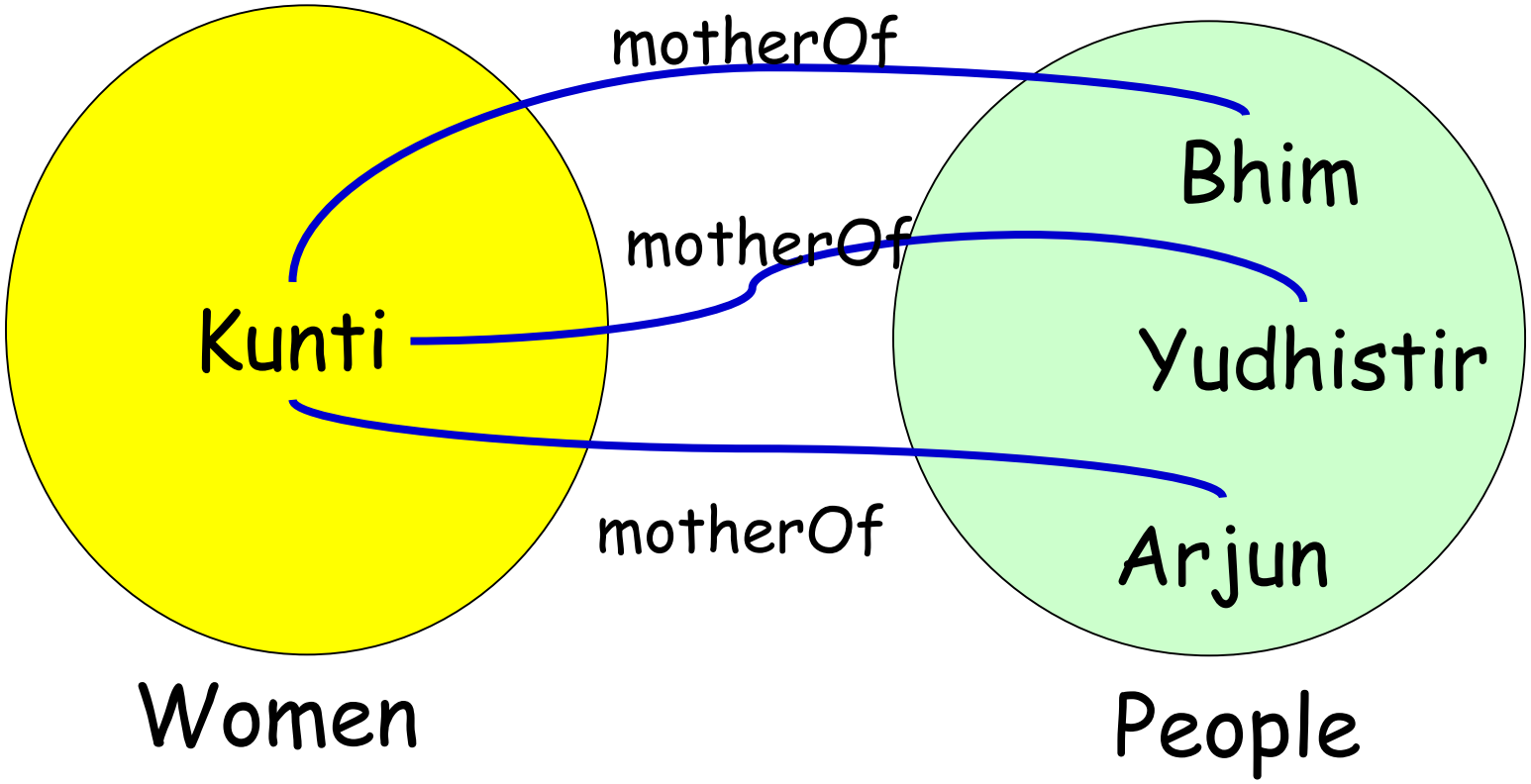
- In a home theatre system,
  - A TV object has an association with a VCR object
    - It may receive a signal from the VCR
  - VCR may be associated with remote
    - It may receive a signal (**command**) to record

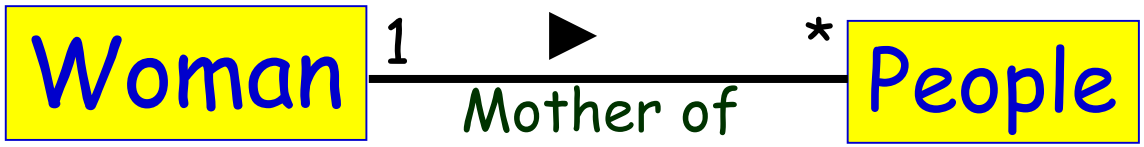


1-1 Association – example

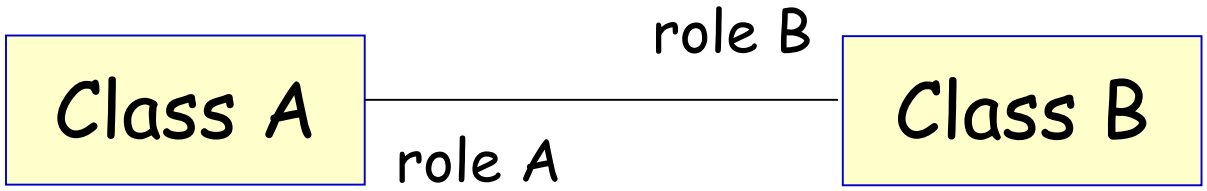


Multiple Association – example

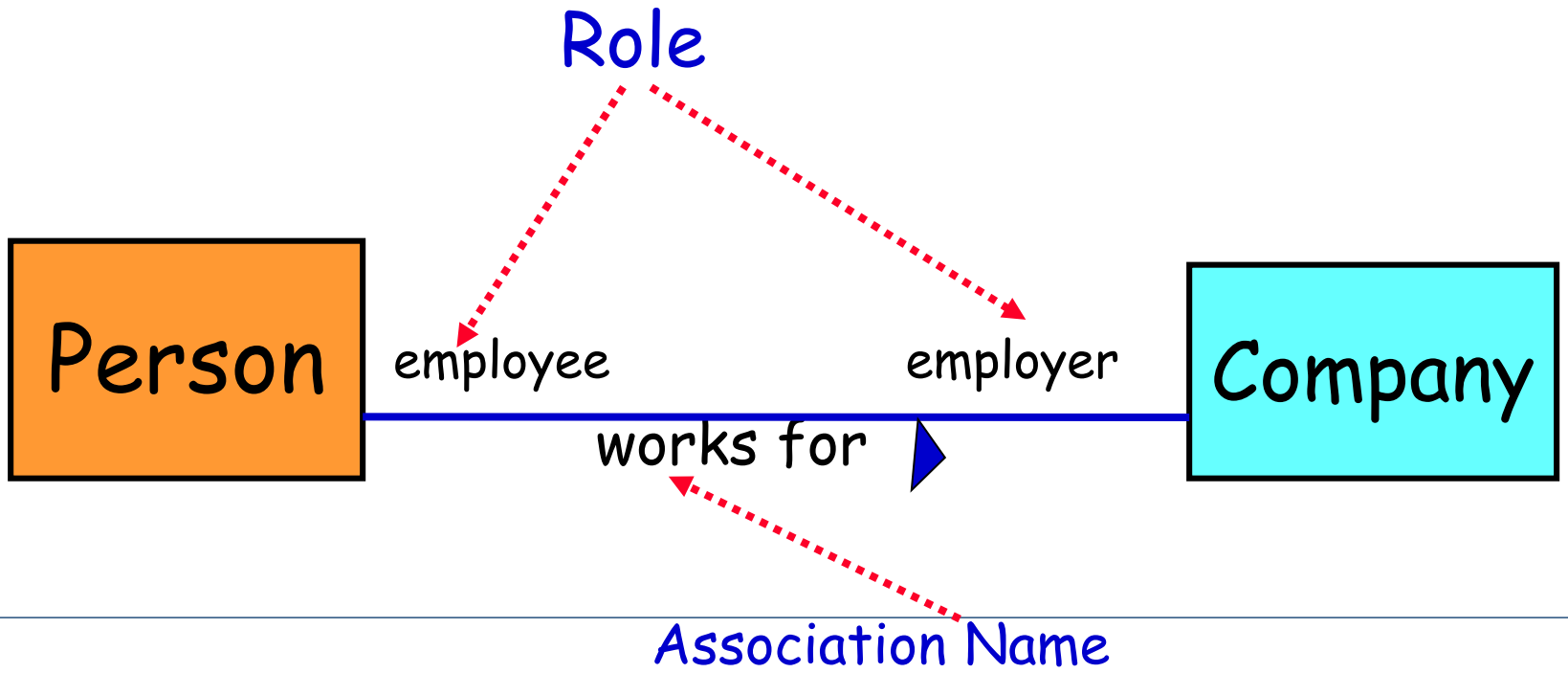




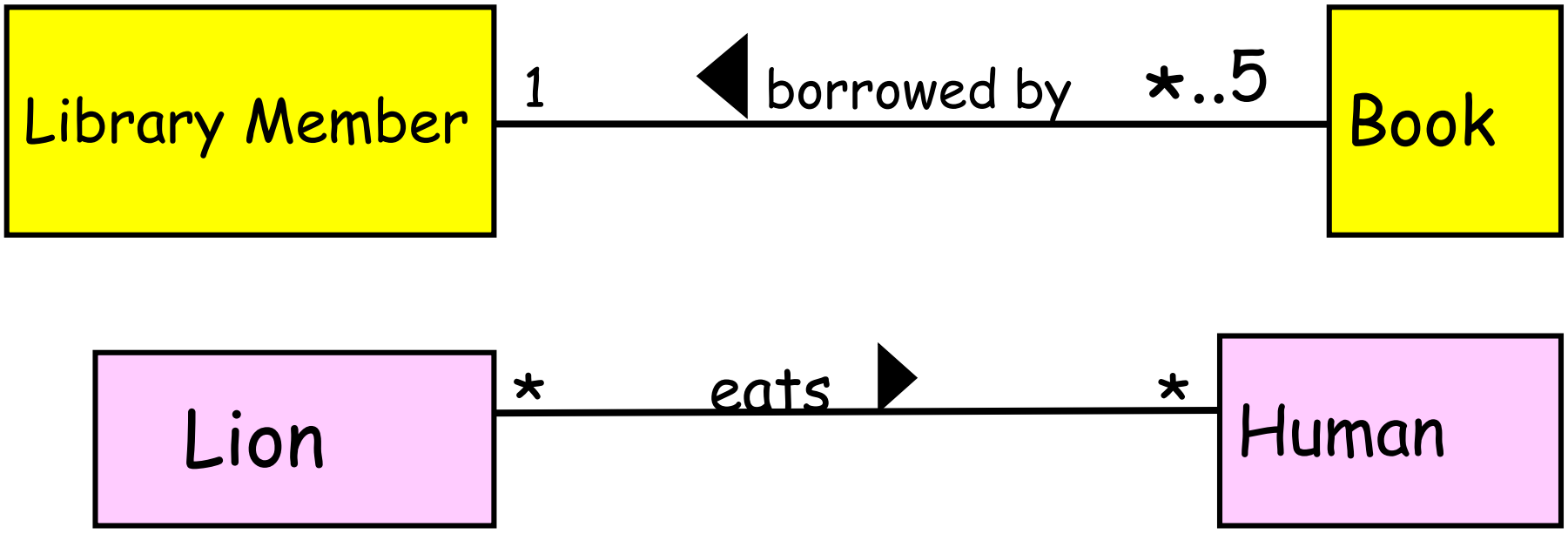
Association UML Syntax



- A Person works for a Company.



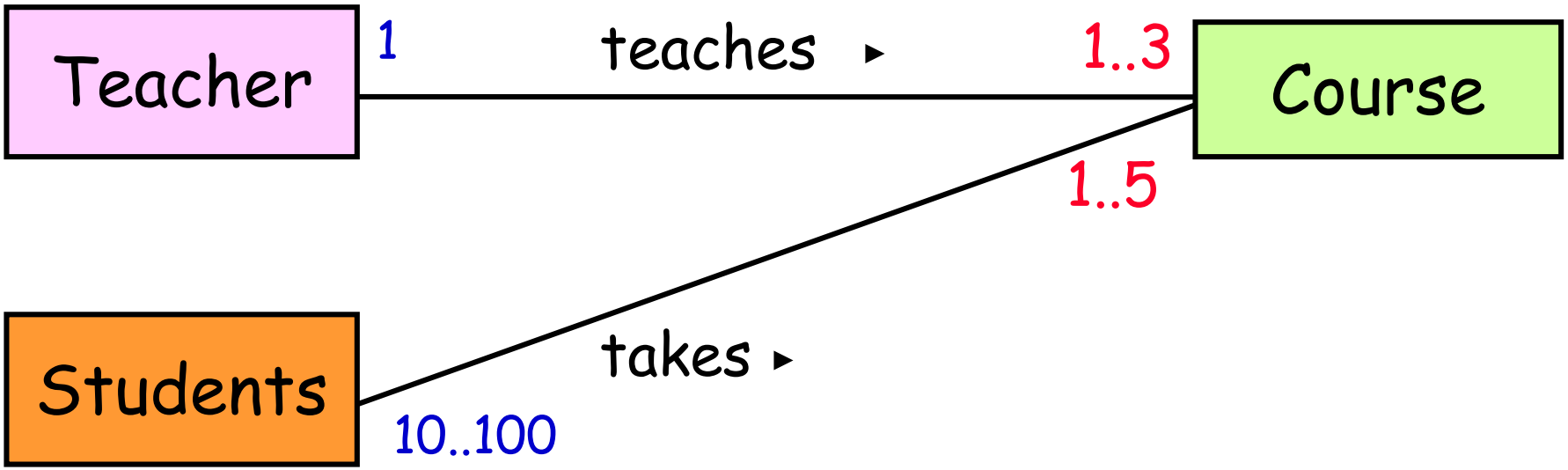
Association - More Examples



Multiplicity: The number of objects from one class that relate with a single object in an associated class.

Association – Multiplicity

- A **teacher** teaches 1 to 3 **courses** (subjects)
- Each course is taught by only one teacher.
- A **student** can take between 1 to 5 courses.
- A course can have 10 to 100 students.



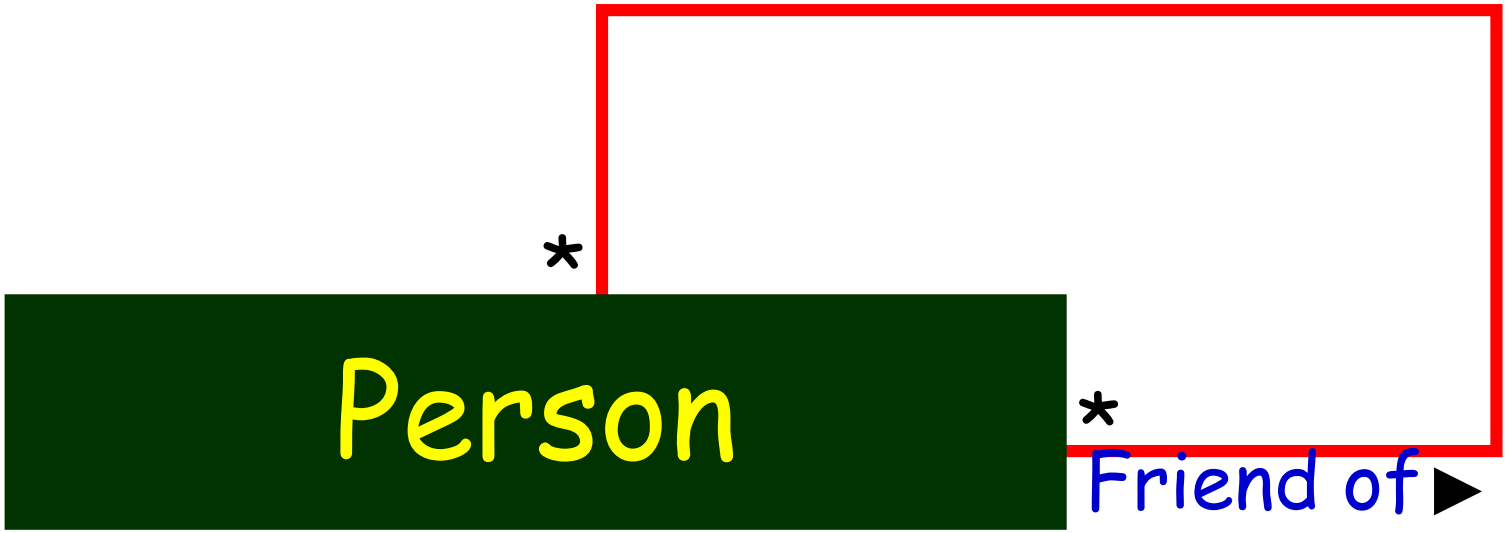
Association and Link

- **A link:**
  - An instance of an association
  - Exists between two or more objects
  - **Dynamically created and destroyed as the run of a system proceeds**

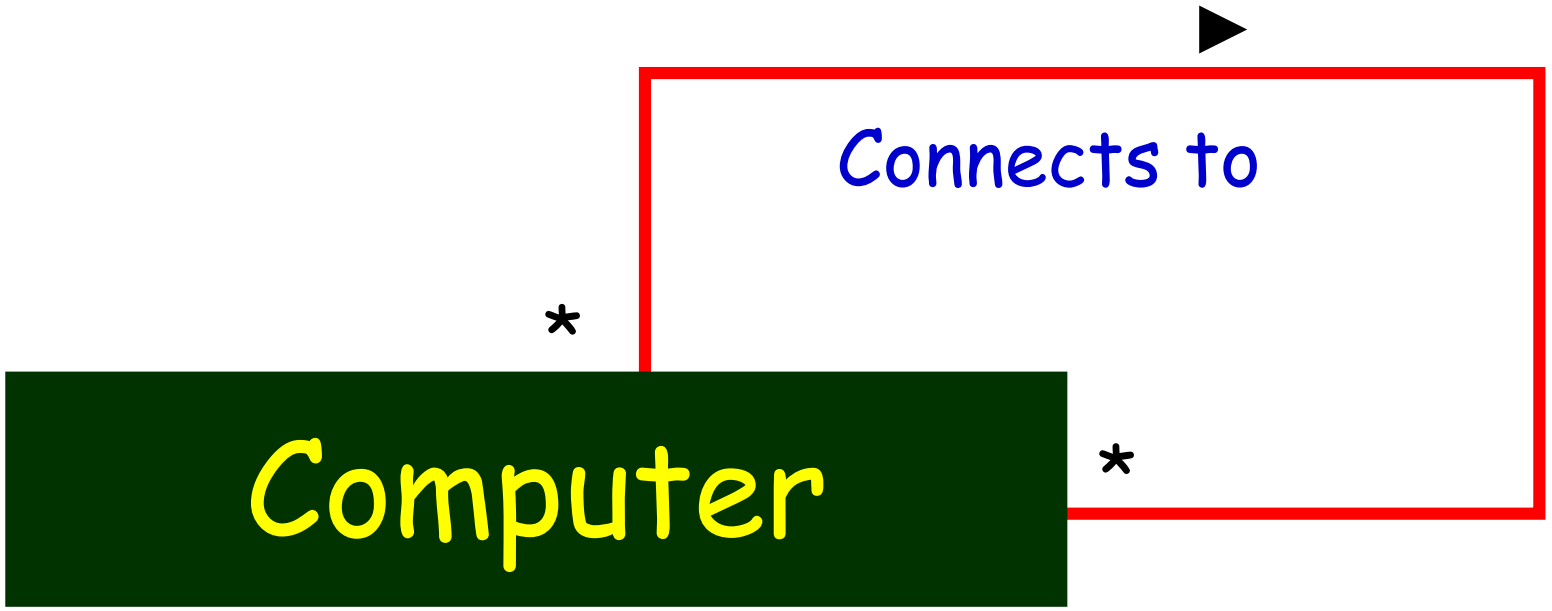
Association Relationship

- A class can be associated with itself (recursive association).
  - **Give an example?**
- An arrowhead used along with name:
  - Indicates direction of association.
- Multiplicity indicates # of instances taking part in the association.

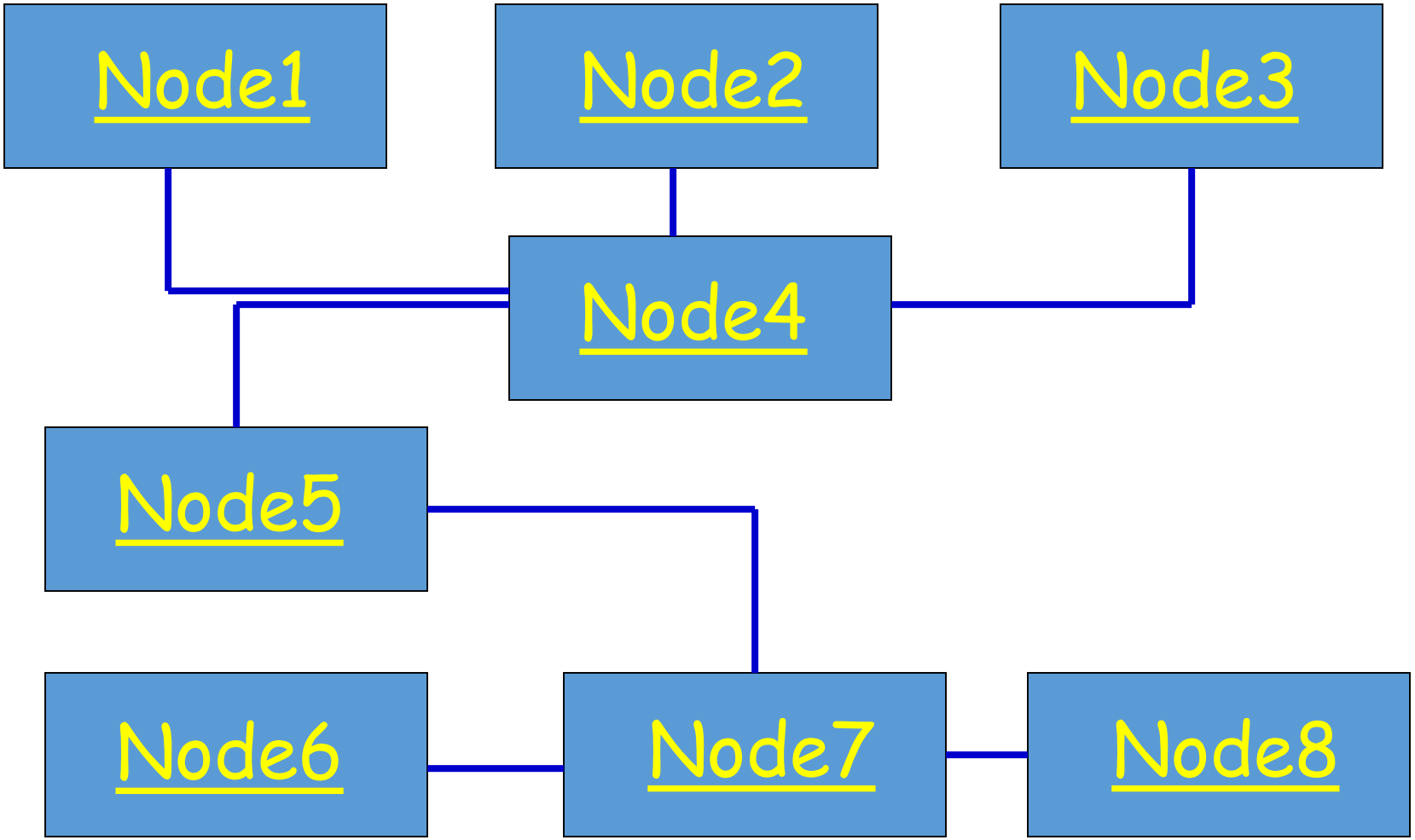
Self Association: Example



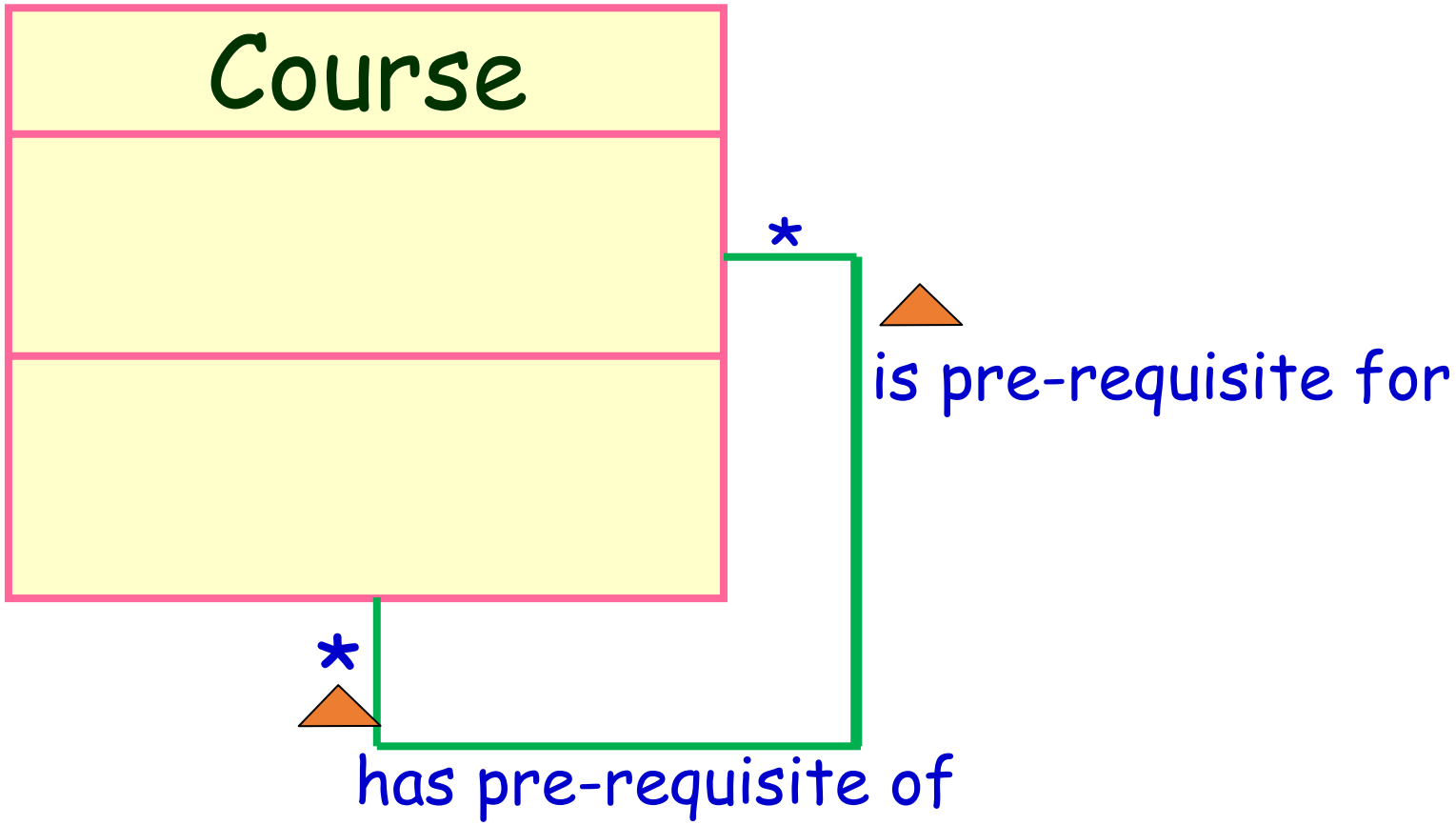
Self Association: Example Computer Network



Computer Network: Object Diagram

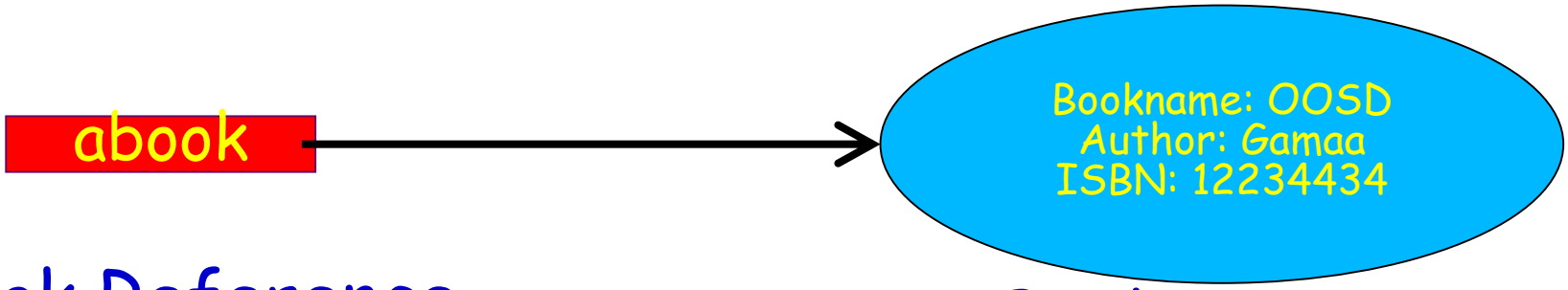


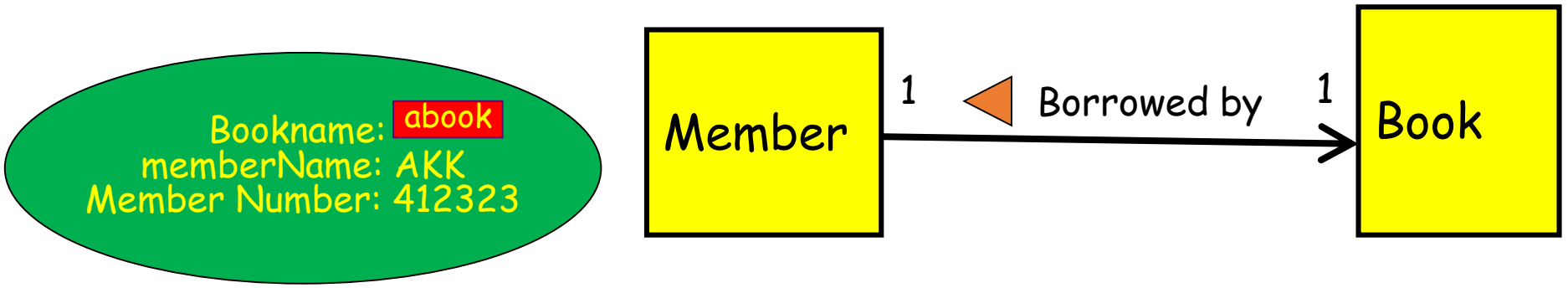
Association: Example



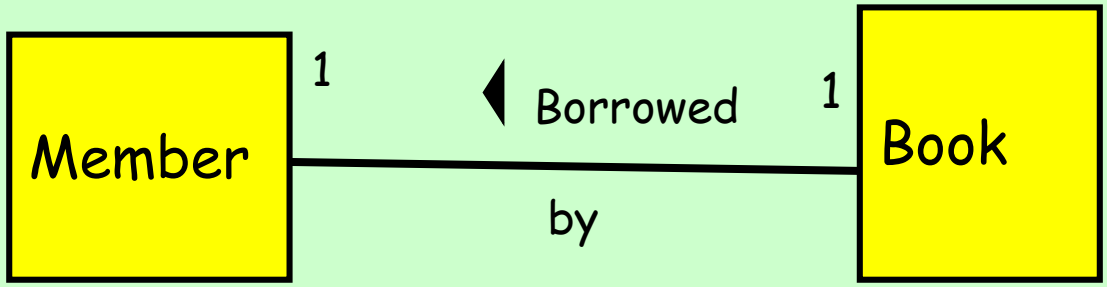
Implementing Association Relationship: Example 1

- To implement in Java:
  - Use a reference variable of one class as an attribute of another class

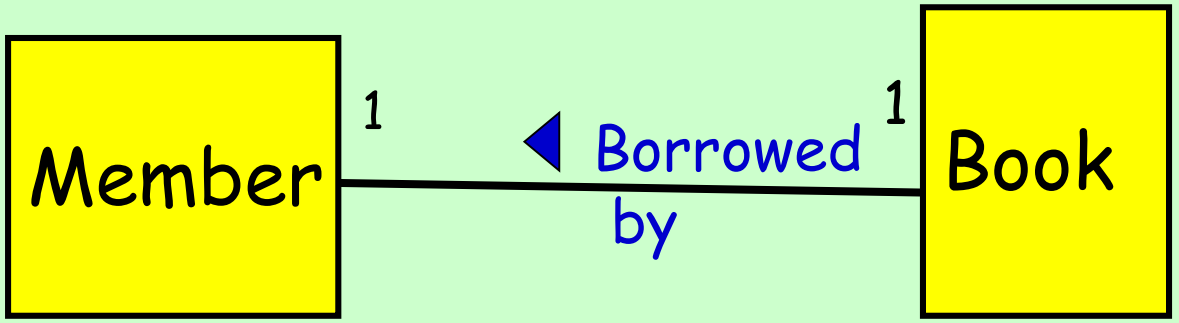




```
public class Member{
private Book book;
public issueBook(Book abook){
    setBook(abook);
    abook.setLender(this);
    }
    setBook(Book abook){
    book=abook;
  }
  …
}
```

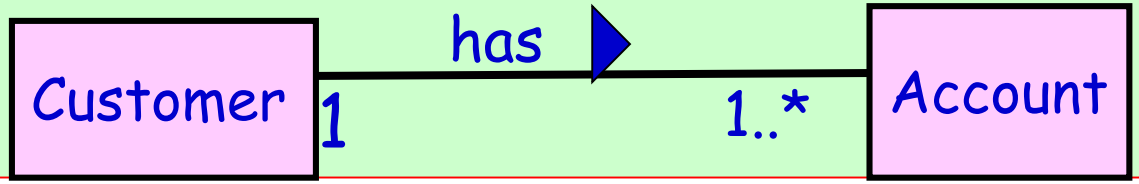


```
public class Book{
private Member member;
setLender(Member aLender){
    member=aLender;
  }
  …
}
```

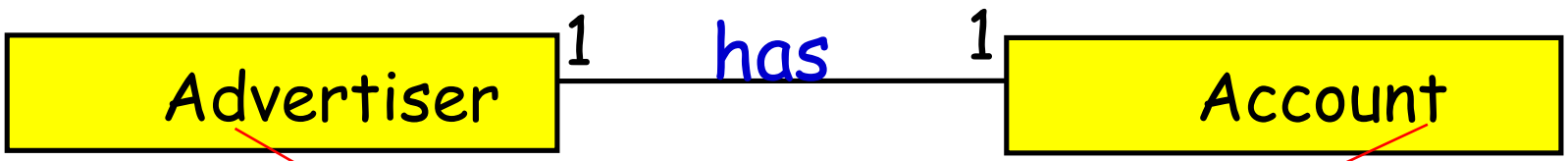


Code for Association Multiplicity

```
class Customer{

        private ArrayList <Account> accounts =
                        new ArrayList<Account>();

    public Customer() {

       Account defaultAccount = new Account();

         accounts.add(defaultAccount);

     }

}
```



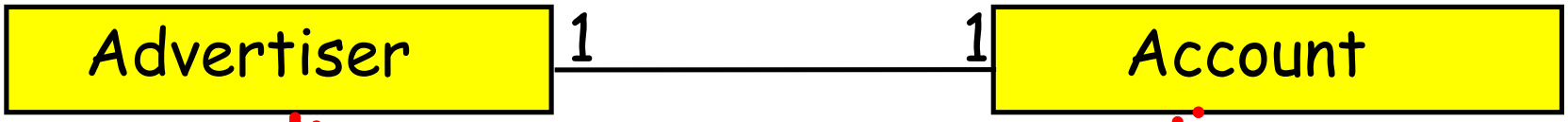
1-1 Association Example 3




```
public class Advertiser {
        private Account account;
        public Advertiser() {
                account = new Account(this);
        }
        public Account getAccount() {
                return account;
        }
}
```

code for Account

1-1 Association



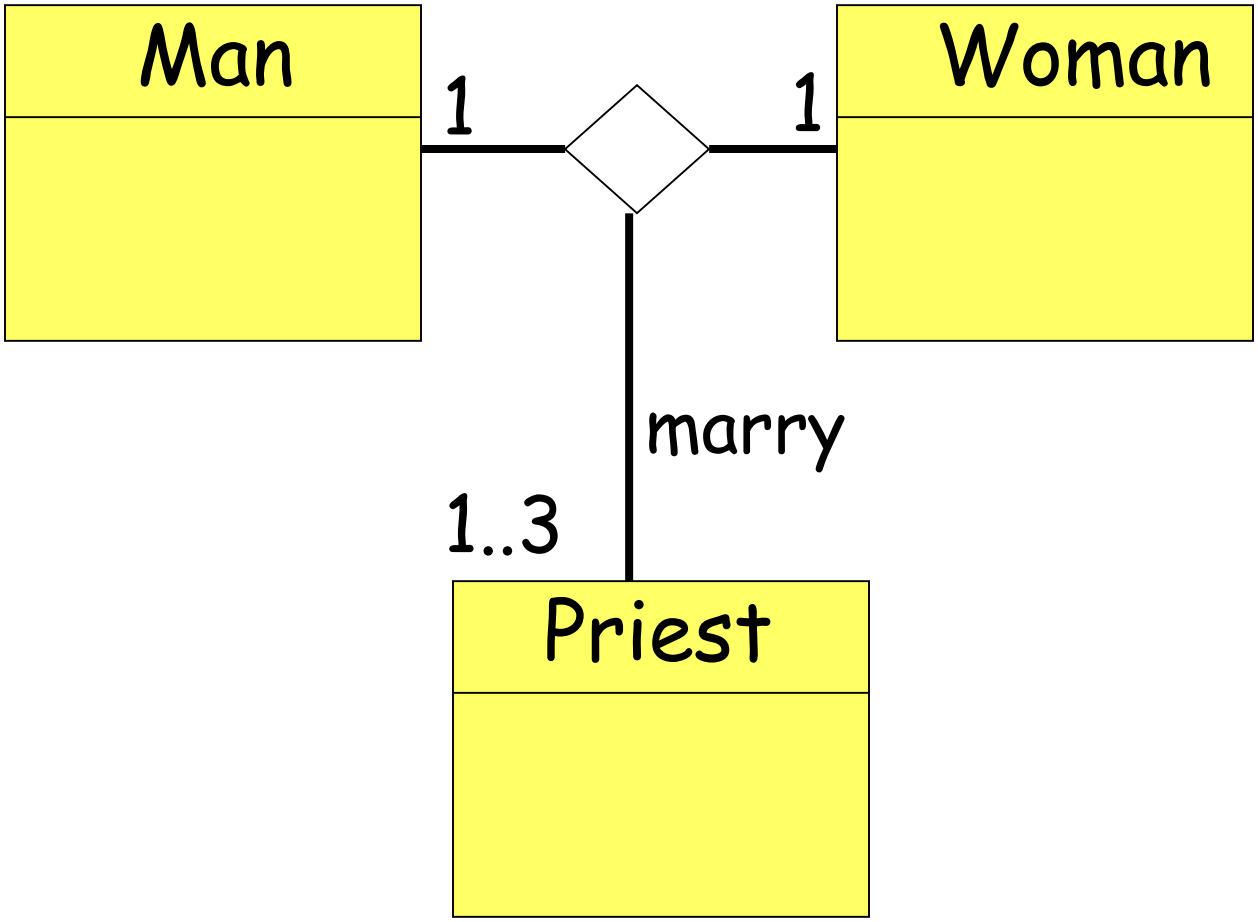

```
public class Advertiser {
    private Account account;
    public Advertiser() {
        account = new Account(this);
    }
    public Account getAccount() {
        return account;
    }
}
```

```
public class Account {
        private Advertiser owner;
    publicAccount(Advertiser owner)
    {
        this.owner = owner;
    }
    public Advertiser getOwner() {
        return owner;
    }
}
```

Ternary Association

- Some times three (or more) classes may be associated:
  - In this case an association end represents the potential number of values at that end when the values at the other end is kept fixed.

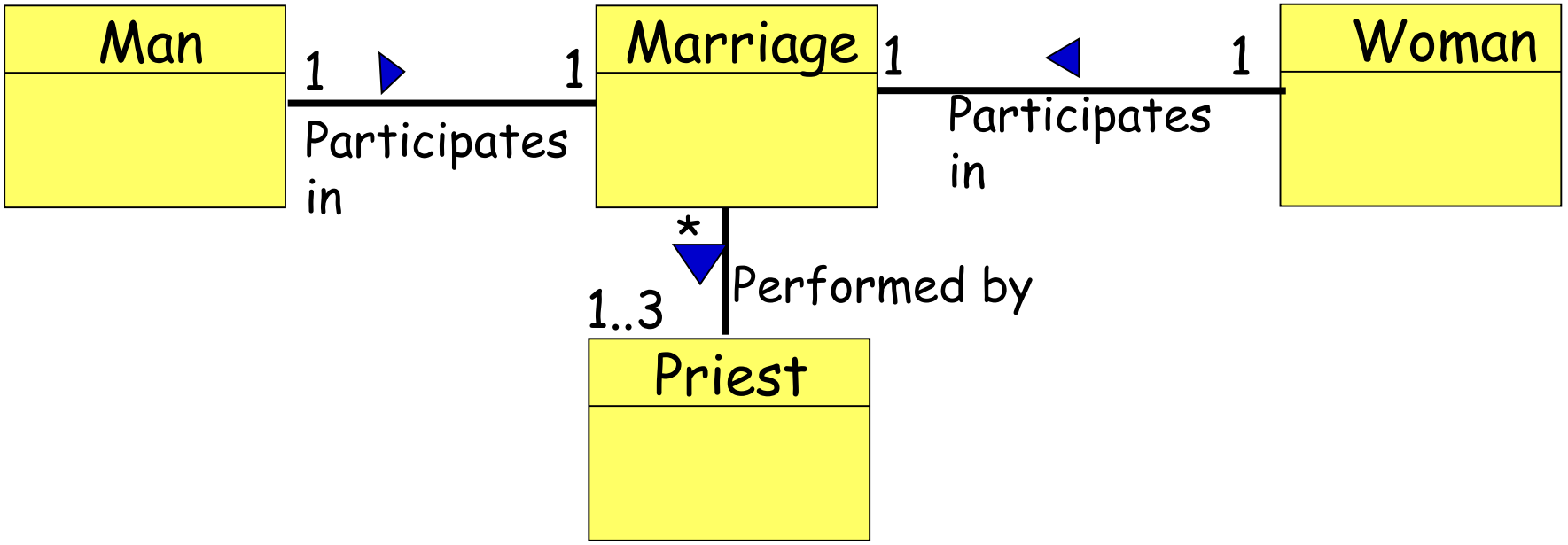
Ternary Association



and we can add more classes to the diamond...

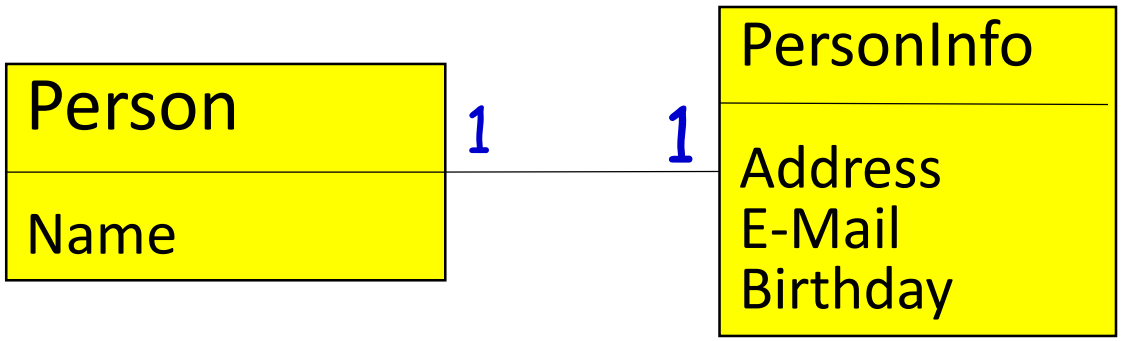
Implementation of Ternary Association

- Dcompose it into a set of binary associations.

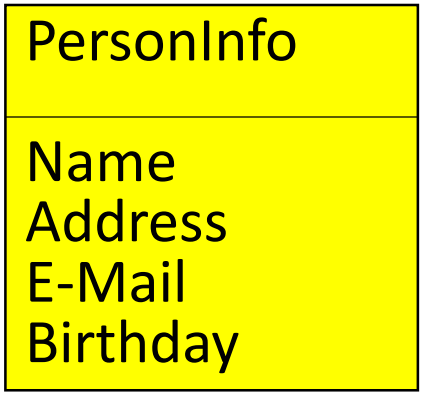


Overdoing Associations

- **Avoid unnecessary Associations**

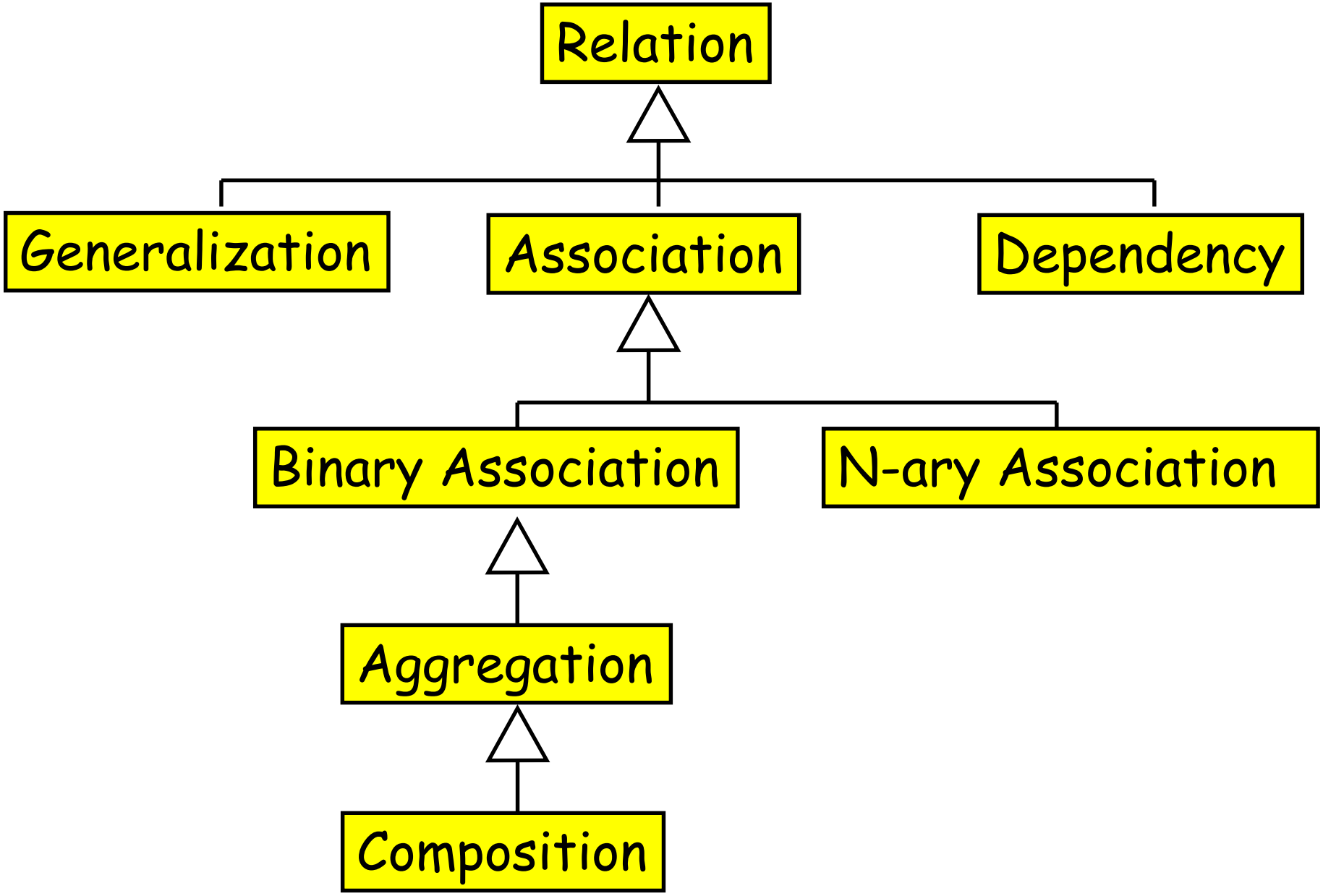


Avoid This…



Do This

Types of Class Relationships

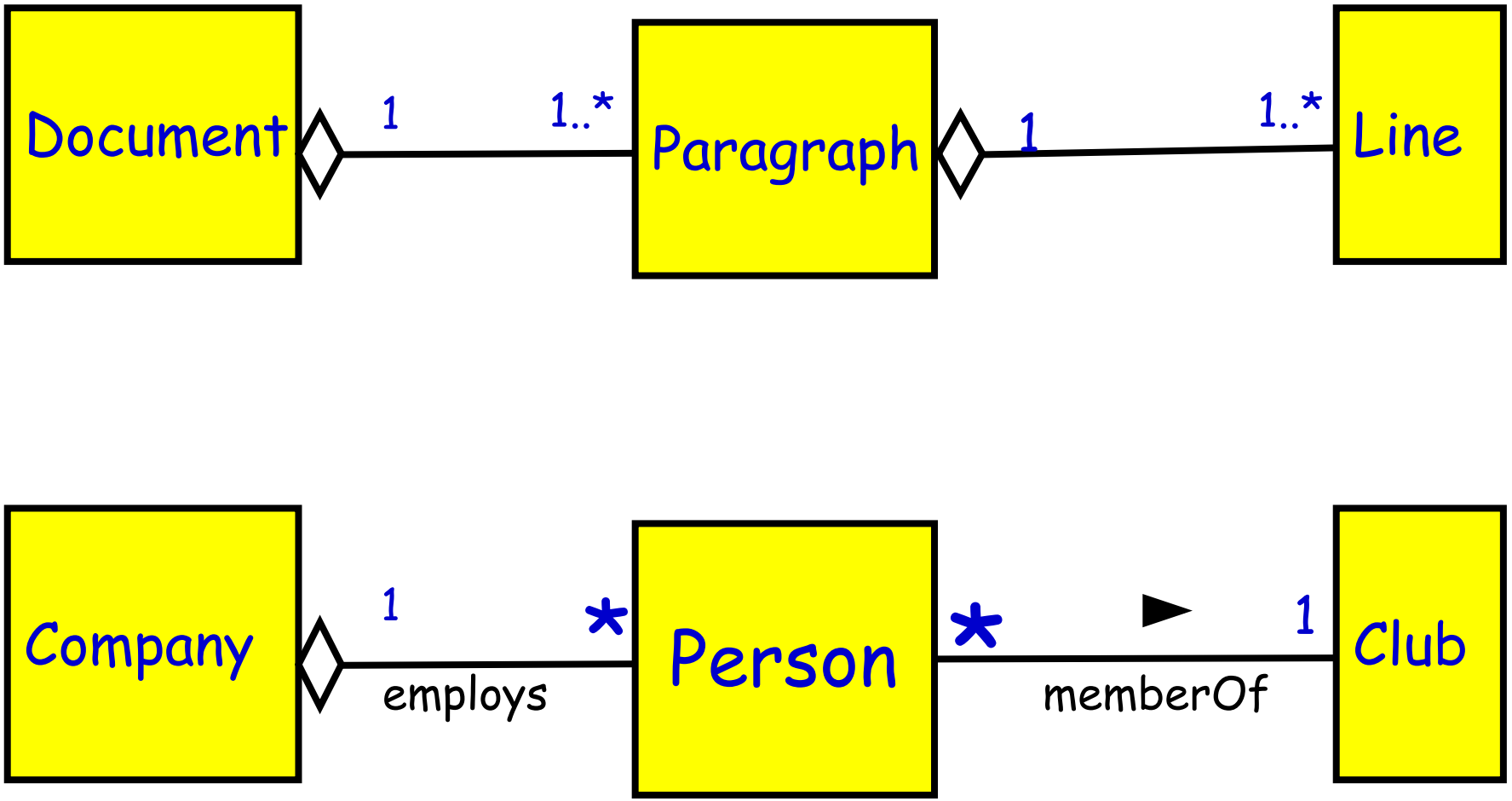


Aggregation Relationship

- Association: when objects of a class need services from other objects
  - Shown by a line joining classes
  - Multiplicity can be represented

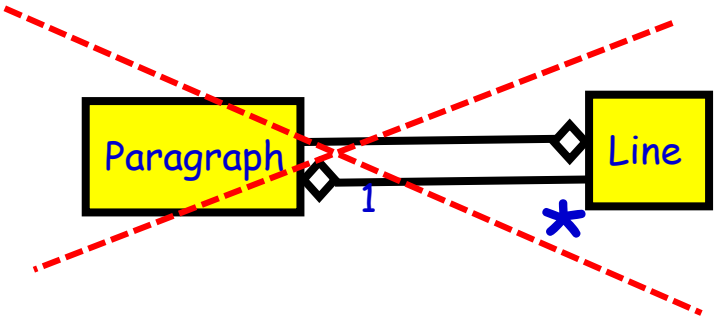
- Aggregation: when an object is **composed** of other objects
  - Captures part-whole relationship
  - Shown with a diamond connecting classes

Aggregation Relationship



Aggregation cont…

- An aggregate object contains other objects.
- Aggregation limited to tree hierarchy:
  - No circular inclusion relation.

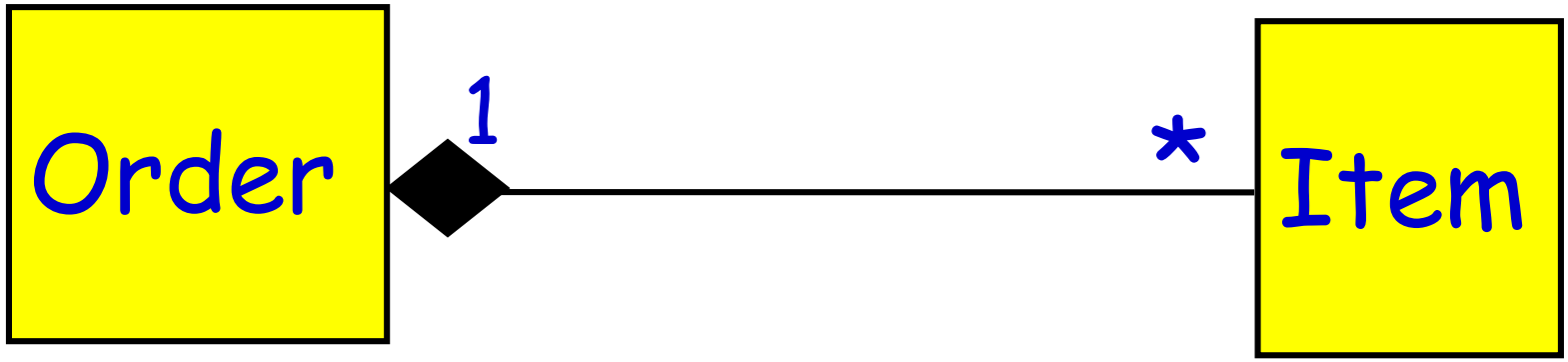


Composition

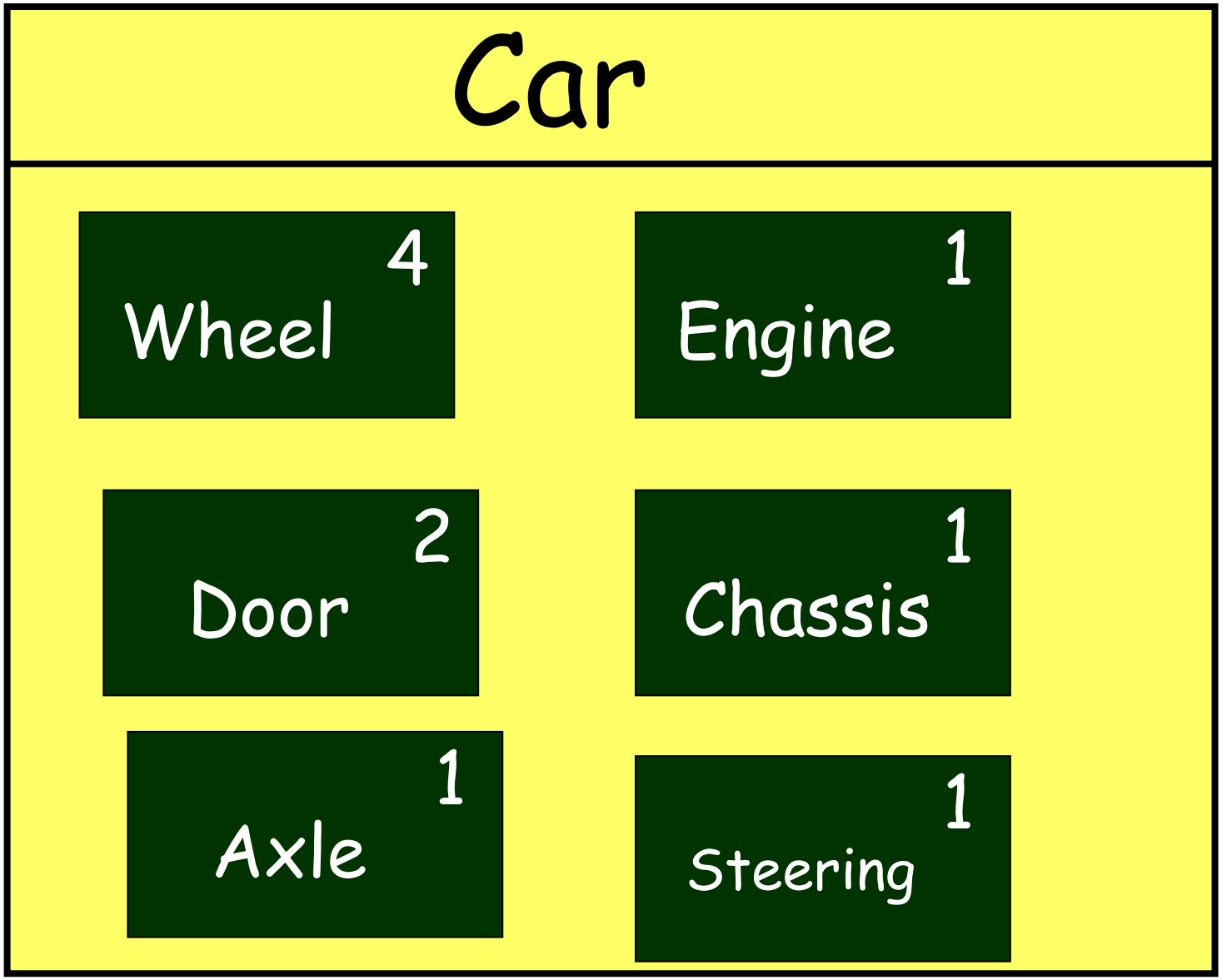
- A stronger form of aggregation
  - The whole is the sole owner of its part.
    - A component can belong to only one whole
  - The life time of the part is dependent upon the whole.
    - The composite must manage the creation and destruction of its parts.

Composition Relationship

- Life of item is same as that of order

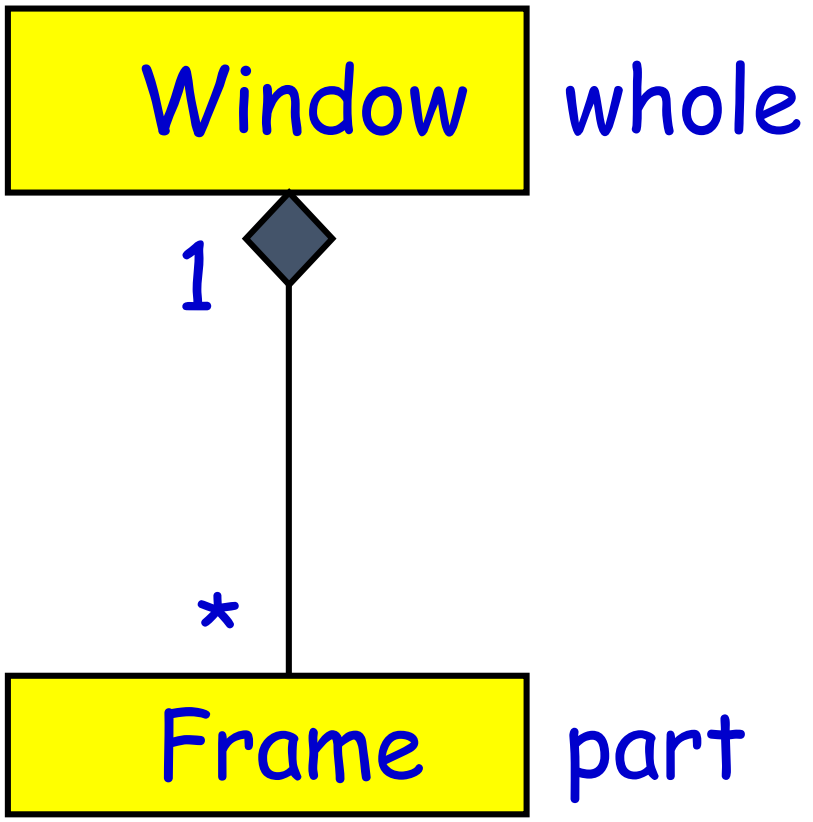


Composition: Alternate Notation



Composition

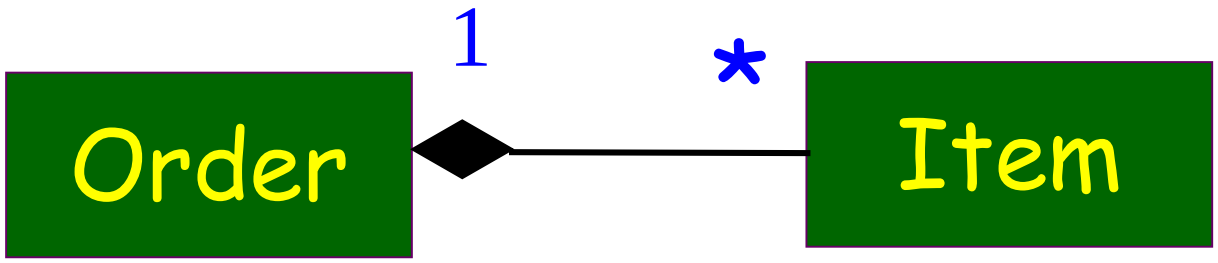
- An object may be a part of ONLY one composite at a time.
  - Whole is responsible for the creation and disposition of its parts.



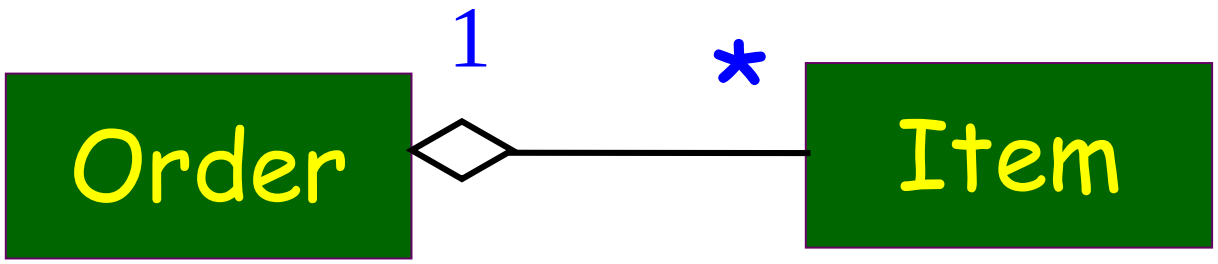
Aggregation vs. Composition

- **Composition**:
  - Composite and components have the same life line.
- **Aggregation**:
  - Lifelines are different.
- Consider an order object:
  - Aggregation: If order items can be changed or deleted after placing order.
  - Composition: Otherwise.

Composition versus Aggregation

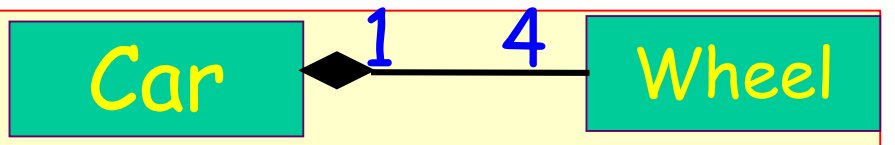


Composition



Aggregation

Implementing Composition




```
public class Car{
  private Wheel wheels[4];
  public Car (){
            wheels[0] = new Wheel();
            wheels[1] = new Wheel();
            wheels[2] = new Wheel();
            wheels[3] = new Wheel();
      }
  }
```

Representing Aggregation in Classes

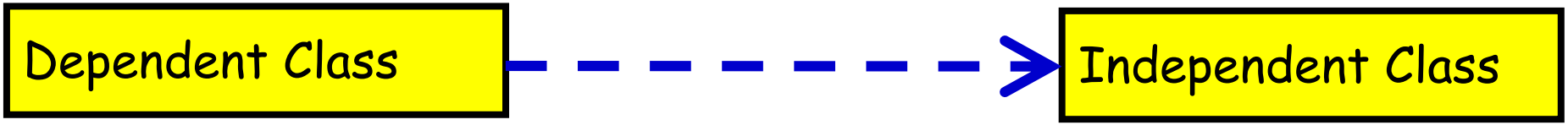
•An aggregation relationship is usually represented as a data field in the aggregated class.

```
public class Name {
  /* Data fields */
  /*Constructors */
  /* Methods */
}
```

```
public class Person {
  /** Data fields */
  private Name name;
  private Address address;

  /** Constructors */
  /** Methods */
}
```

```
public class
Address {
  /* Data fields */

/*Constructors*/
  /** Methods */
}
```

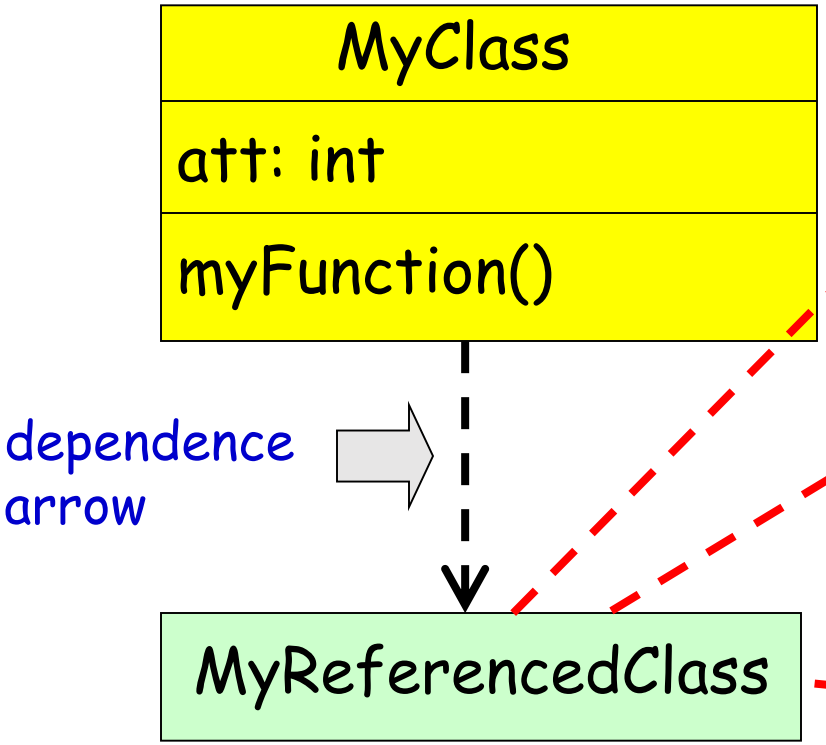
Class Dependency



Dependency

- Commonly Dependence may be caused by:
  - **Local variable**
  - **Parameter**
  - **Return value**

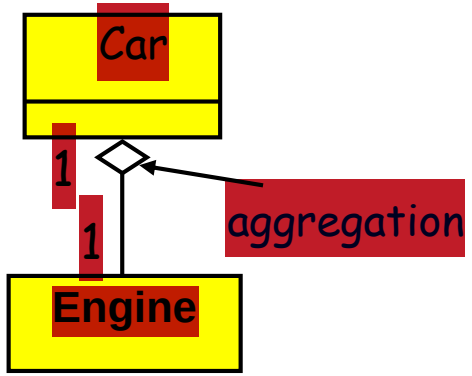
Dependence – Possible Implementations

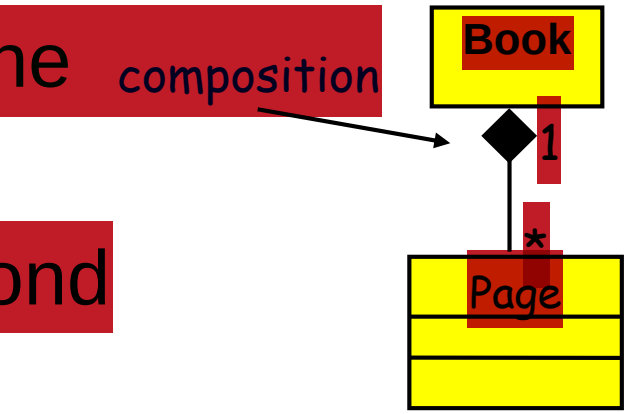


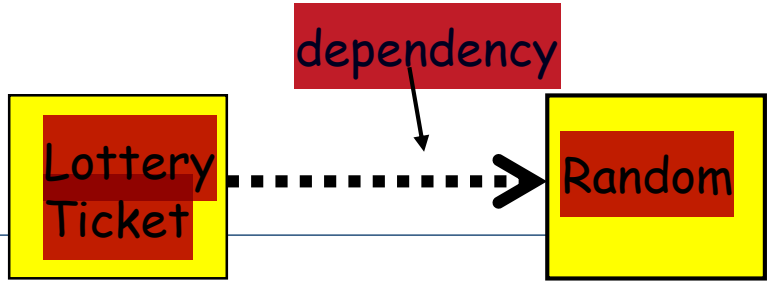

```
class MyDependentClass{

  void myFunction1(
  MyReferencedClass r ) { ..
  }

  MyreferencedClass
  myFunction2( .. ) { .. }

  void myFunction3( .. ){
  MyReferencedClass m .. }

}
```

Association Types

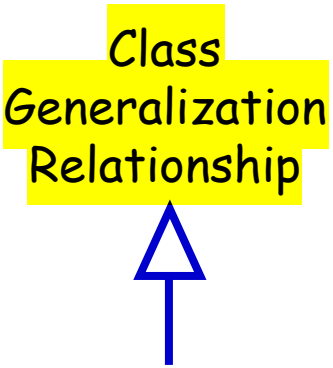
- **aggregation: "is part of"**
  - Symbolized by empty diamond
- **composition: "is made of"**
  - Stronger version of aggregation
  - The parts live and die with the whole
  - Symbolized by a filled diamond
- **dependency:** Depends on
  - Repres. by dotted arrow.

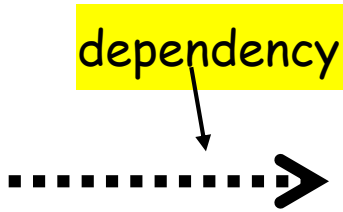


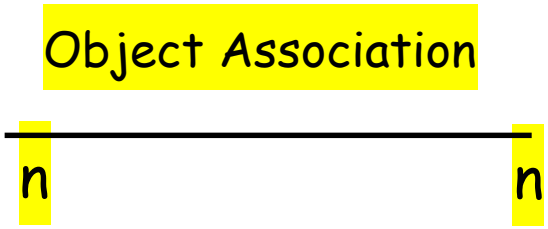


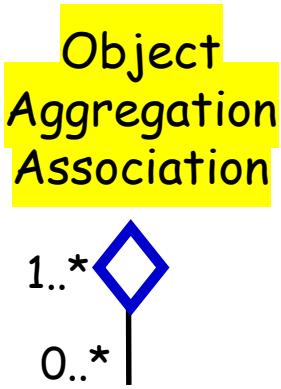


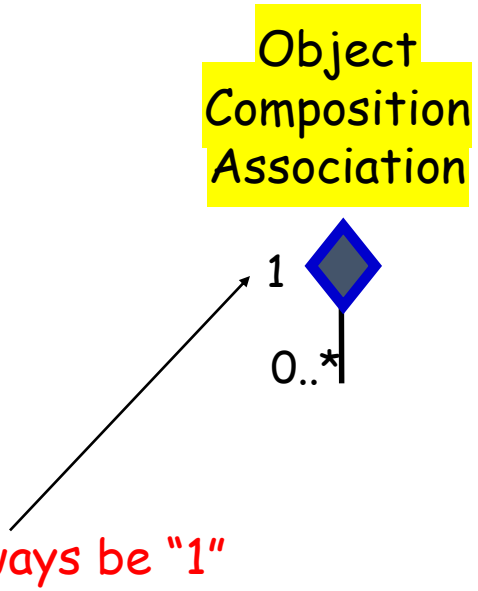
UML Class Relation Notation Summary









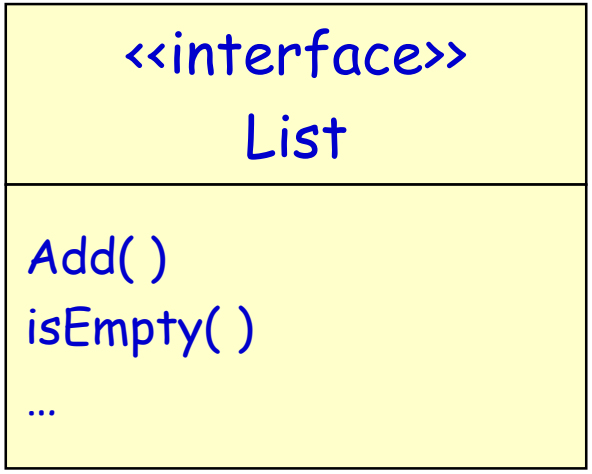


Class Relation Hints

- **Composition**
  - B is a permanent part of A
  - A contains B
  - A is a permanent collection of Bs
- **Subclass / Superclass**
  - A is a kind of B
  - A is a specialization of B
  - A behaves like B
- **Association (Collaboration)**
  - A delegates to B
  - A needs help from B
  - A and B are peers.

Interface

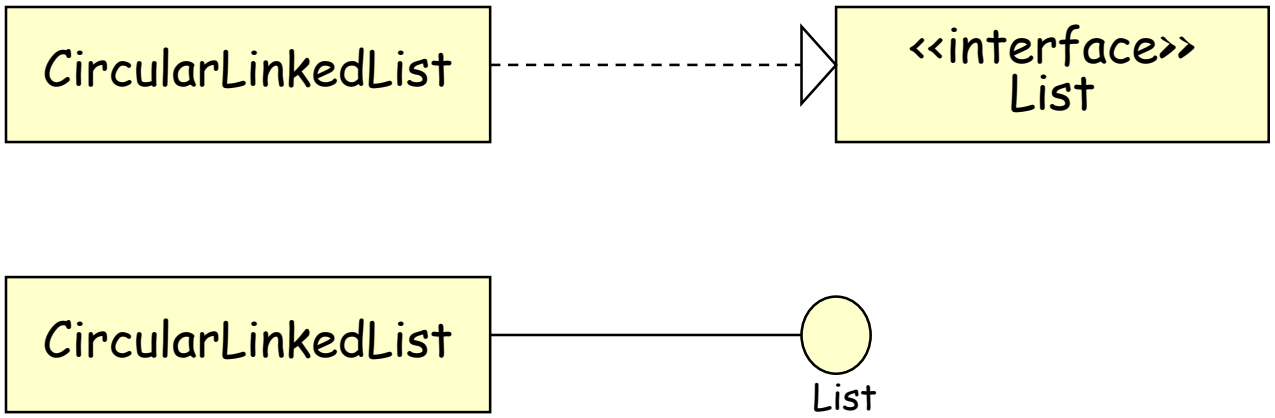
- An interface in UML is a named set of operations.
- Interfaces are used to characterize the behaviour of some classes.
  - **Shown as a stereotyped class.**



- Generalization can be defined between interfaces.

Realizing an Interface

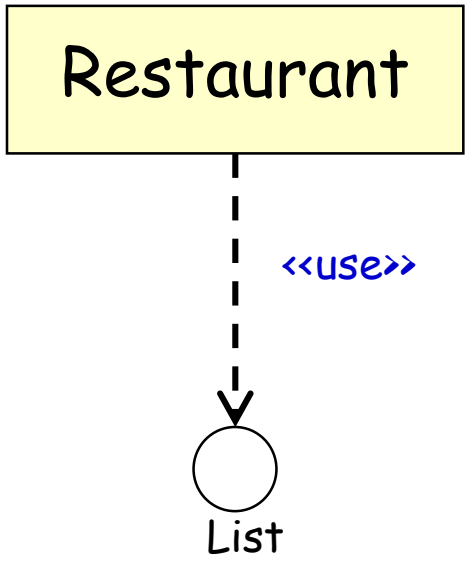
- A **class *realizes*** an **interface** if it provides implementations of all the operations.
  - Similar to the *implements* keyword in Java.
- UML provides two equivalent ways of denoting this relationship:



Both represent: "CircularLinkedList implements all the operations defined by the List interface".

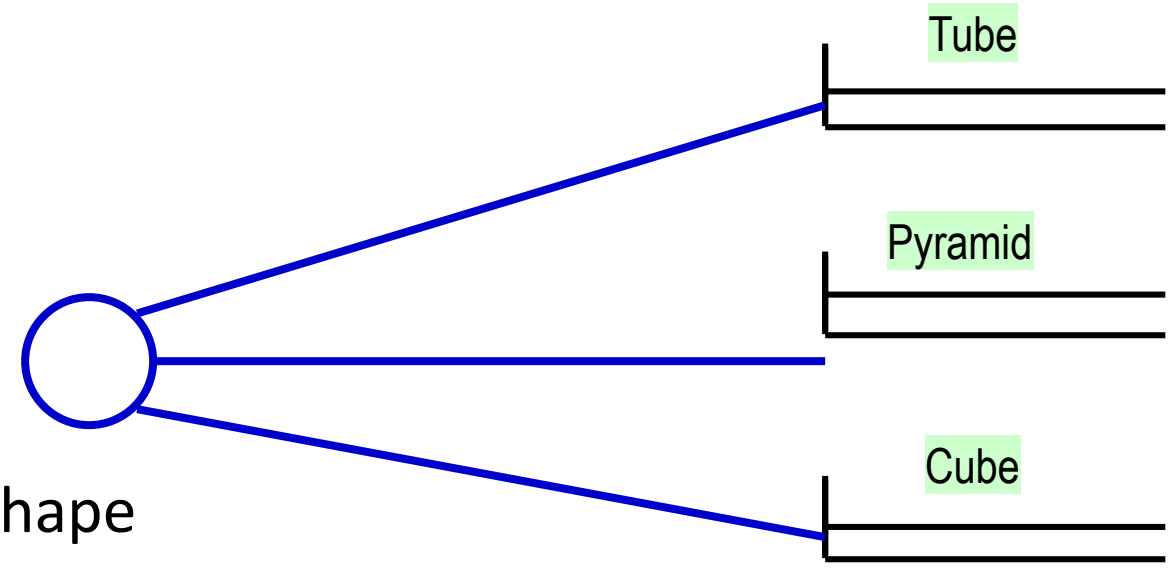
Interface Dependency

- A class can be dependent on an interface.
  - This means that it makes use of the operations defined in that interface.
  - E.g., the Restaurant class makes use of the **List** interface:

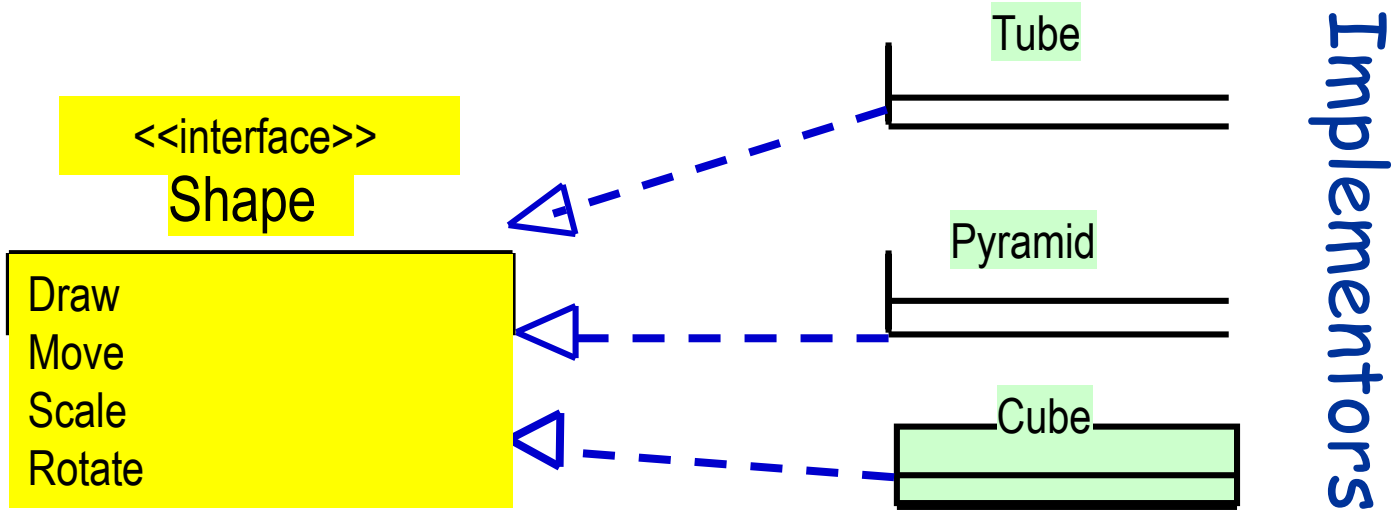


Interface Representations

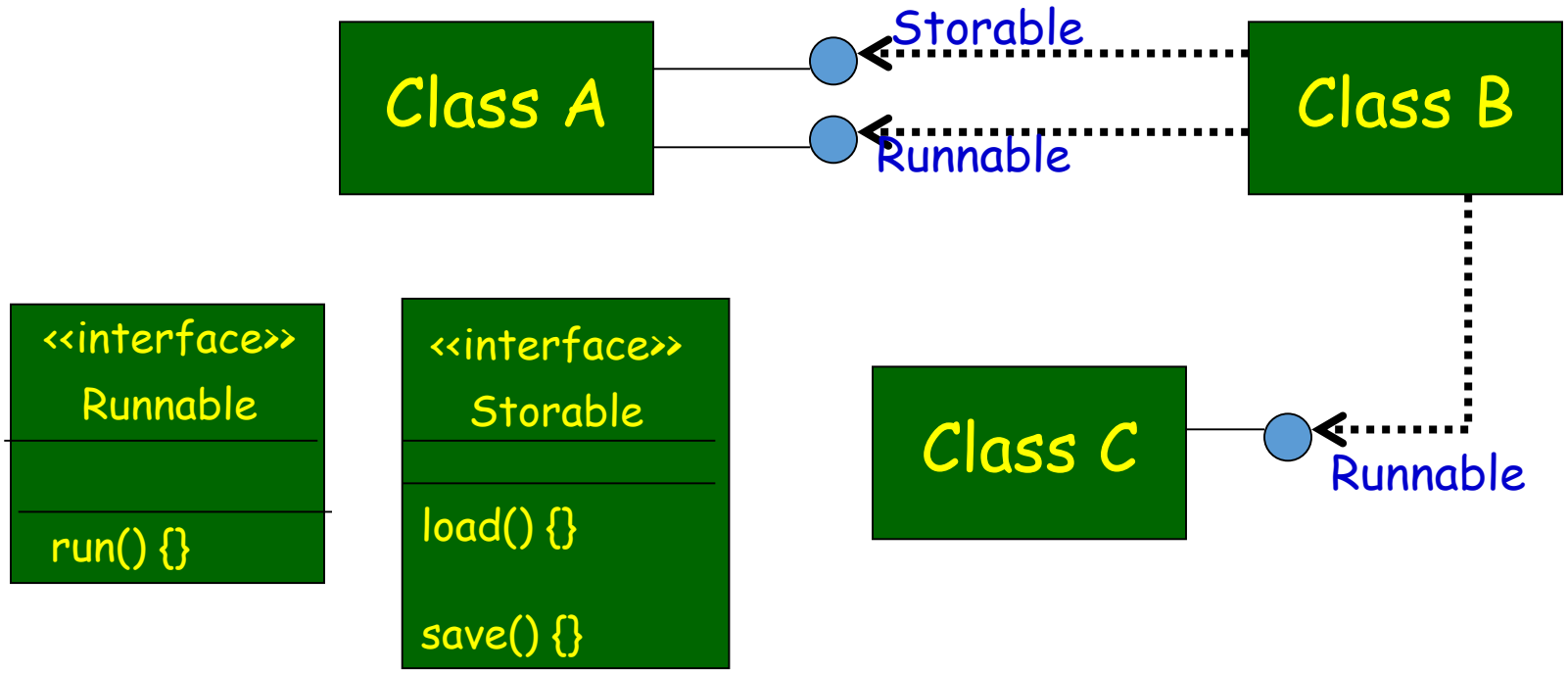
Alternate Representations



Service description



Interface Diagram Example



Class A implements the interfaces Runnable and Storable. Class C implements the interface Runnable. Class B uses the interface Runnable and Storable from A, and Runnable from C.

Advantages of Object-Oriented Development

- Code and design reuse
- Increased productivity
- Ease of testing (?) and maintenance
- Better understandability
- Elegant design:
  - Loosely coupled, highly cohesive objects
  - Essential for solving large problems.

Experience Report

- Initially incurs higher costs
  - After completion of some projects reduction in cost become possible
- Using well-established OO methodology and environment:
  - Projects can be managed with 20% -- 50% of traditional cost of development.

Disadvantages?

- **Non-locality of memory access:**
  - Data distributed across objects
  - Also, objects tend to have complex associations
  - Leads to poor memory access times.
- **Higher overheads:**
  - Object creation, destruction
  - Overhead due to encapsulation, etc.