



面向对象分析与设计

Object-Oriented Analysis and Design

北京理工大学软件学院

马 锐

Email: mary@bit.edu.cn

第2章 面向对象的基本概念

2.0 面向对象方法概述

2.1 对象

2.2 封装

2.3 类

2.4 抽象与分类

2.5 继承

2.6 聚合

2.7 关联

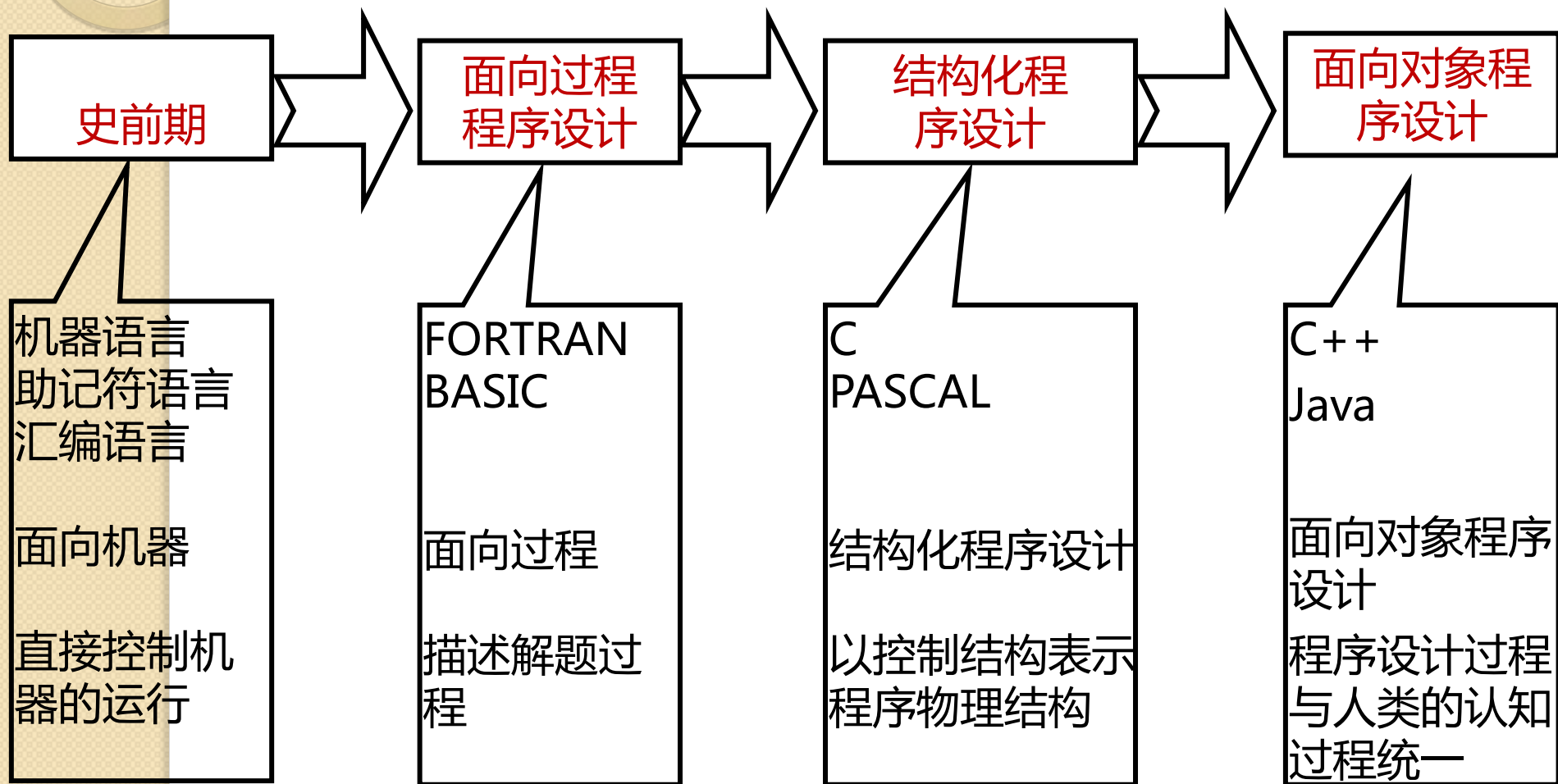
2.8 消息通信

2.9 多态性

2.10 复用

2.0 面向对象方法概述(1)

➤ 程序设计语言发展历史



2.0 面向对象方法概述(2)

示例：编写一个开发票程序计算各项明细

编号	名称	规格	单位	数量	单价	金额
合计						

非面向对象思路

定义数据结构

定义函数

面向对象思路

对象

一组属性
操作：发票总计

2.0 面向对象方法概述(3)

➤ 软件开发方法学

● 结构化方法学：20世纪80年代

- ◆ 自顶向下，逐步求精
- ◆ 采用结构化分析方法
- ◆ 建模概念不能直接映射到问题域
- ◆ 不能很好地适应需求变化
- ◆ 系统缺乏灵活性和可维护性

● 面向对象方法学：20世纪90年代

2.0 面向对象方法概述(4)

➤ 面向对象 (Object-Oriented) 方法

- 一种观察问题、分析问题、解决问题的新方法
- 不仅是一些具体的软件开发技术与策略，而且是一整套关于如何看待软件系统与现实世界的关系，用什么观点来研究问题并进行问题求解，以及如何进行系统构造的软件方法学
- 尽量运用人类的自然思维方式来构造软件系统

2.0 面向对象方法概述(5)

➤ 面向对象方法的优点

- 改变了人们认识世界的方式

- ◆ 从对象出发认识问题域

- ◆ 对象对应着问题域中的事物，其属性和操作分别刻画事物的性质和行为

- ◆ 对象之间的继承、聚合、关联和依赖关系反映了问题域中事物之间的关系

- 缩短了从客观世界到计算机的语言鸿沟

- 缩短了分析与设计间的鸿沟

2.0 面向对象方法概述(6)

- ◆ 面向对象开发过程的各个阶段都使用了一致的概念与表示法
- 有助于软件的维护与复用
 - ◆ 把易变的数据结构和部分功能封装在对象内并加以隐藏
 - 保证了对象行为的可靠性
 - 对它们的修改并不会影响其他的对象，有利于维护，对需求变化有较强的适应性
 - ◆ 封装性和继承性有利于复用对象

2.1 对象(1)

➤ 现实世界的对象

- 某个实际存在的事物，它可以是有形的(一辆汽车)，也可以是无形的(一项计划)
- 对象是构成世界的一个独立单位，它有自己的静态特征和动态特征

➤ 面向对象系统中的对象

- 是系统中用来描述客观事物的一个实体，它是构成系统的一个基本单位

2.1 对象(2)

- 一个对象由一组属性和对这组属性进行操纵的一组操作构成
 - **属性**：用来描述对象静态特征的一个数据项
 - **操作**：用来描述对象动态特征（行为）的一个动作序列
 - **对象标识**：对象的名字
 - ◆ 外部标识
 - ◆ 内部标识

2.1 对象(3)

➤ 示例：手机对象

- 名称

- 属性

- ◆ 生产厂商

- ◆ 型号

- ◆ 颜色

- 操作

- ◆ 电话

- ◆ 短信

- ◆ 照相

- ◆ 微博

我的手机

生产厂商：HTC

型号：G10

颜色：黑色

Telephone()

SMS()

Camera()

Blog()

2.2 封装(1)

- 把对象的属性和操作结合成一个独立的、不可分的实体，并尽可能隐蔽对象的内部细节。只是向外部提供接口，降低对象间的耦合度
- 信息隐藏
 - 外界不能直接存取对象的内部信息（属性）以及隐藏的内部控制，外部也不知道对象操作的内部实现细节
 - 对象对外界仅定义什么操作可被其他对象访问

2.2 封装(2)

➤ 意义

- 使对象能够集中而完整地描述并对应一个具体事物
- 体现了事物的相对独立性，使对象外部不能随意存取对象的内部数据，避免了外部错误对它的“交叉感染”
- 对象的内部的修改对外部的影响很小，减少了修改引起的“波动效应”
- 公开静态的、不变的操作，而把动态的、易变的操作隐藏起来

2.3 类(1)

➤ 类的概念

- 具有相同属性和操作的一组对象的集合
- 类用来创建对象，对象是类的一个实例
- 类为属于该类的全部对象提供了统一的抽象描述，其内部包括属性和操作两个主要部分

➤ 类与对象的关系

- 一个类的所有对象具有相同的属性，但各个对象的属性值不同，并随着程序的执行而变化
- 一个类的所有对象共同使用它们的类定义中给出的操作

2.3 类(2)

➤ 示例：手机类

● 属性

◆ 生产厂商

◆ 型号

◆ 颜色

● 操作

◆ 电话

◆ 短信

◆ 照相

◆ 微博

手机
生产厂商 型号 颜色
Telephone() SMS() Camera() Blog()

2.3 类(3)

➤ 在C++中实现类

封装

```
class Mobile{  
public:  
    Mobile():vendor(""),model(""),color("") {}  
    Mobile(string v,string m,string c):  
        vendor(v),model(m),color(c) { }  
    void Telephone();  
    void SMS(string msg);  
    void Camera();  
    void Blog(string msg);  
private:                                信息隐藏  
    string vendor,model,color;  
};
```


2.3 类(4)

➤ 共享数据与操作

```
class SavingsAccount{  
public:  
    static void Calculate(); .....  
private:  
    static double interestRate; .....  
};
```

SavingsAccount

Object a

.....

Object h

interestRate

Object i

.....

Object z

2.4 抽象与分类(1)

➤ 抽象

- 忽略事物中的个别非本质特征，只注意那些与当前目标有关的本质特征，从而抽取出事物的共性
- **过程抽象**：任何一个完成确定功能的操作序列，其使用者都可把它看作一个单一的实体(类中的操作)
- **数据抽象**：根据施加于数据之上的操作来定义数据类型，并限定数据的值只能由这些操作来修改和观察

2.4 抽象与分类(2)

- 面向对象中的抽象

- ◆ 客观事物 -> 对象 -> 类 -> 一般类

- 不同开发阶段需要进行不同程度的抽象

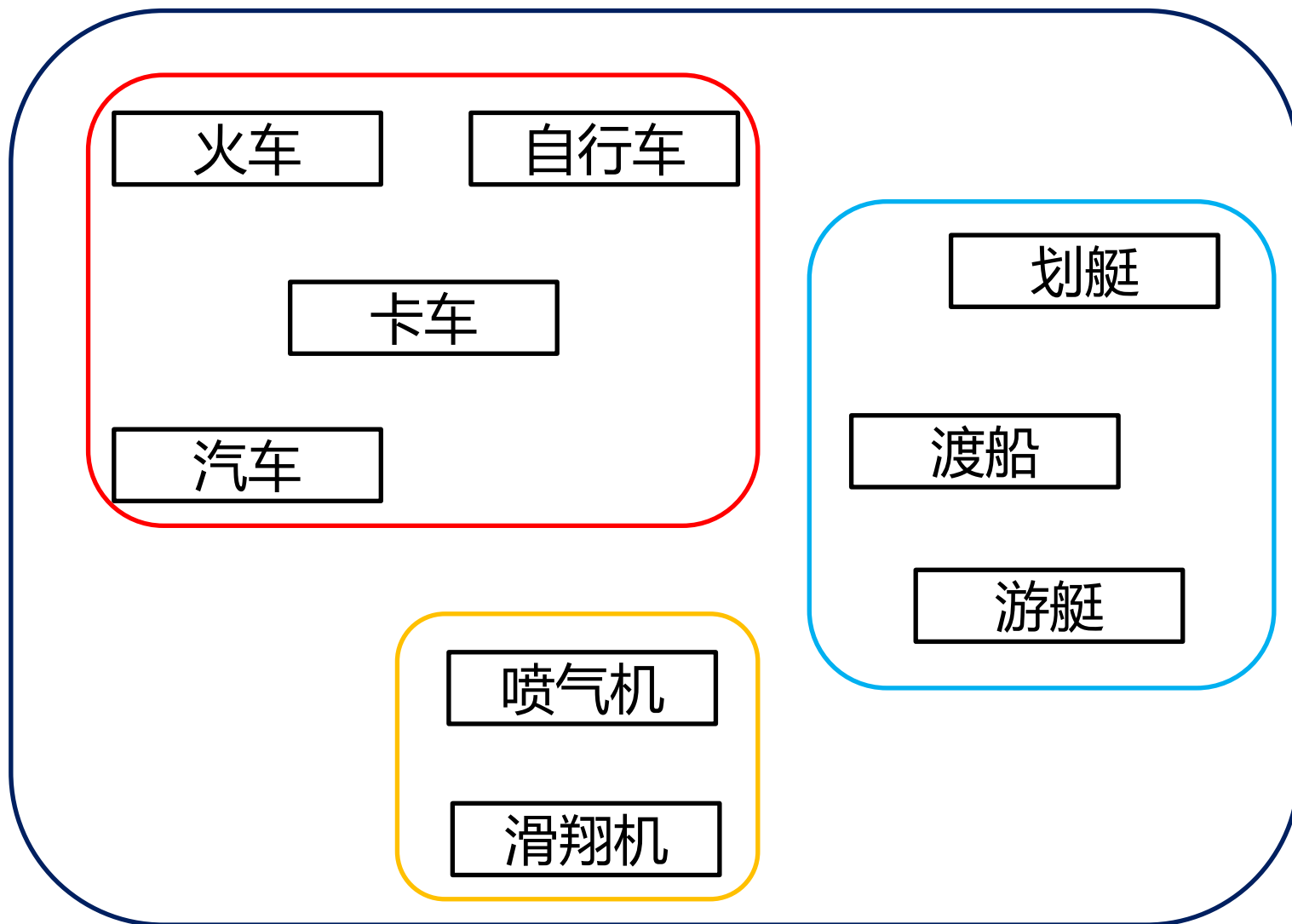
- 分类

- 把具有共同性质的事物划分为一类，得出一个抽象的概念

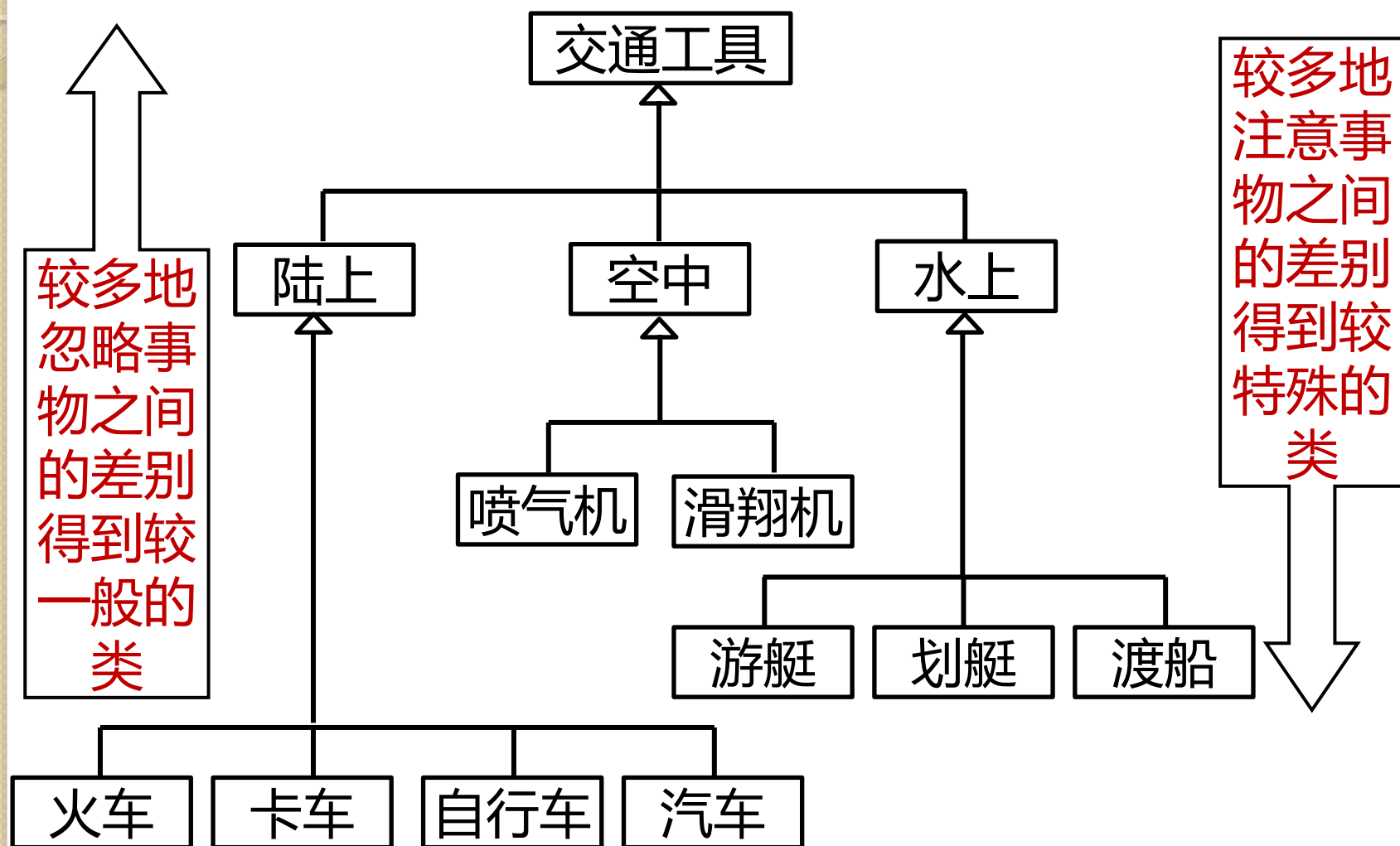
- 如果一个对象是分类（类）的一个实例，它将符合该分类的总体模式

- 不同程度的抽象可得到不同层次的分类

2.4 抽象与分类(3)



2.4 抽象与分类(4)



2.5 继承(1)

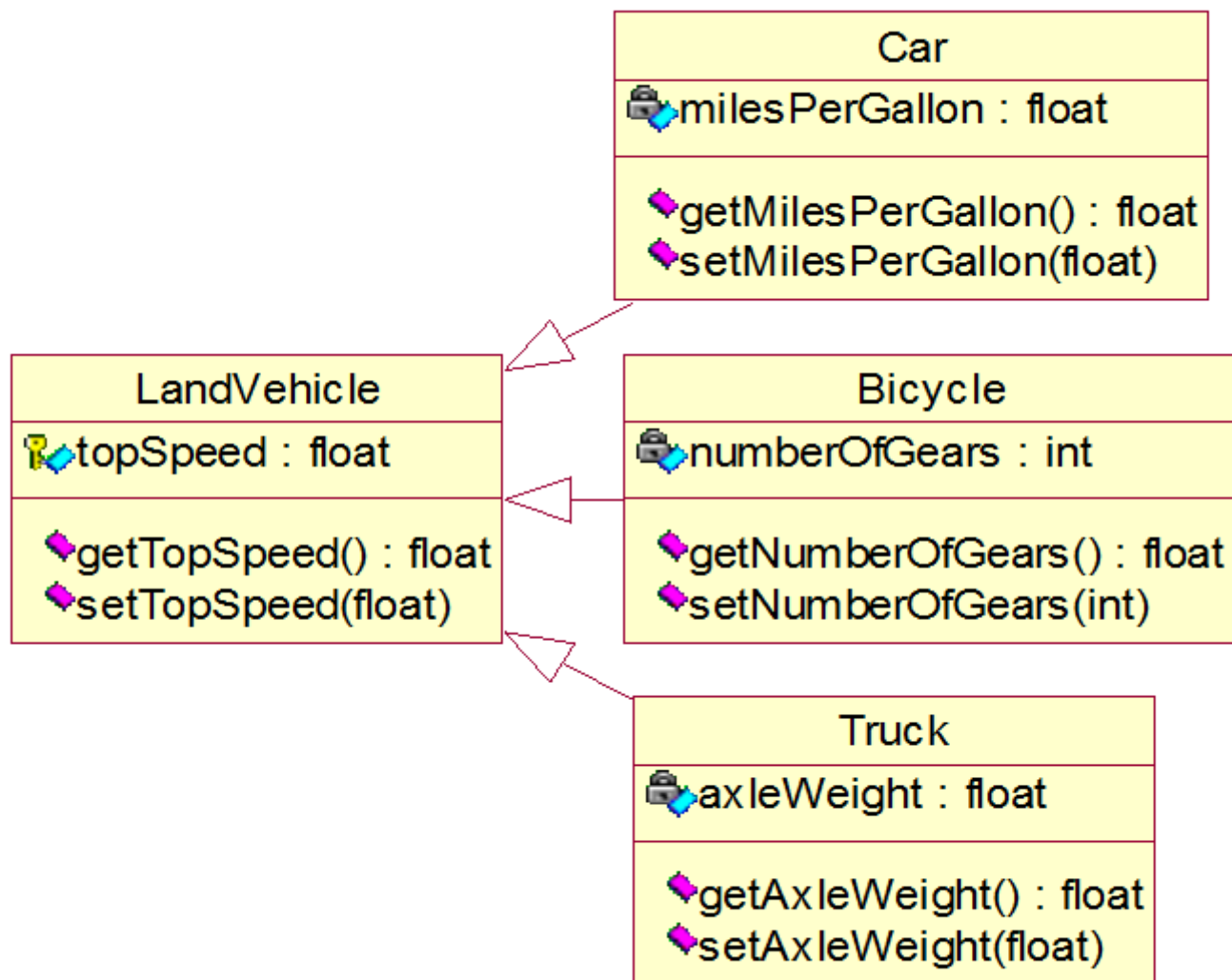
➤ (公有) 继承

- 特殊类拥有其一般类的全部属性与操作，称作特殊类对一般类的继承
- 继承可以指定类（子类）从父类中获取一些特性，再添加它自己的独特特性
- 继承具有传递性
- 继承有利于代码复用

➤ 继承的语义

- is a kind of
- A是B，其中A是子类，B是父类

2.5 继承(2)

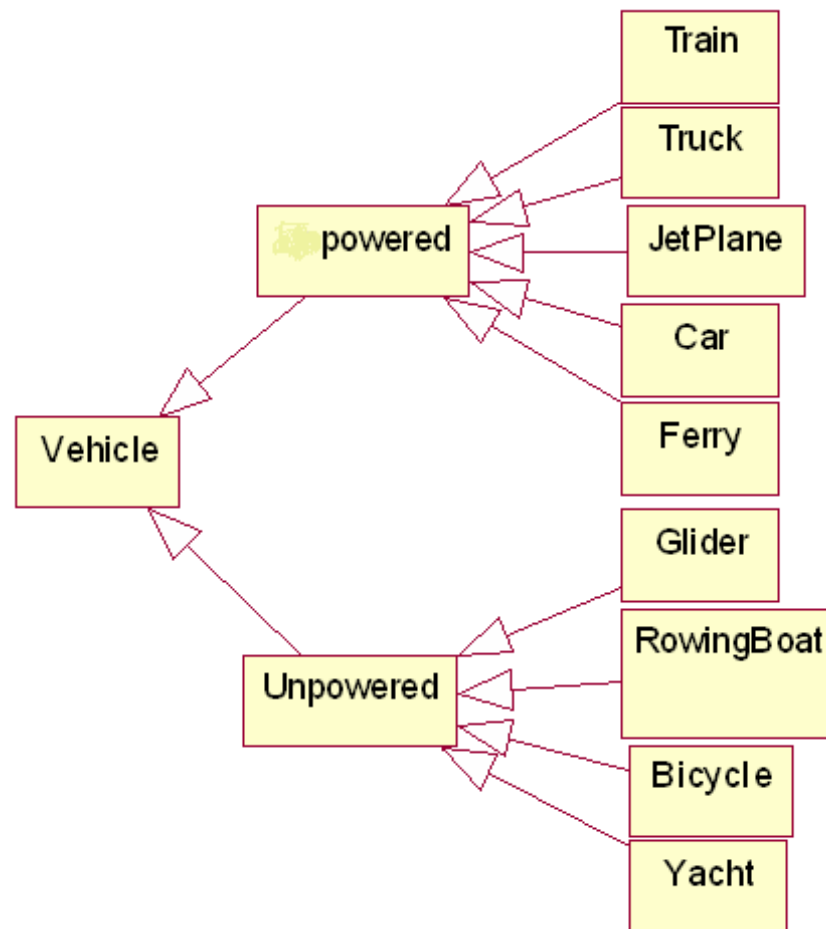
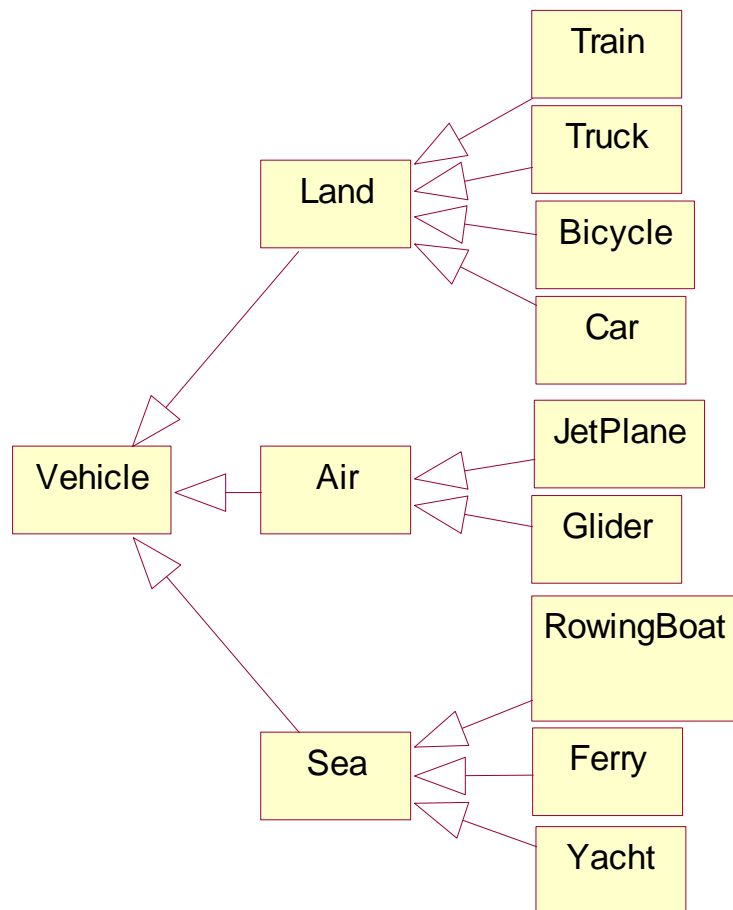


2.5 继承(3)

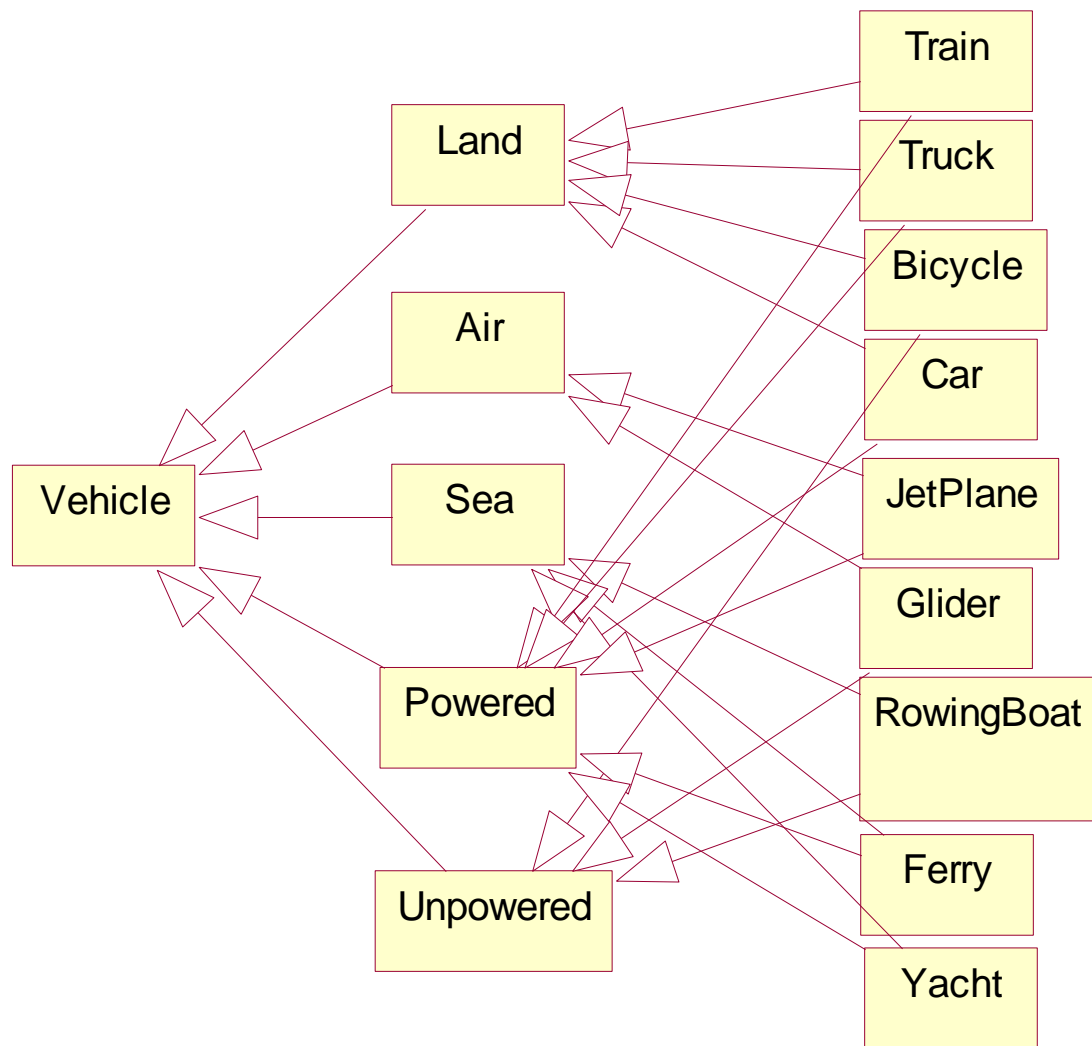
➤ 在C++中实现继承

```
class LandVechile{
public:
    double getTopSpeed();
    .....
private:
    double topSpeed;
};
class Train : public LandVechile {
public:
    int getMaxCoach();
    .....
private:
    int coach;}
```


2.5 继承(4)

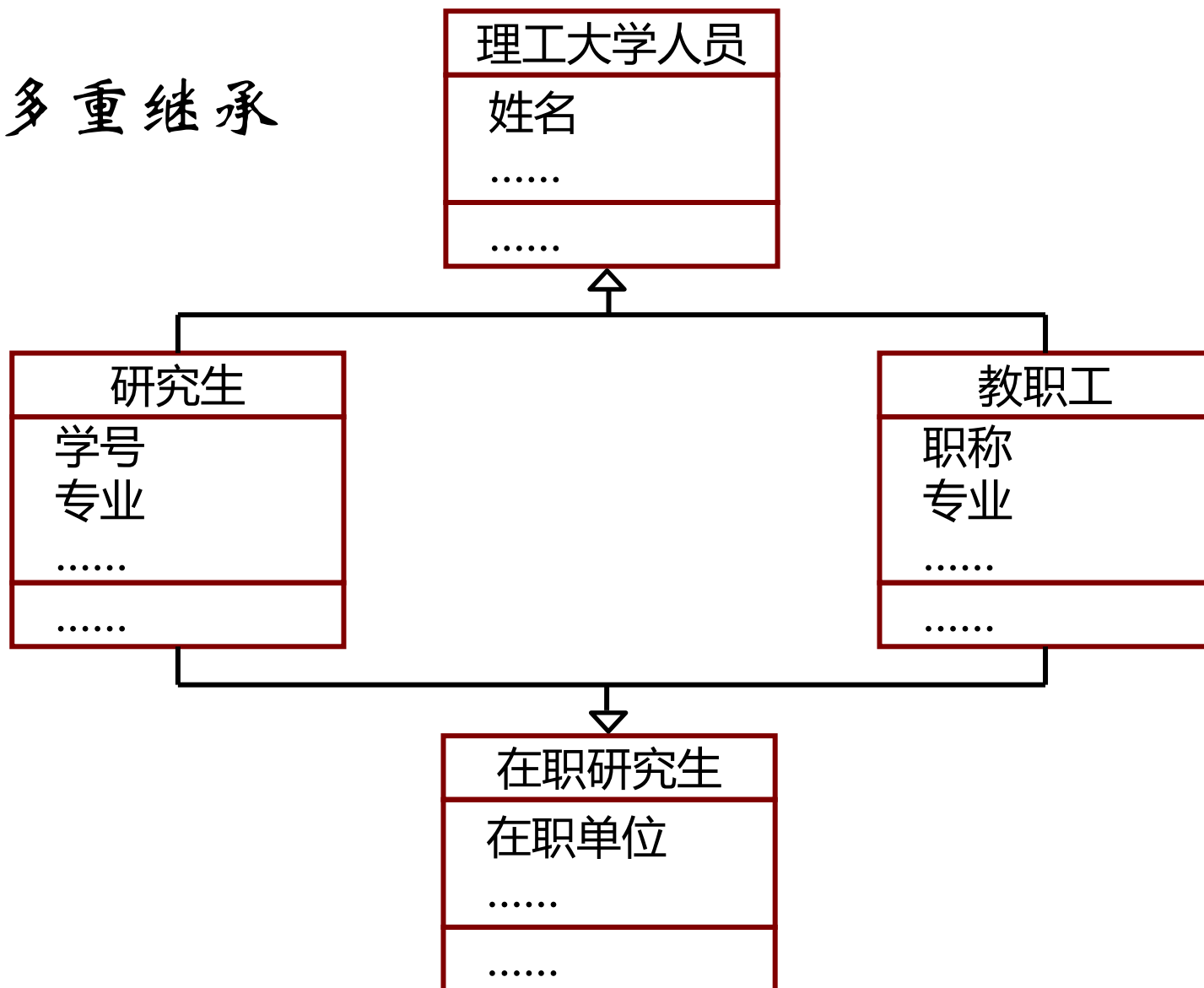


2.5 继承(5)



2.5 继承(6)

➤ 多重继承



2.5 继承(7)

● 优点

- ◆ 功能强大
- ◆ 接近真实情况
- ◆ 允许混合继承

在职研究生	
姓名	来自“理工大学人员”类
学号 专业	来自“研究生”类
职称 专业	来自“教职工”类
在职单位	“在职研究生”类新定义
.....	

2.5 继承(8)

- 缺点

- ◆ 比较复杂（对于设计人员和客户程序
员而言）
- ◆ 导致名称冲突
 - “专业”
- ◆ 导致重复继承
- ◆ 使编译器更难编写
- ◆ 使运行时系统更难编写

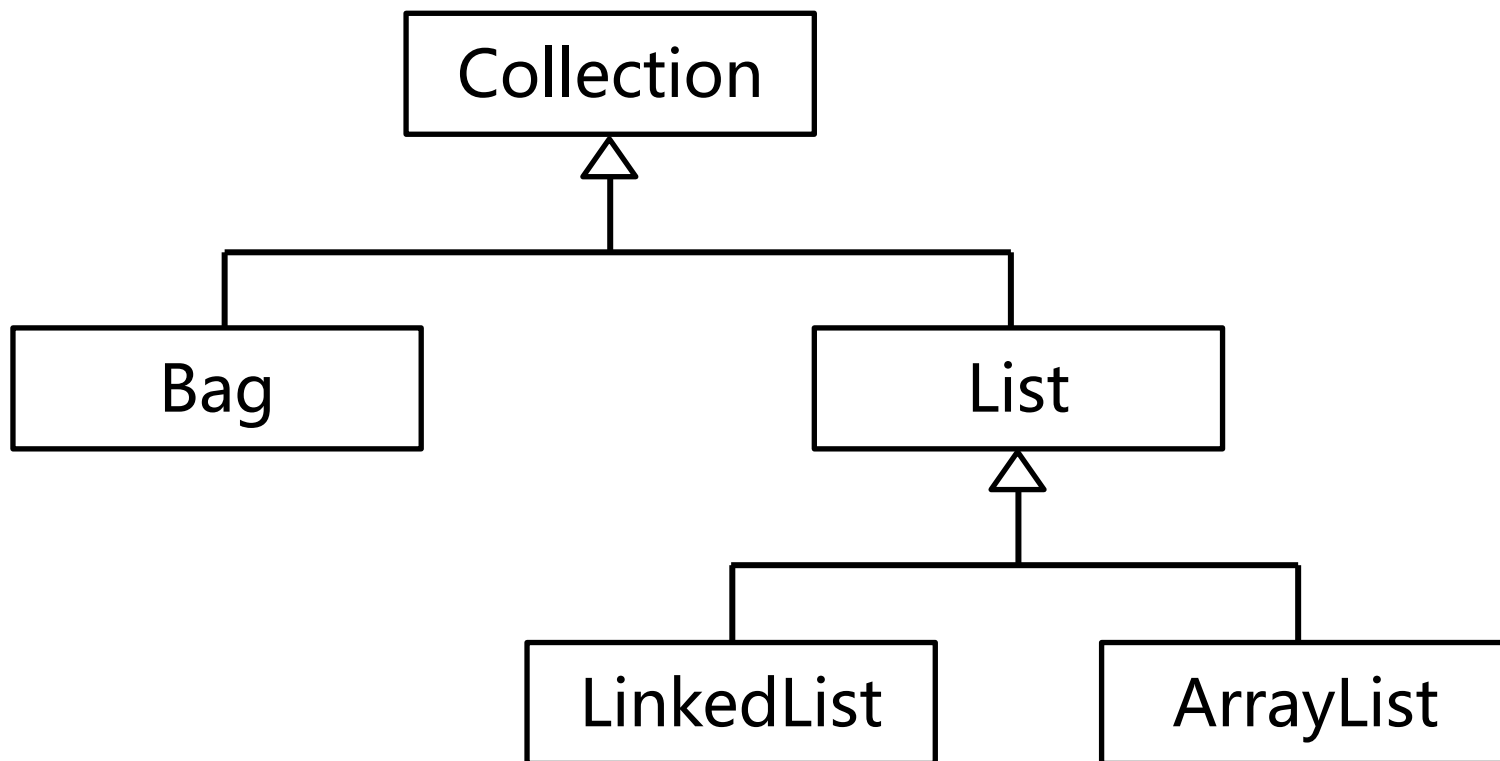
2.5 继承(9)

➤ 类层次结构

● 示例：为集合(Collection)建模

- ◆ **List**: 该集合可以把所有的对象按照插入的顺序放置
- ◆ **Bag**: 该集合中的对象没有排序
- ◆ **LinkedList**: 该集合中的对象使用序列对象进行排序，采用链表方式，更新速度快，但搜索速度较慢
- ◆ **ArrayList**: 该集合中的对象使用数组进行排序，搜索速度快，但更新速度慢

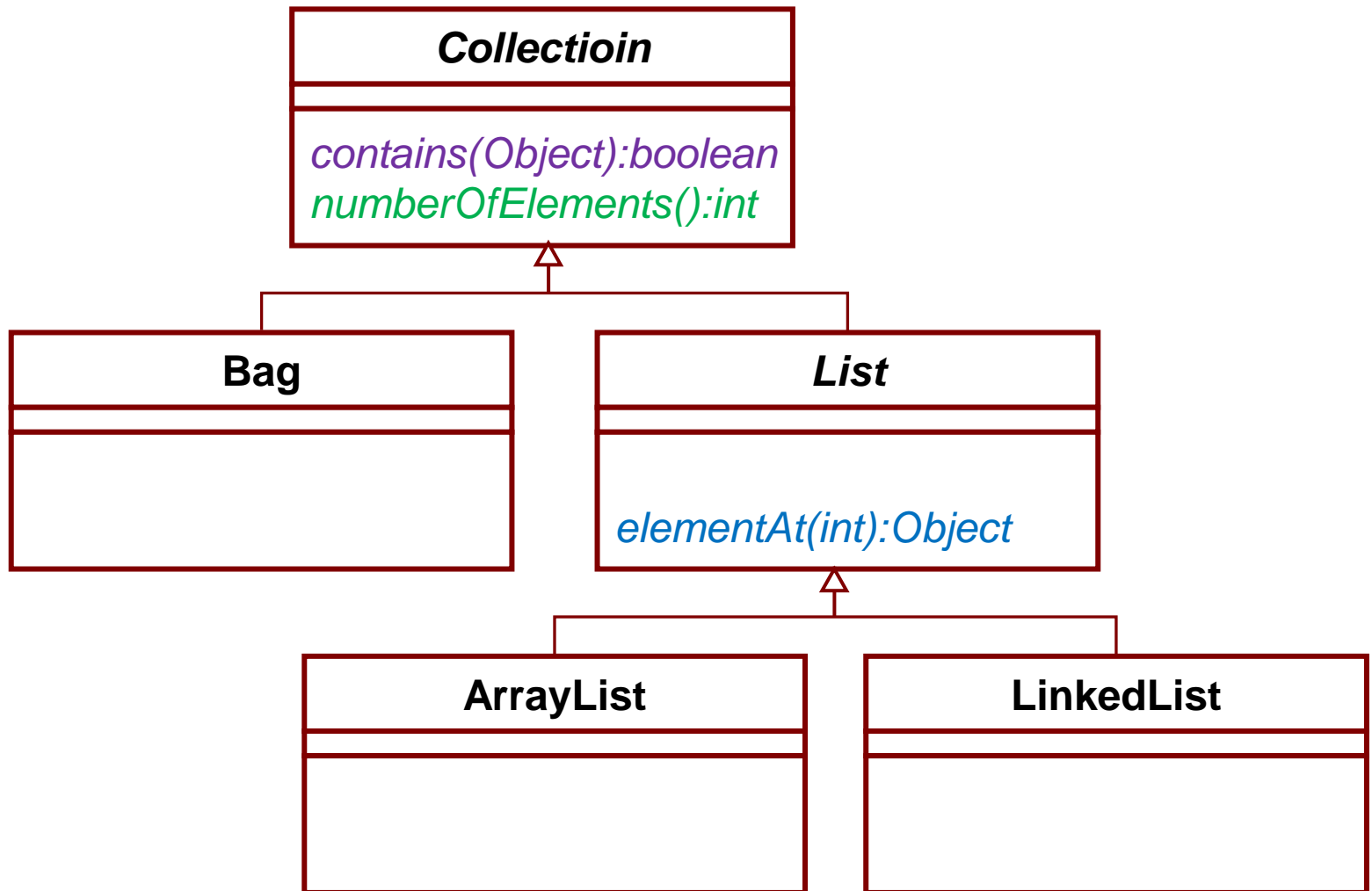
2.5 继承(10)



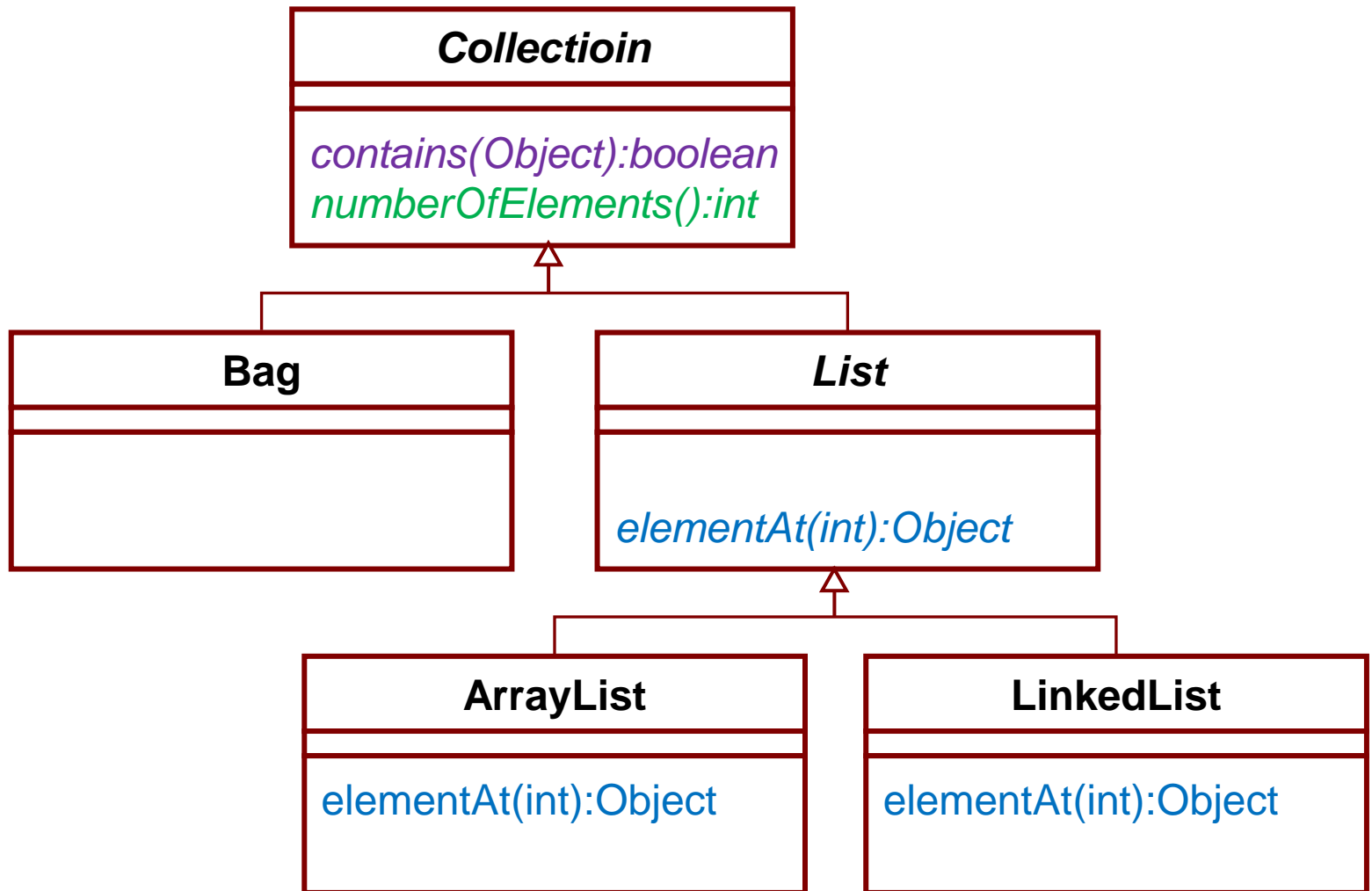
2.5 继承(11)

- 在开发层次结构时，可能涉及共享消息
 - ◆ 将共享消息的层次放置越高越好
 - ◆ contains(Object): 在集合中搜索对象
 - 位于Collection中
 - ◆ numberOfElements(): 返回集合中对象数
 - 位于Collection中
 - ◆ elementAt(int): 在参数指定的位置检索对象
 - 位于List中

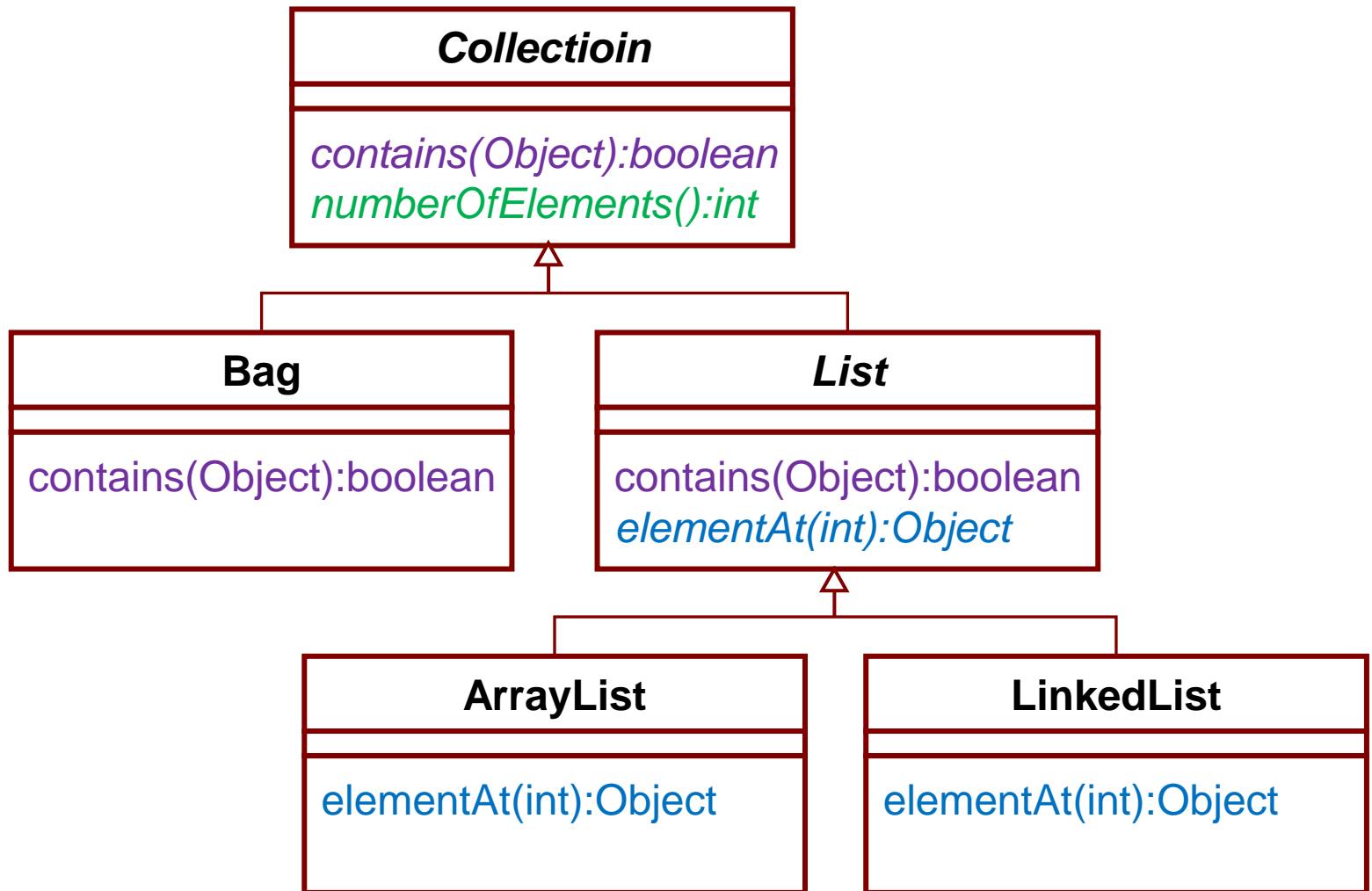
2.5 继承(12)



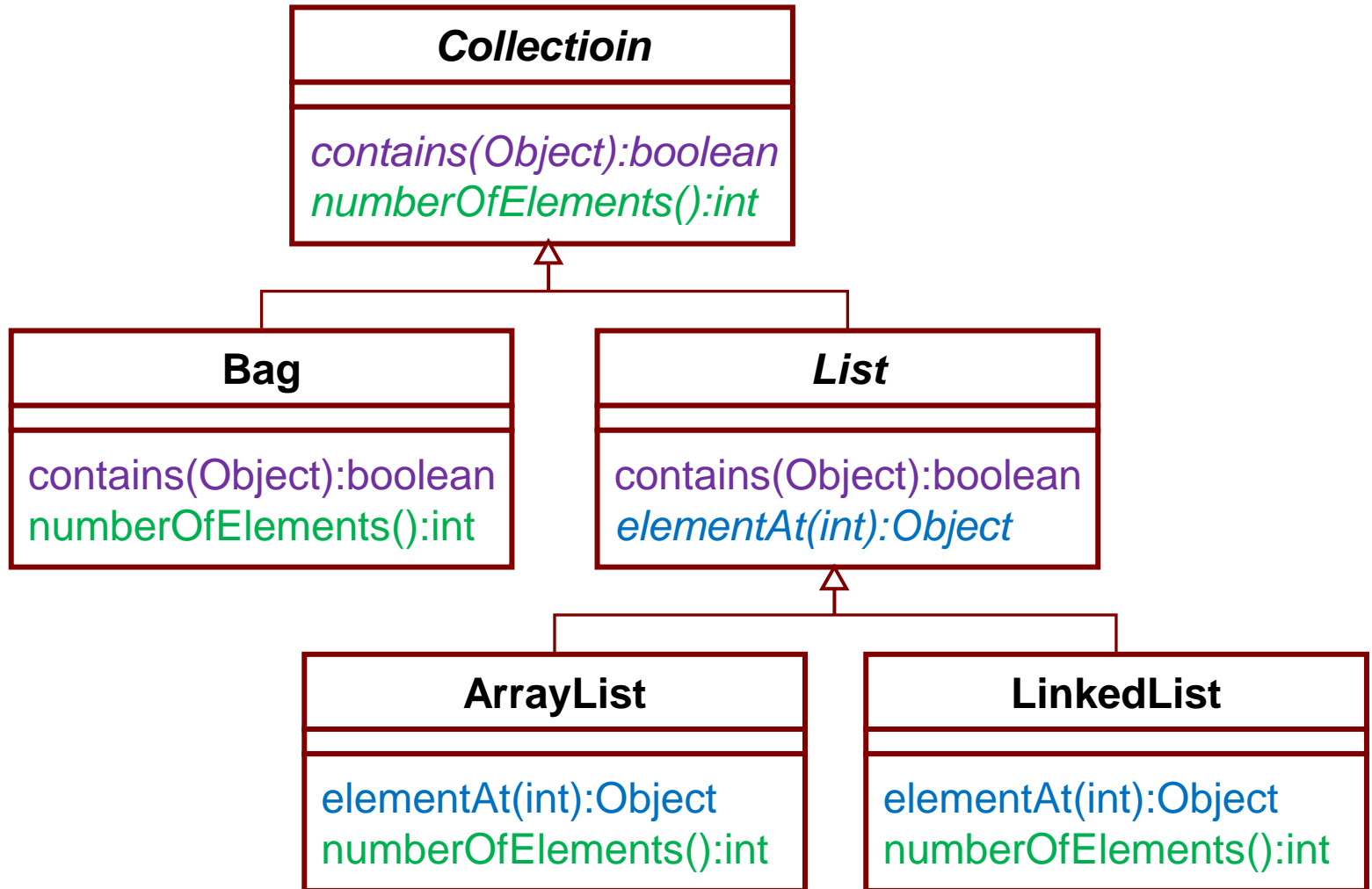
2.5 继承(13)



2.5 继承(14)



2.5 继承(15)



2.5 继承(16)

- 为类实现结构添加代码
 - ◆ 面向对象允许重新定义继承来的方法
 - 如果继承的方法是抽象的，通过重定义将其具体化
 - 子类中的方法需要完成一些额外的工作
 - 为子类提供更好的实现代码（更高效或更准确）

2.5 继承(17)

➤ 抽象方法

➤ 抽象类

- 至少有一个抽象方法的类
- 抽象方法可以是该类本身的方法，也可以是从超类继承来的
- 优点
 - ◆ 支持更丰富、更灵活地建模
 - ◆ 共享更多的代码，因为可以编写具体的方法来实现抽象的方法
 - ◆ 一般不能创建抽象类的实例

2.5 继承(18)

➤ 如何构建继承层次

- 在问题域中查找具体的概念，推导出它们的知识和行为
- 在具体的类中找出共同点，以便引入更一般的超类
- 把超类组合到更一般的超类中，直到找出最一般的根类为止（如Collection）

2.5 继承(19)

➤ 使用继承的规则

- 不用过度使用

- ◆ 在必须的时候使用继承

- ◆ 继承可以用聚合或使用属性替代

- 类应是其父类的一个类型

- ◆ 遵循继承的语义

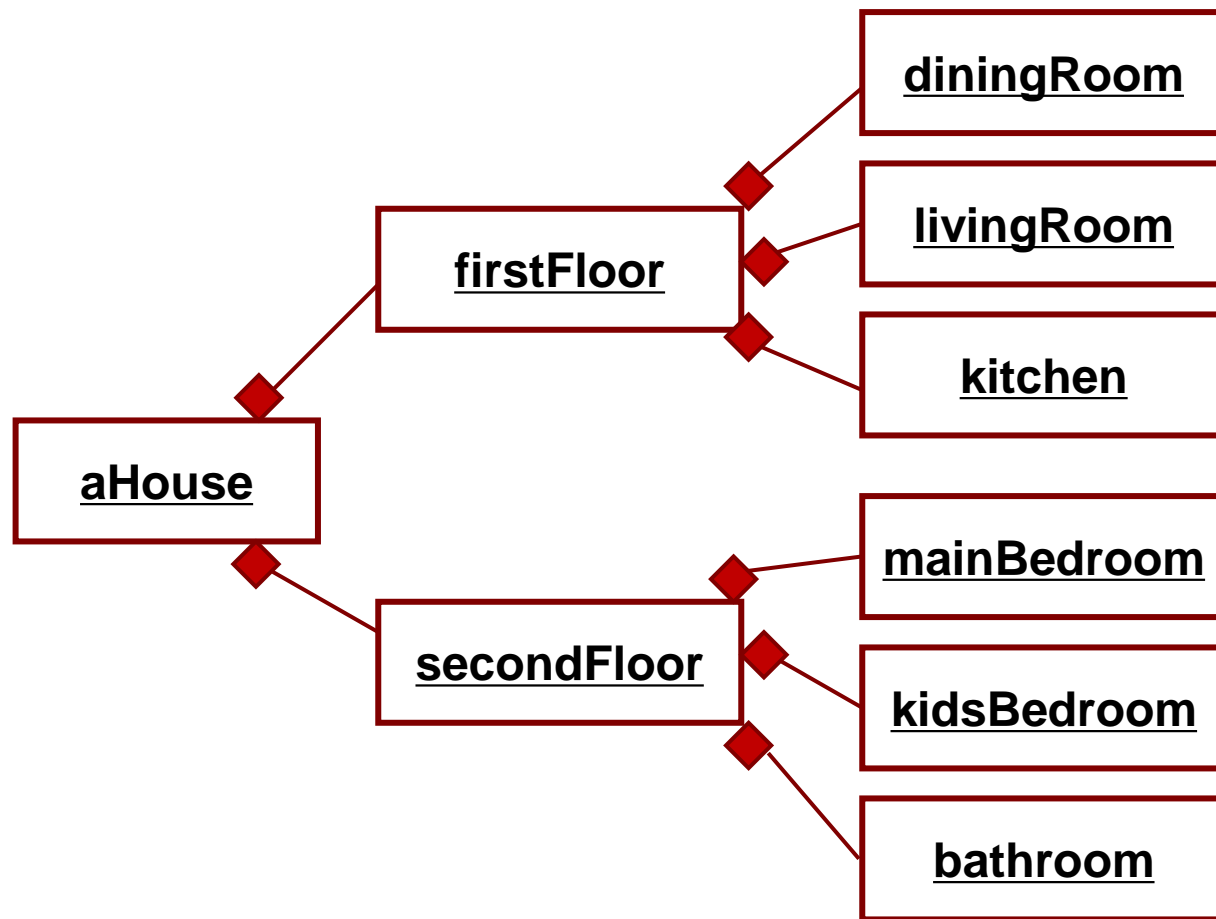
- 类应是其父类的扩展

- ◆ 在子类中只添加新特性，不要删除、禁用或重新解释特性

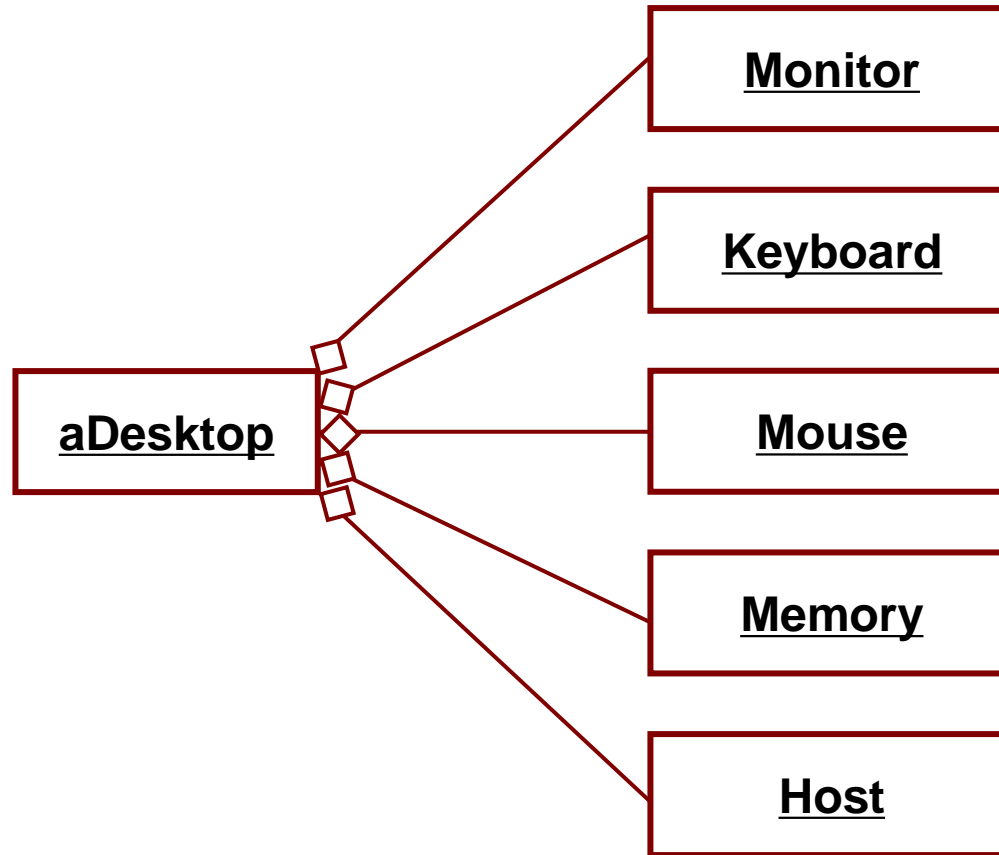
2.6 聚合(1)

- 复杂的对象可以用简单的对象作为其构成部分
- 一个（较复杂的）对象由其他若干（较简单的）对象作为其构成部分，称较复杂的对象为聚集，称较简单的对象为成分，称这种关系为聚合
- 聚合的语义
 - 整体—部分关系
 - “has a” 或 “is a part of”

2.6 聚合 (2)



2.6 聚合 (3)



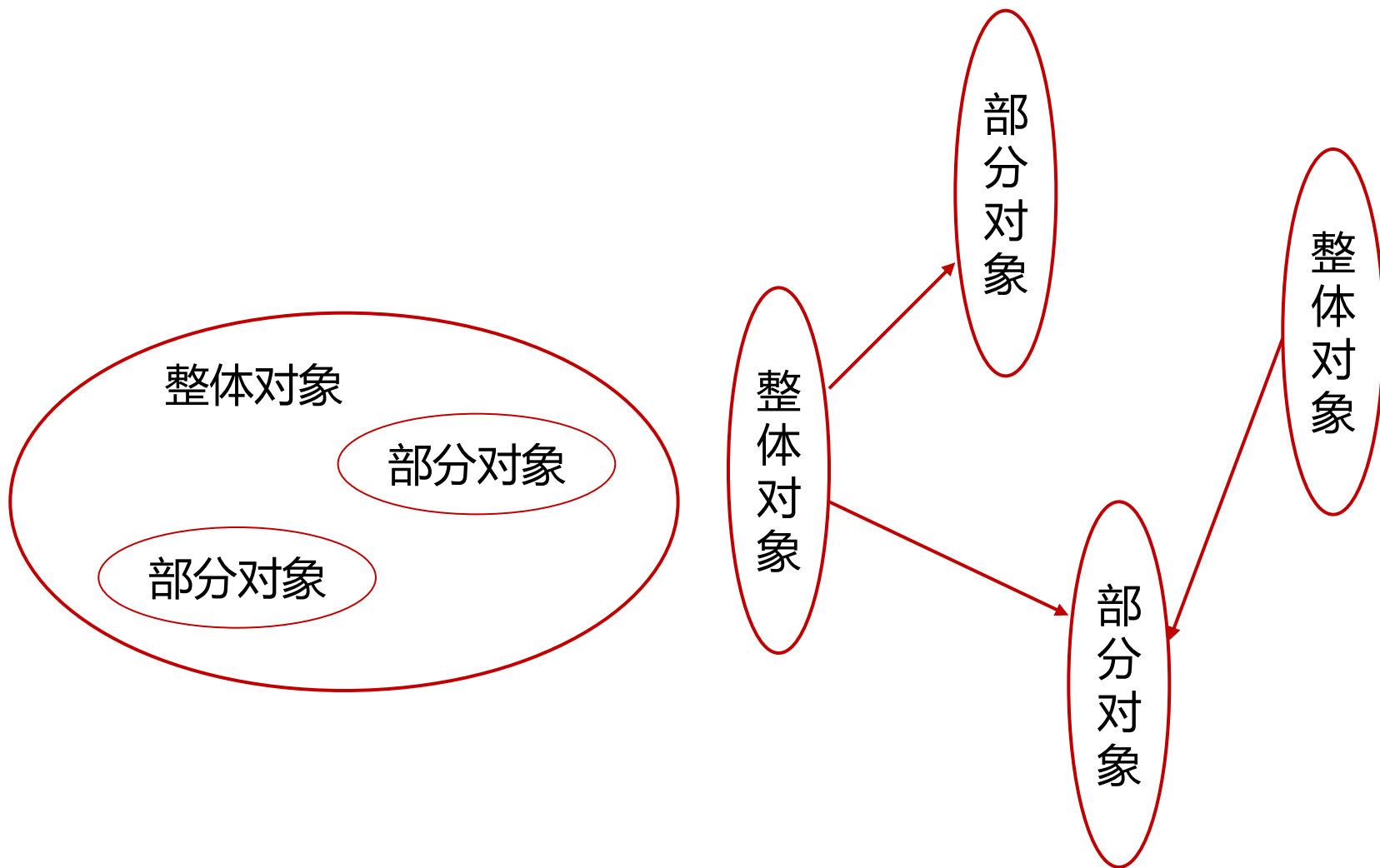
2.6 聚合(4)

➤ 组合(composition)

- 是聚合的一种形式，其部分和整体之间具有很强的“属于”关系
- 整体类的对象管理部分类的对象，决定部分类的对象何时属于它，何时不属于它
- 部分可以先于整体消亡
- 表示：实心菱形

➤ 整体对象与部分对象的关系

2.6 聚合 (5)



2.6 聚合(6)

➤ 示例：实现后进先出的栈(stack)类

- 相关的消息

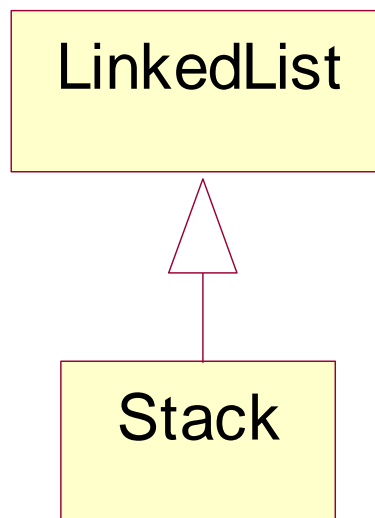
- ◆ `push(Object)`: 把对象添加到栈顶
- ◆ `peek():Object`: 返回栈顶对象
- ◆ `isEmpty():boolean`: 如果栈中没有对象，就返回true，否则返回false
- ◆ `pop():Object`: 从栈顶删除一个对象，并返回该对象

2.6 聚合(7)

- 分析已有的LinkedList类中的消息
 - `addElement()`: 在列表尾部添加一个对象
 - `lastElement()`: 返回列表尾部的对象
 - `numberOfElement():int`: 返回列表中的对象数
 - `removeLastElement()`: 删除列表尾部的对象
- 分析结论
 - 将已有的LinkedList的行为融合进Stack中

2.6 聚合(8)

➤ 使用继承实现栈



2.6 聚合(9)

- 代码实现示意(C++)

```
class Stack : public LinkedList {  
public:  
    void push(Object o) { addElement(o); }  
    Object peek() { return lastElement(); }  
    boolean isEmpty() {  
        return numberOfElement()==0; }  
    Object pop() {  
        Object o=lastElement();  
        removeLastElement();  
        return o;  
    }  
};
```

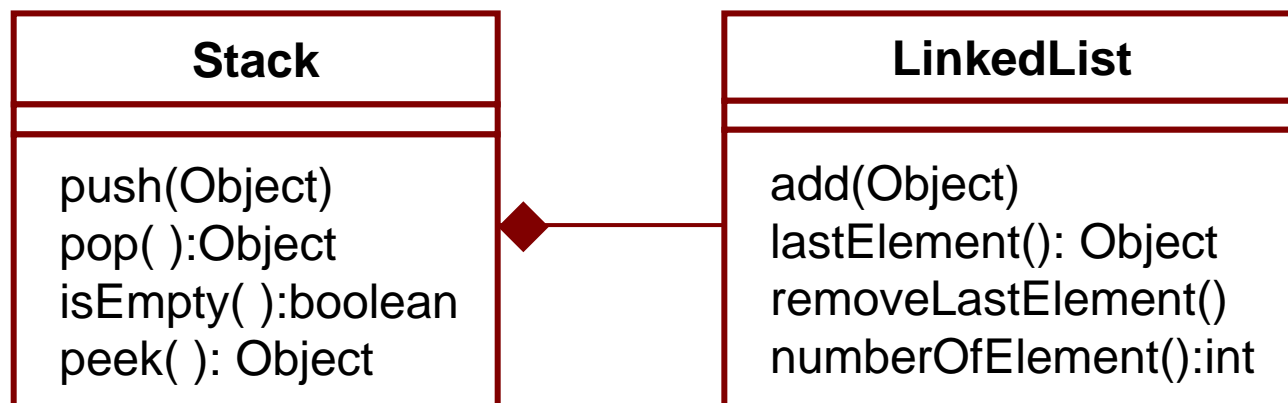
2.6 聚合(10)

- 代码实现示意(Java)

```
public class Stack extends LinkedList {  
    public void push(Object o) { addElement(o); }  
    public Object peek() { return lastElement(); }  
    public boolean isEmpty() {  
        return numberOfElements()==0; }  
    public Object pop() {  
        Object o=lastElement();  
        removeLastElement();  
        return o;  
    }  
};
```

2.6 聚合(11)

➤ 使用聚合实现栈



2.6 聚合(12)

- 代码实现示意(C++)

```
class Stack {  
public:  
    Stack() { list=new LinkedList(); }  
    void push(Object o) { list.addElement(o); }  
    Object peek() { return list.lastElement(); }  
    boolean isEmpty() {  
        return list.numberOfElement()==0; }  
    Object pop() {  
        Object o=listlastElement();  
        list.removeLastElement();    return o;  
    }  
private:  
    LinkedList list;  
};
```

2.6 聚合 (13)

- 代码实现示意(Java)

```
public class Stack {  
  
    public Stack() { list=new LinkedList(); }  
    public void push(Object o) { list.addElement(o); }  
    public Object peek() { return list.lastElement(); }  
    public boolean isEmpty() {  
        return list.numberOfElement()==0; }  
    public Object pop() {  
        Object o=listlastElement();  
        list.removeLastElement();    return o;  
    }  
  
    private LinkedList list;  
};
```

2.6 聚合(14)

➤ 继承与聚合的区别

● 继承的优点

- ◆ 自然
- ◆ 优雅
- ◆ 允许编写一般的代码

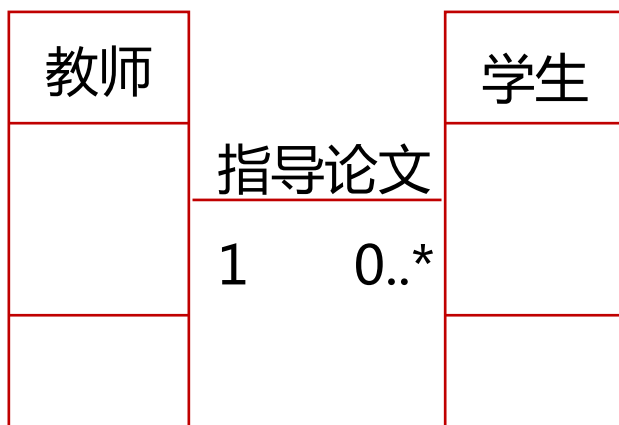
● 继承的缺点

- ◆ 很难做好
- ◆ 在发现设计中的不足时很难改变
- ◆ 客户程序员很难理解
- ◆ 层次结构会“泄露”给客户代码，也难以改变

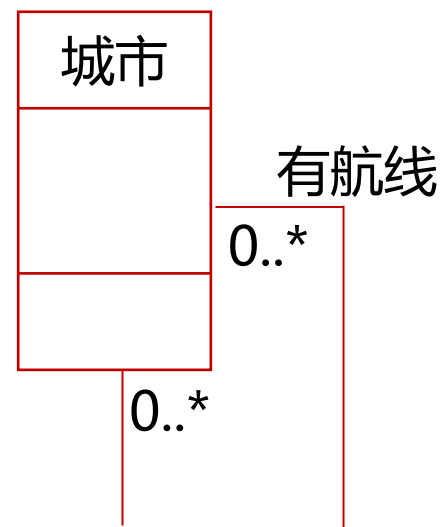
2.6 聚合(15)

- 聚合的优点
 - ◆ 较容易开发
 - ◆ 较容易改变
 - ◆ 客户容易理解
 - ◆ 不会泄露给客户代码
- 聚合的缺点
 - ◆ 代码冗余
 - ◆ 复用程度低

2.7 关联(1)



教师为学生指导论文

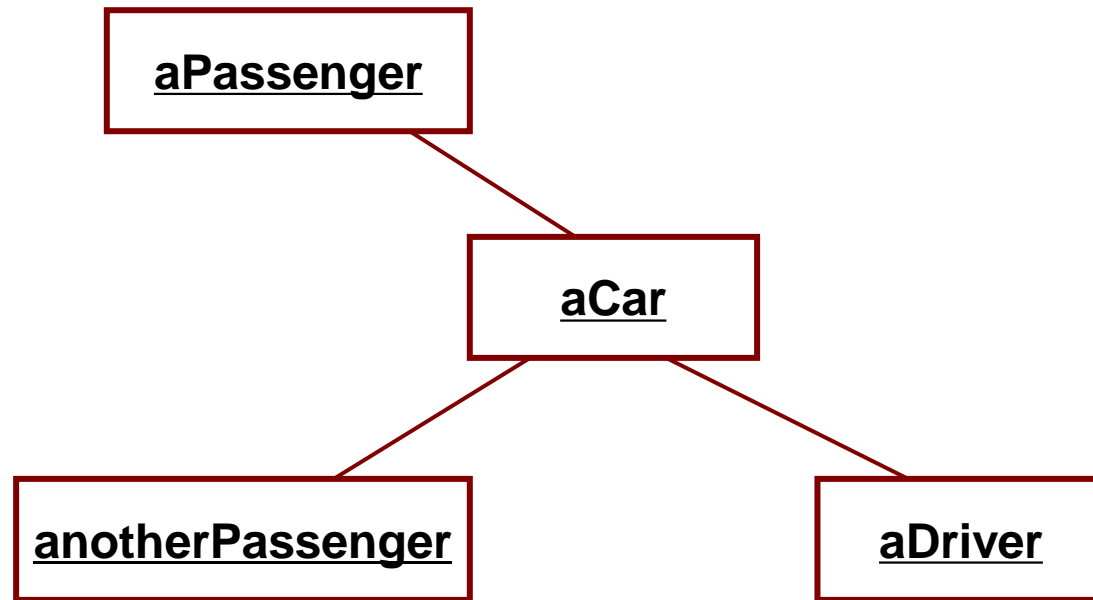


城市之间有航线

2.7 关联(2)

- 如果类的对象之间通过属性有连接关系，那么这些类之间的语义关系就是**关联**
 - 通过关联表达类(一组对象)之间的静态关系
 - 在实例化后，由类产生对象，由关联产生连接对象的链（链是关联的实例）
- 关联与聚合的区别
 - 关联是一种弱连接，不完全相互依赖
 - 聚合是一种强连接，成为一个更大的对象

2.7 关联 (3)



2.7 关联(4)

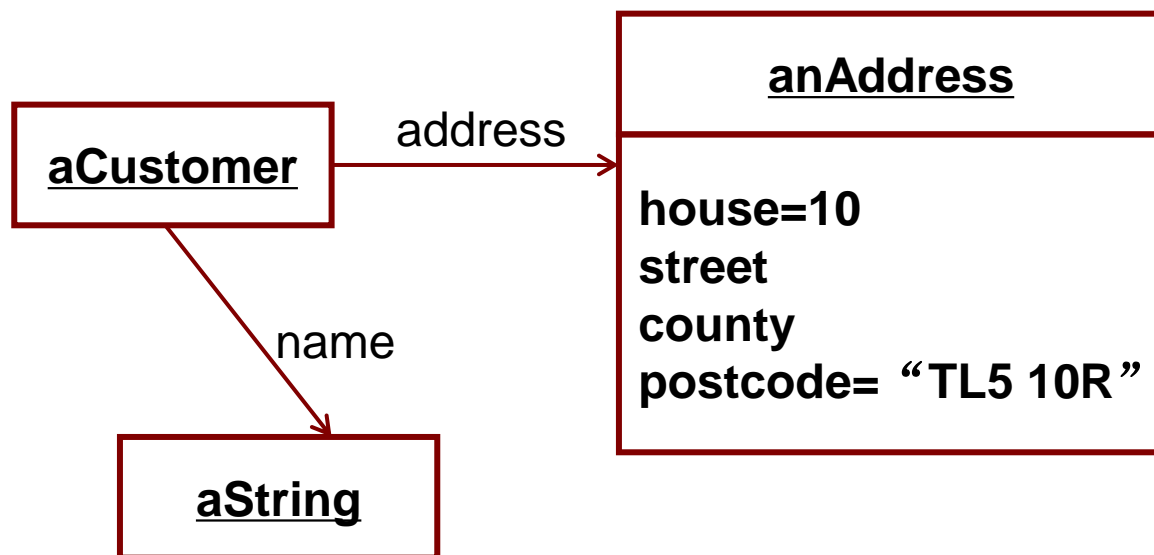
➤ 图与树

- 图是对象组之间连接的一个任意集合
- 树是图的一种特殊情况
 - ◆ 树是一种层次结构，每个节点都有一个父节点，可以有任意多个子节点
- 区别
 - ◆ 对象的任意组合会形成关联
 - ◆ 相互关联，有正确结构的连接形成树

2.7 关联 (5)

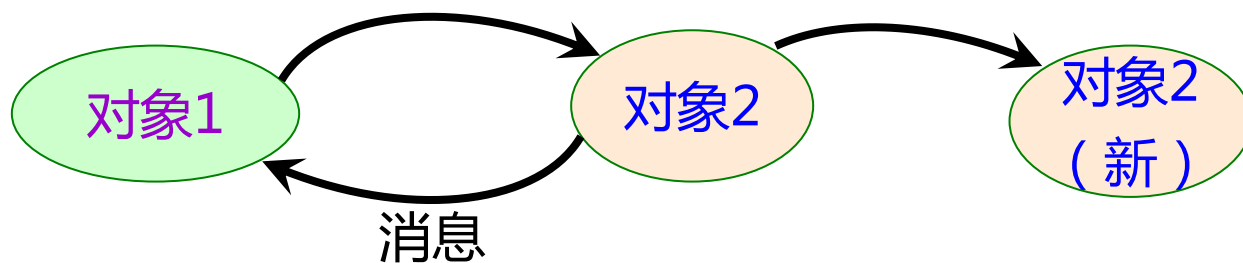
➤ 链接

- 对象图中的连接称为链接 (link)
- 每个链接可看做一个属性
- 可导航性：若要说明一个对象知道另一个对象的位置，则通过箭头说明(指针)



2.8 消息(1)

- 对象之间通过消息进行通信与协作，从而实现对象之间的动态联系
- 对象间通过消息相互作用，状态改变是其收到外部或其它实体消息后，根据自身行为规律处理(操作)的结果

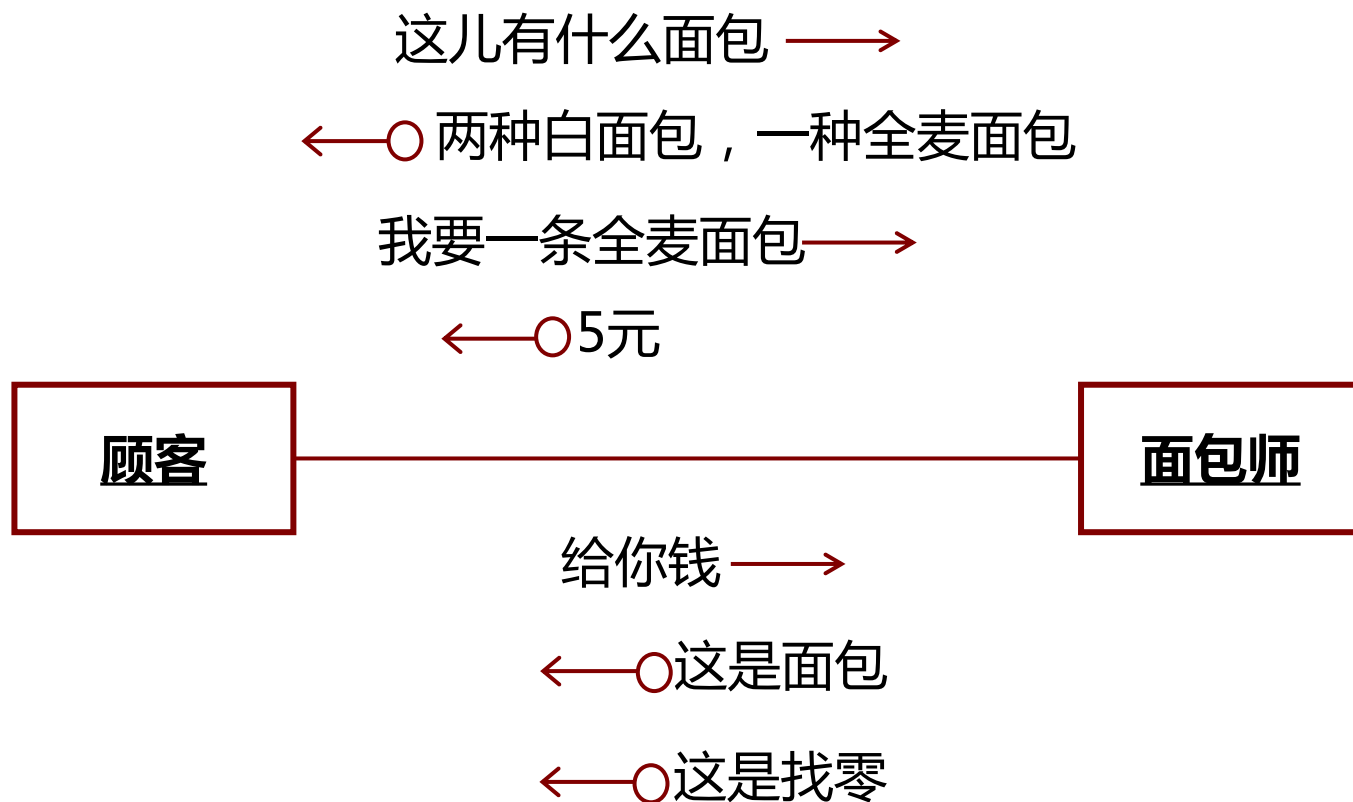


2.8 消息(2)

➤ 示例

顾客走进面包店，问面包师有什么种类的面包。面包师看看柜台，告诉顾客有两种白面包和一种全麦面包。顾客要买面包，现在开始交易：面包师包好面包，交给顾客，并要求顾客付款，顾客给面包师付钱，面包师给顾客找零钱，顾客满意地离开了。

2.8 消息(3)



2.8 消息(4)



2.8 消息(5)

- 软件对象收到消息后，会启动操作
- 消息可以带参数，接受者用其满足请求
 - `getPriceOf(wholemeal)`
- 消息需要指定接受者
- 在OO方法中，把向对象发出的操作请求称为消息
 - `aPerson.getHeight(aUnit);`
- 消息样式
 - 问题消息
 - 命令消息

2.8 消息(6)

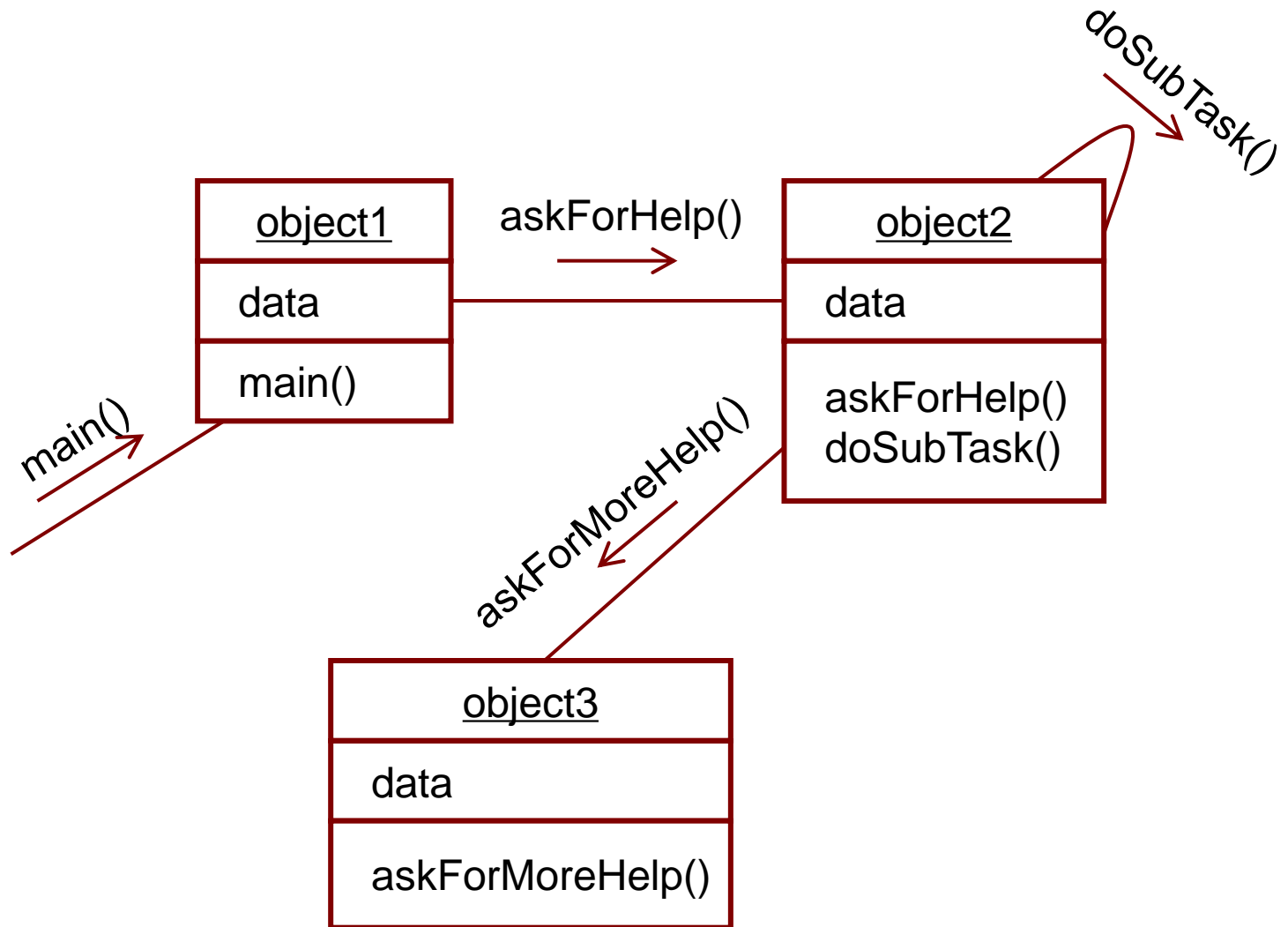
➤ 消息传递机制与函数调用机制的区别

- 在消息传递机制中，每一个消息被发送给指定的接收者（对象）。在命令式编程范型中，函数调用机制没有指定的接收者
- 消息的解释（用来完成操作请求的方法代码集）依赖接收者，并且因接收者的不同而异
- 在面向对象的范型中，通常在运行时才能知道给定消息的特定的接收者

2.8 消息(7)

- 面向对象的程序在工作时，首先要创建对象，把它们连接在一起，让它们彼此发送消息，相互协作
- 问题
 - 谁启动这个过程？
 - 谁创建第一个对象
 - 谁发送第一个消息
- 解决
 - 程序必须有一个入口点，例如main函数

2.8 消息(8)

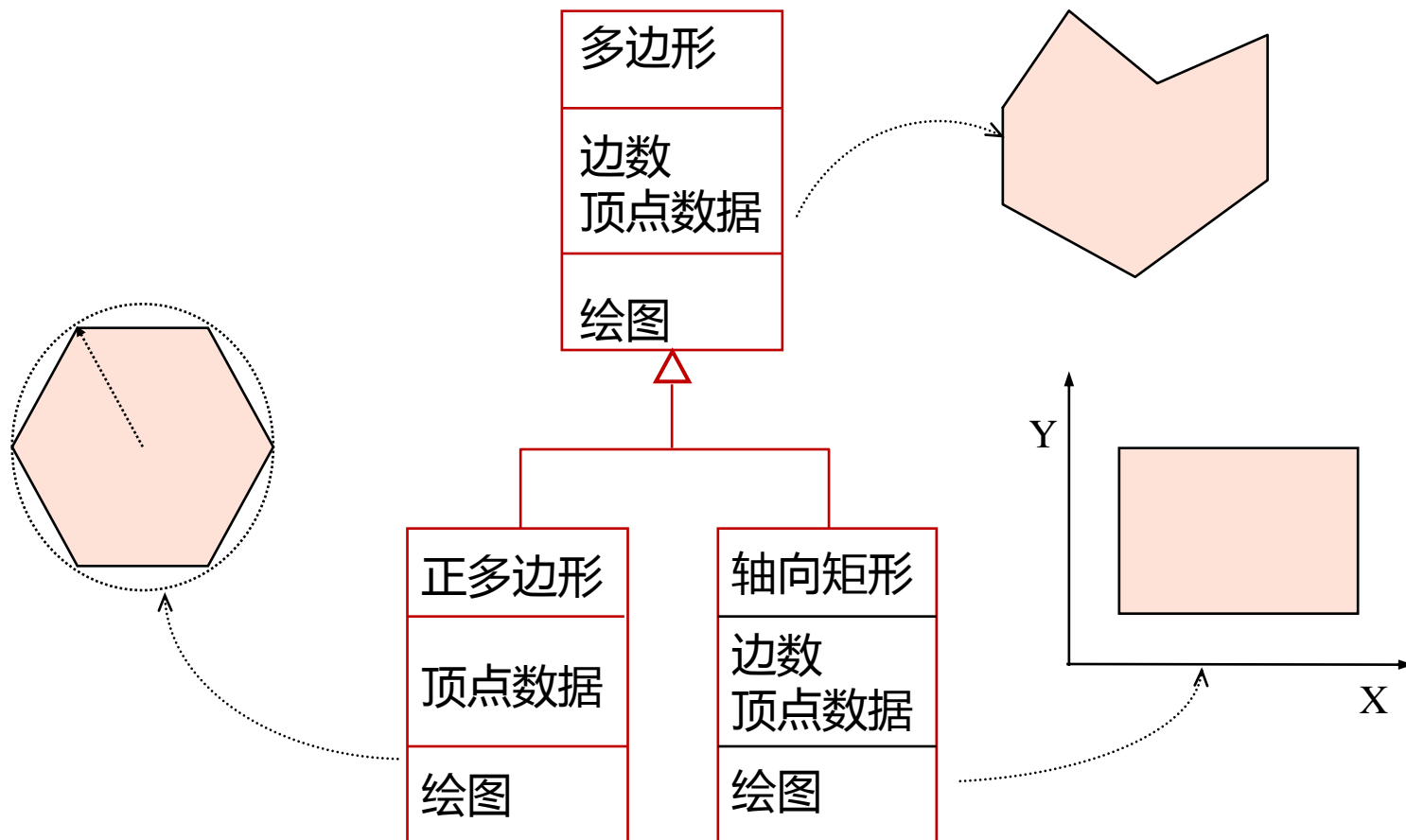


2.9 多态(1)

➤ 多态

- 是指同一个命名可具有不同的语义
- OO方法中，常指在一般类中定义的属性或操作被特殊类继承之后，可以具有不同的数据类型或表现出不同的行为
 - ◆ 多态变量表示值在不同时刻表示不同的类型
 - ◆ 多态消息表示有多个方法与对象相关
- 用途：把具有共同基类的对象组成一组，并对它们进行一致的处理

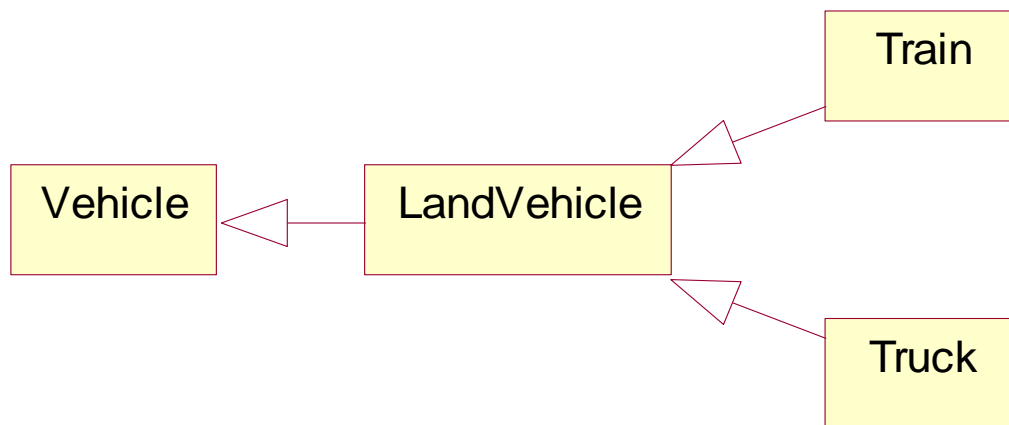
2.9 多态(2)



2.9 多态 (3)

➤ 多态变量

```
LandVehicle* lv=new LandVehicle;
```

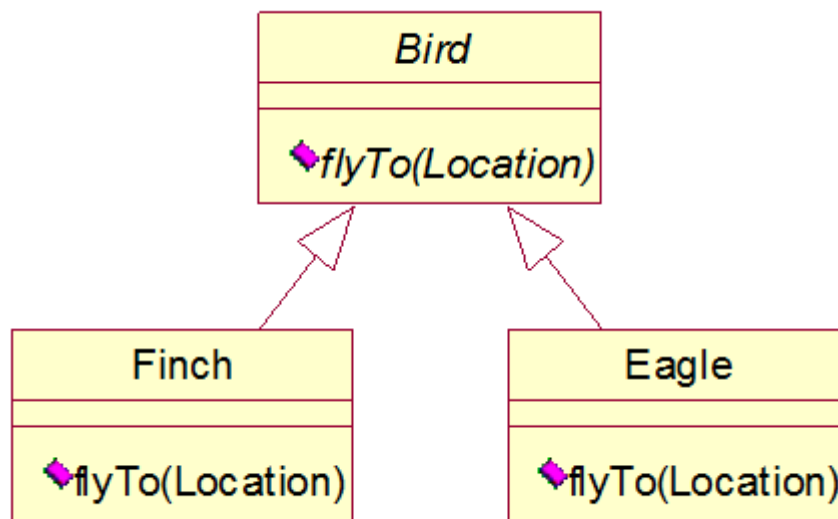


```
LandVehicle* lv=new Train;
LandVehicle* lv=new Truck;
```

lv为
多态变量

2.9 多态(4)

➤ 多态消息



```
Bird* b=new Finch();  
b->flyTo(someLocation);
```

```
Bird* b=new Eagle();  
b->flyTo(someLocation);
```

*flyTo*为
多态消息

2.9 多态 (5)

➤ 动态绑定

- 任何消息都可以关联多个方法
- 方法在多个类中是独立的，或者方法由子类重新定义
- 重定义的方法一般有类似的算法，独立的方法通常有完全不同的算法
- 动态绑定在运行期间将消息关联到方法上

2.9 多态(6)

➤ 动态和静态类型系统

- 静态类型系统 (static)

- ◆ 由编译器完成

- ◆ 静态类型系统禁止编译器间的误用

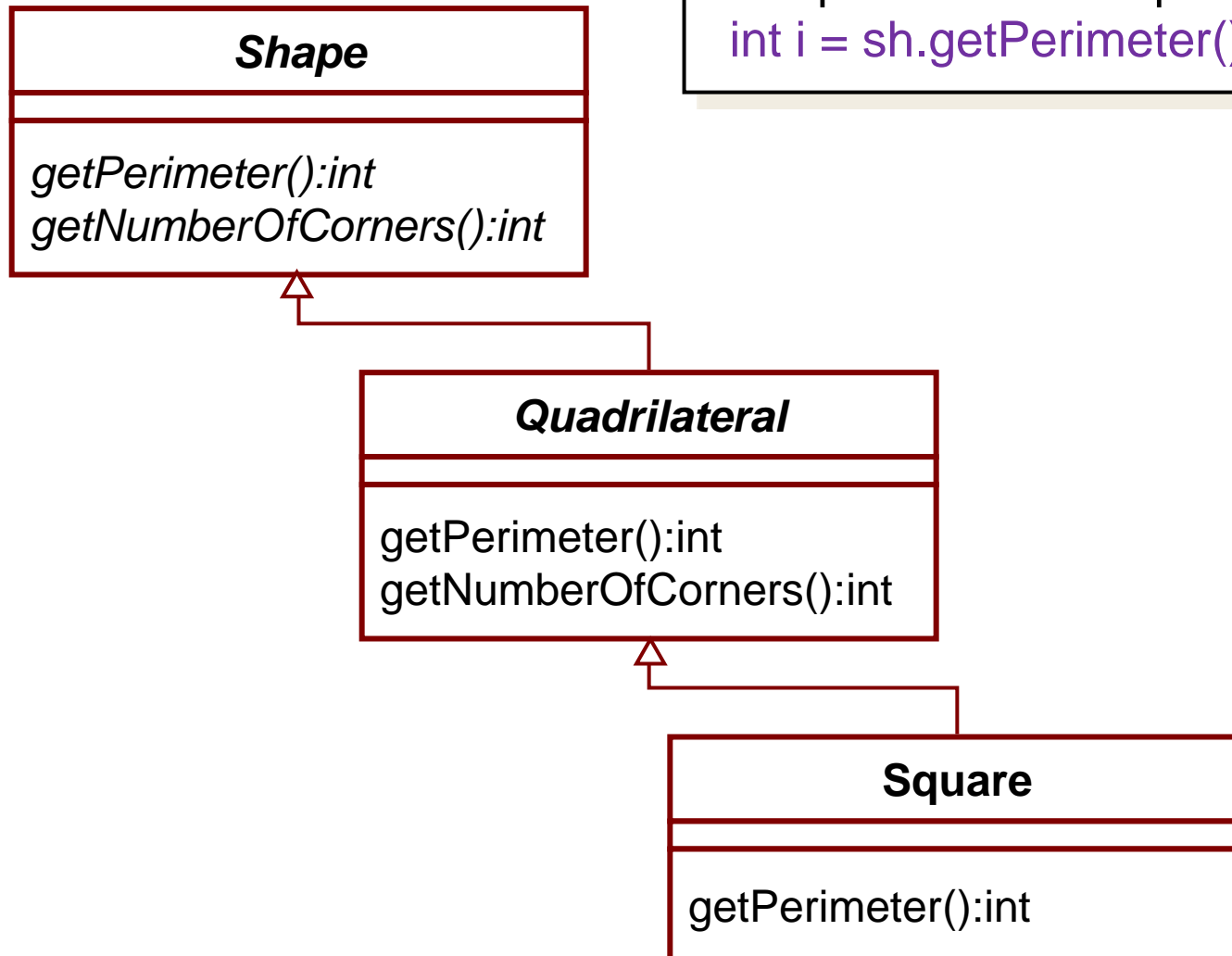
- 动态类型系统 (dynamic)

- ◆ 由运行时系统完成

- ◆ 动态类型系统在程序运行时检查出现误用

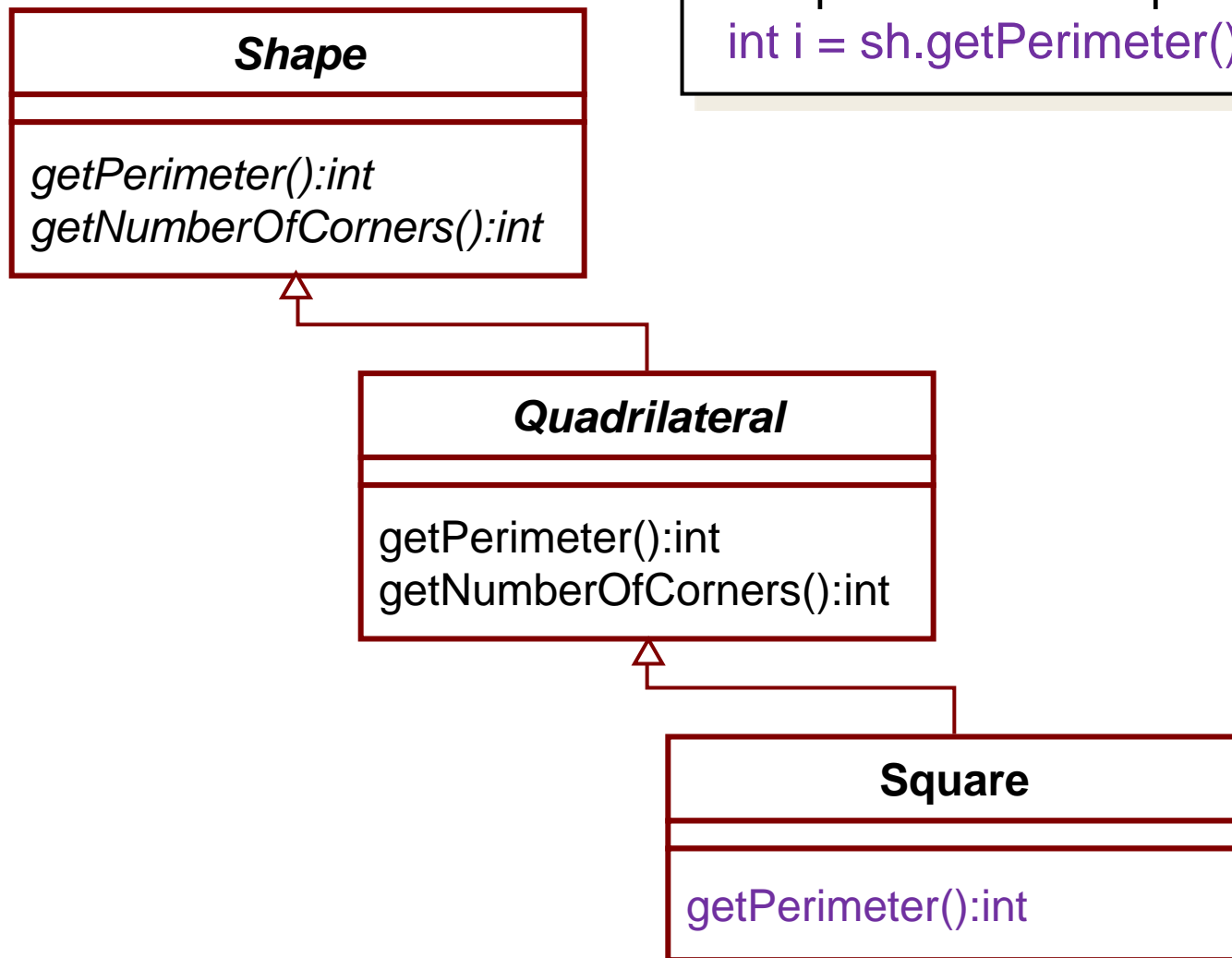
2.9 多态 (7)

```
Shape* sh= new Square;  
int i = sh.getPerimeter();
```

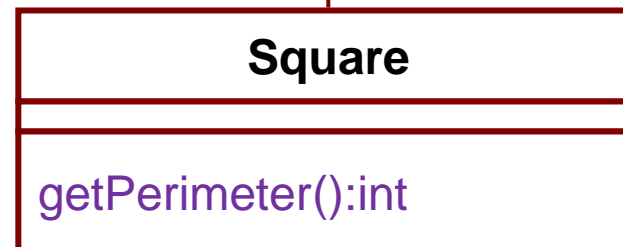
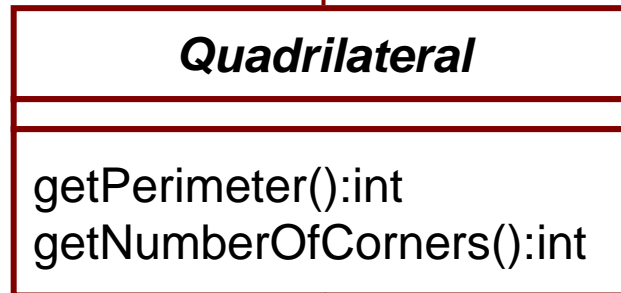
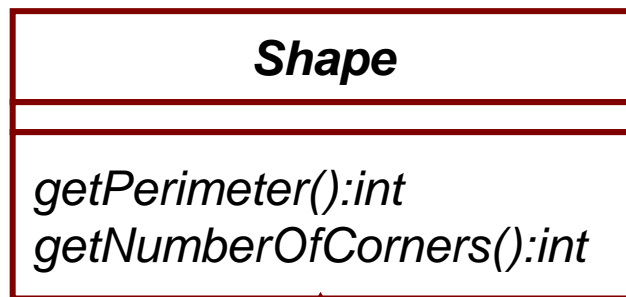


2.9 多态 (7)

```
Shape* sh= new Square;  
int i = sh.getPerimeter();
```



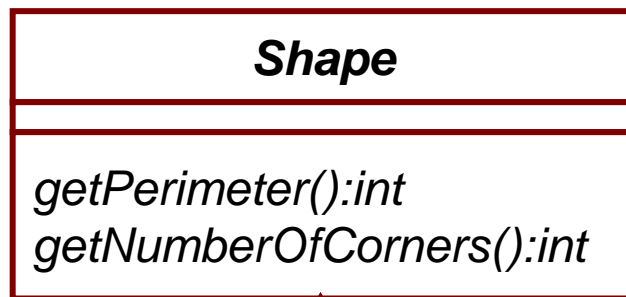
2.9 多态 (7)



```
Shape* sh= new Square;  
int i = sh.getPerimeter();
```

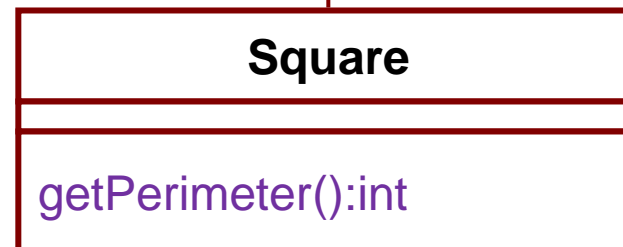
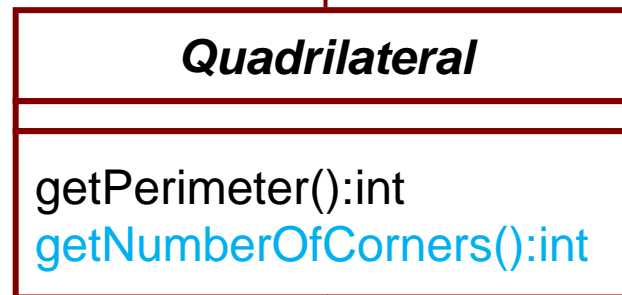
```
int j = sh.getNumberOfCorners();
```

2.9 多态 (7)

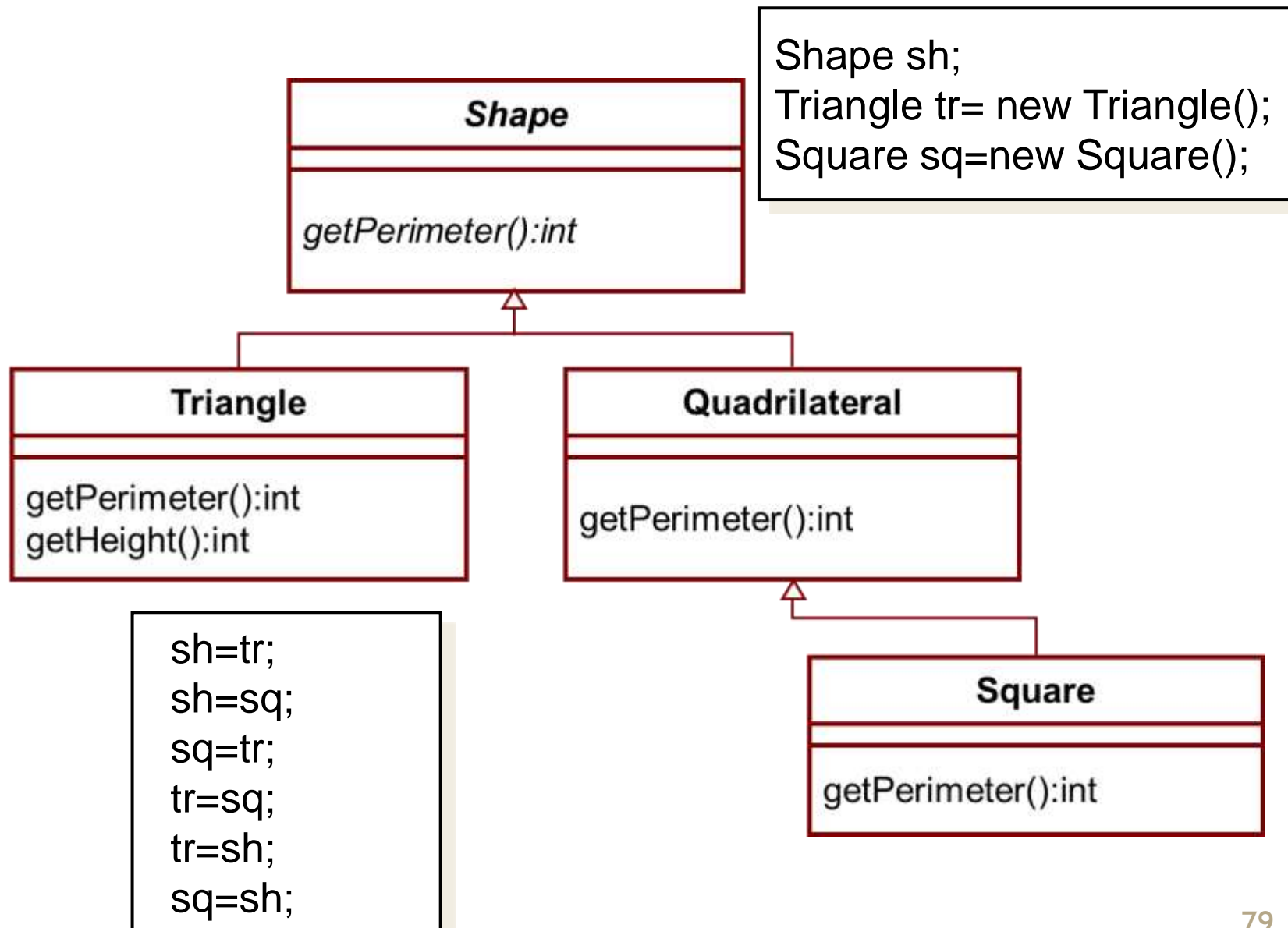


```
Shape* sh= new Square;  
int i = sh.getPerimeter();
```

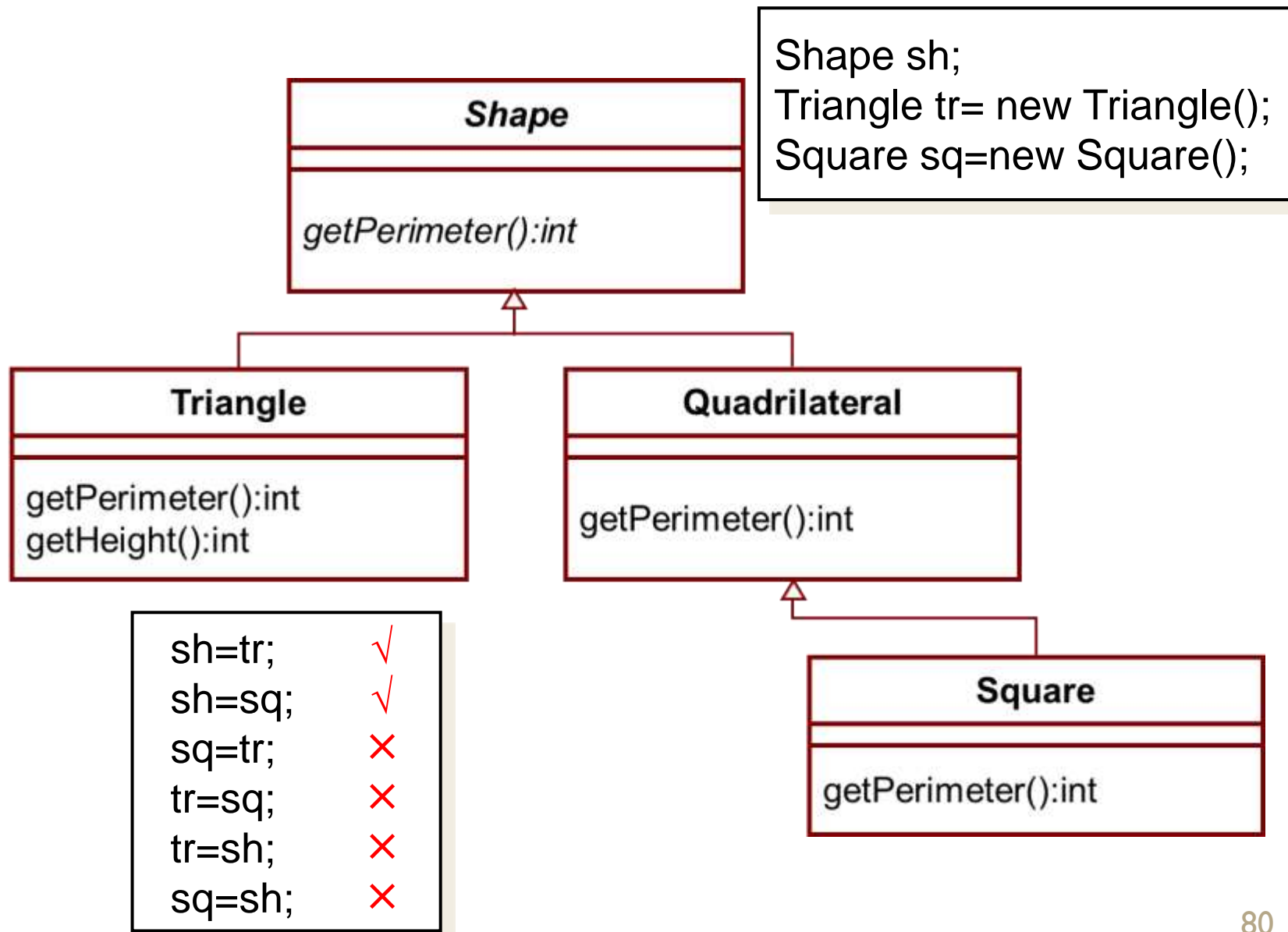
```
int j = sh.getNumberOfCorners();
```



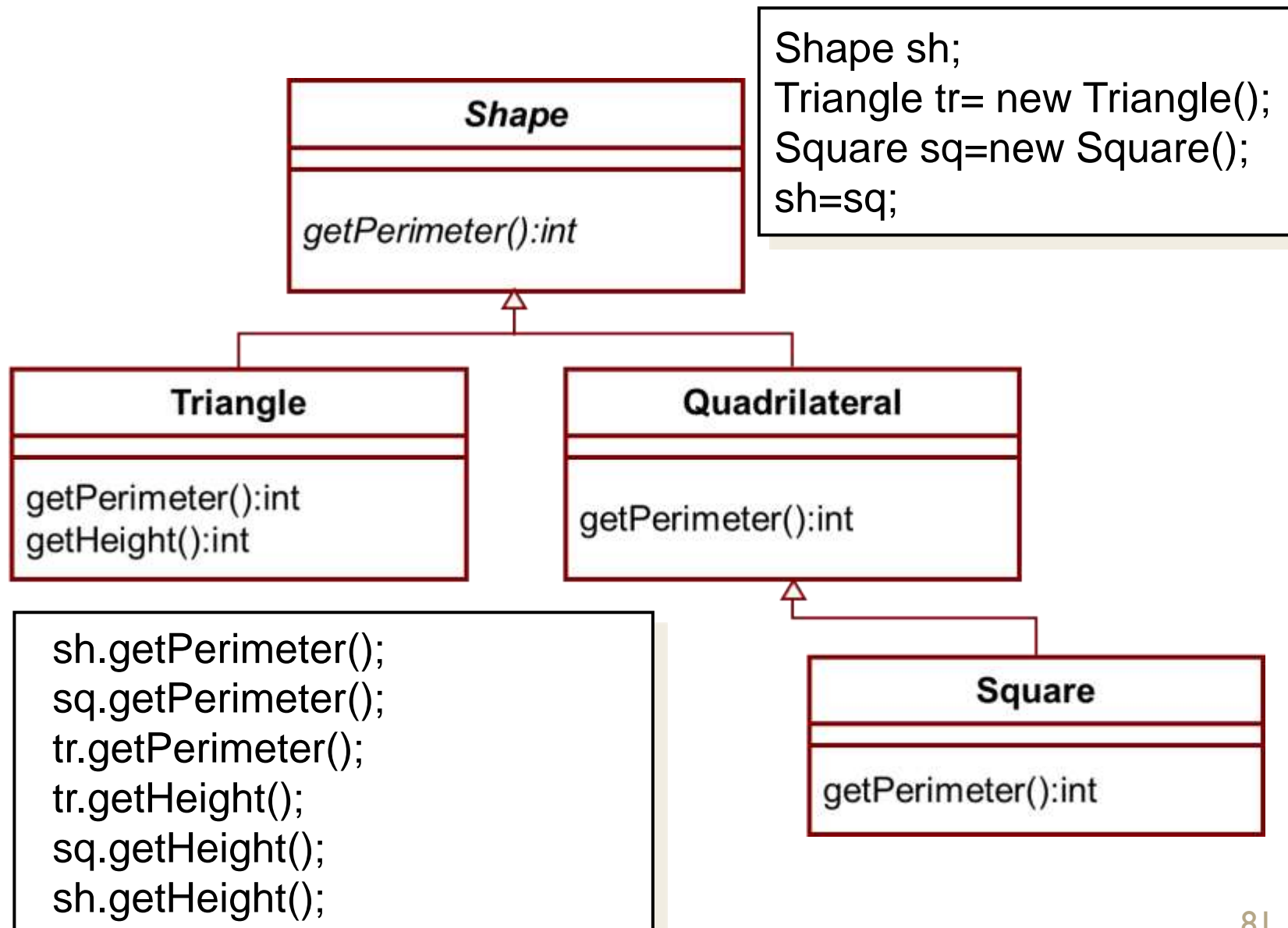
2.9 多态 (8)



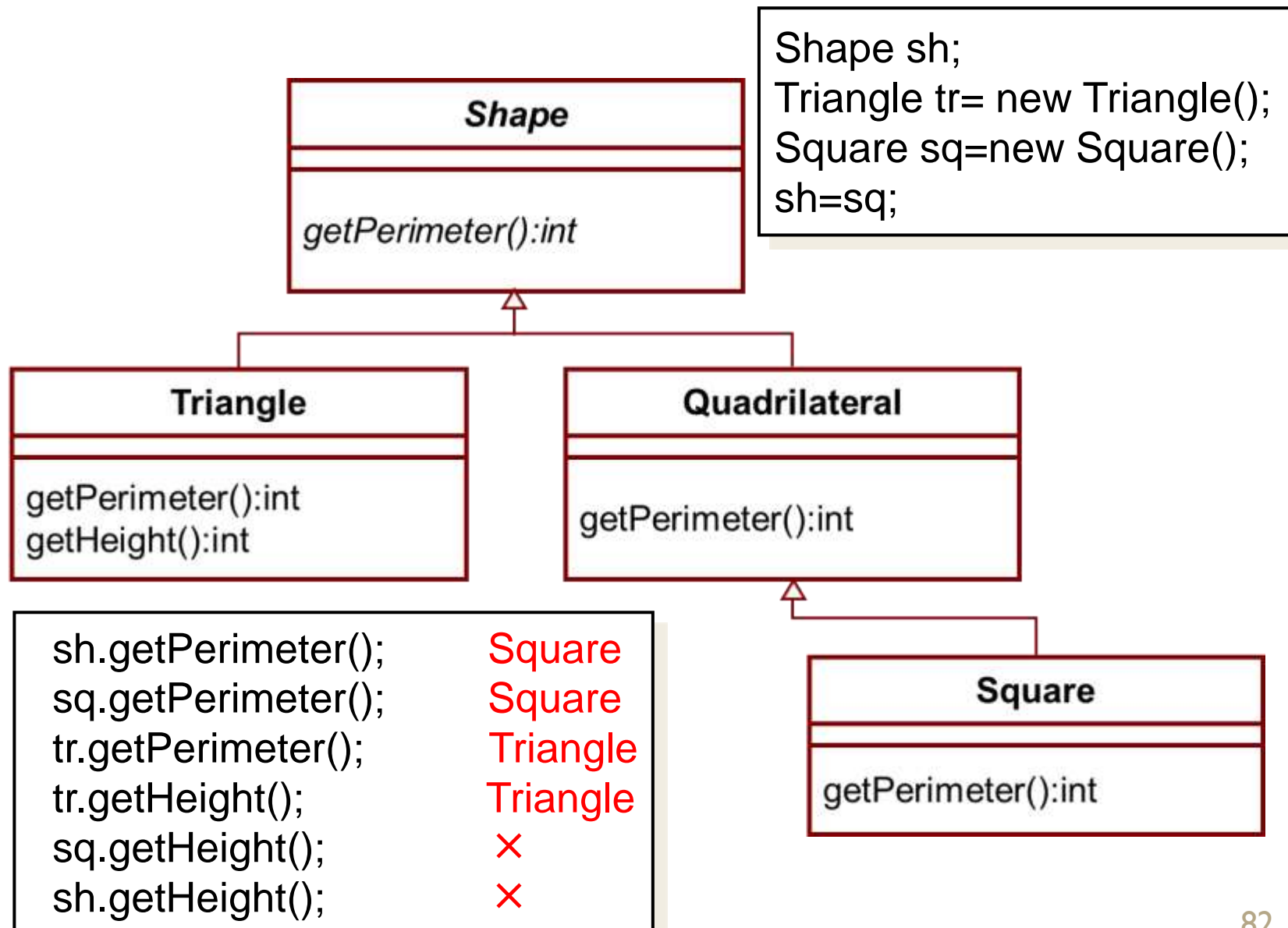
2.9 多态 (8)



2.9 多态 (9)



2.9 多态 (9)



2.9 多态(10)

➤ 多态性规则

● 样式规则

- ◆ 总是使用尽可能高的抽象级别来编程，即总是把字段、本地变量和方法参数的类型声明为继承层次结构中最高的类，再让多态性完成其他工作

● 优点

- ◆ 所使用的抽象级别越高，代码的可重用性就越大

2.10 复用(1)

➤ 重用目的

- 开发更快速、简单
- 维护更容易（代码较少，人为错误就较少）
- 更健壮的代码（每次重用代码时，都会重复测试它，错误就会越来越少）

➤ 重用形式

- 重用系统中的函数
- 重用对象中的方法

2.10 复用(2)

- 重用系统中的类
- 在系统之间重用函数
- 在系统之间重用方法

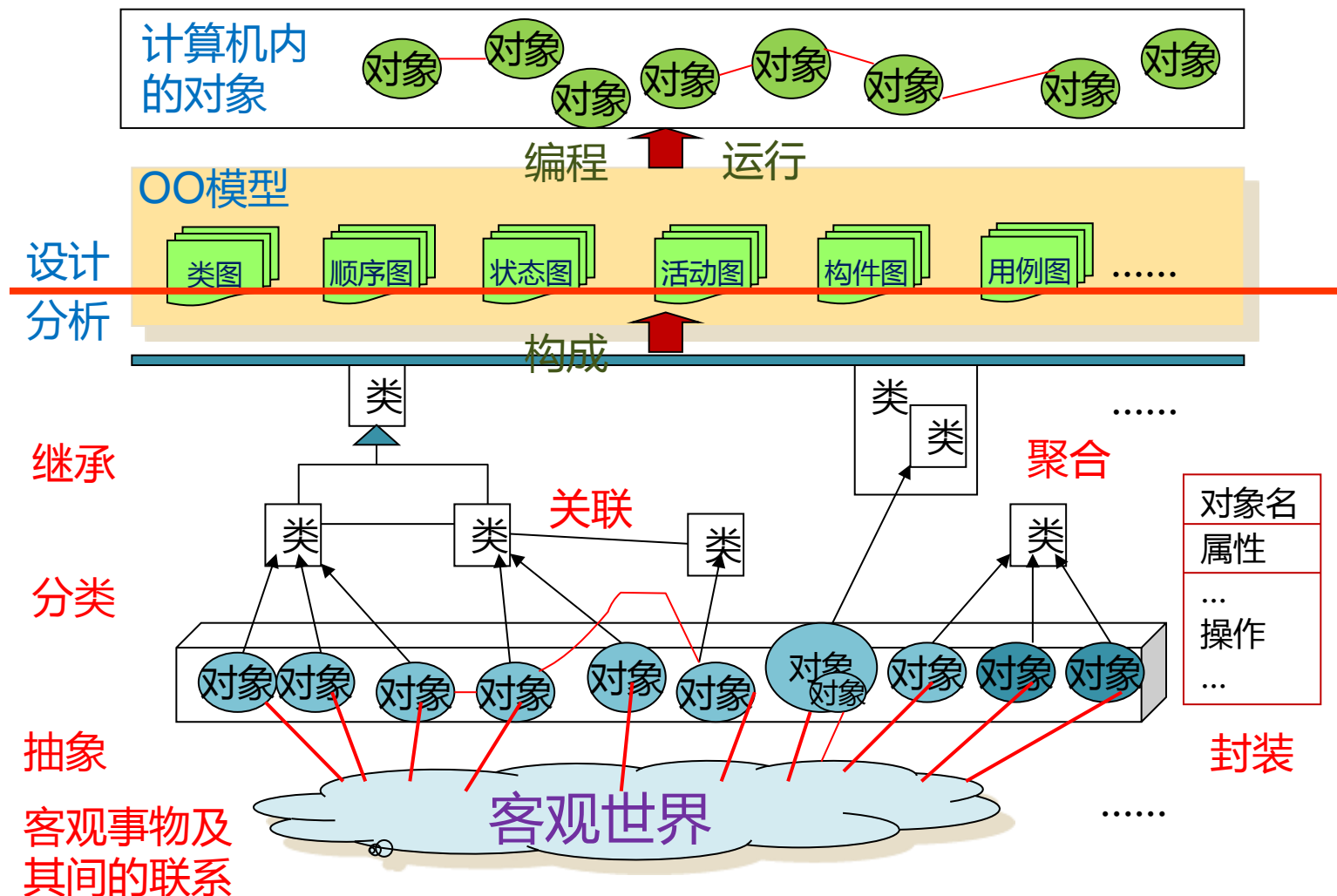
➤ 遵循的规则

- 样式规则
- 完整的文档说明
- 准备编写比需要更多的代码
- 使用模式和框架
- 设计客户-提供者对象

2.10 复用(3)

- 使每个对象只有一个目标：高聚合
- 把接口与业务行为分开
- 为问题和命令设计对象

面向对象基本思想示意图



Example: Rick's Guitars

- Maintain a guitar inventory
- Locate guitars for customers



What your predecessor built

Guitar
serialNumber : String
price : double
builder : String
model : String
type : String
backWood : String
topWood : String
+getSerialNumber() : String
+getPrice() : double
+setPrice(newPrice : double)
+getBuilder() : String
+getModel() : String
+getType() : String
+getBackWood() : String
+getTopWood() : String

Inventory
guitars : List
+addGuitar(serialNumber : String, price : double, builder : String, model : String, type : String, backWood : String, topWood : String) : void
+getGuitar(serialNumber : String) : Guitar
+searchGuitar(searchedFor : Guitar) : Guitar

The Guitar class

```
public class Guitar {

    private String serialNumber, builder, model, type, backWood, topWood;
    private double price;

    public Guitar(String serialNumber, double price,
                  String builder, String model, String type,
                  String backWood, String topWood) {
        this.serialNumber = serialNumber;
        this.price = price;
        this.builder = builder;
        this.model = model;
        this.type = type;
        this.backWood = backWood;
        this.topWood = topWood;
    }
    public String getSerialNumber() {return serialNumber;}
    public double getPrice() {return price;}
    public void setPrice(float newPrice) {
        this.price = newPrice;
    }
    public String getBuilder() {return builder;}
    public String getModel() {return model;}
    public String getType() {return type;}
    public String getBackWood() {return backWood;}
    public String getTopWood() {return topWood;}
}
```

The Inventory class-1

```
public class Inventory {  
    private List guitars;  
    public Inventory() { guitars = new LinkedList(); }  
    public void addGuitar(String serialNumber, double price,  
                           String builder, String model,  
                           String type, String backWood, String topWood) {  
        Guitar guitar = new Guitar(serialNumber, price, builder,  
                                     model, type, backWood, topWood);  
        guitars.add(guitar);  
    }  
    public Guitar getGuitar(String serialNumber) {  
        for (Iterator i = guitars.iterator(); i.hasNext(); ) {  
            Guitar guitar = (Guitar)i.next();  
            if (guitar.getSerialNumber().equals(serialNumber)) {  
                return guitar;  
            }  
        }  
        return null;  
    }  
}
```

The Inventory class-2

```
public Guitar search(Guitar searchGuitar) {
    for (Iterator i = guitars.iterator(); i.hasNext(); ) {
        Guitar guitar = (Guitar)i.next();
        String builder = searchGuitar.getBuilder().toLowerCase();
        if ((builder != null) && (!builder.equals("")) &&
            (!builder.equals(guitar.getBuilder().toLowerCase())))
            continue;
        String model = searchGuitar.getModel().toLowerCase();
        if ((model != null) && (!model.equals("")) &&
            (!model.equals(guitar.getModel().toLowerCase())))
            continue;
        String type = searchGuitar.getType().toLowerCase();
        if ((type != null) && (!searchGuitar.equals("")) &&
            (!type.equals(guitar.getType().toLowerCase())))
            continue;
        String backWood = searchGuitar.getBackWood().toLowerCase();
        if ((backWood != null) && (!backWood.equals("")) &&
            (!backWood.equals(guitar.getBackWood().toLowerCase())))
            continue;
        String topWood = searchGuitar.getTopWood().toLowerCase();
        if ((topWood != null) && (!topWood.equals("")) &&
            (!topWood.equals(guitar.getTopWood().toLowerCase())))
            continue;
        return guitar;
    }
    return null;
}
```

But there's a problem

I'm looking for a
fender Strat.

Not in stock.



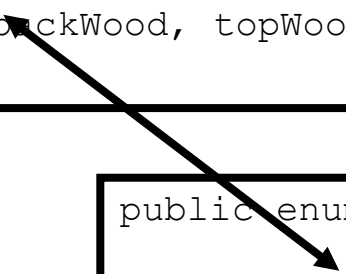
Rick's Guitars
Back Room



fender ≠ Fender

Solution: Remove strings

```
public class Guitar {  
  
    private String serialNumber,  
        model;  
    private double price;  
    private Builder builder;  
    private Type type;  
    private Wood backWood, topWood;  
    ...  
}
```



```
public enum Type {  
  
    ACOUSTIC, ELECTRIC;  
  
    public String toString() {  
        switch(this) {  
            case ACOUSTIC: return "acoustic";  
            case ELECTRIC: return "electric";  
            default:      return "unspecified";  
        }  
    }  
}
```

A better enum implementation

```
public enum Type {  
    ACOUSTIC("acoustic"),  
    ELECTRIC("electric"),  
    UNSPECIFIED("unspecified");  
    String value;  
    private Type(String value)  
    {  
        this.value = value;  
    }  
    public String toString() {  
        return value;  
    }  
}
```

The big picture

Guitar
serialNumber: String
price: double
builder: Builder
model: String
type: Type
backWood: Wood
topWood: Wood
getSerialNumber(): String
getPrice(): double
setPrice(float)
getBuilder(): Builder
getModel(): String
getType(): Type
getBackWood(): Wood
getTopWood(): Wood

Inventory
guitars: List
addGuitar(String, double, Builder , String, Type , Wood , Wood)
getGuitar(String): Guitar
search(Guitar): Guitar

Builder
toString(): S

Type
toString(): Str


Wood
toString(): String

Problem2

- There's often more than one guitar that matches the customer's needs.

Inventory
guitars: List
addGuitar(String, double, Builder, String, Type, Wood, Wood)
getGuitar(String): Guitar
search(Guitar): List

We want search() to be able to return multiple Guitar objects if Rick has more than one guitar that matches his client's specs.



Problem3

You know, when a customer searches for a guitar, they're providing a set of characteristics they are looking for, not a specific guitar. Maybe that's the problem.

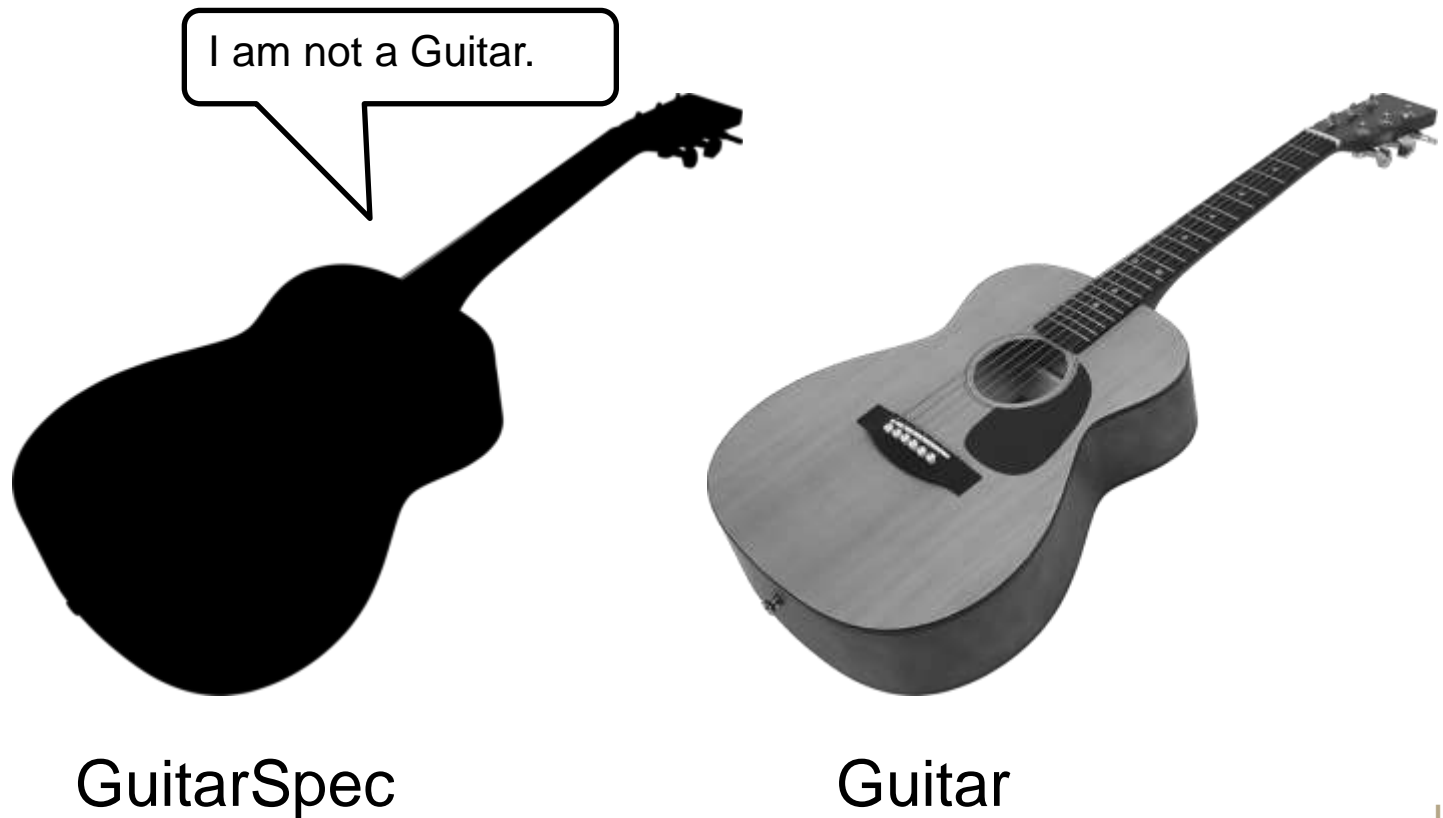


Search() in Inventory class

```
public Guitar search(Guitar searchGuitar) {
    for (Iterator i = guitars.iterator(); i.hasNext(); ) {
        Guitar guitar = (Guitar)i.next();
        String builder = searchGuitar.getBuilder().toLowerCase();
        if ((builder != null) && (!builder.equals("")) &&
            (!builder.equals(guitar.getBuilder().toLowerCase())))
            continue;
        String model = searchGuitar.getModel().toLowerCase();
        if ((model != null) && (!model.equals("")) &&
            (!model.equals(guitar.getModel().toLowerCase())))
            continue;
        String type = searchGuitar.getType().toLowerCase();
        if ((type != null) && (!searchGuitar.equals("")) &&
            (!type.equals(guitar.getType().toLowerCase())))
            continue;
        String backWood = searchGuitar.getBackWood().toLowerCase();
        if ((backWood != null) && (!backWood.equals("")) &&
            (!backWood.equals(guitar.getBackWood().toLowerCase())))
            continue;
        String topWood = searchGuitar.getTopWood().toLowerCase();
        if ((topWood != null) && (!topWood.equals("")) &&
            (!topWood.equals(guitar.getTopWood().toLowerCase())))
            continue;
        return guitar;
    }
    return null;
}
```

Solution: encapsulation

- A guitar or not a guitar
- A GuitarSpec is not a Guitar



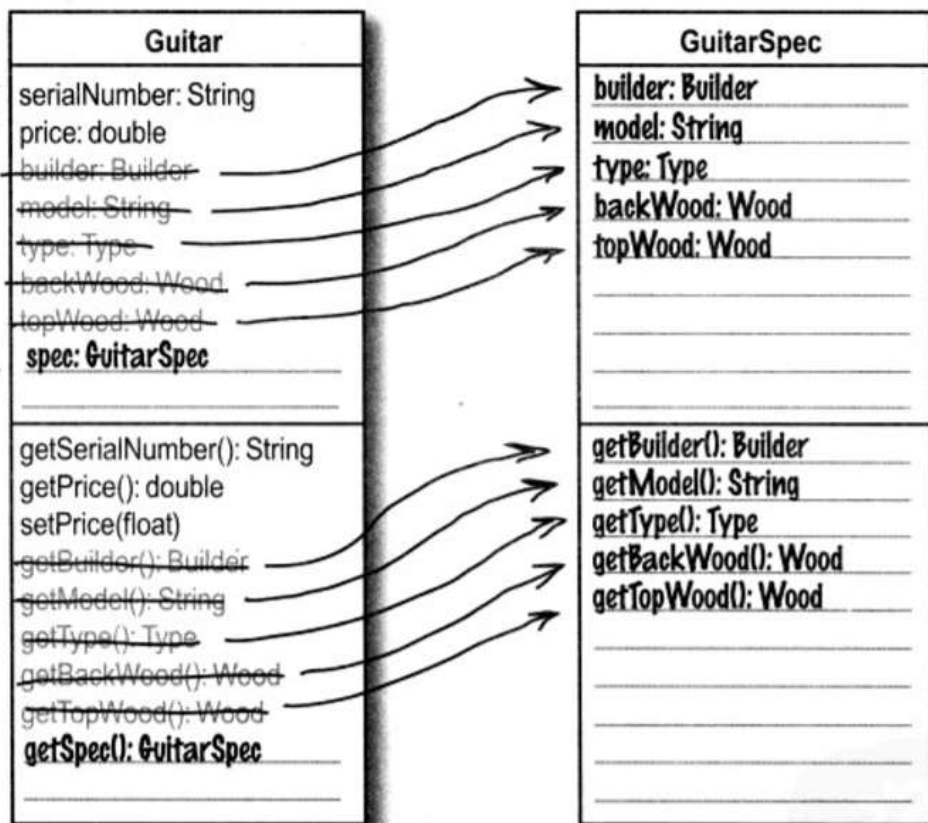
A GuitarSpec is not a Guitar-2

这两个特性对于每一把吉他而言仍是独特的，因此留下来。

这些是 Rick 的客户提供给 `search()` 的特性，因此将它们移到 `GuitarSpec`。

我们也需要为每一把吉他引用 `GuitarSpec` 对象。

这些方法遵循与特性相同的模式：我们移除任何客户规格与 `Guitar` 对象之间的重复。



通过将共同的特性（及其相关的方法）移到能同时使用在搜索及吉他细节的对象中，我们移除了重复的程序代码。

Changes to Inventory class



现在search()接受
GuitarSpec, 代替整个
Guitar 对象。

```
public class Inventory {
```

```
// variables, constructor, and other methods
```

```
public List search(GuitarSpec searchSpec) {  
    List matchingGuitars = new LinkedList();  
    for (Iterator i = guitars.iterator(); i.hasNext(); ) {  
        Guitar guitar = (Guitar)i.next();  
        GuitarSpec guitarSpec = guitar.getSpec();  
        if (searchSpec.getBuilder() != guitarSpec.getBuilder())  
            continue;  
        String model = searchSpec.getModel().toLowerCase();  
        if ((model != null) && (!model.equals("")) &&  
            (!model.equals(guitarSpec.getModel().toLowerCase())))  
            continue;  
    }  
}
```

我们用来比较吉他的信息
全在GuitarSpec中, 而不是
Guitar类。

Problem4



Solution: add new data to GuitarSpec



We can add the number of strings. It's not a problem.

Changes to GuitarSpec

```
public class GuitarSpec {

    private Builder builder;
    private String model;
    private Type type;
    private int numStrings;
    private Wood backWood;
    private Wood topWood;

    public GuitarSpec(Builder builder, String model, Type type,
                     int numStrings, Wood backWood, Wood topWood) {
        this.builder = builder;
        this.model = model;
        this.type = type;
        this.numStrings = numStrings;
        this.backWood = backWood;
        this.topWood = topWood;
    }
    ...

    public int getNumStrings() {
        return numStrings;
    }
    ...
}
```