

《面向对象设计与构造》

Lec3-抽象层次结构

2022

OO课程组

北京航空航天大学

内容

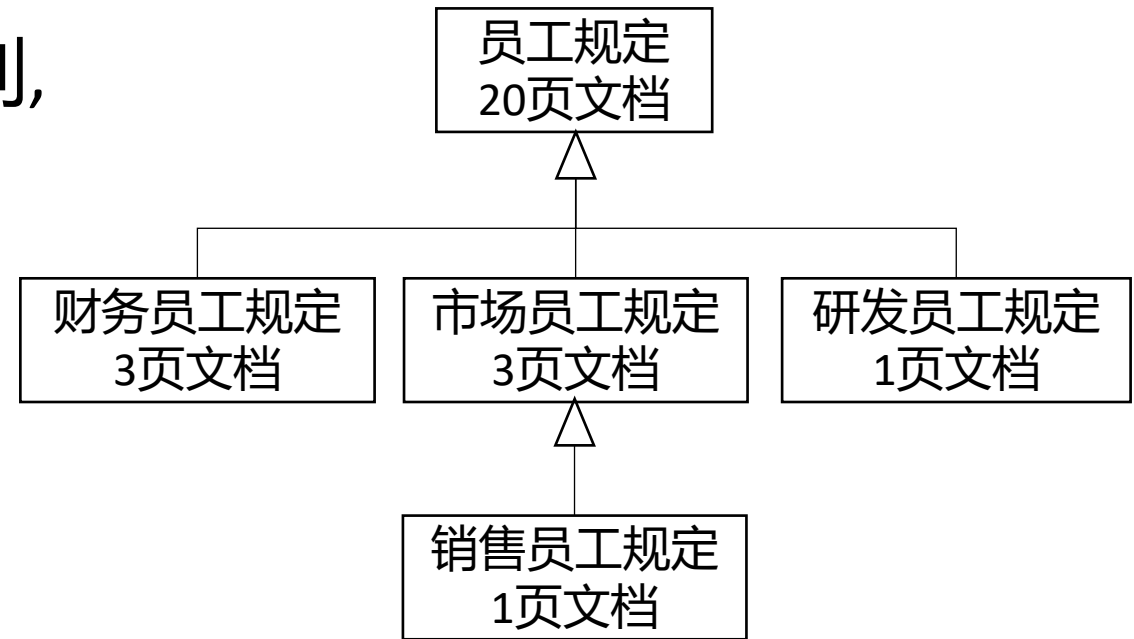
- 什么是抽象层次
- 为什么需要抽象层次
- 建立抽象层次结构
- 层次结构的选择
- 跨层次交互与执行分析
- 抽象层次结构下的归一化处理
- 作业解析

什么是抽象层次

- 数据抽象层次
 - 上层数据可以概括下层数据
 - 下层数据可以扩展上层数据
 - 面向对象表达：继承
- 行为抽象层次
 - 上层行为可以概括下层行为
 - 下层行为可以扩展上层行为
 - 面向对象表达：继承或接口实现

为什么需要抽象层次

- 设有一个员工管理系统
- 基本规定: 工作时间安排, 假期, 福利, 工作守则 ...
 - 所有新招聘员工都要学习基本规定
 - 基本规定手册(比如20页)
- 每个部门都有自己具体的规定
 - 相应部门的员工必须学习(一般1~3页)
 - 既有新增规定
 - 也有对基本规定条款的补充



领域特征分析

- 为什么不针对财务员工直接制定23页的员工规定？ 24页的市场销售员工规定， 21页的研发员工规定？
- 采用多个有侧重点的员工规定的优点：
 - 便于维护：如果公司员工基本规定发生变化，只需修改一次
 - 局部化：市场销售员工的相关规定是局部规定，不对其他员工产生影响
- 领域事实
 - 一般性的员工规定本身就有意义（统一培训等）
 - 部门具体规定也具有重要意义

员工规定

- 考虑下面的员工规定
 - 一周工作40小时
 - 普通员工工资年薪80000, 研发人员增加50000, 市场人员增加30000, 销售人员增加50000
 - 每年2周假期, 销售人员额外增加一周
 - 销售人员使用粉色表格, 其他所有员工报销使用黄色表格
- 每个部门的员工都有特定的职责
 - 财务人员审查报销单据
 - 市场人员负责收集产品的市场反馈
 - 研发人员负责按照客户需求研制产品
 - 销售人员负责产品宣传和销售

- 任务：基于Employee来实现FinancialOfficer类
- 有三种方式
 - 独立方式(冗余方式)
 - 共享方式
 - 抽象层次方式(继承方式)

```
public class Employee {  
    private int weekHours;  
    private double yearSalary;  
    private int yearVacation;  
    private String reForm;  
    public Employee(){  
        weekHours = 40; // 一周工作40小时  
        yearSalary = 80000.0; // 年薪RMB80000  
        yearVacation = 10; // 10天假期  
        reForm = "yellow"; // 报销使用黄色表格  
    }  
    public int getHours() {  
        return weekHours;  
    }  
    public double getSalary() {  
        return yearSalary;  
    }  
    public int getVacationDays() {  
        return yearVacation;  
    }  
    public String getReimbursementForm() {  
        return reForm;  
    }  
}
```

冗余的实现方案

- 这种实现方案割裂了问题域中基本员工规定和部门员工规定直接的关联关系
- 一旦需要调整规定，往往需要修改多个类的实现
- 要调整的规定层次越高，需要修改的类就越多
- **不好的实现方式**

```
public class FinancialOfficer {  
    private int weekHours;  
    private double yearSalary;  
    private int yearVacation;  
    private String reForm;  
    public FinancialOfficer () {  
        weekHours = 40; // 一周工作40小时  
        yearSalary = 80000.0; // 年薪RMB80000  
        yearVacation = 10; // 10天假期  
        reForm = "yellow"; // 报销使用黄色表格  
    }  
    public int getHours() {  
        return weekHours;  
    }  
    public double getSalary() {  
        return yearSalary;  
    }  
    public int getVacationDays() {  
        return yearVacation;  
    }  
    public String getReimbursementForm() {  
        return reForm;  
    }  
    public boolean inspect(Bill[] b) { // 审查报销单据  
        //do the inspection  
    }  
}
```


基于共享的实现方案

- FinancialOfficer共享了Employee所管理的数据，没有出现数据冗余
- FinancialOfficer重复实现Employee类的所有方法，代码冗余
 - 如果Employee类增加了新方法，FinancialOfficer也必须增加实现相应的方法
- 我们期望更简洁的实现方式

```
public class FinancialOfficer {  
    private Employee em;  
    public FinancialOfficer () {  
        em = new Employee(); //初始化em  
    }  
    public int getHours() {  
        return em.getHours();  
    }  
    public double getSalary() {  
        return em.getSalary();  
    }  
    public int getVacationDays() {  
        return em.getVacationDays();  
    }  
    public String getReimbursementForm() {  
        return em.getReimbursementForm();  
    }  
    public boolean inspect(Bill[] b) { //审查报销单据  
        //do the inspection  
    }  
}
```

建立抽象层次的实现方案

- FinancialOfficer无需冗余实现Employee中已经定义的属性和方法
- 只需要增加其所在部门规定的专门规则即可
 - 即inspect方法
- 甚至可以忽略构造方法
- 继承是重用已有类的内容构造新类的机制
 - 自动获得父类的所有属性和方法
 - **但不见得都可以直接访问**
- 继承是一种扩展已有类的机制
 - 增加必要的属性和方法
 - 必要时可以覆盖父类的方法
- 类之间形成层次结构(**数据抽象层次**)

```
public class FinancialOfficer extends Employee {  
    public boolean inspect(Bill[] b) { // 审查报销单据  
        //do the inspection  
    }  
}
```

使用继承机制来实现市场人员类

- 市场人员的相关规定:
 - 多获得30000的年薪
 - 有特定职责：收集市场反馈
- 要求从Employee继承

Employee类提供了这个方法

```
public double getSalary() {  
    return yearSalary;  
}
```



```
public class Marketer extends Employee{  
    public double getSalary() {  
        return yearSalary+30000;  
    }  
}
```



The field Employee.yearSalary is not visible,
为什么?

必须调用Employee类的getSalary方法!

```
public class Marketer extends Employee{  
    public double getSalary() {  
        return super.getSalary()+30000;  
    }  
}
```

使用继承机制来实现销售人员类

- 销售人员隶属于市场部的人员
 - 年薪比市场人员多20000
 - 年假比市场人员多1周
 - 销售人员使用粉色表格
 - 负责产品的销售和宣传

```
public class Employee {  
    ...  
    protected String reForm;  
    ...  
}  
  
    yearSalary = 80000.0; // 年薪RMB80000  
    ...  
    用黄色表格  
}  
  
    Seller(){  
        super();  
        reForm = "pink";  
    }  
}
```

```
public class Seller extends Marketer{  
    public double getSalary() {  
        return super.getSalary()+20000;  
    }  
    public int getVacationDays() {  
        return super.getVacationDays()+5;  
    }  
    public String getReimbursementForm() {  
        return super.getReimbursementForm();  
    }  
    public void advertise(Product p) {  
        // do advertisement  
    }  
}
```

行为抽象层次

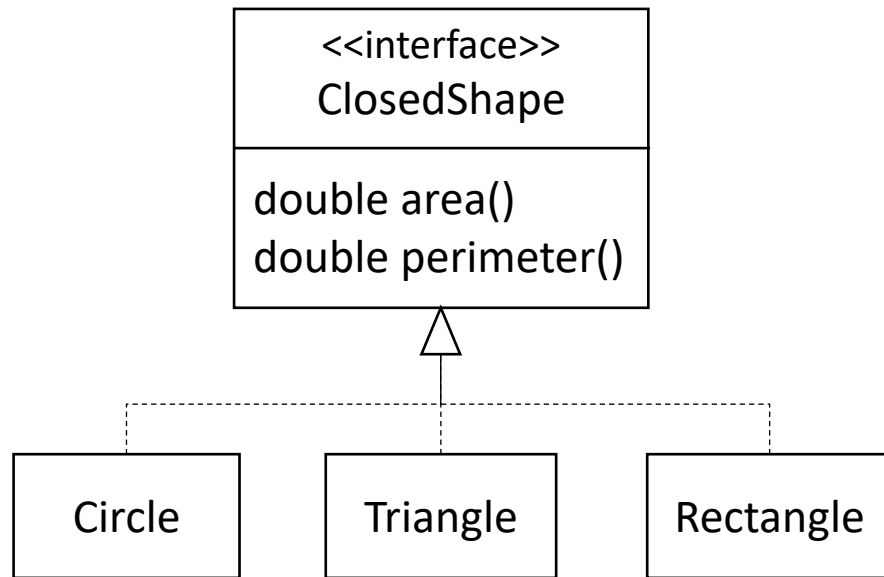
- 有些数据没有共性特征可识别
 - 无法建立数据抽象层次
 - 但是对这些数据有统一的操作
 - 针对Circle, Rectangle和Triangle求周长和面积
- 所有封闭的二维几何形状都可以计算外周长和面积
 - 但每一种形状的计算方式都不同
 - 建立行为抽象层次
- 因子有多种
 - 数据抽象层次 or 行为抽象层次?

通过接口来建立行为抽象层次

```
public class Circle implements ClosedShape {  
    private double radius;  
    public Circle(double radius) {  
        this.radius = radius;  
    }  
    public double area() {  
        return Math.PI * radius * radius;  
    }  
    public double perimeter() {  
        return 2.0 * Math.PI * radius;  
    }  
}
```

```
int i;  
double t_area = 0.0;  
ClosedShape shapes[] = {new Circle(10), new  
Triangle(3,5,7), new Rectangle(10,25)};  
for(i=0;i<shapes.length;i++)  
    t_area += shapes[i].area();
```

```
public interface ClosedShape {  
    public double area();  
    public double perimeter();  
}
```



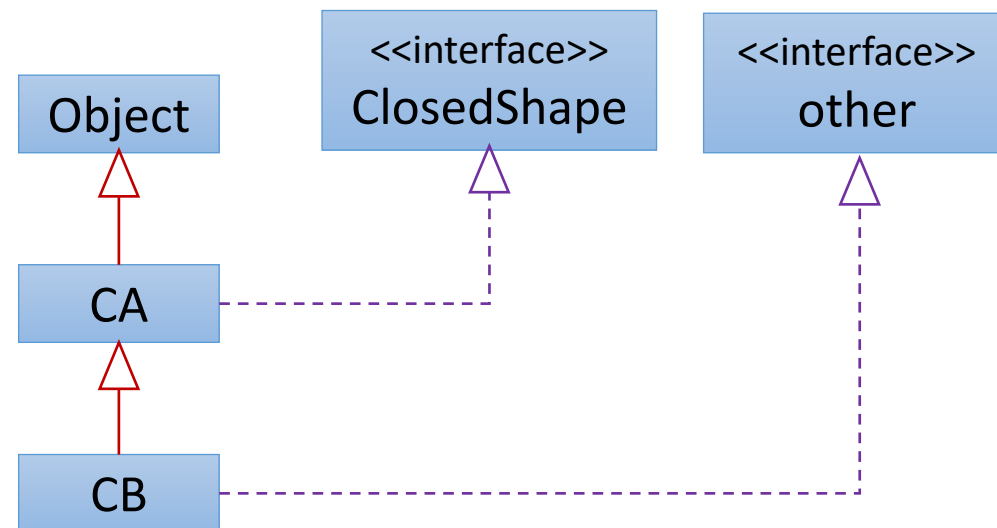
通过接口来建立行为抽象层次

- **interface:** 对一组类共性行为的抽取结果，使得行为设计和行为实现相分离
 - interface同时也是一个类型，可以用来声明对象引用
 - interface间甚至也可以建立继承关系
- ClosedShape为封闭几何图形定义了共性行为`perimeter`和`area`
- 效果：客户代码无需区分不同的几何类型
 - 能够通过ClosedShape来引用任何实现了该接口的几何图形对象
 - 获得所引用对象的面积和周长（无论其实际形状如何）

两种抽象层次结构的混合使用

- 对于Java，任意一个类必然处于一个继承链中
 - 默认情况下是Object-MySelf继承链
- 任意一个类，可以处于0或者多个接口实现链中
 - 一个类可以实现0到多个接口

```
public class CA implements ClosedShape {}  
public class CB extends CA implements other {}
```



Java通过Object类建立统一的抽象层次

- Java语言预定义了一个根类Object，除了原子数据类型(primitive type)，任何类(包括用户定义的类)都默认是Object的子类
- Object提供了三个重要方法
 - `public boolean equals(Object o)`
 - `protected Object clone()`
 - `public String toString()`

为什么要设计这个Object?

- (1)需要一个统一的类型体系(概括能力)
- (2)需要在运行时处理所有对象的能力

抽象层次结构下的对象引用与访问

- 基于所建立的抽象层次，可以使用上层类型的对象引用来无差别的引用和访问使用本层及下层类型创建的任意对象
 - 可以使用Object类型的变量引用任意类型的对象
 - `Object o = new Poly(...);`
- 限制是只能访问对象引用类型所定义的方法
 - 不能通过o来访问Poly的degree方法

假设有个ArrayList容器(al)，里面存储了多种类型的对象，需要确保容器中没有管理**冗余的对象**，即任意两个对象引用指向的对象都是不同的

```
Object pre, next;
for( i=0; i<al.size();i++){
    pre = al.get(i);
    for( j=i; j<al.size();j++){
        next = al.get(j);
        比较pre和next是否为相同对象
    }
}
```

抽象层次结构下的对象等同关系判断

- 通过上层类型来引用下层类型创建的对象带来访问上的便利，但对象可见内容变得“少”了
- Java每个对象都提供equals方法，检查对象间的等同关系
 - 类型是否相似
 - 类型是否相同
 - 状态是否相同
 - 对象是否相同
- 可变对象：两个对象是相同对象(即共享)，equals方法返回true
 - Object类提供的默认实现
- 不可变对象：只要两个对象状态相同，就应该认为是相同对象
 - 需重新实现equals方法

对象相同 → 对象状态相同 → 对象类型相同 → 对象类型相似

重新实现equals方法

- 基于对象属性取值来判断对象等同关系
 - 不可变对象
 - 对象内部状态级别的检查
- 重新实现equals确保层次结构下对象等同判断行为的一致性

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

```
public boolean equals(Counter o) {  
    if(o==null) return false;  
    return o.count == this.count;  
}
```

```
public class Counter{  
    private int count;  
    //method definitions...  
}
```

```
public boolean equals(Object o) {  
    if(o==null) return false;  
    if(this == o) return true;  
    if(o instanceof Counter)  
        return this.count == ((Counter)o).count;  
    return false;  
}
```

四种层次结构

- 数据抽象层次结构
- 行为抽象层次结构
- 组合层次结构
- 递归层次结构

从数据状态维度 在类之间建立的is-a关

从行为能力维度建立的generalization关系 上层

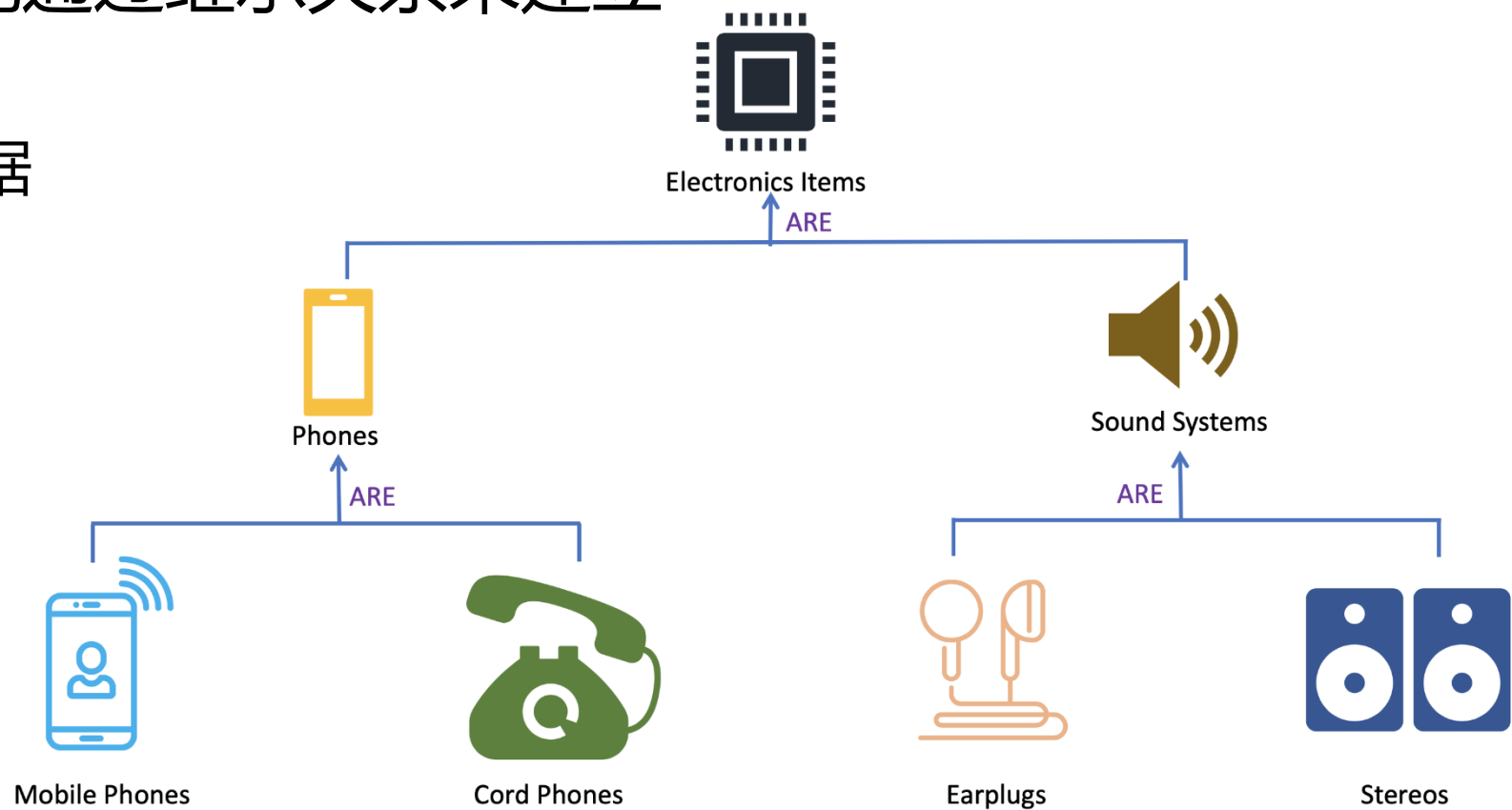
从构成维度建立的part-of关系 上层来通

在数据抽象层次上建立的反向组合层次，
下层的某个数据引用到上层的抽象数据

表达式由项组成，项由因子组成，因子包
括**常量因子**、**变量因子**、**表达式因子**、**函
数因子**，表达式因子引用到表达式

如何建立数据抽象层次

- 数据抽象层次只能通过继承关系来建立
- 两种场景
 - 向上提炼共性数据
 - 向下重用和扩展



向上提炼共性数据

- 设备管理系统
 - 设备类别：计算机，打印机，服务器，硬盘，办公家具等
 - 设备数据：ID、名称、生产公司、购买人、责任人、存放地点等
 - 每个设备均有特定的规格数据
- 方案A：建立一个设备类，通过枚举来标识不同的设备类型
 - 每个类别的设备都有不同的规格数据
- 方案B：为每一类设备单独建立一个类
 - 大量的数据冗余

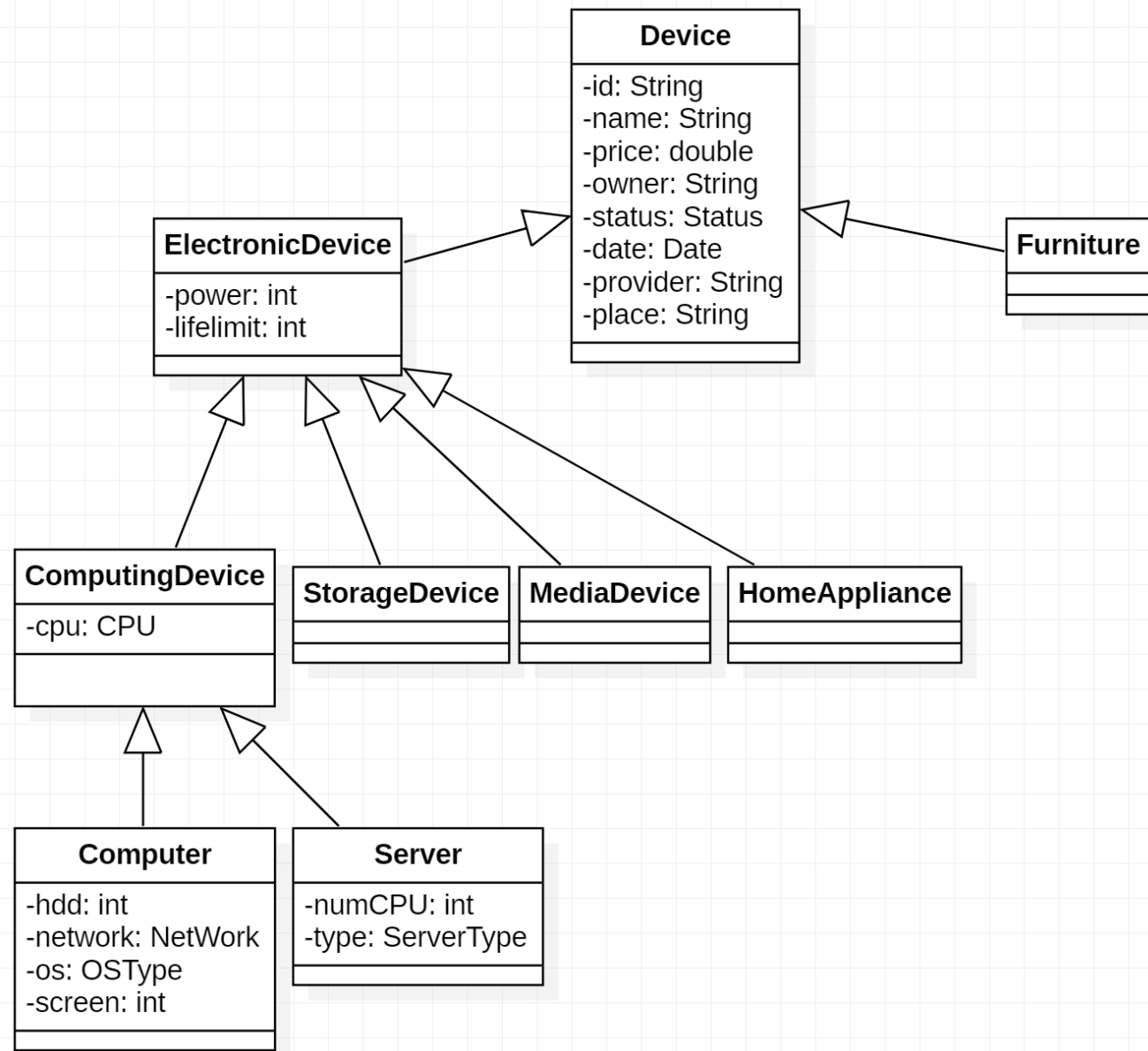


向上提炼共性数据

- 设备管理系统的一个核心功能是清查设备台账
 - 按照类别来清点设备，如存储、服务器、扫描仪等
 - 按照状态来清点设备，比如是否可用，是否有故障，是否已报废等
 - 按照规格来清点设备，比如金额、存储容量、功率等
- 方案A无法区分不同类别设备的规格，不具有扩展性，难以处理
- 方案B为每种设备单独设计一个类，查询的客户代码繁琐
 - 难以统一查询、查询结果难以归纳
- 方法C：A + B
 - 建立通用的Device类
 - 进一步区分不同类别的设备：ElectronicDevice, Furniture, ComputingDevice,...
 - 建立抽象层次(继承)

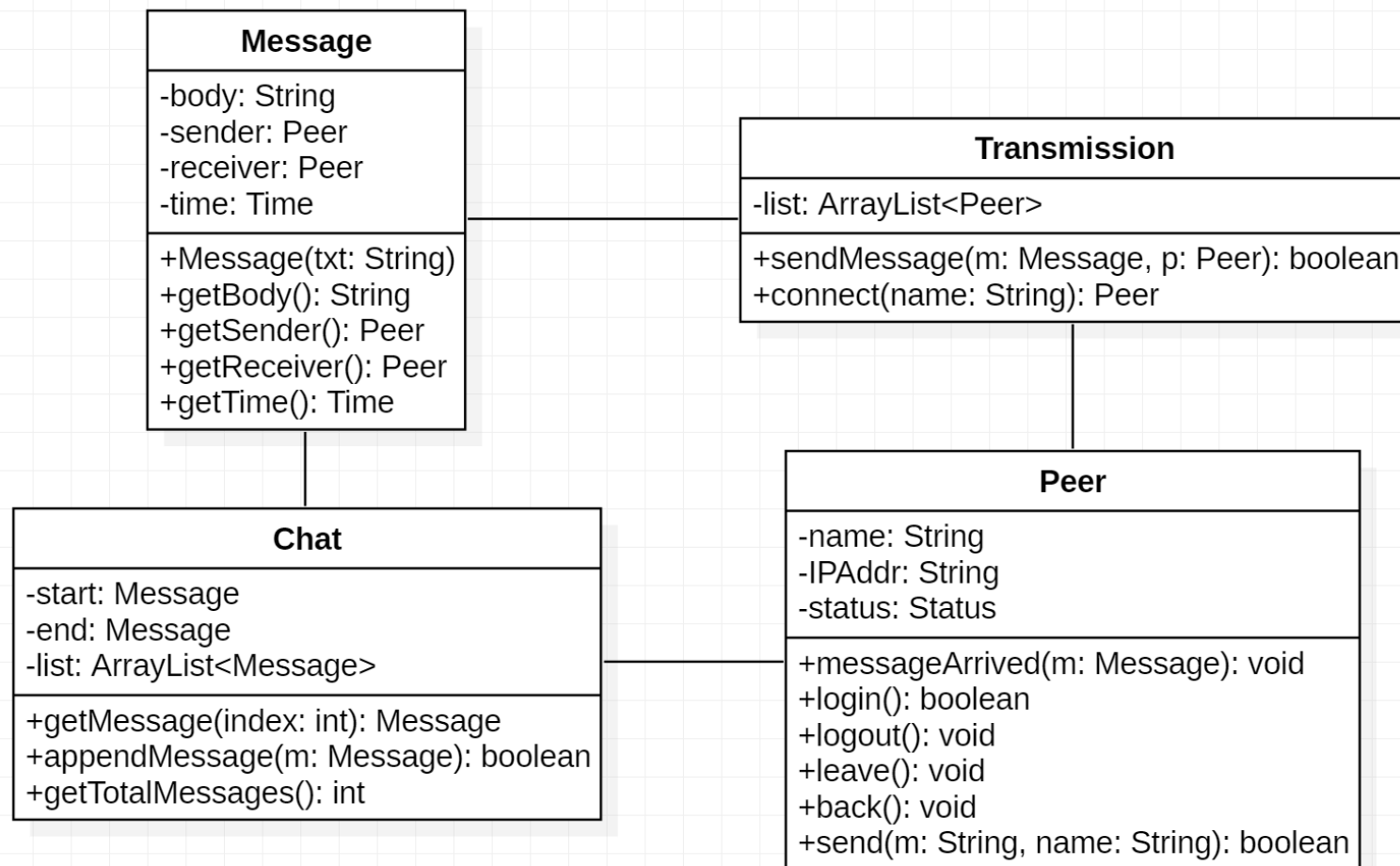
向上提炼共性数据

- 效果1：可以消除多个类中的冗余数据，精简代码
- 效果2：可以通过上层类来引用其任意子类的实例化对象，获得统一化的遍历式处理，便捷查询
 - 哪些设备功率超过500w?
 - 哪些设备的价格超过5w?
 - 哪些设备已超过设计寿命?
 - 哪些设备安装了微软OS?



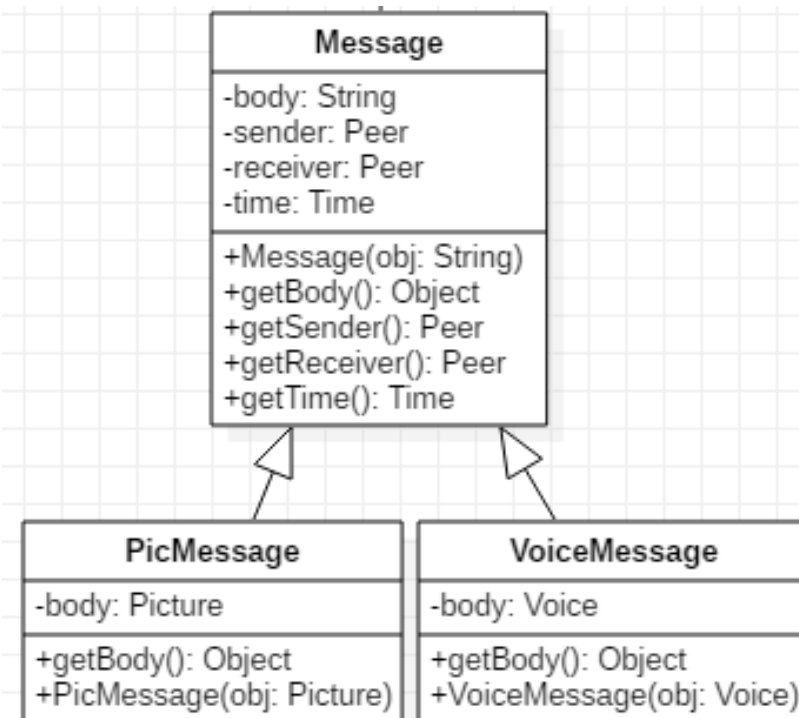
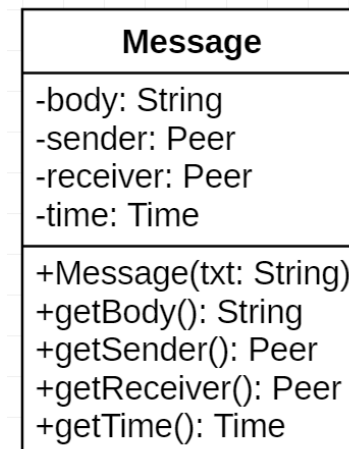
向下重用和扩展

- IMS系统
 - 数据维度和处理维度同步演化到V3
 - 支持文本、图片和音频消息
 - 支持点对点、点对多和多对多三种通讯模式
- 如何重用和扩展 Message 类?
- 如何重用和扩展 Transmission 类?



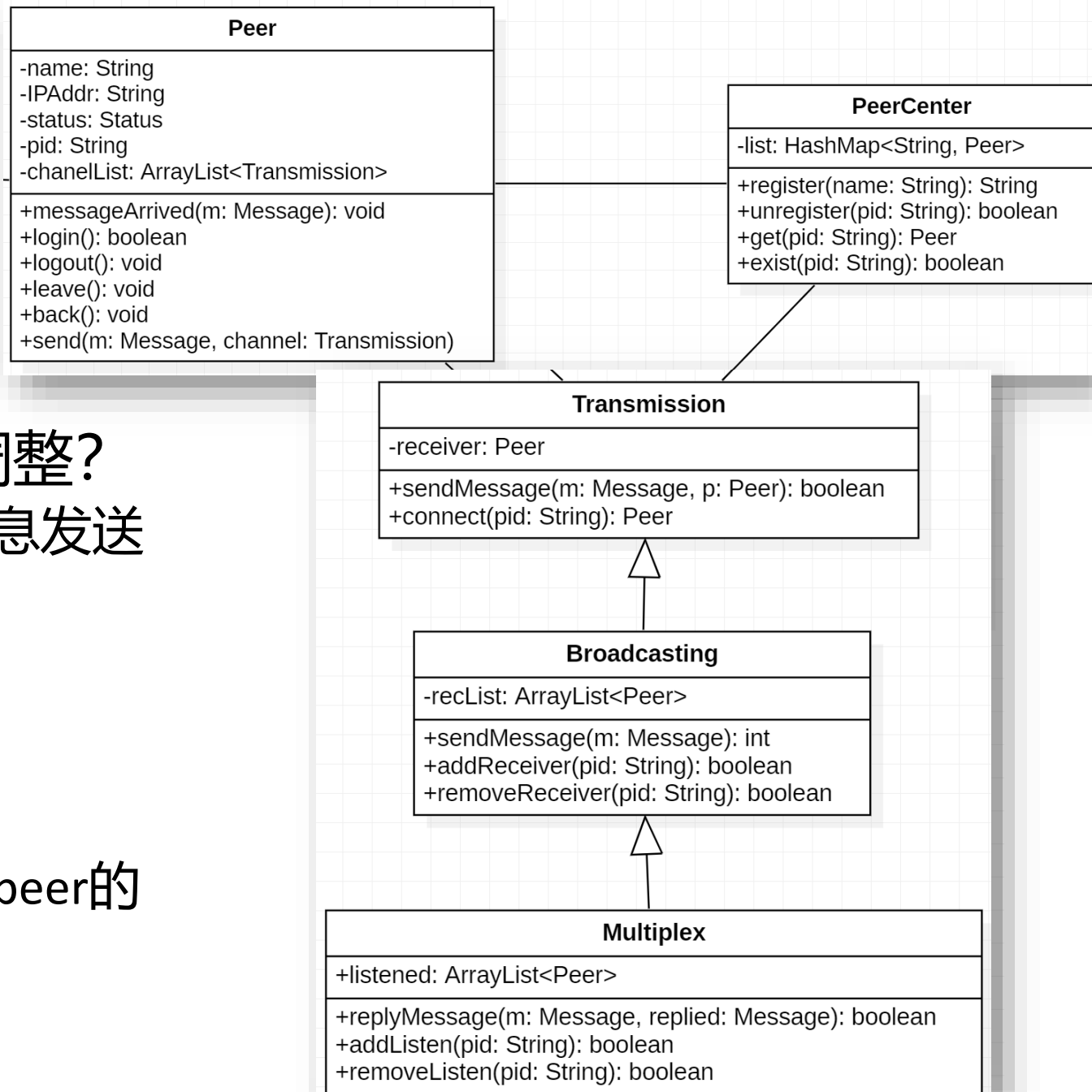
向下重用和扩展

- 保留Message，并让它表示文本消息
 - 增加PicMessage和VoiceMessage来继承它
 - PicMessage和VoiceMessage各自重新定义body属性的类型和初始化方式
 - Message类：getBody():String
 - PicMessage类：getBody():String, Picture or Object?
 - VoiceMessage类：getBody():String, Voice or Object?
- 子类可以重写(override)父类方法，但是必须确保返回值类型一致



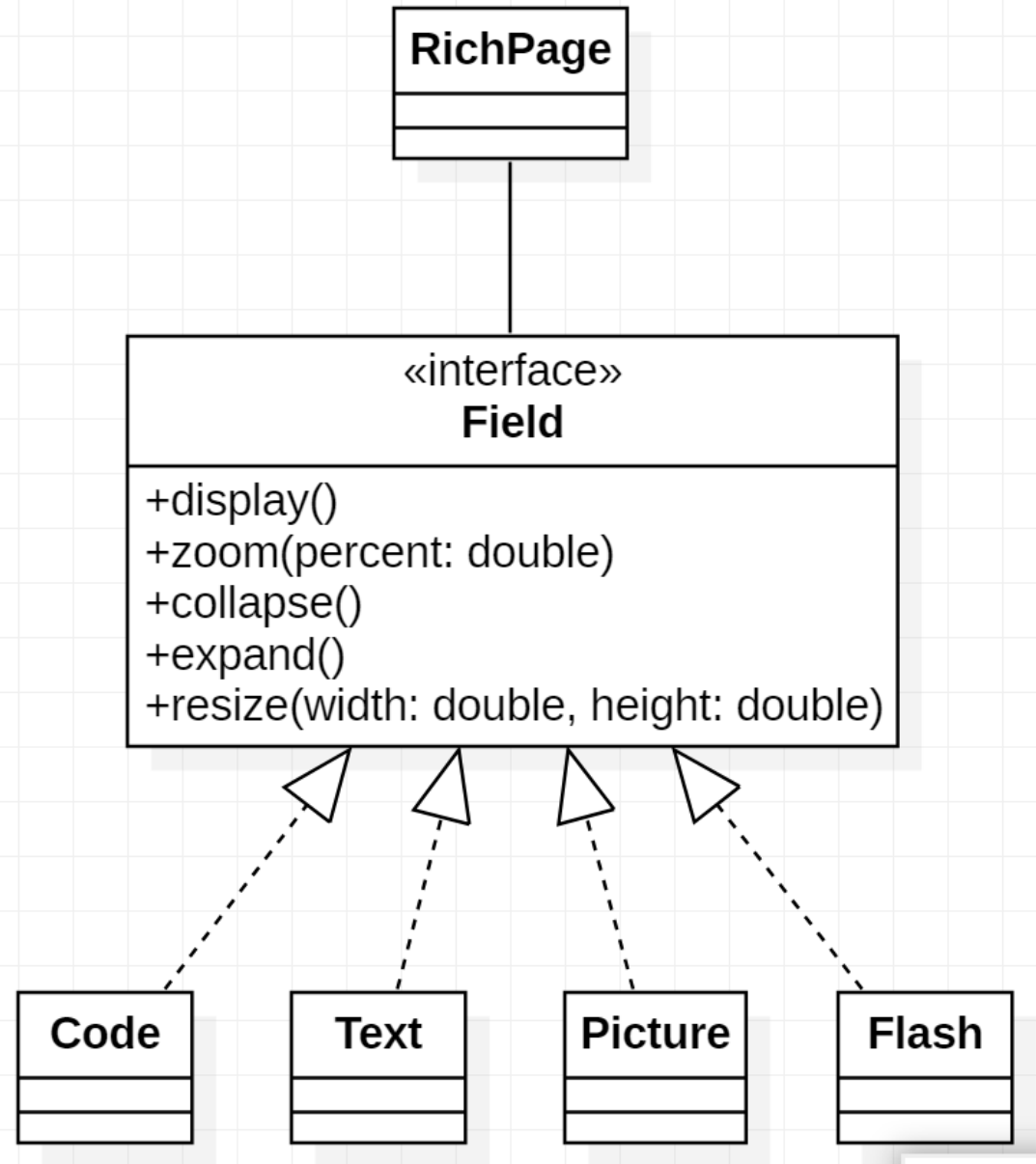
向下重用和扩展

- Transmission是否管的太多?
 - Peer管理不应该是它的职责
- Peer类的send方法是否需要调整?
 - send需要传输信道才能完成消息发送
 - 不能只传送文本消息了
- Transmission默认点对点传输
 - 增加广播传输Broadcasting
 - 增加群组多对多传输Multiplex
 - 群组传输甚至还支持关注特定peer的消息



如何建立行为抽象层次

- 我们有时只关注一些类的共性行为
 - RichPage, 管理着Code, Text, Image, Flash等Field对象
 - Field对象的共性行为: display, zoom, collapse, expand, resize等
- 这些Field对象也许在数据上有共性, 但RichPage不关心



抽象层次结构的选择

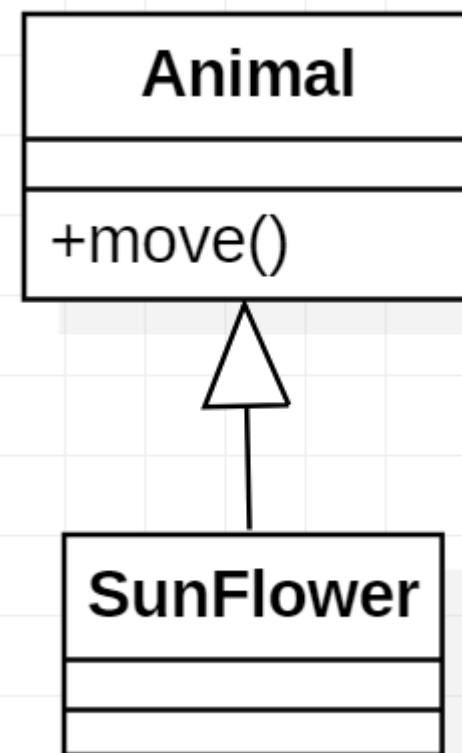
- 有些程序需要有灵活扩展能力，允许增加新的类，同时保持客户代码的原有结构
- 假设有个宠物饲养Game，玩家每天都要带着宠物锻炼
 - 游戏不断投入各种宠物
 - 高级玩家甚至可以创造宠物
- 按照一般理解，动物可以按照纲目划分，建立继承层次
 - 父类为Animal

```
ArrayList feedAnimals;  
for(int i=0;i<feedAnimals.size();i++){  
    Animal am = feedAnimals.get(i);  
    if(am instanceof Bird) ((Bird)am).fly();  
    if(am instanceof Dog) ((Dog)am).walk();  
    if(am instanceof Fish) ((Fish)am).swim();  
    ...  
}
```

```
ArrayList<Animal> feedAnimals;  
for(int i=0;i<feedAnimals.size();i++){  
    Animal am = feedAnimals.get(i);  
    am.move();  
}
```

抽象层次结构的选择

- 奔跑的向日葵如何？



抽象层次结构的选择

```
public interface moveable{  
    public void walk();    //慢速运动  
    public void run();    //中速运动  
    public void rush();    //快速运动  
}
```

```
ArrayList<moveable> feedAnimals;  
moveable ma;  
for(int i=0;i<feedAnimals.size();i++){  
    ma = feedAnimals.get(i);  
    if(in_walkmode)ma.walk();  
    if(in_runmode)ma.run();  
    if(in_rushmode)ma.rush();  
}
```


方法重写场景下的行为推理

```
public class Foo {  
    public void method1() {  
        System.out.println("foo 1");  
    }  
    public void method2() {  
        System.out.println("foo 2");  
    }  
    public String toString() {  
        return "foo";  
    }  
}  
  
public class Bar extends Foo {  
    public void method2() {  
        System.out.println("bar 2");  
    }  
}
```

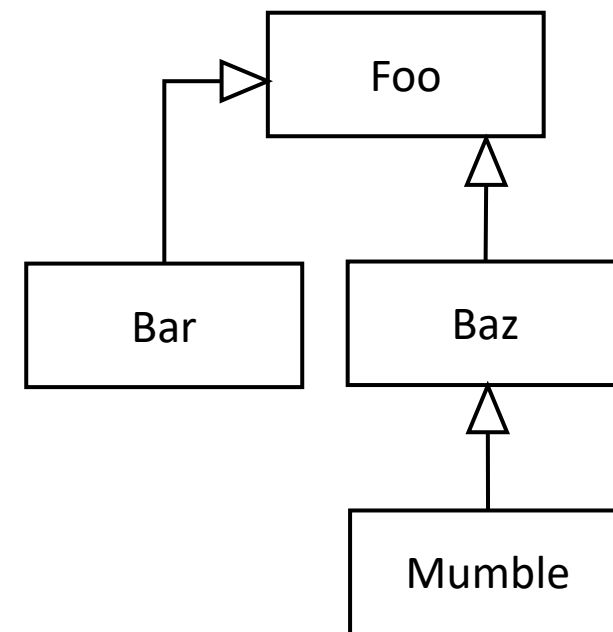
同学们可以在代码上推理执行结果,
并把代码编译运行看看实际情况

```
public class Baz extends Foo {  
    public void method1() {  
        System.out.println("baz 1");  
    }  
    public String toString() {  
        return "baz";  
    }  
}  
  
public class Mumble extends Baz {  
    public void method2() {  
        System.out.println("mumble 2");  
    }  
}
```

```
Foo[] pity = {new Baz(), new Bar(), new  
    Mumble(), new Foo()};  
for (int i = 0; i < pity.length; i++) {  
    System.out.println(pity[i]);  
    pity[i].method1();  
    pity[i].method2();  
    System.out.println();  
}
```

抽象层次结构下的方法执行分析表

method	Foo	Bar	Baz	Mumble
method1	foo 1	<i>foo 1</i>	baz 1	<i>baz 1</i>
method2	foo 2	bar 2	<i>foo 2</i>	mumble 2
toString	foo	<i>foo</i>	baz	<i>baz</i>



跨层次方法调用场景下的行为推理

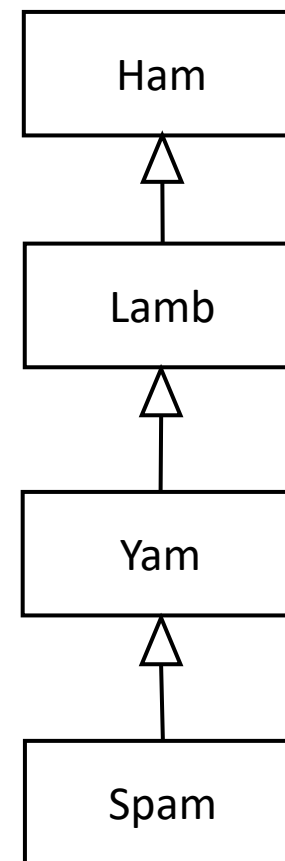
```
public class Ham {
    public void a() {
        System.out.print("Ham a    ");
        b(); }
    public void b() {
        System.out.print("Ham b    "); }
    public String toString() {
        return "Ham"; }
}
public class Lamb extends Ham {
    public void b() {
        System.out.print("Lamb b    "); }
}
```

```
public class Yam extends Lamb {
    public void a() {
        System.out.print("Yam a    ");
        super.a(); }
    public String toString() {
        return "Yam"; }
}
public class Spam extends Yam {
    public void b() {
        System.out.print("Spam b    "); }
}
```

```
Ham[] food = {new Lamb(), new Ham(), new Spam(), new Yam()};
for (int i = 0; i < food.length; i++) {
    System.out.println(food[i]);
    food[i].a();
    System.out.println();
    food[i].b();
    System.out.println();
}
```

抽象层次结构下的方法执行分析表

method	Ham	Lamb	Yam	Spam
a	Ham a b()	<i>Ham a</i> <i>b()</i>	Yam a Ham a b()	<i>Yam a</i> <i>Ham a</i> <i>b()</i>
b	Ham b	Lamb b	Lamb b	Spam b
toString	Ham	<i>Ham</i>	Yam	<i>Yam</i>



跨层次交互

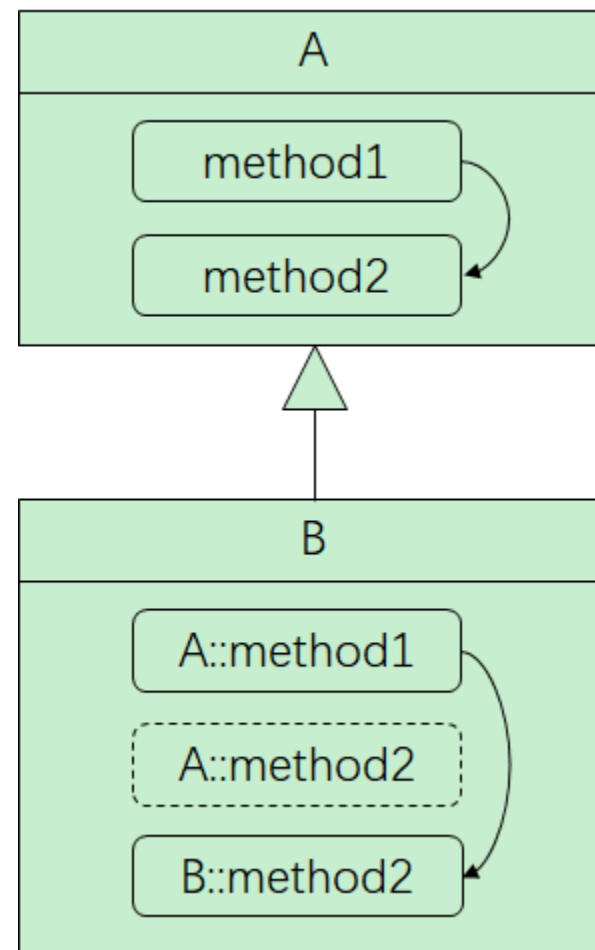
- 规范管理差旅报销，规定了能够报销的项目，如交通和住宿
 - Employee类定义了差旅报销方法
 - 由各个部门规定具体的交通和住宿报销标准

```
public class Employee{  
    ...  
    public float getReimbursement(Bill[] b){  
        float ret = 0.0;  
        for(int i=0;i<b.length;i++){  
            if(!b[i].inspect())continue;//必须是有效的票据  
            if(!b[i].isTransportation() && !b[i].isHotel()) continue;  
            ret += billcheck(b[i]);  
        }  
        return ret;  
    }  
}
```

Developer重写billCheck方法，比如限制住宿标准，超过限制按照标准顶额报销

跨层次交互

- 向上交互
 - 使用super来访问父类所定义的属性和方法
 - 访问的对象仍然是this对象
- 向下交互
 - 父类访问子类的方法?
 - 前提条件：子类重写了父类的方法
 - 父类method1调用method2
 - 子类重写了父类的method2



讨论题

- MyHashSet扩展了HashSet类，重写了add和addAll方法，想要记录用户向容器中加入了多少个元素
- 对于下面的客户代码，请问输出是多少？为什么？

```
class MyHashSet<T> extends HashSet<T> {  
    private int addCount;  
    public MyHashSet() {super(); addCount=0;}  
    public boolean add(T a) {  
        addCount++;  
        return super.add(a);  
    }  
    public boolean addAll(Collection<? extends T> c){  
        addCount += c.size();  
        return super.addAll(c);  
    }  
    public int getAddCount() {  
        return addCount;  
    }  
}
```

```
MyHashSet<String> mhs = new MyHashSet<>();  
mhs.add("A");  
mhs.add("B");  
mhs.add("C");  
mhs.addAll(Arrays.asList("D", "E", "F"));  
System.out.println(mhs.getAddCount());
```

多态是面向对象的核心概念

- **Polymorphism: poly (many) + morphism (forms)**
- 人会说话，在不同社会角色下的说话方式是不同的
 - 作为单位职员的说话方式
 - 作为家庭成员的说话方式
 - 作为朋友交往的说话方式
- 我们希望能够根据其社会角色的不同来理解其说话方式
- 对于程序而言，我们希望根据类的职责来设计和理解其“说话”方式

多态是面向对象的核心概念

- 多态就是提供多种形态、但从外部看来可统一的方法
- 多态也是一种设计与实现机制，获得**不同**但可**归一化的**对象行为
 - 通过overload机制而获得的归一化方法（重载），静态多态
 - 通过override机制而获得的归一化方法（重写），动态多态
 - 通过接口实现机制而获得的归一化方法，动态多态
- 多态机制为表达复杂设计逻辑提供了简化手段
 - Overload：按照不同业务场景来分别提供方法实现，避免一个方法处理多种场景，且多个方法的功能目标**可统一**
 - Override/接口实现：每个类根据其职责(重新)实现相应的方法，与上层所定义方法在使用方式上**可统一**

基于重载的静态多态(Overload)

- 从设计角度看，一个方法规范了类的一种行为能力
- 随着功能的扩展或演化，类需要处理更多的不同场景
- 这时可以区分场景来进行扩展
 - 求和(add)是一个类math的方法，`int add(int, int)`
 - 现在有新的场景，要求支持整数与复数的综合加法：`int add(int, Complex)`
 - 更进一步扩展，支持对不定长集合元素（整数或复数）的求和：`int add(ArrayList)`
- 重载方法间通过参数来区分，编译时(静态)解析方法的多态性

基于重写的动态多态(Override)

```
public class EmployeeMain {  
    public static void main(String[] args) {  
        Developer lisa = new Developer();  
        Seller steve = new Seller();  
        printInfo(lisa);  
        printInfo(steve);  
    }  
    public static void printInfo(Employee e) {  
        System.out.println("salary:"+e.getSalary());  
        System.out.println("v.days:"+  
            e.getVacationDays());  
        System.out.println("r.form:"+  
            e.getReimbursementForm());  
        System.out.println();  
    }  
}
```

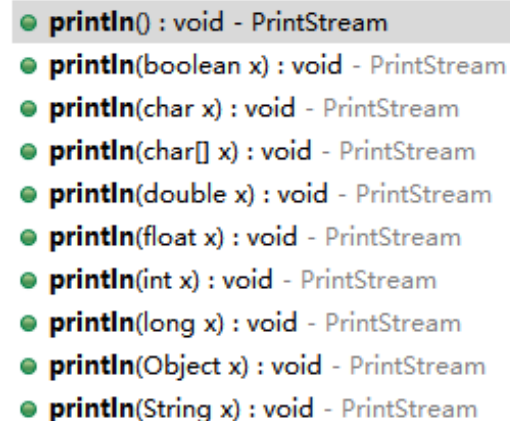
```
salary:130000.0  
v.days:10  
r.form:yellow
```

```
salary:130000.0  
v.days:15  
r.form:pink
```

重写发生在父类与子类间，子类重新实现从父类继承的方法。
调用一个对象方法时，具体调用哪个由运行时的dispatch机制来确定

归一化访问

- 归一化：无需区分对象的个性特征，按照统一方式来进行访问的机制
 - 通过输入参数类型来自动匹配待处理对象，调用者无需区分判断
 - `System.out.println(...)`
 - 通过待处理对象所实现的接口或重写的方法来调用
 - `String toString()`
- 第二种情况必然涉及到抽象层次
 - 通过上层的对象引用来归一化访问待处理对象



```
println() : void - PrintStream
println(boolean x) : void - PrintStream
println(char x) : void - PrintStream
println(char[] x) : void - PrintStream
println(double x) : void - PrintStream
println(float x) : void - PrintStream
println(int x) : void - PrintStream
println(long x) : void - PrintStream
println(Object x) : void - PrintStream
println(String x) : void - PrintStream
```

Press 'Alt+/' to show Template Proposals

通过父类来归一化访问方法

- 通过一个父类型的对象引用容器来管理子类型对象，可获得更简洁的归一化访问代码

```
public class EmployeeMain2 {  
    public static void main(String[] args) {  
        Employee[] e = { new Developer(),    new Marketer(),  
                        new Seller(), new FinancialOfficer() };  
  
        for (int i = 0; i < e.length; i++) {  
            System.out.println("salary: " + e[i].getSalary());  
            System.out.println("v.days: " + e[i].getVacationDays());  
            System.out.println();  
        }  
    }  
}
```

Output:

salary: 130000.0
v.days: 10

salary: 110000.0
v.days: 10

salary: 130000.0
v.days: 15

salary: 80000.0
v.days: 10

如何统一管理不同类型的因子，从而按照统一的方式来展开和合并？

通过接口来归一化访问方法

- 接口可规范一组类的共同行为，客户代码通过多态机制可简化对相应对象的管理
- 任何***实现了相应接口的对象***都可以传递进行处理

```
public void printInfo(ClosedShape s) {  
    System.out.println("shape:"+s);  
    System.out.println("area:"+s.area());  
    System.out.println("perim:"+  
                        s.perimeter());  
}
```

```
Circle c=new Circle(12.0);  
Rectangle r=new Rectangle(4,7);  
Triangle t=new Triangle(3,5,7);  
printInfo(c);  
printInfo(r);  
printInfo(t);
```

基于接口的归一化访问甚至更灵活

- 一个接口可以被多个类实现
 - 通过该接口可归一化访问多种类型的对象
- 一个类实现多个接口
 - 可被多个接口来归一化访问
- 多个接口定义了相同的操作
 - 类只需实现一次
 - 所实现的每个接口类型皆可归一化访问

```
public interface IA{  
    public int m();  
    public int mia();  
}
```

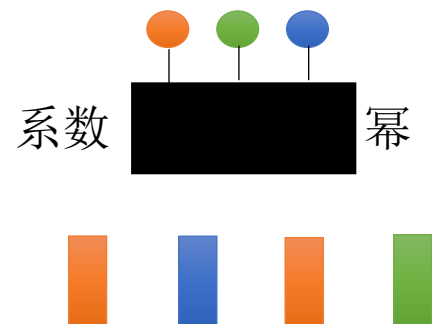
```
public interface IB{  
    public int m();  
    public int mib();  
}
```

```
public class Test implements IA, IB{  
    public int m() {...}  
    public int mia() {...}  
    public int mib() {...}  
}
```

```
IA oa = new Test();  
oa.m();  
IB ob = (Test) oa;  
ob.m();
```

作业分析

- 本次作业的复杂度主要在于引入了因子嵌套规则
 - $\sin(3x^{**2}+2x); f(x^{**3}, 1, h(x))$
- 因子的表示是关键
 - 多种类别
 - 选择接口实现还是继承来建立抽象层次?
- 因子的行为抽象
 - 乘展开
 - 代入展开
 - 合并同类项
 - 解析构造或生成
- 因子的数据抽象(函数化)
 - 系数、幂
 - 实参列表
 - 函数表达式
 - ...系数*slot**幂...



- 多种因子函数
 - 常量函数
 - 幂函数
 - 三角函数
 - 自定义函数
 - 求和函数

作业分析

- 作业1：支持数据的递归表示
- 作业2：增加多种因子结构
- 作业3：呈现更多因子组合方式
- 函数调用本质上是字符串变换
 - 字符串替换？
 - 应围绕字符串的逻辑结构来设计！

