《面向对象程序先导》Lec6-继承与接口的使用

北京航空航天大学计算机学院

吴际

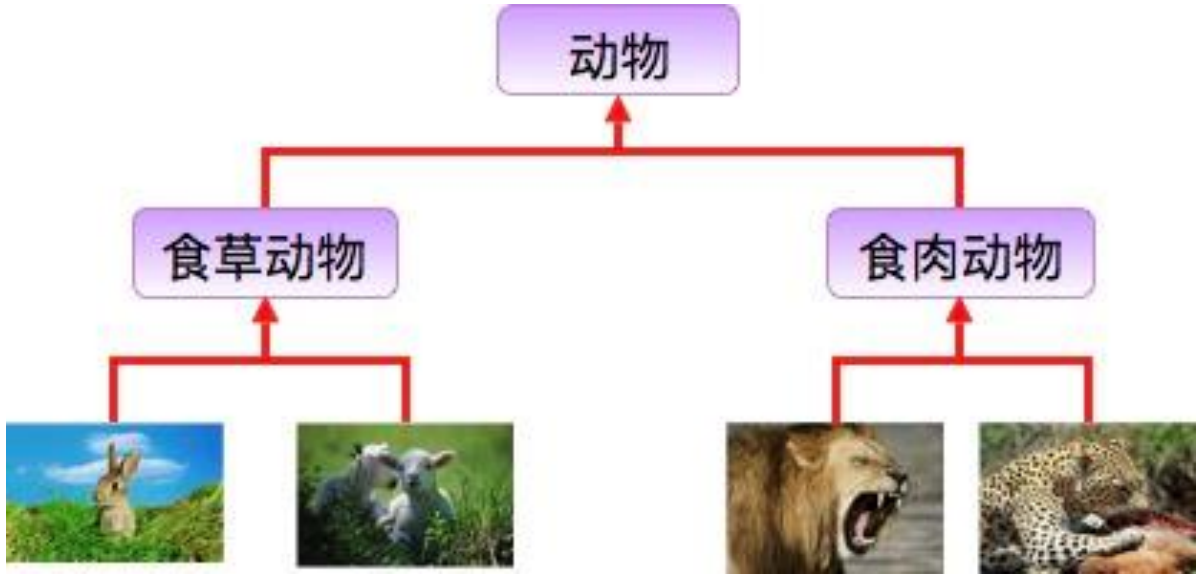
2023.10.20

内容提要

- 继承及其应用
- 接口及其应用
- 综合考虑解决实际问题
- 继承下的Junit测试要点
- 作业内容与建议

继承的定义

- 继承是类之间的一种抽象层次关系
- 继承让子类将获得父类的属性和方法，实现**复用和扩展**
- 继承可以把多个类中的重复内容提取出来形成父类，减少冗余和增强可维护性



- 复用扩展
- 提炼简化

继承在Java中的语法

- Java提供关键字**extends**来定义继承关系
  - 子类**可以访问**父类的公共（public）和受保护（protected）成员
  - 私有（private）成员无法被直接被访问，但可以通过调用父类的方法来访问
- Java只支持单继承，即一个类最多只能有一个父类
- 父类可以概括子类
  - 一个子类型对象可以被父类型变量所引用
  - 使用 **obj instanceof MyCLass** 表达式来判断对象obj是否为MyClass的实例化对象

```
class 父类名 {
    // 父类的属性和方法
}

class 子类名 extends 父类名 {
    // 子类的属性和方法
}
```

继承在Java中的语法

- 在子类中，使用 super.attribute 可以引用父类中定义的**非私有**属性
- 如果子类**重写**了父类的某个方法，那么子类通过super.methodName() 可以调用父类所实现的那个方法
  - (this.)methodName()调用的是子类实现的那个方法
- 在子类构造方法中，一般使用 super(arguments) 来调用父类的构造方法，从而完成对父类所定义属性的初始化

```
class Person {
    private String id;
    public Person(String id){
        this.id = id;
    }
    public String getId(){
        return id;
    }
    public void setId(String id){
        this.id = id;
    }
}

class Student extends Person {
    public Student(String id){
        super(id);
    }
        //不需要写getId()方法,直接用父类的即可
    public void setId(String id){
        if(id == null)return;
        if(id.length()>0)super.setId(id);
        //不可以super.id = id;
        //因为这里id是private修饰的
    }
}
```

为什么要使用继承?

- 假设现在有如下情景:
  - 在普通的药水瓶Bottle的基础上，**细化**为三种药水瓶如下:
    - RegularBottle（普通药水瓶）
    - EfficacyProBottle（**药效**强化药水瓶）
    - VolumeProBottle（**容积**强化药水瓶）
  - **在RegularBottle的属性基础上**，另外两种药水瓶分别增加了eff_ratio**属性**和vol_ratio属性(均为double，表明强化幅度)

假设不使用继承

```
public class VolumeProBottle {
    private String id;
    private String name;
    private int capacity;
    private boolean isCarried;
    private boolean isFull;
    private long price;
    private double vol_ratio;
    //只展示一部分方法，省略剩余方法。
    public VolumeProBottle (String id, String name,
    int capacity, long price, double ratio) {
        this.id = id;
        this.name = name;
        this.capacity = capacity;
        this.isCarried = false;
        this.isFull = true;
        this.price = price;
        this.vol_ratio = ratio;
    }
    public long getPrice() {
        return price;
    }
}
```

重复属性

重复代码

```
public class EfficacyProBottle {
    private String id;
    private String name;
    private int capacity;
    private boolean isCarried;
    private boolean isFull;
    private long price;
    private double eff_ratio;
    //只展示一部分方法，省略剩余方法。
    public EfficacyProBottle (String id, String name,
    int capacity, long price, double ratio) {
        this.id = id;
        this.name = name;
        this.capacity = capacity;
        this.isCarried = false;
        this.isFull = true;
        this.price = price;
        this.eff_ratio = ratio;
    }
    public long getPrice() {
        return price;
    }
    public String getName() {
        return name;
    }
    public int getCapacity() {
        return capacity;
    }
}
```

假设不使用继承

```
public class RegularBottle {
    private String id;
    private String name;
    private int capacity;
    private boolean isCarried;
    private boolean isFull;
    private long price;
    //只展示一部分方法，此处省略剩余方法。
    public RegularBottle(String id, String name, int capacity, long price) {
        this.id = id;
        this.name = name;
        this.capacity = capacity;
        this.isCarried = false;
        this.isFull = true;
        this.price = price;
    }
     public String getName() {
        return name;
    }
    public int getCapacity() {
        return capacity;
    }
}
```

重复属性

重复代码

```
public class Adventurer{
    private String id;
    private String name;
    private HashMap<String,RegularBottle> regularbottles;
    private HashMap<String,EfficacyProBottle> effprobottles;
    private HashMap<String,VolumeProBottle> volprobottles;

    //移除指定id的Bottle,不论是什么种类
    public void removeBottle(String id) {
        if (regularbottles.containsKey(id)){
            regularbottles.remove(id);
        }
        if (effprobottles.containsKey(id)){
            effprobottles.remove(id);
        }
        if (volprobottles.containsKey(id)){
            volprobottles.remove(id);
        }
    }
}
```

冗余式展开

继承的使用

- 不难发现，这三种药水瓶**具有大量的相同属性和访问属性对应的相同方法**
- **Adventurer类逐个编写对三种药水瓶的冗余管理代码，繁琐**
- **不使用继承方案带来的麻烦：**在原代码中所有访问Bottle类的地方都需要改写为针对三种新Bottle的判断和访问
  - 如果后续再增加新的Bottle呢?

- 在实际的开发中，如果**“细化”的类较多**，会出现重复的属性定义和操作代码
- 无法统一管理各种细化类的实例化对象
- 在代码维护过程中容易出现修改不一致，难以debug

使用继承

```
public class Bottle {
    private String id;
    private String name;
    private int capacity;
    private boolean isCarried;
    private boolean isFull;
    private long price;
    public Bottle(String id, String name,
    int capacity, long price) {
        this.id = id;
        this.name = name;
        this.capacity = capacity;
        this.isCarried = false;
        this.isFull = true;
        this.price = price;
    }
    public long getPrice() {
        return price;
    }
}
```

```
public class VolumeProBottle extends Bottle {
    private double radio;
    public VolumeProBottle (String id, String name,
    int capacity, long price, double radio) {
        super(id, name, capacity, price);
        this.radio = radio;
    }
}
public class RegularBottle extends Bottle {
    public RegularBottle(String id, String name,
     int capacity, long price) {
        super(id, name, capacity, price);
    }
}

public class EfficacyProBottle extends Bottle {
    private double radio;
    public EfficacyProBottle (String id, String name,
    int capacity, long price, double radio) {
        super(id, name, capacity, price);
        this.radio = radio;
    }
}
```

```
public class Adventurer {
    private String id;
    private String name;
    private HashMap<String, Bottle>
bottles;

    //移除指定id的Bottle,不论是什
么种类
public void removeBottle(String id) {
        if (bottles.containsKey(id)){
            bottles.remove(id);
        }
    }
}
```

减少冗余，结构清晰！

继承的使用

- 继承带来的便利：
  - 减少冗余：子类不必重复实现父类已有的属性和方法，直接获得
    - 在编写新的药水瓶类**XBottle**的时候，只需要使用**XBottle** extends **Bottle**来声明
    - 只需编写**XBottle**独有的属性或方法
  - 层次结构：建立了类之间的抽象层次结构，上层类可以概括下层类
    - 可以用Bottle来引用Xbottle_obj，Adventurer类**不需要新增容器来储存Xbottle实例**
    - 任何使用Bottle对象的方法，都可以传入Xbottle类型的对象

基于继承的扩展

- 子类在父类基础上，可以围绕子类来增加特有的属性和方法
- 假设有一个消息类Message，存储消息体(String msgBody)、消息编号(int msgID)和消息日期(String msdDate)，提供构造方法、get方法和print方法
- 在此基础上构造一个邮件消息类MailMessage
  - 结构化表示sender, receiver
  - 消息体是否需要扩展?
  - 是否需要增加send和receive方法?
  - print方法是否需要扩展?
- 如果进一步扩展构造一个带有自动回复已阅通知的邮件消息类呢?

基于继承的扩展

- 子类可以覆盖父类已有方法：重写(override)
  - 方法名称、参数列表和返回类型**保持一致**
- **XBottle对父类Bottle进行了方法重写**

Bottle

```
public String getClassName() {
    return "Bottle";
}
```

Xbottle extends Bottle

```
@Override
public String getClassName() {
    return  “XBottle";
}
```

这段代码的输出？

```
Bottle b = new XBottle(…);
System.out.println(b.getClassName());
```

方法重写语法的规则

- **@Override 注解**：**可选**的注解，用于标记方法是重写的父类方法
  - 非强制，可提高代码的可读性和可维护性
- **访问权限**：子类方法的访问权限必须大于等于父类方法的访问权限
  - 如果父类方法为public，子类方法只能为public
- **返回类型**：子类方法的返回类型必须与父类方法的**返回类型一致**，或者是其**子类型**（协变返回类型）
  - 若返回类型是**基本类型，则只能相同**
- **方法名和参数列表**：子类方法的**方法名和参数列表**必须与父类方法**完全相同**
  - **参数个数和参数顺序、参数类型**

继承的优势

- **代码重用**
  - 子类无需重复实现父类的属性和方法，提高了重用度
- **扩展性**
  - 子类可以**添加新的属性和方法**，实现自己独有的功能
  - 实现了对父类的增量式扩展（保持父类不变）
- **继承层次**
  - 形成抽象层次结构，任何子类对象都可以使用父类型来统一管理和引用
  - 实现了对**各种变化**的统一处理能力

继承的使用情景

- **提炼场景**：多个类之间存在**相同的属性和方法**
  - 将**相同（公共）的部分提取出来形成一个类（父类）**
  - 避免出现重复代码
  - 更好的维护这些相同的属性和代码
- **扩展场景**：引入新类，避免对已有类进行修改
  - 新类对已有类进行增量式扩展
  - 避免编写冗余代码

继承的使用场景

- 继续Message类的故事
- 假设有一个专门的类来管理所有的消息，称为MessageBox
  - MessageBox中有一个HashMap<Integer, Message>的容器
  - MessageBox提供基本的插入和检索方法
- 在有了MailMessage类之后，是否需要应用继承来获得一个MailMessageBox类?

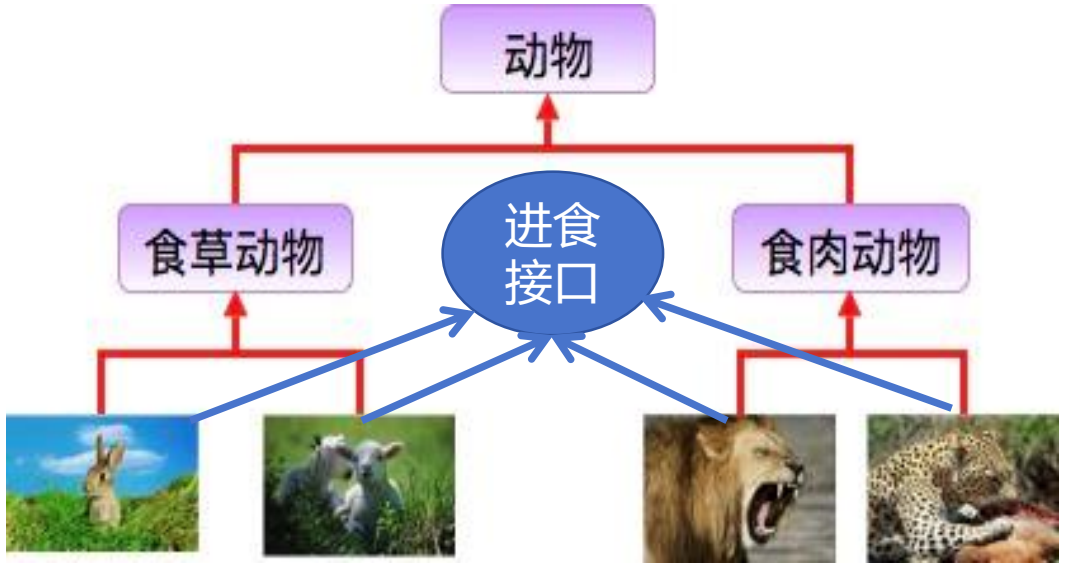
继承层次下的方法调用

- 以MailMessage类为例
  - MailMessage mess = new MailMessage(…);
  - mess.print()调用的是哪个方法?
  - Message mess = new MailMessage(…);
  - mess.print()呢?
- 现在进入到MailMessage类中
  - 任何非static方法中都能使用this和super
  - 三个print分别调用的哪个方法?
  - this是当前对象，super呢?

```
public class MailMessage extends Message{
    …
    public void prepare(){
        String s1 = print();
        String s2 = this.print();
        String s3 = super.print();
        …
    }
    @override
    public String print(){
        …
    }
}
```

接口的定义

- 接口（Interface）是一种定义**方法和常量**的抽象类型，不提供方法实现
- 接口统一并规范定义一组类的行为
- 一个接口可以由N个类实现，一个类可以实现N个接口



接口在Java中的语法

- 要实现接口，类使用 **implements** 关键字，并实现接口中定义的**所有方法**
- @Override表示对抽象方法的**覆写**

```
interface MyInterface {
    // 抽象方法声明
    void myMethod1();
    void myMethod2();
    void myMethod3();
}


class MyClass implements MyInterface {
    @Override
    public void myMethod1() {
        System.out.println("Implementation of myMethod1");
    }
    @Override
    public void myMethod2() {
        System.out.println("Implementation of myMethod2");
    }
    @Override
    public void myMethod3() {
        System.out.println("Implementation of myMethod3");
    }
}
```

接口在Java中的语法

- 抽象方法的声明**不需要public或者private等关键字的修饰**，规定方法名，参数及返回类型即可
- 接口也是一种类型，可以用a **instanceof B**判断对象a是否实现了接口B，即是否类型B的实例
- 接口**无法被实例化**（用new），只能被实现

```
public class MyClass extends fatherClass implements MyInterface1 ,MyInterface2 {…}
```

继承

实现多个接口

使用接口还是继承?

- 假设如下情景:
  - 保持之前继承中例子将药水瓶的细化为三种不同类型的药水瓶
  - 装备、三种药水瓶、食物和冒险者都有price（价格）属性，因而可以将它们都看作是**价值体 Commodity**，且他们都有getPrice()函数来对外表征自己的价值
    - 向上提炼
  - 一个冒险者要管理它拥有的诸多价值体对象
    - 冒险者可以雇佣其他冒险者（因此也是一个价值体对象）
    - 能不能只用一个容器来统一管理其拥有的所有价值体?
    - ----> **如何将不同类型的价值体“统一”起来?**

使用接口还是继承?

- 先试试继承方案
- 建立Commodity类
  - id, name, price属性
  - getPrice方法
- 让**Bottle**、 Equipment、 Food和Adventurer继承自Commodity
  - 让三种具体Bottle继承自Bottle
- 潜在问题:
  - Commodity类几乎没有业务功能，各个子类需要加入大量的扩展
  - 会建立比较深的继承层次

使用接口还是继承?

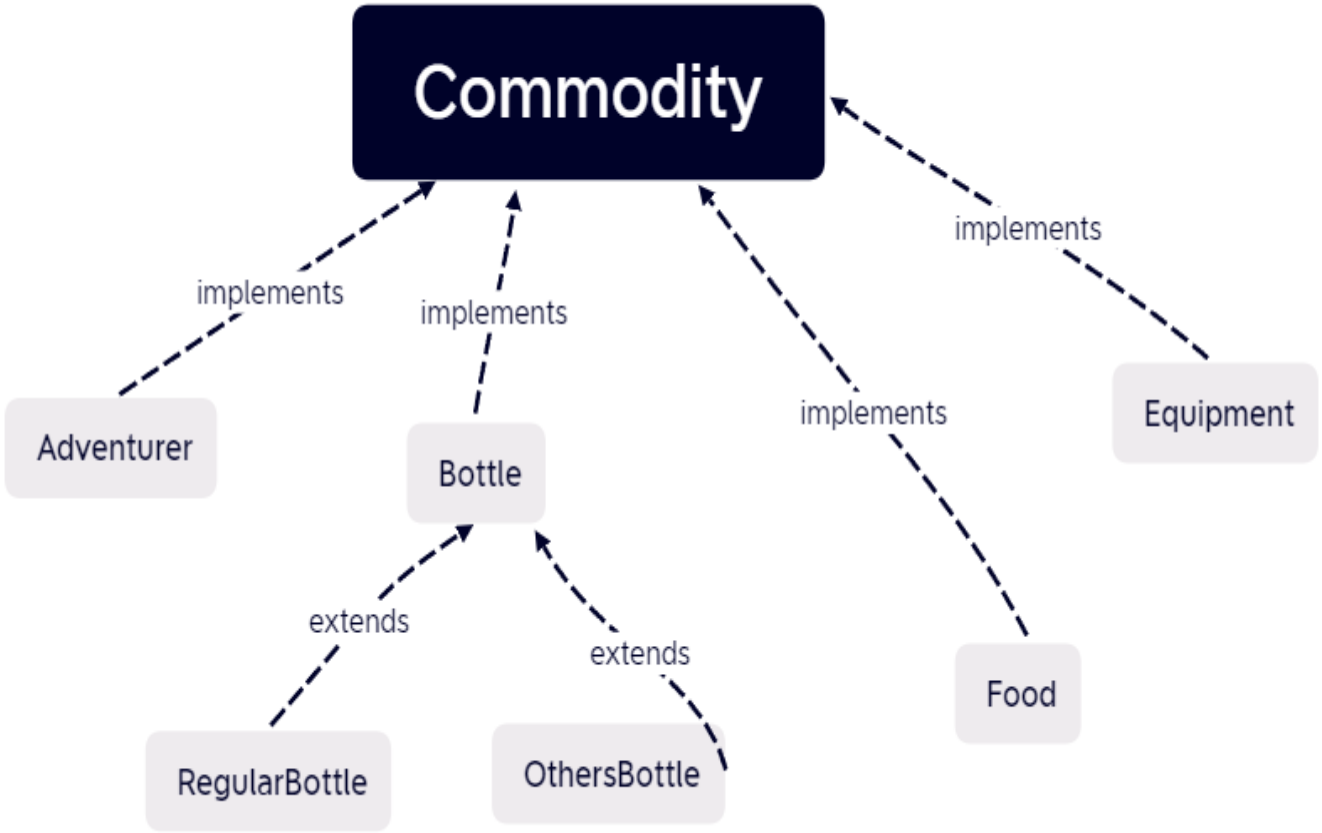
- Bottle，Equipment，Food和Adventurer这四种价值体的数据内涵和业务功能差别很大
- 核心是围绕其价值（price）建立统一的对象管理手段
- 把Commodity定义为interface
- 让Bottle、Equipment、Food和Adventurer实现Commodity接口
  - 三种具体的Bottle仍然继承自Bottle
  - 建立了实现**行为抽象层次**

接口的使用

```
public class Bottle implements Commodity {
   // Bottle的属性和方法
}

public class Equipment implements
Commodity {
   // Equipment的属性和方法
}

public class Adventurer implements
Commodity {
   //commodities统一管理所有价值体
   private ArrayList<Commodity>
commodities;
   //Adventurer的其它属性和方法
}
```



RegularBottle类是否也实现了Commodity接口?

接口和继承的区别

	数量	关系	概念本质
继承	一个类只能有**一个父类**	子类继承了父类的所有属性和方法	建立了数据抽象层次
接口	一个类可以实现**多个接口**	实现接口中所定义的全部方法	建立了行为抽象层次

解决实际问题

- 设计一个父类CommandUtil
  - 提供command(String message)
- 按照指令处理要求设计若干具体的子类
  - AddBottle, AddFood, …
- Manager类管理m个commandUtil对象
  - ArrayList<CommandUtil>
- 根据输入指令自动获得cmd对象
  - cmdUtil = cmdUtilArray.get(operator -1);
  - cmdUtil.command(message);

这些CommandUtil子类要实现相应的command，必须访问相关的容器对象，如何处理?

解决实际问题

- 如果使用接口机制来解决呢?
  - Step 1：设计Command接口（抽象处理能力）
  - Step 2：设计完成具体指令业务的类（具体处理能力）
    - BottleOperator，FoodOperator，…
    - 提供加入、删除和查询等operation
  - Step 3：设计具体指令映射类
    - BottleOperationCommand, FoodOperationCommand, …
    - 实现Command接口 → 交由BottleOperator对象等来完成具体业务操作
    - 分别管理bottle，food等相关的对象
  - Step 4：设计command调度器cmdInvoker
    - 设置ArrayList<Command> cmdList；
    - 从scanner获得具体指令，按照类别创建BottleOperationCommand对象，加入cmdList
    - 按照某种策略执行cmdList中的每个cmd.execute(message);

```
public interface Command {
    void execute(String message);
}
```

继承下的Junit测试

- 单个类的测试
  - 和之前一样
- 当测试子类时，需要注意
  - 子类方法是否调用了父类方法
  - 子类方法是否访问了父类属性
- 当测试其他类方法时，需要注意
  - 传入子类型对象
  - 传入父类型对象

作业介绍与建议

- 仍为冒险者游戏，在第四次作业的基础上迭代开发
- 允许冒险者雇佣另一个冒险者，且赋予装备、药水瓶、食物、冒险者价值的概念
- **细化**装备和药水瓶的类型，具有不同的功效
- 建议把装备、药水瓶、食物和冒险者都看作是**价值体（commodity）**
- 建议**引入Commodity接口来建立行为层次关系：**
  - 冒险者 Adventurer 类、装备 Equipment 类、药水瓶 Bottle 类、食物 Food 类实现该接口
- 装备和药水瓶的细分，建议通过**继承**实现