

《面向对象程序先导》

Lec3-如何管理对象

北京航空航天大学计算机学院

吴际

2023.9.22

国庆期间补课事宜

- 10.7日（周六）补课？

内容提要

- 对象与引用
- 常见容器及其作用
- 层次化对象管理方法
- 作业内容介绍

对象与引用

- 对象是程序通过new产生的实例
- 引用是程序中声明的变量
- 究竟**对象**是什么
 - 堆内存的一个区域，按照类的属性填充了相应的数据
 - 包含属性**数据**和指向类的指针
 - **方法**定义在类中而不是对象中
 - 使用不同的变量去引用一个对象时，不改变对象本身的数据

```
public class Adventurer {  
    private String id;  
    private String name;  
  
    public Adventurer(String id, String name) { //构造方法  
        this.id = id;  
        this.name = name;  
    }  
  
    public void sayHello(){  
        System.out.println("Hello ,I am "+name);  
    }  
}
```

数据

方法

创建对象

```
public class Main {  
    public static void main(String[] args) {  
        Adventure person1 = new Adventurer("1","Bob");  
        Adventure person2 = new Adventurer("2","Alice");  
  
        person1.sayHello();//Hello ,I am Bob  
        person2.sayHello();//Hello ,I am Alice  
  
        person1 = person2;//改变引用对象  
        person1.sayHello();//Hello ,I am Alice  
    }  
}
```

对象与引用

- 究竟**引用**是什么
 - 指向对象**内存地址**的指针
 - 对象引用存在null的可能性, 但对象不可能
 - 通过对象引用可以访问对象的属性和方法

如果对象引用与对象本身的类型不同会发生什么?

引用

```
public class Adventurer {  
    private String id;  
    private String name;  
  
    public Adventurer(String id, String name) { //构造方法  
        this.id = id;  
        this.name = name;  
    }  
  
    public void sayHello(){  
        System.out.println("Hello ,I am "+name);  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Adventure person1 = new Adventurer("1","Bob");  
        Adventure person2 = new Adventurer("2","Alice");  
  
        person1.sayHello();//Hello ,I am Bob  
        person2.sayHello();//Hello ,I am Alice  
  
        person1 = person2;//改变引用对象  
        person1.sayHello();//Hello ,I am Alice  
    }  
}
```

对象与引用

- Java与C的联系与区别

Adventurer adventure = new Adventurer(); **Java**
Adventurer* adventure = (Adventurer*)malloc(sizeof(Adventurer)); **C语言**

- Java的对象引用“相当于”C中的指针变量，指向一片内存空间
 - Java通过new关键字初始化对象空间
 - C语言通过malloc等方法分配指针指向的内存
- 当堆中的某个对象**无法被程序访问**时，java垃圾回收机制会自动回收内存，无需手动处理

对象与引用

- 区别：
 - 对象有**实际存在**的内存区域；引用是一个**标识符**，指向对象所在地址
 - 改变对象内容的唯一方法是通过一个对象引用去访问其方法或数据
 - 对象引用内容的改变可以随时通过变量赋值来实现
- 联系：
 - 对象引用是与被引用对象**紧密关联**，程序访问内存对象的唯一手段是通过指向该对象的引用
 - 一个对象可以被0到多个对象引用所指向
 - 一个对象引用只能指向0到1个内存对象

对象管理问题的引入

- 针对一个类，如果你的程序只需要创建一个对象
 - `MyClass mc = new MyClass(...);`
- 如果需要创建三个对象呢？
 - `MyClass mc1 = new MyClass(...);`
 - `MyClass mc2 = new MyClass(...);`
 - `MyClass mc3 = new MyClass(...);`
- 如果需要创建更多呢？
- 如果根本就不知道会需要创建多少呢？

对象管理

- 对象一旦创建，就必须进行管理
 - 即必须在使用时能够找到它
 - 否则会如何？
- 使用一个个对象引用来管理
 - 简单和方便
 - 代码臃肿繁琐
- 使用数组来管理
 - 简单高效
 - 不能适应动态变化
- 使用容器来管理
 - 简单灵活

Java常见容器及其作用

- **ArrayList** (可伸缩数组)
- **LinkedList** (可伸缩链表)
- **HashMap** (哈希表)
- **HashSet** (哈希集)
- **Queue** (队列)
- **Deque** (双端队列, 栈)
- **TreeMap, PriorityQueue**

Q: 如何选择所需要的容器?

如何**快速寻找**对象?

对象之间的**关系如何表达**?

ArrayList & LinkedList的特点

- 逻辑特征：ArrayList相当于数据结构中的**数组**，LinkedList相当于数据结构中的**链表**。
- 优点**：当我们需要**保持所管理对象的顺序关系并且按照下标访问**时，采用ArrayList或者LinkedList是比较好且比较**简单**的选择
- 缺点**：查找某个对象需要**遍历**，**查找速度较慢**

```
public class Adventurer {  
    //省略其它属性  
    private ArrayList<Bottle> bottles;  
    //构造方法  
    public Adventurer(String id, String name) {  
        this.bottles = new ArrayList<>();  
    }  
    //查找某个id==bottleId的Bottle对象  
    public Bottle getBottle(String bottleId){  
        Bottle result = null;  
        for (Bottle bottle : bottles){    //需要遍历  
            if (bottle.getId().equals(bottleId)){  
                result = bottle;  
                break;  
            }  
        }  
        return result; // 没找到则为null  
    }  
    //加入一个Bottle对象  
    public void addBottle(String bottleId, String bottleName, int capacity) {  
        bottles.add(new Bottle(bottleId, bottleName, capacity));  
    }  
}
```

查

增

```
//删除一个Bottle对象  
public String removeBottle(String bottleId) {  
    Bottle bottleToRemove=null;//待删除的Bottle对象  
    String nameToReturn=null;//需要返回的删除的Bottle的name  
    bottleToRemove = getBottle(bottleId);  
    if (bottleToRemove==null){  
        //这种情况说明没有找到需要删除的Bottle对象  
        //一般不会发生，这样写是为了保证代码鲁棒性  
        //需要根据个人代码情况处理  
    }  
    bottles.remove(bottleToRemove);//删除  
    nameToReturn = bottleToRemove.getName();  
    return nameToReturn;  
}
```

删

removeBottle运行结束时，bottles中是否还有bottleToRemove指针值？
后者指向的内存对象是否会被删除？

HashMap的特点

- 逻辑特征：基于哈希表的**键值对映射**。它允许使用键来**快速查找和访问值**。HashMap中元素没有固定顺序，不允许重复的键
- **优点**：快速**查找(或删除)**某个元素。当我们管理的对象有某个**唯一可标识**的属性（如id等）时，用该属性作为key查找对象比较方便
- **缺点**：内部存储**无序**，难以按照先后或大小等顺序访问

```
public class Adventurer {  
    //省略其它属性  
    private HashMap<Bottle> bottles;  
    //构造方法  
    public Adventurer(String id, String name) {  
        this.bottles = new HashMap<>();  
    }  
    //查找某个id==bottleId的Bottle对象  
    public Bottle getBottle(String bottleId){  
        return bottles.get(bottleId); // 没找到则为null  
    }  
    //加入一个Bottle对象的键值对  
    public void addBottle(String bottleId, String bottleName, int capacity) {  
        bottles.put(bottleId, new Bottle(bottleId, bottleName, capacity));  
    }  
    //删除一个Bottle对象  
    public String removeBottle(String bottleId) {  
        String nameToReturn = getBottle(bottleId).getName();  
        bottles.remove(bottleId); //按照key删除键值对  
        return nameToReturn;  
    }  
}
```

查（快速）

增加键值对

根据key删除

HashSet的特点

- 逻辑特征：基于哈希表实现的**无序集合**，**不允许存储重复的元素**。提供了常数时间的插入和查找操作
 - HashSet本质是只存key的HashMap
- **优点**：保持集合元素的**互异性**，无重复。当需要**实现自动去重**功能去管理对象的时候，使用HashSet是比较好的选择
- **缺点**：内部存储**无序**，如果希望按照先后或大小等顺序访问则造成不便

```
public class Adventurer {  
    //省略其它属性  
    HashSet<Bottle> bottles;  
    //构造方法  
    public Adventurer(String id, String name) {  
        this.bottles = new HashSet<>();  
    }  
    //添加元素  
    public void addBottle (Bottle bottle) {  
        bottles.add(bottle);  
    }  
    //判断是否包含元素  
    public boolean hasBottle(Bottle bottle){  
        return bottles.contains(bottle);  
    }  
    //删除元素  
    public void removeBottle(Bottle bottle){  
        bottles.remove(bottle);  
    }  
  
    //其他操作  
    public void otherOperations(){  
        // 得到HashSet大小，即元素个数  
        int size = bottles.size();  
        // 遍历HashSet中的所有元素  
        for (Bottle obj : bottles) {  
            System.out.println("bottle's name is " + obj.getName());  
        }  
        //.....  
    }  
}
```

增

查

删

遍历操作

其它常用容器

- Queue(**队列**), 是一个**抽象接口**, LinkedList是其具体实现之一
- Deque(**双端队列**), 是一个**抽象接口**, 三种应用场景
 - 普通队列, 一端进另一端出, `Deque q = new LinkedList();`
 - 双端队列, 两端皆可进出, `Deque q = new LinkedList();`
 - 堆栈, 只从一端进出, `Deque q = new LinkedList();`
 - Java老版本的Stack类已经过时, 推荐使用Deque
 - `push->addFirst`, `pop->removeFirst`, `peek->peekFirst`
- TreeMap(**有序映射**), 通过**红黑树**实现, 根据键来排序, 可**自定义比较器**
 - `TreeMap<id, object>`
- PriorityQueue(**优先队列**), 可按优先级进行排序, 基于**堆 (Heap)** 实现

如何合理选择容器

- **需求导向**：使得查找、管理方便，包括遍历，迭代，哈希三种方式
- **仔细考量**：应该仔细考虑容器的优势和缺点
- **灵活管理**：能动态灵活地管理对象，把业务逻辑关注的对象关系融入到容器内在的管理机制中
 - 有序性
 - 互异性
 - 映射性
 - 进出次序

如何合理选择容器

- 场景1
 - 一门课程的一组学生，学号各不相同，每个学生的属性包括学号、姓名、课程成绩
- 场景2
 - 每个学生加入小班号属性，支持按照小班号查询
- 场景3
 - 支持安排小班号+学号的**综合排序**
- 场景4
 - 支持动态选择按照（小班号+学号）或（小班号+成绩）进行**综合排序**

所有对象都扔进一个容器中吗？

- 去银行办理业务首先要取号（request），然后排队等叫号
 - 提供办理业务的类别（现金业务、贷款业务、信用卡业务等）
 - 提供账号基本信息
- 针对银行的服务系统而言，所有的request需要处理
 - 分配到特定类型的窗口
 - 记录request的处理过程
 - 窗口进行具体业务的处理

需要按照request类别建立不同的容器来分别进行管理！

所有对象都扔进一个容器中吗？

- 即便是相同类型的对象，其处理逻辑可能也不同
 - 贷款业务，不同范围的贷款金额需要不同的处理流程
- 针对对象的属性取值（状态）分组管理
 - 创建多个容器分别进行管理
 - 如贷款request，按照贷款额度创建不同的容器
 - 大额贷款容器
 - 小额贷款容器
- 有时甚至需要根据对象状态的变化动态调整到不同的容器
 - 订单处理中心如何设计？

层次化对象管理方法

- 使用容器管理一组对象
- 一个对象可以内置容器来管理其下一层次对象
 - 形成层次化的对象管理结构
- 对象层次关系主要由类之间的关联关系来定义
 - **组合关系**
 - **聚合关系**

组合关系

- **组合 (Composition)**：由多个对象组合形成一个更大对象，**整体与部分**关系
 - 整体离不开其部分而存在
 - 汽车由引擎，车轮等组件启动
- 组合关系可以刻画一个对象内部的组成结构
- 组合关系也为类属性设计提供了依据
- 以对象引用或容器作为属性变量，形成对象之间的整体和部分结构

```
public class Main{
    public static void main(String[] args){
        Car car = new Car();
        car.start();
        car.run();
    }
}
```

```
class Engine {
    public void start() {
        System.out.println("引擎启动! ");
    }
}

class Wheel {
    public void start() {
        System.out.println("车轮转动! ");
    }
}

//肯定还有其他类，如方向盘，车轴.....都不作为重点讨论

class Car {
    // 组合关系，Car类包含Engine类和Wheel类的实例
    private Engine engine;
    //汽车至少有四个轮子，这里不作为重点，只显示一个，同学们理解即可
    private Wheel wheels;
    public Car(Wheel wheel) {
        this.engine = new Engine();
        this.wheels = new Wheel();
    }
    public void start() {
        engine.start();
        System.out.println("汽车启动! ");
    }

    public void run() {
        wheel.start();
        System.out.println("汽车跑动! ");
    }
}
```

聚合关系

- **聚合** (Aggregation) : **弱关联关系**, 以对象引用作为其成员变量, 相应的对象引用可以为null
 - 一个对象使用容器来管理下层对象
 - 下层对象本身独立存在
 - 上周作业中几个主要的类: Adventure类, Bottle类, Equipment类体现的关系就是聚合关系
 - 一个Adventure可以只拥有Bottle, 或只有Equipment, 也可以二者都有或者都没有
 - 与<汽车, 发动机, 引擎>的组合关系具有本质上的差异
- 冒险者可以脱离Bottle、Equipment而存在!

```
//本示例中所有容器都使用ArrayList, 只是为了凸显出类之间的关系
//请同学们自行思考如何使用合适的容器

public class Example{
    private ArrayList<Adventurer> adventurers;

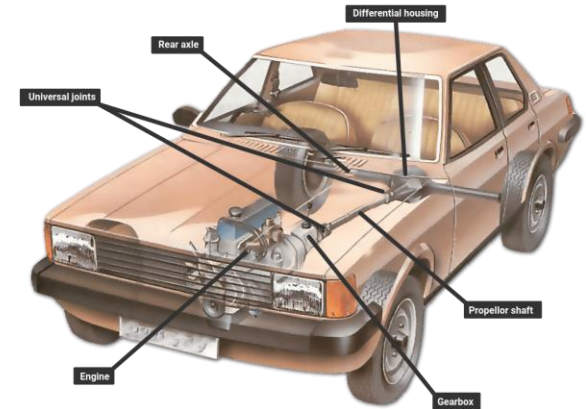
    public Example(){
        adventurers = new ArrayList<>();
    }

    public addAdventurer(Adventurer adventurer){
        adventurers.add(adventurer);
    }
    //.....
}

public class Adventurer(){
    private ArrayList<Bottle> bottles;
    private ArrayList<Equipment> equipments;
    public Adventure(){
        bottles = new ArrayList<>();
        equipments = new ArrayList<>();
    }
    public addXXX(XXX){
        .....
    }
    public removeXXX(XXX){
        .....
    }
}
```

一个对象会被哪些上层来管理？

- 本质上是属性识别问题
 - 类的职责确认
 - 建议：单一职责原则
- 要注意区分组合关系和聚合关系
 - 组合关系：顶层对象之间不会有共享对象
 - → 一个对象只会出现在一个容器中
 - 聚合关系：顶层对象之间会有共享对象
 - → 一个对象可能会出现多个容器中
- 灾难性示例：两部车共享管理同一个发动机对象！



对象管理职责

- 一个类可以使用容器来管理具体对象
 - Java容器不知道宿主类的业务目标
- 从业务角度，类需要提供业务层次的管理行为
 - 宿主类实际上是一个业务层次容器
- 以电子商城的个人账号中订单管理为例
 - 加入一个对象时需要做什么业务检查？
 - 提供哪些业务驱动的对象检索手段？
 - 移除一个对象需要做什么业务检查？
- 这些问题的解决需要上层和下层对象的协同→层次化设计

层次化设计方法

- 以Bottle类和Adventurer类为例，假设有一个给药水瓶扩容的业务
 - 给每个Adventurer中**不为空**的Bottle扩容100
- 某个业务类管理 Adventurer 对象，Adventurer类管理Bottle对象
 - 在业务类中添加addCapacity方法
 - 遍历每一个Adventurer对象，再遍历其管理的bottle对象
- 有什么问题？

```
public class Bottle{
    private int capacity;
    private boolean isEmpty;
    public void addCapacity(int val){
        this.capacity+=val;
    }
    public boolean isEmpty(){
        return this.isEmpty;
    }
}

public class Adventurer(){
    //省略其他属性
    private ArrayList<Bottle> bottles;
    public Adventurer(){
        bottles = new ArrayList<>();
    }
    public ArrayList<Bottle> getBottles(){
        return bottles;
    }
}
```

```
public class Example{
    private ArrayList<Adventurer> adventurers;
    public Example(){
        adventurers = new ArrayList<>();
    }

    public void addCapacity(){
        for (Adventurer adventurer : adventurers){
            ArrayList<Bottle> bottles = adventurer.getBottles();
            for (Bottle bottle : bottles){
                if (!bottle.isEmpty()){
                    bottle.addCapacity(100);
                }
            }
        }
    }
}
```


层次化设计方法

- 问题1：业务类跨层依赖底层的Bottle类
 - Bottle类变化导致业务类也要变化
- 问题2：业务类addCapacity方法**臃肿**
 - 遍历adventurer对象和bottle对象
- 从管理职责角度
 - 业务类管理Adventurer对象, **不应该涉及Bottle对象的处理**
 - Adventurer管理bottle对象, 不应该将私有化保护的bottles暴露出去
- 合理的方案
 - **进一步封装**, 让Adventurer类提供扩容这一业务操作

```
public class Bottle{
    private int capacity;
    private boolean isEmpty;
    public void addCapacity(int val){
        this.capacity+=val;
    }
    public boolean isEmpty(){
        return this.isEmpty;
    }
}

public class Adventurer(){
    //省略其他属性
    private ArrayList<Bottle> bottles;
    public Adventurer(){
        bottles = new ArrayList<>();
    }
    public ArrayList<Bottle> getBottles(){
        return bottles;
    }
}

public class Example{
    private ArrayList<Adventurer> adventurers;
    public Example(){
        adventurers = new ArrayList<>();
    }
    public void addCapacity(){
        for (Adventurer adventurer : adventurers){
            ArrayList<Bottle> bottles = adventurer.getBottles();
            for (Bottle bottle : bottles){
                if (!bottle.isEmpty()){
                    bottle.addCapacity(100);
                }
            }
        }
    }
}
```

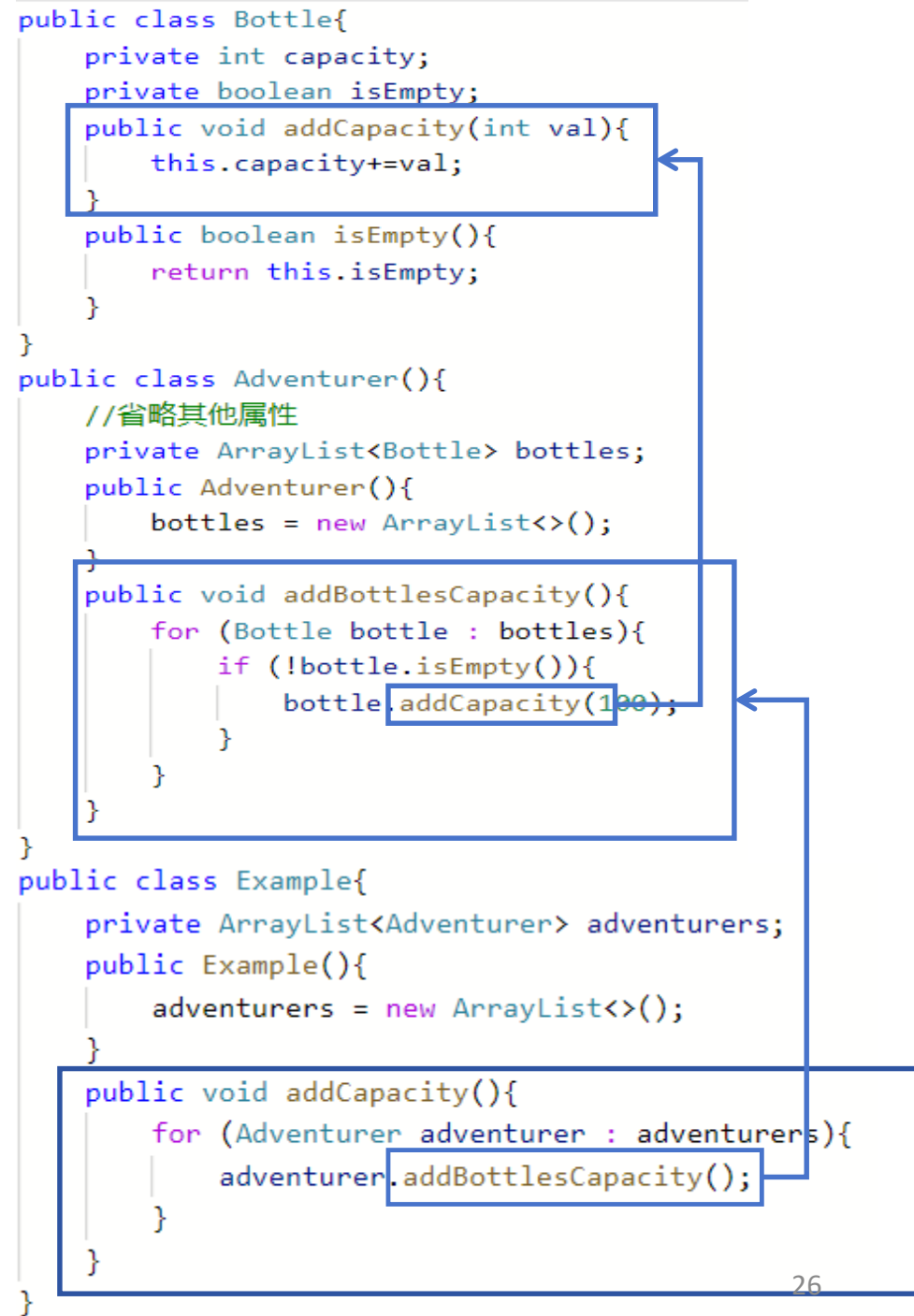
层次化方法设计

- 如右图所示的改进方案，建立了清晰的层次化对象管理效果
- 核心在于理清类之间的关系，遵循了单一职责原则
 - Example → Adventurer → Bottle
 - 层次之间分工明确，**层层代理**，不跨层，每一层的业务都得到了简化
 - 每一层都提供了更好的封装，具有更高的复用度
- 是否仍然有改进空间？
 - 100这个常数是否可以参数化？
 - 如果瓶子里没有药水就不能扩容（逻辑上不合理），怎么办？
 - 是否有更多需要对bottle进行扩展的可能性？

```
public class Bottle{
    private int capacity;
    private boolean isEmpty;
    public void addCapacity(int val){
        this.capacity+=val;
    }
    public boolean isEmpty(){
        return this.isEmpty;
    }
}

public class Adventurer(){
    //省略其他属性
    private ArrayList<Bottle> bottles;
    public Adventurer(){
        bottles = new ArrayList<>();
    }
    public void addBottlesCapacity(){
        for (Bottle bottle : bottles){
            if (!bottle.isEmpty()){
                bottle.addCapacity(100);
            }
        }
    }
}

public class Example{
    private ArrayList<Adventurer> adventurers;
    public Example(){
        adventurers = new ArrayList<>();
    }
    public void addCapacity(){
        for (Adventurer adventurer : adventurers){
            adventurer.addBottlesCapacity();
        }
    }
}
```



作业介绍

- 冒险者游戏的迭代开发
- 冒险者不必每次出行把家底都带上
 - 给他新增一个**背包**
 - 只带必要的装备和药水
- **量化**冒险者的状态，**引入了三个属性**：
 - 体力(Hitpoint), 等级(level), 战斗力(power)
- 冒险者吃东西可提高等级
 - **食物类** (food)
 - 爱吃才会赢

作业分析与建议

- 任务：
 - 在lec2作业的基础上完成冒险者新增加属性的管理
 - 实现冒险者的背包功能，并依据要求实现背包内物品数目的控制
 - 继续开展junit测试：要求method覆盖率 $\geq 90\%$ ，line覆盖率 $\geq 60\%$
- 提示：
 - 请分析容器的优缺点，选择合适的容器进行对象管理
 - 分析清楚类之间的关系，建立层次化的对象管理
 - 本次作业的层次化管理对于下一次迭代至关重要
 - 在IDE上运行junit并收集覆盖率信息