



Java Lab7: 成为米开朗琪罗！

- 前情提要
- 设计模式
- 杂项



前情提要

温故而知新，可以为师矣。

Java之路行至此处，我们已经学习了：

- Java基本语法：

类型、操作符、控制结构、库与模块化编程

- 什么是(Java)面向对象编程(OOP)：

类的定义、对象的创建、**访问控制**、封装、类的复用：**组合与继承**、**Java多态**、抽象类和接口

- 一个设计模式：

工厂模式

前情提要

温故而知新，可以为师矣。

首先，Java不是C

```
public class Main{
    public static void main(String []args){
        //...
    }
    public static void function1(type args){
        //...
    }
    public static void function2(type args){
        //...
    }
    public static void function3(type args){
        //...
    }
    public static void function4(type args){
        //...
    }
} // ...
}
```

不合理，且不合理

前情提要

温故而知新，可以为师矣。

面向对象编程：对象呢？懂了！表白墙上找

```
public class Main{
    public static void main(String []args){
        Main mainObject = new Main();
        Main.function1();
        Main.function2();
        Main.function3();
    }
    public void function1(){
        //...
    }
    public void function2(){
        //...
    }
    public void function3(){
        //...
    }
}
```

合理，但不合理

前情提要

温故而知新，可以为师矣。

■ 为什么是面向对象？

代码复用、访问控制、最重要的：**抽象**

■ 什么是抽象？

“抽象是从众多的事物中抽取出共同的、本质性的特征，而舍弃其非本质的特征的过程”——wikipedia

■ 抽象文化

“在计算机科学中，抽象化是将程序以它的语义来呈现出它的外观，但是**隐藏起它的实现细节**。抽象化使得程序员可以专注在处理少数重要的部分。”

■ 抽象程度的提高有利于我们进行开发

■ 代码的抽象历程：

机器码(打孔纸带 0/1)→文字表示的汇编代码(assembly)→高级语言

■ 编程方式的抽象历程：

“面向内存编程”→面向过程编程→面向对象编程&面向函数编程



```
#include <stdio.h>

int cat_weight = 2147483647;
double cat_length = 30.0;
int cat_lives = 9;

void eat(){
    ++cat_weight;
}

void die(){
    --cat_lives;
}

void grow(double len){
    cat_length += len;
}
```

```
public class Cat{
    int weight = 2147483647;
    double length = 30.0;
    int lives = 9;
    public void eat(){
        this.weight++;
    }
    public void die(){
        this.lives--;
    }
    public void grow(double len){
        this.length+=len;
    }
    public static void main(String[] args){
        Cat cat = new Cat();
        cat.die();
    }
}
```

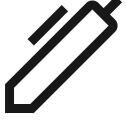
前情提要

温故而知新，可以为师矣。

- 如果使用面向对象编程的思想，当我们遇到一个问题，我们就只需要进行如下思考：
 1. 如何将问题**抽象成** N 个类（对象）？
 2. 如何设计对象？
 3. 如何进行对象与对象之间的交互？
- 回想一下我们之前**使用**过的一些类：
 - String
 - Color
 - Picture
- 回想一下我们之前**设计**过的几个类：
 - BigInt
 - Complex

- 使用的时候：
 - 舒适
 - 不用管实现细节
 - 按图索骥
 - **抽象程度高**，仿佛真的在操作一个真实的对象
- 设计的时候：
 - 模块化的
 - 实现某个功能时只需关注相关属性
 - 访问控制

难的其实是如何抽象，如何设计类。



设计模式

“艺术家用脑，而不是用手去画。”——米开朗基罗

设计模式

“艺术家用脑，而不是用手去画。”——米开朗基罗

■ 背景

物体的电阻 (resistance) 用来衡量当电流通过物体时会收到"阻碍". 电阻越大, 电流同过该物体时受到的阻碍就越大. 欧姆定律刻画了电阻, 电流和电压之间的关系. 其中, V 表示物体两端的电压(电势差), R 为该物体的电阻, I 为通过物体的电流. 电阻可以通过串联和并联的方式组合成为复杂的电阻网络:

- 串联 (Series): 将两个电阻首尾相连. 其等效电阻等于两个电阻的和.
- 并联 (Parallel): 将两个电阻并行排列. 其等效电阻等于倒数和的倒数.

一个通过串联和并联搭建的电阻网络可以表示成为字符串. 定义

- R 表示单个电阻器
- a 为正数, 表示单个电阻器的取值
- N 为电阻网络
- $(-, N, N)$ 表示两个电阻网络串联后的得到的网络
- $(/, N, N)$ 表示两个电阻网络并联后的得到的网络

电阻网络可以按照以下规则表示

1. $N = (-, N, N)|(/, N, N)|R$
2. $R = a$

其中"|"表示或者. 规则1 表示一个电路网络要么由两个网络串联而成, 要么由两个网络并联而成, 要么是单个电阻器. 规则2 表示一个电阻器的电阻取值是多少.

设计模式

“艺术家用脑，而不是用手去画。”——米开朗基罗

定义 Circuit 类用来表示电路网络。

```
public abstract class Circuit {  
    public abstract double resistance();  
}
```

什么东西啊一点思路没有单纯折磨人 我们先来尝试分析一下问题：

首先题目要求的是网络的等效电阻值，我们可以：

1. 一步到位，反正只要电阻值直接算出来，不去实现Circuit相关细节了为什么这个类是abstract算了不管了直接继承他然后一个类写完搞定真麻烦
2. 尝试用OOP的思想解决问题：

1. 如何将问题抽象成 N 个类（对象）？
2. 如何设计类？
3. 如何进行对象与对象之间的交互？

设计模式

“艺术家用脑，而不是用手去画。”——米开朗基罗

如果直接一步到位，我们需要实现：

```
public abstract class Circuit{
    public abstract double resistance();
}

class MyCircuit extends Circuit{
    private double r;
    public MyCircuit(String input){
        //将input通过算法计算出等效电阻值R
        this.r = R;
    }
    public double resistance(){
        return this.r;
    }
}
```

- 抽象了个寂寞 脱裤子放屁
- 如果这样实现，第二问基本上没法做，很麻烦。这就是抽象程度低的缺点，只针对某一特定的问题，泛化性不强。

设计模式

“艺术家用脑，而不是用手去画。”——米开朗基罗

换个思路，如果顺着OOP的思想，那么其实等效电阻可以只是（类中的众多属性或功能之中的）一个而已

- 也就是说我们其实主要需要**把整个电阻网络给构造出来**，等效电阻不过是顺手的一个功能，并且后面需要添加电流电势差等功能都跟等效电阻一样，只是顺手的一个属性或功能。
- 继承与组合，还有拓展性
- 这就是抽象，感觉上真的有一个电阻网络一样。

分析一下如何将问题抽象成 N 个类：

- 首先明确告知了将`Circuit`定义为抽象类。

为什么要定义成抽象类？乐器，肯德基。

设计模式

“艺术家用脑，而不是用手去画。”——米开朗基罗

- 电阻网络可以分成**三类**:

1. 串联电阻网络($-$, N , N), **是一个(is-a)** 电阻网络
2. 并联电阻网络($/$, N , N), **是一个(is-a)** 电阻网络
3. 单个电阻器 R , **是一个(is-a)** 电阻网络

所以逻辑上讲，我们可以设计四个类: `Circuit, Serial, Parallel, Single`，其中`Serial, Parallel, Single`是`Circuit`的子类。

- 已经成功一半了
- 等一下，串联电阻网络里面不也**有一（两）个(has-a)** 电阻网络吗，那怎么办？

就这么办。组合**与**继承，不是组合**或**继承

设计模式

“艺术家用脑，而不是用手去画。”——米开朗基罗

如何设计类？

- 首先思考都需要哪些属性，以及最重要的，这些属性**究竟该属于谁**？

- 例如对于等效电阻属性`resistance`，应该`Serial, Parallel, Single`每个类一个，还是应该给`Circuit`？
- 类似的，电压`voltage`，电流`electricity`该属于谁？
- Has-a

- 然后思考需要哪些方法来实现功能。

- 我们需要一个方法能够解析字符串成 $(-|/, N, N)$ 的方法。
- 得到这些元属性后，我们需要能够将元属性构造成对象的方法。
- (第二问) 我们需要一个能设置电压自动设置电流的方法。
- 多态的好处。

设计模式

“艺术家用脑，而不是用手去画。”——米开朗基罗

- 解析字符串&构造对象：括号配对，**二叉树**，递归，**静态方法构造对象**
- 子类们的构造函数
- （暂时）不关心（不必要的）实现细节，直接抽象地使用。
- 每个类，每个方法都只需要做自己需要做的事情。

设计模式

“艺术家用脑，而不是用手去画。”——米开朗基罗

来试试吧！

- 基础功能
- 添加功能
- 优化 (我特意保留了一部分，这样你才知道有优化的余地)

实际上，我们用到了——

设计模式

“艺术家用脑，而不是用手去画。”——米开朗基罗

工厂模式

工厂模式 (Factory Pattern) 是 Java 中最常用的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。

意图： 定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。

主要解决： 主要解决接口选择的问题。

何时使用： 我们明确地计划不同条件下创建不同实例时。

如何解决： 让其子类实现工厂接口，返回的也是一个抽象的产品。

关键代码： 创建过程在其子类执行。

应用实例： 如果你需要一辆汽车，可以直接从工厂里面提货，而不用去管这辆汽车是怎么做出来的

——工厂模式 / 菜鸟教程 (runoob.com)

设计模式

“艺术家用脑，而不是用手去画。”——米开朗基罗

所以，什么是设计模式？

“在软件工程中，软件设计模式是对软件设计中常见问题的一种通用的、可重用的解决方案。它不是一个可以直接转换成源代码或机器码的成品设计。相反，它是一个如何解决问题的描述或模板，可以在许多不同的情况下使用。设计模式是程序员在设计应用程序或系统时用来解决常见问题的形式化的最佳实践。

面向对象的设计模式通常显示类或对象之间的关系和交互，而不指定所涉及的最终应用程序类或对象。暗示可变状态的模式可能不适合函数式编程语言。如果语言本身就支持解决所要解决的问题，那么有些模式在语言中就没有必要了，而面向对象模式不一定适用于非面向对象语言。

设计模式可以看作是介于程序设计范式和具体算法之间的计算机程序设计的一种结构化方法。”

——*Software design pattern - Wikipedia*

经验之谈。

- 比较好的学习项目：

[iluwatar/java-design-patterns: Design patterns implemented in Java \(github.com\)](https://github.com/iluwatar/java-design-patterns)

</> 杂项

- 一些小问题
 - 关于多继承
 - 关于`getter() setter()`
- 注意事项

</> 杂项

关于多继承

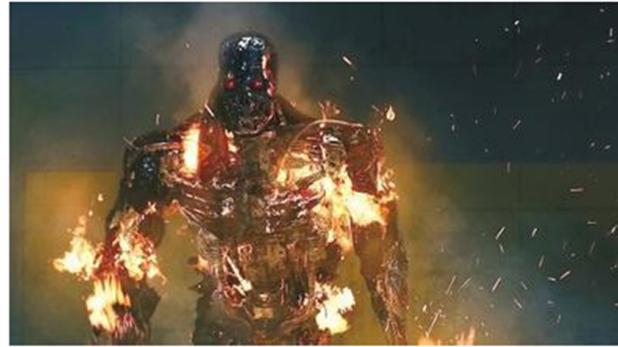
- c++面向对象和java面向对象的区别_百度知道
(baidu.com)
- Java 为什么不支持多继承? - 知乎 (zhihu.com)

重点是实现 (层面的) 多继承和声明 (层面的) 多继承

```
class Dog{  
    void say(){// wolf  
}  
}  
  
class Cat{  
    void say(){ // meow  
}  
}  
  
class Cog extends Cat, Dog{  
    public static void main(String[] args){  
        Cog cog = new cog();  
        cog.say(); // Which one?  
    }  
}
```

C++多继承: 用谁的?
声明多继承: 无所谓

C++设计者眼中的程序员:



Java设计者眼中的程序员:



</> 杂项

关于`getter() setter()`

的确可以暴露，如果1. 所有内外代码都是你自己写；2. 这个模块再也不改了；3. 不会继承它，或者继承但不改变语义。

David John Wheeler有一句名言：

“All problems in computer science can be solved by another level of indirection.”

getter、setter就是个很好的中间层。

直接摘录stackoverflow上一个不错的总结：

[oop - Why use getters and setters?](#)

1. 这两个方法可以方便增加额外功能（比如验证）。
2. 内部存储和外部表现不同。
3. 可以保持外部接口不变的情况下，修改内部存储方式和逻辑。
4. 任意管理变量的生命周期和内存存储方式。
5. 提供一个debug接口。
6. 能够和模拟对象、序列化乃至WPF库等融合。
7. 允许继承者改变语义。

8. 可以将getter、setter用于lambda表达式。（大概即作为一个函数，参与函数传递和运算）
9. getter和setter可以有不同的访问级别。

的回答，但是这里还想从另外一个更宏观的角度，以我的理解做一点补充。

对于OOP，**宏观上来说，设计者都在试图做到的一件事情就是如何当好程序中的上帝。**通过设计良好的接口（这里的良好指的是不多也不少）来对外暴露一个对象的能力，使得使用者只需要充分了解接口，就可以了解这个对象所能提供的能力。

而使用“方法”来表达对象的能力，同使用“变量”相比，从宏观上来说，没有什么区别，只是形式上的不同。但**使用“方法”来表达接口，更容易体现OOP的一个核心理念——“隐藏细节”**

</> 杂项

注意事项

- 作业报告依然建议使用Markdown
- 作业命名规范
- 报告里尽量都写一些