

# JAVA 的面向对象特性

陈德创 19030500217

西安电子科技大学

日期：2020 年 4 月 4 日

## 目录

<b>1</b>	<b>小组名单</b>	<b>3</b>
<b>2</b>	<b>题目</b>	<b>3</b>
<b>3</b>	<b>练习</b>	<b>3</b>
3.1	P15.PassTest(基本类型)	3
3.2	P16,P17.PassTest(引用类型)	3
3.3	P26.DefaultConstructor	4
3.4	P38.PublicVsPackage	4
3.5	P40.PrivateVsPackage	4
3.6	P42.ProtectedVsPackageAndPublic	5
3.7	P51.OrderOfInit	5
3.8	P61.TestInheritance	5
3.9	P76.NotOverriding	6
3.10	P77.PrivOverride	6
3.11	P85.PolyConstruct	6
3.12	P88.PTTI	6
<b>4</b>	<b>题目分析</b>	<b>6</b>
<b>5</b>	<b>程序实现</b>	<b>8</b>
5.1	具体思路	8
5.1.1	MyInteger 类	8
5.1.2	MyNaturalInteger 类	9
5.1.3	Test 类	9
5.1.4	Interact	10
5.2	问题解决	10
<b>6</b>	<b>执行结果</b>	<b>11</b>

7 个人总结	11
附录	13
A <code>protected</code> 修饰猜想验证	13
B 动态绑定	14
C <code>OrderOfInit</code> 拓展代码	15
D 测试结果	16
E 资料参考目录	17

## 1 小组名单

学号	姓名	工作
19030500217	陈德创	完成程序、联系例程、调试、小组讨论

## 2 题目

### 1. PPT 练习

P15,16,17,26,38,40,42,51,61,76,77,85,88

### 2. 程序设计, 要求

- 1). 设计并实现自然数及整数类 (至少能处理二十位自然数或整数数据)
- 2). 应提供数据赋值、数据输出、加减法计算功能 (乘除法选作)
- 3). 设计应体现体系结构 (设计实现完整的继承结构)
- 4). 赋值能正确接收带千分符号的格式及不带千分符号的格式
- 5). 赋值应对于无效的数据向用户提示信息 (选作)
- 6). 应当正确重写 *equals* 和 *toString* 方法来实现数据的比较及格式化
- 7). 自然数类应定义 *toMyInteger* 方法, 创建一个同值的整数类实例
- 8). 应定义成员变量 *length*, 表示当前数据的长度
- 9). 构建测试类, 实现测试功能

## 3 练习

### 3.1 P15.PassTest(基本类型)

这个主要是说 JAVA 中方法参数传递的问题, JAVA 参数是值传递而不是引用传递<sup>1</sup>, 如果是基本类型的话实参和形参是基本无关的 (只是在一开始将实参赋值给形参)。

### 3.2 P16,P17.PassTest(引用类型)

但是要注意的是 JAVA 的类变量为引用变量, 引用变量与实例实体 (我也不知道专业的怎么描述) 是分开储存的, 引用变量只储存了类似于一个指针的东西, 指向实例实体。<sup>2</sup>所以如果在方法中我们改变实例实体的域, 那么实参也会改变 (因为它们指向同一个内容), 如果我们将形参指向一个新的实例, 那么实参的指向是不会发生改变的。(正如 ppt 里面讲的)

还需要注意一点是 JAVA 中的 *String* 是不可变的, 也就是说原来 *str = "Hello"*, 然后改为 *str = "World"*, 并不是 *"Hello"* 改变了, 而是另 *str* 重新指向了 *"World"* (如果内存中不存在的话则会创建再指向)。

<sup>1</sup>探究 java 方法参数传递——引用传递? 值传递!

<sup>2</sup>Java 数据的存储

### 3.3 P26.DefaultConstructor

缺省构造方法：对于没有定义构造方法的类，Java 编译器会自动加入一个特殊的构造方法，该构造方法没有参数且方法体为空，称为缺省构造方法。用缺省构造方法初始化对象时，对象的成员变量初始化为默认值若类中已定义了构造方法，则编译器不再向类中加入缺省构造方法。<sup>3</sup>

ppt 中说的很明白了，加了构造方法后 JAVA 不会再提供缺省构造方法，所以去掉注释程序错误 (找不到参数为空的构造方法)。

### 3.4 P38.PublicVsPackage

default 权限只能被本包内的类访问，而 public 权限可以被任何地方所访问。A 类，B 类和 *PublicVsPackage* 类并不在一个包内，而 B 类为缺省权限，A 类的 *func* 方法也为缺省权限，故 *PublicVsPackage* 类中访问不到 B 类，也访问不到 A 类的 *func* 方法。

### 3.5 P40.PrivateVsPackage

private 权限修饰的方法和属性只能被同类访问到，这里 C 类的 *func1* 方法是 private 所修饰的，所以在 *PrivateVsPackage* 类中无法被访问到。但是 *func2* 方法是可以被访问的。为了进一步测试 private 权限，我们修改代码如下：

```
1  public class PrivateVsPackage{
2      public static void main(String[] args){
3          C obj = new C();C obj2 = new C();
4          obj.data = 5;
5          obj2.data = 15;
6          obj.change(obj2);
7      }
8  }
9  class C{
10     int data;
11     private void func1(){
12         System.out.println("C's method 1");
13     }
14     void func2(){
15         System.out.println("C's method 2");
16         this.func1(); // 同一个类内可以调用func1()
17     }
18     void change(C c){
19         c.data = this.data;
20         System.out.println("this.data : " + this.data);
21         System.out.println("c.date : "+c.data);
22         c.func1();
23     }
24 }
```

---

<sup>3</sup>PPT25 页内容

程序正常运行并输出 `this.data : 5,c.date : 5,C'smethod1`(节省空间, 以',' 代替换行), 说明同类之间 `private` 修饰的方法和属性都是可以被访问到的。

### 3.6 P42.ProtectedVsPackageAndPublic

喜闻乐见的 `protected` 权限, 这个权限我一直拿不准。`protected` 是本包及子类可访问, 如果子类和父类在一个包内那就没什么好说的了, 毕竟 `protected` 比 `default` 还弱一些。如果不在同一包内, 按照我目前的理解, 可以分两种情况。如果子类重写了父类的 `protected` 方法, 那么子类的包内的类都是可以访问到子类的这个方法的 (相当于这是这个方法在这个类中刚定义); 如果不重写, 那么这个方法就相当于子类的一个 `private` 方法, 因为只有子类可以访问这个方法, 而其他类都无法访问到这个方法。而不同包的父类无法调用父类的 `protected` 方法 (对父类来说 `protected` 方法相当于 `default` 方法)。老师给的例程也应证了这一点。

在 `ProtectedVsPackageAndPublic` 类中, 因为这个类既不是 `C` 的子类, 又不与 `C` 在同一包, 所以无论是 `C` 类还是 `CSub` 类都无法访问到 `func` 方法 (被 `protected` 修饰)。`CSub` 类中的 `func` 方法相当于被 `private` 修饰, 所以在 `CSub` 中 `CSub` 对象可以访问 `func` 方法。而对于 `C` 类来说, 它的 `protected` 方法相当于被 `default` 修饰, 显然在不同包的 `CSub` 类中是无法被访问的。

顺着我们的思路, 如果 (不同包) 子类没有重写父类的 `func` 方法, 父类中是可以调用 `sub.func()` 的, 而如果在子类中重写了父类的方法, 那么父类是访问不到的。经过我的实验, 这个是对的, 具体可以看[附录 A](#)。这样也间接说明了, 重写了就是新的。

这里推荐一个菜鸟教程的Java `protected` 关键字详解, 内容详实, 例程丰富。

### 3.7 P51.OrderOfInit

对象的初始化。这里老师讲 `House` 的三个 `Window` 变量放在了三个位置, 但实际上是不影响初始化的顺序的, 即 (不考虑继承和静态) 分配内存空间-> 成员变量默认初始化-> 成员变量显式初始化-> 顺序执行构造方法 (和初始化块) 返回引用。考虑继承实际上是一个回溯的过程, 即对当前要初始化实例的类向上查询是否有父类, 如果有那就执行父类实例的初始化, 执行完毕或者没有父类就执行该类的初始化。执行完毕依次返回。

根据我查找到的一个博客<sup>4</sup>, 如果我们对实例变量直接赋值或者使用实例代码块赋值, 那么编译器会将其中的代码放到类的构造函数中去, 并且这些代码会被放在对超类构造函数的调用语句之后, 构造函数本身的代码之前。这也就是说, 实际上子类属性的显式初始化是在父类初始化之后进行的。这也是这串代码执行结果为 `null` 的原因。

### 3.8 P61.TestInheritance

这里输出 `DrawRectangle,Drawshape`(',' 代替换行), 这里是重写了父类的方法, 并且通过 `super` 关键字调用了父类的方法。如果没有重写, 则会输出两行 `Drawshape`。

---

<sup>4</sup>深入理解 Java 对象的创建过程: 类的初始化与实例化

### 3.9 P76.NotOverriding

前面 ppt 说的很清楚了, 就是子类和父类的同类型同名变量是两套班子 (子类实例包含了一个父类实例)(父类的变量被覆盖, 子类中仍可以通过 *super* 关键字访问到父类的同类型同名变量), 子类变量的更改是不影响父类的 (同样地, 父类也不会影响子类)。所以就算 *Base* 类中的 *i* 变量是 *public* 权限, 程序结果不变。

### 3.10 P77.PrivOverride

注意到这里父类的 *f* 方法是 *private*, 所以其实在子类中再次声明 *f* 方法是不发生重写的。所以在这里父类引用引用子类类型调用 *f* 方法同样是调用父类的 *f* 方法, 而如果父类 *f* 方法修饰为 *public*, 则子类发生重写, 程序结果应与 ppt75 页一致, 即父类引用引用哪个子类就调用子类方法, 引用父类就调用父类方法。参见附录 B。

### 3.11 P85.PolyConstruct

这个根据结果来看方法重写是在父类对象构造之前的, 但是这个依然令我很困惑。子类调用了父类的构造方法, 父类构造又调用了 *draw* 方法, 那么这个方法算作是谁 (父类还是子类) 调用的呢? 又或许我这个思路有些问题。不过虽然不是很透彻地明白原理 (我进行资料查找时也没有找到这一样方面的内容), 不过接用附录 C 中给出的博客, 我类比给出流程:

创建子类的时候会先创建基类, 创建基类的时候调用了 *draw* 方法, 由于子类重写了该方法, 所以调用的是子类中的方法 (经过我实验只有重写会调用子类的)。但是此时子类还未构造, 所以可能在更复杂的环境下会产生一些意想不到的错误。

这里其实也是多态的一种体现: 有同名方法执行子类的。

### 3.12 P88.PTTI

这里就好说了, 父类引用子类实例只能使用父类中已有的方法, 所以 *x[1].u()* 会报错, 因为父类中没有这个方法, 就算 *x[1]* 引用的是子类的实例也没用。而父类引用了父类实例经过强制类型转化之后, 对强制转换结果进行运行时类型识别, 若结果类型与子类类型不符, 所以会报错。

## 4 题目分析

#### 1. 实现至少处理二十位的自然数或整数

这个其实不难, 因为高精度算法也是一个比较常见并且基本的算法了。我们参考 *BigInteger* 类的设计, 利用数组存储数字, 并且每个实例一旦被初始化, 内容无法改变 (即属性为常量)。水平所限, 我们退而求其次, 每一个整型只储存 9 位十进制数。难的是后面的内容, 即整个类型的设计。

#### 2. 设计应体现体系结构

这个应该是这次作业应着重思考的地方, 如何设计类的结构, 也是这次课程的重点。因为

只有两个类，所以结构可以考虑的不多，也就自然数继承整数，整数继承自然数，这两个共同继承一个 *MyNumber* 类。

i. 自然数继承整数

这应该是最自然的一种想法，因为自然数是整数的一种。这样我们在整数中完成所有的功能，在自然数类主要增加数据合法性的判断就可以了。

ii. 整数继承自然数

emmmmm..... 我觉得这样考虑的人不多，因为怎么想也都太奇怪了一些。不过其实实现起来还是比较自然的，因为自然数主要处理数字部分，而整数主要处理符号部分（数字部分交给父类自然数去做）。

iii. 同时继承共同父类

这个我还真没想好怎么设计，但是看到同学给出了这样的设计。我认为这样的话共同父类和整数类几乎没有差别，从而不过多此一举。同学利用异常类予以反驳，说异常类就是子类几乎只是调用父类方法，我们要的就是这个继承结构而已。emmmmm 实力所限，我无从反驳。

这里我们还是采用第 *i* 种方法，其实前两种我都试了一下，直到最后看到 ppt “继承是一种 ‘is a’ 关系”，而且对于使用来说自然数继承整数还是要自然一些。

3. 赋值能接受数字字符串格式

这个也好做。因为我们是利用数组储存数据，循环处理即可，着重要做的是合法性判断。

4. 对于无效的信息对用户做出提示

要着重注意的地方我认为主要是自然数的减法以及用户的数据读入。处理思路有两个，一是抛出异常，二是返回一个类似 *NaN* 的常量，用于表示无效数。在获得实例的时候我们选择前者，在自然数的 *minus* 方法中如果溢出则返回 *null*。

5. 重写 *equals* 和 *toString* 类

i. *equals* 就是符号比较然后逐个数据比较就可以了因为我们实现了 *Comparable* 接口，所以就直接调用 *compareTo* 了。据说是重写 *equals* 时也要重写 *hashCode*<sup>5</sup>，所以我们也重写了，不过不知道对不对.....

ii. *toString* 利用 *StringBuilder* 把原来的整型逐个“装”起来就可以了，注意补零以及符号处理。

6. 自然数定义 *toMyInteger* 方法

这个其实好弄，因为我们的属性都是 *final*，所以这实际上就是新创建了一个。

7. 成员变量 *length*

初始化的时候顺便初始化了就好了。反正我们设计的都是常量，所以这个其实不用太关心。实际上在设计的时候，我发现用 *length* 表示数组长度而不是数据长度会方便很多，所以在设计的时候我们也是这样，只是在最后的 *getLength* 方法中转化一下。

8. 测试类

测试类我认为要满足两个需求，一是测试我们设计的真确性，二是用户使用。对于第一个写一个对拍就可以了，对于第二个因为我不会 *GUI*，所以就用命令行来代替吧。大体实现

---

<sup>5</sup>重写 *equal()* 时为什么也得重写 *hashCode()* 之深度解读 *equal* 方法与 *hashCode* 方法渊源



和上次 *MyCalendar* 类的实现差不多，比如：

```
1      int a 123231312 \\初始化一个整数a并赋值为123231312
2      unsigned b 213 \\初始化一个自然数b并赋值为213
3      add a b \\将a和b求和
```

问题的焦点在于我们如何去储存变量(比如实例中的 a,b),我认为用一个字符串到 *MyInteger* 的 *HashMap* 就可以了，增加查询都很方便。然后就是命令解析和错误提示了。我们将命令解析和错误提示分开，也就是说我们定义一个 *commandParse* 方法，用于解析一串字符串命令，然后根据这个方法的返回值我们进行错误提示。好处就是条例清晰，逻辑性强，并且可以做到统一处理。

## 5 程序实现

### 5.1 具体思路

#### 5.1.1 *MyInteger* 类

##### 1. 储存方式

我们用 *int[] val* 储存数字，用 *int symbol* 储存符号。数均为 *final*，意味着这些值在对象被实例化的时候就已经确定，且直到对象被回收都不会改变。说实话我不知道这样做的好处是什么，为了安全性还是常量可以提高运行效率，这里只是仿造了 *BigInteger* 类的实现。数字是以 *val[0]* 为最大位的顺序储存的。

##### 2. 构造方法

我们只提供一个构造方法 *MyInteger(int[] val,int symbol)*，为包权限 (因为考虑到别人用我们的类一般是在不同的包，所以用户是访问不到这个构造方法的)。设置静态方法 *public static getInstance(String num)*，这是外部除四则运算和 *clone* 外唯一一个获得实例的接口。这个方法会调用会判断字符串的合法性，然后调用 *static preString* 获得我们需要的 *val* 数组，最后调用构造方法。没把 *getInstance* 和 *preString* 和在一起主要还是为了方便子类的调用。

##### 3. add AND minus

因为正负号的原因 (加一个负数等于减一个正数)，所以我们还是对符号和数字分开解决。于是我们 (参考 *BigInteger* 类) 定义了两个静态方法，*int[] add(int[] x,int[] y)* 与 *int[] minus(int[] x,int[] y)*，用于处理数字部分的加减，注意 *minus* 方法中默认 *x* 表示的数字大于 *y* 表示的数字 (在调用时保证)。这样我们的成员方法 *add,minus* 的主要功能就是符号判断，然后调用静态 *add,minus*，*new* 一个 *MyInteger* 就可以了。

(静态方法) 具体的加法实现就是一个模拟的过程，从末位各位相加，注意进位加到下一位。最后有点细节就是到最后可能多一个进位，所以要新建一个数组。减法也是如此，为了程序简便，我们默认借一位，然后实行减法。这个最后处理的细节是可能最高位是零，所以需要也要新建一个数组。

##### 4. multiply

还是符号与数字分开。做一个优化处理，即如果某一个为零，直接返回 0。对于数字部分，



容易想到一个  $m$  位的数字乘以一个  $n$  位的数字得到的最高是一个  $m+n$  位, 最少是  $m+n-1$  位。所以我们声明一个  $m+n$  位的数组 *result* 用于储存结果, 然后从末尾二重循环依次乘过来就可以了, 也是一个模拟的思路。注意 *x.val[xIndex] \* y.val[yIndex]* 乘出来应该在 *result[xIndex + yIndex + 1]*, 进位的话进到 *xIndex + yIndex* 位, 也就是下次操作的位。最后细节处理就是最高位 *result[0]* 可能为 0, 因为我们声明 *result* 的位数为  $m+n$  位, 而  $m$  位数字乘  $n$  位数字结果长度可能位  $m+n-1$  位 (如  $1000 * 100$ )。

#### 5. toString

分为两部分, 符号处理 (如果是负数前面加 '-', 如果是正数则不加), 另一个是数字部分, 这个好数, 用 *StringBuilder* 就可以解决, 从 *val[0]* 到 *val[length - 1]* 依次装进去, 要注意的是除了 *val[0]* 其他部分如果不足 9 位十进制数 (我们规定的数组每位储存的十进制数长度), 要补零。这里 *String.format* 可以很轻松的办到。

#### 6. compareTo

我们程序实现了 *Comparable* 接口, 具体实现也是符号判断与数值判断分开的, 因为正数一定大于负数和零嘛。我们定义了一个成员方法 *compareAbs* 用于比较数字部分的大小 (也就是绝对值的大小), 这样分类讨论一下就可以了。

#### 7. clone

emmmmmm 这个方法我重写的很敷衍, 把异常去掉了, 没有调用父类的方法, 直接按照当前的域值 *new* 了一个同样的。

#### 8. 其他细节

我们定义了几个类变量, *POSITIVE*, *NEGATIVE*, *ZERO*, 前两个为 *int* 类型用来表示符号, *ZERO* 为 *MyInteger* 类型为了返回零方便, 也是为了保证零唯一吧。

### 5.1.2 MyNaturalInteger 类

这个实际上没啥好说的了, 直接调用父类的方法, 注意每个方法我重载了参数为 *MyInteger* 和 *MyNaturalInteger*, 为了保持兼容。像 *int* 和 *long* 一样, 只要运算中出现了 *MyInteger*, 那么返回值就为 *MyNaturalInteger*。但是在编写测试类之后我更改了实现<sup>6</sup>, 取消了重载, 将参数设为 *MyInteger*, 返回类型也是 *MyInteger*, 只是如果只是自然数运算返回实例为 *MyNaturalInteger*。

这样的话原本子类的 *minus* 方法如果溢出是要抛出异常的, 因为子类重写不能抛出比父类大的异常, 而我们是不想让父类也有异常, 所以就改为如果子类运算溢出, 那么返回 *null*。

要注意的是 *getInstance* 的处理, 负数也是不被允许的了。整体实现与父类相仿。

**toMyInteger** 直接 *new* 了一个 *MyInteger* 就可以了。

### 5.1.3 Test 类

这个就没什么好说的了, 就是构造两个个数字字符串, 然后分别用 *BigInteger* 类和我们的类进行运算, 然后将结果转化为字符串比较就可以了。构造数字字符串我们可以用多个 *Math.random*

---

<sup>6</sup>见子类多态问题

转化成 *int* 然后拼接起来。具体代码附件，代码还是基本经得起检验的 (不过是等于只检验了构造、运算和转化为字符串部分，而且是随机化的，偶然性较大)。

### 5.1.4 Interact

细节方面主要还是原始命令字符串 *preCmd* 的处理，我们对原始命令字符串 *preCmd* 加上 `split`，利用 *split* 按空格分割到 *command* 这个字符串数组，可以有效防止后期处理的时候下标溢出。

因为我们加减乘是支持变量名或者数字的，这里细节其实不好处理，因为是二目运算符，算起来有四种情况，再算上加减乘一共有 12 种情况还有整数、自然数的判断，48 种情况，用 if 判断怕不是要把人绕死。这里我们巧妙地定义了一个 *operate* 方法，接受两个 *MyInteger* 类型参数 (可能指向 *MyNaturalInteger* 类型实例) 和一个字符串，字符串即为操作 (*add, minus, multiply*)，返回一个 *MyInteger* 类型引用 (可能指向 *MyNaturalInteger* 类型实例)，为运算完的数。整数自然数的判断交给 *add, minus, multiply* 方法实现中去做。为此我稍微改变了一下架构。

这里还有就是我们支持 *add a b c* 和 *add a b* 两种选择，前者将 *a + b* 储存在 *c* 中，后者直接将结果打印并储存在默认的 *ans* 中。这里的处理时我们判断 *command[3].equals(",")*，如果是的话就是直接打印，不是的话就是储存。(说起来这整个处理我还是比较得意的)

## 5.2 问题解决

### 1. 乘法运算中数字部分出现负数

这个是因为我的数字部分每一位是储存的九位十进制数，然后乘法运算的时候要相乘，所以就爆掉了 *int* 的数据范围，虽然我是用一个 *long* 型的 *sum* 来储存这个结果的，但是运算过程中就已经爆掉了。所以在运算的时候就要进行强制类型转化。如图：

```
for (int xIndex = this.length - 1; xIndex >= 0; xIndex--){
    for (int yIndex = y.length - 1; yIndex >= 0; yIndex--){
        sum = sum + (long) val[xIndex] * y.val[yIndex] + result[xIndex + yIndex + 1];
    }
}
```

val为int类型，这里如果不化成long会爆掉数据范围，出现负数

### 2. final

这个可把我弄惨了，因为要用 *final*，那就必须在构造方法中确定成员变量的值。这对于父类还好说，对于子类因为第一行就要调用父类的方法，所以是没办法再去进行合法性判断的。不过采用 *getInstance* 来获得实例变量之后，这个问题迎刃而解，因为构造方法是不对外开放的，*getInstance* 方法可以保证调用构造方法时的数据合法性。

### 3. 父类构造方法权限问题

父类和子类的构造方法我们显然是不想让外部直接访问到的 (因为里面没有合法性判断)，但是如果父类构造方法设成 *private* 权限，子类无法访问到，那么也就相当于父类无法被继承。所以只好退而求其次我们设为包权限。如果有一种权限只有父类及其子类可以访问到，那想必是极好的。

### 4. 交互类相关问题

交互类我预想的是可以 *int* 一个变量储存起来，这样要用到 *Map*，然后重写一下 *hashCode*。

问题主要是这样的话 *add, minus* 之类的运算肯定是要支持多态, 即 *add a b*, 其中 *a, b* 可以为变量名, 也可以为数字字符串。这样处理起来很棘手, 一大推 *if* 会导致程序冗长。这里我们新建了两个父类类型, 并让他们指向变量 *a* (如果 *a* 是一个已有的变量), 或者数字字符串所代表的 *MyInteger* 类型值 (如果 *a* 是一个数字字符串)。注意数字字符串我们默认为 *MyInteger* 类型。

## 5. 子类多态问题

在子类的操作方法 (加减乘中), 我重载了参数为自然数和参数为整数的两种, 在测试类设计中, 我发现 *jvm* 对重载的选择是根据引用类型而不是实例类型。这也就是说, 如果我以父类引用子类实例为参数调用子类的方法, 那么 *jvm* 会选择父类类型参数的方法, 而不是子类类型参数的方法。由此, 我取消了子类方法的重载, 即子类中的加减乘方法都接受父类类型, 返回父类类型。也就是说如下代码:

```
1 MyNaturalInteger a = getInstance("1234");
2 MyNaturalInteger b = getInstance("213213");
3 MyNaturalInteger c = a.add(b);
```

会报错, 但是 *add* 方法返回的是子类实例, 所以这一步要进行强制向下转型。正如 *short* 类型相加也需要强制类型转换才能赋给 *short* 一样。这样做的好处我尚不明朗, 不过确确实实在编写交互类的时候轻松了很多, 因为就直接全部弄成 *MyInteger* 类型的引用就可以了, 实例在操作方法中进行区分, 很方便。

## 6. 其他问题

其实在实践过程中遇到了很多问题, 但是因为沒有做好记录, 前后时间间隔又太长, 所以一时间想不起来了。最主要的问题其实还是继承结构的问题, 在敲定了自然数继承整数之后其他只顾想办法就可以了。

# 6 执行结果

因为这次的作业主要是一个工具类的设计, 还有人机交互类的设计。在这里放几个截图简单示意一下吧。因为图片过大, 放在了[附录 D](#)。

# 7 个人总结

怎么说呢, 这一次作业做的时间很长, 交上去报告又回炉重造了一下, 感觉完善了很多, 也算基本完成任务吧。报告上, 问题还是在于论述不清, 很难将自己的想法表达出来, 而且一旦表达很容易牵扯到很多技术上的细枝末节, 偏离了原有的方向。

对于这次作业, 我觉得如果培养目标放在类的继承关系上, 这个题目会令学生将过多的精力放在算法设计上, 而不是思考类的继承关系上 (因为自然数继承整数是自然的)。

整体来说, 在编写交互类的时候问题还是很多的, 不过这次写完感觉自己能力还算有一个不错的提升。这次的命令解析要比日历类难很多, 主要还是难在命令的多态上, 即一个命令可能支持多个操作形式。

不过话说回来，这次作业令我受益匪浅。因为自己稍微有点基础，所以算法方面不用太操心，但是具体实现上还是困难重重，还是有些功夫不到家。特别是思考问题的角度方面，之前写高精度都是自己用，不用过多的担心些什么，但是现在目标是别人用，考虑的多得多。这或许就是工程代码和竞赛代码的区别吧。(想起了一个笑话，一位测试工程师走进酒吧，点了一杯咖啡，点了 `null` 杯啤酒，点了 `-1` 杯啤酒，点了 `NaN` 杯啤酒，点了一个桌子(笑))

再接再厉吧。

# 附录

## A protected 修饰猜想验证

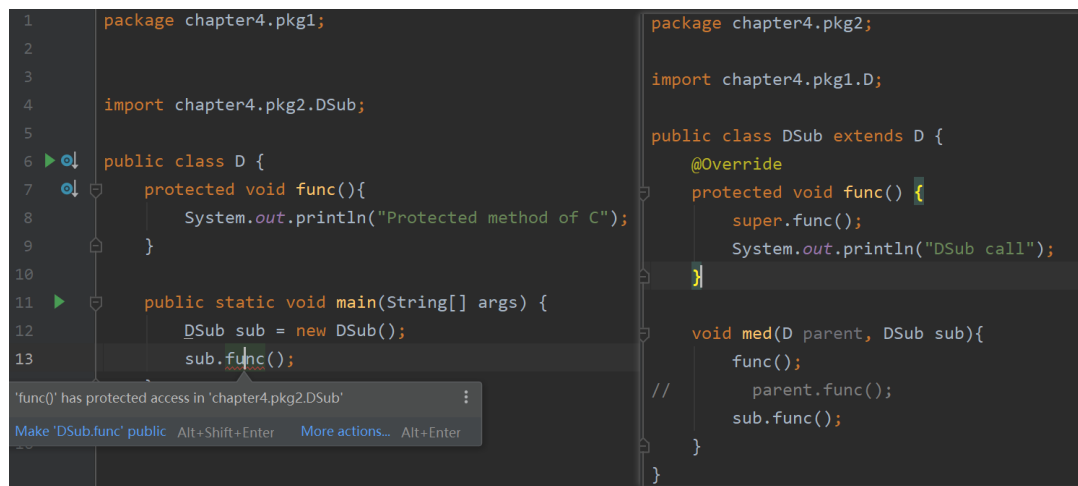
图 1: 子类不重写父类的方法



```
1 package chapter4.pkg1;
2
3
4 import chapter4.pkg2.DSub;
5
6 public class D {
7     protected void func(){
8         System.out.println("Protected method of C");
9     }
10
11     public static void main(String[] args) {
12         DSub sub = new DSub();
13         sub.func();
14     }
15 }
16
```

```
1 package chapter4.pkg2;
2
3 import chapter4.pkg1.D;
4
5 public class DSub extends D {
6     // @Override
7     // protected void func() {
8     //     super.func();
9     //     System.out.println("DSub call");
10 }
11
12 void med(D parent, DSub sub){
13     func();
14     parent.func();
15     sub.func();
16 }
```

图 2: 子类重写父类的方法



```
1 package chapter4.pkg1;
2
3
4 import chapter4.pkg2.DSub;
5
6 public class D {
7     protected void func(){
8         System.out.println("Protected method of C");
9     }
10
11     public static void main(String[] args) {
12         DSub sub = new DSub();
13         sub.func();
14     }
15 }
16
```

```
1 package chapter4.pkg2;
2
3 import chapter4.pkg1.D;
4
5 public class DSub extends D {
6     @Override
7     protected void func() {
8         super.func();
9         System.out.println("DSub call");
10 }
11
12 void med(D parent, DSub sub){
13     func();
14     parent.func();
15     sub.func();
16 }
```

'func()' has protected access in 'chapter4.pkg2.DSub'

Make 'DSub.func' public Alt+Shift+Enter More actions... Alt+Enter

## B 动态绑定

图 3: 动态绑定

```
1 package chapter4.pkg2;
2
3 public class PrivOverride {
4     public void f() { System.out.println("private f()"); }
5     public static void main(String[] args) {
6         PrivOverride po = new PrivOverride();
7         po.f();
8         po = new Derived();
9         po.f();
10        ((Derived) po).f();
11        Derived de = new Derived();
12        de.f();
13    }
14 }
15 class Derived extends PrivOverride {
16     public int a;
17     public void f() { System.out.println("public f()"); }
18 }
19
```

PrivOverride > f()

Run: PrivOverride x

C:\Users\RainCurtain\.m2\repository\org\jetbrains\annotations\17.0.0\annotations-17.0.0.jar chapter4.pkg2.f

private f()  
public f()  
public f()  
public f()

## C OrderOfInit 拓展代码

注: 参考自:例题: 子类重写父类方法后的调用规则

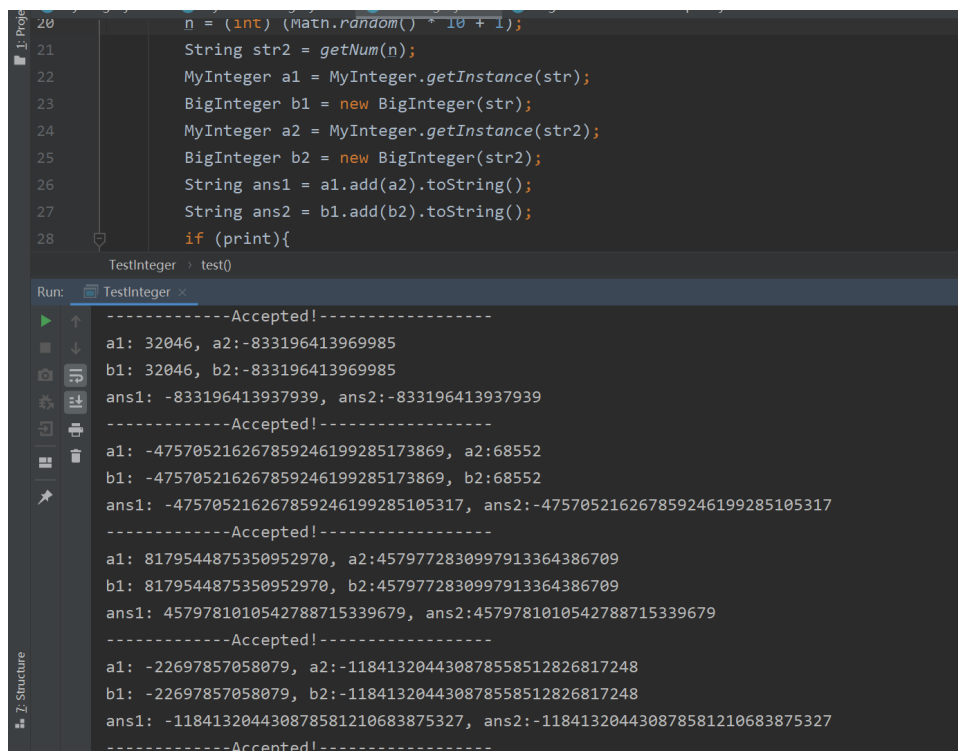
```
1  public class Base {
2
3      private String baseName= "base";
4      public Base(){
5          callName();
6      }
7
8      public void callName(){
9          System.out.println(baseName);
10     }
11
12     static class Sub extends Base{
13         private String baseName = "sub";
14         public void callName(){
15             System.out.println(baseName);
16         }
17     }
18
19     public static void main(String[] args){
20         Base b = new Sub();
21     }
22
23 }
```



## D 测试结果

程序测试结果如下 (对  $\text{L}^{\text{T}}\text{E}^{\text{X}}$  不熟悉, 这个排版我无能为力啊)(尾页还有):

图 4: 加法测试



```
20     n = (int) (Math.random() * 10 + 1);
21     String str2 = getNum(n);
22     MyInteger a1 = MyInteger.getInstance(str);
23     BigInteger b1 = new BigInteger(str);
24     MyInteger a2 = MyInteger.getInstance(str2);
25     BigInteger b2 = new BigInteger(str2);
26     String ans1 = a1.add(a2).toString();
27     String ans2 = b1.add(b2).toString();
28     if (print){
TestInteger -> test()

Run: TestInteger x
-----Accepted!-----
a1: 32046, a2:-833196413969985
b1: 32046, b2:-833196413969985
ans1: -833196413937939, ans2:-833196413937939
-----Accepted!-----
a1: -475705216267859246199285173869, a2:68552
b1: -475705216267859246199285173869, b2:68552
ans1: -475705216267859246199285105317, ans2:-475705216267859246199285105317
-----Accepted!-----
a1: 8179544875350952970, a2:4579772830997913364386709
b1: 8179544875350952970, b2:4579772830997913364386709
ans1: 4579781010542788715339679, ans2:4579781010542788715339679
-----Accepted!-----
a1: -22697857058079, a2:-118413204430878558512826817248
b1: -22697857058079, b2:-118413204430878558512826817248
ans1: -118413204430878581210683875327, ans2:-118413204430878581210683875327
-----Accepted!-----
```

```

ans1: 8788652965726885532390645122998175931150, ans2:8788652965726885532390645122998175931150
-----Accepted!-----
a1: 6268753705634146229, a2:2697688637770085264327369914797421252488
b1: 6268753705634146229, b2:2697688637770085264327369914797421252488
ans1: 2697688637770085264333638668503055398717, ans2:2697688637770085264333638668503055398717
-----Accepted!-----
a1: 3774742000533377621135959759298784316161, a2:-841651274957456361278709671512
b1: 3774742000533377621135959759298784316161, b2:-841651274957456361278709671512
ans1: 3774741999691726346178503398020074644649, ans2:3774741999691726346178503398020074644649
-----Accepted!-----
a1: 6199634073585431118647520, a2:8898383528
b1: 6199634073585431118647520, b2:8898383528
ans1: 6199634073585440017031048, ans2:6199634073585440017031048
-----Accepted!-----
a1: -670052722170778751324586229846, a2:-768288540614239140844874515887
b1: -670052722170778751324586229846, b2:-768288540614239140844874515887
ans1: -1438341262785017892169460745733, ans2:-1438341262785017892169460745733
-----Accepted!-----
a1: 358331087822055142131509115799367829409, a2:80432924604535980851
b1: 358331087822055142131509115799367829409, b2:80432924604535980851
ans1: 358331087822055142211942040403903810260, ans2:358331087822055142211942040403903810260
-----Accepted!-----
a1: -33635424671341446474998477315, a2:-163773345951419943358002223864682168497209
b1: -33635424671341446474998477315, b2:-163773345951419943358002223864682168497209
ans1: -1637733459514233069004693565311157166974524, ans2:-1637733459514233069004693565311157166974524
-----Accepted!-----

```

## E 资料参考目录

在这里列举一下报告中参考的博客，同时收集起来方便自己再次查阅。

探究 java 方法参数传递——引用传递？值传递！

Java 数据的存储

Java protected 关键字详解

深入理解 Java 对象的创建过程：类的初始化与实例化

例题: 子类重写父类方法后的调用规则

从 jvm 角度看懂类初始化、方法重写、重载。

Java 之动态绑定与静态绑定

重写 equal() 时为什么也得重写 hashCode() 之深度解读 equal 方法与 hashCode 方法渊源

JVM 内存模型 (jvm 入门篇)

图 5: 减法测试

```
-----Accepted!-----
a1: 54438868667928212056514556805016425803366704587954, a2:-578683358890511
b1: 54438868667928212056514556805016425803366704587954, b2:-578683358890511
ans1: 54438868667928212056514556805016426382050063478465, ans2:54438868667928212056514556805016426382050063478465
-----Accepted!-----
a1: 788829347226025457316294, a2:5578838769
b1: 788829347226025457316294, b2:5578838769
ans1: 788829347226019878477525, ans2:788829347226019878477525
-----Accepted!-----
a1: 4605448194720175018442669180408164683516, a2:121175816277014260197456719241456863015
b1: 4605448194720175018442669180408164683516, b2:121175816277014260197456719241456863015
ans1: 4484272378443160758245212461166707820501, ans2:4484272378443160758245212461166707820501
-----Accepted!-----
a1: -180837426696027852953538824439871481047147729, a2:82933
b1: -180837426696027852953538824439871481047147729, b2:82933
ans1: -180837426696027852953538824439871481047230662, ans2:-180837426696027852953538824439871481047230662
-----Accepted!-----
a1: 5862094445165264800588291227158846471579, a2:-57406593633062123260480353387742771669740711
b1: 5862094445165264800588291227158846471579, b2:-57406593633062123260480353387742771669740711
ans1: 57412455727507288525280941678969930516212290, ans2:57412455727507288525280941678969930516212290
-----Accepted!-----
a1: 989492921584537459616858, a2:4157554249
b1: 989492921584537459616858, b2:4157554249
ans1: 989492921584533302062609, ans2:989492921584533302062609
-----Accepted!-----
a1: 60738093416781946640109723634582160933, a2:5071780555953021828638879711835841574495
b1: 60738093416781946640109723634582160933, b2:5071780555953021828638879711835841574495
ans1: -5011042462536239881998769988201259413562, ans2:-5011042462536239881998769988201259413562
-----Accepted!-----
```

图 6: 乘法测试

```
testInteger
-----
a1: -7390399531768, a2:82989
b1: -7390399531768, b2:82989
ans1: -613321866741894552, ans2:-613321866741894552
-----Accepted!-----
a1: 28367511316367548778, a2:73151438815961821156677
b1: 28367511316367548778, b2:73151438815961821156677
ans1: 2075124268420365325031751540717723177890706, ans2:2075124268420365325031751540717723177890706
-----Accepted!-----
a1: 711798940997622344741209766589613742393, a2:383117952899039213431535994073265616437
b1: 711798940997622344741209766589613742393, b2:383117952899039213431535994073265616437
ans1: 272702953150713069622592947049269457045204994297534471894936131680149664513741,
ans2:272702953150713069622592947049269457045204994297534471894936131680149664513741
-----Accepted!-----
a1: 87509, a2:-130854341262545
b1: 87509, b2:-130854341262545
ans1: -11450932549544050405, ans2:-11450932549544050405
-----Accepted!-----
a1: 57161, a2:7648726609971914547284013464409631887
b1: 57161, b2:7648726609971914547284013464409631887
ans1: 437208861752604607437301493639118968292807, ans2:437208861752604607437301493639118968292807
-----Accepted!-----
a1: 29693, a2:53470471624462240414495146992962321729054761147124
b1: 29693, b2:53470471624462240414495146992962321729054761147124
ans1: 1587698713945157304627604399662030219100823022741552932, ans2:1587698713945157304627604399662030219100823022741552932
-----Accepted!-----
a1: -38603835792992495452962946085795529125531721, a2:36081
b1: -38603835792992495452962946085795529125531721, b2:36081
ans1: -1392864999246962228438356057721588486378310025401, ans2:-1392864999246962228438356057721588486378310025401
-----Accepted!-----
```

图 7: 交互类: 基本测试

```
Print 打印帮助文档
-----HELP DOCUMENT-----
int a 123,          声明一个值为123的整数a
unsigned a 123,     声明一个值为123的自然数a
add/minus/multiply a b, 将a,b相加/减/乘, 结果储存在ans变量中
add/minus/multiply a b c, 将a,b相加/减/乘, 结果储存在c变量中
show a,            打印a的值
help,              打印此文档
quit,              退出
int a 3123123213131232131434234234234  声明变量并赋初值
show a  打印变量 (包括类型)
int : 3123123213131232131434234234234
int b 2493208423908932084239084023984392084329084390
show b
int : 2493208423908932084239084023984392084329084390
unsigned c 2132131232103891238190238120938219038109238219038123902183901381290  声明自然数
show c
unsigned int : 2132131232103891238190238120938219038109238219038123902183901381290
minus a b c
int : -2132131232103891238190238120938219034986115005906891770749667147056  整数减自然数, 结果为整数
show ans
int : -2132131232103891238190238120938219034986115005906891770749667147056  结果自动打印并赋给默认变量ans
minus b c d
int : -2132131232103891238190238120938219034986115005906891770749667147056  b-c并赋给d, 如果不存在d则会创建
show d
int : -2132131232103891238187744912514310106024999135014139510099572296900
multiply c d
int : -454598359091285733135859325216267994692293196946349161741509624654751100346527189156493484472545688072614785177005754638127985001000
quit
-----THANK FOR USING-----
```

图 8: 交互类: 错误提示展示

```
Input 'help' to show HelpDocument.
int  没有数字, 显示数字格式错误
-----ILLEGAL NUMBER ERROR-----
asdf 找不到命令
-----COMMAND NOT FOUND-----
int a
-----ILLEGAL NUMBER ERROR-----
unsigned a 123
unsigned b 1234
minus a b 自然数减法溢出
-----ILLEGAL NUMBER ERROR-----
minus a c
-----ILLEGAL NUMBER ERROR----- 没有c这个变量, 所以c会被认为是一个数字字符串
show c
-----CAN NOT FOUND VARIABLE----- 没有c这个变量, 找不到c
int a
-----ILLEGAL VARIABLE NAME-----
int a 1234 a已经被声明过了, 这里重复声明了
-----ILLEGAL VARIABLE NAME-----
quit
-----THANK FOR USING-----

Process finished with exit code 0
```