

Java 第四章 面向对象特性上机练习

22009200601 汤栋文

2023 年 3 月 28 日

目录

1.	个人信息	- 2 -
2.	任务重述	- 2 -
3.	PPT 练习	- 2 -
3.1	PassTest on P15 P16 P17	- 2 -
3.2	DefaultConstructor on P26	- 3 -
3.3	Permission on P38 P40 P42	- 3 -
3.4	OrderOfInit on P51	- 4 -
3.5	TestInheritance on P61 P65 P66	- 4 -
3.6	ConstructorOverloading on P72 P73	- 6 -
3.7	Overriding on P77 P78	- 6 -
3.8	PolyConstruct on P86	- 8 -
3.9	RTTI on P89	- 8 -
4.	模拟饲养宠物程序	- 9 -
4.1	完整代码	- 9 -
4.2	回答问题	- 11 -
5.	编写复数类	- 12 -
5.1	成员变量	- 12 -
5.2	加减乘除运算	- 12 -
5.3	重写三个方法	- 12 -
5.4	重写其他方法	- 13 -
5.5	其他便捷组件	- 13 -
5.6	其他说明	- 14 -
5.7	测试类和结果展示	- 14 -
5.8	任务总结	- 15 -
6.	总结	- 15 -

1. 个人信息

姓名：汤栋文

学号：22009200601

日期：2023 年 3 月 28 日

2. 任务重述

1. 执行 PPT《JAVA 面向对象特性》中练习
(页码: 15, 16, 17, 26, 38, 40, 42, 51, 61, 65, 66, 72, 73, 77, 78, 86, 89)
2. 补全模拟养宠物的程序并回答问题。
3. 编写一个程序，实现复数运算：
 - a) 设计并实现复数类。
 - b) 设计并实现依据各种基本数值类型值构建复数对象实例的方法。
 - c) 设计并实现复数的加减乘除运算。实现任意数值类型数据与复数对象实例的运算功能。
 - d) 参考 Integer 类源码，重写复数的 equals 方法、toString 方法及 hashCode 方法。
 - e) 构建测试类，该类实现与用户的交互，接收用户键入的数据，并完成类功能的测试。

3. PPT 练习

3.1 PassTest on P15 P16 P17

1. 基本类型:这个主要是说 JAVA 中方法参数传递的问题,JAVA 基本类型(例如 int,double, boolean)是值传递而不是引用传递,对于基本类型来说实参和形参是基本无关的(只是在开始将实参赋值给形参)。
2. 引用类型:如果我们给函数传入一个引用类型,那么相当于在 C 中传入一个指针。当我们修改了其中的内容时,在函数外面查找到的这个值也会发生改变,因为他们在内存中实际上只存储了一份。这个引用传入前或者是传入后其实都指向同一块区域。
3. 但是如果我们尝试在函数内部改变了这个引用,(不是改变这个引用指向的那个对象的内容),那么现在存在了两个引用,函数内部的引用指向函数内部的实例内容,而外部的指向外部的内容。两者互不相干。

```
public class PassTest16 {
    public void changeStr(String value) { // 引用类型参数
        value = new String( original: "different");
    } //方法中改变形参所指对象
    public static void main(String[] args) {
        String str;
        PassTest16 pt = new PassTest16();
        str = new String( original: "Hello");
        pt.changeStr(str); // 引用类型参数的传递
        System.out.println("Str value is: " + str);
    }
}

public class PassTest17 {
    float ptValue;
    public void changeObjValue(PassTest17 ref) { // 引用类
        ref.ptValue = 99.0f; //改变参数所指对象的成员变量值
    }
    public static void main(String[] args) {
        PassTest17 pt = new PassTest17();
        pt.ptValue = 101.0f;
        pt.changeObjValue(pt); // 引用类型参数的传递
        System.out.println("Pt value is: " + pt.ptValue)
    }
}
```

PassTest16 × PassTest17 ×

"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:" "C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:D:
Str value is: Hello Pt value is: 99.0

3.2 DefaultConstructor on P26

1. 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。
2. 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。
3. 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)。

```
class Bird {
    int i;
    public Bird(int j){ i=j; }
}

public class DefaultConstructor {
    public static void main(String[] args) {
        Bird nc = new Bird(); // 缺省构造方法
    }
}
```

Expected 1 arguments but found 0
Remove 1st parameter from constr

3.3 Permission on P38 P40 P42

1. 这张表很好的反映了 JAVA 中的这些权限。
2. 其中 public 修饰的变量或方法可以被任何地方访问。这相当于把类封装成一个黑箱以后留下来的几个口, 让用户可以从这几个孔进行适当的操作而不损坏其他的东西。
3. default 权限只能被本包内的类访问, 通常情况下一个包中的类之间联系比较紧密。
4. protected 权限相当于在 default 的基础上允许子类的访问, 个人认为这个权限是为了解决继承以后再次开发问题。例如我如果想要在 Number 类的基础上构建 Complex 类, 我总不能把我的 Complex 写到 Number 一个包里面。而是需要在包外部继承 Number 类, 然后使用相关内容。如果用 Number 里面的函数用 default 权限那么我将无法使用, 如果 Number 里面用 public 权限, 在正常使用时其实是不希望让用户接触到这个函数的。如果把类比作一个黑箱子, public 是箱子上留的洞, 那么 protected 相当于给开发者留了一块可以拆卸的板子, 在有需要时可以修改些内容, 而在平时使用时却不会打开。(这个例子可能不太合适...但是应该表达清楚了我的理解)

	同一个类	同一个包	子类	全局
private	√			
default	√	√		
protected	√	√	√	
public	√	√	√	√

3.4 OrderOfInit on P51

1. 对象的初始化。这 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，不考虑继承和静态的情况下，分配内存空间->成员变量默认初始化->成员变量显式初始化->执行构造方法->返回引用。
2. 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。
3. 在这个例子中，我们可以看到，先打印了 father 又打印了 son，这说明 Java 先执行了父类变量的初始化，然后执行父类的构造函数，而后执行子类变量的初始化，然后执行子类的构造函数。实际上子类属性的初始化是在父类完全完成初始化之后进行的。

```

class Window {Window(int m) {System.out.println("window "+m);}}
class House {
    Window w1 = new Window(1);
    House() { System.out.println("House");}
    Window w2 = new Window(2);
    void f() { System.out.println("f()"); }
    Window w3 = new Window(3);
    public class OrderOfInit {
        public static void main(String[] args) {
            House h = new House(); h.f(); }
    }
}

class Father {
    String father_name = "father";
    Father() {System.out.println(father_name); }
}

class Son extends Father {
    String father_name = "son";
    Son() {System.out.println(father_name); }
}

public class OrderOfInit {
    public static void main(String[] args) {
        Son h = new Son(); }
}

```

OrderOfInit (3) x

```

"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:D:\Pro
window 1
window 2
window 3
House
f()
father
son

```

3.5 TestInheritance on P61 P65 P66

1. 输出 DrawRectangle, Drawshape.
2. 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。如果没有调用 super 那么就会输出两行 DrawRectangle。
3. 这种方法可以让我们重写同名函数的同时还可以调用父类的同名函数，而不会引起函数名冲突，或者是需要把父类对应函数的代码块粘贴过来等等这些不必要的麻烦。

```

public class TestInheritance{
    public static void main(String[] args){
        Rectangle rect=new Rectangle();
        rect.newDraw(); }
}

class Shape{
    public void draw(){ System.out.println("Draw shape"); }
}

class Rectangle extends Shape{
    public void draw(){ System.out.println("Draw Rectangle"); }
    public void newDraw(){
        draw(); super.draw(); }
}

```

TestInheritance x

```

"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:D:\Pro
Draw Rectangle
Draw shape

```

4. 当父类的构造方法被重写以后，即父类不使用默认构造方法而是自己编写了构造方法，那么在子类的构造方法中，必须要调用一下父类的构造方法。这是因为 java 要在子类使用时保证父类的正确性。在子类初始化之前，先调用父类的构造方法，从而保证父类中的成员变量被正确的初始化，从而保证子类也能正常工作。

```
public class ConstructSubObj{
    public static void main(String[] args){
        Undergraduate ug=new Undergraduate( id: 12345678);
    }
}
class Person{
    Person() { System.out.println("Person"); }
}
class Student extends Person{
    Student(int id) { System.out.println("Student "+id); }
}
class Undergraduate extends Student{
    Undergraduate(int id) {
        super(id); //必须使用，因为student没有默认构造方法
        System.out.println("Undergraduate");
    }
}
```

ConstructSubObj x

"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:D:\Prog
Person
Student 12345678
Undergraduate

5. 初始化的顺序问题在 3.4 中已经有讨论，P66 按照 3.4 中讨论的初始化顺序分析。就很容易得到这个结果。

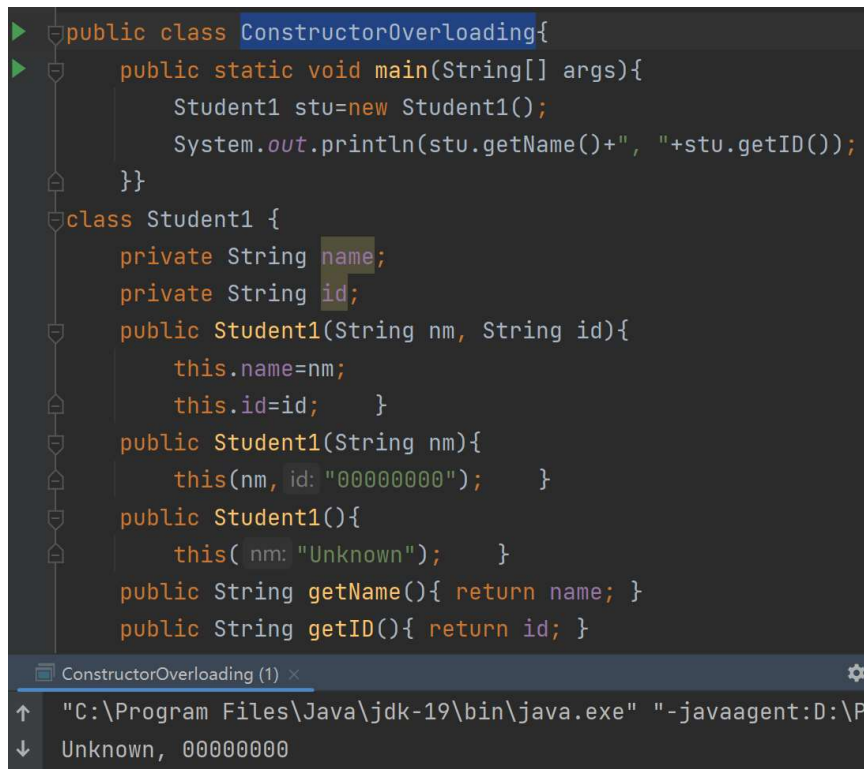
```
class Meal { Meal() { System.out.println("Meal()"); } }
class Bread { Bread() { System.out.println("Bread()"); } }
class Cheese { Cheese() { System.out.println("Cheese()"); } }
class Lettuce { Lettuce() { System.out.println("Lettuce()"); } }
class Lunch extends Meal {
    Lunch() { System.out.println("Lunch()"); }
}
class PortableLunch extends Lunch {
    PortableLunch() { System.out.println("PortableLunch()"); }
}
public class Sandwich extends PortableLunch {
    private Bread b = new Bread();
    private Cheese c = new Cheese();
    private Lettuce l = new Lettuce();
    public Sandwich() { System.out.println("Sandwich()"); }
    public static void main(String[] args) {
```

Sandwich x

"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:D:\Prog
Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
Sandwich()

3.6 ConstructorOverloading on P72 P73

1. 当一个类中出现了很多个名字相同，权限相同，返回值相同只有参数不同的函数时。当我们在调用这个函数时，Java 会帮我们自动选择到对应参数正确的那个函数。
2. 这种方法其实变相地实现了向一个函数中传入不同类型的参数，虽然不是真的动态类型，但是用户在调用时可以减少很多不必要的麻烦。这种方式相比较于 C 更加灵活；相比较于 Python 那种真正的动态类型，又具有更好的安全性和更高的效率。
3. P73 说明了这种机制定义的函数可以被继承，在子类中也可以正常调用。



```

public class ConstructorOverloading{
    public static void main(String[] args){
        Student1 stu=new Student1();
        System.out.println(stu.getName()+" "+stu.getID());
    }
}
class Student1 {
    private String name;
    private String id;
    public Student1(String nm, String id){
        this.name=nm;
        this.id=id;    }
    public Student1(String nm){
        this(nm, id: "00000000");    }
    public Student1(){
        this( nm: "Unknown");    }
    public String getName(){ return name; }
    public String getID(){ return id; }
}

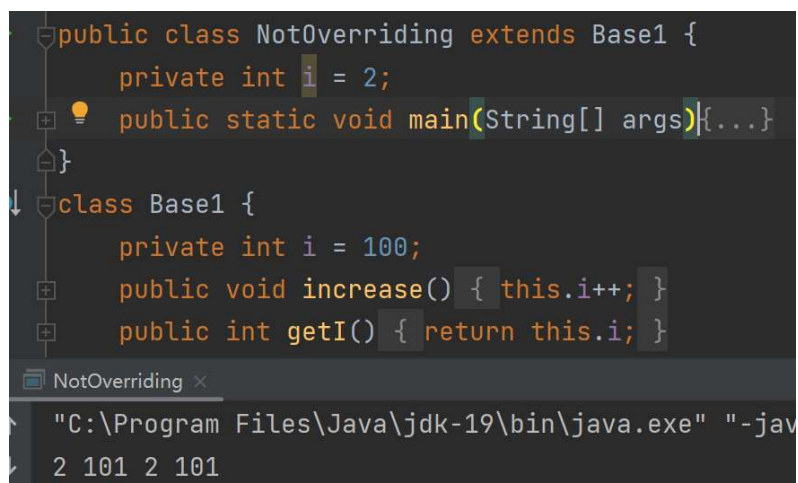
```

ConstructorOverloading (1) ×

↑ "C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:D:\P
 ↓ Unknown, 00000000

3.7 Overriding on P77 P78

1. 子类和父类的同类型同名变量是两套完全不同的内容。父类的变量虽然被覆盖，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，同样父类也不会影响子类。



```

public class NotOverriding extends Base1 {
    private int i = 2;
    public static void main(String[] args){...}
}
class Base1 {
    private int i = 100;
    public void increase() { this.i++; }
    public int getI() { return this.i; }
}

```

NotOverriding ×

"C:\Program Files\Java\jdk-19\bin\java.exe" "-jav
 2 101 2 101

2. 如果在外部，通过向上转型，把子类对象转换成父类对象同样可以访问到父类对应的成员变量，这个例子就在我们之前的作业题里面。
3.
 1. 在子类中可以重写父类中的方法。父类中的 `private` 方法只有在父类中才能访问，子类即使继承父类也不能访问 `private` 方法，所以在子类中定义的同名的 `public` 方法或是 `default` 方法都是与父类中完全不同的两个方法，换句话说，父类中的 `private` 方法，无法被子类重写。而子类在重写父类中的 `default` 或者 `protect` 或者 `public` 类型的方法时不能拥有比父类更严格的访问控制权限。
 2. Java 的父类和子类可以相互转化。子类可以随意的向上转型为父类的对象，但父类向下转行为子类的对象时可能会受限。
4. `Base b = new Child()` 先构造了一个 `Child` 对象，这个对象中包含了三个方法，第 1 个方法是 `base` 中以 `private` 修饰 `func1`（虽然我们没有限制访问但是它是存在的）、第 2 个方法是 `child` 类中以 `public` 修饰的 `func1`、第 3 个方法是 `child` 的类中以 `public` 修饰的 `function2`。并向上转型为 `Base` 对象，此时 `Base` 类中以 `private` 修饰的 `func1` 被显现（可以调用），而 `child` 的类中的 `func1` 被隐藏（不可调用）。时但这只是一个标识的改变，这三个函数的储存方式并没有发生变化。所以调用 `b.func2` 时还可以正常工作。
5. `(Child)b` 这个操作把 `b` 转化回了 `child` 的对象。那么此时作为一个正常的 `child` 的对象，`Base` 类的 `private` 修饰的 `func1` 被隐藏，而 `Base` 类以 `public` 定义的 `func2` 被重写。所以都是调用的 `Child` 中的方法。

(P78 和这个 `Child`，`Base` 类似，感觉这个例子更好一点，P78 页的不再展示)

```

class Child extends Base {
    public void func1() { System.out.println("Child func1 pr
    public void func2() { System.out.println("Child func2 pr
}

public class Base {
    private void func1() { System.out.println("Base func1 pr
    public void func2() { System.out.println("Base func2 pri
    public static void main(String[] args){
        Base b = new Child();
        b.func1();
        b.func2();
        ((Child)b).func1();
        ((Child)b).func2();
    }
}

```

Child x

```

"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:D:\Pro
Base func1 print.
Child func2 print.
Child func1 print.
Child func2 print.

```

3.8 PolyConstruct on P86

1. 这个根据结果来看方法重写是在父类对象构造之前的。子类调用了父类的构造方法，父类构造又调用了 draw 方法，根据实验结果，父类会调用到子类的 draw 方法。
2. 额...说实话有点奇怪，这和 3.4 中得到的结论有点不同.....我暂且把这个理解为，出现同名方法就会执行子类的。但按照我的理解，在这个时候子类的还没有构建完成，子类的 draw 方法应该根本不存在。也许是成员变量和方法的构造流程不一样？还是我对 3.4 的理解有错误... (.....请老师指点一下.....)

```
class RoundGlyph extends Glyph {
    private int radius = 1;
    RoundGlyph(int r) {
        radius = r;
        System.out.println("RG.RoundGlyph(), radius = " + radius);
    }
    void draw() { System.out.println("RG.draw(), radius = " + radius); }
}

public class PolyConstruct {
    public static void main(String[] args) {
        new RoundGlyph(5);
    }
}
```

PolyConstruct x

```
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:D:\Progr
Glyph() before draw()
RG.draw(), radius = 0
Glyph() after draw()
RG.RoundGlyph(), radius = 5
```

3.9 RTTI on P89

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，毕竟这个方法在这个实例中根本就不存在。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。

```
public class RTTI {
    public static void main(String[] args) {
        Useful[] x = { new Useful(), new MoreUseful() };
        x[0].f();
        x[1].g();
        // Compile time: method not found
        x[1].u();
        ((MoreUseful) x[1]).u(); // Downcast
        ((MoreUseful) x[0]).u(); // 抛出异常
    }
}

class Useful {...}
class MoreUseful extends Useful {...}
public class RTTI {
    public static void main(String[] args) {
        Useful[] x = { new Useful(), new MoreUseful() };
        x[0].f();
        x[1].g();
        // Compile time: method not found in Useful:
        // ! x[1].u();
        ((MoreUseful) x[1]).u(); // Downcast/RTTI
        ((MoreUseful) x[0]).u(); // 抛出异常
    }
}
```

RTTI x

J IDEA Community Edition 2022.3.2\lib\idea_rt.jar=54536:D:\
er4_demo.Useful cannot be cast to class Chapter4_demo.MoreUseful

4. 模拟饲养宠物程序

4.1 完整代码

```
1. // 所有宠物的父类
2. class Pet {
3. // 补充定义宠物属性，至少包括宠物的名字、宠物的食物偏好索引号、宠物的食
4. // 量，三个成员变量。成员变量访问权限必须定义为私有权限
5.     private String Name;
6.     private int AmountNeedEat;
7.     private int FavoriteIndex;
8. // 初始化默认的宠物实例对象
9.     public Pet() {
10.         Name = "MyPet";
11.         AmountNeedEat = 1;
12.         FavoriteIndex = 1;
13.     }
14. // 补充定义根据输入参数，初始化宠物的实例对象
15.     public Pet(String Name, int AmountNeedEat, int FavoriteIndex) {
16.         this.Name = Name;
17.         this.AmountNeedEat = AmountNeedEat;
18.         this.FavoriteIndex = FavoriteIndex;
19.     }
20. // 补充定义返回宠物偏好的食物索引号
21.     public int getFavoriteIndex() {
22.         return this.FavoriteIndex;
23.     }
24. // 补充定义返回宠物的名字
25.     public String toString() {
26.         return this.Name;
27.     }
28. // 补充定义宠物进食。投喂的食物符合偏好，就吃，且饱食度减一，返回已进食；
29. // 投喂不符合偏好，或饱食度为零时，拒食，返回未进食
30.     public boolean eat(int foodindex) {
31.         if(foodindex!=this.FavoriteIndex || this.AmountNeedEat==0){
32.             return false;
33.         } else {
34.             this.AmountNeedEat--;
35.             return true;
36.         }
37.     }
38. // 补充定义宠物饥饿判断。饱食度为零返回不饿，否则返回饿
39.     public boolean isHungry(){
40.         return this.AmountNeedEat!=0;
41.     }
42. // 宠物根据自己的状态可以干点什么了。
```

```
43.     public String doSomething() { return "doSomething";}
44. }
45. // 宠物狗类
46. class Dog extends Pet {
47.     // 补充定义需要的成员变量。成员变量的访问权限必须定义为私有权限
48.     private String Color="BLACK";
49.     // 根据输入参数，初始化狗的实例对象
50.     public Dog(String Name, int AmountNeedEat, int FavoriteIndex) {
51.         super(Name,AmountNeedEat,FavoriteIndex);
52.     }
53.     public Dog(String Name, int AmountNeedEat, int FavoriteIndex, String Color) {
54.         super(Name,AmountNeedEat,FavoriteIndex);
55.         this.Color = Color;
56.     }
57.     // 补充定义宠物狗的干事方法。如果没吃饱，返回 still want to eat more food.
58.     // 如果吃饱了，返回 Let's go outside for a wallow.
59.     public String doSomething() {
60.         if(isHungry()) return "still want to eat more food.";
61.         else return "Let's go outside for a wallow.";
62.     }
63. }
64. // 宠物猫类
65. class Cat extends Pet {
66.     // 补充定义需要的成员变量。成员变量的访问权限必须定义为私有权限
67.     private String Color="BLACK";
68.     // 补充定义根据输入参数，初始化猫的实例对象
69.     public Cat(String Name, int AmountNeedEat, int FavoriteIndex) {
70.         super(Name,AmountNeedEat,FavoriteIndex);
71.     }
72.     public Cat(String Name, int AmountNeedEat, int FavoriteIndex, String Color) {
73.         super(Name,AmountNeedEat,FavoriteIndex);
74.         this.Color = Color;
75.     }
76.     // 补充定义宠物猫的干事方法。如果没吃饱，返回 still want to eat more food.
77.     // 如果吃饱了，返回 Let's go to bed for a sleep.
78.     public String doSomething() {
79.         if(isHungry()) return "still want to eat more food.";
80.         else return "Let's go to bed for a sleep.";
81.     }
82. }
83. // 宠物饲养程序
84. public class PetRaise {
85.     public static void main(String[] args) {
86.         // 随机购买了 10 种食物，食物偏好索引为 1,2
87.         int food[] = new int[10];
88.         for (int i = 0;i < 10;i ++)
```

```
89.         food[i] = (int)(Math.random()*2) + 1;
90.         // 补充程序：随机领养了两只宠物
91.         // 第一个宠物的名字：Tom；食物偏好 1；饭量：3
92.         // 第二个宠物的名字：Pluto；食物偏好 2；饭量：6
93.         Pet[] pet = {new Pet(),new Pet()};
94.         if((int)(Math.random()*2) + 1==1){
95.             pet[0] = new Cat("Tom",3,1);
96.         } else { pet[0] = new Dog("Tom",3,1);}
97.         if((int)(Math.random()*2) + 1==1){
98.             pet[1] = new Cat("Tom",6,2);
99.         } else { pet[1] = new Dog("Tom",6,2);}
100.        // 补充程序：给领养的两个宠物喂食，直到宠物吃饱或无合适食物
101.        int food_one_number = 0;
102.        int food_two_number = 0;
103.        for(int i:food){ if(i==1) food_one_number++; else food_two_number++;}
104.
105.        while (pet[0].isHungry() && food_one_number>0){
106.            pet[0].eat(1);
107.            food_one_number--;
108.        }
109.        while (pet[1].isHungry() && food_two_number>0){
110.            pet[1].eat(2);
111.            food_two_number--;
112.        }
113.        // 补充程序：如果有某个宠物未吃饱，则输出是哪种食物不足
114.        for (int i = 0;i < 2;i ++){
115.            if(pet[i].isHungry()){
116.                System.out.println("Food "+pet[i].getFavoriteIndex()+" is not enough.");
117.            }
118.        }
119.        // 补充程序：使用增强 for 循环带每个宠物活动（doSomething）
120.        for(Pet i:pet){
121.            System.out.println(i.doSomething());
122.        }
123.    }
124. }
```

4.2 回答问题

1. 设计 Pet 类的目的是什么？Pet 类所有宠物的父类，它具有所有 Pet 共有的属性。在设计其他宠物类或者修改共有属性时更方便，不容易出错。
2. 以程序为例说明多态的作用？同名函数可以同时存在，同名函数存在时调用子类的函数，参数不同时 Java 自动帮你选择适合的函数。这个程序中，如果子类不定义 doSomething 程序也能正常运行，它会执行父类的方法。而子类中定义了 doSomething 就会自动调用子类的方法。提高了代码的可维护性，提高了代码的拓展能力。

5. 编写复数类

5.1 成员变量

复数由实部和虚部组成，这里定义两个成员变量和构造函数。

```
// Construct
private double real=0;
private double imag=0;
public Complex(double real, double imag){
    this.real = real;
    this.imag = imag;
}
```

5.2 加减乘除运算

为了实现任何数字类型同 Complex 类型的运算，函数输入时自动向上转型为 Number 对象。在函数内部通过 instanceof 来判断类型进而进行相应的操作，这样就实现了一个函数传入所有的数字类型参数。（也可以通过多态实现，但是那样算下来...排列组合...每一种运算都要写 $6*6=36$ 个函数，过于复杂...）可惜的是 Java 不支持运算符重载，所以在进行复数运算时只能通过函数进行。（注：此处的 equals 不是 override 的 equals）

```
// add
private static Complex complexAddcomplex(Complex _complex1, Complex _complex2){...}
public static Complex add(Number x1, Number x2){...}
// sub
private static Complex complexSubcomplex(Complex _complex1, Complex _complex2){...}
public static Complex sub(Number x1, Number x2){...}
// multiply
private static Complex complexMultiplycomplex(Complex _complex1, Complex _complex2){...}
public static Complex multiply(Number x1, Number x2){...}
// divide
private static Complex complexDividecomplex(Complex _complex1, Complex _complex2){...}
public static Complex divide(Number x1, Number x2){...}
// compare
private static boolean complexEqualscomplex(Complex _complex1, Complex _complex2){...}
private boolean equals(Number _x) {...}
public static boolean equals(Number x1, Number x2){...}
```

5.3 重写三个方法

三个方法的重写。hashCode 我上网查了一些资料，其实就是要让不同值的 hashCode 尽可能得不同，因此我采用了求数组地 hashCode 的方法处理这个问题。很简单直接，如代码所示。Equals 方法同加减乘除类似，可以传入任何类型的数字与 Complex 类型作比较，按照值返回是否相等。例：(1+0i == 1.0) -> true

```
// override
@Override
public int hashCode() {
    double temp = this.real*31 + this.imag;
    return Double.hashCode(temp);
}

@Override
public boolean equals(Object obj) {
    if(obj instanceof Complex) return equals((Complex)obj);
    else return Complex.equals(this,(Number)obj);
}

@Override
public String toString() {
    String str;
    if (this.imag<0) str = String.format("%.2f", this.real)+String.format("%.2f", this.imag)+"i";
    else str = String.format("%.2f", this.real)+"+"+String.format("%.2f", this.imag)+"i";
    return str;
}
```

5.4 重写其他方法

由于继承自抽象类 Number 所以这几个函数必须重写

```
// Force conversion
public int intValue() throws NumberFormatException{...}
public long longValue() throws NumberFormatException{...}
public float floatValue() throws NumberFormatException{...}
public double doubleValue() throws NumberFormatException{...}
public byte byteValue() throws NumberFormatException{...}
public short shortValue() throws NumberFormatException{...}
```

5.5 其他便捷组件

提供一系列方法，返回复数的其他特征，例如模长旋转角度等。

提供了一个 copy 方法实现深拷贝。

提供一个 show 方法实现快捷打印，（其实没有必要，正常应该使用 toString 再传到 println）

```
// Get
public Complex getConjugate() { return new Complex(this.real, -this.imag); }
public double getModulus() { return this.real*this.real + this.imag*this.imag; }
public double getTheta() { return Math.atan(this.imag/this.real); }
public double getReal() { return real; }
public double getImag() { return imag; }

// Other tools
private static boolean double_equal(double x,double y){...}
public Complex copy(){
    return new Complex(this.real, this.imag);
}

public void show(String ... end){...}
```


5.6 其他说明

这里把加减乘除的运算，全部定义成了静态方法，传入两个数，然后返回另一个 Complex 对象。这是因为根据我们的使用习惯，我们会希望 $c=a+b$ 时不要改变 a 和 b 的值，如果是 $a.add(b)$ ，这种写法，虽然也可以实现 $a+b$ 但这相当于是 $a+=b$ 的写法，如果我不希望改变 a 的值，还需要把 a 深拷贝一份，怪麻烦的。当然，也可以 $a.add(b)$ 返回一个新的 Complex 对象而不改变 a 的值，但是这样的结构总觉得显得不太对称，给人一种 $a+b$ 和 $b+a$ 会得到不同结果的感觉，所以随后采用了 $Complex.add(a,b)$ 这种结构。

5.7 测试类和结果展示

我知道这是一个很潦草的测试类，但是...实在懒得写了...

它简单的输出了两个复数的加减乘除运算和他们本身。

```
public class Test {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        do {
            System.out.print("Input A by real imag: ");
            Complex A = new Complex(s.nextDouble(), s.nextDouble());
            System.out.print("Input B by real imag: ");
            Complex B = new Complex(s.nextDouble(), s.nextDouble());
            System.out.println("Add: " + Complex.add(A, B));
            System.out.println("Sub: " + Complex.sub(A, B));
            System.out.println("Multiply: " + Complex.multiply(A, B));
            System.out.println("Divide: " + Complex.divide(A, B));
            System.out.print("A: " + A + "; ");
            System.out.println("B: " + B);
            System.out.print("Input 0 to exit or anything else to continue: ");
        } while (s.nextInt() != 0);
        s.close();
    }
}
```

```
"C:\Program Files\Java\jdk-19\bin\java.exe" "-java
Input A by real imag: 10 10
Input B by real imag: 20 20
Add: 30.00+30.00i
Sub: -10.00-10.00i
Multiply: 0.00+400.00i
Divide: 0.50+0.00i
A: 10.00+10.00i; B: 20.00+20.00i
Input 0 to exit or anything else to continue:
```

5.8 任务总结

这个复数的任务，对 Java 中面向对象的机制等都有了更深入的理解。其实我尝试过好几种实现方法，尝试传入不同的数据类型加深了我对向上转型还有对象之间的强制类型转化也有了更多理解。当函数多了以后，慢慢意识到了严格的权限带来的好处。对于异常处理机制也有了些许了解。总之就是受益匪浅。

6. 总结

要学的东西还很多，要走的路还很长；
革命尚未成功，同志仍须努力。