

# 写代码的终极思路：

---

- 1, 场景化需求分析
- 2, 准确逻辑化需求分析
- 3, 寻找解决模型——设计模式
- 4, 编写调试, 重复1-4
- 5, 功能测试和性能测试

## 代码学习方法

---

- 1, 整体认知：要知道效果
- 2, 跟着源代码敲一遍, 解决报错问题 (百度定性, 定范围|帮助文档查查查)
- 3, 一行一行地分析代码, 在看代码的同时, 需要理解里面出现的所有你现在还不能理解的东西, 理解过程 (百度定性, 定范围——5篇博客之内|帮助文档查查查), 必须加入你自己的骚操作, 摆弄摆弄确保自己懂了这个道理—— (首先达标源代码的代码水平, 加入自己的尝试)
- 4, 不要去管有多少还有学, 有多少已经学了, 有时间, 就做研究——无我地进行

## OOP 核心上 (Object-Oriented Programming)

---

- 1, 对象变量——引用
- 2, 方法定义, 方法传参, 方法重载, 构造函数 (初始化)
- 3, this, static
- 4, 访问修饰符与包的隔离 (public, private, protected, default)
- 5, 成员变量, 成员方法
- 6, 枚举类型

## 对象变量——引用

---

```
1 public class Demo01 {
2     //需求: 大家都有女朋友, 但是生活上有这样的几种情况
3     //女朋友外号——baby    honey=new GF();
4     //女朋友的男朋友    BF a=new BF(1);
5     //a=new BF(2);
6
7     //多个变量一个对象, 多个对象与一个变量
8     public static void main(String[] args) {
9         BF baby=new BF(); //堆空间中唯一开辟的BF空间
10        BF honey=baby;
11
12        GF 老婆=new GF(); //路人甲    gc    坏叔叔收割new GF()
```

```

13      老婆=new GF(); //只有出现一次new，意味着堆空间中开辟了一个空间
14      老婆=new GF();
15      老婆=new GF();
16      老婆=new GF();
17      老婆=new GF();
18  }
19  }
20  //女朋友
21  class GF{
22
23  }
24  //男朋友
25  class BF{
26
27  }
28

```

## 引用与指向——对象变量与堆空间开辟的独立对象空间

`new GirlFriend();`

代表整出一个模型生产出来的对象。

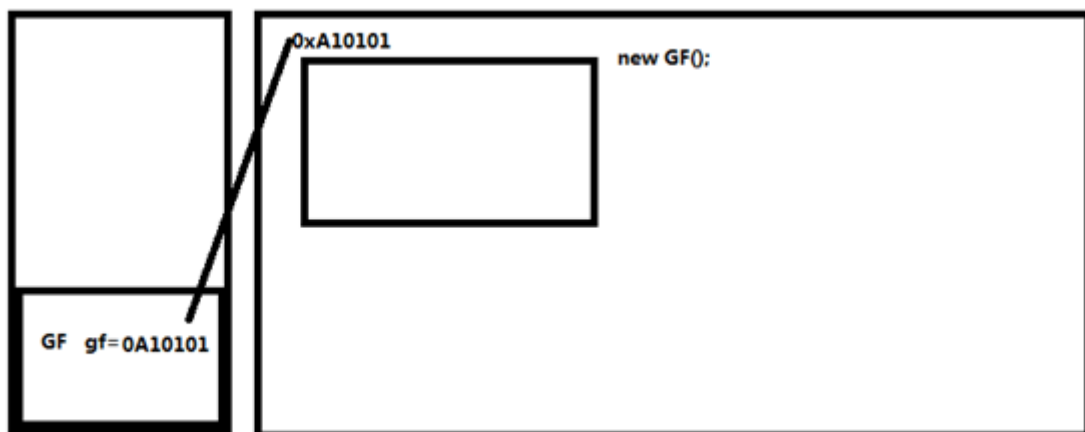
意味着在堆空间中开辟了独立的运行（内存）空间，没有办法在方法中找到这个独立空间的起始地址。

为了访问到他，我们需要给一个名字变量（引用-把目标引过来用他，指针）

`GirlFriend gf=new GirlFriend();`

Gf就是GirlFriend的一个对象，引用，指向堆中空间

= 等号就是指向，代表这个堆中对象的起始地址



一个对象模型

```
class GirlFriend{
```

稍微有点结构的模型

```

1  class GirlFriend{
2      String name;
3      String height;
4      char sex;

```

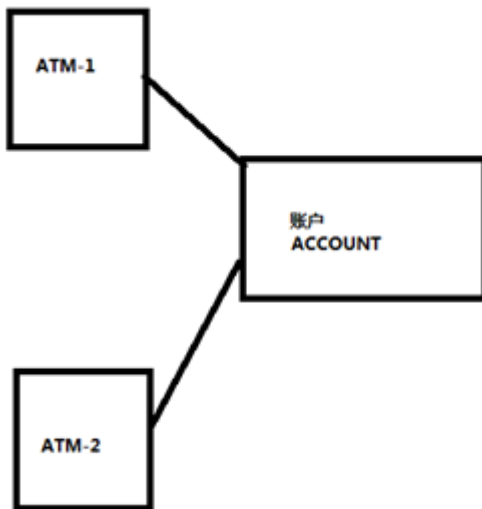
```

5
6     void drink() {
7
8     }
9     void eat() {
10
11    }
12    void bit() {
13
14    }
15 }
16

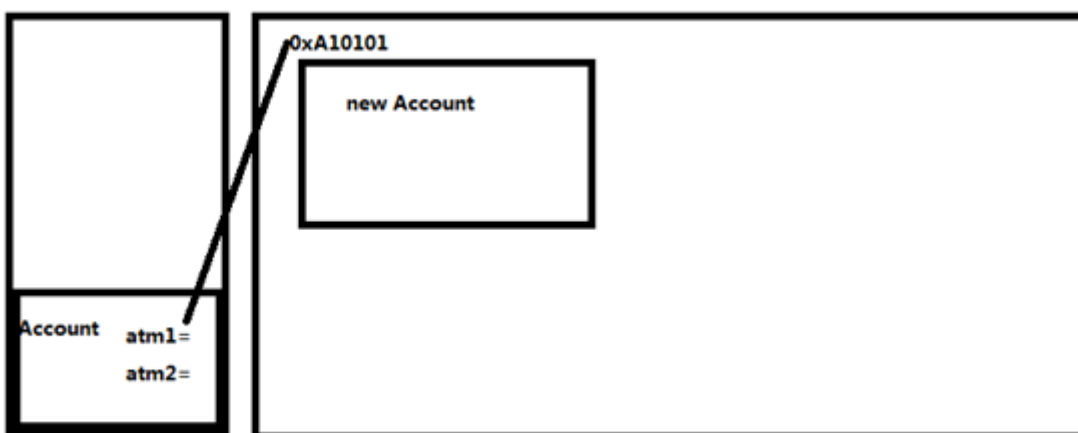
```

总结：无论是简单还是复杂结构的模型，都称为一个类

## 多个引用，一个对象



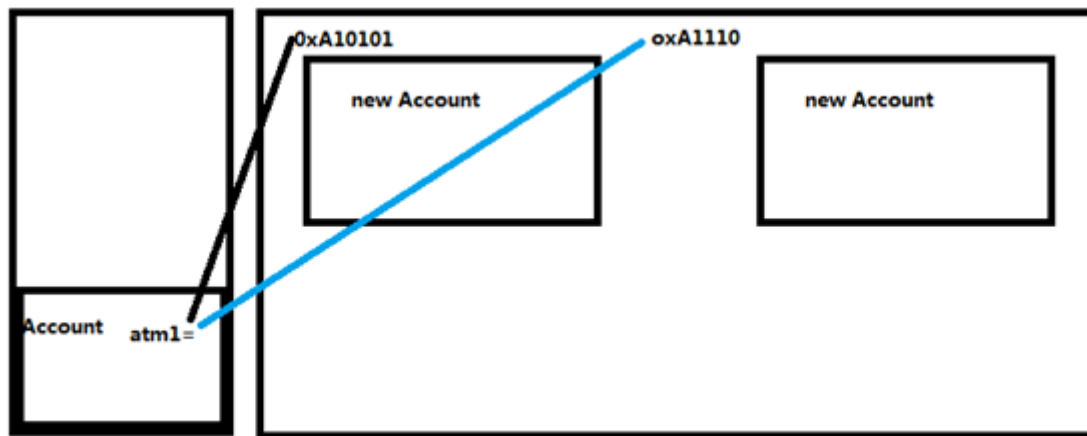
堆空间中的一个开辟空间，可以被多个引用指



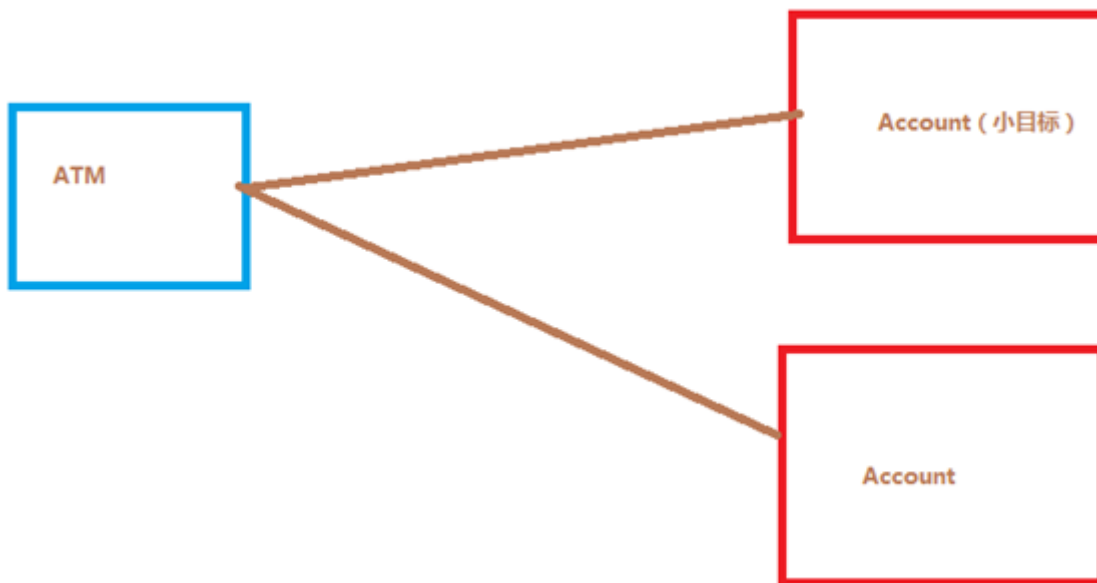
通俗解释：

在多个ATM机上取钱，实际上是操作的同一个账户，atm机就等于不同的引用，而那个被共同指向的账户就等于account对象

## 一个引用多个对象



```
1 Account atm1=new Account();
2     atm1=new Account();//前面那个空间没有了指向，//成为了无名之地
3 }
4 }
5 class Account{
6 }
7
```



## 练习题

请大家创建两个案例，分别体现多个引用一个对象和一个引用多个对象

## 方法定义，方法重载，构造函数

方法-method：一个模型的功能，一个对象的行为，一个function（函数）

## 方法定义：

### 返回值

- 1, 无返回值 void——函数内部从上到下执行逻辑
- 2, 有返回值

```
1 //返回基本类型
2 int getAge() {
3     return 18;
4 }
5 boolean isPerson() {
6     return false;
7 }
8 //造人方法-返回对象类型
9 Person create(Person p){
10     return new Person(); //返回一个新person
11 }
```

a) 返回基本数据类型

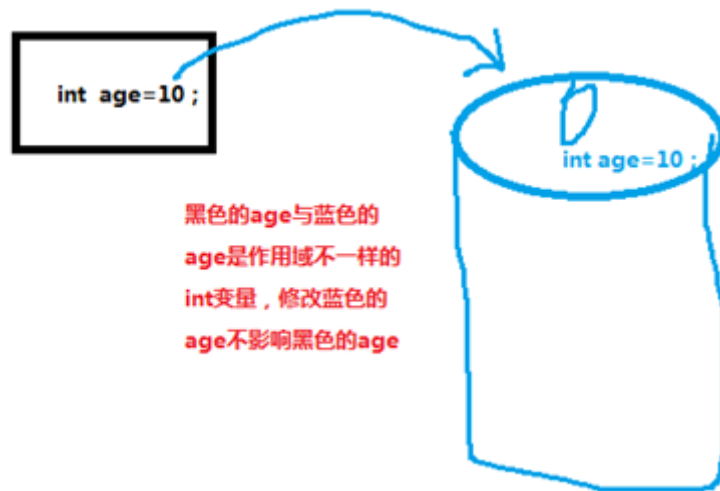
b) 返回对象类型

### 方法传参

#### 基本类型传参

```
1
2 public static void main(String[] args) {
3
4     int age=10;
5     m4(age);
6     System.out.println(age);
7
8 }
9 //研究传入的基本类型变量的特性
10 static void m4(int age) { //对于基本类型来说，传入的变量会被copy-复制 一份
11     age+=10;
12     System.out.println(age);
13 }
14
15 //传任意基本类型的参数变量 和 任意的传入顺序
16 void m3(boolean isPerson,int age,float account) {
17
18 }
19 //传任意基本类型的参数变量
20 void m2(int age,boolean isPerson,float account) {
21
22 }
23 //传基本数据类型
24 void m1(int age){
25
26 }
27 //无参数
28 void m() {
29
```

对于方法传参，如果是基本类型参数，会copy一份，在函数内部是对copy后的那一份进行的相关操作，这些操作不影响传入之前参数那个本值



### 对象类型传参

对象类型 传参的基本细节跟基本类型传参的细节相同，对于参数copy来说有一些变化

```

1  class BoyF{
2      int age=18;
3  }
4  public class Demo03 {
5      //跟男朋友gank了一下
6      static void gank(BoyF bf) {
7          bf.age=3; //很开心的样子。仿佛3岁一样
8      }
9
10
11     public static void main(String[] args) {
12
13         BoyF bf=new BoyF();
14         gank(bf);
15         System.out.println(bf.age); //只要被男朋友gank了，永久性伤害
16     }

```



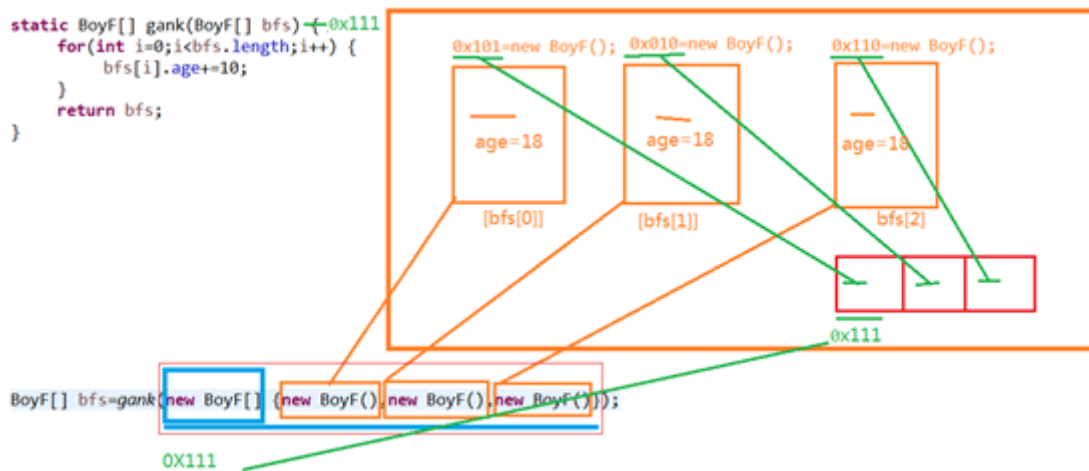
结论：方法传入的是类型变量，则传入的是堆中的对象空间首地址

## 引申一下：数组对象作为方法参数传递对象

注意，对象数组和数组对象是有区别滴

对象数组：以对象为元素，创建一个多对象的数组结构容器

数组对象：把数组当做一个对象



```
1 //gank了一队表哥
2 static BoyF[] gank(BoyF[] bfs) {
3     for(int i=0;i<bfs.length;i++) {
4         bfs[i].age+=10;
5     }
6     return bfs;
7 }
8
9 public static void main(String[] args) {
10
11     // BoyF bf=new BoyF();
12     // gank(bf);
13     // System.out.println(bf.age); //只要被男朋友gank了，永久性伤害
14     BoyF[] bfs=gank(new BoyF[] {new BoyF(),new BoyF(),new BoyF()});
15     for(BoyF boy:bfs) {
16         System.out.println(boy.age);
17     }
18 }
```

## 方法重载

```
1 void drink(Milk milk) {
2     System.out.println("牛奶");
3 }
4 void drink(int i) { //1
5     System.out.println("咪一口");
6 }
7 //先ao一下，在咪一口，在吸一口牛奶，最后滋溜一下
8 void drink(Ao ao,int i,Milk milk,Object obj) {
9     System.out.println("吸一口酒");
10 }
11 //先ao一下，在咪一口，在吸一口牛奶，最后滋溜一下
12 void drink() {
13     drink(null); //
14     drink(1);
```

```

15     drink(null,1,null,null);
16 }
17

```

函数重载：函数名相同，只要参数个数不同，参数顺序不同，则为不同的函数

**可变参数示例：**

```

1     void drink() {
2         drink(1,1,1,1,1,1,1,2,1,1,1,1,1); //int[] a=
        {1,1,1,1,1,1,1,2,1,1,1,1,1}
3         drink(new int[] {1,2,3,4,5},5);
4     }
5     void drink(char a,int i) {
6
7     }
8     //可变参数
9     void drink(int... i) { //与(int[] i)相似
10        System.out.println("1");
11    }
12    void drink(int[] as,int a) {
13        System.out.println("2");
14    }
15    void drink(int a,int b,int c) {
16
17    }
18

```

可变参数实际上就是可以自定义传入函数参数的个数，会自动封装一个数组来接受例如：

```

    drink(1,1,1,1);==new int[]{1,1,1,1}

```

注意，可变参数不仅仅可以使用到基本数据类型，还可以用于对象类型

## 练习题

设计一个英雄类（奶妈）（姓名，hp，提示一下自己被加血量），奶妈补奶方法——功能

奶妈的技能：

对一个英雄补奶——奶妈自己掉血——hp减少

大补——对一群英雄补奶——终极技能r

买一个救赎，装备指定地域补奶

增加一个技能，治疗群体补奶

天赋补奶——吸血天赋对自己补奶

备注：补奶等于加血 hp，对于所有的补奶（加血）方法有不同的hp的值被加

```

1     package com.haoyu;
2
3     public class Demo05 {
4         //模块化

```



```

5      static Hero[] initHeros() {
6          Hero gailun=new Hero("盖伦");
7          Hero vn=new Hero("vn");
8          Hero yasuo=new Hero("压缩");
9          Hero[] heros= {gailun,vn,yasuo};
10         return heros;
11     }
12
13     public static void main(String[] args) {
14
15         new MilkMotherHero().hpAdd(0,0,100,100,initHeros());
16
17         //我自己--奶妈
18         //Hero naima=new MilkMotherHero();//多态
19         //MilkMotherHero naima=new MilkMotherHero();
20         //
21         //使用救赎
22         //naima.hpAdd(0,0,100,100,heros);
23
24     }
25
26 }
27
28 class Hero{
29     int x=0;
30     int y=0;
31
32     Hero(){ }
33     Hero(String name){
34         this.name=name;
35     }
36     String name;
37     int hp;//0
38     //加血提示方法
39     void hpAddedHigh() {
40         System.out.println("加血让我舒服。好嗨哟! "+this);
41     }
42
43     public String toString() {
44         return "Hero [name=" + name + ", hp=" + hp + "]";
45     }
46 }
47
48
49 class EquipMement{
50     int hpp=1000;
51     String name;
52     //希望装备被new出来的同时，就给name赋值
53     //像这样没有返回值，并且方法名称与类名 完全相同（所有字母内容相同，大小写相同）
54     //称为构造方法--当一个对象在初始化的时候，首先调用的就是构造方法，用来做初始化操作
55     EquipMement() { //如果这个不带参数的构造函数没有写在代码中，又没有其他的构造函数，
系统会默认地生成一个无参数的构造函数
56
57     }
58     EquipMement(String name) { //如果出现了带参数的狗杂方法，但是又米有无参数的构造方法，默认只存在这个有参数的构造方法
59
60         //原来那个无参数的构造方法失效
61         //对于jiushu=new EquipMement()这个对象来说

```

```

61         //this=jiushu;
62         this.name=name;
63     }
64
65     //装备救赎的加血功能
66     void hpAdd(int x,int y,int width,int height,Hero... heros) {
67         for(int i=0;i<heros.length;i++) {
68             heros[i].hp+=hpp;
69             heros[i].hpAddedHigh();
70         }
71     }
72 }
73
74 //奶妈英雄 extends 继承 (继承遗产)
75 class MilkMotherHero extends Hero{
76     int x=0;
77     int y=0;
78
79     String name="奶妈";
80     int hp=100;
81     int hpp=1;
82     EquipMement[] ems= {new EquipMement("救赎");};//ems[0]=new EquipMement()
83     //加血提示方法
84     void hpAddedHigh() {
85         System.out.println("加血让我舒服。好嗨哟! "+this);
86     }
87     //加血合集
88     //单体加血
89     void hpAdd(Hero hero) {
90         //对方血曾加
91         hero.hp+=hpp;
92         hero.hpAddedHigh();
93         //本身血减少
94         hp-=hpp;
95     }
96     //群体加血
97     void hpAdd(Hero[] heros) {
98         for(int i=0;i<heros.length;i++) {
99             heros[i].hp+=hpp;
100             heros[i].hpAddedHigh();
101         }
102         //本身也要加血
103         hp+=hpp;
104         //自嗨
105         this.hpAddedHigh();
106     }
107     //天赋加血
108     void hpAdd() {
109         hp+=hpp;
110         this.hpAddedHigh();
111     }
112     //使用装备加血
113     void hpAdd(int x,int y,int width,int height,Hero[] heros) {
114         //检查一下是否有救赎装备
115         if(ems!=null) {
116             for(int i=0;i<ems.length;i++) {
117                 if(ems[i].name=="救赎") {

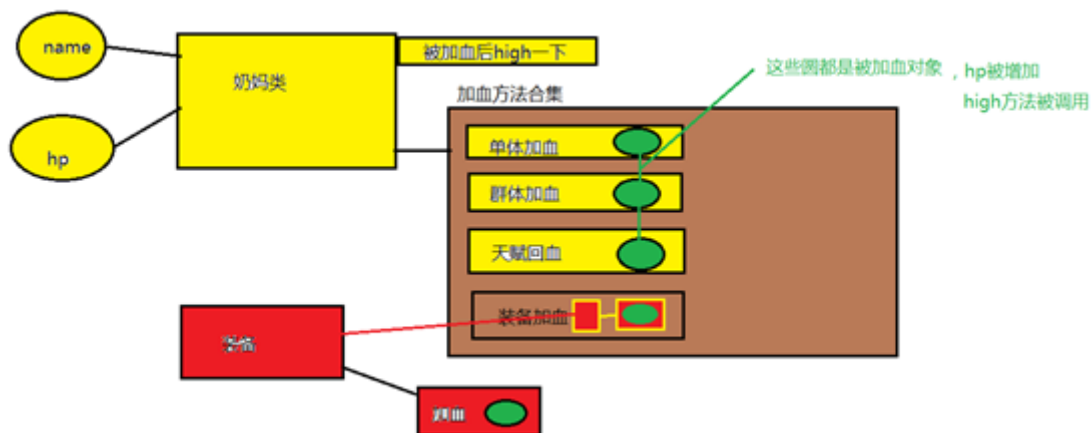
```

```

118 //TODO 以后要注意，这里其实还有一个判断，就是这些英雄是否在这个区
域
119
120 Hero[] herosTemp=new Hero[heros.length+1];
121 for(int index=0;index<herosTemp.length;index++) {
122     if(index<heros.length) {
123         herosTemp[index]=heros[index];
124     }else {
125         //奶妈对自己操作，把自己（我）this 加进去
126         herosTemp[index]=this;
127     }
128 }
129 //执行救赎方法
130 ems[i].hpAdd(x,y,width,height,herosTemp);
131 }
132 }
133 }
134 }
135
136 public string toString() {
137     return "MilkMotherHero [name=" + name + ", hp=" + hp + "]";
138 }
139
140 }
141

```

结构分析图：



## 内存分析TODO

## 构造函数

步骤 1：什么是构造方法

步骤 2：隐式的构造方法

步骤 3：提供一个有参的构造方法

步骤 4：构造方法的重载

步骤 1：

## 什么是构造方法

方法名和类名一样（包括大小写）

没有返回类型

实例化一个对象的时候，必然调用构造方法

注释：

```
1 public class Demo06 {
2
3     public static void main(String[] args) {
4         new Light();
5     }
6
7 }
8 //灯泡 桌子 门 学生 教室 教师
9 class Light{
10     //不管你写不写这个方法，自动生成一个跟类名相同的方法，默认有一个
11     //这个特殊的方法就是构造方法
12     //在创建一个模型对象的时候会被调用
13     Light(){
14         System.out.println("aaa");
15     }
16     //一般方法都有返回值，无返回值的声明
17     void close() {
18
19     }
20
21     void open() {
22
23     }
24 }
25
```

步骤 2：

## 隐式的构造方法

Hero类的构造方法是

```
1 public Hero(){
2
3 }
```

这个无参的构造方法，如果不写，就会默认提供一个

步骤 3：

## 提供一个有参的构造方法

一旦提供了一个有参的构造方法

同时又**没有显式**的提供一个无参的构造方法

那么默认的无参的构造方法，就“木有了”

```

public class Demo06 {


    public static void main(String[] args) {
        //      new Light();
        new Desk();
    }

}
class Desk{
// Desk(){}
//如果定义了一个带参数的构造函数，原来那个默认的空参构造函数就消失了
    Desk(int i){

    }

}
}

```



步骤 4：

## 构造方法的重载

```

public static void main(String[] args) {
    //      new Light();
    new Desk('a',1);
}

class Desk{
    Desk(){}
    //如果定义了一个带参数的构造函数，原来那个默认的空参构造函数就消失了
    Desk(int i){

    }
    Desk(int i,int j){

    }
    Desk(char a,int b){

    }
    Desk(int b,char a){

    }
}

```

```

1  public class Demo06 {
2      public static void main(String[] args) {
3          new Classroom(1);
4          new Classroom("多功能厅");
5          new Classroom(true);
6          new Classroom(100,"里面有钢琴");
7      }
8  }
9  //new 教室出来
10 //console打印以下4句
11 //5-小教室
12 //"多功能厅"-多工厅教室
13 //true-真的是一个教室
14 //100-"里面有钢琴"-音乐教室
15 class Classroom{
16     public Classroom(int i) {
17         System.out.println("小教室");
18     }
19     public Classroom(String name) {
20         System.out.println(name);
21     }
22     public Classroom(boolean flag) {

```

```

23         System.out.println("真的是一个教室 ");
24     }
25     public Classroom(int num,String name) {
26         System.out.println("音乐教室");
27     }
28 }
29

```

步骤 5：

## 练习-构造方法

为装备类设计4个参数的构造方法

这四个参数分别是，装备等级-int，装备合成的子装备-类，装备的价格-double，装备的功能-String

创建装备的时候以上4个属性可以打组合

打印定义时输入的内容

```

1  //装备类
2  class Equipment{
3      //
4      private int level=1;
5      final static private int defaultLevel=1;
6      final static private double defaultPrice=50.0;
7      //
8      private Equipment[] es;
9      //
10     private double price;
11     //e-description
12     private String eDescription;
13     public Equipment() {
14         //super();
15     }
16
17     public Equipment(String eDescription) {
18         this(defaultLevel,null,defaultPrice,eDescription);
19     }
20
21     //TODO ?static -defaultLevel
22     public Equipment(double price, String eDescription) {
23         this(defaultLevel,null,price,eDescription);
24     }
25
26     public Equipment(Equipment[] es, double price, String eDescription) {
27         this(defaultLevel,es,price,eDescription);
28     }
29
30     public Equipment(int level, Equipment[] es, double price, String
eDescription) {
31         this.level = level;
32         this.es = es;
33         this.price = price;
34         this.eDescription = eDescription;
35         System.out.println(this.level+this.eDescription+this.price);
36         print();

```

```

37     }
38
39     public void print() {
40         for(Equipment e:this.es) {
41             System.out.println(e);
42         }
43     }
44
45     public String toString() {
46         return "Equipment [level=" + level + ", price=" + price + ",
eDescription="
47             + eDescription + "]";
48     }
49
50
51
52     public int getLevel() {
53         return level;
54     }
55     public void setLevel(int level) {
56         this.level = level;
57     }
58     public Equipment[] getEs() {
59         return es;
60     }
61     public void setEs(Equipment[] es) {
62         this.es = es;
63     }
64     public double getPrice() {
65         return price;
66     }
67     public void setPrice(double price) {
68         this.price = price;
69     }
70     public String geteDescription() {
71         return eDescription;
72     }
73     public void seteDescription(String eDescription) {
74         this.eDescription = eDescription;
75     }
76 }
77

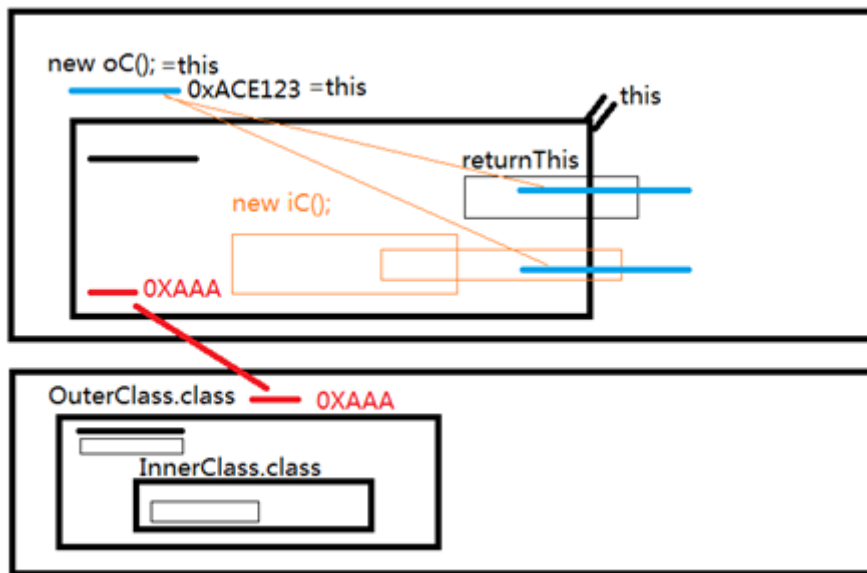
```

## This与static

---

### This

#### 内部类调用外部类中的this原理



```

1      OuterClass oc=new OuterClass();
2      //且认为70dea4e就是对象oc在堆空间的起始位置
3
4      System.out.println(oc.getOuterThis()); //oc=com.haoyu.OuterClass@70dea4e
5      //假设
6      //如果把一个对象放在打印函数里面，对象会直接调用本身的toString方法，把本对象在堆
7      //空间里的地址形成一个字符串
8      //由于这个字符串的@后的十六进制数是唯一的，所以可以暂时认为它就是对象oc在堆空间
9      //的起始位置
10     System.out.println(oc);
11     // oc.print();
12     System.out.println(oc==oc.getOuterThis()); //true
13     //粗鲁理解: this==oc==0x70dea4e
14     System.out.println(oc==oc.getInnerContainsOuterClassThis());
15 }
16
17 class OuterClass extends Object{
18
19     class InnerClass{
20         public OuterClass returnOuterThis() {
21             return OuterClass.this;
22         }
23     }
24
25     public OuterClass getInnerContainsOuterClassThis() {
26         //创建outerclass对象
27         //this=0x70dea4e=在其他地方已经new OuterClass()
28         InnerClass ic=this.new InnerClass();
29         return ic.returnOuterThis();
30     }
31
32     public OuterClass getOuterThis() {
33         return this;
34     }
35     public void print() {
36         System.out.println(this.getClass().getName());
37         System.out.println(Integer.toHexString(hashCode()));
38     }
39 }
40 //native 本地函数

```



```

37     //public native int hashCode();//70dea4e
38     //toString 可以构建一个对象的字符串形式
39     public String toString() {
40         return getClass().getName() + "@" + Integer.toHexString(hashCode());
41     }
42 }
43

```

## This内部类的实例， builder模式

```

1  //this--对象对自己说：“我”
2  //Outer.class 外部类
3  class Teacher{//new Teacher()=0xAAA
4      //Inner.class 内部类
5      //内部类可以随意调用外部类的成员变量的
6      //一旦内部类声明了跟外部类一样的成员变量，方法的时候，覆盖外部类的方法
7      class Builder{//new Teacher().new Builder()=0xBBB
8          public Builder age(int age) {
9              //Teacher.this=0xAAA
10             Teacher.this.age=age;
11             return this;//0xBBB
12         }
13         public Builder subject(String subject) {
14             //Teacher.this=0xAAA
15             Teacher.this.subject=subject;
16             return this;//0xBBB
17         }
18         public Teacher build() {
19             return Teacher.this;//0xAAA
20         }
21     }
22
23     public static Builder builder() {
24         //有点问题
25         return (new Teacher()).new Builder();
26     }
27
28     private int age;//=0
29     private String subject;
30
31     public Teacher() {
32         //调用带参数的构造方法
33         this(1,"哲学");
34     }
35
36     public Teacher(int age, String subject) {
37         //this有跟new出来的对象地址关联起来的功能
38         this.age = age;
39         this.subject = subject;
40     }
41     public Teacher getT() {
42         return this;//0xAAA=new Teacher();
43     }
44
45     public int getAge() {

```

```

46         return age;
47     }
48     public void setAge(int age) {
49         this.age = age;
50     }
51     public String getSubject() {
52         return subject;
53     }
54     public void setSubject(String subject) {
55         this.subject = subject;
56     }
57     @Override
58     public String toString() {
59         return "Teacher [age=" + age + ", subject=" + subject + "]";
60     }
61 }
62
63 Teacher t2=Teacher.builder().age(2).subject("语文").build();//链式表达--内部类
64

```

## This调用构造函数

```

1     public Teacher() {
2         //调用带参数的构造方法
3         this(1,"哲学");
4     }
5
6     public Teacher(int age, String subject) {
7         //this有跟new出来的对象地址关联起来的功能
8         this.age = age;
9         this.subject = subject;
10    }
11

```

## This成员变量传参

```

this.age = age;    this.subject = subject;

```

## 把对象本身作为返回值

```

public Teacher getT() {    return this;//0XAAA=new Teacher(); }

```

## 内部类的使用

## Static

在内存中只存储一份变量，可以类名.成员变量 和 类名.成员方法的方式调用，先于对象创建之前创建内容，通常用来做静态初始化，和方法简单调用

## 类的成员变量--类属性--【类的属性并非对象的属性，重点区别】

- 1, 类属性 (类的成员变量) 定义
- 2, 访问类属性 (类的成员变量)
- 3, 什么时候使用对象属性, 什么时候使用类属性

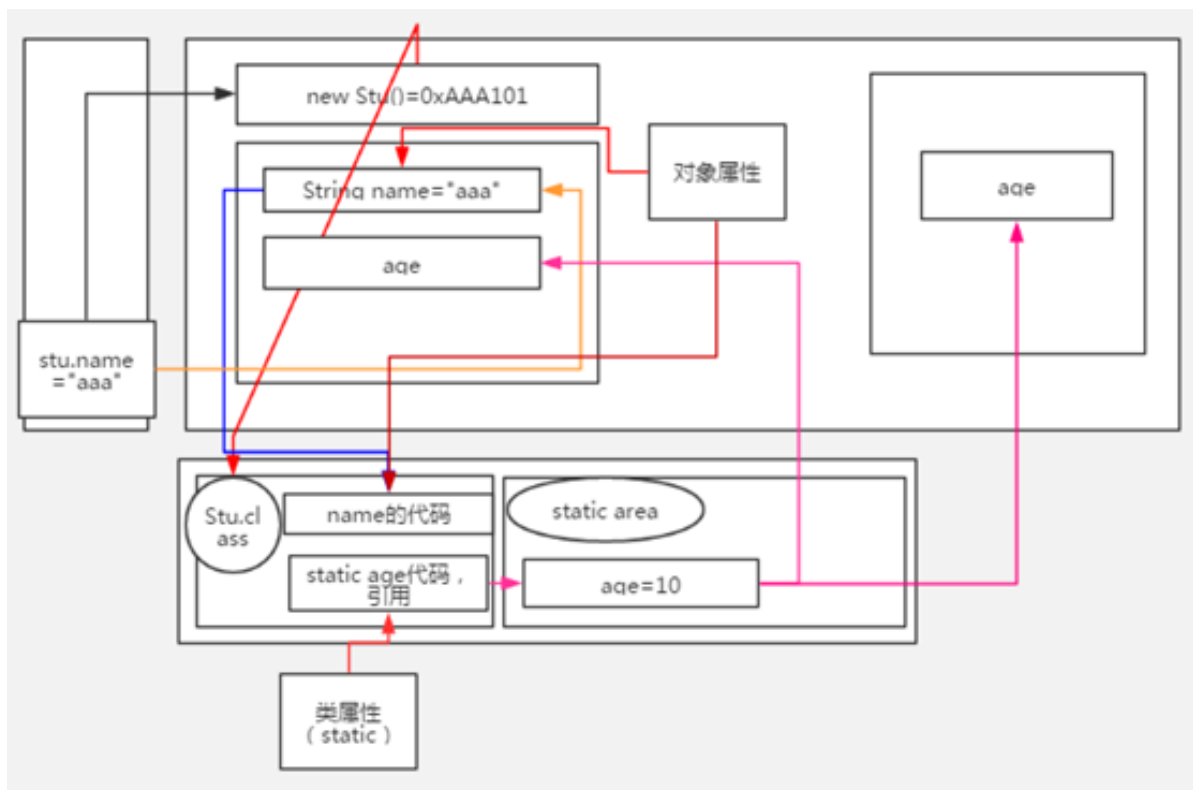
### 类属性 (类的成员变量) 定义

类【是class ClassXX 的成员变量 不是对象new ClassXX () 的成员变量】的成员变量——静态属性, 静态成员变量

```
Class Student
```

```
    Static int age;//静态属性
```

```
    String name;//对象属性
```



对象属性: 也叫做实例属性, 非静态属性

```
New Student().name="aaaa"
```

如果一个属性声明成类属性 (static修饰符), 所有的对象都共享一个属性值, 类似案例参照ATM机

### 访问类 (class-static变量) 属性

```
ATM.account=10;
```

```
类名. static成员变量
```

```
new ATM("").account=10;
```

```
对象.static成员变量
```

## 什么时候使用对象属性，什么时候使用类属性

如果一个英雄，他的装备不一样，就设计成**对象属性**

如果每个英雄的法量上限制都是1000000，可以设计成**类属性**

## 类方法

- 1, 类方法定义
- 2, 调用类方法
- 3, 什么时候使用对象方法，什么时候使用类方法

### 类方法定义

**类方法**：静态方法 static 方法

**对象方法**：实例方法，非静态方法

访问一个对象方法必须建立在一个对象的前提上

```
New Stu().nostatic();
```

访问类方法，不需要对象，直接可以访问

```
Stu.static()
```

### 调用类方法

调用static方法

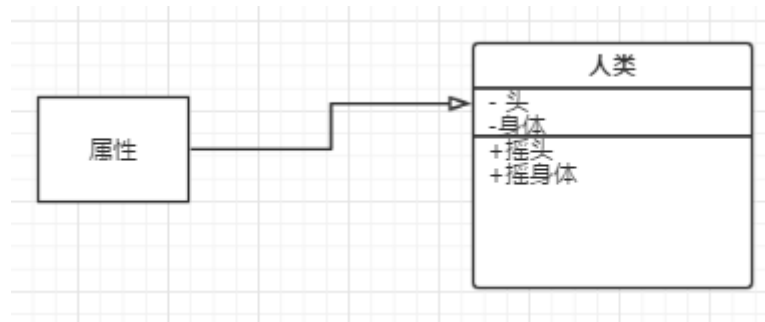
```
1 class Hero{
2     static void topHp(){
3     }
4 }
5 //类名.方法
6 Hero.topHp();
7 //对象.方法
8 new Hero().topHp();
```

## 什么时候使用对象方法，什么时候使用类方法

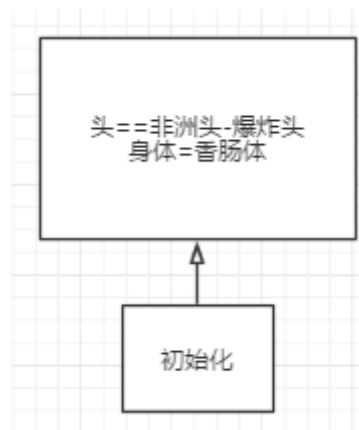
一局游戏，打印玩耍了多场时间，这个具体的时间并没有跟每个英雄关联起来，这样的共有时间属性的调用显示功能，就可以设计成类方法

```
1 Game.printTime();
2 //类设计
3 class Game{
4     static void printTime(){
5         //time--out
6     }
7 }
```

## 对象属性初始化——研究的是被new出来的东西



### 对象中的成员变量

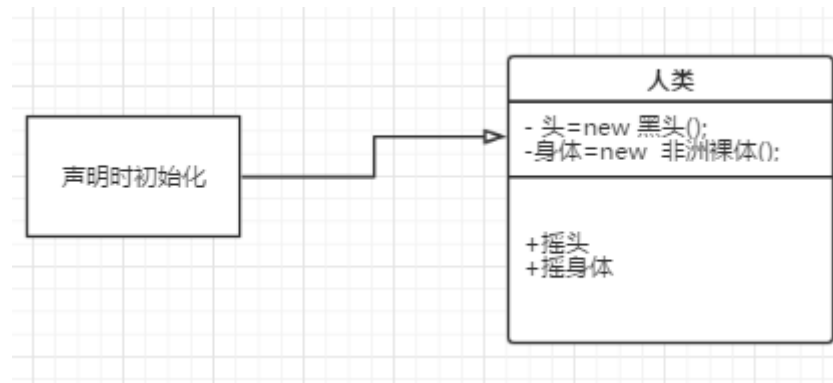


初始化：

成员变量第一次赋值

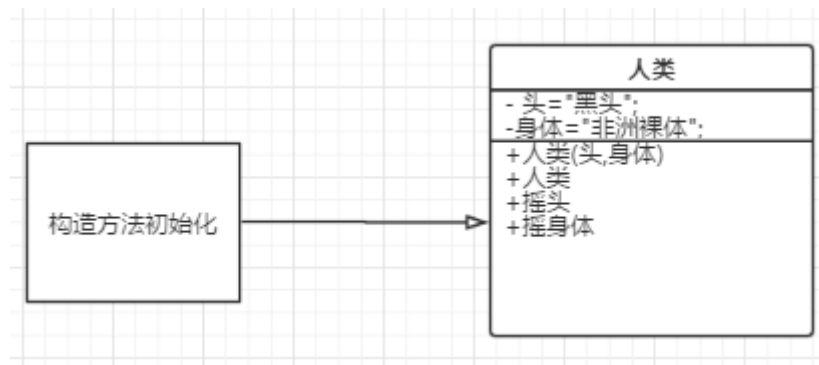
### 对象属性初始化的方式

1, 声明时初始化



```
1 class BlackPerson{
2     String head=new String("黑头");// String head="黑头";
3     String body=new String("非洲裸体");
4 }
5
```

2, 构造方法初始化

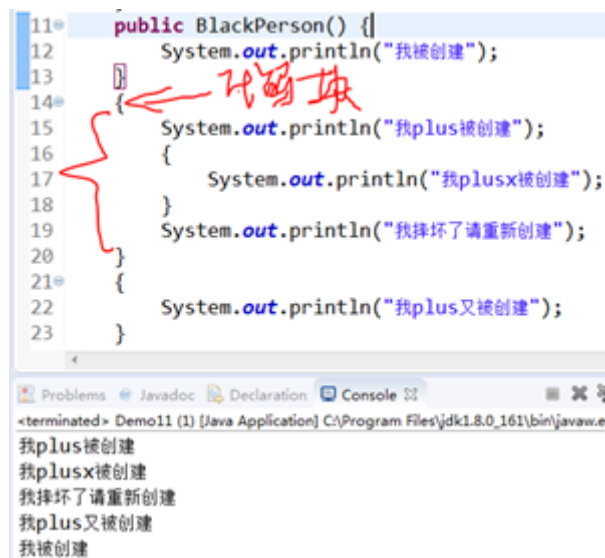


```

1 public BlackPerson(String head, String body) {
2     super();
3     this.head = head;
4     this.body = body;
5 }
6 public BlackPerson() {
7     super();
8 }
9

```

### 3, 代码块初始化



### 形式上的案例

只要是类中的{}，就叫一个代码块，在构造函数执行之前被调用，一般来说一个对象一个初始化代码块。

代码块初始化实用案例，打王者荣耀，**后羿**—class Hero—new Hero("后羿")

注释：以上三个概念均是针对后羿这个英雄对象的

```
25 class Hero{
26     private String name;
27     private int hp;
28     private int mp;//法量
29     //自定义的交给构造函数
30     public Hero(String name) {
31         this.name = name;
32     }
33     {
34         //系统默认
35         hp=100;
36         mp=100;
37     }
38     @Override
39     public String toString() {
```

Problems Javadoc Declaration Console

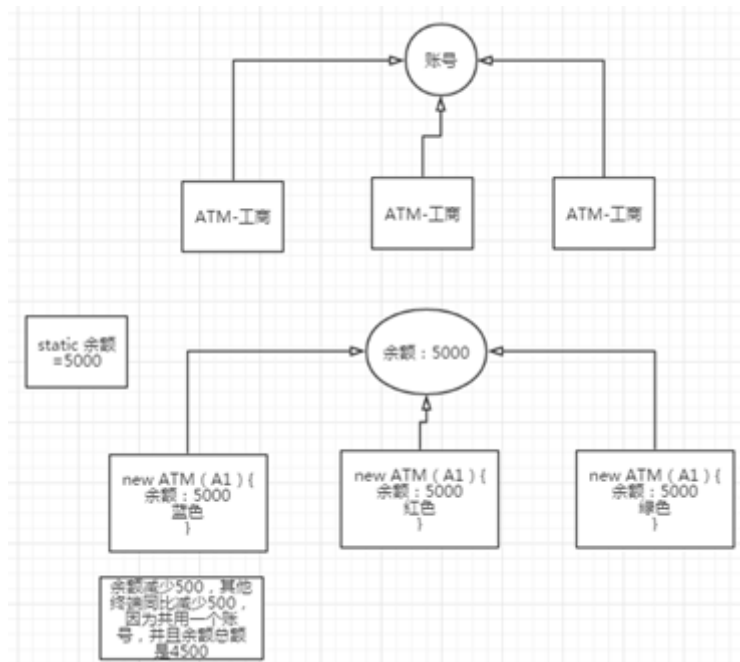
<terminated> Demo11 (1) [Java Application] C:\Program Files\

Hero [name=后羿, hp=100, mp=100]

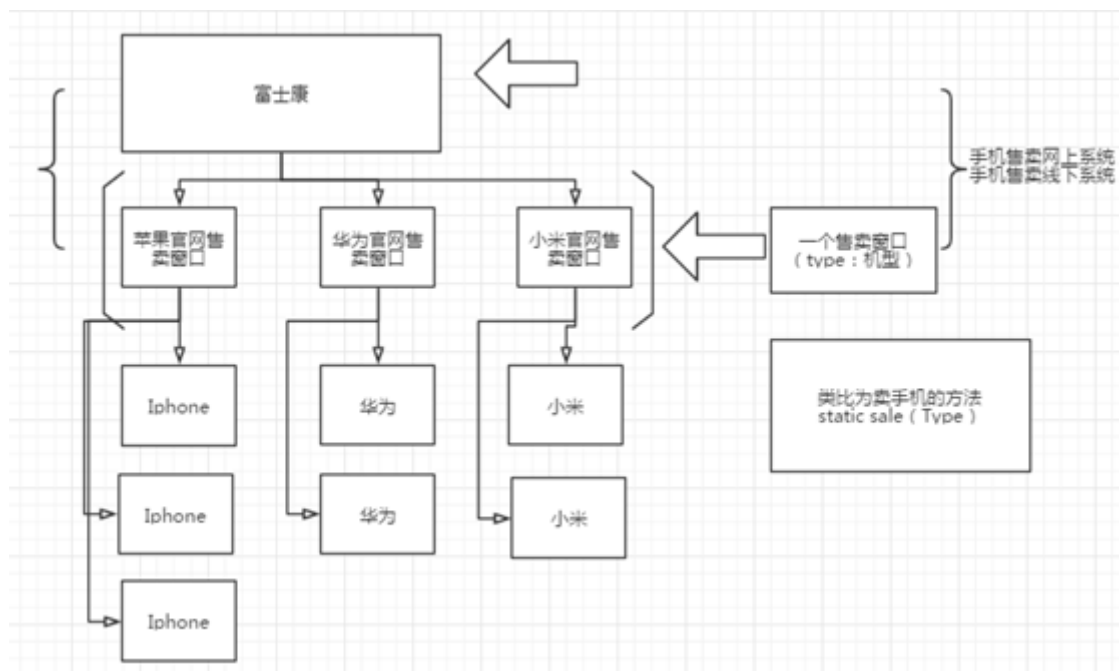
```
1 class Hero{
2     private String name;
3     private int hp;
4     private int mp;//法量
5     //自定义的交给构造函数
6     public Hero(String name) {
7         this.name = name;
8     }
9     {
10        //系统默认
11        hp=100;
12        mp=100;
13    }
14    @Override
15    public String toString() {
16        return "Hero [name=" + name + ", hp=" + hp + ", mp=" + mp + "];"
17    }
18 }
19
20 public class Demo11 {
21
22     public static void main(String[] args) {
23
24         Hero h=new Hero("后羿");
25         System.out.println(h);
26
27     }
28
29 }
30
```

## 类属性初始化——研究的是class{}代码层面的

直觉案例一：

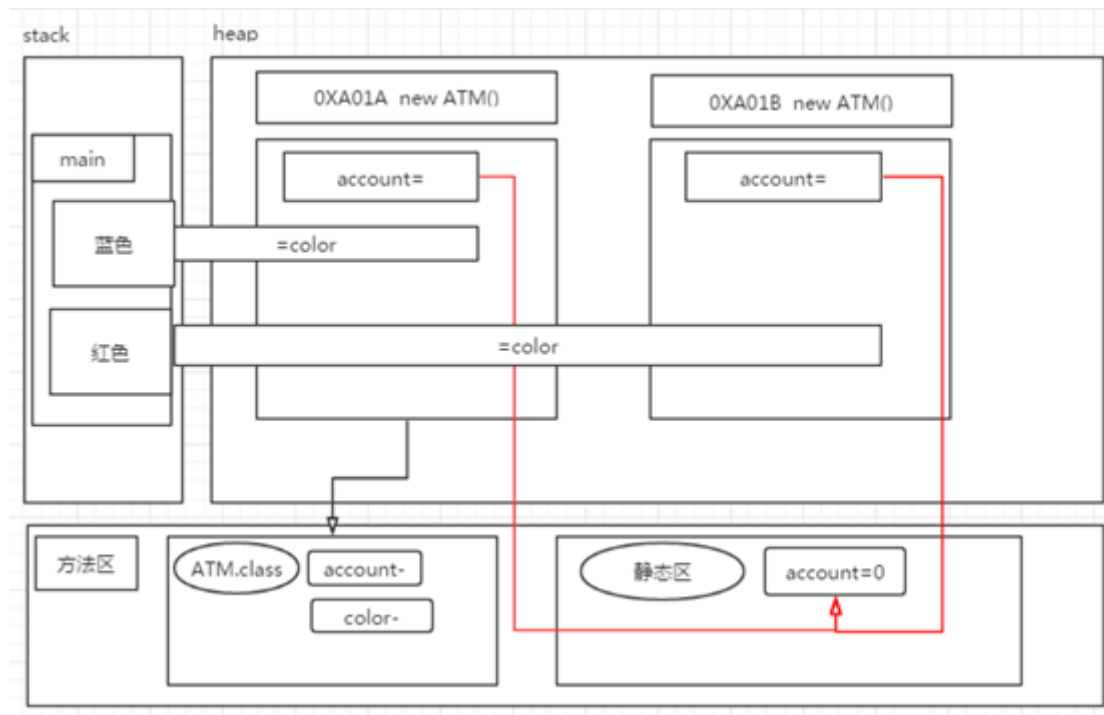


直觉案例二：



声明属性时初始化





```

1  class ATM{
2      //在定义类的时候声明属性时初始化
3      private static int account=0;//默认账号中的余额是0元
4      private String color;
5
6      public void setAccount(int account2) {
7          account+=account2;//account=account+account2;
8      }
9
10     public ATM() {
11     }
12     public ATM(String color) {
13         this.color = color;
14     }
15     @Override
16     public String toString() {
17         return "ATM [color=" + color + "]-[account=" + account + "];"
18     }
19 }
20
21 public class Demo11 {
22
23     public static void main(String[] args) {
24         ATM ma1=new ATM("粉色");
25         ATM ma2=new ATM("红色");
26         ATM ma3=new ATM("蓝色");
27         print(ma1,ma2,ma3);
28         ma1.setAccount(10);
29         print(ma1,ma2,ma3);
30         ma2.setAccount(8);
31         print(ma1,ma2,ma3);
32         ma3.setAccount(1);
33         print(ma1,ma2,ma3);
34     }
35     public static void print(ATM... mas) {
36         for(ATM ma:mas) {

```

```

37         System.out.println(ma);
38     }
39 }
40
41 }
42

```

来个静态方法变形

```

1  class Bank{
2      //在银行存钱
3      public static void setAccount(int account2) {
4          ATM.account+=account2;
5      }
6      static class ATM{
7          //在定义类的时候声明属性时初始化
8          private static int account=0;//默认账号中的余额是0元
9          private String color;
10         public void setAccount(int account2) {
11             account+=account2;//account=account+account2;
12         }
13         public ATM() {
14         }
15         public ATM(String color) {
16             this.color = color;
17         }
18         @Override
19         public String toString() {
20             return "ATM [color=" + color + "]-[account=" + account + "];
21         }
22     }
23 }
24 public class Demoll {
25
26     public static void main(String[] args) {
27         //类名.方法
28         Bank.setAccount(5000);
29
30         ATM ma1=new ATM("粉色");
31         ATM ma2=new ATM("红色");
32         ATM ma3=new ATM("蓝色");
33         print(ma1,ma2,ma3);
34         ma1.setAccount(10);
35         print(ma1,ma2,ma3);
36         ma2.setAccount(8);
37         print(ma1,ma2,ma3);
38         ma3.setAccount(1);
39         print(ma1,ma2,ma3);
40     }
41     public static void print(ATM... mas) {
42         for(ATM ma:mas) {
43             System.out.println(ma);
44         }
45     }
46

```

```
47 }
48
```

## 静态代码块初始化

```
1  class Bank{
2      //在银行存钱
3      public static void setAccount(int account2) {
4          System.out.println("Bank-setAccount");
5          //只要class ATM类名出现了一次，就会初始化class ATM中的所有的static变量
6          ATM.account+=account2;
7      }
8      static class ATM{
9          //静态代码块
10         static {
11             account=0;
12             System.out.println("static-init-针对class ATM类型滴，只要ATM.class
13             相关的信息一出现，就自动完成static初始化");
14         }
15         //在定义类的时候声明属性时初始化
16         private static int account;//默认账号中的余额是0元
17         private String color;
18         public void setAccount(int account2) {
19             account+=account2;//account=account+account2;
20         }
21         public ATM() {
22             //针对实例对象的代码块初始化，用来初始化实例对象的变量
23             {
24                 this.color="白色";
25                 System.out.println("代码块--"+color);
26             }
27             public ATM(String color) {
28                 this.color = color;
29                 System.out.println("构造方法--"+color);
30             }
31             @Override
32             public String toString() {
33                 return "ATM [color=" + color + "]-[account=" + account + "];"
34             }
35         }
36     }
37     public class Demo11 {
38
39         public static void main(String[] args) {
40             //类名.方法
41             Bank.setAccount(5000);
42             Bank.setAccount(5000);
43             ATM atm=new ATM();
44             System.out.println(atm);//5000? 10000?
45             //
46             // ATM ma1=new ATM("粉色");
47             // ATM ma2=new ATM("红色");
48             // ATM ma3=new ATM("蓝色");
49             // print(ma1,ma2,ma3);
```

```

50 //      ma1.setAccount(10);
51 //      print(ma1,ma2,ma3);
52 //      ma2.setAccount(8);
53 //      print(ma1,ma2,ma3);
54 //      ma3.setAccount(1);
55 //      print(ma1,ma2,ma3);
56     }
57     public static void print(ATM... mas) {
58         for(ATM ma:mas) {
59             System.out.println(ma);
60         }
61     }
62
63 }
64

```

## OOP核心下（继承，接口，抽象）

- 1, 接口-api, 继承, super关键字, Object 超类
- 2, 多态
- 3, 继承, 接口, 抽象之封装
- 4, Final
- 5, 抽象类
- 6, 内部类, 基于接口, 继承
- 7, 默认方法
- 8, 最佳实践

## 接口，继承，super关键字，Object 超类

案例实践

需求：王者荣耀，两类英雄，**约定**某些英雄是法系英雄，有些是物理系英雄，法系的魔法AP攻击，物理系的物理AD攻击

- 1, 设计接口ad, ap

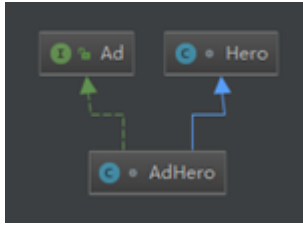
```

1 //约定两类英雄
2 interface AD{
3     //物理攻击
4     public void physicsAttack();
5 }
6
7 interface AP{
8     public void spellsAttack();
9 }
10

```

## 2, 搞一些英雄进来, 实现ad和ap功能

注意: 设计两类英雄APHero, ADHero, 他们均有血量, 法量, 姓名, 然后他们遵守AD和AP的约定, 所以需要让AD和AP这种约定称为一种特殊地类, 就是上述的interface AD AP, 我们设计的具体的英雄还应该有一个英雄模板Hero{血量, 法量, 姓名}, 因此继承结构如下:



```
1  //约定两类英雄
2  interface AD{
3      //物理攻击
4      public void physicsAttack();
5  }
6
7  interface AP{
8      public void spellsAttack();
9  }
10 //英雄的基本类
11 class Hero{
12     private String name;
13     private int hp;
14     private double price;
15     private int mp;
16     public String getName() {
17         return name;
18     }
19     public void setName(String name) {
20         this.name = name;
21     }
22     public int getHp() {
23         return hp;
24     }
25     public void setHp(int hp) {
26         this.hp = hp;
27     }
28     public double getPrice() {
29         return price;
30     }
31     public void setPrice(double price) {
32         this.price = price;
33     }
34     public int getMp() {
35         return mp;
36     }
37     public void setMp(int mp) {
38         this.mp = mp;
39     }
40 }
41 /*
42 *
43 */
44 //ad的英雄继承ad-物理 的约定
45 class AdHero extends Hero implements AD{
```

```

46     @Override
47     public void physicsAttack() {
48         System.out.println("实现AD物理攻击");
49     }
50 }
51
52 class ApHero extends Hero implements AP{
53     @Override
54     public void spellsAttack() {
55         System.out.println("实现AP法术攻击");
56     }
57 }
58 }
59

```

3, 设计一个既有ad功能也有ap功能的英雄

```

1  //接口interface可以多继承（实现implements），父类只能单继承
2  class XManHero extends Hero implements AD,AP{
3      @Override
4      public void spellsAttack() {
5          System.out.println("实现AD物理攻击");
6      }
7      @Override
8      public void physicsAttack() {
9          System.out.println("实现AP法术攻击");
10     }
11 }
12 }
13

```

4, 根据上述案例，什么时候使用interface这种接口

学习一个知识点，由浅入深，切不可揠苗助长，引入概念后，需要在多加实践上才去思考该如何使用，如果没有这个大量实践的过程，是不能达到真正地深刻理解的，我们刚接触一个难的概念，更应该大量实践，之后再就清楚具体运用（熟练运用）

案例练习

需求：

场景化需求分析：

定义一个通讯接口

接口中有打电话，发短信，网络连接功能

实现接口的有电话，平板

其中电话可以和平板进行打电话，发短信，相互连接网络

定义一个工作接口

接口中有开会，安排工作，审核工作

于是分别有5个人在不同的两个会议室里开远程会议

分别用手机和平板进行会议，其中手机会时不时给平板发短信，平板会时不时给手机开视频和打语音电话

逻辑化：

所有通讯设备必须满足通讯条件和通讯流程，符合这个通讯规范约束的设备才是通讯设备，代码体现为设计一个约束类（interface-接口-通讯设备规范结构接口）

无论是哪种通讯方式，底层结构都是有俩个对象，相互通讯传输，互为主语，所以推导出，通讯方式一定要有通讯对象，而这个通讯对象一定是符合通讯规范约束的通讯设备。

接受通讯规范约束的通讯设备A，与另外一个B，进行任意形式（三种之一）通讯，B能够以任意通讯形式返回数据。

寻找解决模型：

解决场景思考：思考选择设计模式——模板方法

代码：

```
1 package com.haoyu;
2
3 public class Demo18 {
4
5     public static void main(String[] args) {
6         // System.out.println(CommunicationStatus.internetStatus);
7         // System.out.println(CommunicationStatusPlus.phoneStatus);
8         Phone iphone=new Phone("iphone");
9         Phone android=new Phone("android");
10
11        android.setCurrentCommunicationStatus(CommunicationStatus.SMSStatus);
12        iphone.call(android);
13    }
14 }
15 //设置状态_枚举-->就是类型
16 //Status: 状态
17 //phoneStatus: 电话状态
18 //SMSStatus: 信息状态
19 //internetStatus: 网络状态
20 //CommunicationStatus.phoneStatus
21 enum CommunicationStatus{
22     phoneStatus,SMSStatus,internetStatus;
23 }
24 //如果你用不来上面的枚举，也可以整下面这个结构
25 //class CommunicationStatusPlus{
26 //    static String phoneStatus="phoneStatus";
27 //}
28 //Communication 交流 Tool工具
29 interface CommunicationTool{
30     void call(DefaultCommunicationToolTemplate ct);
31     void sendsMS(DefaultCommunicationToolTemplate ct);
32     void internet(DefaultCommunicationToolTemplate ct);
33 }
34 //Template 模板
35 //abstract 抽象 abstract class 理解成 和interface的区别在于既可以定义约束，又在实现方法
36 //abstract 跟interface一样，不可以被new出来
```

```

37 abstract class DefaultCommunicationToolTemplate implements
CommunicationTool{
38     private String name;
39     //0是没有发信息的状态
40     private int state=0;
41     //默认通讯状态是网络连接状态
42     //current :当前
43     private CommunicationStatus
currentCommunicationStatus=CommunicationStatus.phoneStatus;
44     //可以随时对这个通讯状进行改动
45     public CommunicationStatus getCurrentCommunicationStatus() {
46         return currentCommunicationStatus;
47     }
48     public void setCurrentCommunicationStatus(CommunicationStatus
currentCommunicationStatus) {
49         this.currentCommunicationStatus = currentCommunicationStatus;
50     }
51     public abstract void call(DefaultCommunicationToolTemplate ct);
52     public abstract void sendSMS(DefaultCommunicationToolTemplate ct);
53     public abstract void internet(DefaultCommunicationToolTemplate ct);
54     //content:内容
55     //protected:只能子类调用
56     protected void defaultCommunication(DefaultCommunicationToolTemplate
ct,String content) {
57         if(state==0) {
58             System.out.println(this+content);
59             state=1;
60             ct.communicationStatusChoose(this);//this--phone
61         }
62     }
63     //可以通过这个方法来选择执行上面那条语句
64     //主要用来做回送信息
65     public void communicationStatusChoose(DefaultCommunicationToolTemplate
ct) {
66         switch(this.currentCommunicationStatus) {
67             case phoneStatus:call(ct);break;
68             case SMSStatus:sendSMS(ct);break;
69             case internetStatus:internet(ct);break;
70         }
71     }
72     //
73     public int getState() {
74         return state;
75     }
76     public void setState(int state) {
77         this.state = state;
78     }
79     public DefaultCommunicationToolTemplate(String name) {
80         this.name = name;
81     }
82     public String getName() {
83         return name;
84     }
85     public void setName(String name) {
86         this.name = name;
87     }
88
89     @Override

```



```

90     public String toString() {
91         return "CommunicationTool [name=" + name + ",
currentCommunicationStatus="
92             + currentCommunicationStatus + "]";
93     }
94 }
95 class Phone extends DefaultCommunicationToolTemplate{
96     public Phone(String name) {
97         super(name);
98     }
99     // //0是没有发信息的状态
100    // private int state=0;
101    // public int getState() {
102    //     return state;
103    // }
104    // public void setState(int state) {
105    //     this.state = state;
106    // }
107    //ct--CommunicationTool简写: 通讯工具
108    public void call(DefaultCommunicationToolTemplate ct) {
109        //     ///java.lang.StackOverflowError
110        //     if(state==0) {
111        //         System.out.println(this+"疯狂打call");
112        //         state=1;
113        //         ct.communicationStatusChoose(this);//this--phone
114        //     }
115        defaultCommunication(ct,"疯狂打call");
116    }
117    public void sendsMS(DefaultCommunicationToolTemplate ct) {
118        defaultCommunication(ct,"疯狂打sms");
119    }
120    public void internet(DefaultCommunicationToolTemplate ct) {
121        defaultCommunication(ct,"疯狂打ip");
122    }
123 }
124 /*
125 i-state=0
126 iphone-call(android){
127     -->iphone
128     i-state=1
129     android.choose(iphone);
130     {
131         case phoneStatus:call(ct){
132             a-state=1;
133             iphone.choose(android){
134                 call(){
135                     state=1;
136                     syso-->call;
137                     break;
138                 }
139             }
140         }
141     }
142 }
143 */
144

```

# 对象转型

- 1, 引用类型与对象类型的概念
- 2, 向上转型, 向下转型, 没有继承联系的两个类的转换
- 3, Interface层面 (向上, 向下转型)
- 4, Instanceof

## 引用类型与对象类型的概念

```
1      public static void main(String[] args) {
2
3          Animal cat=new Animal();
4          //引用--> cat
5          //对象--> cat--> 【new Animal()】
6
7      }
8  }
9  class Animal{
10 }
```

上诉例子: cat就是引用对象, new Animal () 就是对象

引用对象cat的类型: Animal

new Animal 的类型: Animal

正常情况下引用对象 (那个指向堆内存地址的变量) 的类型和堆中被new出来的那个对象的类型, 是一样的

## 向上转型, 向下转型, 没有继承联系的两个类的转换

不管是向上转, 还是向下转, 引用类型和对象类型不一致的时候才有需要转换的需求

```
1  public class Demo19 {
2      public static void main(String[] args) {
3
4          Animal cat = new Animal();
5          // 引用--> cat
6          // 对象--> cat--> 【new Animal()】
7
8          // 苹果手机继承了普通手机的结构, 我们把苹果手机当做普通手机来使用
9          NomaIphone phone = new Iphone();//向上转型
10         phone.call();
11         phone.sms();
12         // phone.siri();这里是有问题滴, 必须是你知道这是苹果手机, 你才知道要用siri, 如果你
           只是把这个当做一个般手机, 你
13             //只知道要打电话和发短信
14             //需要知道这个普通手机能够运行苹果系统, (Iphone) 实际上就是声明一下要当做
           iphone来看看使用
15         ((Iphone)phone).siri();//向下转型
16         //
17         //农夫山泉继承了饮料约束的结构      我们把农夫山泉当做饮料来使用
18         Drink drink=new DrinkNF();//向上转型
19         //现在要实现农夫山泉特殊的甜味, 对drink引用变量进行向下转型, 告诉使用者这里要使用
           农夫山泉的方法了
```

```

20         ((DrinkNF)drink).addSth();//向下转型
21
22         //java.lang.ClassCastException
23         NomalPhone iphone=new NomalPhone();//山寨鸡      如果要向上转型，实例类型
                        必须是引用类型的子类
24         ((Iphone)iphone).siri();
25
26         //苍老师继承了动物的结构，我们把苍老师当成。。。
27         Animal mc=new CangLaoShi();
28         mc.jiao();
29         //苍老师上录播课
30         ((CangLaoShi)mc).lessonVedio();
31
32     }
33 }
34 //接口 约束 drink
35 interface Drink{
36     void useMouse() ;
37 }
38 class DrinkNF implements Drink{
39     public void useMouse() {
40
41     }
42     public void addSth() {
43
44     }
45 }
46
47 class NomalPhone {
48     public void call() {
49
50     }
51     public void sms() {
52
53     }
54 }
55 class Iphone extends NomalPhone {
56     // 模拟的人工智能
57     public void siri() {
58         System.out.println("siri");
59     }
60 }
61 class Animal {
62
63     public void jiao() {
64
65     }
66
67 }
68 class CangLaoShi extends Animal{
69
70     public void lessonVedio() {
71
72     }
73 }

```

案例分析，上午案例中使用了向上转型

```

1      abstract DefaultCommunicationToolTemplate
2
3      Phone extends DefaultCommunicationToolTemplate
4
5      new Phone
6
7      public void call(DefaultCommunicationToolTemplate ct) -->new Phone
8
9      DefaultCommunicationToolTemplate ct=new Phone();
10

```

往方法中传递一个符合通讯约束规范的通讯设备，于是我们传进去了一个通讯设备android

## 没有继承关系的一旦转换，会报错

```

1  //java.lang.ClassCastException
2      NomalPhone iphone=new NomalPhone();
3  //山寨鸡 如果要向上转型，实例类型必须是引用类型的子类
4      ((Iphone)iphone).siri();

```

## Interface层面（向上，向下转型）

任意类都继承Object

Class Object

java.lang.Object

public class Object

Class `Object` 是类 `Object` 结构的根。每个class都有 `Object` 作为超类。所有对象（包括数组）都实现了这个类的方法。

从以下版本开始：

JDK1.0

Interface xxx

Interface yyy extends xxx

Abstract class zzz implements yyy

Class aaa extends zzz

## 接口向上转型，向下转型

```

1  public class Demo20 {
2      public static void main(String[] args) {
3          //把可乐当成水喝
4          water3 color=new BColor();
5          ((BColor)color).shaJiJing();
6      }
7  }
8  //定义约束

```

```

9  interface Water3{
10     public void drink();
11 }
12 //可乐
13 class BColor implements Water3{
14     public void drink() {
15         System.out.println("喝阔乐");
16     }
17     public void shaJiJing() {
18         System.out.println("杀鸡精");
19     }
20 }
21

```

## Instanceof—引用对象的类型是否是指定类型的系列（自己到子类）

Instanceof Animal 判断一个引用所指向的对象，是否是Animal类型，animal的子类

```

1  package com.haoyu;
2  class Herox{
3  }
4  class ADHerox extends Herox{
5  }
6  class APHerox extends Herox{
7  }
8  public class Demo21 {
9
10     public static void main(String[] args) {
11         ADHerox ad=new ADHerox();
12         APHerox ap=new APHerox();
13         Herox h1=ad;
14         Herox h2=ap;
15         //判断引用h1指向对象，是否是adherox
16         // System.out.println(h1 instanceof ADHerox);//true
17         // //apherox
18         // System.out.println(h2 instanceof APHerox);//true
19         // //是否是herox的子类
20         // System.out.println(h1 instanceof Herox);//true
21         //动物 苍老师
22         TeaCang tc=new TeaCang();
23         wildAnimal wa=tc;
24         //是不是苍老师类型的?
25         System.out.println(wa instanceof TeaCang);
26         //是不是野生动物类型的?
27         System.out.println(wa instanceof wildAnimal);
28     }
29 }
30 //野生动物
31 class wildAnimal{
32
33 }
34 class TeaCang extends wildAnimal{
35
36 }

```

## 重写（基于继承结构的）

继承结构下，子类如果包含一个与父类方法相同的方法，子类自己的方法就会覆盖父类的方法

（private）私有的成员变量以及方法是不能够传递给子类的

如果只是需要子类继承父类的成员变量或者方法，则需protected修饰

**两部手机都是iphone4**，me-i4在我的房间，fu-i4在父的房间，现在我在我的房间拿iphone4→me-i4，我进入父的房间，拿iphone4，手机却是fu-i4\*\*

```
1
2 class Demo22 {
3
4     public static void main(String[] args) {
5         // Type mismatch: cannot convert from Aniaml1 to Cang
6         // Cang cang=new Aniaml1();
7         // Aniaml1 a1=(Aniaml1)new Cang();
8         // Aniaml1 a2=new Cang();
9         // Life a3=new Cang();
10        // new Cang() {
11        //     void actionMovie() {
12        //         System.out.println("苍式moive");
13        //     }
14        // }.actionMoive(1);// extends Cang();
15        // 在以上结构不变的情况下，需要一句语法执行父类和子类的相同方法的不同执行
16        new CangPlus();//使用cangplus对象 打印父类cang中的private name
17        System.out.println(new CangPlus().getName());
18        //就近原则
19
20    }
21 }
22 class Cang extends Aniaml1 {
23     private String name="cangjielun";
24     protected String getName() {
25         System.out.println(this);
26         System.out.println(this.name);
27         return this.name;
28     }
29     private void actionMovie() {
30         System.out.println("苍式moive2");
31     }
32     protected void actionMoive(int a) {
33
34     }
35 }
36 class CangPlus extends Cang{
37     string name="杰伦";
38     public String getName() {
39         super.getName();
40         System.out.println(this);
41         return this.name;
42     }
43 }
44 interface Life {
45
```

```

46     }
47     class Aniam11 implements Life {
48
49     }
50

```

## 多态-- (多种状态)

### 操作符的多态

+可以作为算数运算，也可以作为字符串连接

```
int a=1+1; String a1=a+"pp";
```

### 类的多态

父类引用指向子类对象，方便业务场景的模块拆与装

使用多态，就类似同一把枪换弹夹，激光弹夹打激光弹，跑弹夹打炮弹

不适用多态，类似换枪，激光枪打激光，炮弹枪打炮弹，手枪 打手枪

```

1  public class Demo23 {
2
3      public static void main(String[] args) {
4          int a=1+1;
5          String a1=a+"pp";
6
7          //多态//换弹夹
8          PersonAnimal pa=new PersonAnimal();
9          pa.eat();
10         pa=new Fu();
11         pa.eat();
12         pa=new Son();
13         pa.eat();
14         pa=new GrandSon();
15         pa.eat();
16         //等于换枪
17         PersonAnimal papa=new PersonAnimal();
18         papa.eat();
19         Fu fu=new Fu();
20         fu.eat();
21         Son son=new Son();
22         son.eat();
23         GrandSon gs=new GrandSon();
24         gs.eat();
25
26         fu.fuEat();
27         son.sonEat();
28         gs.grandSonEat();
29
30     }
31
32 }
33 //智人
34 class PersonAnimal{

```

```

35     public void eat() {
36         System.out.println("蛋鞭");
37     }
38 }
39 class Fu extends PersonAnimal{
40     public void eat() {
41         System.out.println("锁阳");
42     }
43     public void fuEat() {
44         System.out.println("锁阳");
45     }
46 }
47 class Son extends Fu{
48     public void eat() {
49         System.out.println("韭菜");
50     }
51     public void sonEat() {
52         System.out.println("韭菜");
53     }
54 }
55 class GrandSon extends Son{
56     public void eat() {
57         System.out.println("吃纸");
58     }
59     public void grandSonEat() {
60         System.out.println("吃纸");
61     }
62 }

```

## Super关键字

- o 定义一个父类，带无参构造函数
- o 实例化子类，调用父类的构造函数
- o 给父类加有参数的构造函数
- o 子类调用父类的有参数构造函数
- o 调用父类属性
- o 调用父类方法

```

1  package com.haoyu;
2
3  public class Demo25 {
4
5  }
6  class Fux{
7      public Fux() {}
8      public Fux(String name) {
9
10     }
11     public String name;
12     private int age;
13     protected int id;

```



```

14     public int getAge() {
15         return age;
16     }
17     //父类的public修饰关键字和protected都可以被子类super关键字调用
18     public void setAge(int age) {
19         this.age = age;
20     }
21     protected int getId() {
22         return id;
23     }
24 }
25 class Sonx extends Fux{
26
27     //默认有一个无参构造函数
28     //继承结构，必须要调用一个父类构造函数，如果没有默认的，也必须要调用一个带参数的
29     public Sonx() {
30         //默认方法 super();
31         super(); //显示地写出来，调用父类的构造方法，把父类对象给new出来
32         super.name=""; //使用super关键字，代表new出来的父类对象
33         //getAge()实际上等于 super.getAge(),只是子类里没有getAge的方法，因此这里的
34         getAge就可以省略super
35         //当然，使用super.getAge();更加的直观
36         //如果子类中也有getAge的方法，这里的调用就是this.getAge();
37         //super.getAge(); //使用父类的公共方法可以调用父类的私有变量
38         getAge();
39         super.id=10; //protected修饰符：代表只能被子类访问属性
40     }
41     public Sonx(String name) {
42         //默认方法 super()
43         super(name);
44     }
45 }

```

## Object类

### public class Object

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

Object是所有自定义和系统自带类class的顶级父类，写不写继承都继承它

boolean	<b>equals(Object obj)</b> Indicates whether some other object is "equal to" this one.
protected void	<b>finalize()</b> Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	<b>getClass()</b> Returns the runtime class of this Object.
int	<b>hashCode()</b> Returns a hash code value for the object.
void	<b>notify()</b> Wakes up a single thread that is waiting on this object's monitor.
void	<b>notifyAll()</b> Wakes up all threads that are waiting on this object's monitor.
String	<b>toString()</b> Returns a string representation of the object.
void	<b>wait()</b> Causes the current thread to wait until another thread invokes the <b>notify()</b> method or the <b>notifyAll()</b> method for this object.
void	<b>wait(long timeout)</b> Causes the current thread to wait until either another thread invokes the <b>notify()</b> method or the <b>notifyAll()</b> method for this object, or a specified amount of time has elapsed.
void	<b>wait(long timeout, int nanos)</b> Causes the current thread to wait until another thread invokes the <b>notify()</b> method or the <b>notifyAll()</b> method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

Equals:值相等

```
public boolean equals(Object obj) { return (this == obj);}
```

## 一般使用的时候需要重写equals

```
1 package com.haoyu;
2
3 public class Demo26 extends Object{
4
5     public static void main(String[] args) {
6
7         SupportHerox sh1=new SupportHerox();
8         sh1.name="a";
9         sh1.hp=120;
10        sh1.id=1;
11        SupportHerox sh2=new SupportHerox();
12        sh2.name="a";
13        sh2.hp=120;
14        sh2.id=1;
15        System.out.println(sh1.equals(sh2)); //对象中的指定值是否相同
16        System.out.println(sh1==sh2); //比对的是堆空间中的地址，判断的是对象是否相等
17
18    }
19
20 }
21 //辅助英雄
22 class SupportHerox{
23     public String name;
24     public int hp;
25     public int id; //1是蓝方 2是红方
26
27     //先不管这些代码时什么意思
28     //简单理解为：如果name, hp, id的值都相同，就算是堆空间中的地址不相同，我们都认为这个两个对象是一个对象
29     @Override
30     public int hashCode() {
31         final int prime = 31;
32         int result = 1;
33         result = prime * result + hp;
```

```

34         result = prime * result + id;
35         result = prime * result + ((name == null) ? 0 : name.hashCode());
36         return result;
37     }
38     //alt+shift+s  hide自动生成代码重写选中值
39     @Override
40     public boolean equals(Object obj) {
41         if (this == obj)
42             return true;
43         if (obj == null)
44             return false;
45         if (getClass() != obj.getClass())
46             return false;
47         SupportHerrox other = (SupportHerrox) obj;
48         if (hp != other.hp)
49             return false;
50         if (id != other.id)
51             return false;
52         if (name == null) {
53             if (other.name != null)
54                 return false;
55         } else if (!name.equals(other.name))
56             return false;
57         return true;
58     }
59
60 }

```

## toString:把对象打印成字符串

```

1     getClass().getName() + '@' + Integer.toHexString(hashCode())
2     @Override
3     public String toString() {
4         return "SupportHerrox [name=" + name + ", hp=" + hp + ", id=" + id + "]";
5     }

```

## Finalize：当一个对象没有任何引用指向的时候，他就满足垃圾回收条件

当这个对象被回收的时候，他的finalize方法就会被调用

这个方法不是由开发人员调用，而是有jvm自动调用

```

1  SupportHerox sh;
2  for(int i=0;i<1000000;i++) {
3      //一直new新对象出来
4      //每次创建一个对象，前一个sh就移到新的对象上来，而上一个就没有引用指向他了
5      //这些失去引用对象的孤魂野鬼，满足被垃圾回收的条件
6      //当垃圾堆积多了的时候，jvm就会触发垃圾回收机制，每个被回收的对象就会调用finalize方法
7      sh=new SupportHerox();//0xAAF  0xAA10
8  }
9  public void finalize() throws Throwable{
10     System.out.println(this+"这个辅助英雄正在被销毁");
11 }

```

一般情况下不去改动这个finalize方法

## Final

- 1, final修饰类
- 2, final修饰方法
- 3, final修饰基本类型变量
- 4, final引用
- 5, 常量

```

1  //01
2  //final修饰类，意味着这个类无法被继承，子类如果要去继承会直接出现编译错误
3  //还有运行就直接报错叫你修改
4  final class Fuy{
5
6  }
7  //class Ziy extends Fuy{
8
9  //}
10 //02
11 //final修饰方法，意味着方法不能被重写
12 class Fuyy{
13     final int getAge() {
14         return 50;
15     }
16 }
17 class Sony extends Fuyy{
18     //子类这里无法重写或者覆盖父类带有final修饰的方法
19     // int getAge() {
20     //     return 1;
21     // }
22 }
23 //03 final修饰基本类型变量，对引用变量进行修饰
24 //表示该对象只有一次机会被赋值，一旦赋值成功，就不能再被赋值了
25 //04final修饰引用类型 和上一样
26 class Sonyy{
27     //基本变量 int
28     final int age=1;
29     {
30         // age=10;
31     }
32     //应用类型 String 类似的只要是类，也都满足这个效果

```

```

33     final String name="aaaa";
34     // {
35     //     name="ppp";
36     // }
37     public void method() {
38     //     name="[[[[";
39         final String name2="aaaa";
40     //     name2="xxxx";
41     }
42 }
43 //05常量，对所有代码公开，无法更改，作为全局参数变量
44 //
45 class Sonyyy{
46     //常量--国籍，常量的引用名称一般是全大写
47     public static final String COUNTRY="中国";
48 }

```

## 类图——uml图的一种，uml建模（帮助理清思路）

