# 第五章 面向对象基础

## 一、面向对象编程思想

首先解释一下“思想”。

先问你个问题：你想做个怎样的人？

可能你会回答：我想做个好人，孝敬父母，尊重长辈，关爱亲朋……

你看，这就是思想。这是你做人的思想，或者说，是你做人的原则。

做人有做人的原则，编程也有编程的原则。这些编程的原则呢，就是编程思想。

常见的编程思想有：

**1、面向过程（POP）编程思想**

面向过程POP（Process-Oriented Programming）强调的是系统的数据被加工处理的过程，数据仅仅作为输入和输出存在，在程序设计中主要以函数或者过程为程序的基本组织方式。

比如说，C语言中，数据与处理这些数据的函数是分离的，程序的最小单元是函数。

**2、面向对象（OOP）编程思想**

前辈们说，这样不好，应该把数据和与这些数据相关的行为放在一起，用一个叫做类的东西装起来。于是，在Java中，程序的最小单元变成了类。——这种思想叫做面向对象编程。

面向对象（Object Oriented Programming）是以**对象**为中心，将对象的内部组织与外部环境区别开来，将表征对象的内部属性数据与外部隔离开来，其行为与属性构成了一个整体，而系统功能则表现为一系列对象之间的相互作用的序列，能更加形象的模拟或表达现实世界。

当然，会使用Java语言并不代表已经领会了面向对象的思想。面向对象有三大特性：封装、继承、多态。封装有什么作用？封装时要注意什么？继承有什么优点和缺点？什么时候使用继承？多态有什么好处？什么时候使用多态？……这些问题的解答有一些指导原则，比如单一职责原则、里氏替换原则、开闭原则、接口隔离原则、依赖倒置原则等等。这些原则就是编程思想。

举个例子来说，你写了一个程序，现在用的是SQLServer的数据库，但客户说，将来也可能换成MySQL、Oracle数据库。那你可能就会考虑了：如何在现有的基础上，做最少的改动，就能从SQLServer换到MySQL、Oracle。于是，你就会开始想这些问题：如何能达到更少的变动？是否需要创建接口？在什么地方创建接口？接口中定义哪些方法？如何让调用者更方便地获取接口的实现类的对象？……这个，就是你的编程思想。就是那些原则在你的实践中的应用。

**3、小结**

面向过程：POP: Process-Oriented Programming

以函数（方法）为最小单位

以过程，步骤为主，考虑怎么做

面向对象：OOP: Object Oriented Programming

以类/对象为最小单位，类包括：数据+方法

以对象（谁）为主，考虑谁来做，谁能做

面向对象仍然包含面向过程，只不过关注点变了，关注谁来做

程序员的角色：

面向过程：程序员是具体执行者

面向对象：程序员是指挥者

面向对象的开发过程：其实就是不断创造对象，使用对象，指挥对象做事情

面向对象的设计过程：就是管理和维护对象之间的关系

**4、举例说明**

经典案例：把大象装进冰箱总共分几步？

面向过程：

1. 把冰箱门打开
2. 把大象放进去
3. 把冰箱门关上

面向对象：

人{

拉（冰箱）{

冰箱.打开();

}

指挥(动物){

动物.进入();

}

推(冰箱){

冰箱.关闭();

}

}

冰箱{

打开(){ }

存储(){ }

关闭(){ }

}

大象{

进入(){ }

}

长颈鹿{

进入(){ }

}

面向对象更加注重前期的设计：

1. 就是类的设计
2. 设计类就是设计类的成员：属性、方法...等

面向对象的学习主线：

1. 类与对象
2. 面向对象的基本特征和高级特性
3. 相关关键字和API

## 二、类与对象

### 1、类与对象的概念

将现实生活中一类事物的共性内容进行提取，抽象成相应的Java类，用Java类对其进行描述。

例如：柴老师、宋老师、张老师、苍老师

共性的内容：姓名、年龄、性别

吃饭、睡觉、讲课

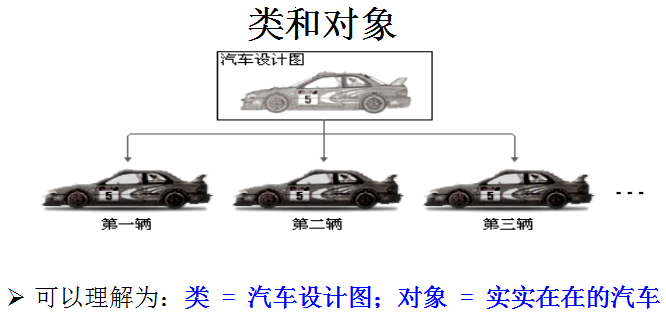
类与对象是面向对象的核心概念：

类(class)：对现实生活中一类事物的描述，抽象的

对象(object)：是实际存在的该类事物的每个个体，因而也称实例(instance)，是一个实实在在的个体，具体的，

|  |
| --- |
| **public** **class** TestOOP1 {  **public** **static** **void** main(String[] args) {  Teacher chai = **new** Teacher();  chai.name = "cai";  chai.age = 30;  chai.gender = '女';  System.***out***.println("姓名："+chai.name + ",年龄："+chai.age +",性别："+chai.gender);  chai.eat();  chai.sleep();  chai.teach();    Teacher cang = **new** Teacher();  cang.name = "苍老师";  cang.age = 18;  cang.gender = '女';  System.***out***.println("姓名："+cang.name + ",年龄："+cang.age +",性别："+cang.gender);  cang.eat();  cang.sleep();  cang.teach();  }  }  **class** Teacher{  String name;  **int** age;  **char** gender;  **void** eat(){  System.***out***.println("吃饭");  }  **void** sleep(){  System.***out***.println("睡觉");  }  **void** teach(){  System.***out***.println("讲授技术");  }  } |

### 2、类与对象的关系



要描述特性或完成功能，需要拿具体的对象，而不能拿类来解决。

例如：要开车只能拿对象——实实在在的车，不能拿汽车类——汽车的设计图来开。

总结：类是创建对象的模板

### 3、类的设计

类的设计其实就是**类的成员**的设计，即**事物的描述数据**及**其操作数据的行为方法**等等的设计。

* 数据描述：用属性表示，对应类中的成员变量
* 行为描述：用方法表示，对应类中的成员方法
* 其他都是为了数据和行为服务的：对应类中的其他成员

#### 类的定义格式

|  |  |  |
| --- | --- | --- |
| 【修饰符】 class 类名{  成员声明;  } | 【修饰符】 class 类名{  属性列表  方法列表  } | 【修饰符】 class 类名{  属性列表  代码块  构造器列表  方法列表  内部类  } |



示例代码：

|  |
| --- |
| **class** Teacher{  String name;  **int** age;  **char** gender;  **void** eat(){  System.***out***.println("吃饭");  }  **void** sleep(){  System.***out***.println("睡觉");  }  **void** teach(){  System.***out***.println("讲授技术");  }  **void** getInfo(){  System.***out***.println(name + "老师，今年" + age + "岁");  }  } |
| **class** Person{  //属性或成员变量  String name;  **boolean** isMarried;  //代码块  {  name = "匿名";  }  //构造器  Person(){}  Person(String n,**boolean** m){  name = n;  isMarried = m;  }  //方法  String display(){  **return** "姓名：" + name + ",婚否：" + isMarried;  }  //内部类  **class** Pet{  String name;  **double** weight;  }  } |

### 4、对象的创建

（1）声明引用变量来引用对象：数据类型 变量名;

例如：Teacher tea;

因为该变量是用来引用对象的，因此也称为对象名。

（2）为引用变量赋值一个对象： 变量名 = new 类名();

例如：tea = new Teacher(); //通过new 构造器()来创建对象

可以写成：Teacher tea = new Teacher();

（3）通过引用变量操作对象的属性和方法：对象名.属性、对象名.方法

tea.name = “柴林燕”;

tea.teach();

回忆之前接触的：

Scanner input = new Scanner(System.in); int age = input.nextInt();

Random rand = new Random(); int num = rand.nextInt(100);

### 5、匿名对象

我们也可以不定义对象的句柄，即引用变量，而直接调用这个对象的方法等，这样的对象叫做匿名对象。

如：new Person().shout();

使用情况：

（1）如果对一个对象只需要进行一次方法调用，那么就可以使用匿名对象。

（2）我们经常将匿名对象作为实参传递给一个方法调用。

## 三、类的成员之一：属性

在Java中对象的属性、字段也称为成员变量。

### 1、声明属性的位置

类中，方法外

### 2、声明属性的格式

语法格式：

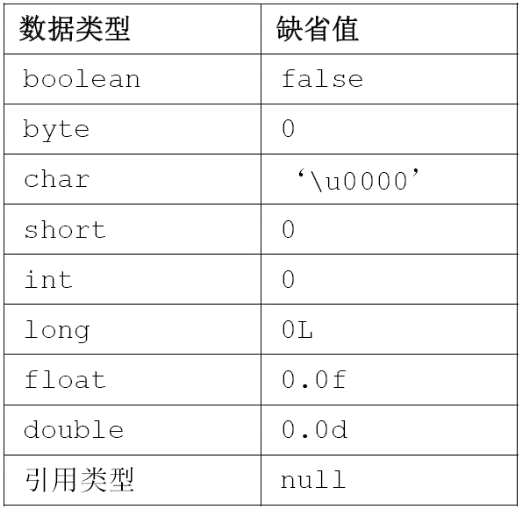
【修饰符】 数据类型 属性名 ; //默认初始值赋值

【修饰符】 数据类型 属性名 = 初始值 ; //显式初始化赋值

|  |
| --- |
| 【修饰符】 class 类{  【修饰符】 数据类型 属性名 ;  } |
| class Student{  String name;  int age;  } |

### 3、属性的特点

（1）属性也是变量，因此在使用前也要有初始值，不同的是，成员变量如果**不显式初始化**，会有默认值。

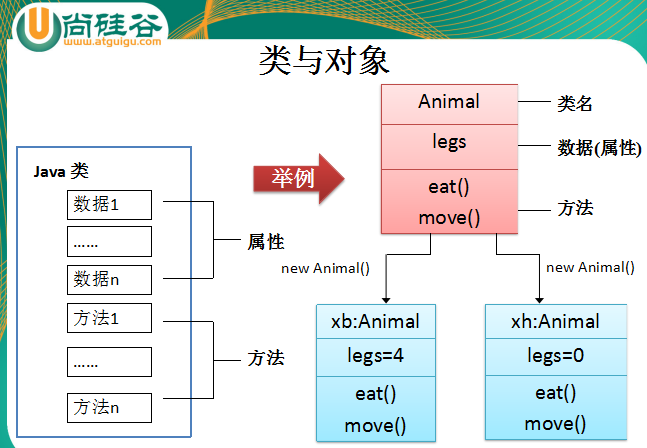


（2）如果创建了一个类的多个对象，对于类中的属性和方法，每个对象都拥有一个独立的副本，且互不干扰。

|  |
| --- |
| public class TestField{  public static void main(String[] args){  //1、先有对象  //用Student类来创建一个Student的对象  //stu是对象名，也称为对象的句柄，也称为对象的引用  //第一个学生对象  Student stu = new Student();    //2、显示属性  System.out.println("为属性赋值之前：");//有默认值  System.out.println("stu.name：" + stu.name);  System.out.println("stu.age：" + stu.age);    //3、赋值  stu.name = "邬少杰";  stu.age = 20;    //显示属性值  System.out.println("为属性赋值之后：");  System.out.println("stu.name：" + stu.name);  System.out.println("stu.age：" + stu.age);    //另一个学生对象  Student stu2 = new Student();  stu2.name = "白益坤";  stu2.age = 23;  System.out.println("另一个学生的信息：");  System.out.println("stu2.name：" + stu2.name);  System.out.println("stu2.age：" + stu2.age);    System.out.println("第一个学生的信息：");//不会受到stu2的影响  System.out.println("stu.name：" + stu.name);  System.out.println("stu.age：" + stu.age);  }  }  //在TestField类外面声明  class Student{  String name;  int age;  } |

### 4、练习

#### （1）练习1



|  |
| --- |
| 举例:  public class Zoo{  public static void main(String args[]){  Animal xb=new Animal();  Animal xh=new Animal();  xb.legs=4;  xh.legs=0;  System.out.println(xb.legs); //4  System.out.println(xh.legs); //0  xb.legs=2;  System.out.println(xb.legs); //2  System.out.println(xh.legs); //0  }  } |

#### （2）练习2

|  |
| --- |
| //1、声明一个圆类Circle  //2、圆类中有一个属性：半径 radius  //3、创建两个圆对象，一个半径是1.2，一个半径是2.3  //4、打印显示两个圆的半径值  //5、根据半径求面积，并显示  //提示：面积 = Math.PI \* 半径 \* 半径;  public class TestCircle{  //程序的入口  public static void main(String[] args){  //3、创建两个圆对象，一个半径是1.2，一个半径是2.3  Circle c1 = new Circle();  c1.radius = 1.2;    Circle c2 = new Circle();  c2.radius = 2.3;    //4、打印显示两个圆的半径值  System.out.println("c1的半径为：" + c1.radius);  System.out.println("c2的半径为：" + c2.radius);    //5、根据半径求面积，并显示  double area1 = Math.PI \* c1.radius \* c1.radius;  double area2 = Math.PI \* c2.radius \* c2.radius;    System.out.println("c1的面积是：" + area1);  System.out.println("c2的面积是：" + area2);  }  }  //1、声明一个圆类Circle  class Circle{  //2，圆类中有一个属性：半径 radius  double radius;  } |

### 5、属性的访问机制

#### （1）本类中

本类中的方法可以直接访问本类中的成员变量。

**特殊**：static成员访问非static成员，编译不通过

#### （2）跨类

如果要访问其他类的成员，那么必须要通过“对象.成员”

用户使用new操作符创建一个对象后，可以使用“对象.成员”来获取对象的属性和行为。

例如： Teacher tea = new Teacher(); tea.name = “柴林燕”; tea.teacher();

**特殊说明：**如果被访问的类成员是static修饰的，那么可以使用“类名.类成员”访问的简化形式

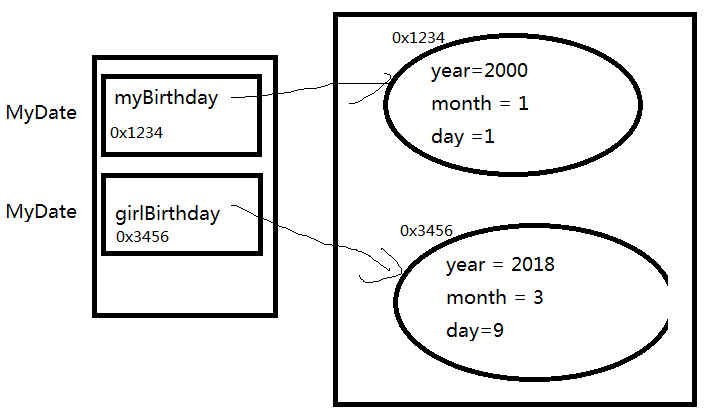
**注意：跨类还要考虑访问权限的问题**

|  |
| --- |
| public class TestAccess{  public static void main(String[] args){  Circle c1 = new Circle();  c1.radius = 1.2;//跨类，那么必须要通过“对象.成员”    Circle c2 = new Circle();  c2.radius = 2.3;//跨类，那么必须要通过“对象.成员”    //如果要访问其他类的成员，那么必须要通过“对象.成员”  System.out.println("c1的半径为：" + c1.radius);  System.out.println("c1的面积是：" + c1.getArea());    System.out.println("c2的半径为：" + c2.radius);  System.out.println("c1的面积是：" + c2.getArea());  }  }  class Circle{  double radius;    //本类中的方法可以直接访问本类中的成员变量  public double getArea(){  return Math.PI \* radius \* radius;  }  } |

### 6、Java的内存分析

#### （1）对象的内存分析图

|  |
| --- |
| /\*  (1)声明一个日期类型MyDate：有属性：年year,月month，日day  （2）创建2个日期对象，分别赋值为：你的出生日期，你对象的出生日期  （3）并显示信息  \*/  public class TestMyDate{  public static void main(String[] args){  MyDate myBirthday = new MyDate();  myBirthday.year = 2000;  myBirthday.month = 1;  myBirthday.day = 1;    System.out.println("我的生日是：" + myBirthday.year + "年" + myBirthday.month + "月" + myBirthday.day + "日");    MyDate girlBirthday = new MyDate();  girlBirthday.year = 2018;  girlBirthday.month = 3;  girlBirthday.day = 9;    System.out.println("女朋友的生日是：" + girlBirthday.year + "年" + girlBirthday.month + "月" + girlBirthday.day + "日");  }  }  class MyDate{  int year;  int month;  int day;  } |

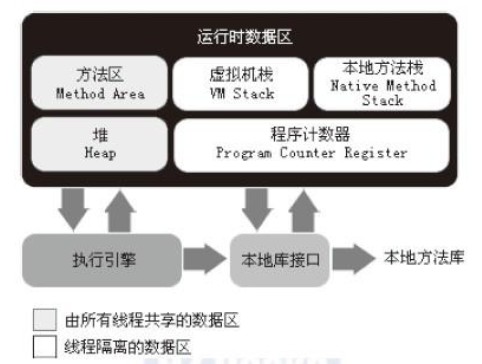


1. new对象，等价于在堆中申请一块空间，用来存储该对象的专属信息
2. 对象的引用（对象名），等价于将该对象的堆内存的首地址存储在引用变量（对象名）中

#### （2）Java的内存管理（暂时了解）

经常人把Java内存区分为堆内存（Heap）和栈内存（Stack），这种分法比较粗糙，Java内存区域的划分实际上远比这复杂。这种划分方式的流行是因为大多数程序员最关注的、与对象内存分配关系最密切的内存区域是这两块。

根据《Java虚拟机规范（第2版）》的规定，Java虚拟机所管理的内存将会包括以下几个运行时数据区域，如下图所示：



堆（Heap），此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。这一点在Java虚拟机规范中的描述是：所有的对象实例以及数组都要在堆上分配。

通常所说的栈（Stack），是指虚拟机栈。虚拟机栈用于存储局部变量表等。局部变量表存放了编译期可知长度的各种基本数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference类型，它不等同于对象本身，是对象在堆内存的首地址）。 方法执行完，自动释放。

方法区（Method Area）用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

#### （3）Java的内存管理之垃圾回收（了解）

分配：由JVM自动为其分配相应的内存空间

释放：由JVM提供垃圾回收机制自动的释放内存空间

垃圾回收机制（GC：Garbage Collection）：将垃圾对象所占用的堆内存进行回收。Java的垃圾回收机制是JVM提供的能力，由单独的系统级垃圾回收线程在空闲时间以不定时的方式动态回收。

**垃圾对象**：不再被任何引用指向的对象。

面试题：

问在程序中是否可以通知垃圾回收机制过来回收垃圾？

能，通过调用System.gc();或Runtime.getRuntime().gc();

再问？调用了System.gc();或Runtime.getRuntime().gc();后是立刻执行垃圾回收吗？

不是，该调用并不会立刻启动垃圾回收机制开始回收，但会加快垃圾回收机制的运行。

|  |
| --- |
| public class TestGC{  public static void main(String[] args)throws Exception{  for(int i=0; i<10; i++){  MyClass m = new MyClass();//这里本次循环完，本次创建的对象就成为垃圾了  System.out.println("创建第" + (i+1) + "的对象：" + m);  }    //通知垃圾回收机制来收集垃圾  System.gc();    //为了延缓程序结束  for(int i=0; i<10; i++){  Thread.sleep(1);  System.out.println("程序在继续....");  }  }  }  class MyClass{  //这个方法是垃圾回收机制在回收它的对象时，自动调用，理解成对象留临终遗言的方法  public void finalize(){  System.out.println("轻轻的我走了.....");  }  } |

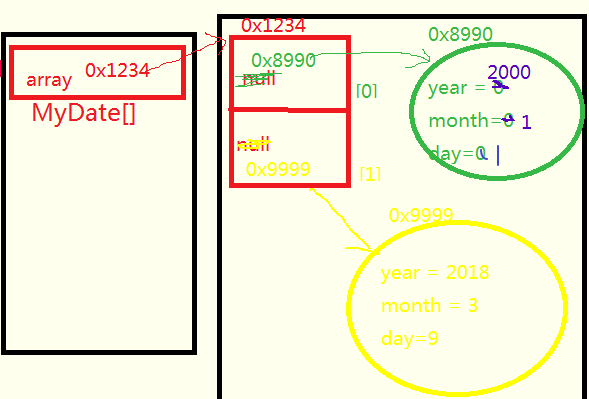
### 7、面试题：实例变量与局部变量的区别？

1. 声明的位置不同：实例变量在类中方法外，局部变量在方法内或代码块内
2. 数据的存储位置不同：实例变量在堆中，局部变量在栈中
3. 初始值的获得不同：实例变量有默认值，局部变量必须手动初始化
4. 修饰符：实例变量可以多种修饰符，局部变量最多只能使用final
5. 作用域：实例变量（本类中直接访问，其他类通过对象访问），局部变量从声明处开始，到所属的}就结束
6. 生命周期：实例变量随着对象的创建而出生，对象被垃圾回收机制收回才被收回。局部变量是当代码执行超过作用域范围，该局部变量就失效。

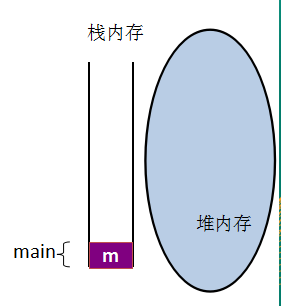
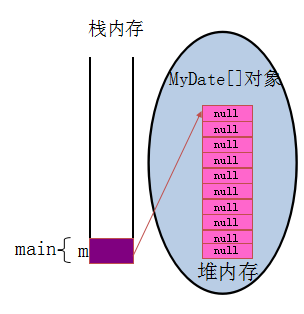
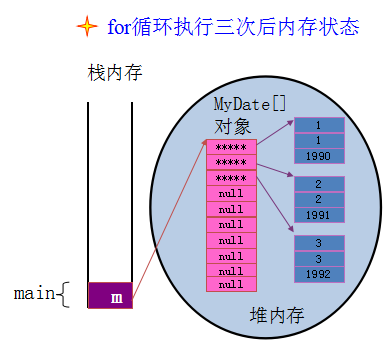
### 8、对象数组

元素类型是引用数据类型的数组称为对象数组。因为元素中存储的是对象。

|  |
| --- |
| /\*  (1)声明一个日期类型MyDate：有属性：年year,月month，日day  （2）创建2个日期对象，分别赋值为：你的出生日期，你对象的出生日期，放到数组中  （3）遍历显示  \*/  public class TestMyDateArray{  public static void main(String[] args){  MyDate[] array = new MyDate[2];    array[0] = new MyDate();  array[0].year = 2000;  array[0].month = 1;  array[0].day = 1;    System.out.println(array[0]);  System.out.println("我的生日是：" + array[0].year + "年" + array[0].month + "月" + array[0].day + "日");    array[1] = new MyDate();  array[1].year = 2018;  array[1].month = 3;  array[1].day = 9;    System.out.println(array[1]);  System.out.println("女朋友的生日是：" + array[1].year + "年" + array[1].month + "月" + array[1].day + "日");  }  }  class MyDate{  int year;  int month;  int day;  } |



|  |
| --- |
| MyDate[] m; //MyDate是自定义类型，好比Student  m = new MyDate[10]; //引用类型的元素的默认值null  for ( int i=0; i<10; i++ ) {  m[i] =new MyDate(); //每一个元素又new对象  m[i].year = 1990 + i;  m[i].month = i + 1;  m[i].day = i + 1;  } |

### 9、练习

#### 练习1

声明一个矩形，属性：长、宽，在测试类中创建矩形的对象，并显示长、宽的值及其面积。

#### 练习2

（1）声明一个学生类Student，包含姓名name和成绩score属性

（2）在测试类TestStudentArray中创建一个长度为5的学生数组

（3）从键盘输入为学生对象的姓名和成绩属性赋值

（4）遍历显示学生信息

（5）使用冒泡排序对数组按照学生成绩从低到高排序

（6）再次遍历显示学生信息

|  |
| --- |
| /\*  对象数组：数组中存的是对象，数组的元素类型是引用数据类型  int[] 的元素是int类型  String[] 的元素是String类型  Student[] 的元素是Student类型  （1）声明一个学生类Student，包含姓名name和成绩score属性  （2）在测试类TestStudentArray中创建一个长度为5的学生数组  （3）从键盘输入为学生对象的姓名和成绩属性赋值  （4）遍历显示学生信息  （5）使用冒泡排序对数组按照学生成绩从低到高排序  （6）再次遍历显示学生信息  \*/  import java.util.Scanner;  //（1）声明一个学生类Student，包含姓名name和成绩score属性  class Student{  String name;  int score;  }  class TestStudentArray{  public static void main(String[] args){  //在测试类TestStudentArray中创建一个长度为5的学生数组  //元素的数据类型[] 数组名 = new 元素的数据类型[长度];  Student[] array = new Student[3];    Scanner input = new Scanner(System.in);    //从键盘输入为学生对象的姓名和成绩属性赋值  for(int i=0; i<array.length; i++){  //（1）先创建元素的对象，即学生对象  array[i] = new Student();    //(2)为学生对象的属性赋值  System.out.print("请输入第" + (i+1) + "个学生的姓名：");  array[i].name = input.next();  System.out.print("请输入第" + (i+1) + "个学生的成绩：");  array[i].score = input.nextInt();  }    //遍历显示学生信息，使用for  /\*  for(int i=0; i<array.length; i++){  System.out.println("第" + (i+1) + "个学生的姓名：" + array[i].name);  System.out.println("第" + (i+1) + "个学生的成绩：" + array[i].score);  }  \*/    //foreach  //for(元素的数据类型 元素名 ： 数组名){}  for(Student s : array){  System.out.println("学生的姓名：" + s.name);  System.out.println("学生的成绩：" + s.score);  }    //排序：从小到大，每轮从左边开始把大往右边移动  for(int i=1; i<array.length; i++){  //i=1, j=0,1,2,3 j<4 j<array.length-i  //i=2, j=0,1,2 j<3 j<array.length-i  for(int j=0; j<array.length-i; j++){  //前一个元素比后一个元素大，就交换  //前一个学生的成绩，比后一个学生的成绩高，就交换  if( array[j].score > array[j+1].score ){  //交换两个学生，交换两个数组的元素  Student temp = array[j];  array[j] = array[j+1];  array[j+1] = temp;  }  }  }    System.out.println("排序后：" );  for(Student s : array){  System.out.println("学生的姓名：" + s.name);  System.out.println("学生的成绩：" + s.score);  }  }  } |

## 六、类的成员之二：方法

### 1、方法的概念

方法也叫函数，是一个独立功能的定义，是一个类中最基本的功能单元。

把一个功能封装为方法的目的是，可以实现代码重用，从而简化代码。

方法的使用原则：

（1）必须先声明后使用

类，变量，方法都要先声明后使用

（2）不调用不执行，调用一次执行一次。

### 2、方法的声明格式

**（1）方法声明的位置**

类中方法外

**（2）方法的声明格式**

【修饰符】 返回值类型 方法名(【形参列表】){

//方法体：实现功能的语句

}

说明：

* 方法名：代表功能，见名知意，好的方法名，看到方法，就能知道方法的功能是什么
* 修饰符：方法的修饰符很多，将会一一学习，例如：public，static等
* 返回值类型：说明方法运行后有结果返回，那个结果的数据类型。如果无需返回结果，只能写void
  + return语句

（1）return; 用于提前结束方法。用于返回值类型是void的方法中，可选。

（2）return 返回值; 用于结束方法并返回结果。用于返回值类型不是void的方法中，必选。return后面返回值的数据类型与方法签名中的返回值类型必须一致或兼容

* 参数列表：属于方法的局部变量，可以有0~n个，每个参数需要指明“数据类型 形参名”，数据类型与形参名之间用“空格”分割，而多个参数之间使用“,”分割。形参的值，在方法被调用时由实参决定。

**（3）声明方法前需要明确的几点**

* 方法的功能是什么，取个“好名”

原则：方法名要见名知意，而且一个方法实现一个独立的完整的功能

例如Math数学工具类：

求平方根：sqrt()

获取随机数：random()

数组工具类：

排序：sort()

复制：copyOf()

copyOfRange()

* 明确是否需要返回结果给调用者

明确是否需要返回值类型，返回值类型是什么

原则：尽量让方法中最后的输出改装成返回值

* 明确是否有对象属性以外的未知的数据参与运算

明确是否需要参数，需要几个参数，分别是什么类型的

原则：尽量让方法中输入的数据改装成参数

* 明确是否该方法的访问有什么限制和要求，即考虑修饰符（后续讲解）
* 明确是否该方法会抛出什么异常（后续讲解）

### 3、方法的调用格式

方法只有**被调用才会执行**。

调用规则：

（1）调用方法通过方法名调用。

* 本类中

本类中的方法可以直接访问本类中的成员方法。

**特殊：**static方法访问非static方法，编译不通过

* 跨类

如果要访问其他类的成员方法，那么必须要通过“对象.成员方法”

**特殊说明：**如果被访问的方法是static修饰的，那么可以使用“类名.方法”访问的简化形式

**注意：**跨类还要考虑访问权限的问题

（2）如果被调用方法有形参列表，那么必须在调用方法时传递实参列表，实参列表不需要写数据类型，实参的个数、类型与形参列表的顺序要一一对应。

（3）若被调用方法声明了“返回值类型”，说明该方法运行后有结果返回，若调用者需要用到这个结果，可以用一个变量接收，或把该方法调用作为表达式直接使用，或者作为另一个方法调用的实参；

如果被调用方法声明了void，说明该方法运行后没有结果返回，那么该调用语句**只能**单独作为一个语句。

### 4、方法的四种形式

根据有参无参，有返回值无返回值，排列组合后方法有4种情形：

（1）无返回值无参数

（2）有返回值无参数

（3）无返回值有参数

（4）有返回值有参数

|  |
| --- |
| class Circle{  double radius;    //封装一个求面积的方法  //或者说，圆这个类别的对象，就应该有获取面积的功能  double getArea(){  double area = Math.PI \* radius \* radius;  return area;  }    //封装一个求周长的方法  double getZhouChang(){  return 2 \* Math.PI \* radius;  }    void printInfo(){  System.out.println(“半径：” + radius +”，面积是：” + getArea() + “，周长：” + getZhouChang());  }  }  class TestMethod{  public static void main(String[] args){  //1、先创建对象  Circle c = new Circle();    //2、获取c对象的面积  double area = c.getArea();  System.out.println("area=" + area);    //3、为属性赋值  c.radius = 3.0;  System.out.println("面积是：" + c.getArea());  System.out.println("周长是：" + c.getZhouChang());  }  } |
| /\*  定义银行账户类Account，有属性：卡号cid，余额balance  银行账户类Account有方法：  （1）getInfo()，返回String类型，返回卡的详细信息  （2）取钱方法withdraw()，参数自行设计，如果取钱成功返回true，失败返回false  （3）存钱方法save(),参数自行设计，如果存钱成功返回true,失败返回false  \*/  **public** **class** TestMethod {  **public** **static** **void** main(String[] args) {  Account a = **new** Account();  a.cid = "11223344";  a.balance = 1000;    System.***out***.println(a.getInfo());    **boolean** flag = a.withdraw(2000);  System.***out***.println(flag?"取钱成功":"取钱失败");  System.***out***.println(a.getInfo());    **boolean** result = a.deposit(1000);  System.***out***.println(result?"存钱成功":"存钱失败");  System.***out***.println(a.getInfo());  }  }  **class** Account{  String cid;  **double** balance;    **boolean** withdraw(**double** money){  **if**(money<0 || money > balance){  **return** **false**;  }  balance -= money;  **return** **true**;  }    **boolean** deposit(**double** money){  **if**(money<0){  **return** **false**;  }  balance += money;  **return** **true**;  }    String getInfo(){  **return** "账户：" + cid + "，余额：" + balance;  }  } |

### 6、方法参数的值传递机制

方法的参数传递机制：

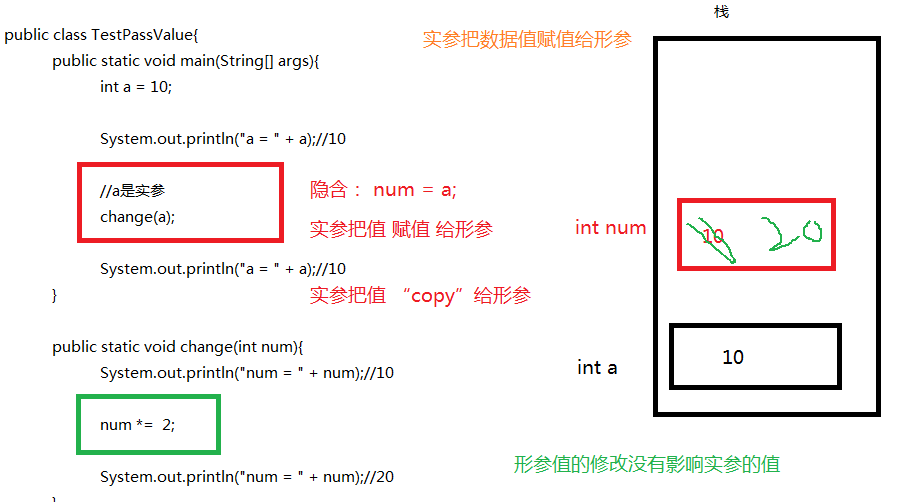
方法的值传递机制：实参给形参传值

（1）基本数据类型形参：实参给形参传的是“数据值”，形参值的修改不会影响实参

（2）引用数据类型形参：实参给形参传的是“地址值”，形参对象的属性值修改会影响实参对象的属性值。

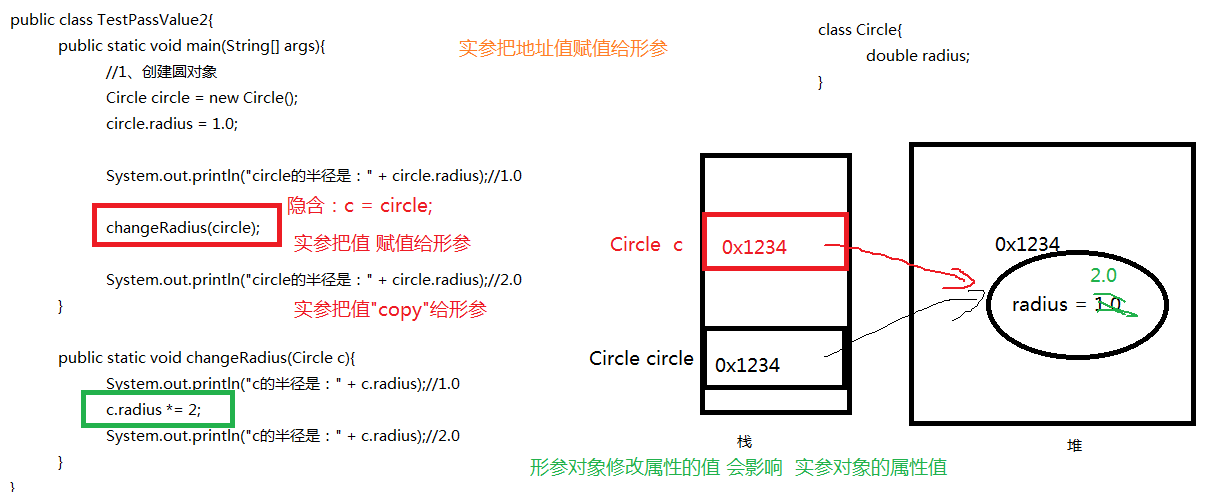
#### 1、基本数据类型

|  |
| --- |
| public class TestPassValue{  public static void main(String[] args){  int a = 10;    System.out.println("修改之前a = " + a);//10    //a是实参  change(a);    System.out.println("修改之后a = " + a);//10  }    //方法的功能：修改一个变量的值，为它原来的两倍  //修饰符 返回值类型 方法名（形参列表）{}  //修饰符：public static，和main在同一个类中，为了在main()中直接调用，这里加上public static  //int num是形参  public static void change(int num){  System.out.println("修改之前num = " + num);//10    num \*= 2;    System.out.println("修改之后num = " + num);//20  }  } |
| 修改之前a = 10  修改之前num = 10  修改之后num = 20  修改之后a = 10 |



#### 2、引用数据类型

|  |
| --- |
| /\*  Java的数据类型：  （1）基本数据类型（8种）  （2）引用数据类型：  类、接口、数组  类的类型：Student,Circle,String等  String name = "张三";  Circle circle = new Circle();    \*/  public class TestPassValue2{  public static void main(String[] args){  //1、创建圆对象  Circle circle = new Circle();  circle.radius = 1.0;    System.out.println("circle的半径是：" + circle.radius);//1.0    //2、调用changeRadius方法，修改半径的值  //实参要与形参的类型，个数，顺序一致  changeRadius(circle);    System.out.println("circle的半径是：" + circle.radius);//2.0  }    //把c这个圆的半径修改为原来的两倍  //形参的格式：数据类型 形参名  public static void changeRadius(Circle c){  System.out.println("c的半径是：" + c.radius);//1.0  c.radius \*= 2;  System.out.println("c的半径是：" + c.radius);//2.0  }  }  class Circle{  double radius;  } |



#### 3、练习

练习1：

|  |
| --- |
| public class PassValueExer{  public static void main(String[] args){  int m = 1;  int n = 2;    swap(m,n);//要求，调用该方法后，使得打印结果是m=2,n=1    System.out.println("m = " + m);  System.out.println("n = " + n);  }    //在此处补充代码完成....  } |

分析：

|  |  |
| --- | --- |
| public class TestPassValue {  public static void main(String[] args) {  int a = 1;  int b = 2;  System.out.println("交换之前：a = " + a + ",b = " + b);  swap(a,b);  System.out.println("交换之后：a = " + a + ",b = " + b);  }  public static void swap(int x, int y) {  System.out.println("交换之前：x = " + x + ",y = " + y);  int temp = x;  x = y;  y = temp;  System.out.println("交换之后：x = " + x + ",y = " + y);  }  } |  |
| 交换之前：a = 1,b = 2  交换之前：x = 1,y = 2  交换之后：x = 2,y = 1  交换之后：a = 1,b = 2 | |

答案：

|  |
| --- |
| /\*  要求：使得打印结果是m=2,n=1  考点：  （1）方法的值传递机制  （2）转弯  答案一：机智  答案二：等待揭晓  \*/  public class PassValueExer{  public static void main(String[] args){  int m = 1;  int n = 2;    swap(m,n);//要求，调用该方法后，使得打印结果是m=2,n=1    System.out.println("m = " + m);  System.out.println("n = " + n);  }    //方法的功能，交换两个变量的值  public static void swap(int m, int n){    int temp = m;  m = n;  n = temp;    System.out.println("m = " + m);  System.out.println("n = " + n);  System.exit(0);  }  } |

练习2

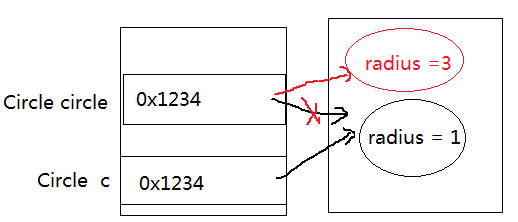
|  |
| --- |
| import java.util.Arrays;  public class PassValueExer2{  public static void main(String[] args){  int[] array = {3,2,5,1,7};    //调用sort方法，实现从大到小排序  //在此处补充代码  ....    //显示结果  System.out.println("排序后的结果是：" + Arrays.toString(array));  }    //要求使用冒泡排序完成  public static void sort(//形参？){    }  } |

答案：

|  |
| --- |
| /\*  考点：  1、方法的参数传递机制  2、冒泡排序  \*/  import java.util.Arrays;  public class PassValueExer2{  public static void main(String[] args){  int[] array = {3,2,5,1,7};    //调用sort方法，实现排序  sort(array);//实参给形参的是地址，数组的首地址    //遍历结果  System.out.println("排序后的结果是：" + Arrays.toString(array));  }    //功能：用冒泡排序，实现为数组排序，而且从大到小  //形参的类型？我要把什么传过来  //传递数组  //接收的类型也肯定是数组，即形参接收实参，即形参的类型是int[]  public static void sort(int[] arr){  //冒泡排序  //在这里对谁排，对arr进行排序  for(int i=1; i<arr.length; i++){//多少轮  //每一轮，从左往后--》for(int j=0;...)  //要实现从大到小-->前面的元素比后面的元素小，就交换  //每一轮几次，  //假设，数组的长度为5  //第一轮：4次，i=1, j=0,1,2,3 j<4 j< arr.length-i  //第二轮：3次，i=2, j=0,1,2  for(int j=0; j<arr.length-i; j++){  if(arr[j] < arr[j+1]){  int temp = arr[j];  arr[j] = arr[j+1];  arr[j+1] = temp;  }  }  }  }  } |

#### 4、错误示范

|  |
| --- |
| public class TestParamValue {  public static void main(String[] args) {  Circle c = new Circle();  c.radius = 1.0;  System.out.println("修改之前圆c的半径是：" + c.radius);  change(c);  System.out.println("修改之后圆c的半径是：" + c.radius);  }  public static void change(Circle circle){  System.out.println("修改之前圆circle的半径是：" + circle.radius);  **circle = new Circle();//谨慎**  circle.radius = 3.0;  System.out.println("修改之前圆circle的半径是：" + circle.radius);  }  }  class Circle{  double radius;  } |
| 修改之前圆c的半径是：1.0  修改之前圆circle的半径是：1.0  修改之前圆circle的半径是：3.0  修改之后圆c的半径是：1.0 |



### 7、方法重载（Overload）

方法的重载：在同一个类中，出现的多个方法名称相同，参数列表不同的情形称为方法重载。

关键词：

（1）在同一个类中多个方法

（2）方法名称必须相同

（3）参数列表必须不同（参数的个数或数据类型不同）

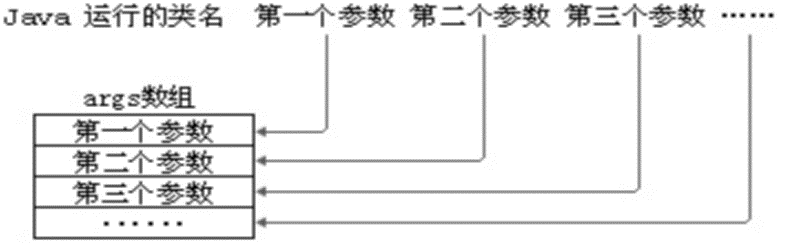
注意：与返回值类型无关

|  |
| --- |
| /\*  面试题：Overload和Override的区别？  方法的重载：Overload  在同一个类中，出现了方法名称相同，方法的形参列表不同，这样的两个或多个方法，称为方法的重载。  和返回值类型无关。  关键词：（1）同一个类中 （2）方法名称相同 （3）形参列表不同：个数或类型  \*/  public class TestOverload{  public static void main(String[] args){  int x = 4;  double y = 9;    //1、创建对象  MyMath m = new MyMath();  //double max = m.getMax(x,y);  //System.out.println("x,y中的较大者是：" + max);    int r = 1;  int s = 0;  int t = 4;  int big = m.getMax(r,s,t);  System.out.println("r,s,t中的较大者是：" + big);  }  }  class MyMath{  //方法一：返回两个整数中的较大者  public int getMax(int a, int b){  System.out.println("两个整数的最大值");  //写法一：  /\*  int max = a>=b ? a : b;  return max;  \*/    return a>=b ? a : b;  }    //方法二：返回两个浮点数，小数中的较大者  public double getMax(double a, double b){  System.out.println("两个小数的最大值");  //写法二：  if(a>=b){  return a;  }else{  return b;  }  }    //方法三：返回三个整数中的最大值  public int getMax(int a, int b, int c){  System.out.println("三个整数的最大值");  /\*  int max = a>=b ? a : b;  max = max >= c ? max : c;  return max;  \*/  /\*  int max = getMax(a,b);  max = getMax(max,c);  return max;  \*/  /\*  int max = getMax(getMax(a,b),c);  return max;  \*/  return getMax(getMax(a,b),c);  }  } |

### 8、命令行参数[了解]

命令行参数：给main方法传递的参数。

由于java虚拟机需要调用类的main()方法，所以该方法的访问权限必须是public，又因为java虚拟机在执行main()方法时不必创建对象，所以该方法必须是static的，该方法接收一个String类型的数组参数，该数组中保存执行java命令时传递给所运行的类的参数。



|  |
| --- |
| /\*  命令行参数：给main方法传递的参数  运行Java程序：  java 类名 实参1 实参2 实参3...  用空格分隔  例如：  java TestCommandParam  java TestCommandParam chai lin yan  是否可以传整数？  java TestCommandParam 1 2 3  \*/  public class TestCommandParam{  //形参：String[] args  public static void main(String[] args){  //System.out.println(args);  //System.out.println(args.length);    for(int i=0; i<args.length; i++){  System.out.println("第" + (i+1) + "个参数的值是：" + args[i]);  }    //求和  int sum = 0;  for(int i=0; i<args.length; i++){  //sum += args[i];//编译错误，args[i]是String类型  sum += Integer.parseInt(args[i]);//把String转成int  }  System.out.println("sum = " + sum);  }  } |

### 9、可变参数[掌握]

可变参数：在声明方法的形参列表时，如果出现这种情况，需要在调用这个方法时，

某个参数由调用者决定需由调用者传递0~n个实参，那么这个时候就需要把这个形参设计为可变参数。

可变：参数的个数

可变参数：

1、如何声明

数据类型... 可变参数名

要求：

（1）一个方法只能有一个可变参数

（2）可变参数的位置一定是最后一个

2、如何使用它

（1）在声明它的方法中，把它当做数组使用

（2）在调用含可变参数的方法

A：非可变参数的部分，实参的类型，个数，顺序与形参仍然要一一对应

B：可变参数的部分，实参的类型要与形参一致或兼容，个数可以是0~n个

C：可变参数的部分，实参还可以是对应类型的数组

|  |
| --- |
| public class VarParams{  public static void main(String[] args){  System.out.println("3或5的最大值是：" + getMax(3,5));//实参是3和5, 3给了num， 5给了args  System.out.println("3,5,1,7,9的最大值是：" + getMax(3,5,1,7,9));//3,5,1,7,9是实参， 3给了num，5,1,7,9给了args  System.out.println("3的最大值是：" + getMax(3));//3是实参, 3给了num，args没有对应的实参  //System.out.println("3的最大值是：" + getMax());//编译不通过    int[] array = {4,5,6,7};  System.out.println("3和{4,5,6,7}的最大值是：" + getMax(3,array));//3给了num，array给了args  }    //方法的功能：可以求1~n个整数中的最大值  //思考：把int... args修改为int[] args会有什么不同  public static int getMax(int num, int... args){  //args是数组，等价于int[]  //先假设num最大，然后与args中的元素一一比较，如果有比它大的，就修改结果  int max = num;    for(int i=0; i<args.length; i++){  if(args[i] > max){  max = args[i];  }  }    return max;  }    } |
| **public** **class** TestVarParams {  **public** **static** **void** main(String[] args) {  Game g = **new** Game();  // g.tellMe("张三", 23, '男');  g.tellMe("李四", 24, '男', "范冰冰", "如花");  }  }  **class** Game {  **public** **void** tellMe(String name, **int** age, **char** gender, String... args) {  System.***out***.println("你的名字是：" + name);  System.***out***.println("你今年：" + age + "岁了");  **if** (gender == '男') {  System.***out***.println("你是一位帅哥.");  } **else** {  System.***out***.println("你是一位美女.");  }  **if** (args.length == 0) {  System.***out***.println("你还没谈过恋爱");  } **else** {  System.***out***.println("你的前任们有：");  **for** (String string : args) {  System.***out***.println(string);  }  }  }  } |

3、陷阱：

（1）这两个方法不是重载方法，编译器认为是同一个。因为在getSum方法中，它俩都是按照数组处理的。

|  |
| --- |
| **public** **static** **int** getSum(**int**[] array){  **int** sum=0;  **for**(**int** i=0;i<array.length;i++){  sum+=array[i];  }  **return** sum;  }  **public** **static** **int** getSum(**int**... nums){  **int** sum=0;  //编译器把nums编译成int[]  **for**(**int** i=0;i<nums.length;i++){  sum+=nums[i];  }  **return** sum;  } |

但是它俩在方法调用时是不一样的。上面的方法调用时，必须传递一个int[]的实参数组，而下面的方法，可以传递一个int[]的实参数组，也可以传0~n个整数值。

（2）这两个方法是重载，但是没法用

|  |
| --- |
| **public** **static** **void** main(String[] args) {  System.***out***.println(*getSum*(1,2,3,6,7));  System.***out***.println(*getSum*(1,2,3));  }  **public** **static** **int** getSum(**int**... nums){  **return** 0;  }  **public** **static** **int** getSum(**int** a,**int**... nums){  **return** 0;  } |

4、说明：关于可变参数的方法重载

|  |
| --- |
| **public** **class** Test {  **public** **static** **void** main(String[] args) {  *method*();  *method*(1);//这个会调用一个参数的method方法，其他的都调用可变参数  *method*(1,2);  *method*(1,2,3);  }  **public** **static** **void** method(**int** a){  System.***out***.println("一个参数");  }  **public** **static** **void** method(**int**... a){  System.***out***.println("可变参数");  }  } |

## 七、练习

1、编写一个Student类，包含name、gender、age、id、score属性，分别为String、String、int、int、double类型。

类中声明一个say方法，返回String类型，方法返回信息中包含所有属性值。

在另一个TestStudent类中的main方法中，创建Student对象，并访问say方法和所有属性，并将调用结果打印输出。

2、将对象作为参数传递给方法。题目要求：

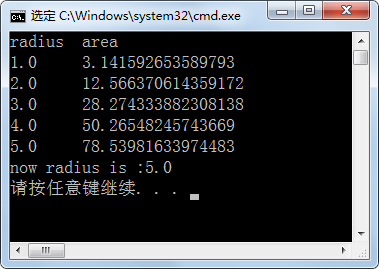
（1）定义一个Circle类，包含一个double型的radius属性代表圆的半径，一个getArea()方法返回圆的面积。

（2）定义一个类PassObject，在类中定义一个方法printAreas()，该方法的定义如下：

**public void printAreas(Circle c, int times)**

在printAreas方法中依次将Circle对象的半径设置为1到times之间的每个整数半径值，并求出对应的面积。例如，times为5，则输出半径1，2，3，4，5，以及对应的圆面积。

（3）在main方法中创建Circle对象，并调用printAreas()方法，调用完毕后输出圆当前半径值。程序运行结果如图所示。



3、定义一个丈夫Husband类，有姓名、年龄、妻子属性

定义一个妻子Wife类，有姓名、年龄、丈夫属性

丈夫类中有一个getInfo方法，其中，能显示自己的姓名，年龄，和他的妻子的姓名，年龄

妻子类中有一个getInfo方法，其中，能显示自己的姓名，年龄，和她的丈夫的姓名，年龄

定义一个测试类，创建妻子和丈夫对象，然后测试

4、定义银行账户类Account，有属性：卡号cid，余额balance，所属用户Customer

银行账户类Account有方法：

（1）getInfo()，返回String类型，返回卡的详细信息

（2）取钱方法withdraw()，参数自行设计，如果取钱成功返回true，失败返回false

（3）存钱方法save(),参数自行设计，如果存钱成功返回true,失败返回false

其中Customer类有姓名、身份证号、联系电话、家庭地址等属性

Customer类有方法say()，返回String类型，返回他的个人信息。

在测试类Bank中创建银行账户类对象和用户类对象，并设置信息，与显示信息

# 第六章 面向对象基本特征

## 一、面向对象的基本特征之一：封装Encapsulation

封装是面向对象编程的核心思想。将对象的属性和行为封装起来，其载体就是类。类通常对客户隐藏其实现细节，这就是封装的思想。

封装（Encapsulation）：把该隐藏的隐藏起来，该暴露的暴露出来。

封装需要通过访问控制修饰符（权限修饰符）来实现。

* public：公共的，可用于修饰类、属性、方法。表示在其他任何想要使用的地方都可见。
* protected：受保护的，可用于修饰属性、方法。表示在本包或其他包的子类中可见。
* 缺省的：可用于修饰类、属性、方法。表示在本包下可见。
* private：私有的，可用于修饰属性、方法。只能在本类中可见，本类以外的地方不可见。



封装的好处：安全、简单、将变化隔离、重用

**属性的封装：**

1. 属性私有化（private）
2. 提供公共的（public）的getter/setter方法

## 二、类的成员之三：构造器

构造器，也称为构造方法，是类的成员之一。

### 1、格式

[访问控制修饰符] 类名(){

}

或

[访问控制修饰符] 类名（形参列表）{

}

### 2、作用

1. 与new一起使用创建对象
2. 为对象属性进行初始化

### 3、特殊

1. 构造器的名称必须与类名相同
2. 构造器没有返回值类型（思考：是否可以使用return语句，如果可以，该怎么写？）
3. 若一个类中没有显式的提供任何构造器时，系统会为其提供一个默认的无参构造器。
4. 若一个类中显式声明了任何构造器，那么系统将不再为其提供默认的无参构造器。
5. 一个类中可以重载多个构造器（重载的多个构造器名称都与类名相同，形参列表必须各不相同）

### 4、为属性赋初始值的方式

1. 使用默认值
2. 显式初始化
3. 构造器初始化

注意：如果三者同时存在，那么按照（1）（2）（3）的顺序进行

setter方法是为属性修改值用的。

## 三、this关键字

this表示当前对象

1. 在方法内部使用，即表示调用该方法的对象
2. 在构造方法中使用，即表示正在初始化的对象

用法：

* this.属性

1. 当构造器或方法中有局部变量与实例变量同名时，用于区分局部变量和实例变量。

格式：this.属性

（2）当构造器或方法中没有局部变量与实例变量同名时，可以不加this.

* this.方法

表示调用当前对象的其他方法，通常情况下，省略this.

* this()或this(实参列表)

用于调用本类的其他构造方法，必须写在构造器的首行，要避免递归调用

格式：this(); 或 this(实参列表);

补充：使用this访问属性和方法时，如果在本类中未找到，会从父类中查找（从父类继承的当做自己的一样使用）。

## 四、包

### 1、包的作用

（1）避免类的重名

（2）用于控制访问权限

（3）用于划分项目的结构层次，通常将功能相近的类划分到同一个包中

### 2、package

用于确定当前源文件中的类的位置

写在.java源文件可执行代码的首行

包的命名规范：所有字母都小写xxx.yyy.zzz形式。（通常以所在公司域名的倒置，例如：com.atguigu.bean）

每个“.”代表一层目录

### 3、import

作用：用于确定需要引入的类的位置，使得在代码中使用其他包的类可以使用简写名，而不用全名称

属龙：

* 写在package和class声明之间
* import语句可以有多个，并排列出
* import java.util.\*; “\*”表示导入该util包下所有类或接口，不包括子包。
* 若在一个类中使用了相同类名不同包名的两个类时。如java.util.Date和java.sql.Date。选择其中一个使用import，另一个使用全限定类名的方式：java.sql.Date date = new java.sql.Date（1234）;
* JDK1.5之后引入了静态导入：import static 包.类.静态成员; 或 import static 包.类.\*; 使得在本源文件中使用该静态成员时可以使用简写名（静态成员名）不需要使用全名称（包.类名.静态成员名）。

### 4、编译时使用-d选项

javac命令提供了一个-d 选项，可用来自动创建class文件所在包的各级子目录。

运行格式：

javac -d <包目录结构的起始位置> <源文件名.java>

例如：

javac -d . Flyer.java Pet.java

javac -d . Bird.java

javac -d . TestAbstractInterface.java

### 5、使用jar命令打包应用程序

可使用jar命令将应用程序的所有类打包为一个.jar文件以方便部署应用程序

打包过程如下：

创建文本文件，在其中指明主类。例如tempfile.txt：

Main-Class: 包含main方法的class的全名 此处只写类名，不用写.class

<换行> 此处必须有一个空行

例如：



在应用程序相对于包的根目录中，执行jar命令打包程序，例如：

jar cmf tempfile.txt MyProgram.jar ./

可以在命令行执行.jar类型打包应用程序。例如：

java -jar MyProgram.jar

### 6、JDK中主要的包介绍

1. java.lang----包含一些Java语言的核心类，如String、Math、Integer、System和Thread，提供常用功能。

2. java.net----包含执行与网络相关的操作的类和接口。

3. java.io ----包含能提供多种输入/输出功能的类。

4. java.util----包含一些实用工具类，如定义系统特性、接口的集合框架类、使用与日期日历相关的函数。

5. java.text----包含了一些java格式化相关的类

6. java.sql----包含了java进行JDBC数据库编程的相关类/接口

7. java.awt----包含了构成抽象窗口工具集（abstract window toolkits）的多个类，这些类被用来构建和管理应用程序的图形用户界面(GUI)。

8. java.applet----包含applet运行所需的一些类。

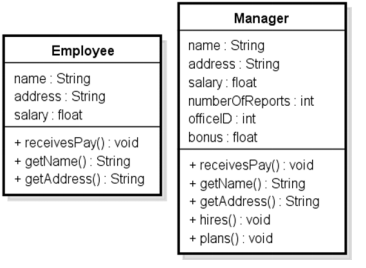
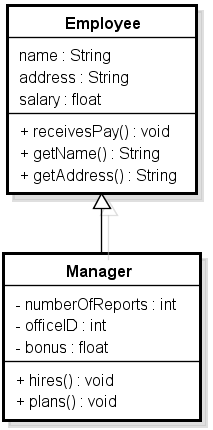
## 五、面向对象的基本特征之二：继承Inheritance

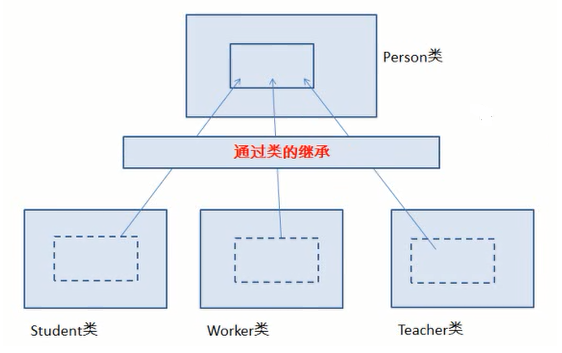
### 1、为什么要有继承？

* 当多个类中存在相同属性和行为时，将这些内容抽取到单独一个类中，那么多个类无需再定义这些属性和行为，只要继承那个类即可。提高了代码的复用性。
* 继承的出现让类与类之间产生了关系，可以创建更为特殊的类型。
* 利于可维护性。

此处的多个类称为**子类**（派生类SubClass），单独的这个类称为**父类**（基类或超类SuperClass）。

可以理解为:“子类 is a 父类”



### 2、如何继承？

类继承语法规则：class Subclass extends Superclass{ }

在Java 中，继承的关键字用的是“extends”，即子类不是父类的子集，而是对父类的“扩展”。

### 3、继承的特点

（1）子类可以继承父类所有的属性和方法，包括私有的

父类中的成员，无论是公有(public)还是私有(private)，均被子类继承。

只是因为访问控制修饰符的原因，子类不能对继承的私有成员直接进行访问，但可通过继承的公有方法来访问。



（2）子类不会继承父类的构造器，但是一定会调用父类的构造器

默认：调用父类的无参构造，super()写或不写都会调用父类的无参构造，如果要写，必须在子类构造器首行；

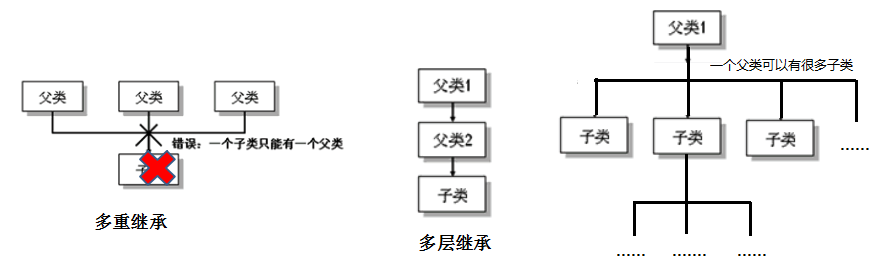
如果父类没有无参构造，必须在子类的构造器首行使用super(实参列表)显式调用父类的有参构造，否则编译报错；

注意：**在创建子类对象时，一定会先为从父类继承的属性进行初始化，所以要调用父类的构造器（☆☆☆）**

（3）子类可以扩展父类没有的属性和方法

（4）Java只支持单继承，不允许多重继承，但是Java支持多层继承

1. 一个子类只能有一个父类：只有唯一的亲生父亲
2. 一个父类可以派生出多个子类：子孙满堂
3. Java支持多层继承，代代相传
   * class SubDemo extends Demo{ } //ok
   * class SubDemo extends Demo1,Demo2...//error



### 4、重写

方法的重写（Override）:当父类中的方法对于子类来说不适用的时，子类可以对父类中的方法进行重写/覆写。

要求：两同两小一大

* 方法名：必须完全一致
* 形参列表：必须完全一致
* 返回值类型：

如果是基本数据类型和void，必须完全一致

如果是引用数据类型，必须是<=，即重写的方法的返回值类型<=被重写方法的返回值类型，Student<Person

* 修饰符：重写的方法的修饰符范围>=被重写方法的修饰符范围（public>protected>缺省>private）
* 抛出的异常：异常部分再讲

补充：

重写的方法不能是static的，final的，子类不可见的

总结：重载（Overload）与重写（Override）什么区别

（1）重载是在同一个类中，重写是子类与父类中

（2）重载与重写都要求方法的名称完全一致

（3）重载要求参数列表必须不同，重写要求参数列表必须相同

（4）重载和返回值类型无关，重写有很严格的要求

|  |  |  |
| --- | --- | --- |
| **名词** | **Overload**  **方法的重载** | **Override**  **方法的重写** |
| **位置** | 同一个类中 | 在父子类中 |
| **方法名** | 必须相同 | 必须相同 |
| **形参列表** | **必须不同** | **必须相同** |
| **返回值类型** | 无关 | <=  如果返回值类型是void和基本数据类型，那么必须一致，如果是引用数据类型，重写方法的返回值类型可以和被重写方法的返回值类型一样或是它的子类 |
| **抛出的**  **异常列表** | 无关 | <=  重写方法抛出的异常类型要么和被重写方法一样，要么是被重写方法抛出异常的子类 |
| **权限修饰符** | 无关  （建议一致） | >=  重写方法的权限修饰符要么和被重写方法一样，要么比被重写方法的权限范围大 |

### 5、演示权限修饰符的可见性



#### 1、修饰类

|  |  |
| --- | --- |
| **package** com.atguigu.pkg.one;  **class** First {  } | **package** com.atguigu.pkg.two;  /\*  \* 某个类不是公共的，那么在其他包，通过导包也是用不了的这个类，例如First，这里用不了  \*/  /\*import com.atguigu.pkg.one.First;  public class Second extends First{  }  \*/ |
| **package** com.atguigu.pkg.one;  **public** **class** One {  } | **package** com.atguigu.pkg.two;  **import** com.atguigu.pkg.one.One;  **public** **class** Two **extends** One{  } |

#### 2、修饰成员

|  |  |
| --- | --- |
| **package** com.atguigu.pkg.one;  **public** **class** One {  **private** String privateField = "privateField";  String defaultField = "defaultField";  **protected** String protectedField = "protectedField";  **public** String publicField = "publicField";  } | **package** com.atguigu.pkg.two;  **import** com.atguigu.pkg.one.One;  **public** **class** Two **extends** One{    **public** **void** printInfo(){  System.***out***.println("父类的privateField = " + getPrivateField());  System.***out***.println("父类的defaultField = " + getDefaultField());  System.***out***.println("父类的protectedField = " + protectedField);  System.***out***.println("父类的publicField = " + publicField);  }  } |

### 6、super关键字

super：父类的

用法：

* super.属性

如果子类有与父类同名的属性时，可以用super.属性明确表示是父类的属性。

* super.方法

如果子类重写了父类的方法时，可以用super.方法明确调用父类被重写的方法

* super()或super(实参列表)

super()显式调用父类的无参构造

super(实参列表)显示调用父类的有参构造

注意：

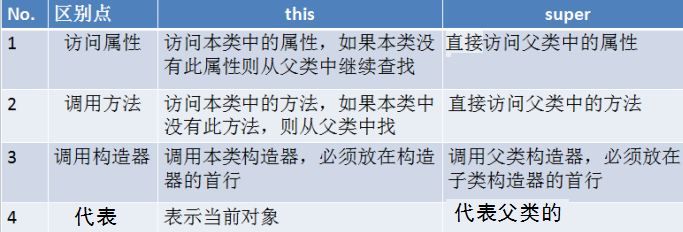
（1）通过super引用属性、方法、构造器时，都要求该成员是可见的，即该成员的修饰符不能是private的，跨包的话还不能是缺省的。

（2）super的追溯不仅限于直接父类

（3）如果某个属性或方法前面使用“this.”，那么先从本类中查找，如果未找到，会沿着父类查找

如果某个属性或方法前面使用“super.”，那么先从直接父类中查找，如果未找到，会沿着继承关系往上找

如果某个属性或方法前面既没有“this.”，也没有“super.”，遵循就近原则，开始找



|  |
| --- |
| **package** com.atguigu.pkg.one;  **public** **class** Fu {  **private** String privateField = "fu\_privateField";  String defaultField = "fu\_defaultField";  **protected** String protectedField = "fu\_protectedField";  **public** String publicField = "fu\_publicField";    **public** String getPrivateField() {  **return** privateField;  }  **public** **void** setPrivateField(String privateField) {  **this**.privateField = privateField;  }  **public** String getDefaultField() {  **return** defaultField;  }  **public** **void** setDefaultField(String defaultField) {  **this**.defaultField = defaultField;  }  **public** String getProtectedField() {  **return** protectedField;  }  **public** **void** setProtectedField(String protectedField) {  **this**.protectedField = protectedField;  }  **public** String getPublicField() {  **return** publicField;  }  **public** **void** setPublicField(String publicField) {  **this**.publicField = publicField;  }  } |
| **package** com.atguigu.pkg.two;  /\*  \* this.属性：  \* 用于当局部变量与成员变量重名时，用this.表示属性  \* this.属性，先从本类声明的属性开始查找，  \* 如果没有，看是否从父类继承了该属性（这个继承的属性一定是非private、如果是跨包，还必须是非缺省的）  \*  \*  \* this.方法：  \* this.方法表示当前对象拥有的方法，先从本类声明的方法开始查找，  \* 如果没有，看是否从父类继承了该方法（这个方法也一定是非private、如果是跨包，还必须是非缺省的）  \*  \*  \* super.属性  \* 如果没有重名的问题，那么可以不需要使用super.属性  \* 如果有重名的问题，那么super.属性表示是访问父类的属性，这个属性一定是非private、如果是跨包，还必须是非缺省的  \*  \* super.方法  \* super.方法表示访问父类的方法，（这个方法也一定是非private、如果是跨包，还必须是非缺省的）  \*  \* 结论：  \* this.：先从本类找，本类没有，从父类找  \* super.：直接从父类开始往上找  \*  \*/  **import** com.atguigu.pkg.one.Fu;  **public** **class** Son **extends** Fu{  **public** **void** printInfo(){  //子类中没有，去父类中找  System.***out***.println("父类的privateField = " + getPrivateField());  System.***out***.println("父类的defaultField = " + getDefaultField());  System.***out***.println("父类的protectedField = " + protectedField);  System.***out***.println("父类的publicField = " + publicField);    System.***out***.println("-----------------------------");    //子类中没有，去父类中找  System.***out***.println("父类的privateField = " + **this**.getPrivateField());  System.***out***.println("父类的defaultField = " + **this**.getDefaultField());  System.***out***.println("父类的protectedField = " + **this**.protectedField);  System.***out***.println("父类的publicField = " + **this**.publicField);    System.***out***.println("-----------------------------");    //直接从父类中找  System.***out***.println("父类的privateField = " + **super**.getPrivateField());  System.***out***.println("父类的defaultField = " + **super**.getDefaultField());  System.***out***.println("父类的protectedField = " + **super**.protectedField);  System.***out***.println("父类的publicField = " + **super**.publicField);  }  } |
| **package** com.atguigu.pkg.two;  **import** com.atguigu.pkg.one.Fu;  **public** **class** Zi **extends** Fu{  **private** String privateField = "zi\_privateField";  String defaultField = "zi\_defaultField";  **protected** String protectedField = "zi\_protectedField";  **public** String publicField = "zi\_publicField";    //如果子类与父类有重名的属性，如下规则访问子类或父类的属性  **public** **void** printInfo(){  //就近原则  System.***out***.println("子类的privateField = " + privateField);  System.***out***.println("子类的defaultField = " + defaultField);  System.***out***.println("子类的protectedField = " + protectedField);  System.***out***.println("子类的publicField = " + publicField);    System.***out***.println("-----------------------------");    //指明了先从本类声明的开始查找  System.***out***.println("子类的privateField = " + **this**.privateField);  System.***out***.println("子类的defaultField = " + **this**.defaultField);  System.***out***.println("子类的protectedField = " + **this**.protectedField);  System.***out***.println("子类的publicField = " + **this**.publicField);    System.***out***.println("-----------------------------");    //指明了从父类开始查找  System.***out***.println("父类的privateField = " + **super**.getPrivateField());  System.***out***.println("父类的defaultField = " + **super**.getDefaultField());  System.***out***.println("父类的protectedField = " + **super**.protectedField);  System.***out***.println("父类的publicField = " + **super**.publicField);  }  **public** String getPrivateField() {  **return** privateField;  }  **public** **void** setPrivateField(String privateField) {  **this**.privateField = privateField;  }  **public** String getDefaultField() {  **return** defaultField;  }  **public** **void** setDefaultField(String defaultField) {  **this**.defaultField = defaultField;  }  **public** String getProtectedField() {  **return** protectedField;  }  **public** **void** setProtectedField(String protectedField) {  **this**.protectedField = protectedField;  }  **public** String getPublicField() {  **return** publicField;  }  **public** **void** setPublicField(String publicField) {  **this**.publicField = publicField;  }  } |

## 六、面向对象的基本特征之三：多态Polymorphism

多态：一类事物的多种表现形态。人：男人，女人，妖 动物：阿猫，阿狗

### 1、什么是多态

在Java中

（1）重载：在一个类中一个方法功能的多种表现形态

重写：父子类对于同一个方法表现出不同的形式

1. 对象的多态性：若编译时类型和运行时类型不一致，就出现多态（Polymorphism）

父类的引用指向子类的对象

Java程序的编译运行分为两种状态：在多态的情况下

编译时，“看左边”，看的是父类的引用（父类中不具备子类特有的方法）

运行时，“看右边”，看的是子类的对象（实际运行的是子类重写父类的方法）

### 2、向上转型与向下转型

引用数据类型之间的转换：前提都先有继承

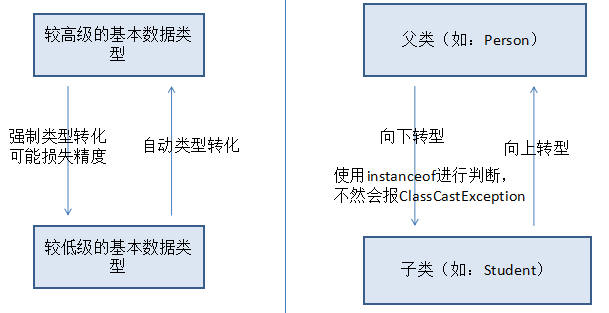
\* 向上转型upcasting：子类转父类 系统自动完成

注意：一个引用类型变量如果声明为父类的类型，但实际引用的是子类对象，那么该变量就不能再访问子类中添加的属性和方法

\* 向下转型downcasting：父类转子类 需要使用强制类型转换符()

注意：前提父类原先指向的就是本身对象才能向下转型成功，如果父类原先指向的是其他类型（父类的，兄弟类的）的对象，那么就会发生java.lang.ClassCastException类型转换异常

|  |
| --- |
| **public** **static** **void** main(String[] args) {  //基本数据类型  **int** i = 12;  **long** myLong = i;//自动类型转化，存储范围小的自动转为大的  **int** n = (**int**)myLong;//强制类型转换，存储范围大的强制转为小的，损失精度    //多态引用  Human m = **new** Man();//向上转型  Human w = **new** Woman();//向上转型    Man man = **new** Man();//本态引用  Human h1 = man;//多态引用，向上转型  Man m1 = (Man)h1;//向下转型    //Man m2 = (Man)w;//从概念上来说,w是Human类型，强制转为Man类型，存在风险。w本身指向的是Woman，如果强制转为Man就会出现异常java.lang.ClassCastException:    Human h3 = **new** Human();  // Man m3 = (Man)h3;//java.lang.ClassCastException:  } |



### 3、instanceof

\* 引用类型对象能用的运算符：

\* =：赋值 引用变量 = 对象;

\* ==和!=：比较两个对象的地址

\* +：必须与String拼接

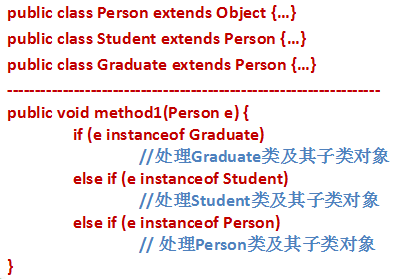
\* 其他的运算符都不能直接用于对象

\*

\* 唯一一个只用于引用类型的运算符：instanceof

x instanceof A：检验x是否为类A的对象，返回值为boolean型。

* 要求x所属的类与类A必须是子类和父类的关系，否则编译错误。
* 如果x属于类A的子类B，x instanceof A值也为true。



### 4、多态的原则

多态的前提：（1）要有继承或实现关系（2）要有方法的重写

成员方法：

编译时：要查看引用变量所属的类中是否有所调用的方法。

运行时：调用实际对象所属的类中的重写方法。

注意：

成员变量：不具备多态性，只看引用变量所属的类。

静态方法：不能被重写，也没有多态性

|  |
| --- |
| **public** **class** TestPolymorphism3 {  **public** **static** **void** main(String[] args) {  Animal a = **new** Cat();//多态引用  System.***out***.println(a.leg);//父类的  Cat c = (Cat)a;  System.***out***.println(c.leg);//子类的  }  }  **class** Animal{  **int** leg = 0;  }  **class** Cat **extends** Animal{  **int** leg = 4;  } |
| **public** **class** TestStatic{  **public** **static** **void** main(String[] args) {  Father f = **new** Son();  f.*test*();//父类的方法  }  }  **class** Father{  **public** **static** **void** test(){  System.***out***.println("父类的方法");  }  }  **class** Son **extends** Father{  **public** **static** **void** test(){  System.***out***.println("子类的方法");  }  } |

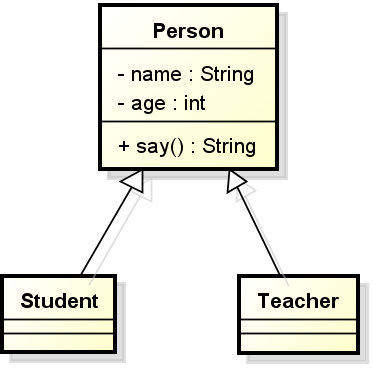
### 5、多态的应用

#### 1、多态数组

多态数组：元素的类型是父类的类型，存储的是父类或子类的对象

需求：要求创建五个年龄不等的Person、Student和Teacher对象，并将他们按年龄排序输出，应如何实现？

需求：要求创建五个Person、Student和Teacher对象，各自做自我介绍



对于这种处于同一个继承层次中的对象，使用多态数组来存放是最为简便的方法之一。

Person[] person = {new Person("张三", 32),

new Student("李四", 21, 120, 90.0),

new Student("王五", 22, 119, 91.5),

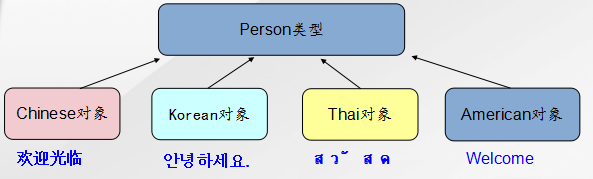
new Teacher("刘老师", 35, 10, "Java EE"),

new Teacher("张老师", 11)};

#### 2、多态参数

多态参数：形参类型是父类的类型，实参是父类或子类的对象





有一个方法welcome()，可以接待不同国家的贵宾。

|  |
| --- |
| **class** DoorMan{  **public** **void** welcome(Person p){  p.greet();  }  }  **class** Person {  **public** **void** greet(){  System.***out***.println("鞠躬");  }  }  **class** Chinese **extends** Person{  **public** **void** greet(){  System.***out***.println("你好");  }  }  **class** Korean **extends** Person{  **public** **void** greet(){  System.***out***.println("안녕하세요");  }  }  **class** Thai **extends** Person{  **public** **void** greet(){  System.***out***.println("สวัส");  }  }  **class** American **extends** Person{  **public** **void** greet(){  System.***out***.println("welcome");  }  } |

#### 3、多态属性+多态参数

一个人Human带一个宠物Pet参加聚会，而宠物有很多种类：Dog，Cat等

|  |
| --- |
| **package** com.atguigu.polymorphism;  /\*  \* 声明一个人Human，每一个人都有一个宠物Pet，  \* 宠物的子类有：Dog,Cat。。。  \*  \* 多态的应用之三：多态的属性、多态的参数  \* 属性的类型，形参的类型：父类的类型  \* 属性的值，实参：子类的对象  \*/  **public** **class** TestUser3 {  **public** **static** **void** main(String[] args) {  Human h1 = **new** Human(**new** Dog());//匿名对象  h1.getPet().shut();    Human h2 = **new** Human(**new** Cat());//匿名对象  h2.getPet().shut();  }  }  **class** Human{  **private** Pet pet;  **public** Human(Pet pet) {  **super**();  **this**.pet = pet;  }  **public** Human() {  **super**();  }  **public** Pet getPet() {  **return** pet;  }  **public** **void** setPet(Pet pet) {  **this**.pet = pet;  }    }  **class** Pet{  **public** **void** shut(){  System.***out***.println("叫");  }  }  **class** Dog **extends** Pet{  **public** **void** shut(){  System.***out***.println("汪汪叫");  }  }  **class** Cat **extends** Pet{  **public** **void** shut(){  System.***out***.println("喵喵叫");  }  } |

## 七、对象的关联关系（了解）

对象的关联 — 简单地说，是指一个对象中使用了另一个对象。

### 1、依赖关系（Dependency）

对象之间最弱的一种关联方式，是临时性的关联。代码中一般指由局部变量、函数参数、返回值建立的对于其他对象的调用关系。

**class** A{

**public** B method(C c,D d){

E e = **new** E();

...

B b = **new** B();

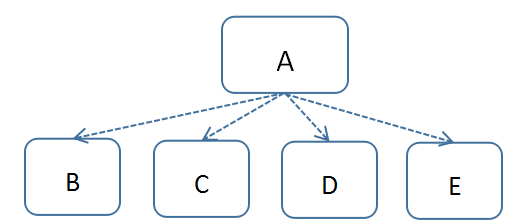
...

**return** b;

}

}

这个代码结构中，表示A类依赖了B,C,D,E类



### 2、关联关系（Association）

对象之间一种引用关系，比如客户类与订单类之间的关系。这种关系通常使用类的属性表达。

关联可以有方向，即导航。一般不作说明的时候，导航是双向的，不需要在线上标出箭头。大部分情况下导航是单向的，可以加一个箭头表示。

|  |
| --- |
| **class** Employee{  **private** **int** eid;//员工编号  **private** String name;//员工姓名  **private** Computer coumputer;//员工所使用的电脑  //....  }  **class** Computer{    } |

Employee

Computer

|  |
| --- |
| **class** Husband{  **private** Wife wife;  }  **class** Wife{  **private** Husband husband;  } |

Husband

Wife

关联表示类之间的“持久”关系，这种关系一般表示一种重要的业务之间的关系，需要保存的，或者说需要“持久化”的，或者说需要保存到数据库中的。另外，依赖表示类之间的是一种“临时、短暂”关系，这种关系是不需要保存的.

### 3、聚合（Aggregation）

聚合（关联关系的一种）：表示has-a的关系。与关联关系一样，聚合关系也是通过实例变量来实现这样关系的。关联关系和聚合关系来语法上是没办法区分的，从语义上才能更好的区分两者的区别。

如汽车类与引挚类，轮胎类之间的关系就是整体与个体的关系。

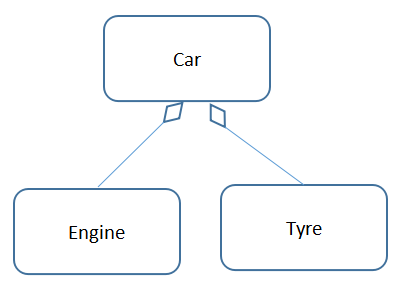
与关联关系一样，聚合关系也是通过实例变量来实现的。空心菱形

class Car{

private Engine engine;//引擎

private Tyre[] tyres;//轮胎

}



关联和聚集（聚合）的区别：

关联关系所涉及的两个对象是处在同一个层次上的。比如人和自行车就是一种关联关系，而不是聚合关系，因为人不是由自行车组成的。

聚合关系涉及的两个对象处于不平等的层次上，一个代表整体，一个代表部分。比如电脑和它的显示器、键盘、主板以及内存就是聚集关系，因为主板是电脑的组成部分。

### 4、组合（Composite）

对象A包含对象B，对象B离开对象A没有实际意义。是一种更强的关联关系。人包含手，手离开人的躯体就失去了它应有的作用。

组合：表示contains-a的关系，是一种强烈的包含关系。组合类负责被组合类的生命周期。也使用属性表达组合关系，是关联关系的一种，是比聚合关系强的关系。

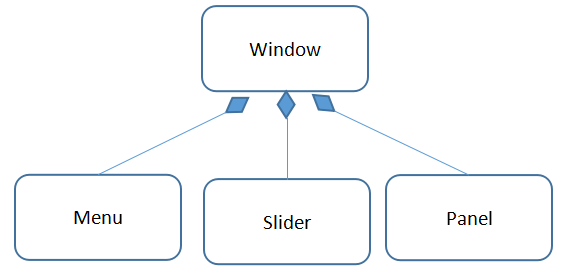
class Window{

private Menu menu;//菜单

private Slider slider;//滑动条

private Panel panel;//工作区

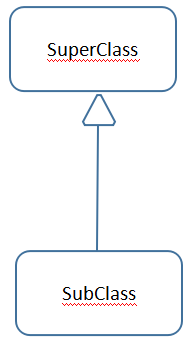
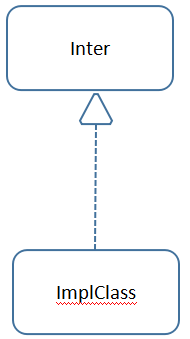
}



### 5、继承（Generalization，又称为泛化，is-a 的关系）

类与类的继承关系，类与接口的实现关系。

场景：父与子、动物与人、植物与树

## 练习

1、创建程序，在其中定义两个类：Person和TestPerson类。

定义如下：定义Person类,有4个属性：String name; int age; String school; String major，属性私有化，其中

用setAge()设置人的合法年龄[0,130]，用getAge()返回人的年龄。

在TestPerson类中实例化Person类的对象b，调用setAge()和getAge()等方法，体会Java的封装性。

2、定义Person类的3个构造方法:

第一个构造方法Person(String n, int a)设置类的name和age属性；

第二个构造方法Person(String n, int a, String s)设置类的name, age 和school属性；

第三个构造方法Person(String n, int a, String s, String m)设置类的name, age ,school和major属性；

在main方法中分别调用不同的构造方法创建的对象，并输出其属性值。

如果此时想要Person类有一个无参构造，怎么做？

如果想要把年龄初始化为18岁怎么做？

3、编写类Triangle，声明私有的底边长base和高height，同时声明公共方法访问私有变量；

在TriangleManager类中，定义一个方法：public static double getArea(Trangle t)计算三角形的面积

在测试类中测试

4、定义一个丈夫Husband类，有姓名、年龄、妻子属性，其中妻子属性没有getter和setter，而是通过marry方法，赋值，妻子是私有的不提供公共的getter方法。

定义一个妻子Wife类，有姓名、年龄、丈夫属性，其中丈夫属性没有getter和setter，而是通过marryTo方法，赋值，丈夫是私有的不提供公共的getter方法

丈夫类中有一个getInfo方法，其中，能显示自己的姓名，年龄，和他的妻子的姓名，年龄

妻子类中有一个getInfo方法，其中，能显示自己的姓名，年龄，和他的丈夫的姓名，年龄

5、（1）Room类：在com.atguigu.bean包中

属性：房号、面积、地址，使用封装

方法：getDetails 返回String，描述房间详细信息

构造器：0-3个参数的重载构造器，使用this（）调用

（2）TestRoom类：在com.atgugui.test包中

分别用4个构造器创建4个room对象，打印对象的详细信息

1、编写Computer类，包含CPU、内存、硬盘等属性，getDetails方法用于返回Computer的详细信息

编写PC子类，继承Computer类，添加特有属性和方法

编写NotePad子类，继承Computer类，添加特有属性和方法

编写Test类，在main方法中创建PC和NotePad对象，分别访问对象中特有的属性、方法，以及从Computer类继承的属性和方法并打印输出。

2、改写Computer类，将所有属性声明为private，getDetails方法用于返回Computer的详细信息

在PC子类中直接访问继承的私有属性，结果如何？

在Computer类中对私有属性添加公有的get/set方法，在PC子类中通过这些公有的get/set方法访问私有属性，结果如何？

3、在PC类中，覆盖getDetails方法，方法返回PC机的详细信息。

在Test类中调用getDetails方法，确认输出结果。

4、在Computer类中添加新的重载构造器，并调用原构造器

在PC类中添加新的重载构造器，并调用原构造器

使用PC类中新的重载构造器创建PC实例，调用getDetails方法获取输出结果，确认属性值

5、根据下图实现类。在TestCylinder类中创建Cylinder类的对象，设置圆柱的底面半径和高，并输出圆柱的体积。

6、综合练习

编写一个Person类，包括属性（name、age），构造器、方法say(返回自我介绍的字符串）。

编写一个Student类，继承Person类，增加ID、Score属性，以及构造器，并覆盖say方法。

编写Test类，创建分别Person和Student对象，调用say方法输出自我介绍。

1、编写Computer类，包含CPU、内存、硬盘等属性，getDetails方法用于返回Computer的详细信息

使用多态引用分别创建Computer、PC和NotePad实例，并将实例放在一个Computer类型的数组中。

遍历数组元素，分别调用getDetails方法获取输出结果。

2、在Computer类中增加price属性表示价格。

将数组元素按价格进行排序，并遍历打印输出。

3、在Test类中提供一个静态方法listPrice,以Computer引用变量为参数，打印输出电脑价格。

在main方法中，分别以Computer、PC、NotePad对象为参数，调用listPrice方法。

4、在方法listPrice中，判断Computer参数的真实对象，并调用不同对象上的特有方法

# 项目2：客户信息管理系统

模拟实现一个基于文本界面的《客户信息管理软件》

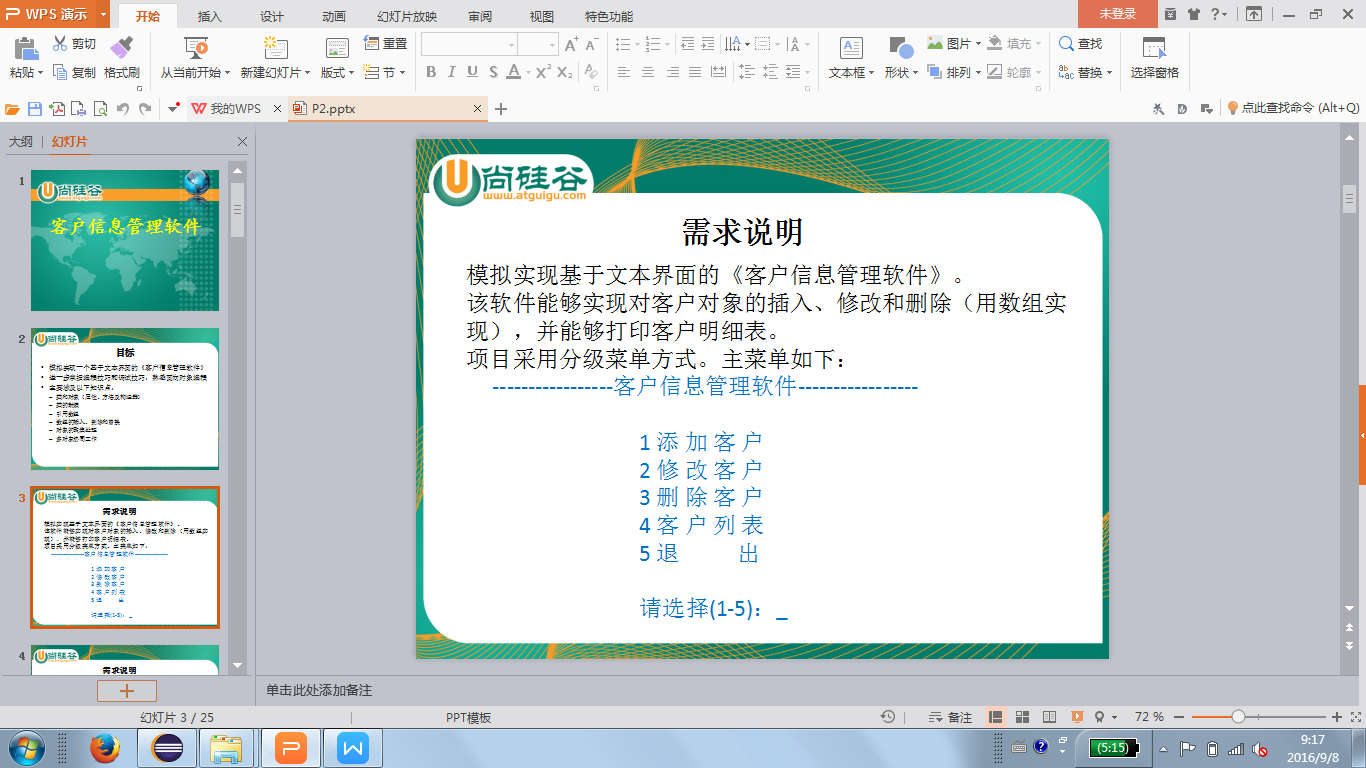
进一步掌握编程技巧和调试技巧，熟悉面向对象编程

主要涉及以下知识点：

* 类和对象（属性、方法及构造器）
* 类的封装
* 引用数组
* 数组的插入、删除和替换
* 对象的聚集处理
* 多对象协同工作

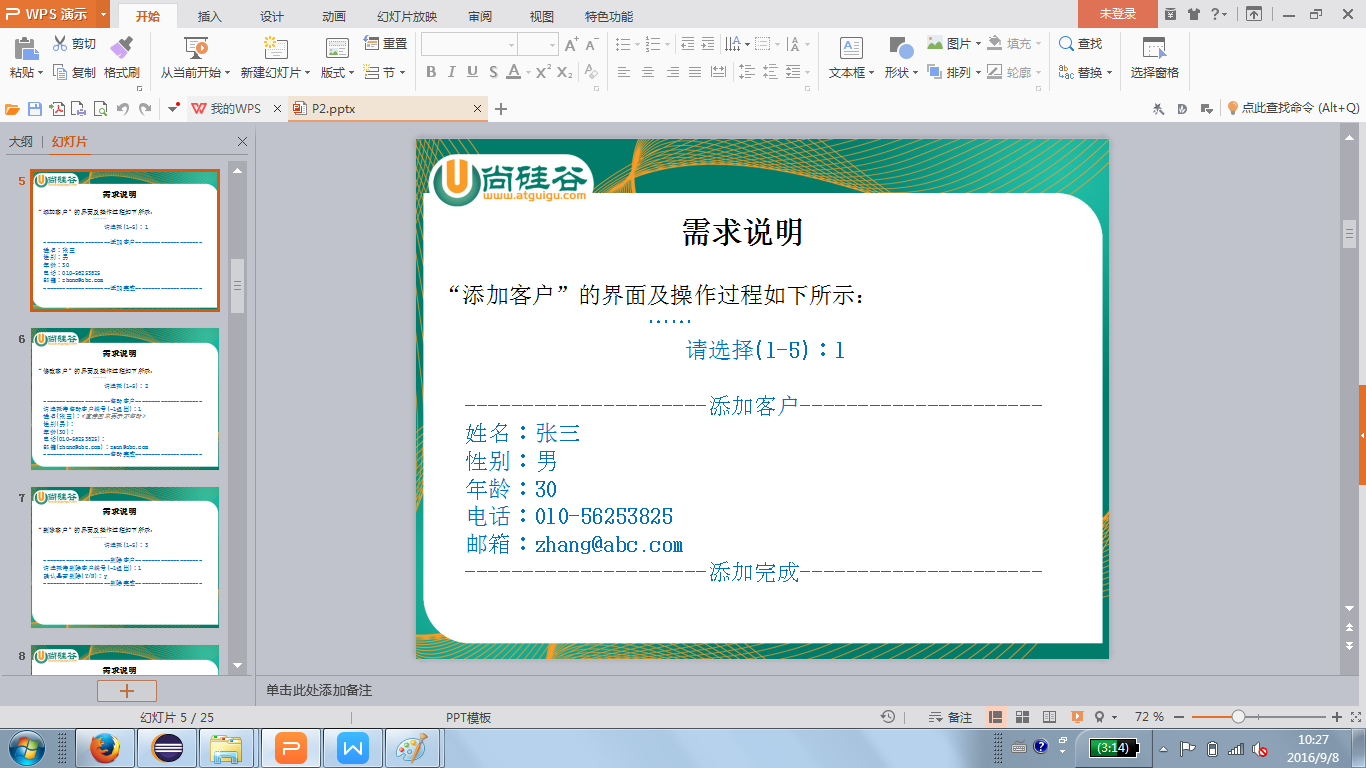
## 一、需求说明

### 1、主界面

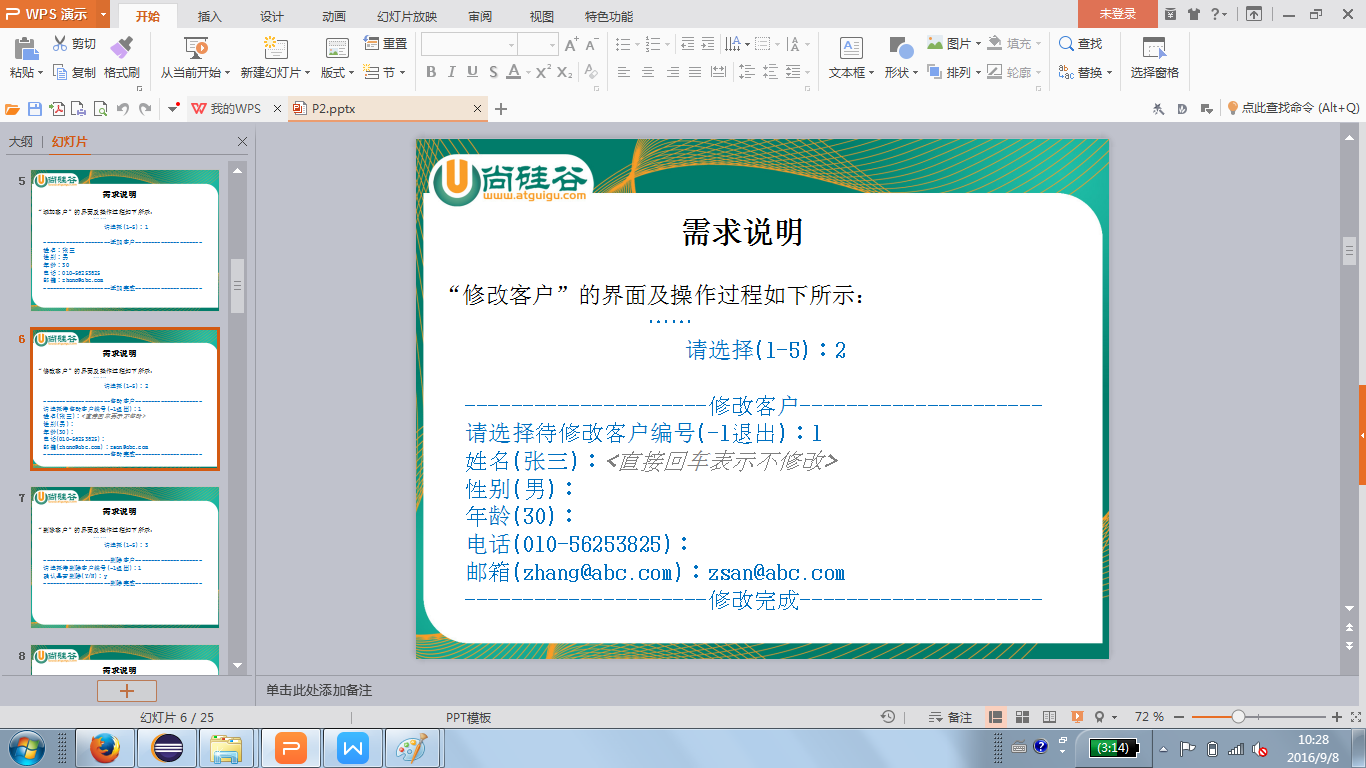


* 每个客户的信息被保存在Customer对象中。
* 以一个Customer类型的数组来记录当前所有的客户
* 每次“添加客户”（菜单1）后，客户（Customer）对象被添加到数组中。
* 每次“修改客户”（菜单2）后，修改后的客户（Customer）对象替换数组中原对象。
* 每次“删除客户”（菜单3）后，客户（Customer）对象被从数组中清除。
* 执行“客户列表 ”（菜单4）时，将列出数组中所有客户的信息

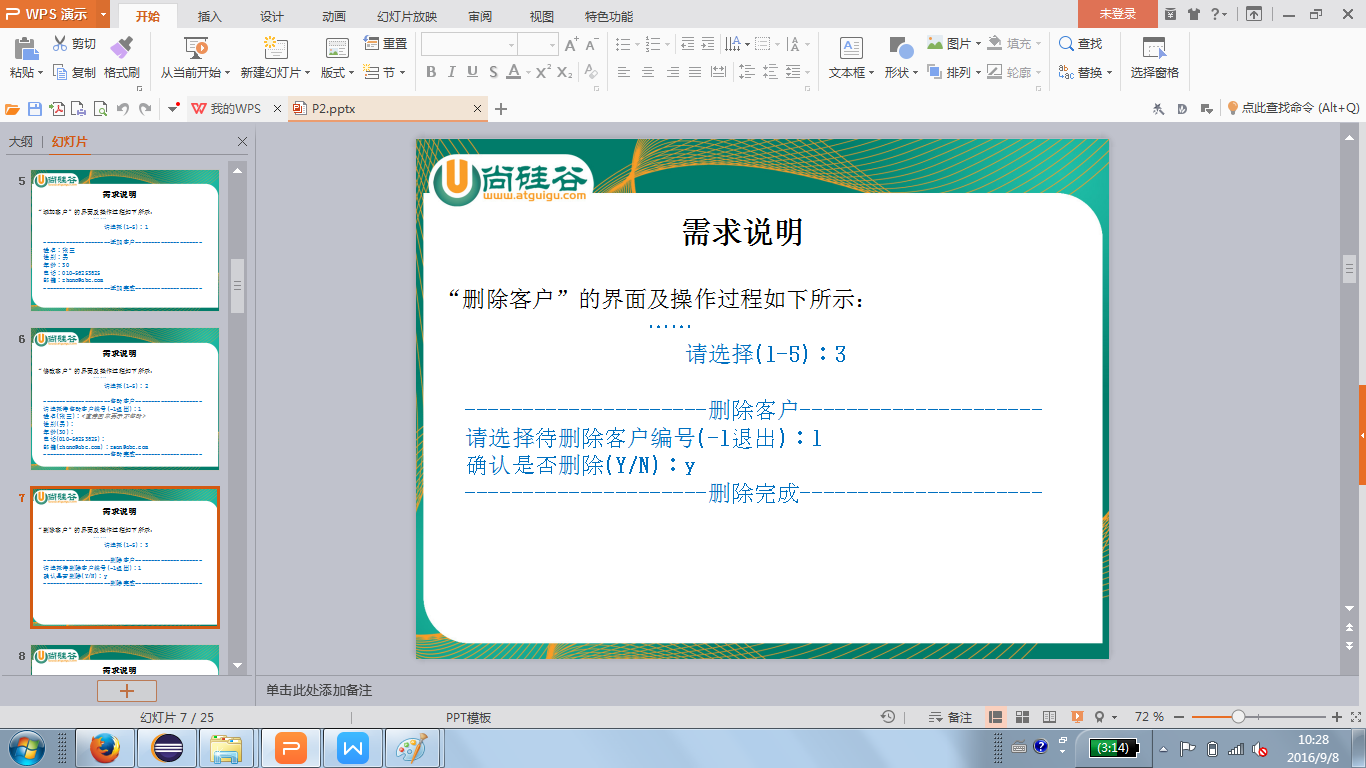
### 2、添加客户



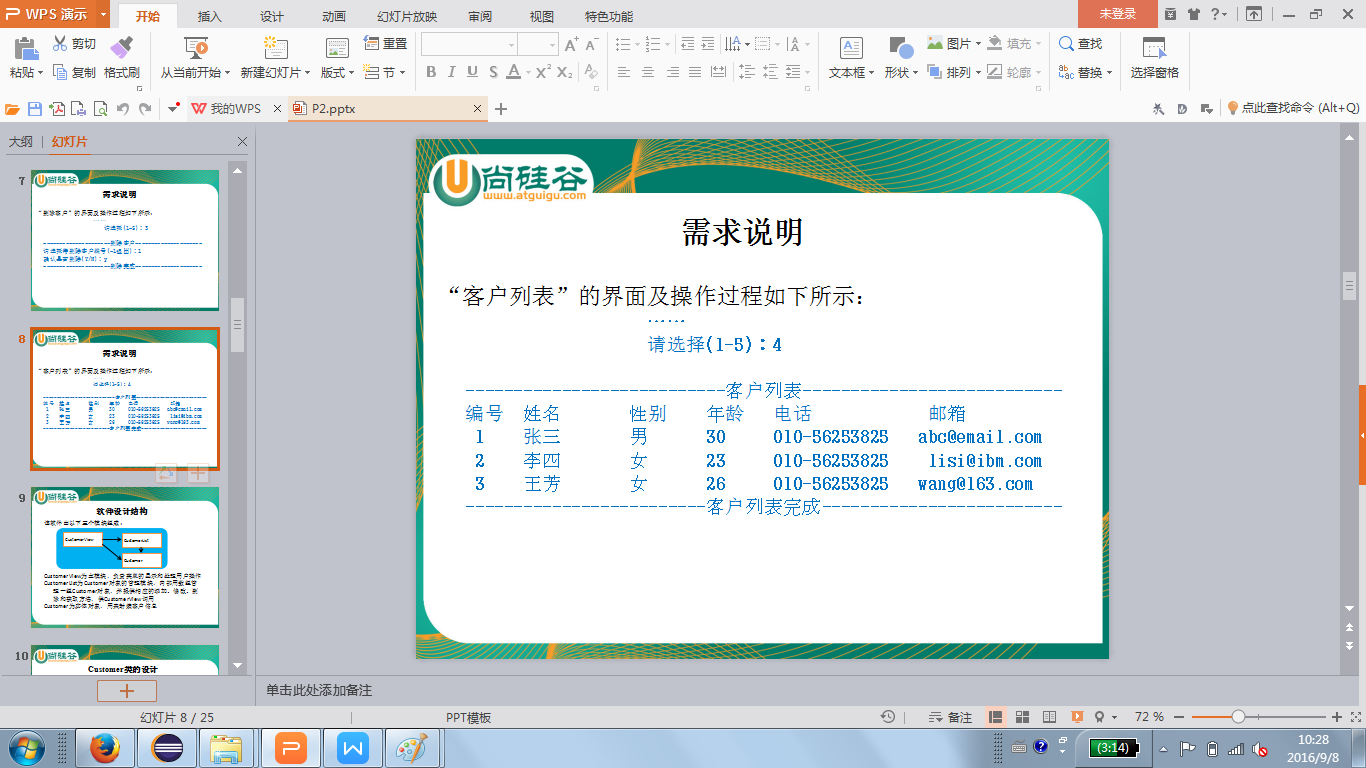
### 3、修改客户



### 4、删除客户



### 5、客户列表



## 二、键盘输入工具类CMUtility.java

项目中提供了CMUtility.java类，可用来方便地实现键盘访问。

1、public static char readMenuSelection()

用途：该方法读取键盘，如果用户键入’1’-’5’中的任意字符，则方法返回。返回值为用户键入字符。

2、public static char readChar() 和 public static char readChar(char defaultValue)

用途：这两个方法功能相同，均从键盘读取一个字符，并将其作为方法的返回值。

参数： defaultValue — 如果用户不输入字符而直接回车，方法将以defaultValue 作为返回值。（提示：此方法可在修改客户时调用）

3、public static int readInt() 和 public static int readInt(int defaultValue)

用途：这两个方法功能相同，均从键盘读取一个长度不超过2位的 整数，并将其作为方法的返回值。

参数： defaultValue — 如果用户不输入字符而直接回车，方法将以defaultValue 作为返回值。

4、public static String readString(int limit) 和 public static String readString(int limit, String defaultValue)

用途：这两个方法功能相同，均从键盘读取一个长度不超过limit的字符串，并将其作为方法的返回值。

参数：limit — 指定字符串的最大长度

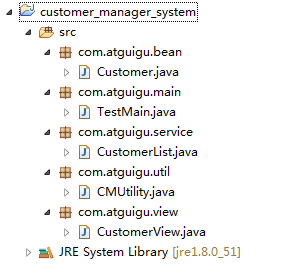
defaultValue — 如果用户不输入字符而直接回车，方法将以defaultValue 作为返回值。

5、public static char readConfirmSelection() ：

用途：从键盘读取‘Y’或’N’，并将其作为方法的返回值。

## 三、开发详解

### 3.1 项目结构



#### 1、先建立5个包及相应的类

（1）com.atguigu.main：用来存放main()所在的类作为程序的入口，例如：TestMain.java

（2）com.atguigu.bean：用来存放项目中的实体类，例如Customer.java

在日常的Java项目开发中，entity（实体类）是必不可少的，它们一般都有很多的属性，并有相应的setter和getter方法。entity（实体类）的作用一般是和数据表做映射。所以快速写出规范的entity（实体类）是java开发中一项必不可少的技能。

在项目中写实体类一般遵循下面的规范：

* 根据你的设计，定义一组你需要的私有属性。
* 根据这些属性，创建它们的setter和getter方法。（eclipse等集成开发软件可以自动生成。具体怎么生成？请自行百度。）
* 提供带参数的构造器和无参数的构造器。
* 重写父类中的eauals()方法和hashcode()方法。（如果需要涉及到两个对象之间的比较，这两个功能很重要。本项目中不需要）
* 实现Comparable并重写compareTo()方法（如果需要涉及到对象排序。本项目中不需要）
* 实现序列化并赋予其一个版本号。（本项目中不需要）

（3）com.atguigu.service：用来存放项目中业务逻辑类，例如CustomerList.java

业务逻辑层（Business Logic Layer）无疑是系统架构中体现核心价值的部分。它的关注点主要集中在业务规则的制定、业务流程的实现等与业务需求有关的系统设计，也即是说它是与系统所应对的领域（Domain）逻辑有关。

说白了，就是每个项目的具体业务功能的管理和实现。

在本项目中就是客户信息的添加、修改、删除、查询（返回客户列表）。

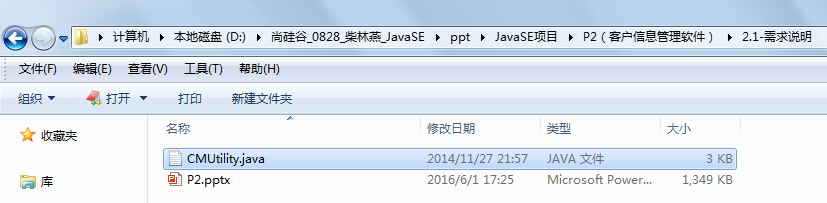
注意：只关注逻辑，不设计界面的呈现。

（4）com.atguigui.view：用来存放项目中的视图类（即UI类），例如CustomerView.java

表示层：只负责与用户（程序的使用者）进行交互，通过界面的交互来获取用户的输入或呈现结果给用户看。

（5）com.atguigu.util：用来存放项目中的一些工具类，例如CMUtility.java

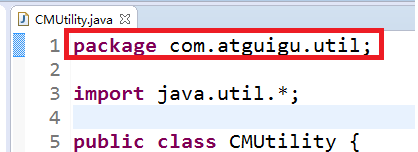
注意：在本项目中CMUtility.java文件直接从给的资料中复制到com.atguigu.util包中。



复制，然后选择com.atguigu.util包，粘贴即可



因为给的CMUtility.java类，没有声明任何包，因此直接复制进去后就报错了，所以要给CMUtility.java类加上包的声明



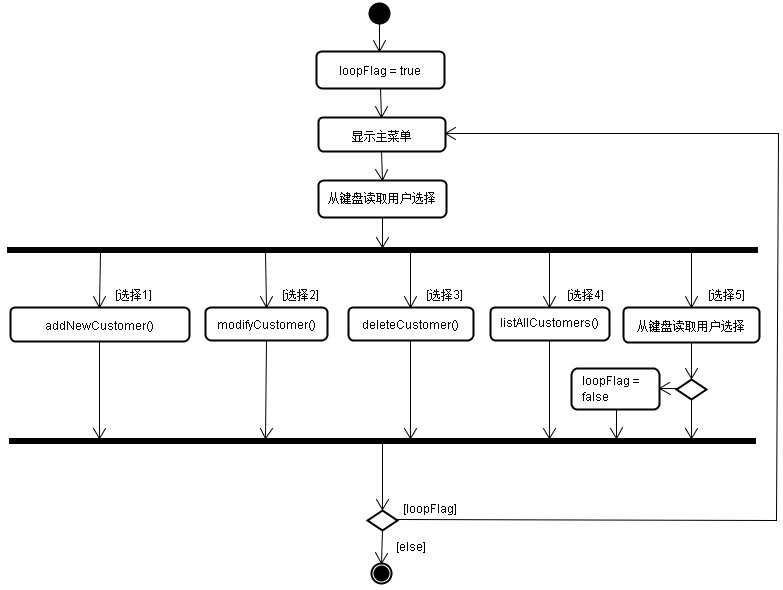
（6）com.atguigu.dao：在实际项目中还有一层数据访问层，用于存放和数据库交互的类。本项目不需要

#### 2、类之间的调用关系

#### （1）整体调用关系



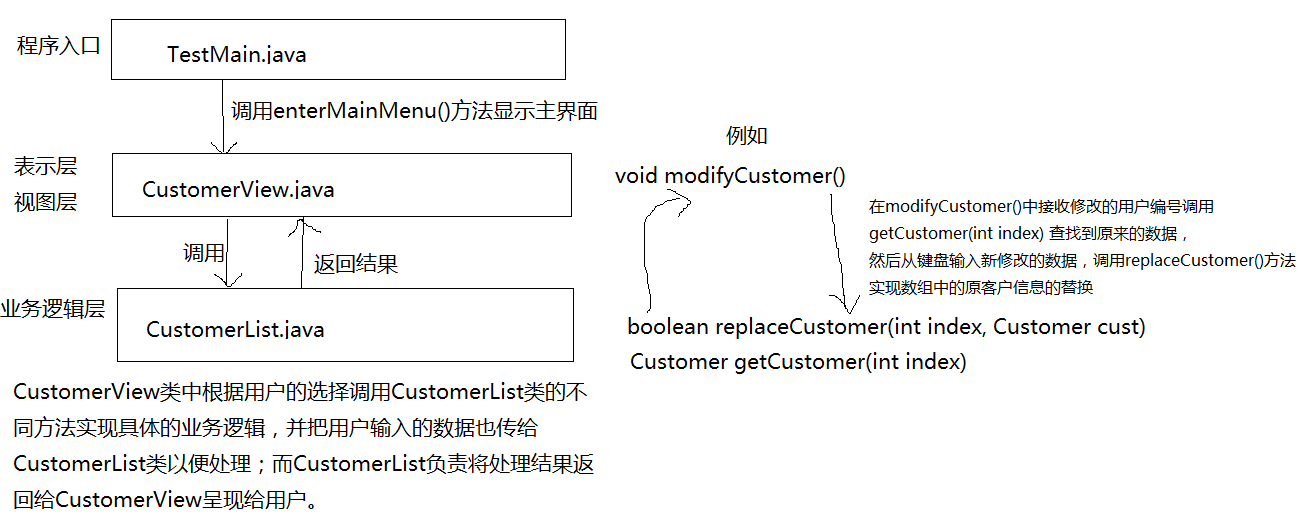
#### （2）enterMainMenu()方法的活动图



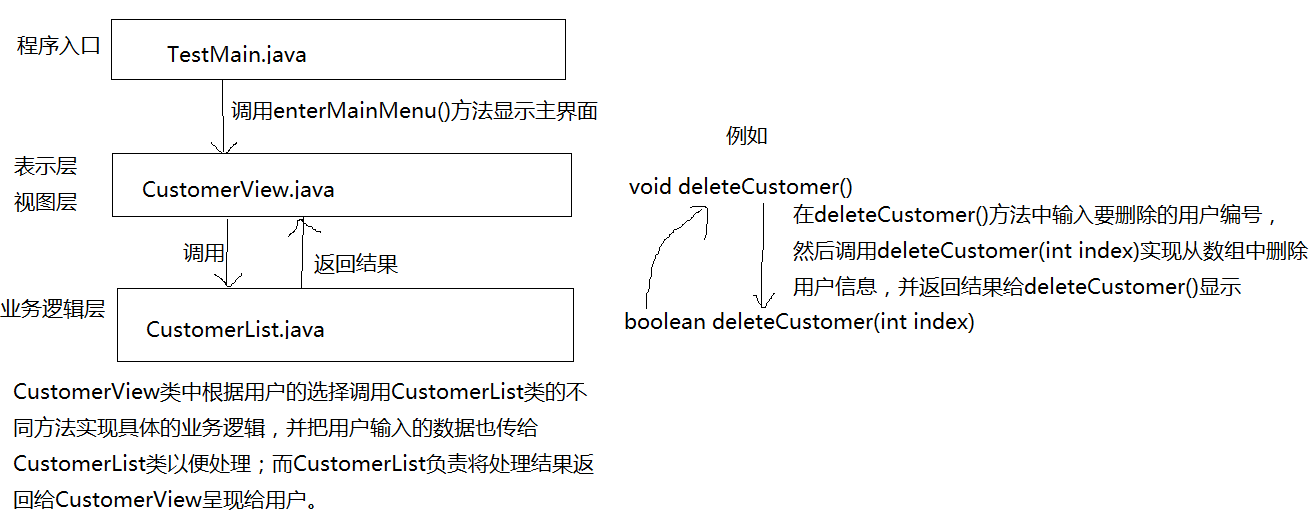
#### （3）添加用户功能



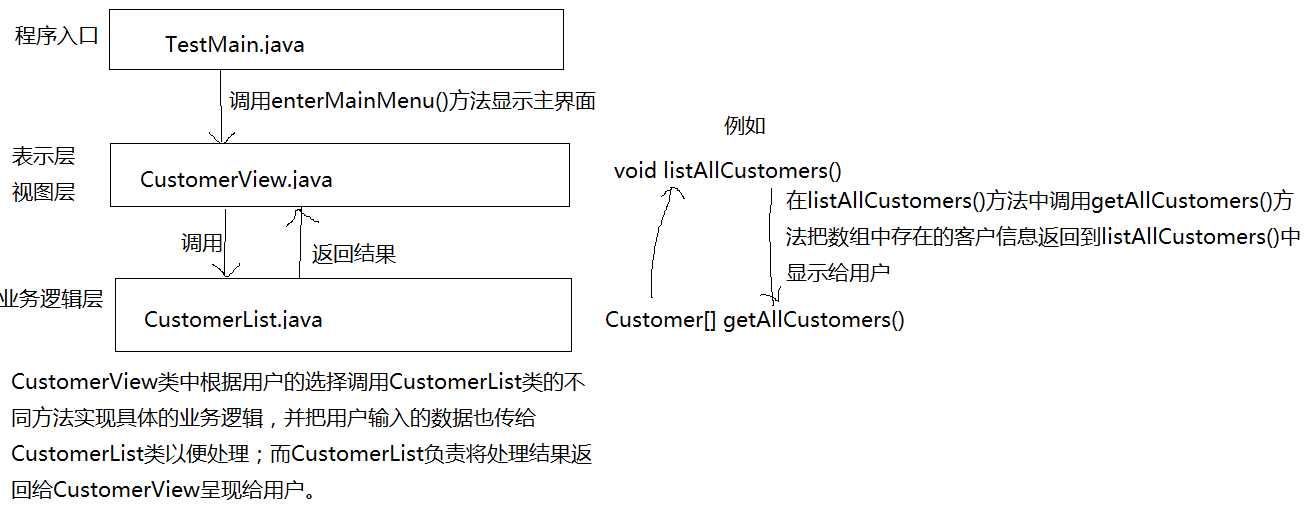
#### （4）修改用户功能



#### （5）删除用户功能



#### （6）用户列表功能



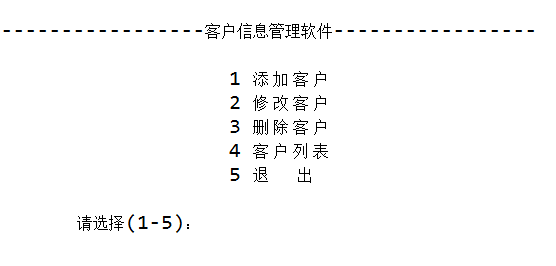
### 3.2 实现代码

#### 1、实体类：Customer.java

|  |
| --- |
| **package** com.atguigu.bean;  **public** **class** Customer {  **private** String name;  **private** **char** gender;  **private** **int** age;  **private** String phone;  **private** String email;  **public** Customer(String name, **char** gender, **int** age) {  **this**(name, gender, age, "", "");  //如果没有填写手机号码或email的，默认空字符串，这样就不会默认为null，  //避免显示时显示null，对于非编程人员看不懂  }  **public** Customer(String name, **char** gender, **int** age,String phone, String email) {  **this**.name = name;  **this**.gender = gender;  **this**.age = age;  **this**.phone = phone;  **this**.email = email;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **char** getGender() {  **return** gender;  }  **public** **void** setGender(**char** gender) {  **this**.gender = gender;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  **public** String getPhone() {  **return** phone;  }  **public** **void** setPhone(String phone) {  **this**.phone = phone;  }  **public** String getEmail() {  **return** email;  }  **public** **void** setEmail(String email) {  **this**.email = email;  }  //方便显示用户信息  **public** String getDetails() {  **return** name + "\t" + gender + "\t" + age + "\t" + phone + "\t" + email;  }  } |

可以在main方法中创建Customer对象，并调用对象的各个方法，以测试该类是否编写正确。

#### 2、编写主界面



#### 编写CustomerView.java

|  |
| --- |
| **package** com.atguigu.view;  **import** com.atguigu.util.CMUtility;  **public** **class** CustomerView {  **public** **void** enterMainMenu() {  **boolean** loopFlag = **true**;  **do** {  System.***out***.println("\n-----------------客户信息管理软件-----------------\n");  System.***out***.println(" 1 添 加 客 户");  System.***out***.println(" 2 修 改 客 户");  System.***out***.println(" 3 删 除 客 户");  System.***out***.println(" 4 客 户 列 表");  System.***out***.println(" 5 退 出\n");  System.***out***.print(" 请选择(1-5)：");    **char** key = CMUtility.*readMenuSelection*();  System.***out***.println();  **switch** (key) {  **case** '1':  addNewCustomer();  **break**;  **case** '2':  modifyCustomer();  **break**;  **case** '3':  deleteCustomer();  **break**;  **case** '4':  listAllCustomers();  **break**;  **case** '5':  System.***out***.print("确认是否退出(Y/N)：");  **char** yn = CMUtility.*readConfirmSelection*();  **if** (yn == 'Y') loopFlag = **false**;  **break**;  }  } **while** (loopFlag);  }  **private** **void** addNewCustomer() {  }    **private** **void** modifyCustomer() {  }  **private** **void** deleteCustomer() {  }  **private** **void** listAllCustomers() {  }  } |

#### （2）编写TestMain.java

|  |
| --- |
| **package** com.atguigu.main;  **import** com.atguigu.view.CustomerView;  **public** **class** TestMain {  **public** **static** **void** main(String[] args) {  CustomerView cView = **new** CustomerView();  cView.enterMainMenu();  }  } |

运行测试

#### 3、编写CustomerList.java类

CustomerList为Customer对象的管理模块，内部用数组管理一组Customer对象

本类封装以下信息：

Customer[] customers：用来保存客户对象的数组

int total = 0 ：记录已保存客户对象的数量

该类至少提供以下方法：

public CustomerList(int totalCustomer)

public boolean addCustomer(Customer customer)

public boolean replaceCustomer(int index, Customer cust)

public boolean deleteCustomer(int index)

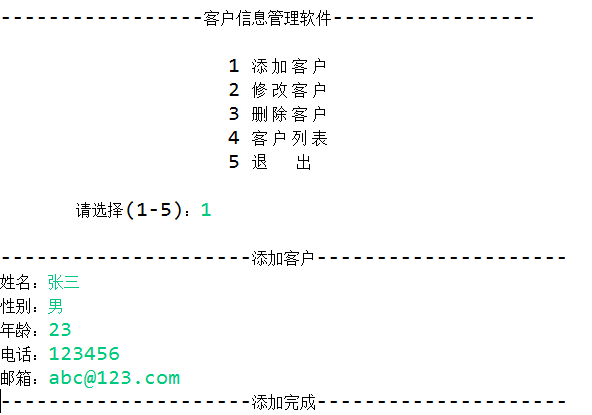
public Customer[] getAllCustomers()

public Customer getCustomer(int index)

|  |
| --- |
| **package** com.atguigu.service;  **import** com.atguigu.bean.Customer;  **public** **class** CustomerList {  **private** Customer[] customers;//用来保存客户对象的数组  **private** **int** total = 0;//记录已保存客户对象的数量  //初始化CustomerList对象时，初始化customers数组  **public** CustomerList(**int** totalCustomer) {  customers = **new** Customer[totalCustomer];  }    //添加客户  **public** **boolean** addCustomer(Customer customer) {  **return** **true**;//先确保语法正确，所以return 语句不可少  }    //修改客户  **public** **boolean** replaceCustomer(**int** index, Customer cust) {  **return** **true**;//先确保语法正确，所以return 语句不可少  }  //删除客户  **public** **boolean** deleteCustomer(**int** index) {  **return** **true**;//先确保语法正确，所以return 语句不可少  }  //返回所有客户列表  **public** Customer[] getAllCustomers() {  **return** **null**;//先确保语法正确，所以return 语句不可少，Customer[]是个引用类型，所以先返回null  }  //查找某个客户  **public** Customer getCustomer(**int** index) {  **return** **null**;//先确保语法正确，所以return 语句不可少，Customer 是个引用类型，所以先返回null  }  } |

#### 4、编写添加用户功能

“添加客户”操作是否正确，给用户的提示是否明确合理；测试当添加的客户总数超过10时，运行是否正确



#### （1）在CustomerView.java中实现addNewCustomer()方法

|  |
| --- |
| **private** **void** addNewCustomer() {  System.***out***.println("---------------------添加客户---------------------");  System.***out***.print("姓名：");  String name = CMUtility.*readString*(4);  System.***out***.print("性别：");  **char** gender = CMUtility.*readChar*();  System.***out***.print("年龄：");  **int** age = CMUtility.*readInt*();  System.***out***.print("电话：");  String phone = CMUtility.*readString*(15);  System.***out***.print("邮箱：");  String email = CMUtility.*readString*(15);  Customer cust = **new** Customer(name, gender, age, phone, email);  **boolean** flag = customers.addCustomer(cust);  **if** (flag) {  System.***out***.println("---------------------添加完成---------------------");  } **else** {  System.***out***.println("----------------记录已满,无法添加-----------------");  }  } |

#### （2）在CustomerList.java类中实现addCustomer()方法

|  |
| --- |
| // 添加客户  **public** **boolean** addCustomer(Customer customer) {  //如果已保存客户数量已经到达数组保存极限，那么保存失败  **if** (total >= customers.length)  **return** **false**;  customers[total++] = customer;  //每添加一个客户，total计数器就增1  **return** **true**;  } |

#### 5、编写客户列表功能

#### （1）在CustomerView.java类中实现listAllCustomers()方法

|  |
| --- |
| **private** **void** listAllCustomers() {  System.***out***.println("---------------------------客户列表---------------------------");  Customer[] custs = customers.getAllCustomers();  **if** (custs.length == 0) {  System.***out***.println("没有客户记录！");  } **else** {  System.***out***.println("编号 姓名\t 性别\t 年龄\t电话\t\t邮箱");  }  **for** (**int** i = 0; i < custs.length; i++) {  System.***out***.println((i+1) + "\t" + custs[i].getDetails());  }  System.***out***.println("-------------------------客户列表完成-------------------------");  } |

#### 在CustomerList.java中实现

|  |
| --- |
| // 返回所有客户列表  **public** Customer[] getAllCustomers() {  **return** customers;  } |

#### 测试报错



原因在于实际客户只有一个，而刚才在CustomerList.java的 getAllCustomers()中返回的是整个保存客户信息的数组，而那里面有未存数据的null元素

#### 修改BUG

修改CustomerList.java的 getAllCustomers()方法如下

|  |
| --- |
| // 返回所有客户列表  **public** Customer[] getAllCustomers() {  //创建一个新数组，返回已经保存的所有客户信息  Customer[] cust = **new** Customer[total];  **for** (**int** i = 0; i < total; i++) {  cust[i] = customers[i];  }  **return** cust;  } |

#### 运行测试

|  |
| --- |
| -----------------客户信息管理软件-----------------  1 添 加 客 户  2 修 改 客 户  3 删 除 客 户  4 客 户 列 表  5 退 出  请选择(1-5)：1  ---------------------添加客户---------------------  姓名：张三  性别：男  年龄：23  电话：123456  邮箱：a@b.com  ---------------------添加完成---------------------  -----------------客户信息管理软件-----------------  1 添 加 客 户  2 修 改 客 户  3 删 除 客 户  4 客 户 列 表  5 退 出  请选择(1-5)：4  ---------------------------客户列表---------------------------  编号 姓名 性别 年龄 电话 邮箱  1 张三 男 23 123456 a@b.com  -------------------------客户列表完成------------------------- |

#### 6、提高测试效率

每次运行都要添加客户，特别麻烦，如果添加客户功能已经没有问题，那么可以添加一些初始化的测试数据，方便测试。

|  |
| --- |
| **public** CustomerView() {  Customer cust1 = **new** Customer("张三", '男', 30, "010-52776920", "abc@email.com");  Customer cust2 = **new** Customer("李玟", '女', 28, "010-52776920", "abc@email.com");  customers.addCustomer(cust1);  customers.addCustomer(cust2);  } |

这样程序一运行，在客户数组中就已经默认添加了两个客户了

#### 7、编写客户修改功能

#### （1）在CustomerView.java中实现modifyCustomer() 方法

|  |
| --- |
| **private** **void** modifyCustomer() {  System.***out***.println("---------------------修改客户---------------------");    System.***out***.print("请选择待修改客户编号：");  **int** index = CMUtility.*readInt*();    //查找客户的原有信息  Customer cust = customers.getCustomer(index - 1);  **if** (cust == **null**) {  System.***out***.println("----------无法找到指定客户,修改失败--------------");  **return** ;//结束修改方法  }  System.***out***.print("姓名(" + cust.getName() + ")：");  String name = CMUtility.*readString*(4, cust.getName());  System.***out***.print("性别(" + cust.getGender() + ")：");  **char** gender = CMUtility.*readChar*(cust.getGender());  System.***out***.print("年龄(" + cust.getAge() + ")：");  **int** age = CMUtility.*readInt*(cust.getAge());  System.***out***.print("电话(" + cust.getPhone() + ")：");  String phone = CMUtility.*readString*(15, cust.getPhone());  System.***out***.print("邮箱(" + cust.getEmail() + ")：");  String email = CMUtility.*readString*(15, cust.getEmail());  cust = **new** Customer(name, gender, age, phone, email);  **boolean** flag = customers.replaceCustomer(index - 1, cust);//下标从0开始  **if** (flag) {  System.***out***.println("---------------------修改完成---------------------");  } **else** {  System.***out***.println("---------------------修改失败--------------");  }  } |

#### 在CustomerList.java中实现getCustomer()方法

|  |
| --- |
| // 查找某个客户  **public** Customer getCustomer(**int** index) {  **if** (index < 0 || index >= total) **return** **null**;    **return** customers[index];  } |

#### （3）在CustomerList.java中实现replaceCustomer()方法

|  |
| --- |
| // 修改客户  **public** **boolean** replaceCustomer(**int** index, Customer cust) {  **if** (index < 0 || index >= total) **return** **false**;    customers[index] = cust;  **return** **true**;  } |

#### （4）运行测试

|  |
| --- |
| -----------------客户信息管理软件-----------------  1 添 加 客 户  2 修 改 客 户  3 删 除 客 户  4 客 户 列 表  5 退 出  请选择(1-5)：4  ---------------------------客户列表---------------------------  编号 姓名 性别 年龄 电话 邮箱  1 张三 男 30 010-52776920 abc@email.com  2 李玟 女 28 010-52776920 abc@email.com  -------------------------客户列表完成-------------------------  -----------------客户信息管理软件-----------------  1 添 加 客 户  2 修 改 客 户  3 删 除 客 户  4 客 户 列 表  5 退 出  请选择(1-5)：2  ---------------------修改客户---------------------  请选择待修改客户编号：3  ----------无法找到指定客户,修改失败--------------  -----------------客户信息管理软件-----------------  1 添 加 客 户  2 修 改 客 户  3 删 除 客 户  4 客 户 列 表  5 退 出  请选择(1-5)：2  ---------------------修改客户---------------------  请选择待修改客户编号：1  姓名(张三)：张三丰  性别(男)：  年龄(30)：35  电话(010-52776920)：  邮箱(abc@email.com)：  ---------------------修改完成---------------------  -----------------客户信息管理软件-----------------  1 添 加 客 户  2 修 改 客 户  3 删 除 客 户  4 客 户 列 表  5 退 出  请选择(1-5)：4  ---------------------------客户列表---------------------------  编号 姓名 性别 年龄 电话 邮箱  1 张三丰 男 35 010-52776920 abc@email.com  2 李玟 女 28 010-52776920 abc@email.com  -------------------------客户列表完成------------------------- |

#### 8、编写删除客户功能

#### （1）在CustomerView.java中实现deleteCustomer()方法

|  |
| --- |
| **private** **void** deleteCustomer() {  System.***out***.println("---------------------删除客户---------------------");    System.***out***.print("请选择待删除客户编号：");  **int** index = CMUtility.*readInt*();  Customer cust = customers.getCustomer(index - 1);  **if** (cust == **null**) {  System.***out***.println("----------无法找到指定客户,删除失败--------------");  **return** ;  }  System.***out***.print("确认是否删除(Y/N)：");  **char** yn = CMUtility.*readConfirmSelection*();  **if** (yn == 'N') **return**;  **boolean** flag = customers.deleteCustomer(index - 1);  **if** (flag) {  System.***out***.println("---------------------删除完成---------------------");  } **else** {  System.***out***.println("----------无法找到指定客户,删除失败--------------");  }  } |

#### （2）在CustomerList.java中实现

|  |
| --- |
| // 删除客户  **public** **boolean** deleteCustomer(**int** index) {  **if** (index < 0 || index >= total) **return** **false**;    //后面的元素往前移动，这样每次添加的时候都在最后添加  **for** (**int** i = index + 1; i < total; i++) {  customers[index] = customers[i];  }    customers[--total] = **null**;  **return** **true**;  } |

#### （3）运行测试

|  |
| --- |
| -----------------客户信息管理软件-----------------  1 添 加 客 户  2 修 改 客 户  3 删 除 客 户  4 客 户 列 表  5 退 出  请选择(1-5)：4  ---------------------------客户列表---------------------------  编号 姓名 性别 年龄 电话 邮箱  1 张三 男 30 010-52776920 abc@email.com  2 李玟 女 28 010-52776920 abc@email.com  -------------------------客户列表完成-------------------------  -----------------客户信息管理软件-----------------  1 添 加 客 户  2 修 改 客 户  3 删 除 客 户  4 客 户 列 表  5 退 出  请选择(1-5)：3  ---------------------删除客户---------------------  请选择待删除客户编号：3  ----------无法找到指定客户,删除失败--------------  -----------------客户信息管理软件-----------------  1 添 加 客 户  2 修 改 客 户  3 删 除 客 户  4 客 户 列 表  5 退 出  请选择(1-5)：3  ---------------------删除客户---------------------  请选择待删除客户编号：1  确认是否删除(Y/N)：Y  ---------------------删除完成---------------------  -----------------客户信息管理软件-----------------  1 添 加 客 户  2 修 改 客 户  3 删 除 客 户  4 客 户 列 表  5 退 出  请选择(1-5)：4  ---------------------------客户列表---------------------------  编号 姓名 性别 年龄 电话 邮箱  1 李玟 女 28 010-52776920 abc@email.com  -------------------------客户列表完成------------------------- |

# 第七章 面向对象的高级特性

## 7.1 根父类java.lang.Object

### 7.1.1 Object类是所有Java类的根父类

#### 1、根父类

如果在类的声明中未使用extends关键字指明其父类，则默认父类为Object类

public class Person {

...

}

等价于：

public class Person extends Object {

...

}

#### 2、根据继承的特点

（1）父类中所有的属性和方法都会继承到子类中，或者说所有对象都具有Object类的方法，包括数组对象

（2）所有的子类都会调用它的构造器，super()

#### 3、根据多态的特性

（1）Object类型的引用变量可以接收任何类型的对象

Object obj=new Person();

Object obj=new Dog();

Object obj = “hello”;

（2）Object类型的形参可以接收任何类型的实参

例如：void method(Object obj){}

调用：method(new Person());

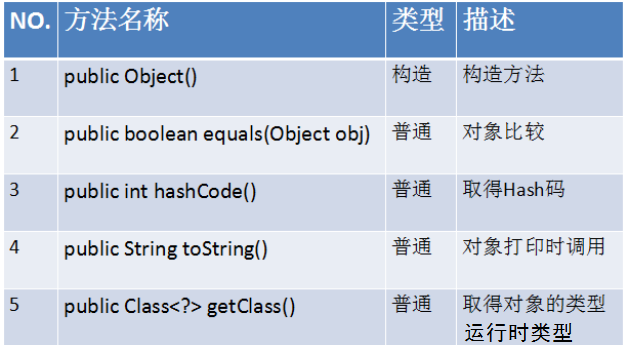
method(new Computer());

method("hello");

method(null);

（3）数组的多态性：当数组的元素类型是Object时，表示这个数组可以存放任意类型的对象

### 7.1.2 Object类中的主要方法



#### 1、==与equlas()方法

1、==运算符

（1）基本数据类型比较值:只要两个变量的值相等，即为true

注意：当基本数据类型使用“==”进行比较时，符号两边的数据必须类型兼容（类型相同或可自动类型转换），否则编译出错；

（2）引用类型比较引用：比较两个对象的地址值，两个引用变量指向同一个对象的堆空间时，才为true

Person p1=new Person();

Person p2=new Person();

if(p1==p2) //false

p1=p2;//如果有这样的赋值语句后 if(p1==p2) //true

注意：当引用类型使用“==”进行比较时，符号两边的数据类型必须兼容（类型相同或有父子类关系），否则编译出错；

Person p=new Person();

String str="hello";

if(p==str)//错误，p和String类型不兼容，Person和String类型无父子类关系

Student stu = new Student();//Student extends Person

if(p==stu)//编译通过

1. equals方法

格式:obj1.equals(obj2)

（1）equals()：所有类都继承了Object，也就获得了equals()方法。

（2）自定义类可以重写equals方法

注意：如果一个类型没有重写过equals方法，那么调用equals等价于==

说明：当用equals()方法进行比较时，对类File、String、Date及包装类（Wrapper Class）来说，是比较类型及内容；原因是在这些类中重写了Object类的equals()方法。

重写equals()方法的原则：

* 对称性：如果x.equals(y)返回是“true”，那么y.equals(x)也应该返回是“true”。
* 自反性：x.equals(x)必须返回是“true”。
* 类推性：如果x.equals(y)返回是“true”，而且y.equals(z)返回是“true”，那么z.equals(x)也应该返回是“true”。
* 一致性：如果x.equals(y)返回是“true”，只要x和y内容一直不变，不管你重复x.equals(y)多少次，返回都是“true”。
* 任何情况下，x.equals(null)，永远返回是“false”；x.equals(和x不同类型的对象)永远返回是“false”。

#### 2、hashcode方法

* 如果两个对象的hashCode值不相等，那么这两个对象一定不相等，即调用==和equals返回true
* 如果两个对象的hashCode值相等，那么这两个对象可能相等，可能不相等，即调用==和equals方法可能返回true，可能false
* 如果两个对象调用==或equals相等，那么这两个对象的hashCode值一定相等。

|  |
| --- |
| public static void main(String[] args) {  String s1 = "Aa";  String s2 = "BB";  System.out.println(s1.hashCode());//2112  System.out.println(s2.hashCode());//2112  } |

#### 3、toString()方法

1、默认从Object类继承的toString()方法返回的是

getClass().getName() + '@' + Integer.toHexString(hashCode())

即该对象的类名称@该对象hashcode哈希码的无符号十六进制表示

2、在进行String与其它类型数据的连接操作时，自动调用toString()方法

Date now=new Date();

System.out.println(“now=”+now); 相当于

System.out.println(“now=”+now.toString());

3、可以根据需要在用户自定义类型中重写toString()方法

如String 类重写了toString()方法，返回字符串的值。

s1=“hello”;

System.out.println(s1);//相当于System.out.println(s1.toString());

4、基本类型数据转换为String类型时，调用了对应包装类的toString()方法

int a=10; System.out.println(“a=”+a);

#### 4、finalize()方法

A：这个方法不是程序员手动调用的，而是垃圾回收器调用，当该对象被垃圾回收器回收之前调用（留临终遗言）

B：每一个对象的finalize方法只能被调用一次

因为如果某个对象重写的finalize()中有一个变量再次引用了this当前对象，那么当前对象就复活了，垃圾回收器这次就不能将其回收。

但是当它下次再变成垃圾后，垃圾回收器再碰到它，就直接回收，不再调用它的finalize()方法。

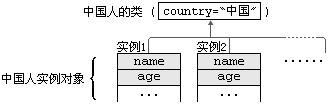
C：一般情况下不需要重写，只有这个类是表示某种连接资源时，例如FileInputStream或FileOutputStream，可能需要重写，使得会被回收之前，以配置系统资源或执行其他清除。

|  |
| --- |
|  |

## 7.2 static关键字

### 1、为什么要使用static关键字

当我们编写一个类时，其实就是在描述其对象的属性和行为，而并没有产生实质上的对象，只有通过new关键字才会产生出对象，这时系统才会分配内存空间给对象，其方法才可以供外部调用。我们有时候希望无论是否产生了对象或无论产生了多少对象的情况下，某些特定的数据在内存空间里只有一份，例如所有的中国人都有个国家名称，每一个中国人都共享这个国家名称，不必在每一个中国人的实例对象中都单独分配一个用于代表国家名称的变量。



### 2、类属性、类方法的设计思想

类属性（类变量）作为该类各个对象之间共享的变量。在设计类时,分析哪些类属性不因对象的不同而改变，将这些属性设置为类属性。相应的方法设置为类方法。

如果方法与调用者无关，则这样的方法通常被声明为类方法，由于不需要创建对象就可以调用类方法，从而简化了方法的调用。

练习：

编写一个类实现银行活期类账户，包含的属性有“帐号”、“密码”、“存款余额”、“年利率”、“最小余额”，定义封装这些属性的方法。编写主类，使用银行账户类，输入、输出3个储户的上述信息。

考虑：哪些属性可以设计成static属性。

### 3、static修饰的范围

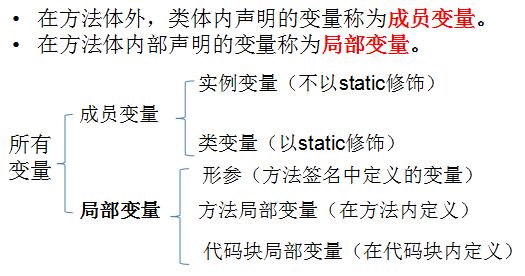
使用范围：在Java类中，可用static修饰属性、方法、代码块、内部类

#### （1）类变量(class Variable)

当某个属性的值是所有对象共享的，那么这样的属性应该声明为静态的，这样的属性我们称为类变量。

被修饰后的成员具备以下特点：

* 随着类的初始化而初始化，优先于实例对象的创建
* 修饰的成员，被所有对象所共享
* 访问权限允许时，可不创建对象，直接“类名.xx”调用
* 类变量的值存储在方法区
* 类变量的get/set方法也是静态的



#### （2）类方法(class Method)

当方法的功能实现、调用和该类的对象无关时，我们可以把这样的方法声明为static，我们称为静态方法。

* static修饰的方法，只有访问权限允许，可以通过"类名.方法"访问，因此static的方法也称为类方法。
* 在static方法内部只能访问类变量与类方法（static属性和方法），不能访问类的实例变量和成员方法（非static属性和方法）。
* static方法内部不能有this和super。
  + 如果在静态方法中有局部变量与类变量重名，可以使用“本类名.xxx”
  + 如果在静态方法中想要访问父类的静态成员，可以使用“父类名.xxx”
* static修饰的方法不能被重写，或者说不会被覆盖

因为静态方法是类方法，是属于类的，和对象无关

### 4、静态与非静态的访问原则（☆）

（1）同一个类中：

* 静态成员中**不能访问**非静态的成员
* 非静态成员中**可以访问**静态成员

（2）不同类中：

* 访问其他类的非静态成员必须使用“对象.非静态成员”的格式
* 访问其他类的静态成员**建议使用“类名.静态成员”**的格式，也可以使用“对象.静态成员”

## 7.3 类的成员之四：代码块

初始代码块：一般来说是用于类或实例成员变量的初始化，分为静态代码块和非静态代码块（构造代码块）。

### 1、静态代码块

1. 可以为类变量（静态的属性）初始化
2. 随着类的初始化而初始化，只执行一次
3. 如果子类初始化时，它的父类没有初始化会先初始化父类
4. 在静态代码块中不能访问非静态成员（属性、方法、内部类）
5. 在静态代码块中不能使用this和super关键字

如果有重名问题，需要访问本类或父类的静态成员，那么可以使用“本类名.xx”、“父类名.xx”

如果没有重名问题，直接访问即可。

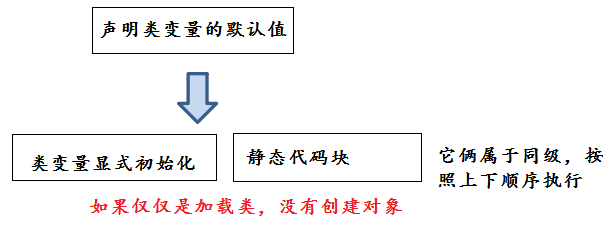
### 2、非静态代码块（构造块）

1. 可以为实例变量（非静态的属性）初始化
2. 随着对象的创建而初始化，每创建一个对象，就执行一次
3. 创建子类对象时，需要先为从父类继承的属性进行初始化，所以会导致父类的构造块和构造器先执行
4. 本类的非静态代码块优先于构造器执行

### 3、初始化顺序

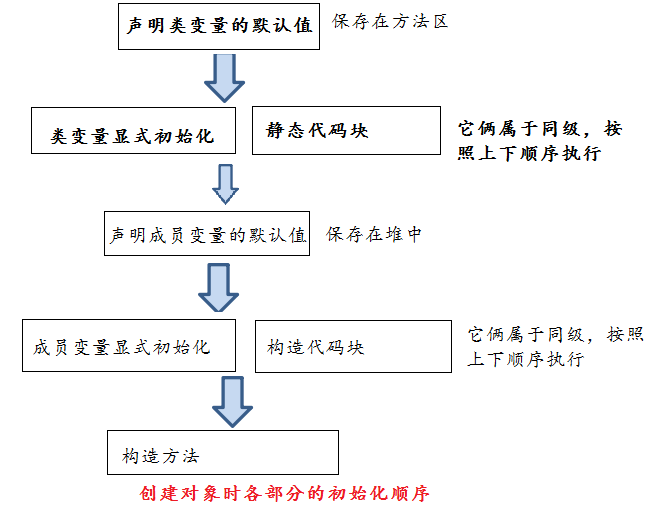
#### （1）类初始化

如果不需要创建对象，那么紧紧是类初始化



#### （2）创建对象

如果要创建对象，那么要先看类是否之前初始化过，如果没有，那么要先初始化类，然后才能创建对象。



#### （3）总结升华（☆☆☆）

A：编译器会把“类变量的显式初始化”、“静态代码块”的内容按顺序合并为一个<clinit>的类初始化方法。所以类的初始化过程就是执行<clinit>方法，即类变量的显式初始化与静态代码块一定是一起执行的，而且因为类初始化只要一次，因此<clinit>只有一个，且只会执行一次。

B：执行子类的<clinit>方法时，如果父类也没有初始化，会先执行父类的<clinit>方法。

C：编译器会把“非静态变量的显示初始化”、“非静态代码块”的内容按顺序合并到**每一个**构造器中，构成一个个的<init>实例初始化方法，有几个构造器，就有几个<init>方法。而且“非静态变量的显示初始化”、“非静态代码块”的内容在上面，原来构造器中的代码在下面。因此对象的初始化，即执行它的<init>方法，即非静态变量的显示初始化、非静态代码块和构造器中的代码是一起执行的，而且非静态变量的显示初始化、非静态代码块的内容先于构造器的代码执行。

D：执行子类的<init>方法时，一定会先调用父类的<init>方法，因为要先为从父类继承的属性先完成初始化。（super([实参列表])）

E：记住不管是类变量还是实例变量都有默认值，再用<clinit>或<init>初始化之前，就有默认值。

### 4、练习

#### （1）练习1

|  |  |
| --- | --- |
| **package** com.atguigu.exer1;  **public** **class** TestStaticExer1 {  **public** **static** **void** main(String[] args) {  Son son = **new** Son();  }  }  **class** Father{  **static**{  System.***out***.println("（1）父类的静态代码块");  }  {  System.***out***.println("（2）父类的构造器");  }  Father(){  System.***out***.println("（3）父类的无参构造");  }  }  **class** Son **extends** Father{  **static**{  System.***out***.println("（4）子类的静态代码块");  }  {  System.***out***.println("（5）子类的构造器");  }  Son(){  System.***out***.println("（6）子类的无参构造");  }  } | (1)(4)(2)(3)(5)(6) |

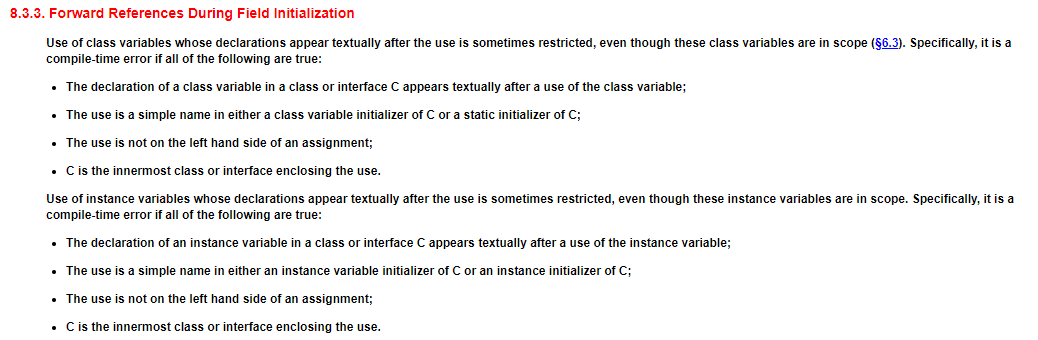
#### （2）练习2

|  |  |
| --- | --- |
| **package** com.atguigu.exer2;  **public** **class** TestStaticExer2 {  **public** **static** **void** main(String[] args) {  Zi zi = **new** Zi();  }  }  **class** Fu{  **private** **static** **int** *i* = *getNum*("（1）i");  **private** **int** j = *getNum*("（2）j");  **static**{  *print*("（3）父类静态代码块");  }  {  *print*("（4）父类构造代码块");  }  Fu(){  *print*("（5）父类构造器");  }  **public** **static** **void** print(String str){  System.***out***.println(str + "->" + *i*);  }  **public** **static** **int** getNum(String str){  *print*(str);  **return** ++*i*;  }  }  **class** Zi **extends** Fu{  **private** **static** **int** *k* = *getNum*("（6）k");  **private** **int** h = *getNum*("（7）h");  **static**{  *print*("（8）子类静态代码块");  }  {  *print*("（9）子类构造代码块");  }  Zi(){  *print*("（10）子类构造器");  }  **public** **static** **void** print(String str){  System.***out***.println(str + "->" + *k*);  }  **public** **static** **int** getNum(String str){  *print*(str);  **return** ++*k*;  }  } | （1）i->0  （3）父类静态代码块->1  （6）k->0  （8）子类静态代码块->1  （2）j->1  （4）父类构造代码块->2  （5）父类构造器->2  （7）h->1  （9）子类构造代码块->2  （10）子类构造器->2 |

#### （3）练习3

|  |  |
| --- | --- |
| **package** com.atguigu.exer3;  **public** **class** TestStaticExer3 {  **public** **static** **void** main(String[] args) {  MyClass obj = **new** MyClass();  }  }  **class** MyClass{  **static**{  *i* = 100;//可以  // i++;//错误  MyClass.*i*++;  // System.out.println("（1）静态代码块 i=" + i);//错误  System.***out***.println("（1）静态代码块 i=" + MyClass.*i*);  }  {  j = 100;  // j++;  **this**.j++;  // System.out.println("（2）构造代码块j=" + j);  System.***out***.println("（2）构造代码块j=" + **this**.j);  }  MyClass(){  j = 200;  j++;  System.***out***.println("（3）构造器j=" + j);  }  **private** **static** **int** *i* = *getNum*("（4）i");  **private** **int** j = *getNum*("（5）j");    **public** **static** **void** print(String str){  System.***out***.println(str + "->" + *i*);  }  **public** **static** **int** getNum(String str){  *print*(str);  **return** ++*i*;  }  } | （1）静态代码块 i=101  （4）i->101  （2）构造代码块j=101  （5）j->102  （3）构造器j=201 |

查看：<https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.3.2>



#### （4）练习4

|  |  |
| --- | --- |
| **package** com.atguigu.exer4;  **public** **class** T {  **public** **static** **int** *k* = 0;  **public** **static** T *t1* = **new** T("t1");  **public** **static** T *t2* = **new** T("t2");  **public** **static** **int** *i* = *print*("i");  **public** **static** **int** *n* = 99;    **public** **int** j = *print*("j");  {  *print*("构造块");  }  **static**{  *print*("静态块");  }  **public** T(String str){  System.***out***.println((++*k*) + ":" + str + " i=" + *i* + " n=" + *n*);  ++*n*;  ++*i*;  }  **public** **static** **int** print(String str){  System.***out***.println((++*k*) + ":" + str + " i=" + *i* + " n=" + *n*);  ++*n*;  **return** ++*i*;  }  **public** **static** **void** main(String[] args) {    }  } | 1:j i=0 n=0  2:构造块 i=1 n=1  3:t1 i=2 n=2  4:j i=3 n=3  5:构造块 i=4 n=4  6:t2 i=5 n=5  7:i i=6 n=6  8:静态块 i=7 n=99 |

## 7.4 其他关键字

### 1、final修饰符

在Java中声明类、属性和方法时，可使用关键字final来修饰,表示“最终”。

* final标记的类（太监类）不能被继承。提高安全性，提高程序的可读性。

String类、System类、StringBuffer类

* final标记的方法不能被子类重写。

Object类中的getClass()。

* final标记的变量(成员变量或局部变量)即称为常量。名称大写，且只能被赋值一次。

final标记的成员变量必须在声明的同时或在每个构造方法中或代码块中显式赋值，然后才能使用。

final double PI=3.14;

### 2、native关键字

使用native关键字说明这个方法是原生函数，也就是这个方法是用C/C++等非Java语言实现的，并且被编译成了DLL，由java去调用。

（1）为什么要用native方法

java使用起来非常方便，然而有些层次的任务用java实现起来不容易，或者我们对程序的效率很在意时，问题就来了。例如：有时java应用需要与java外面的环境交互。这是本地方法存在的主要原因，你可以想想java需要与一些底层系统如操作系统或某些硬件交换信息时的情况。本地方法正是这样一种交流机制：它为我们提供了一个非常简洁的接口，而且我们无需去了解java应用之外的繁琐的细节。

（2）native声明的方法，对于调用者，可以当做和其他Java方法一样使用

一个native method方法可以返回任何java类型，包括非基本类型，而且同样可以进行异常控制。

native method的存在并不会对其他类调用这些本地方法产生任何影响，实际上调用这些方法的其他类甚至不知道它所调用的是一个本地方法。JVM将控制调用本地方法的所有细节。

如果一个含有本地方法的类被继承，子类会继承这个本地方法并且可以用java语言重写这个方法（如果需要的话）。

### 3、关于关键字常见问题

1. 构造器只能有访问控制修饰符
2. 外部类不能使用static修饰，可以使用final修饰
3. 属性可以同时使用static和final修饰，叫做全局常量
4. 方法可以同时使用static和final修饰，都不能被重写
5. 局部变量只能使用final修饰
6. 代码块只能使用static修饰

## 7.5 抽象类

类用于描述现实生活中一类事物。类中有属性、有方法等成员。

某种情况下，父类只能知道子类应该具备一个怎样的方法，但是不能够明确知道如何实现该方法。

只能在子类中才能确定如何去实现方法体。

例如：定义一个几何图形。所有几何图形都应该具备一个计算面积的方法。

但是不同的几何图形计算面积的方式完全不同。

Java中为上述问题提供了相应的解决办法：

Java允许父类中只是提供一个方法的声明（方法的签名），但是不提供具体的实现。

具体的实现由子类来完成，该方法被称为“抽象方法”。

拥有一个或者多个抽象方法的类，被称为“抽象类”。

### 1、抽象方法

用abstract来修饰一个方法时，该方法叫做抽象方法。

抽象方法：只有方法的声明，没有方法的实现，以分号结束：

[访问控制修饰符] abstract 返回值类型 方法名（[形参列表]）;

1. 含有抽象方法的类必须是抽象类
2. 子类继承抽象类时，子类也会继承父类的抽象方法，因此必须对父类的抽象方法进行重写（实现），否则子类也得是抽象类

### 2、抽象类

使用abstract修饰的类称为“抽象类”

格式： [访问控制修饰符] abstract class 抽象类名 [extends 父类名]{

}

（1）拥有一个或者多个抽象方法的类“必须”是抽象类

（2）抽象类不能直接创建对象，即不能实例化

（3）有时抽象类中没有抽象方法，（目的只有一个：不让你创建对象）

（4）抽象类就是用来被继承的，子类继承抽象类时，必须对父类的抽象方法进行实现，否则子类也得是抽象类

（5）抽象类也是类，因此原来类中可以有的成员，抽象类都可以有，那么抽象类不能直接创建对象，为什么还有构造器呢？供子类调用，子类创建对象时，需要为从父类继承的属性初始化

问：抽象类与普通类的区别？

\* （1）抽象类是使用abstract修饰，

\* 普通类没有abstract修饰

\* （2）抽象类不能实例化

\* 普通类可以实例化

\* （3）抽象类中可能有一个或多个抽象方法

\* 普通类不能有抽象方法

### 3、不能和abstract一起使用的关键字

abstract只能修饰类与方法

\* （1）abstract和final

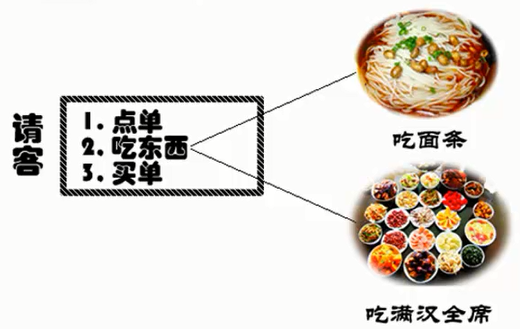
\* （2）abstract和static

\* （3）abstract和private

abstract不能修饰属性、构造器、局部变量、代码块

### 4、抽象类的应用：模板方式设计模式

场景（一）：



场景（二）：客户到银行办理业务

* 取号排队
* 办理具体现金、转账、企业、个人、理财业务等
* 给银行工作人员评分

模板方法模式介绍：

模板方法模式是编程中经常用得到的模式。当处理某个流程的代码已经都具备，但是其中某个节点的代码暂时不能确定或易变，我们就可以使用模板方法模式。在父类处理流程的方法中，定义操作的算法骨架，将某些不能确定的步骤代码或易变的代码抽取成一个抽象方法，然后延迟到子类中实现。

开发中常见的场景：

（1）工作流中

（2）各个框架、类库中也都有他的影子，比如常见的有：

* 数据库访问的封装
* Junit单元测试
* JavaWeb的Servlet中关于doGet/doPost方法调用
* Hibernate中模板程序
* Spring中JDBCTemlate、HibernateTemplate等

编写一个计算某段代码执行时间的方法

\* 提示：System.currentTimeMillis()返回当前系统时间，以毫秒返回

|  |
| --- |
| **public** **class** TestTemplate {  **public** **static** **void** main(String[] args) {  MyCalTime m = **new** MyCalTime();  m.getTime();  }  }  **class** MyCalTime **extends** CalTime {  **protected** **void** method() {  **long** sum = 0;  **for** (**int** i = 0; i <= 10000000; i++) {  sum += i;  }  System.***out***.println("结果：" + sum);  }  }  **abstract** **class** CalTime {  **public** **final** **void** getTime() {  // 先获取开始时间  **long** start = System.*currentTimeMillis*();    method();  // 先获取结束时间  **long** end = System.*currentTimeMillis*();  System.***out***.println("耗时：" + (end - start) + "毫秒");  }  **protected** **abstract** **void** method();  } |

方法回调（钩子方法）：

好莱坞原则：”Don’t call me,we’ll call you back”

在好莱坞，当艺人把简历递交给好莱坞的娱乐公司时，所能做的就是等待，整个过程由娱乐公司控制，演员只能被动的服务安排，在需要的时候再由公司安排具体环节的演出。

在软件开发中，我们可以将call翻译为调用。子类不能调用父类，而通过父类调用子类。这些调用步骤已经在父类中写好了，完全由父类控制整个过程。

## 7.6接口

### 1、接口的概念

1、有时必须从几个类中派生出一个子类，继承它们所有的特征。但是，Java不支持多重继承。有了接口，就可以得到多重继承的效果。

例如：电视机，既能作为传统的电视节目接收播放器，又能作为录像视频播放器，还能作为上网的显示器。

2、有时必须从几个类中抽取出一些共同的行为特征，而它们之间又没有is-a的关系，仅仅是具有相同的行为特征而已。

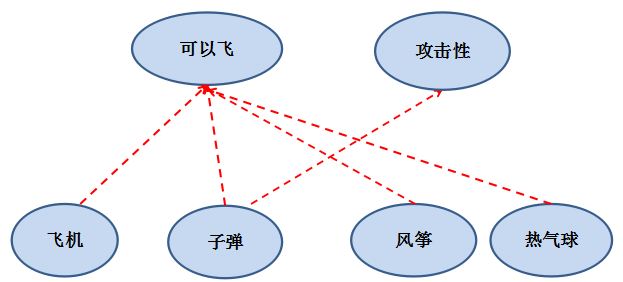
例如：鼠标、键盘、打印机、扫描仪、摄像头、充电器、MP3机、手机、数码相机、移动硬盘等都支持USB连接。

所以，接口就是一个规范，一个标准。

之前我们使用继承描述is-a的关系。例如：动物、哺乳动物、爬行动物、人等

现在如果要描述：飞机、子弹、导弹、石头、热气球、风筝等它们之间的关系呢

它们具有共同的行为：可以飞



### 2、如何声明接口

用interface关键字来定义

格式：访问控制修饰符 interface 接口名{}

### 3、如何使用接口

（1）类实现接口：用implements关键字

格式：[修饰符] class 实现类名 implements 接口名1,接口名2...{ }

（2）接口继承接口：用extends关键字

格式：[修饰符] interface 子接口名 extends 接口名1,接口名2...{ }

### 4、接口的特点

1、接口不能直接实例化

2、接口中的成员：

JDK1.8之前：接口是抽象方法和常量值的定义的集合。

接口中的所有成员变量都默认是由public static final修饰的。

接口中的所有方法都默认是由public abstract修饰的。

JDK1.8之后：（见JDK1.8接口的新特性）

3、接口没有构造器、代码块，因为接口中没有类变量和实例变量需要初始化，接口中只有常量

4、接口就是用来被实现的，当类实现了接口时，该类需要实现接口所有的抽象方法，否则该类也得是抽象类，不然编译报错。一个类只能继承一个父类，但可以同时实现多个接口。如果一个类既要继承父类又要实现接口，那么继承在前，实现在后：

访问控制修饰符 class 类名 extends 父类 implements 接口1，接口2。。。{

//...

}

5、一个接口可以继承其他一个或多个接口

6、与继承关系类似，接口与实现类的对象之间构成多态引用

|  |
| --- |
| **interface** Flyable{  **long** ***MAX\_SPEED*** = 7900000;//这里单位是毫米/秒，7.9千米/秒，超过这个速度，就变成卫星  **long** ***MIN\_SPEED*** = 1;//1毫米/秒  **void** fly();  }  **class** Plane **implements** Flyable{  **public** **void** fly() {  System.***out***.println("飞机靠发动机飞行");  }  }  **class** Kite **implements** Flyable{  **public** **void** fly() {  System.***out***.println("风筝靠风飞行");  }  }  **class** FireBalloon **implements** Flyable{  **public** **void** fly() {  System.***out***.println("热气球靠热空气飞行");  }  }  **interface** Attack{  **void** attack();  }  **class** Bullet **implements** Flyable,Attack{  **public** **void** fly(){  System.***out***.println("靠助推器飞行");  }  **public** **void** attack() {  System.***out***.println("子弹可以攻击");  }  } |
| **public** **static** **void** main(String[] args) {  Plane p = **new** Plane();//本态引用  Flyable fp = p;//多态引用  Flyable fk = **new** Kite();//多态引用  Flyable fs = **new** Bullet();//多态引用 ，子弹是Flyable接口的实现类  Attack a = **new** Bullet();//多态引用，子弹是攻击性武器的实现类  Object o = **new** Bullet();//多态引用，子弹是Object类的子类  } |

### 5、为什么要定义接口？

接口的本质是契约，标准，规范，就像我们的法律一样。制定好后大家都要遵守，也因为有规范，变得有“法”可依。项目的具体需求是多变的，我们必须以不变应万变才能从容开发，此处的“不变”就是“规范”。因此，我们开发项目往往都是面向接口编程！

接口就是比“抽象类”还“抽象”的“抽象类”，可以更加规范的对子类进行约束。全面的专业的实现了：规范和具体实现的分离。

继承表现的是一个"是不是 is - a"的关系，而接口体现了现实世界中“如果你是/要...则必须能/有...”的思想。

例如：你要工具类帮你的对象排序，你的类就要实现java.lang.Comparable接口或Comparator接口

你的对象要序列化，你的类就要实现java.io.Serializable接口

你的对象要克隆，你的类就要实现java.lang.Cloneable接口

### 6、JDK1.8接口的新特性

Java 8中，你可以为接口添加默认方法和静态方法。从技术角度来说，这是完全合法的，只是它看起来违反了接口作为一个抽象定义的理念。

#### （1）默认方法

默认方法：默认方法使用 default 关键字修饰。可以通过实现类对象来调用。

比如：java 8 API中对Iterable、Collection、List、Comparator等接口提供了丰富的默认方法。

|  |
| --- |
| interface A{  default void methodA(){ //此处public可以省略  System.out.println("A的默认实现");  }  } |
| class SubA implements A{    } |
| new SubA().methodA(); |

#### （2）静态方法

静态方法：使用 static 关键字修饰。可以通过接口名直接调用静态方法。

|  |
| --- |
| interface A{  public static void testA(){//public可以省略  System.out.println("A的静态方法");  }  } |
| A.testA(); |

我们之前经常在一些与接口一起相互使用的类中使用静态方法。你可以在标准库中找到像Collection/Collections或者Path/Paths这样成对的接口和类。

例如：

以Paths 类为例，它只有一些工厂方法。在Java 8 中，如果在Path 接口中添加这样一个方法：这样Paths 类就再没有存在的必要了。

|  |
| --- |
| public final class **Paths** {  private Paths() { }  public static Path get(String first, String... more) {  return FileSystems.getDefault().getPath(first, more);  }  ....  } |
| public interface **Path** {  public static Path get(String first, String... more) {  return FileSystems.getDefault().getPath(first, more);  }  ...  } |

例如：在Collections中声明了如下方法：

|  |
| --- |
| public static <T> List<T> nCopies(int n, T o) // 构造含有n 个o 类型实例的列表 |

它可以作为List 接口的一个静态方法。这样你就可以调用List.nCopies(10,"Fred")，而不是Collections.nCopies(10, "Fred")，就能够更清楚地表示返回的结果是一个List 对象。

#### （3）特殊情况：接口中的默认方法冲突问题

（1）当一个实现类实现了多个接口，而多个接口中出现了相同的默认方法（方法名与形参列表相同）时，实现类必须重写该默认方法，以解决冲突问题。即放弃接口中的默认实现，改为实现类自己提供实现，如果想要保留其中一个父接口中的方法实现，那么只能通过“父接口.super.默认方法”的方式

|  |
| --- |
| interface Filial{//孝顺的  default void help(){  System.out.println("老妈，我来救你了");  }  }  interface Spoony{//痴情的  default void help(){  System.out.println("老婆，别怕，我来了");  }  }  class Man implements Filial,Spoony{  @Override  public void help() {  // Filial.super.help();  // Spoony.super.help();  System.out.println("快跑吧");  }    } |

（2）当一个子类在继承某个父类，同时实现一个父接口时，如果父类中出现了和接口中的默认方法相同（方法名与形参列表相同）时，那么子类默认选择父类中的方法，而忽略接口中的默认方法。当然程序员也可以做出自己的选择。

|  |
| --- |
| public class TestDefault {  public static void main(String[] args) {  Man m = new Man();  m.help();  }  }  interface Filial{//孝顺的  default void help(){  System.out.println("老妈，我来救你了");  }  }  interface Spoony{//痴情的  default void help(){  System.out.println("老婆，别怕，我来了");  }  }  class Father{  public void help(){  System.out.println("儿子，救我老婆");  }  }  class Man extends Father implements Filial,Spoony{    } |

### 7、小结

1、类与接口的关系：

（1）类与类之间是继承，而且是单继承

（2）类与接口之间是实现，可以多实现

（3）接口与接口之间是继承，可以多继承

2、抽象类与接口的区别？

接口就是比“抽象类”还“抽象”的“抽象类”

1.语法层面上的区别

1）抽象类使用abstract class声明，接口使用interface声明

2）抽象类可以提供成员方法的实现细节，而JDK1.8之前接口中只能存在public abstract 方法，JDK1.8之后可以有默认实现和静态方法；

　　3）接口中的字段只能是public static final类型的；而抽象类中的成员变量没有这个要求

1. 接口中不能含有代码块，而抽象类可以有代码块

5）抽象类必须有构造器，而接口没有构造器

　　6）一个类只能继承extends一个抽象类，而一个类却可以实现implements多个接口。

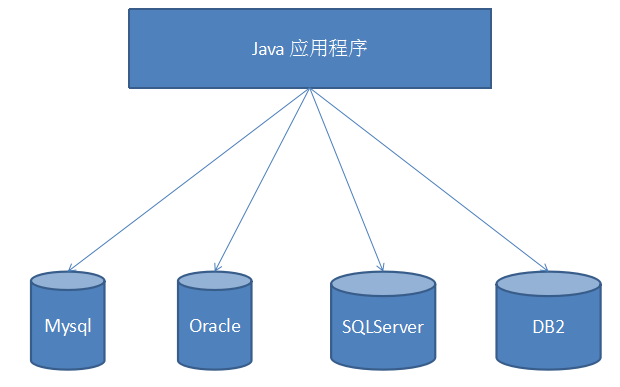
2.设计层面上的区别

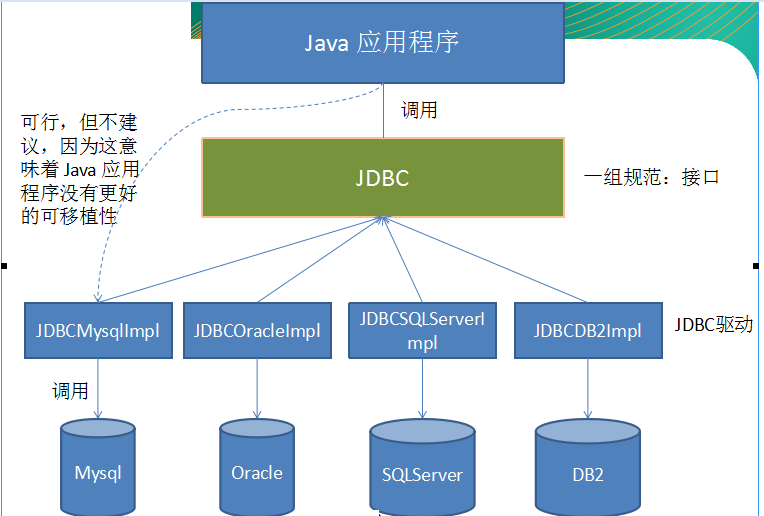
1）抽象类是对一种事物的抽象，即对类抽象，包括属性、行为，但是接口却是对类局部（行为）进行抽象。

2）抽象类作为很多子类的父类，它是一种模板式设计。而接口是一种行为规范，它是一种辐射式设计。

### 8、接口的应用（了解）

#### （1）接口的应用：JDBC





#### （2）接口的应用：工厂模式（了解）

**工厂模式：实现了创建者与调用者的分离**

**其实设计模式和面向对象设计原则都是为了使得开发项目更加容易扩展和维护，解决方式就是一个“分工”。**

社会的发展也是这样，分工越来越细。

原始社会的人：人什么都要要会，自己种植，自己打猎，自己织衣服，自己治病 --》全能型人才

现在的人：可以只会一样，其他都不会，只会Java也能活，不会做饭，不会开车，不会.... -->专业型人才

**面向对象的设计原则**（总共六个）：这里说几个和工厂模式相关的

* OCP（开闭原则，Open-Closed Principle）：一个软件的实体应当对扩展开放，对修改关闭。

当我们写完的代码，不能因为需求变化就修改。我们可以通过新增代码的方式来解决变化的需求。如果每次需求变动都去修改原有的代码，那原有的代码就存在被修改错误的风险，当然这其中存在有意和无意的修改，都会导致原有正常运行的功能失效的风险，这样很有可能会展开可怕的蝴蝶效应，使维护工作剧增。

说到底，开闭原则除了表面上的可扩展性强以外，在企业中更看重的是维护成本。

所以，开闭原则是设计模式的第一大原则，它的潜台词是：控制需求变动风险，缩小维护成本。

* DIP（依赖倒转原则，Dependence Inversion Principle）：要针对接口编程，不要针对实现编程。

如果A中关联B，那么尽量使得B实现某个接口，然后A与接口发生关系，不与B实现类发生关联关系。

依赖倒置的潜台词是：面向抽象编程，解耦调用和被调用者。

* LOD（迪米特法则，Law Of Demeter）：只与你直接的朋友通信，而避免和陌生人通信。

要求尽量的封装，尽量的独立，尽量的使用低级别的访问修饰符。这是封装特性的典型体现。

一个类如果暴露太多私用的方法和字段，会让调用者很茫然。并且会给类造成不必要的判断代码。所以，我们使用尽量低的访问修饰符，让外界不知道我们的内部。这也是面向对象的基本思路。这是迪米特原则的一个特性，无法了解类更多的私有信息。

另外，迪米特原则要求类之间的直接联系尽量的少，两个类的访问，通过第三个中介类来实现。

迪米特原则的潜台词是：不和陌生人说话，有事去中介。

工厂模式的分类：

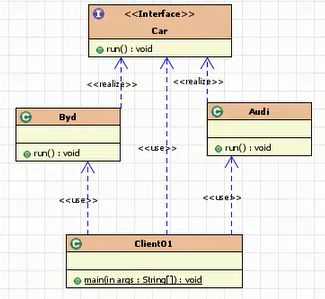
* 简单工厂模式：用来生产同一等级结构中的任意产品。（对于增加新的产品，可能需要修改已有代码）
* 工厂方法模式：用来生产同一等级结构中的固定产品。（支持增加任意产品）
* 抽象工厂模式：用来生产不同产品族的全部产品。（对于增加新的产品，无能为力；支持增加产品族）

**核心本质：**

用调用工厂方法获取实例对象，代替直接new操作。将调用者跟我们的实现类解耦。

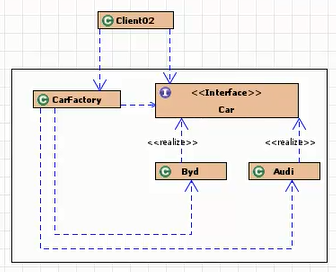
1、无工厂模式

|  |
| --- |
| **package** com.atguigu.pattern.factory.nofactory;  **public** **class** TestNoFactory {  **public** **static** **void** main(String[] args) {  Car a = **new** Audi();  Car b = **new** BYD();  a.run();  b.run();  }  }  **interface** Car{  **void** run();  }  **class** Audi **implements** Car{  **public** **void** run() {  System.***out***.println("奥迪在跑");  }  }  **class** BYD **implements** Car{  **public** **void** run() {  System.***out***.println("比亚迪在跑");  }  } |



1、简单工厂模式

|  |
| --- |
| **package** com.atguigu.pattern.factory.simple;  **public** **class** TestSimple {  **public** **static** **void** main(String[] args) {  Car a = CarFactory.*getAudi*();  Car b = CarFactory.*getByd*();  a.run();  b.run();  }  }  **interface** Car{  **void** run();  }  **class** Audi **implements** Car{  **public** **void** run() {  System.***out***.println("奥迪在跑");  }  }  **class** BYD **implements** Car{  **public** **void** run() {  System.***out***.println("比亚迪在跑");  }  }  **class** CarFactory{  /\*public Car getCar(String type){  if("奥迪".equals(type)){  return new Audi();  }else if("比亚迪".equals(type)){  return new BYD();  }else{  return null;  }  //增加新产品，需要修改代码  }\*/  //增加新产品，需要增加方法，方法过多  **public** **static** Car getAudi(){  **return** **new** Audi();  }  **public** **static** Car getByd(){  **return** **new** BYD();  }  } |



调用者只要知道他要什么，从哪里拿，如何创建，不需要知道。分工，多出了一个专门生产Car的实现类对象的工厂类。把调用者与创建者分离。

小结：

简单工厂模式也叫静态工厂模式，就是工厂类一般是使用静态方法，通过接收的参数的不同来返回不同的实例对象，或者用不同的方法返回不同的实例对象。

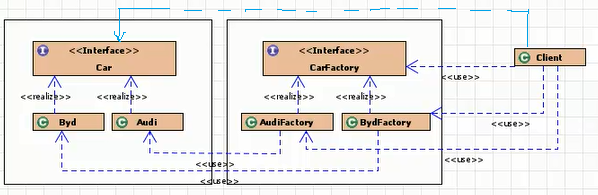
对于增加新产品，要么要修改代码，要么要增加方法。

2、工厂方法模式

为了避免简单工厂模式的缺点，不完全满足OCP（对扩展开放，对修改关闭）。

工厂方法模式和简单工厂模式最大的不同在于，简单工厂模式只有一个（对于一个项目或者一个独立的模块而言）工厂类，而工厂方法模式有一组实现了相同接口的工厂类。

|  |
| --- |
| **package** com.atguigu.pattern.factory.method;  **public** **class** TestFactoryMethod {  **public** **static** **void** main(String[] args) {  Car a = **new** AudiFactory().getCar();  Car b = **new** BydFactory().getCar();  a.run();  b.run();  }  }  **interface** Car{  **void** run();  }  **class** Audi **implements** Car{  **public** **void** run() {  System.***out***.println("奥迪在跑");  }  }  **class** BYD **implements** Car{  **public** **void** run() {  System.***out***.println("比亚迪在跑");  }  }  **interface** Factory{  Car getCar();  }  **class** AudiFactory **implements** Factory{  **public** Audi getCar(){  **return** **new** Audi();  }  }  **class** BydFactory **implements** Factory{  **public** BYD getCar(){  **return** **new** BYD();  }  } |



根据设计理论建议，选择工厂方法设计模式，但是根据复杂度来说，实际工作更多选择简单工厂模式。

#### （3）接口的应用：代理设计模式（了解）

代理模式（Proxy pattern）：

核心作用：

通过代理，控制对对象的访问！

可以详细控制访问某个（某类）对象的方法，在调用这个方法前做前置处理，调用这个方法后做后置处理。（即：AOP的微观实现）。从而实现将统一流程代码放到代理类中处理。代理模式是AOP（Aspect Oriented Programming面向切面编程）的核心实现机制！

应用场景：

* 安全代理：屏蔽对真实角色的直接访问。
* 远程代理：通过代理类处理远程方法调用（RMI）

分类

* 静态代理（静态定义代理类）
* 动态代理（动态生成代理类）：JDK自带的动态代理，需要反射等知识

代理模式的核心角色：

* 抽象角色：主题

定义代理角色和真实角色的公共对外方法

* 真实角色

实现抽象角色，定义真实角色所需实现的业务逻辑，供代理角色调用。

关注真正的业务逻辑！

* 代理角色

实现抽象角色，是真实角色的代理，通过调用真实角色的业务逻辑方法来实现具体功能，并可以附加自己的操作。将统一的流程控制放到代理角色中处理！

|  |  |
| --- | --- |
| **package** com.atguigu.test;  //接口  **public** **interface** IUserDao {  **void** save();  } | **package** com.atguigu.test;  //接口实现：目标对象  **public** **class** UserDaoImpl **implements** IUserDao {  **public** **void** save() {  System.***out***.println("----已经保存数据!----");  }  } |
| **package** com.atguigu.test;  //代理对象,静态代理  **public** **class** UserDaoProxy **implements** IUserDao{  //接收保存目标对象  **private** IUserDao target;    **public** UserDaoProxy(IUserDao target){  **this**.target=target;  }  **public** **void** save() {  System.***out***.println("开始事务...");  target.save();//执行目标对象的方法  System.***out***.println("提交事务...");  }  } | **package** com.atguigu.test;  /\*\*  \* 测试类  \*/  **public** **class** App {  **public** **static** **void** main(String[] args) {  //目标对象  IUserDao target = **new** UserDaoImpl();  //代理对象,把目标对象传给代理对象,建立代理关系  UserDaoProxy proxy = **new** UserDaoProxy(target);  proxy.save();//执行的是代理的方法  }  } |

## 7.7 类的成员之五：内部类Inner Class

### 1、为什么要有内部类

当一个事物的内部，还有一个部分需要一个完整的结构进行描述，而这个内部的完整的结构又只为外部事物提供服务，那么整个内部的完整结构最好使用内部类。而且声明为内部类可以直接使用访问外部类的所有的成员，包括私有的

源代码示例：JRE核心类库集合的迭代器大量使用内部类

### 2、内部类的分类

静态内部类

非静态内部类：

成员内部类

局部内部类

匿名内部类

### 3、成员内部类

#### （1）如何声明成员内部类

声明的位置：在类中方法外

格式如下：

[修饰符] class 外部类{

[修饰符] class 成员内部类{

}

}

#### （2）成员内部类的特点

* 内部类仍然是一个独立的类，在编译之后内部类会被编译成独立的.class文件，但是前面冠以外部类的类名和$符号。例如：Outer$Inner.class
* 成员内部类可以使用修饰符public,protected,default,private
* 成员内部类还可以使用final和abstract修饰
* 成员内部类中**不可以包含静态成员**
* 成员内部类可以直接使用外部类的所有成员，包括私有的。如果成员内部类有与外部类的非静态属性重名时，可以通过“外部类名.this.属性”进行区别，如果与外部类的静态属性重名时，可以通过“外部类名.类变量”进行区别

#### （3）如何使用成员内部类

* 在外部类中使用成员内部类：
  + 在外部类的静态成员中**不能**使用非静态的成员内部类
  + 在外部类的非静态成员中，直接创建内部类的对象来访问内部类的属性与方法。此时把它当做一个普通的类即可
* 在外部类的外面使用成员内部类：

需要外部类的对象才能创建成员内部类的对象

|  |
| --- |
| **package** com.innerclass.member;  **public** **class** TestMemberClass {  **public** **static** **void** main(String[] args) {  Outer out = **new** Outer();  out.outerMethod();    Outer.Inner oi = out.**new** Inner();  oi.innerMethod();    Outer.Inner obj = out.getInner();  obj.innerMethod();  }  }  **class** Outer{  **private** **int** value = 5;  **public** **static** **void** outerStaticMethod(){  /\*Inner in = new Inner();//错误  in.innerMethod();\*/  }  **public** **void** outerMethod(){  System.***out***.println("外部类的方法");  /\*Inner in = new Inner();//可以  in.innerMethod();\*/  }  **class** Inner{  **public** **void** innerMethod(){  System.***out***.println("内部类的方法");  System.***out***.println("内部类的方法访问外部类的私有成员："+value);  }  }  //通过外部类的某个方法返回内部类的对象  **public** Inner getInner(){  **return** **new** Inner();  }  } |

### 4、静态内部类

#### （1）如何声明静态内部类

声明的位置：在类中方法外

格式如下：

[修饰符] class 外部类{

[修饰符] static class 静态内部类{

}

}

#### （2）什么时候声明静态内部类

* 当内部类需要包含静态成员时
* 当想要在外部类的静态成员部分使用内部类时

#### （3）静态内部类的特点

* 内部类仍然是一个独立的类，在编译之后内部类会被编译成独立的.class文件，但是前面冠以外部类的类名和$符号。例如：Outer$Inner.class
* 静态内部类可以使用修饰符public,protected,default,private
* 静态内部类还可以使用final和abstract修饰
* 静态内部类中**可以包含静态成员**
* 静态内部类可以直接使用外部类的静态成员，包括私有的。**但不能使用外部类的非静态成员**。

静态内部类中有与外部类的静态属性重名时，如果要表示是外部类的属性，那么用“外部类名.属性”

#### （4）如何使用静态内部类

* 在外部类中使用静态内部类：就和使用其他普通类一样
  + 在外部类的静态成员中
  + 在外部类的非静态成员中
* 在外部类的外面使用成员内部类：

使用静态内部类的静态成员：外部类名.静态内部类名.静态内部类的静态成员

使用静态内部类的非静态成员：外部类名.静态内部类名 obj = new 外部类名.静态内部类名();

obj.静态内部类的非静态成员

|  |
| --- |
| **package** com.innerclass.staticinner;  **public** **class** TestStaticInnerClass {  **public** **static** **void** main(String[] args) {  Outer.StaticInnerClass.*innerStaticMethod*();  Outer.StaticInnerClass os = **new** Outer.StaticInnerClass();  os.innerMethod();  }  }  **class** Outer{  **public** **static** **void** outerStaticMethod(){  StaticInnerClass.*innerStaticMethod*();  StaticInnerClass si = **new** StaticInnerClass();  si.innerMethod();  }  **public** **static** **void** outerMethod(){  StaticInnerClass.*innerStaticMethod*();  StaticInnerClass si = **new** StaticInnerClass();  si.innerMethod();  }  **static** **class** StaticInnerClass{  **public** **static** **void** innerStaticMethod(){  System.***out***.println("内部类的静态方法");  }  **public** **void** innerMethod(){  System.***out***.println("内部类的非静态方法");  }  }  } |

### 5、局部内部类

#### （1）如何声明局部内部类

声明的位置：在外部类的方法或代码块中

格式如下：

[修饰符] class 外部类{

方法{

class 局部内部类{

}

}

}

#### （2）局部内部类的特点

* 内部类仍然是一个独立的类，在编译之后内部类会被编译成独立的.class文件“外部类的类名$数字编号内部类名.class”。
* 局部内部类和局部变量地位类似，不能使用public,protected,private,static这些成员修饰符，但是可以使用abstract或final
* 局部内部类不能使用static修饰，因此也不能包含静态成员。
* 只能在声明它的方法或代码块中使用，而且是先声明后使用。除此之外的任何地方都不能使用该类，因为作用域的问题。
* 局部内部类可以使用外部类的成员，包括私有的。但是是否可以使用外部类的非静态成员要看所在方法是否是非静态的。
* 局部内部类可以使用外部方法的局部变量，但是必须是final的。由局部内部类和局部变量的生命周期不同所致。

#### （3）如何使用局部内部类

* 只能在声明它的方法或代码块中使用，而且是先声明后使用。除此之外的任何地方都不能使用该类。
* 但是它的对象可以通过外部方法的返回值返回使用，返回值类型只能是局部内部类的父类或父接口类型。

|  |
| --- |
| **package** com.innerclass.local;  **public** **class** TestLocalInnerClass {  **public** **static** **void** main(String[] args) {  Outer out = **new** Outer();  Object obj = out.outerMethod();  System.***out***.println(obj);  }  }  **class** Outer{  **int** value = 5;  **public** Object outerMethod(){  **final** **int** localValue = 10;  **class** LocalInnerClass{  **public** **void** innerMethod(){  System.***out***.println("局部内部类的方法");  System.***out***.println("局部内部类的方法可以使用外部类的成员："+value);  System.***out***.println("局部内部类的方法可以使用外部类的局部变量："+localValue);  }  }  //先声明后使用  LocalInnerClass li = **new** LocalInnerClass();  li.innerMethod();  **return** li;  }  } |

### 6、匿名内部类

#### （1）如何声明匿名内部类

声明的位置：任何可以创建对象的语句中

格式如下：

new 父类/父接口(){

....

}

意思是创造一个实现（继承）了接口（父类）的类的对象。

#### （2）匿名内部类的特点

* 匿名内部类是类，它编译后的字节码文件名是：外部类名$数字编号.class
* 匿名内部类必须继承父类或实现接口
* 匿名内部类只能有一个对象
* 匿名内部类对象只能使用多态形式引用
* 匿名内部类是特殊的局部内部类，局部内部类的所有限制对它都适用

#### （3）如何使用匿名内部类

三种使用方法：

继承式

接口式

参数式

|  |
| --- |
| **package** com.innerclass.anonymous;  **import** java.util.Arrays;  **import** java.util.Comparator;  **public** **class** TestAnonymous {  **public** **static** **void** main(String[] args) {  //继承式  Car car = **new** Car(250){  **public** **void** run(){  System.***out***.println("改进型：" + getSpeed());  }  };    Car[] cars = **new** Car[3];  cars[0] = car;  cars[1] = **new** Car(120);  cars[2] = **new** Car(140);    **for** (Car c : cars) {  c.run();  }    //接口式  Comparator com = **new** Comparator() {  @Override  **public** **int** compare(Object o1, Object o2) {  Car c1= (Car) o1;  Car c2= (Car) o2;  **return** c1.getSpeed() - c2.getSpeed();  }  };    Arrays.*sort*(cars,com);    //参数式  Arrays.*sort*(cars,**new** Comparator() {  @Override  **public** **int** compare(Object o1, Object o2) {  Car c1= (Car) o1;  Car c2= (Car) o2;  **return** c1.getSpeed() - c2.getSpeed();  }  });      **for** (Car c : cars) {  c.run();  }  }  }  **class** Car{  **private** **int** speed;    **public** Car() {  **super**();  }  **public** Car(**int** speed) {  **super**();  **this**.speed = speed;  }  **public** **int** getSpeed() {  **return** speed;  }  **public** **void** setSpeed(**int** speed) {  **this**.speed = speed;  }  **public** **void** run(){  System.***out***.println("标准发动机助跑" + speed);  }  } |

# 第八章 枚举与注解

## 8.1 枚举

### 1、什么是枚举

某些类的对象只有确定的有限个时，可以把这样的类声明为枚举类。

例如：

星期：Monday(星期一)......Sunday(星期天)

性别：Man(男)、Woman(女)

月份：January(1月)......December(12月)

季节：Spring(春节)......Winter(冬天)

七彩标准色：Red(红)......Purple(紫)

支付方式：Cash（现金）、WeChatPay（微信）、Alipay(支付宝)、BankCard(银行卡)、CreditCard(信用卡)

就职状态：Busy、Free、Vocation、Dimission

订单状态：Nonpayment（未付款）、Paid（已付款）、Fulfilled（已配货）、Delivered（已发货）、Return（退货）、Checked（已确认）

### 2、如何定义枚举类型

要点：

* 私有化类的构造器，保证不能在类的外部创建其对象
* 在类的内部创建枚举类的实例。声明为：public static final

JDK1.5之前：

|  |
| --- |
| **public** **class** TestEnumType {  **public** **static** **void** main(String[] args) {  Season s = Season.***SPRING***;  System.***out***.println(s);  }  }  **class** Season{  **public** **static** **final** Season ***SPRING*** = **new** Season();  **public** **static** **final** Season ***SUMMER*** = **new** Season();  **public** **static** **final** Season ***AUTUMN*** = **new** Season();  **public** **static** **final** Season ***WINTER*** = **new** Season();    **private** Season(){  }    **public** String toString(){  **if**(**this** == ***SPRING***){  **return** "SPRING";  }**else** **if**(**this** == ***SUMMER***){  **return** "SUMMER";  }**else** **if**(**this** == ***AUTUMN***){  **return** "AUTUMN";  }**else**{  **return** "WINTER";  }  }  } |

JDK1.5之后：

* 使用 enum 定义的枚举类，默认继承了 java.lang.Enum类。因此不能再继承其他类。

[修饰符] enum 枚举类{

常量对象列表;

其他成员...

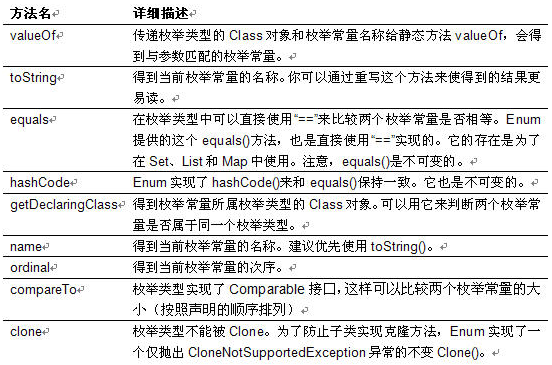
}

* 枚举类的所有构造器只能使用 private 访问控制符
* 枚举类的所有实例必须在枚举类的首行显式列出(, 分隔 ; 结尾)，它们实际上都是public static final修饰的常量对象。
* JDK 1.5 之后可以在 switch 表达式中使用Enum定义的枚举类的对象作为表达式, case 子句可以直接使用枚举值的名字

|  |
| --- |
| **package** com.enumtype.after15;  **public** **class** TestEnumType {  **public** **static** **void** main(String[] args) {  Season s = Season.***SPRING***;  **switch**(s){  **case** ***SPRING***:  System.***out***.println("春暖花开");**break**;  **case** ***SUMMER***:  System.***out***.println("夏日炎炎");**break**;  **case** ***AUTUMN***:  System.***out***.println("秋高气爽");**break**;  **case** ***WINTER***:  System.***out***.println("白雪皑皑");**break**;  }  }  }  **enum** Season{  ***SPRING***,***SUMMER***,***AUTUMN***,***WINTER***;  } |

### 3、java.lang.Enum类的方法

除了toString方法，都是final修饰的方法，因此都不能重写



还有两个API文档中没有，但枚举类类拥有并且常用的方法

* values()方法：返回枚举类型的对象数组。该方法可以很方便地遍历所有的枚举值。
* valueOf(String str)：可以把一个字符串转为对应的枚举类对象。要求字符串必须是枚举类对象的“名字”。如不是，会有运行时异常。

### 4、枚举的特别说明

#### （1）枚举中定义属性

* 枚举类对象的属性不应允许被改动, 所以应该使用 private final修饰。**不是语法要求，而是语义要求**。
* 枚举类的使用 private final 修饰的属性应该在构造器中为其赋值。
* 若枚举类显式的定义了带参数的构造器, 则在列出枚举值时也必须对应的传入参数。

|  |
| --- |
| **package** com.enumtype.before15;  **public** **class** TestWeekField {  **public** **static** **void** main(String[] args) {  Week w = Week.***MONDAY***;  System.***out***.println(w);  }  }  **enum** Week{  ***MONDAY***("星期一"),  ***TUESDAY***("星期二"),  ***WEDNESDAY***("星期三"),  ***THURSDAY***("星期四"),  ***FRIDAY***("星期五"),  ***SATURDAY***("星期六"),  ***SUNDAY***("星期日");  **private** **final** String DESCRPTION;  **private** Week(String dESCRPTION) {  DESCRPTION = dESCRPTION;  }  **public** String toString(){  **return** DESCRPTION;  }  } |

#### （2）枚举类实现接口（了解）

若每个枚举值在调用实现的接口方法呈现相同的行为方式，则只要统一实现该方法即可。

若需要每个枚举值在调用实现的接口方法呈现出不同的行为方式, 则可以让每个枚举值分别来实现该方法

|  |
| --- |
| **interface** Change{  **void** degenerate();  }  **interface** Checkable{  **void** check();  }  **enum** Gender **implements** Change,Checkable{  ***MAN***{  **public** **void** degenerate(){  System.***out***.println("咔嚓一刀");  }  },***WOMAN***{  **public** **void** degenerate(){  System.***out***.println("比较复杂");  }  };  **public** **void** check(){  System.***out***.println("脱光");  }  } |

### 5、关于枚举的拓展

java.util.EnumSet和java.util.EnumMap是两个枚举集合。EnumSet保证集合中的元素不重复；EnumMap中的key是enum类型，而value则可以是任意类型。关于这个两个集合的使用就不在这里赘述，可以参考JDK文档。

## 8.2 注解Annotation

注解Annotation是从JDK5.0开始引入的新特性。

### 1、Annotation的作用：

* 注解不是程序本身，可以对程序作出解释。（这一点，跟注释没什么区别）
* 可以被其他程序（比如：编译器，Checker Framework等）读取。（注解信息处理流程，是注解和注释的重大区别。如果没有注解信息处理流程，则注解毫无意义）

### 2、Annotation的格式：

注解是以“@注释名”在代码中存在的，还可以添加一些参数值，例如：@SuppressWarnings(value=”unchecked”)

### 3、Annotation在哪里使用

Annotation 可以像修饰符一样被使用, 可用于修饰包，类, 构造器, 方法, 成员变量, 参数, 局部变量。相当于给它们添加了额外的辅助信息，而且有些注解我们可以通过反射机制编程实现对这些元数据的访问。

### 4、常见的注解示例

#### （1）生成文档

@author 标明开发该类模块的作者，多个作者之间使用,分割

@version 标明该类模块的版本

@see 参考转向，也就是相关主题

@since 从哪个版本开始增加的

@param 对方法中某参数的说明，如果没有参数就不能写

@return 对方法返回值的说明，如果方法的返回值类型是void就不能写

@throws/@exception 对方法可能抛出的异常进行说明 ，如果方法没有用throws显式抛出的异常就不能写

其中 @param @return 和 @exception 这三个标记都是只用于方法的。

@param的格式要求：@param 形参名 形参类型 形参说明

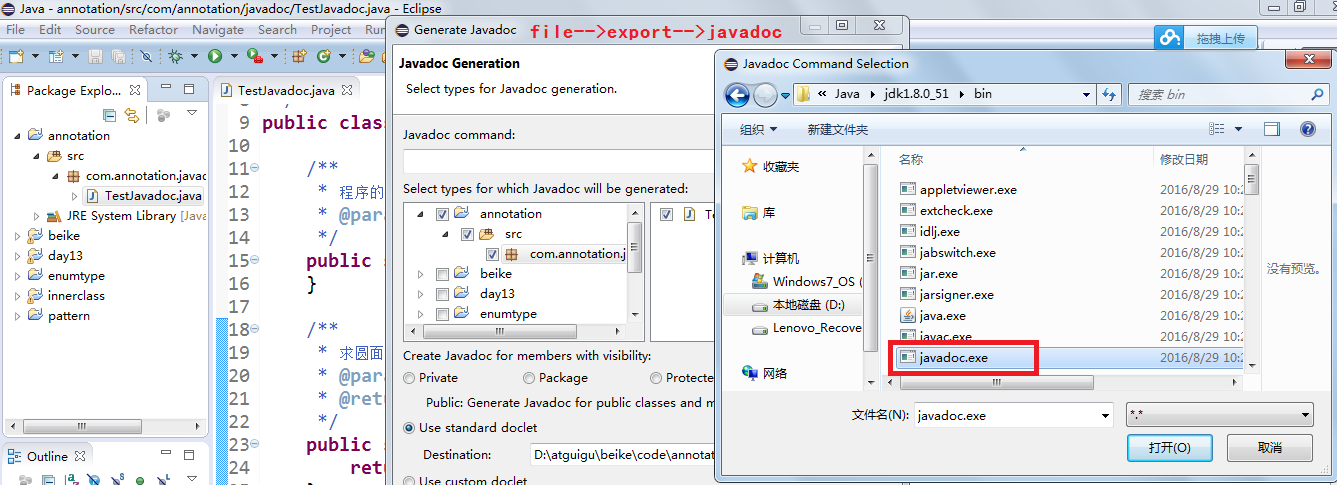
@return 的格式要求：@return 返回值类型 返回值说明

@throws的格式要求：@exception 异常类型 异常说明

@param和@exception可以并列多个

javadoc.exe就是这些注解的信息处理流程

|  |
| --- |
| **package** com.annotation.javadoc;  /\*\*  \*  \* **@author** Irene  \* **@version** 1.0  \* **@see** Math.java  \*  \*/  **public** **class** TestJavadoc {  /\*\*  \* 程序的主方法，程序的入口  \* **@param** args String[] 命令行参数  \*/  **public** **static** **void** main(String[] args) {  }    /\*\*  \* 求圆面积的方法  \* **@param** radius double 半径值  \* **@return** double 圆的面积  \*/  **public** **static** **double** getArea(**double** radius){  **return** Math.***PI*** \* radius \* radius;  }  } |



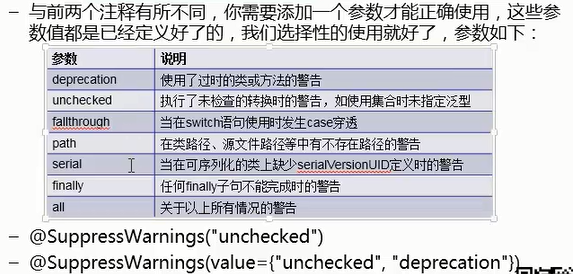
#### （2）在编译时进行格式检查

JDK中系统内置了常用的三个注解：

（1）@Override：按照重写的要求检查方法的格式

（2）@Deprecated：过时，表示不鼓励程序员使用这样的元素，因为存在危险或有更好的实现

（3）@SuppressWarnings：抑制警告



|  |
| --- |
| **package** com.annotation.javadoc;  **public** **class** TestAnnotation {  **public** **static** **void** main(String[] args) {  @SuppressWarnings("unused")  **int** a = 10;  }  @Deprecated  **public** **void** ~~print~~(){  System.***out***.println("过时的方法");  }  @Override  **public** String toString() {  **return** "重写的toString方法()";  }  } |

#### （3）跟踪代码依赖性，实现替代配置文件功能

Servlet3.0提供了注解(annotation),使得不再需要在web.xml文件中进行Servlet的部署

|  |
| --- |
| **package** com.servlet;  **import** java.io.IOException;  **import** javax.servlet.ServletException;  **import** javax.servlet.annotation.WebServlet;  **import** javax.servlet.http.HttpServlet;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  @WebServlet("/login")  **public** **class** LoginServlet **extends** HttpServlet {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **protected** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  }  **protected** **void** doPost(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  doGet(request, response);  }  } |
| <servlet>  <servlet-name>LoginServlet</servlet-name>  <servlet-class>com.servlet.LoginServlet</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>LoginServlet</servlet-name>  <url-pattern>/login</url-pattern>  </servlet-mapping> |

spring框架中关于“事务”的管理

|  |
| --- |
| **@Transactional(propagation=Propagation.REQUIRES\_NEW,**  **isolation=Isolation.READ\_COMMITTED,**  **readOnly=false,**  **timeout=3)**  public void buyBook(String username, String isbn) {  //1.查询书的单价  int price = bookShopDao.findBookPriceByIsbn(isbn);  //2. 更新库存  bookShopDao.updateBookStock(isbn);  //3. 更新用户的余额  bookShopDao.updateUserAccount(username, price);  } |
| <!-- 配置事务属性 -->  <tx:advice transaction-manager="dataSourceTransactionManager" id="txAdvice">  <tx:attributes>  <!-- 配置每个方法使用的事务属性 -->  <tx:method name="buyBook" propagation="REQUIRES\_NEW"  isolation="READ\_COMMITTED" read-only="false"  timeout="3" />  </tx:attributes>  </tx:advice> |

#### （4）JUnit框架中的注解

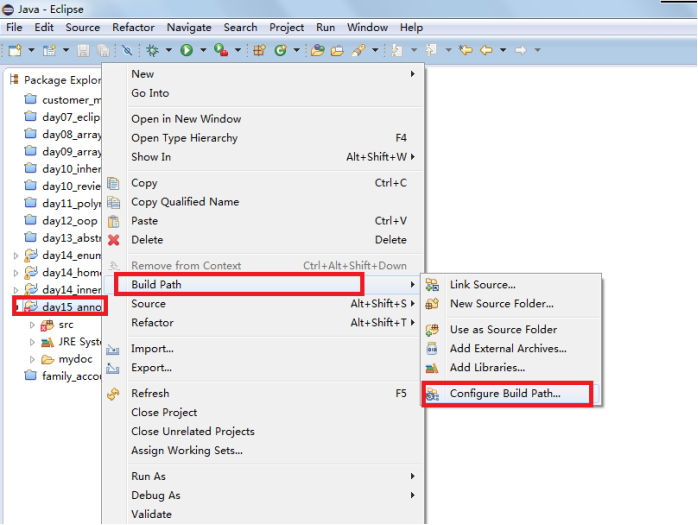
JUnit是由 Erich Gamma 和 Kent Beck 编写的一个回归测试框架（regression testing framework）,供Java开发人员编写单元测试之用。多数Java的开发环境都已经集成了JUnit作为单元测试的工具。

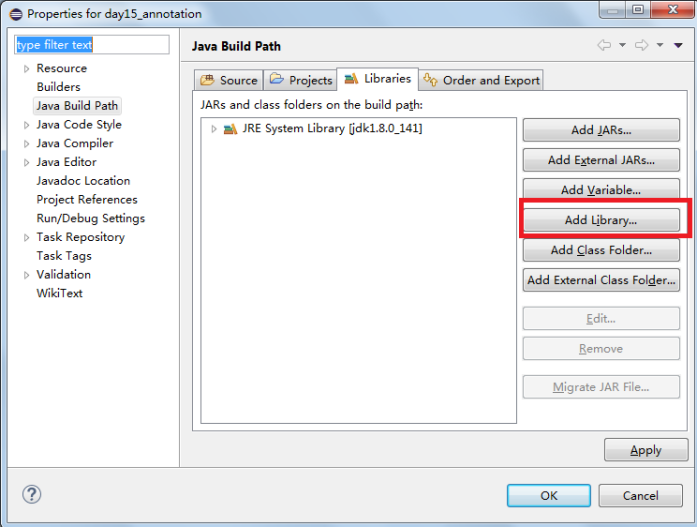
Junit测试是程序员测试，即所谓白盒测试，因为程序员知道被测试的软件如何（How）完成功能和完成什么样（What）的功能。

如何使用JUnit

要使用JUnit，必须在项目的编译路径中必须引入JUnit的库，即相关的.class文件组成的jar包。

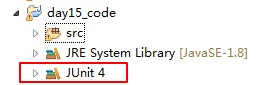
如何把JUnit的jar添加到编译路径如图所示：











JUnit的常用注解

使用JUnit测试的类必须是public的。

JUnit4常见的注解和要求：这些方法都必须是public，无参，无返回值。

@Test：标记在非静态的测试方法上。只有标记@Test的方法才能被作为一个测试方法单独测试。一个类中可以有多个@Test标记的方法。运行时如果只想运行其中一个@Test标记的方法，那么选择这个方法名，然后单独运行，否则整个类的所有标记了@Test的方法都会被执行。

@Test(timeout=1000)：设置超时时间，如果测试时间超过了你定义的timeout，测试失败

@Test(expected)： 申明出会发生的异常，比如 @Test（expected = Exception.class）

了解：

@BeforeClass：标记在静态方法上。因为这个方法只执行一次。在类初始化时执行。

@AfterClass：标记在静态方法上。因为这个方法只执行一次。在所有方法完成后执行。

@Before：标记在非静态方法上。在@Test方法前面执行，而且是在每一个@Test方法前面都执行

@After：标记在非静态方法上。在@Test方法后面执行，而且是在每一个@Test方法后面都执行

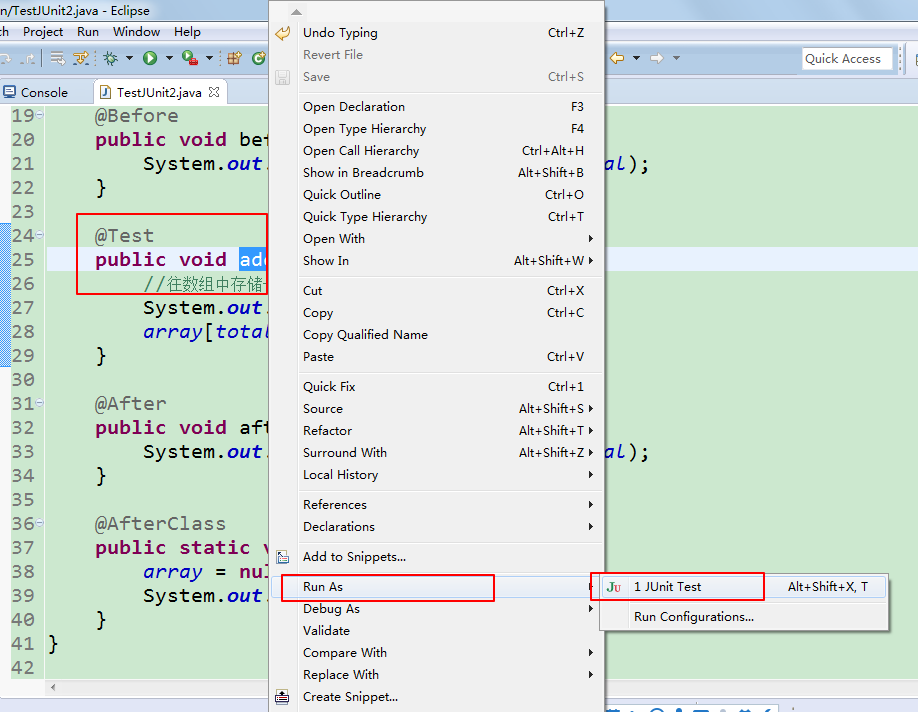
@Ignore：标记在本次不参与测试的方法上。这个注解的含义就是“某些方法尚未完成，暂不参与此次测试”。

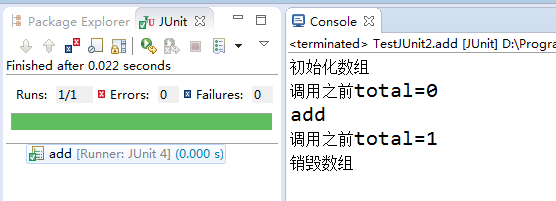
@BeforeClass、@AfterClass、@Before、@After、@Ignore都是配合@Test它使用的，单独使用没有意义。

|  |
| --- |
| **import** org.junit.After;  **import** org.junit.AfterClass;  **import** org.junit.Before;  **import** org.junit.BeforeClass;  **import** org.junit.Test;  **public** **class** TestJUnit2 {  **private** **static** Object[] *array*;  **private** **static** **int** *total*;    @BeforeClass  **public** **static** **void** init(){  System.***out***.println("初始化数组");  *array* = **new** Object[5];  }    @Before  **public** **void** before(){  System.***out***.println("调用之前total=" + *total*);  }    @Test  **public** **void** add(){  //往数组中存储一个元素  System.***out***.println("add");  *array*[*total*++] = "hello";  }    @After  **public** **void** after(){  System.***out***.println("调用之前total=" + *total*);  }    @AfterClass  **public** **static** **void** destroy(){  *array* = **null**;  System.***out***.println("销毁数组");  }  } |

如何运行Junit测试方法

一个类中可以有多个@Test标记的方法，运行时如果只想运行其中一个@Test标记的方法，那么选择这个方法名，然后单独运行，否则整个类的所有标记了@Test的方法都会被执行。





### 5、自定义注解与反射读取注解

* 定义新的 Annotation 类型使用 **@interface** 关键字
* Annotation 的成员变量在 Annotation 定义中以无参数方法的形式来声明。 其方法名和返回值定义了该成员的名字和类型. 我们称为配置参数。类型只能是八种基本数据类型、String类型、Class类型、enum类型、Annotation类型、以上所有类型的数组
* 可以在定义 Annotation 的成员变量时为其指定初始值, 指定成员变量的初始值可使用 default 关键字
* 如果只有一个参数成员，建议使用参数名为value
* 如果定义的注解含有配置参数，那么使用时必须指定参数值，除非它有默认值。格式是“参数名 = 参数值”，如果只有一个参数成员，且名称为value，可以省略“value=”
* 没有成员定义的 Annotation 称为标记; 包含成员变量的 Annotation 称为元数据 Annotation

注意：自定义注解必须配上注解的信息处理流程才有意义。

|  |
| --- |
| **package** com.annotation.javadoc;  **import** java.lang.annotation.Annotation;  **import** java.lang.annotation.ElementType;  **import** java.lang.annotation.Retention;  **import** java.lang.annotation.RetentionPolicy;  **import** java.lang.annotation.Target;  @MyAnnotation(value="尚硅谷")  **public** **class** TestMyAnnotation {  **public** **static** **void** main(String[] args) {  Class clazz = TestMyAnnotation.**class**;  Annotation a = clazz.getAnnotation(MyAnnotation.**class**);  MyAnnotation m = (MyAnnotation) a;  String info = m.value();  System.***out***.println(info);  }  }  @Retention(RetentionPolicy.***RUNTIME***)  @Target(ElementType.***TYPE***)  **@interface** MyAnnotation{  String value() **default** "auguigu";  } |

### 6、元注解

Java5.0定义了4个标准的meta-annotation类型，它们被用来提供对其它 annotation类型作说明。

（1）@Target

（2）@Retention

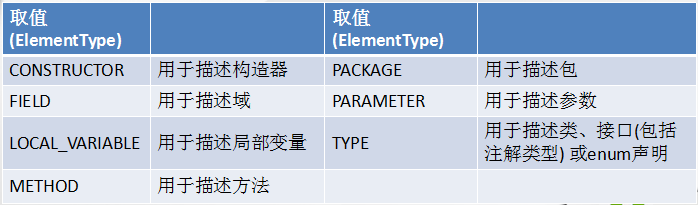
（3）@Documented

（4）@Inherited

在java.lang.annotation包

#### （1）@Target

作用：用于描述注解的使用范围（即：被描述的注解可以用在什么地方



#### （2）@Retention

@Retention定义了该Annotation被保留的时间长短

作用：表示需要在什么级别保存该注释信息，用于描述注解的生命周期（即：被描述的注解在什么范围内有效）

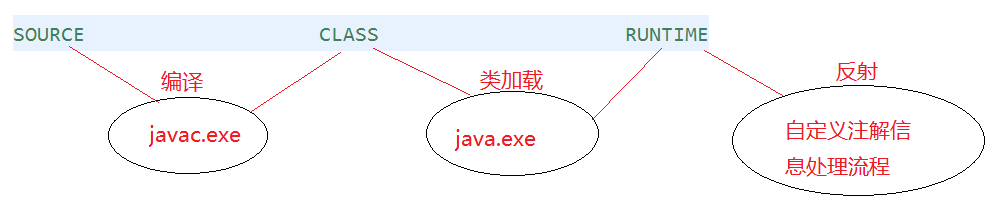
@Rentention 包含一个 RetentionPolicy 类型的成员变量, 使用 @Rentention 时必须为该 value 成员变量指定值:

取值（RetentionPoicy）有：

SOURCE:在源文件中有效（即源文件保留）

CLASS:在class文件中有效（即class保留） 这是默认值

RUNTIME:在运行时有效（即运行时保留）当运行 Java 程序时, JVM 会保留注释. 程序可以通过反射获取该注释



#### （3）@Documented

Documented 注解表明这个注解应该被 javadoc工具记录。默认情况下,javadoc是不包括注解的，但如果声明注解时指定了 @Documented,则它会被 javadoc 之类的工具处理。

#### （4）@Inherited

允许子类继承父类中的注解

### 7、JDK1.8注解的新特性

Java 8对注解处理提供了两点改进：可重复的注解及可用于类型的注解。此外，反射也得到了加强，在Java8中能够得到方法参数的名称。这会简化标注在方法参数上的注解。

#### （1）可重复注解

|  |
| --- |
| **package** com.atguigu.anno;  **import** java.lang.annotation.ElementType;  **import** java.lang.annotation.Target;  **public** **class** TestOldAnnotation {  @RoleAnnotations({@RoleAnnotation(role="admin"),@RoleAnnotation(role="manager"),@RoleAnnotation(role="saler")})  **public** **void** add(){  }  }  @Target(ElementType.***METHOD***)  **@interface** RoleAnnotation{  String role() **default** "admin";  }  //表示target是所有  **@interface** RoleAnnotations{  RoleAnnotation[] value();  } |

JDK1.8之后，可读性更好

|  |
| --- |
| package com.atguigu.anno;  import java.lang.annotation.ElementType;  import java.lang.annotation.Repeatable;  import java.lang.annotation.Target;  public class TestNewAnnotation {    **@LimitAnnotation(role="admin")**  **@LimitAnnotation(role="manager")**  **@LimitAnnotation(role="saler")**  public void test(){    }  }  @Target(ElementType.METHOD) 此处的target必须与LimitAnnotation一致  @interface LimitAnnotations{  LimitAnnotation[] value();  }  **@Repeatable(LimitAnnotations.class)**  @Target(ElementType.METHOD)  @interface LimitAnnotation{  String role() default "admin";  } |

#### （2）类型注解

JDK1.8之后，关于元注解@Target的参数类型ElementType枚举值多了两个：

|  |
| --- |
| **public** **enum** ElementType {  /\*\* Class, interface (including annotation type), or enum declaration \*/  ***TYPE***,  /\*\* Field declaration (includes enum constants) \*/  ***FIELD***,  /\*\* Method declaration \*/  ***METHOD***,  /\*\* Formal parameter declaration \*/  ***PARAMETER***,  /\*\* Constructor declaration \*/  ***CONSTRUCTOR***,  /\*\* Local variable declaration \*/  ***LOCAL\_VARIABLE***,  /\*\* Annotation type declaration \*/  ***ANNOTATION\_TYPE***,  /\*\* Package declaration \*/  ***PACKAGE***,  /\*\*  \* Type parameter declaration  \*  \* **@since** 1.8  \*/  ***TYPE\_PARAMETER***,  /\*\*  \* Use of a type  \*  \* **@since** 1.8  \*/  ***TYPE\_USE***  } |

在java 8之前，注解只能是在声明的地方所使用，java8开始，注解可以应用在任何地方。

ElementType.TYPE\_PARAMETER 表示该注解能写在类型变量的声明语句中（eg：泛型声明）。

ElementType.TYPE\_USE 表示该注解能写在使用类型的任何语句中。

|  |
| --- |
| **package** com.atguigu.anno;  **import** java.lang.annotation.ElementType;  **import** java.lang.annotation.Target;  **public** **class** TestTypeDefine<@TypeDefine() U> {    **private** U u;    **public** <@TypeDefine() T> **void** test(T t){    }  }  @Target({ElementType.***TYPE\_PARAMETER***})  **@interface** TypeDefine{  } |
| package com.atguigu.anno;  import java.lang.annotation.ElementType;  import java.lang.annotation.Target;  @MyAnnotation  public class TestAnnotation<U>{  @MyAnnotation  private String name;  public static void main(String[] args) {  TestAnnotation <@MyAnnotation String> t = null;  int a = (@MyAnnotation int)2L;  @MyAnnotation int b = 10;  }    public static <@MyAnnotation T> void method(T t){    }    public static void test(@MyAnnotation String arg)throws @MyAnnotation Exception{    }  }  @Target(ElementType.TYPE\_USE)  @interface MyAnnotation{  } |

类型注解被用来支持在Java的程序中做强类型检查。配合第三方插件工具Checker Framework（使用Checker Framework可以找到类型注解出现的地方并检查），可以在编译的时候检测出runtime error（eg：UnsupportedOperationException； NumberFormatException；NullPointerException异常等都是runtime error），以提高代码质量。这就是类型注解的作用。

|  |
| --- |
| package checker;  import org.checkerframework.checker.nullness.qual.NonNull;  public class TestChecker {  public static void main(String[] args) {  Object obj = null;  printNonNullToString(obj);  }  public static void printNonNullToString(@NonNull Object object) {  System.out.println(object.toString());  }  } |
|  |
| 进入源代码目录 通过javac编译 -processor 后面接注释处理流程 |
| javac -processor org.checkerframework.checker.nullness.NullnessChecker TestChecker.java |
|  |
| 警告处理 |
| javac -Xbootclasspath/p:D:\software\eclipse\checker-framework-2.1.13\checker\dist\jdk8.jar -processor org.checkerframework.checker.nullness.NullnessChecker TestChecker.java |
| 要配置classpath=.;%JAVA\_HOME%/lib/dt.jar;%JAVA\_HOME%/lib/tools.jar;D:\software\eclipse\checker-framework-2.1.13\checker\dist\checker.jar;D:\software\eclipse\checker-framework-2.1.13\checker\dist\checker-qual.jar; |

注意java 5,6,7版本是不支持注解@NonNull，但checker framework 有个向下兼容的解决方案，就是将类型注解@NonNull 用/\*\*/注释起来。

这样javac编译器就会忽略掉注释块，但用checker framework里面的javac编译器同样能够检测出@NonNull错误。

## 8、练习

1、

声明Week枚举类，其中包含星期一至星期日的定义；

在TestWeek类中声明方法中printWeek(Week week)，根据参数值打印相应的中文星期字符串。

提示，使用switch语句实现。

在main方法中从命令行接收一个1-7的整数(使用Integer.parseInt方法转换)，分别代表星期一至星期日，打印该值对应的枚举值，然后以此枚举值调用printWeek方法，输出中文星期。

2、

（1）.编写一个Person类，使用Override注解它的toString方法

（2）.自定义一个名为“MyTiger”的注解类型，它只可以使用在方法上，带一个String类型的value属性，然后在第（1）题中的Person类上正确使用。

3、判断输出结果为何？

public class Test {

public Test() {

Inner s1 = new Inner();

s1.a = 10;

Inner s2 = new Inner();

s2.a = 20;

Test.Inner s3 = new Test.Inner();

System.out.println(s3.a);

}

class Inner {

public int a = 5;

}

public static void main(String[] args) {

Test t = new Test();

Inner r = t.new Inner();

System.out.println(r.a);

}

}

4、扩展题



答案：

public class TestAnonymous2 {

public static void main(String[] args) {

Test.function().method();

}

}

interface Inter{

void method();

}

class Test{

public static Inter function(){

return new Inter(){

@Override

public void method() {

System.out.println("测试");

}};

}

}

5、我没有指定父类，也没有实现接口，还能不能写匿名内部类。

答案：

new Object(){

void fun(){}

}.fun();

6、对成员内部类的继承说明

public class Test1 extends WithInner.Inner {

Test1(WithInner w){

w.super();

}

}

class WithInner{

class Inner{

}

}

# 第九章 Java异常处理

## 1 异常的概述

### 1.1 什么是异常

异常，又称为例外，是指不可预知的非正常的情况，例如：正常开车突然路口窜出一条狗来，或者正常上下班的路上，某个街角偶遇了爱情，或者在每天晨跑的小树林遭遇了歹徒。这些突发的情况的发生是不可预知的，where,when......，但是如果处理好了，美好的生活不受影响，可以继续，如果处理不好，可能就“挂”了。

Java语言将程序执行中发生的不正常情况称为“异常”，异常是一种对程序运行过程中发生错误时进行通知的机制。如：网络连接失败、用户输入错误或打开文件失败等。

|  |
| --- |
| 提示提示：（1）语法错误不属于异常，这种情况编译不通过，无法运行；（2）逻辑错误不属于异常，这种情况属于BUG，必须修正； |

### 1.2 Java异常的堆栈式抛出机制

示例代码：

|  |
| --- |
| **package** com.atguigu.exception;  **public** **class** TestException {  **public** **static** **void** main(String[] args) {  **int** a = Integer.*parseInt*(args[0]);  **int** b = Integer.*parseInt*(args[1]);  System.***out***.println(a + "/" + b + "=" + a/b);  }  } |

命令行执行：

|  |
| --- |
| java com.atguigu.exception.TestException 12 3 |

运行结果显示12/3=4，结果正常。

#### 1. ArrayIndexOutOfBoundsException数组下标越界异常

命令行执行：

|  |
| --- |
| java com.atguigu.exception.TestException |

出现异常形式，如图10-1：



图10- 1 ArrayIndexOutOfBoundsException数组下标越界异常

#### 2. NumberFormatException数字格式化异常

命令行执行：

|  |
| --- |
| java com.atguigu.exception.TestException a b |

出现异常形式，如图10-2：

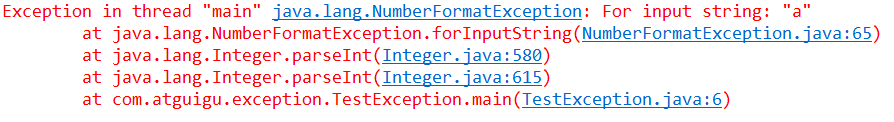


图10- 2 NumberFormatException数字格式化异常

#### 3. ArithmeticException算术异常

命令行执行：

|  |
| --- |
| java com.atguigu.exception.TestException 12 0 |

出现异常形式，如图10-3：



图10- 3 ArithmeticException算术异常

#### 4. 结论

在Java中异常以对象的形式出现，不同的异常对象中封装了相应的错误信息。

Java是采用面向对象的方式来处理异常的。处理过程：

* **抛出异常**：在执行一个方法时，如果在某句代码发生异常，则会在该句代码处生成代表该异常的一个对象，停止当前执行路径，并把异常对象提交给JRE，不同的异常对象中封装了相应的错误信息。
* **捕获异常**：JRE得到该异常后，寻找相应的代码来处理该异常。JRE在方法的调用栈中查找，从生成异常的方法开始回溯，直到找到相应的异常处理代码为止。如果没有找到处理代码，那么程序终止运行

Java中异常的是以堆栈式抛出机制来进行错误通知的，如图10-4。

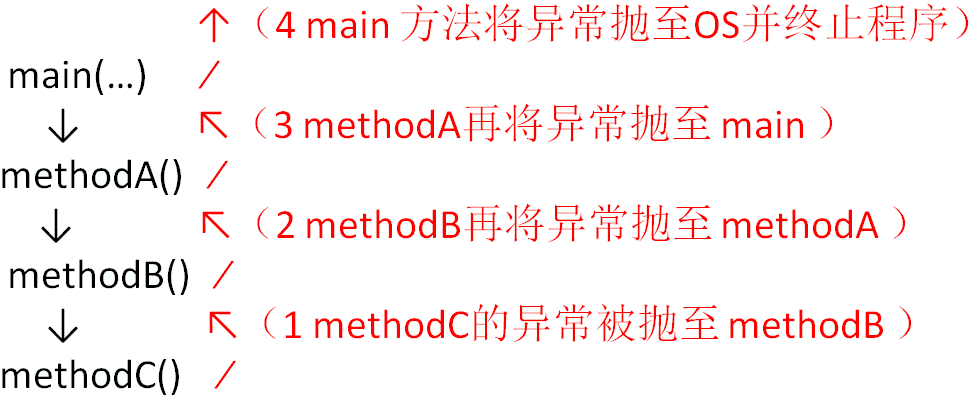


图10- 4 抛出异常

## 2 异常体系结构

如10.1.2所示的ArrayIndexOutOfBoundsException数组下标越界异常，NumberFormatException数字格式化异常，ArithmeticException算术异常，查看它们的API，例如图10-5,10-6,10-7。



图10- 5 ArrayIndexOutOfBoundsException数组下标越界异常



图10- 6 NumberFormatException数字格式化异常



图10- 7 ArithmeticException算术异常

### 2.1 异常的分类

Java语言提供了两大类异常，如图10-8：

* 编译时异常（受检异常Checked）

编译时异常：指程序中必须接受检查和处理的异常，由Exception 类表示。

* 运行时异常（非受检异常UnChecked）

运行时异常：程序中可以不接受检查和处理，或致命性错误异常，由RuntimeException类或Error类表示。

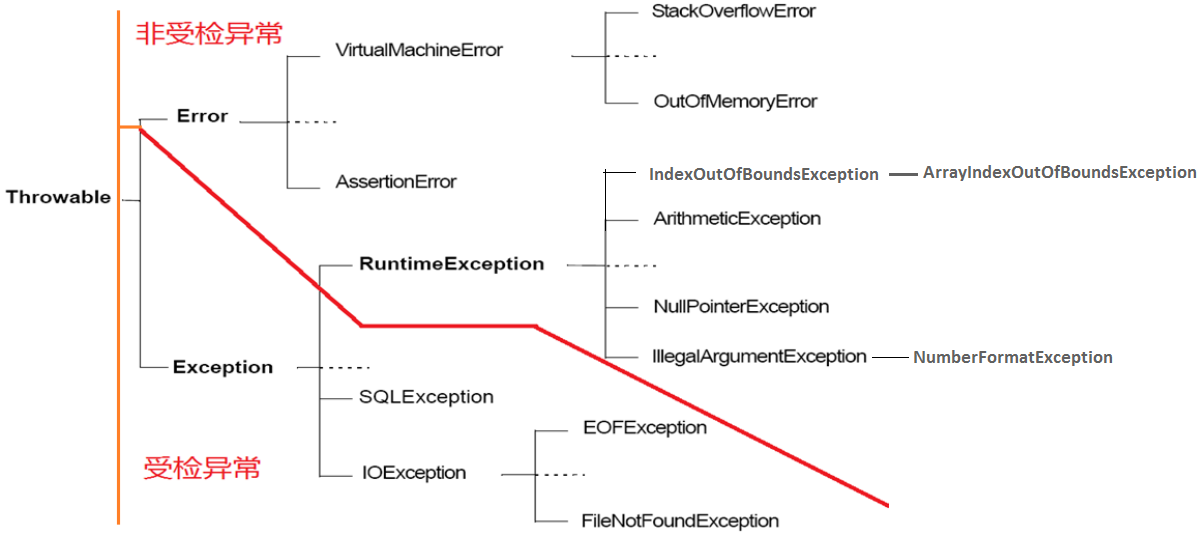


图10- 8 异常分类

### 2.2 Throwable

Throwable 类是 Java 语言中所有错误或异常的超类。只有当对象是此类（或其子类之一）的实例时，才能通过 Java 虚拟机或者 Java throw 语句抛出。类似地，只有此类或其子类之一才可以是 catch 子句中的参数类型。

两个子类的实例，Error 和 Exception，通常用于指示发生了异常情况。通常，这些实例是在异常情况的上下文中新近创建的，因此包含了相关的信息（比如堆栈跟踪数据）。

### 2.3 Error

Error 是 Throwable 的子类，用于指示合理的应用程序不应该试图捕获的严重问题。大多数这样的错误都是异常条件。虽然 ThreadDeath 错误是一个“正规”的条件，但它也是 Error 的子类，因为大多数应用程序都不应该试图捕获它。

在执行该方法期间，无需在其 throws 子句中声明可能抛出但是未能捕获的 Error 的任何子类，因为这些错误可能是再也不会发生的异常条件。

### 2.4 Exception

Exception 类及其子类是 Throwable 的一种形式，它指出了合理的应用程序想要捕获的条件。

应该在程序中尽可能的预知并处理异常。

* checkedException：受检异常。这一类异常，编译时必须对其进行处理，如果不处理，编译不通过。（checked表示编译器检查过已经处理过的）。例如：疲劳必须休息、喝酒必须不能开车、刹车必须好用。
* RuntimeException（uncheckedException）：运行时异常（非受检异常）。一类特殊的异常，如被0除、数组下标越界等，其产生比较频繁，处理麻烦，如果显式的声明或捕获将会对程序可读性和运行效率影响很大。因此由系统自动检测并将它们交给缺省的异常处理程序，即交给JVM处理，用户不显式处理。但是应该尽量避免！！！例如：前车急刹车、有人加塞、路口突然窜出的电动车等等。换句话说，出现RuntimeException一定是你的问题，可以不捕获，因为小心点这些异常是可以避免的，例如：数组下标越界异常ArrayIndexOutOfBoundsException，空指针异常NullPointerException，类型转换异常ClassCastException等等。

## 3 异常处理机制

### 3.1 捕获异常（try...catch...finally）

可使用try…catch…finally语句捕获异常，避免程序不正常终止

语法：

try {

……

} [catch (<异常类型> <异常引用变量>) {

……

} ]\*

[finally {

……

}]

说明：

（1）\*可使用多个catch子句捕获多个异常，[]表示可选。

（2）可以有如下组合形式：try...catch，try...finally，try...catch...finally。

#### 1. try

try块中是可能发生异常的代码。其中某句代码发生异常，就会跳过try块中该句后面的代码，该句代码处会产生并抛出一种或几种类型的异常对象，它后面的catch语句要分别对这些异常做相应的处理。

注意：

一个try必须带一个catch块或finally块或两者都有，即try块不能单独存在。

当异常处理的代码执行结束以后，是不会回到try语句块去执行尚未执行的代码的。只会执行try...catch...finally后面的代码。

#### 2. catch

* 每个try语句块可以伴随一个或多个catch语句，用于处理可能产生的不同类型的异常对象。
* catch块如果多个，异常对象将从上到下依次匹配类型，一旦某个catch的异常类型匹配成功，其他的catch将不再执行，即一个异常只能进入一个catch块。
* catch可以有多个，但是多个catch中的异常类型如果是“互斥关系（没有继承关系）”，顺序随意；如果是“包含关系（继承关系）”，必须小上大下/子上父下。

#### 3. finally

* 有些语句，不管是否发生了异常，都必须要执行，那么就可以把这样的语句放到finally语句块中。
* 通常在finally中关闭程序块已打开的资源，比如：文件流、释放数据库连接等。

#### 4. 运行示意图

try...catch...finally的运行示意图，如图10-9



图10- 9 try...catch...finally运行示意图

#### 5. 示例

（1）示例1

|  |
| --- |
| **package** com.atguigu.exception;  **public** **class** TestException {  **public** **static** **void** main(String[] args) {  **try** {  **int** a = Integer.*parseInt*(args[0]);  **int** b = Integer.*parseInt*(args[1]);  System.***out***.println(a + "/" + b + "=" + a/b);  } **catch** (NumberFormatException e) {  e.printStackTrace();  } **catch** (ArrayIndexOutOfBoundsException e) {  e.printStackTrace();  } **catch** (ArithmeticException e) {  e.printStackTrace();  }  System.***out***.println("程序继续...");  }  } |

（2）示例2

|  |
| --- |
| **package** com.atguigu.exception;  **import** java.io.FileNotFoundException;  **import** java.io.FileReader;  **import** java.io.IOException;  **public** **class** TestFileReader {  **public** **static** **void** main(String[] args) {  FileReader fr = **null**;  **try** {  fr = **new** FileReader("d:/a.txt");  **char** c = (**char**) fr.read();  System.***out***.println(c);  } **catch** (FileNotFoundException e) {  e.printStackTrace();  } **catch** (IOException e) {  //IOException不能放前面，因为FileNotFoundException 是子类  e.printStackTrace();  } **finally**{  //可以嵌套  **try** {  **if**(fr!=**null**){//需要判断非空  fr.close();  }  } **catch** (IOException e) {  e.printStackTrace();  }  }  System.out.println("程序继续...");  }  } |

### 3.2 显式声明抛出异常（throws）

如果一个方法出现Checked Exception，但是并不能确定如何处理这种异常或者不立刻处理它，则此方法应**显示地**声明抛出异常，表明该方法将不对这些异常进行处理，而由该方法的**调用者**负责处理。

在方法声明中用**throws**语句可以声明抛出异常的列表，throws后面的异常类型可以是方法中产生的异常类型，也可以是它的父类。如果一个方法抛出多个受检异常，就必须在方法的签名中列出所有的异常，之间以逗号隔开。

#### 1. 示例代码

|  |
| --- |
| **package** com.atguigu.exception;  **import** java.io.FileNotFoundException;  **import** java.io.FileReader;  **import** java.io.IOException;  **public** **class** TestFileReader2 {  **public** **static** **void** main(String[] args) {  **try** {  *readFile*("d:/a.txt");  } **catch** (IOException e) {  e.printStackTrace();  }  }    **public** **static** **void** readFile(String filename) **throws** FileNotFoundException,IOException{  FileReader fr = **new** FileReader("d:/a.txt");  **char** c = (**char**) fr.read();  System.***out***.println(c);  fr.close();  }  } |

#### 2. 重写方法对throws异常的要求

重写方法不能抛出比被重写方法范围更大的异常类型。在多态的情况下，对重写方法的调用--异常的捕获按父类声明的异常处理。即

* 父类被重写的方法没有声明抛出checked受检异常，那么重写的方法也不能声明抛出异常
* 子类重写方法声明抛出的异常的类型和父类被重写的方法声明抛出异常类型一致
* 子类重写方法声明抛出的异常的类型是父类被重写的方法声明抛出异常的子类
* 子类重写方法可以在方法内部处理异常，而不声明抛出异常

正确的示例代码：

|  |
| --- |
| **class** A{  **public** **void** method()**throws** IOException{}  }  **class** B **extends** A{  **public** **void** method(){}  }  **class** C **extends** A{  **public** **void** method() **throws** FileNotFoundException{}  }  **class** D **extends** A{  **public** **void** method() **throws** IOException,FileNotFoundException{}  }  **class** E **extends** A{  **public** **void** method() **throws** IOException,RuntimeException{}  } |

错误的示例代码：

|  |
| --- |
| **class** A{  **public** **void** method()**throws** IOException{}  }  **class** F **extends** A{  **public** **void** method() **throws** Exception {}  }  **class** G **extends** A{  **public** **void** method() **throws** IOException,SQLException{}  } |

## 4 手动抛出异常（throw）

Java异常类对象除在程序执行过程中出现异常时由系统自动生成并抛出，也可以根据需要手动创建并抛出。可以抛出的异常必须是Throwable或其子类的实例。

首先要创建异常类对象，然后通过throw语句实现抛出操作，提交给Java运行环境。

语法格式：

|  |
| --- |
| throw 异常对象; |

不管系统自动生成的还是手动创建的，异常对象一旦抛出都需要使用捕获try...catch或继续抛出throws的方式处理。

其次，throw语句可以代替return语句结束方法。

|  |
| --- |
| **package** com.atguigu.exception;  **public** **class** TestThrow {  **public** **static** **void** check(String username,String password){  **if**(username==**null** || "".equals(username.trim())){  **throw** **new** RuntimeException("用户名不能为空");  }  **if**(password==**null** || "".equals(password.trim())){  **throw** **new** RuntimeException("密码不能为空");  }  }  } |

## 5 自定义异常

在程序中，可能会遇到任何标准异常类都没能充分的描述清楚的问题，这种情况下可以创建自己的异常类。

* 从Exception类或者它的子类派生一个子类即可
* 习惯上，自定义异常类应该包含2个构造器：一个是无参构造，另一个是带有详细信息的构造器
* 自定义的异常只能通过throw抛出。
* 自定义异常最重要的是异常类的名字，当异常出现时，可以根据名字判断异常类型。

|  |
| --- |
| **package** com.atguigu.exception;  **public** **class** TestMyException {  **public** **static** **void** login(String username,String password) **throws** WrongUsernameOrPassWordException{  **if**(!("admin".equals(username) && "123".equals(password))){  **throw** **new** WrongUsernameOrPassWordException("用户名或密码错误");  }  System.***out***.println("登录成功");  }  }  **class** WrongUsernameOrPassWordException **extends** Exception{  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **public** WrongUsernameOrPassWordException() {  **super**();  }  **public** WrongUsernameOrPassWordException(String message) {  **super**(message);  }  } |

## 6 finally与return

执行顺序：

1、执行try,catch，给“返回值”临时变量赋值

2、执行finally

3、return

return语句两个作用：给返回值赋值，结束方法运行

### 6.1 从finally的return回来

示例代码：

|  |
| --- |
| **package** com.atguigu.exception;  **public** **class** TestReturn {  **public** **static** **void** main(String[] args) {  **int** result = *test*("a");  System.***out***.println(result);  }  **public** **static** **int** test(String str){  **try**{  Integer.*parseInt*(str);  **return** 1;  }**catch**(NumberFormatException e){  **return** -1;  }**finally**{  System.out.println("test结束");  **return** 0;  }  }  } |

运行结果：

|  |
| --- |
| test结束  0 |

### 6.2 从catch的return回来

示例代码

|  |
| --- |
| **package** com.atguigu.exception;  **public** **class** TestReturn {  **public** **static** **void** main(String[] args) {  **int** result = *test*("a");  System.***out***.println(result);  }  **public** **static** **int** test(String str){  **try**{  Integer.*parseInt*(str);  **return** 1;  }**catch**(NumberFormatException e){  **return** -1;  }**finally**{  System.***out***.println("test结束");  }  }  } |

运行结果：

|  |
| --- |
| test结束  -1 |

### 6.3 从try的return回来

示例代码：

|  |
| --- |
| **package** com.atguigu.exception;  **public** **class** TestReturn {  **public** **static** **void** main(String[] args) {  **int** result = *test*("12");  System.***out***.println(result);  }  **public** **static** **int** test(String str){  **try**{  Integer.*parseInt*(str);  **return** 1;  }**catch**(NumberFormatException e){  **return** -1;  }**finally**{  System.***out***.println("test结束");  }  }  } |

运行结果：

|  |
| --- |
| test结束  1 |

## 7 小结

* 要避免使用异常处理代替错误处理，这样会降低程序的清晰性，并且效率低下。
* 不要进行小粒度的异常处理——应该将整个任务包装在一个try语句块中。
* 异常往往在高层处理。例如：（高层）领导--->员工。调用关系。有问题赶紧上报

Java的异常处理机制总体示意图，如图10-10：

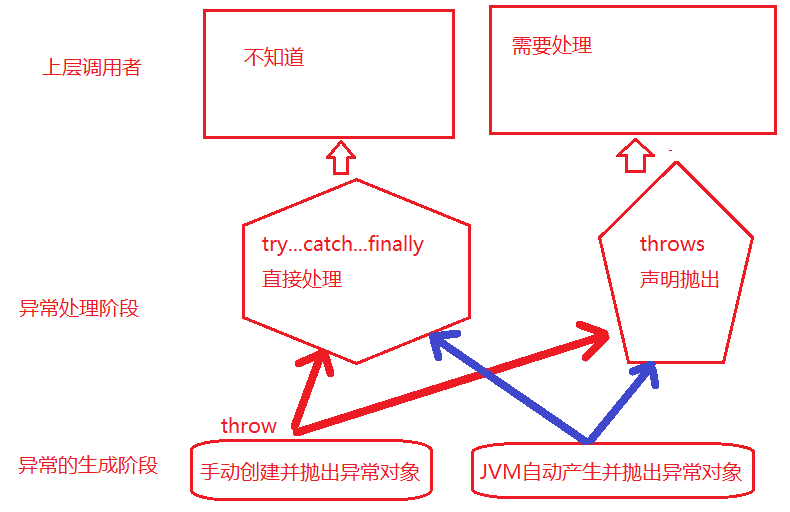


图10- 10 Java异常处理机制示意图

## 8 练习

1、编写TestException类，在main方法中接收两个命令行参数，将它们转换为整数，并用第二个数除以第一个数，打印结果。

在命令行运行程序，给出两个参数，测试以下情况，观察运行结果：

其中某个参数不是数字

第二个参数为0

提示：

分别接收两个参数的语句为：

int n1 = Integer.parseInt(args[0]);

int n2 = Integer.parseInt(args[1]);

改写TestException类的main方法，在其中捕获可能发生的异常。

运行并测试各种异常情况，确认是否均被捕获。

2、判断程序的输出结果

public class ReturnExceptionDemo {

static void methodA() {

try {

System.out.println("进入方法A");

throw new RuntimeException("制造异常");

} finally {

System.out.println("用A方法的finally");

}

}

static void methodB() {

try {

System.out.println("进入方法B");

return;

} finally {

System.out.println("调用B方法的finally");

}

}

public static void main(String[] args) {

try {

methodA();

} catch (Exception e) {

System.out.println(e.getMessage());

}

methodB();

}

}

3、编写应用程序EcmDef.java，接收命令行的两个参数，要求不能输入负数，计算两数相除。

对数据类型不一致(NumberFormatException)、缺少命令行参数(ArrayIndexOutOfBoundsException、

除0(ArithmeticException)及输入负数(EcDef 自定义的异常)进行异常处理。

提示：

(1)在主类(EcmDef)中定义异常方法(ecm)完成两数相除功能。

(2)在main()方法中使用异常处理语句进行异常处理。

(3)在程序中，自定义对应输入负数的异常类(EcDef)。

(4)运行时接受参数 java EcmDef 20 10

//args[0]=“20” args[1]=“10”

(5)Interger类的static方法parseInt(String s)将s转换成对应的int值。如int a=Interger.parseInt(“314”); //a=314;

# 项目3：开发团队调度系统

模拟实现一个基于文本界面的《团队人员调度软件》

熟悉Java面向对象的高级特性，进一步掌握编程技巧和调试技巧

主要涉及以下知识点：

* 类的继承和多态
* 对象的关联
* static和final修饰符
* 特殊类的使用
* 异常处理

## 一、需求说明

模拟实现基于文本界面的《团队人员调度软件》。

该软件实现以下功能：

1. 软件启动时，根据给定的数据创建公司部分成员列表（数组）
2. 根据菜单提示，基于现有的公司成员，组建一个开发团队以开发一个新的项目
3. 组建过程包括将成员插入到团队中，或从团队中删除某成员，还可以列出团队中现在成员的列表
4. 开发团队成员包括架构师、设计师和程序员

本软件采用单级菜单方式工作。

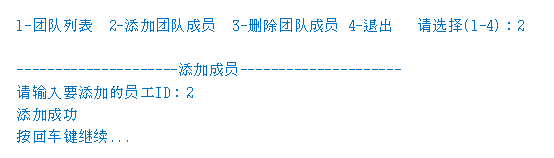
### 1、主界面

当软件运行时，主界面显示公司成员（部分）的列表，如下：



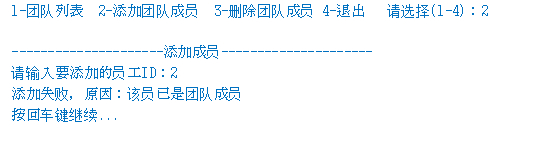
### 2、添加团队成员

当选择“添加团队成员”菜单时，将执行从列表中添加指定（通过ID）成员到开发团队的功能：



添加成功后，按回车键将重新显示主界面。

如果添加操作因某种原因失败，将显示类似以下信息（失败原因视具体原因而不同）：

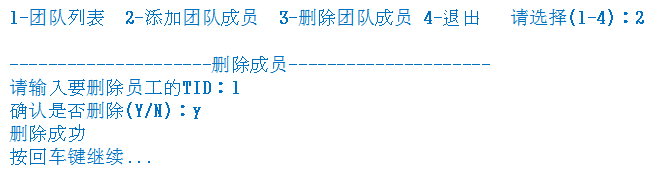


失败信息包含以下几种：

1. 成员已满，无法添加
2. 该成员不是开发人员，无法添加
3. 该员已是团队成员
4. 该员正在休假，无法添加
5. 该员工已是团队成员
6. 团队中只能有一名架构师
7. 团队中只能有两名设计师
8. 团队中只能有三名程序员

### 3、删除团队成员

当选择“删除团队成员”菜单时，将执行从开发团队中删除指定（**通过TeamID**）成员的功能：



删除成功后，按回车键将重新显示主界面。

### 4、团队列表

当选择“团队列表”菜单时，将列出开发团队中的现有成员，例如：



注意有两个编号，一个是在公司的员工编号ID，一个是在开发团队的编号TID。

## 二、键盘输入工具类TSUtility.java

项目中提供了TSUtility.java类，可用来方便地实现键盘访问。

该类提供了以下静态方法：

public static char readMenuSelection()

用途：该方法读取键盘，如果用户键入’1’-’4’中的任意字符，则方法返回。返回值为用户键入字符。

public static void readReturn()

用途：该方法提示并等待，直到用户按回车键后返回。

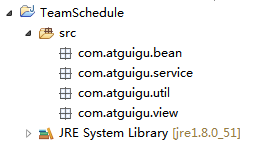
public static int readInt()

用途：该方法从键盘读取一个长度不超过2位的整数，并将其作为方法的返回值。

public static char readConfirmSelection() ：

用途：从键盘读取‘Y’或’N’，并将其作为方法的返回值。

## 三、项目结构

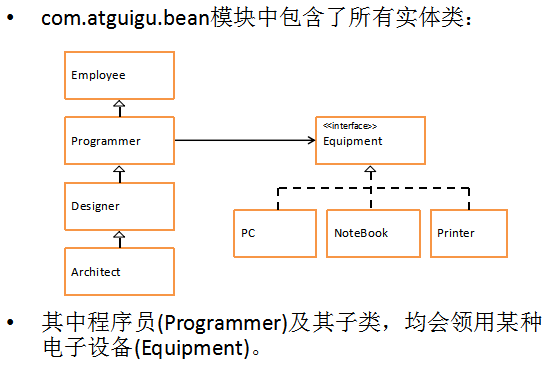


先建立4个包。

1. com.atguigu.bean：用来存放基本的JavaBean。例如员工类等基础数据封装JavaBean
2. com.atguigu.service：service模块为实体对象（Employee及其子类如程序员等）的管理模块， NameListService和TeamService类分别用各自的数组来管理公司员工和开发团队成员对象。
3. com.atguigu.util：用来存放工具类。例如键盘输入工具类TSUtility.java
4. com.atguigu.view：用来存放视图类，负责菜单的显示和处理用户操作。例如TeamView.java

## 四、开发详解

### 1、编写实体类

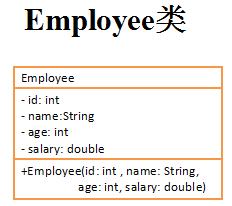


从中可以看出架构师继承设计师，设计师继承程序员，程序员继承员工类。

程序员及其子类关联一个设备对象。即在程序员中有一个设备的引用，或者说有一个Equipment类型的属性。

而设备又分为三种，都实现了设备接口分别为PC电脑类，NoteBook笔记本类，Printer打印机类。

#### （1）Employee类



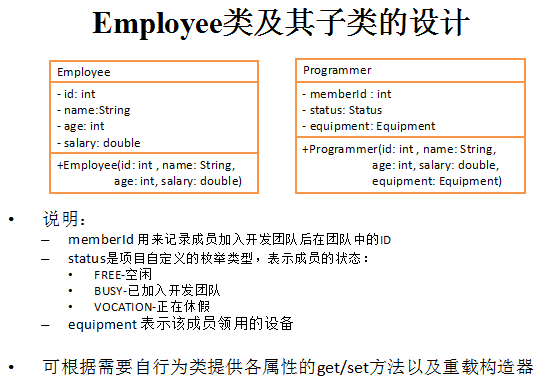
|  |
| --- |
| **package** com.atguigu.bean;  **public** **class** Employee {  **private** **int** id;  **private** String name;  **private** **int** age;  **private** **double** salary;  **public** Employee() {  }  **public** Employee(**int** id, String name, **int** age, **double** salary) {  **this**.id = id;  **this**.name = name;  **this**.age = age;  **this**.salary = salary;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  **public** **int** getId() {  **return** id;  }  **public** **void** setId(**int** id) {  **this**.id = id;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **double** getSalary() {  **return** salary;  }  **public** **void** setSalary(**double** salary) {  **this**.salary = salary;  }  **public** String getDetails() {  **return** id + "\t" + name + "\t" + age+ "\t" +salary;  }  @Override  **public** String toString() {  **return** getDetails();  }  } |



看显示格式，提供getDetails()返回员工的详细信息，并且格式使用”\t”控制。

|  |
| --- |
| **public** String getDetails() {  **return** id + "\t" + name + "\t" + age+ "\t" +salary;  }  @Override  **public** String toString() {  **return** getDetails();  } |

#### （2）Employee的子类Programmer





从上面分析，可以看出，

* 程序员有一个Status的属性，表名该员工的状态。而状态只有三种，因此可以使用枚举类型。
* 程序员有一个Equipment的属性，表名该员工领用的设备。

因此需要先写Status枚举类和Equipment接口类

#### （3）Status枚举类

|  |
| --- |
| **package** com.atguigu.bean;  **public** **enum** Status {  ***FREE***, ***BUSY***, ***VOCATION***  } |

#### （4）Equipment接口



|  |
| --- |
| **package** com.atguigu.bean;  **public** **interface** Equipment {  **public** String getDescription();  } |

#### （5）Programmer类

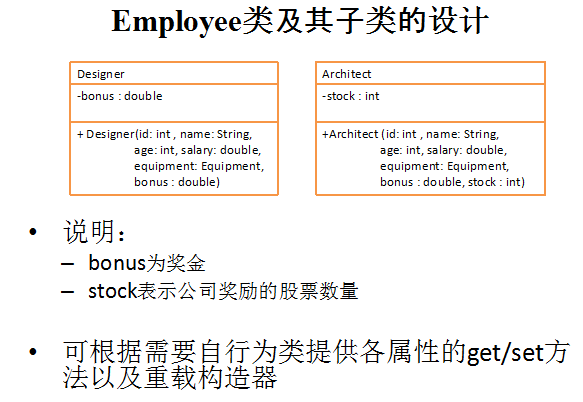
|  |
| --- |
| **package** com.atguigu.bean;  **public** **class** Programmer **extends** Employee {  **private** **int** memberId;  **private** Status status = Status.FREE;  **private** Equipment equipment;  **public** Programmer() {  }  **public** Programmer(**int** id, String name, **int** age,  **double** salary, Equipment equipment) {  **super**(id, name, age, salary);  **this**.equipment = equipment;  }  **public** Status getStatus() {  **return** status;  }  **public** **void** setStatus(Status status) {  **this**.status = status;  }  **public** Equipment getEquipment() {  **return** equipment;  }  **public** **void** setEquipment(Equipment equipment) {  **this**.equipment = equipment;  }  **public** **int** getMemberId() {  **return** memberId;  }  **public** **void** setMemberId(**int** memberId) {  **this**.memberId = memberId;  }  @Override  **public** String toString() {  **return** getDetails() + "\t程序员\t" + status + "\t\t\t" + equipment ;  }  } |



从上面分析：

* Status属性默认都是FREE。
* 打印程序员Programmer对象的格式。Employee通用的详细信息后面再拼接上 "\t程序员\t" + status + "\t\t\t" + equipment

#### （6）Employee类及其子类Designer和Architect



|  |
| --- |
| **package** com.atguigu.bean;  **public** **class** Designer **extends** Programmer{  **private** **double** bonus;  **public** Designer() {  }  **public** Designer(**int** id, String name, **int** age, **double** salary,  Equipment equipment, **double** bonus) {  **super**(id, name, age, salary, equipment);  **this**.bonus = bonus;  }  **public** **double** getBonus() {  **return** bonus;  }  **public** **void** setBonus(**double** bonus) {  **this**.bonus = bonus;  }  @Override  **public** String toString() {  **return** getDetails() + "\t设计师\t" + getStatus() + "\t" +  getBonus() +"\t\t" + getEquipment();  }  } |
| **package** com.atguigu.bean;  **public** **class** Architect **extends** Designer {  **private** **int** stock;  **public** Architect() {  }  **public** Architect(**int** id, String name, **int** age, **double** salary,  Equipment equipment, **double** bonus, **int** stock) {  **super**(id, name, age, salary, equipment, bonus);  **this**.stock = stock;  }  **public** **int** getStock() {  **return** stock;  }  **public** **void** setStock(**int** stock) {  **this**.stock = stock;  }  @Override  **public** String toString() {  **return** getDetails() + "\t架构师\t" + getStatus() + "\t" +  getBonus() + "\t" + getStock() + "\t" + getEquipment();  }  } |



格式控制：

|  |
| --- |
| @Override  **public** String toString() {  **return** getDetails() + "\t设计师\t" + getStatus() + "\t" +  getBonus() +"\t\t" + getEquipment();  } |
| @Override  **public** String toString() {  **return** getDetails() + "\t架构师\t" + getStatus() + "\t" +  getBonus() + "\t" + getStock() + "\t" + getEquipment();  } |

#### （7）NoteBook、PC、Printer类





观察显示格式：

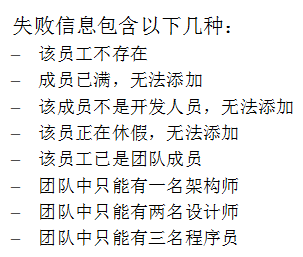
* NoteBook显示的是型号（价格）
* PC显示的是型号（显示器）
* Printer显示的是名称（类型）

|  |
| --- |
| **package** com.atguigu.bean;  **public** **class** NoteBook **implements** Equipment{  **private** String model;//型号  **private** **double** price;//价格    **public** NoteBook() {  **super**();  }  **public** NoteBook(String model, **double** price) {  **super**();  **this**.model = model;  **this**.price = price;  }  **public** String getModel() {  **return** model;  }  **public** **void** setModel(String model) {  **this**.model = model;  }  **public** **double** getPrice() {  **return** price;  }  **public** **void** setPrice(**double** price) {  **this**.price = price;  }  @Override  **public** String getDescription() {  **return** model+"("+price+")";  }  @Override  **public** String toString() {  **return** getDescription();  }    } |
| **package** com.atguigu.bean;  **public** **class** PC **implements** Equipment{  **private** String model;//型号  **private** String display;//显示器    **public** PC() {  **super**();  }  **public** PC(String model, String display) {  **super**();  **this**.model = model;  **this**.display = display;  }  **public** String getModel() {  **return** model;  }  **public** **void** setModel(String model) {  **this**.model = model;  }  **public** String getDisplay() {  **return** display;  }  **public** **void** setDisplay(String display) {  **this**.display = display;  }  @Override  **public** String getDescription() {  **return** model +"("+ display+")";  }  @Override  **public** String toString() {  **return** getDescription();  }  } |
| **package** com.atguigu.bean;  **public** **class** Printer **implements** Equipment {  **private** String type;//类型  **private** String name;//名称  **public** Printer() {  **super**();  }  **public** Printer(String type, String name) {  **super**();  **this**.type = type;  **this**.name = name;  }  **public** String getType() {  **return** type;  }  **public** **void** setType(String type) {  **this**.type = type;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  @Override  **public** String getDescription() {  **return** name + "(" + type + ")";  }  @Override  **public** String toString() {  **return** getDescription();  }    } |

### 2、编写异常类TeamException

用于封装整个项目中的异常，例如

添加员工到开发团队时会遇到如下失败情况



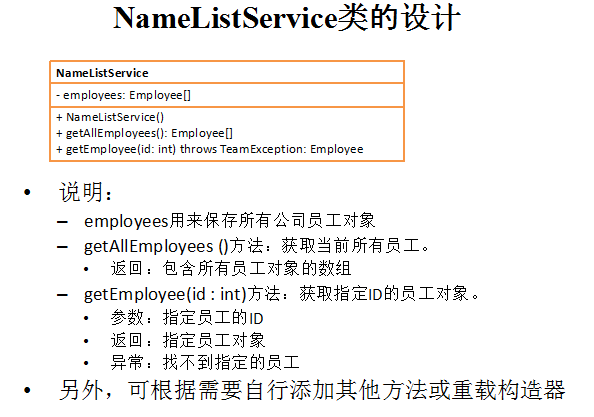
例如从团队中删除员工会遇到如下失败情况

* 找不到该成员，无法删除

|  |
| --- |
| **package** com.atguigu.service;  **public** **class** TeamException **extends** Exception {  **public** TeamException() {  **super**();  }  **public** TeamException(String message) {  **super**(message);  }  } |

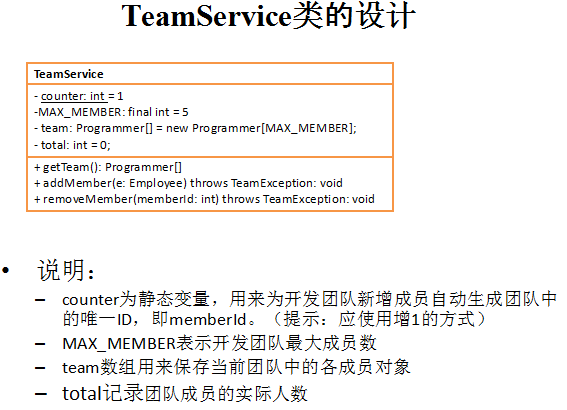
### 3、编写业务类NameListService

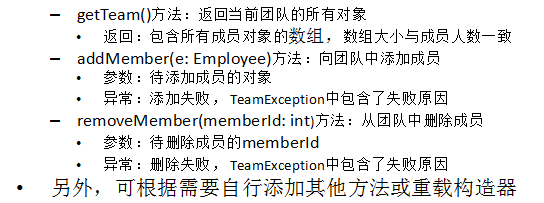
service模块为实体对象（Employee及其子类如程序员等）的管理模块， NameListService和TeamService类分别用各自的数组来管理公司员工和开发团队成员对象



|  |
| --- |
| **package** com.atguigu.service;  **import** com.atguigu.bean.Employee;  **public** **class** NameListService {  **private** Employee[] employees;  **public** Employee[] getAllEmployees() {  **return** employees;  }  **public** Employee getEmployee(**int** id) **throws** TeamException {  **for** (Employee e : employees) {  **if** (e.getId() == id)  **return** e;  }  **throw** **new** TeamException("该员工不存在");  }  } |

### 4、编写业务类TeamService





|  |
| --- |
| **package** com.atguigu.service;  **import** com.atguigu.bean.Architect;  **import** com.atguigu.bean.Designer;  **import** com.atguigu.bean.Employee;  **import** com.atguigu.bean.Programmer;  **import** com.atguigu.bean.Status;  **public** **class** TeamService {  **private** **static** **int** *counter* = 1;//用于控制团队中团队成员的ID，每增加一个就自动增1，删除成员是不减1，一直往上增，只是个编号  //就如同银行的叫号系统，过号了，也是继续往下增  //如同学号，有同学退学，休学啥的，原来那个学号也不用了，一直往下递增  **private** **final** **int** MAX\_MEMBER = 5; //团队最多容纳的人数  **private** Programmer[] team = **new** Programmer[MAX\_MEMBER];  **private** **int** total = 0;//记录团队的实际人数    **private** **int** numOfArch = 0, numOfDsgn = 0, numOfPrg = 0;//分别用于记录架构师，设计师，程序员的数量  **public** TeamService() {  }    //供团队列表时调用，返回实际团队的成员信息，不能直接返回team，否则其中可能有空元素  **public** Programmer[] getTeam() {  Programmer[] team = **new** Programmer[total];  **for** (**int** i = 0; i < total; i++) {  team[i] = **this**.team[i];  }  **return** team;  }  //添加人员到团队中  **public** **void** addMember(Employee e) **throws** TeamException {  **if** (total >= MAX\_MEMBER)  **throw** **new** TeamException("成员已满，无法添加");  **if** (!(e **instanceof** Programmer))  **throw** **new** TeamException("该成员不是开发人员，无法添加");  Programmer p = (Programmer)e;  **switch** (p.getStatus()) {  **case** ***BUSY*** :**throw** **new** TeamException("该员已是团队成员");  **case** ***VOCATION***:**throw** **new** TeamException("该员正在休假，无法添加");  }  **if** (isExist(p))  **throw** **new** TeamException("该员工已是团队成员");  //注意if..else if条件的顺序，Programmer包含Designer，Designer包含Architect  **if** (p **instanceof** Architect) {  **if** (numOfArch >= 1) **throw** **new** TeamException("团队中只能有一名架构师");  } **else** **if** (p **instanceof** Designer) {  **if** (numOfDsgn >= 2) **throw** **new** TeamException("团队中只能有两名设计师");  } **else** **if** (p **instanceof** Programmer) {  **if** (numOfPrg >= 3) **throw** **new** TeamException("团队中只能有三名程序员");  }  p.setStatus(Status.***BUSY***);//修改员工状态为忙，这样别的团队就不会再选择他了  p.setMemberId(*counter*++);//设置在团队中的memberId  team[total] = p;//把人员添加到开发团队的数组中  total++;//总人数+1    //相应工种的人的数量+1  **if** (p **instanceof** Architect) {  numOfArch++;  } **else** **if** (p **instanceof** Designer) {  numOfDsgn++;  } **else** **if** (p **instanceof** Programmer) {  numOfPrg++;  }  }  //判断新添加的员工是否已经是团队成员  **private** **boolean** isExist(Programmer p) {  **for** (**int** i = 0; i < total; i++) {  **if** (team[i].getId() == p.getId()) **return** **true**;  }  **return** **false**;  }  //删除团队成员  **public** **void** removeMember(**int** memberId) **throws** TeamException {  **int** n = 0;//记录要删除的员工的下标  **for** (; n < total; n++) {  **if** (team[n].getMemberId() == memberId) {  team[n].setStatus(Status.***FREE***);  **break**;  }  }  **if** (n == total)//上一个for中没有break出来，那么退出循环是因为n<total不成立，那么说没有找到该成员  **throw** **new** TeamException("找不到该成员，无法删除");  //删除时，把后面的元素往前移动  **for** (**int** i = n + 1; i < total; i++) {  team[i - 1] = team[i];//后一个元素复制给前一个元素  }  team[--total] = **null**;//总人数减1，最后一个元素因为已经复制到前一个元素中了，因此置为null  }  } |

### 5、把键盘输入工具类TSUtility添加到项目中

直接将提供的TSUtility.java文件拷贝到com.atguigu.util包中，发现需要修改包名。

第二发现出现乱码，因为提供的文件是UTF-8格式的，如果你默认的项目字符编码为GBK的话，就会乱码，解决办法可以用记事本打开TSUtility.java文件，把代码复制到eclipse中。或者修改项目的字符编码为UTF-8，不过设置项目字符编码最好在项目一开始。

|  |
| --- |
| **package** com.atguigu.util;  **import** java.util.\*;  **public** **class** TSUtility {  **private** **static** Scanner *scanner* = **new** Scanner(System.***in***);  **public** **static** **char** readMenuSelection() {  **char** c;  **for** (; ; ) {  String str = *readKeyBoard*(1, **false**);  c = str.charAt(0);  **if** (c != '1' && c != '2' &&  c != '3' && c != '4') {  System.***out***.print("选择错误，请重新输入：");  } **else** **break**;  }  **return** c;  }  **public** **static** **void** readReturn() {  System.***out***.print("按回车键继续...");  *readKeyBoard*(100, **true**);  }  **public** **static** **int** readInt() {  **int** n;  **for** (; ; ) {  String str = *readKeyBoard*(2, **false**);  **try** {  n = Integer.*parseInt*(str);  **break**;  } **catch** (NumberFormatException e) {  System.***out***.print("数字输入错误，请重新输入：");  }  }  **return** n;  }  **public** **static** **char** readConfirmSelection() {  **char** c;  **for** (; ; ) {  String str = *readKeyBoard*(1, **false**).toUpperCase();  c = str.charAt(0);  **if** (c == 'Y' || c == 'N') {  **break**;  } **else** {  System.***out***.print("选择错误，请重新输入：");  }  }  **return** c;  }  **private** **static** String readKeyBoard(**int** limit, **boolean** blankReturn) {  String line = "";  **while** (*scanner*.hasNextLine()) {  line = *scanner*.nextLine();  **if** (line.length() == 0) {  **if** (blankReturn) **return** line;  **else** **continue**;  }  **if** (line.length() < 1 || line.length() > limit) {  System.***out***.print("输入长度（不大于" + limit + "）错误，请重新输入：");  **continue**;  }  **break**;  }  **return** line;  }  } |

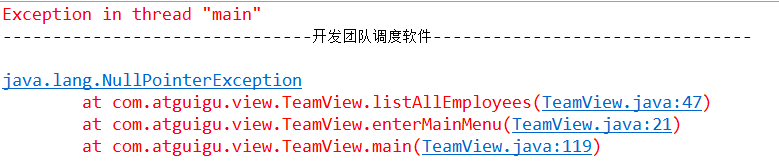
### 6、编写主界面TeamView



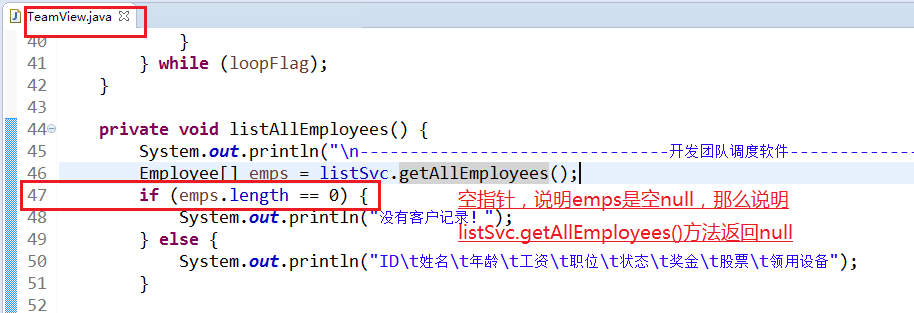
|  |
| --- |
| **package** com.atguigu.view;  **import** com.atguigu.bean.Employee;  **import** com.atguigu.service.NameListService;  **import** com.atguigu.util.TSUtility;  **public** **class** TeamView {  **private** NameListService listSvc = **new** NameListService();  **public** **void** enterMainMenu() {  **boolean** loopFlag = **true**;  **char** key = 0;  **do** {  **if** (key != '1') listAllEmployees();  System.***out***.print("1-团队列表 2-添加团队成员 3-删除团队成员 4-退出 请选择(1-4)：");  key = TSUtility.*readMenuSelection*();  System.***out***.println();  **switch** (key) {  **case** '1':  listTeam();  **break**;  **case** '2':  addMember();  **break**;  **case** '3':  deleteMember();  **break**;  **case** '4':  System.***out***.print("确认是否退出(Y/N)：");  **char** yn = TSUtility.*readConfirmSelection*();  **if** (yn == 'Y') loopFlag = **false**;  **break**;  }  } **while** (loopFlag);  }  **private** **void** listAllEmployees() {  System.***out***.println("\n-------------------------------开发团队调度软件--------------------------------\n");  System.***out***.println("-------------------------------------------------------------------------------");  }  **private** **void** listTeam() {  System.***out***.println("\n--------------------团队成员列表---------------------\n");  System.***out***.println("-----------------------------------------------------");  }  **private** **void** addMember() {  System.***out***.println("---------------------添加成员---------------------");  }  **private** **void** deleteMember() {  System.***out***.println("---------------------删除成员---------------------");  }  **public** **static** **void** main(String[] args) {  TeamView view = **new** TeamView();  view.enterMainMenu();  }  } |

### 7、列出公司所有员工列表

|  |
| --- |
| **private** **void** listAllEmployees() {  System.***out***.println("\n-------------------------------开发团队调度软件--------------------------------\n");  Employee[] emps = listSvc.getAllEmployees();  **if** (emps.length == 0) {  System.***out***.println("没有客户记录！");  } **else** {  System.***out***.println("ID\t姓名\t年龄\t工资\t职位\t状态\t奖金\t股票\t领用设备");  }  **for** (Employee e : emps) {  System.***out***.println(" " + e);  }  System.***out***.println("-------------------------------------------------------------------------------");  } |



说明







软件启动时，根据**给定的数据**创建公司部分成员列表（数组）

通过给定的Data.java中的数据创建员工对象为NameListService类中的employees赋值

### 8、把Data.java复制到项目中

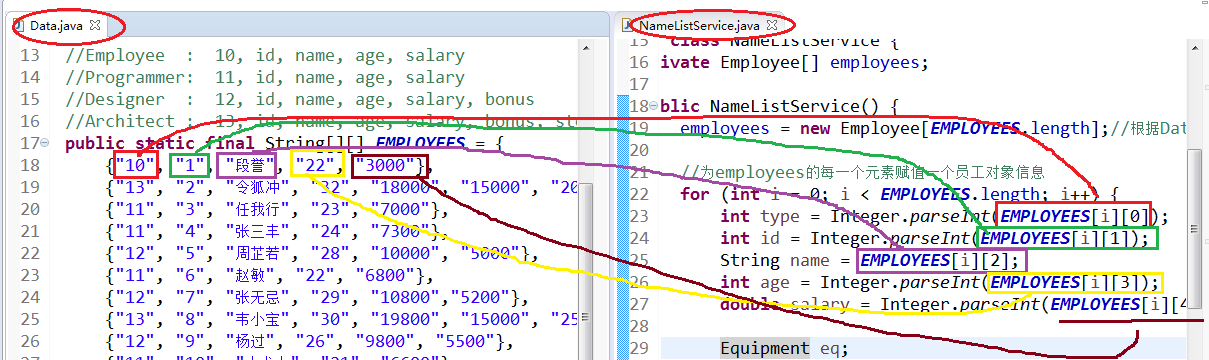
把Data.java使用记事本打开复制到com.atguigu.service.Data类中，这样复制不会有乱码，注意包package的声明是否正确。

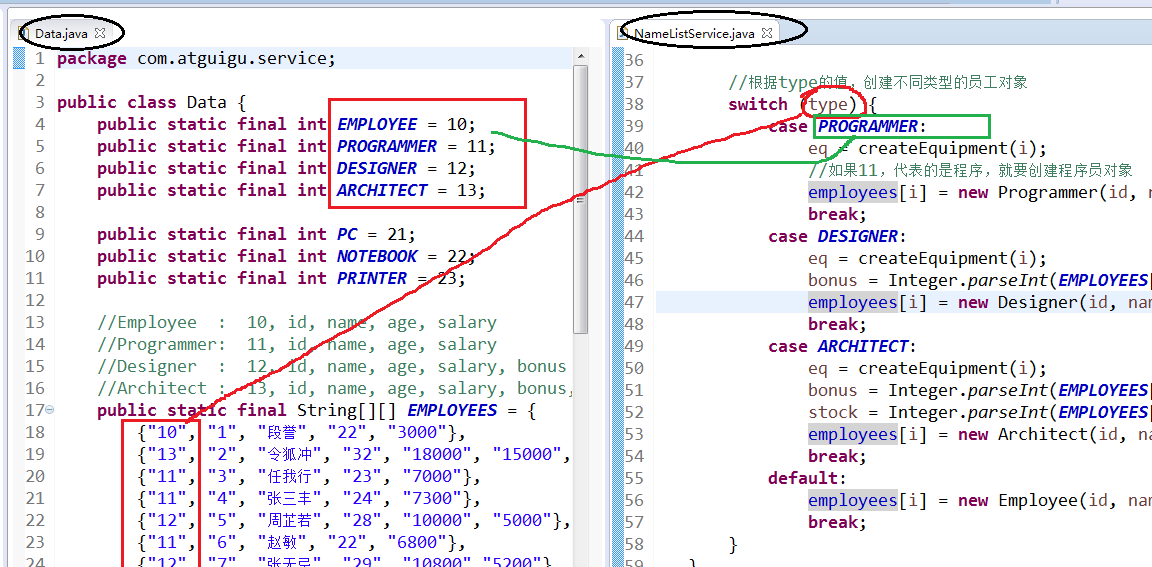
|  |
| --- |
| **package** com.atguigu.service;  **public** **class** Data {  **public** **static** **final** **int** ***EMPLOYEE*** = 10;  **public** **static** **final** **int** ***PROGRAMMER*** = 11;  **public** **static** **final** **int** ***DESIGNER*** = 12;  **public** **static** **final** **int** ***ARCHITECT*** = 13;  **public** **static** **final** **int** ***PC*** = 21;  **public** **static** **final** **int** ***NOTEBOOK*** = 22;  **public** **static** **final** **int** ***PRINTER*** = 23;  //Employee : 10, id, name, age, salary  //Programmer: 11, id, name, age, salary  //Designer : 12, id, name, age, salary, bonus  //Architect : 13, id, name, age, salary, bonus, stock  **public** **static** **final** String[][] ***EMPLOYEES*** = {  {"10", "1", "段誉", "22", "3000"},  {"13", "2", "令狐冲", "32", "18000", "15000", "2000"},  {"11", "3", "任我行", "23", "7000"},  {"11", "4", "张三丰", "24", "7300"},  {"12", "5", "周芷若", "28", "10000", "5000"},  {"11", "6", "赵敏", "22", "6800"},  {"12", "7", "张无忌", "29", "10800","5200"},  {"13", "8", "韦小宝", "30", "19800", "15000", "2500"},  {"12", "9", "杨过", "26", "9800", "5500"},  {"11", "10", "小龙女", "21", "6600"},  {"11", "11", "郭靖", "25", "7100"},  {"12", "12", "黄蓉", "27", "9600", "4800"}  };  //PC :21, model, display  //NoteBook:22, model, price  //Printer :23, type, name  **public** **static** **final** String[][] ***EQIPMENTS*** = {  {},  {"22", "联想Y5", "6000"},  {"21", "宏碁 ", "AT7-N52"},  {"21", "戴尔", "3800-R33"},  {"23", "激光", "佳能 2900"},  {"21", "华硕", "K30BD-21寸"},  {"21", "海尔", "18-511X 19"},  {"23", "针式", "爱普生20K"},  {"22", "惠普m6", "5800"},  {"21", "联想", "ThinkCentre"},  {"21", "华硕","KBD-A54M5 "},  {"22", "惠普m6", "5800"}  };  } |

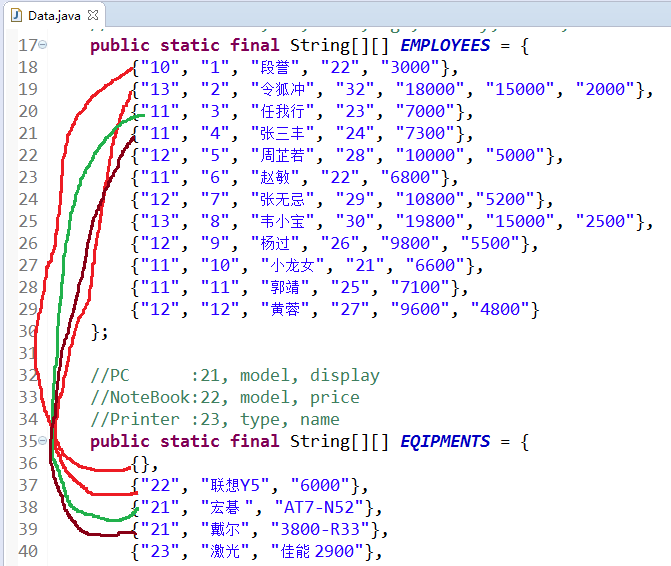
### 9、修改NameListService类

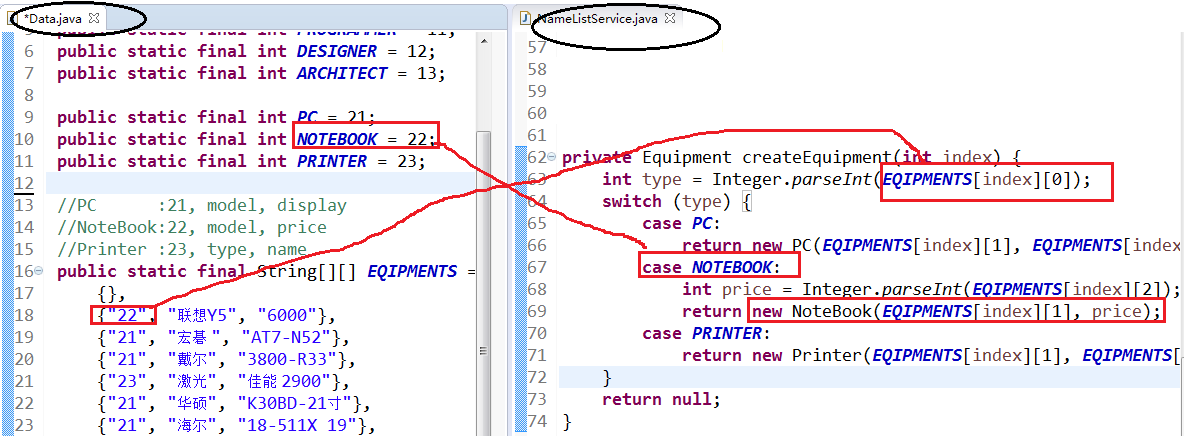
在NameListService类的构造器中为employees数组赋值。**为NameListService补充两个方法**

|  |
| --- |
| **public** NameListService() {  employees = **new** Employee[***EMPLOYEES***.length];//根据Data.EMPLOYEES数组的长度来确定员工人数  //为employees的每一个元素赋值一个员工对象信息  **for** (**int** i = 0; i < ***EMPLOYEES***.length; i++) {    //二维数组的每一行的第1列是用来区分员工是属于什么类型的  **int** type = Integer.*parseInt*(***EMPLOYEES***[i][0]);    //这些属性是每个员工都有的。剩下的属性要根据员工的职务类型不同有所不同  **int** id = Integer.*parseInt*(***EMPLOYEES***[i][1]);  String name = ***EMPLOYEES***[i][2];  **int** age = Integer.*parseInt*(***EMPLOYEES***[i][3]);  **double** salary = Integer.*parseInt*(***EMPLOYEES***[i][4]);  Equipment eq;  **double** bonus;  **int** stock;    //根据type的值，创建不同类型的员工对象  **switch** (type) {  **case** ***PROGRAMMER***:  eq = createEquipment(i);//第i行的员工对应第i行的设备  //如果11，代表的是程序，就要创建程序员对象  employees[i] = **new** Programmer(id, name, age, salary, eq);  **break**;  **case** ***DESIGNER***:  eq = createEquipment(i);  bonus = Integer.*parseInt*(***EMPLOYEES***[i][5]);  employees[i] = **new** Designer(id, name, age, salary, eq, bonus);  **break**;  **case** ***ARCHITECT***:  eq = createEquipment(i);  bonus = Integer.*parseInt*(***EMPLOYEES***[i][5]);  stock = Integer.*parseInt*(***EMPLOYEES***[i][6]);  employees[i] = **new** Architect(id, name, age, salary, eq, bonus, stock);  **break**;  **default**:  //普通员工没有设备  employees[i] = **new** Employee(id, name, age, salary);  **break**;  }  }  }  **private** Equipment createEquipment(**int** index) {  **int** type = Integer.*parseInt*(***EQIPMENTS***[index][0]);  **switch** (type) {  **case** ***PC***:  **return** **new** PC(***EQIPMENTS***[index][1], ***EQIPMENTS***[index][2]);  **case** ***NOTEBOOK***:  **int** price = Integer.*parseInt*(***EQIPMENTS***[index][2]);  **return** **new** NoteBook(***EQIPMENTS***[index][1], price);  **case** ***PRINTER***:  **return** **new** Printer(***EQIPMENTS***[index][1], ***EQIPMENTS***[index][2]);  }  **return** **null**;  } |





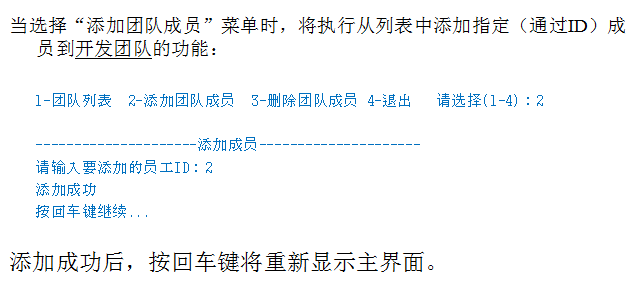


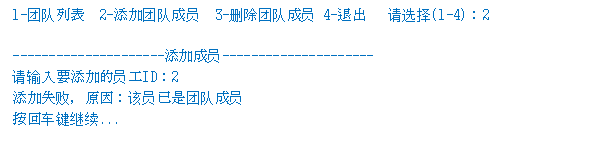


再次运行程序



### 10、实现团队成员的添加功能，完善TeamView类





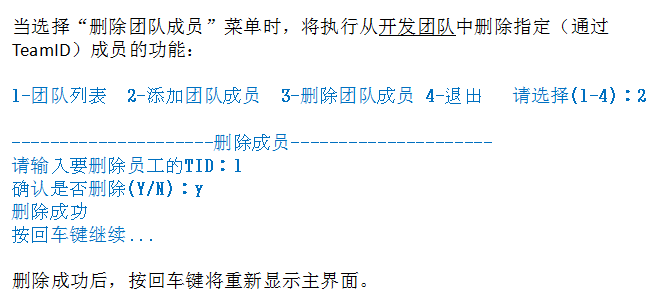
|  |
| --- |
| **private** **void** addMember() {  System.***out***.println("---------------------添加成员---------------------");  System.***out***.print("请输入要添加的员工ID：");  **int** id = TSUtility.*readInt*();  **try** {  //从NameListService类的管理的总员工数组中拿到某个员工对象  Employee e = listSvc.getEmployee(id);    //把他添加到TeamService管理的团队成员数组中  teamSvc.addMember(e);    //如果中间没有异常，就说明添加成功  System.***out***.println("添加成功");  } **catch** (TeamException e) {  System.***out***.println("添加失败，原因：" + e.getMessage());  }    TSUtility.*readReturn*();//回车继续  } |

### 11、实现团队列表功能，完善TeamView类



|  |
| --- |
| **private** **void** listTeam() {  System.***out***.println("\n--------------------团队成员列表---------------------\n");    //从TeamService类管理的团队数组中拿到所有已经添加的团队成员  Programmer[] team = teamSvc.getTeam();  **if** (team.length == 0) {  System.***out***.println("开发团队目前没有成员！");  } **else** {  System.***out***.println("TDI/ID\t姓名\t年龄\t工资\t职位\t奖金\t股票");  }  //因为不同类型的员工所包含的信息不同，因此要分别讨论显示  //注意if...else if的顺序，包含关系是“子上父下”  **for** (Programmer p : team) {  **if** (p **instanceof** Architect) {  Architect a = (Architect) p;//向下转型，才能访问架构师中的奖金与股票信息  System.***out***.println(" " + a.getMemberId() + "/" + a.getDetails() + "\t架构师\t" + a.getBonus() + "\t" + a.getStock());  } **else** **if** (p **instanceof** Designer) {  Designer d = (Designer) p;//向下转型，才能访问设计师中的奖金信息  System.***out***.println(" " + d.getMemberId() + "/" + d.getDetails() + "\t设计师\t" + d.getBonus());  } **else** **if** (p **instanceof** Programmer) {  System.***out***.println(" " + p.getMemberId() + "/" + p.getDetails() + "\t程序员");  }    }  System.***out***.println("-----------------------------------------------------");  } |

### 12、实现删除团队成员功能，完善TeamView类



|  |
| --- |
| **private** **void** deleteMember() {  System.***out***.println("---------------------删除成员---------------------");  System.***out***.print("请输入要删除员工的TID：");  **int** id = TSUtility.*readInt*();  System.***out***.print("确认是否删除(Y/N)：");  **char** yn = TSUtility.*readConfirmSelection*();  **if** (yn == 'N') **return**;  **try** {  //调用TeamService的删除团队成员方法，把某团队成员从团队数组中删除  teamSvc.removeMember(id);  System.***out***.println("删除成功");  } **catch** (TeamException e) {  System.***out***.println("删除失败，原因：" + e.getMessage());  }  TSUtility.*readReturn*();//回车继续  } |

### 13、完整的视图类TeamView

|  |
| --- |
| **package** com.atguigu.view;  **import** com.atguigu.bean.Architect;  **import** com.atguigu.bean.Designer;  **import** com.atguigu.bean.Employee;  **import** com.atguigu.bean.Programmer;  **import** com.atguigu.service.NameListService;  **import** com.atguigu.service.TeamException;  **import** com.atguigu.service.TeamService;  **import** com.atguigu.util.TSUtility;  **public** **class** TeamView {  **private** NameListService listSvc = **new** NameListService();  **private** TeamService teamSvc = **new** TeamService();  **public** **void** enterMainMenu() {  **boolean** loopFlag = **true**;  **char** key = 0;  **do** {  **if** (key != '1') listAllEmployees();  System.***out***.print("1-团队列表 2-添加团队成员 3-删除团队成员 4-退出 请选择(1-4)：");  key = TSUtility.*readMenuSelection*();  System.***out***.println();  **switch** (key) {  **case** '1':  listTeam();  **break**;  **case** '2':  addMember();  **break**;  **case** '3':  deleteMember();  **break**;  **case** '4':  System.***out***.print("确认是否退出(Y/N)：");  **char** yn = TSUtility.*readConfirmSelection*();  **if** (yn == 'Y') loopFlag = **false**;  **break**;  }  } **while** (loopFlag);  }  **private** **void** listAllEmployees() {  System.***out***.println("\n-------------------------------开发团队调度软件--------------------------------\n");  Employee[] emps = listSvc.getAllEmployees();  **if** (emps.length == 0) {  System.***out***.println("没有客户记录！");  } **else** {  System.***out***.println("ID\t姓名\t年龄\t工资\t职位\t状态\t奖金\t股票\t领用设备");  }  **for** (Employee e : emps) {  System.***out***.println(" " + e);  }  System.***out***.println("-------------------------------------------------------------------------------");  }  **private** **void** listTeam() {  System.***out***.println("\n--------------------团队成员列表---------------------\n");  Programmer[] team = teamSvc.getTeam();  **if** (team.length == 0) {  System.***out***.println("开发团队目前没有成员！");  } **else** {  System.***out***.println("TDI/ID\t姓名\t年龄\t工资\t职位\t奖金\t股票");  }  **for** (Programmer p : team) {  **if** (p **instanceof** Architect) {  Architect a = (Architect) p;  System.***out***.println(" " + a.getMemberId() + "/" + a.getDetails() + "\t架构师\t" + a.getBonus() + "\t" + a.getStock());  } **else** **if** (p **instanceof** Designer) {  Designer d = (Designer) p;  System.***out***.println(" " + d.getMemberId() + "/" + d.getDetails() + "\t设计师\t" + d.getBonus());  } **else** **if** (p **instanceof** Programmer) {  System.***out***.println(" " + p.getMemberId() + "/" + p.getDetails() + "\t程序员");  }    }  System.***out***.println("-----------------------------------------------------");  }  **private** **void** addMember() {  System.***out***.println("---------------------添加成员---------------------");  System.***out***.print("请输入要添加的员工ID：");  **int** id = TSUtility.*readInt*();  **try** {  Employee e = listSvc.getEmployee(id);  teamSvc.addMember(e);  System.***out***.println("添加成功");  } **catch** (TeamException e) {  System.***out***.println("添加失败，原因：" + e.getMessage());  }    TSUtility.*readReturn*();//回车继续  }  **private** **void** deleteMember() {  System.***out***.println("---------------------删除成员---------------------");  System.***out***.print("请输入要删除员工的TID：");  **int** id = TSUtility.*readInt*();  System.***out***.print("确认是否删除(Y/N)：");  **char** yn = TSUtility.*readConfirmSelection*();  **if** (yn == 'N') **return**;  **try** {  teamSvc.removeMember(id);  System.***out***.println("删除成功");  } **catch** (TeamException e) {  System.***out***.println("删除失败，原因：" + e.getMessage());  }  TSUtility.*readReturn*();//回车继续  }  **public** **static** **void** main(String[] args) {  TeamView view = **new** TeamView();  view.enterMainMenu();  }  } |