Java进阶1 第1天

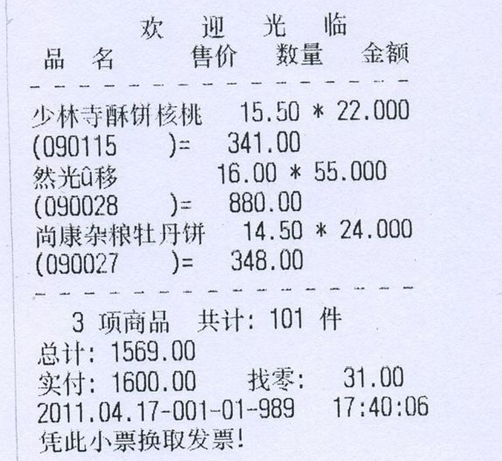
**【学习目标】理解、了解、应用、记忆**

通过今天的学习，参训学员能够：（解释的时候说出二级目标的掌握程度）

1. **【应用】独立编写超市购物小票案例**
2. 【理解】阐述超市购物小票需求及所用到的知识点
3. 【应用】独立编码定义商品项类
4. 【应用】独立编码完成小票数据初始化功能
5. 【应用】独立编码完成购物小票主干逻辑
6. 【应用】独立编码完成为商品数量和金额赋值功能
7. 【应用】独立编码完成购物小票打印功能
8. **【理解】阐述并解释面向对象相关内容**
9. 【应用】独立编码完成一个Person类的定义
10. 【应用】能够动手操作Eclipse中的三个视图
11. 【理解】阐述基本数据类型在内存中如何存储
12. 【理解】阐述引用数据类型在内存中如何存储
13. 【理解】解释this关键字的作用及用法
14. 【理解】阐述匿名对象的使用格式及使用场景
15. **【应用】继承的概述及特点**
16. 【应用】独立编写代码完成一个对象作为另一个类的成员变量的练习
17. 【理解】阐述继承的作用和好处
18. 【应用】阐述继承的代码格式,独立编写一个继承的案例
19. 【应用】独立编写代码通过set和get方法给对象设置属性
20. 【理解】阐述java中继承的特点
21. **【理解】方法重写及继承内存图解**
22. 【理解】阐述方法重写的概念及格式
23. 【了解】阐述Override注解的作用及使用方法
24. 【理解】阐述继承中子父类创建的先后顺序
25. 【理解】描述继承后子父类内存关系图
26. **【应用】this与super**
27. 【应用】阐述this和super访问普通成员的格式
28. 【应用】阐述super和this如何访问父类构造方法
29. 【理解】能够独立写出继承的案例

# 基础复习超市购物小票案例

## 超市购物小票需求分析



票脚：统计和提醒信息

票体：主体内容

票头：说明信息

模拟真实购物逻辑，以上述图片为需求原型，根据不同购买物品，完成购物小票内容打印到控制台。(简化项目逻辑，票脚部分只完成数据统计)

**数据：**

 将超市购物小票案例中零散数据(名称、货号、单价、数量、计价单位、金额)封装为货物对象。

 所有的货物对象放置到集合中统一管理。

**逻辑：**

用户循环进行三个操作：

 输入购买数量，代表为所购买货物的数量赋值，从而计算每项商品金额

 打印小票，将已有数据打印

 退出系统(因为该程序为循环操作，无法终止，如果不想再进行操作，则退出系统)

## 超市购物小票知识点分析

**复习知识点:**

变量定义：记录部分数据

集合的使用：用于存储多个完整数据

运算符的使用：计算变量

流程控制语句的使用：

if语句：用于判断是否购买了商品

switch语句：用于进行三个操作(输入购买数量、打印小票、退出系统)的分支

for语句：switch外部使用for循环，当执行一个操作后，继续进行下一个操 作选择

方法封装：

由于代码量比较多，可以将输入购买数量、打印小票封装为方法使用

字符串/键盘录入：

打印小票完整过程均为字符串的拼写及打印，数量输入为键盘录入Scanner 完成

**步骤:**

定义GoodsItem类(名称,货号,单价,数量,计价单位,金额)

成员位置定义集合，定义方法向集合中存储所有GoodsItem数据

实现主干逻辑

打印欢迎语句

使用for/while进行循环进行操作

使用switch给出操作选择：1输入购买数量2打印小票3退出

完成switch中三个case的逻辑

运用方法封装功能：将1输入购买数量、2打印小票封装为独立方法，在case中 调用，提高代码可读性。

## 超市购物小票商品项类的定义

商品项类，将要购买的商品封装到一个GoodItems类中,编码如下:

### 案例代码一

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: GoodsItem  \* **@Description**: 商品项类  \* **@date** 2017年11月13日 上午10:41:21  \* Company www.igeekhome.com  \*  \* 定义商品项类:  \* 名称,货号,单价,数量,计价单位,金额  \*  \*/  **public** **class** GoodsItem {  // 成员变量  /\*\*  \* **@Fields** name : 名称  \*/  **private** String name;  /\*\*  \* **@Fields** id : 货号  \*/  **private** String id;  /\*\*  \* **@Fields** price : 单价  \*/  **private** **double** price;  /\*\*  \* **@Fields** number : 数量  \*/  **private** **int** number;  /\*\*  \* **@Fields** unit : 计价单位  \*/  **private** String unit;  /\*\*  \* **@Fields** money : 金额  \*/  **private** **double** money;  // 构造方法  /\*\*  \* **@Title**: GoodsItem  \*/  **public** GoodsItem() {  }  /\*\*  \* **@Title**: GoodsItem  \* **@param** name  \* **@param** id  \* **@param** price  \* **@param** number  \* **@param** unit  \* **@param** money  \*/  **public** GoodsItem(String name, String id, **double** price, **int** number, String unit, **double** money) {  **this**.name = name;  **this**.id = id;  **this**.price = price;  **this**.number = number;  **this**.unit = unit;  **this**.money = money;  }  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  /\*\*  \* **@return** the id  \*/  **public** String getId() {  **return** id;  }  /\*\*  \* **@param** id the id to set  \*/  **public** **void** setId(String id) {  **this**.id = id;  }  /\*\*  \* **@return** the price  \*/  **public** **double** getPrice() {  **return** price;  }  /\*\*  \* **@param** price the price to set  \*/  **public** **void** setPrice(**double** price) {  **this**.price = price;  }  /\*\*  \* **@return** the number  \*/  **public** **int** getNumber() {  **return** number;  }  /\*\*  \* **@param** number the number to set  \*/  **public** **void** setNumber(**int** number) {  **this**.number = number;  }  /\*\*  \* **@return** the unit  \*/  **public** String getUnit() {  **return** unit;  }  /\*\*  \* **@param** unit the unit to set  \*/  **public** **void** setUnit(String unit) {  **this**.unit = unit;  }  /\*\*  \* **@return** the money  \*/  **public** **double** getMoney() {  **return** money;  }  /\*\*  \* **@param** money the money to set  \*/  **public** **void** setMoney(**double** money) {  **this**.money = money;  }  } |

## 超市购物小票初始化数据

成员位置定义集合，定义方法向集合中存储所有GoodsItem数据

### 案例代码二

|  |
| --- |
| **package** com.igeek\_04;  **import** java.util.ArrayList;  **import** com.igeek\_03.GoodsItem;  /\*\*  \* **@ClassName**: ShoppingReceipt  \* **@Description**: 超市购物小票类  \* **@date** 2017年11月13日 上午10:48:20  \* Company www.igeekhome.com  \*  \*/  **public** **class** ShoppingReceipt {    /\*\*  \* **@Fields** data : 成员位置定义集合,存储所有的商品项对象  \*/  **static** ArrayList<GoodsItem> *data* = **new** ArrayList<GoodsItem>();    **public** **static** **void** main(String[] args) {  //为集合准备数据  *initData*();  }  /\*\*  \* **@Title**: initData  \* **@Description**: 定义方法,向成员位置的集合赋值  \*/  **public** **static** **void** initData() {  //创建多个商品项对象  GoodsItem sls = **new** GoodsItem("少林寺核桃", "090115", 15.5, 0, "个", 0);  GoodsItem shk = **new** GoodsItem("尚康饼干", "090027", 14.5, 0, "个", 0);  //将多个商品项对象放入集合中  *data*.add(sls);  *data*.add(shk);  }  } |

## 超市购物小票主干逻辑

### 实现主干逻辑

打印欢迎语句

使用for/while进行循环进行操作

使用switch给出操作选择：1输入购买数量2打印小票3退出

完成switch中三个case的逻辑

运用方法封装功能：将1输入购买数量、2打印小票封装为独立方法，在case中调用，提高代码可读性。

### 案例代码三

|  |
| --- |
| **package** com.igeek\_05;  **import** java.util.ArrayList;  **import** java.util.Scanner;  **import** com.igeek\_03.GoodsItem;  /\*\*  \* **@ClassName**: ShoppingReceipt  \* **@Description**: 超市购物小票类  \* **@date** 2017年11月13日 上午10:48:20  \* Company www.igeekhome.com  \*  \*/  **public** **class** ShoppingReceipt {    /\*\*  \* **@Fields** data : 成员位置定义集合,存储所有的商品项对象  \*/  **static** ArrayList<GoodsItem> *data* = **new** ArrayList<GoodsItem>();    **public** **static** **void** main(String[] args) {  //为集合准备数据  *initData*();    //打印欢迎语句  System.***out***.println("欢迎使用超市管理系统!");    //使用死循环完成反复操作的逻辑  //for(;;) {}  **while**(**true**) {  //使用switch给出操作选择：1输入购买数量2打印小票3退出  //提示信息  System.***out***.println("请输入您要进行的操作:1输入购买数量 2打印小票 3退出");    //通过键盘录入得到数据  Scanner sc = **new** Scanner(System.***in***);  **int** optNumber = sc.nextInt();  //完成switch中三个case的逻辑  **switch** (optNumber) {  **case** 1:  System.***out***.println("给所有数据量赋值");  **break**;  **case** 2:  //调用打印购物小票的方法  System.***out***.println("打印购物小票");  **break**;  **case** 3:  System.***out***.println("欢迎下次光临");  //退出程序  System.*exit*(0);  **default**:  System.***out***.println("请输入正确的数字!");  **break**;  }  }  }  /\*\*  \* **@Title**: initData  \* **@Description**: 定义方法,向成员位置的集合赋值  \*/  **public** **static** **void** initData() {  //创建多个商品项对象  GoodsItem sls = **new** GoodsItem("少林寺核桃", "090115", 15.5, 0, "个", 0);  GoodsItem shk = **new** GoodsItem("尚康饼干", "090027", 14.5, 0, "个", 0);  //将多个商品项对象放入集合中  *data*.add(sls);  *data*.add(shk);  }  } |

## 超市购物小票为商品项赋值数量与金额

定义一个方法为所有的数据赋值数量和金额

### 案例代码四

|  |
| --- |
| **package** com.igeek\_06;  **import** java.util.ArrayList;  **import** java.util.Scanner;  **import** com.igeek\_03.GoodsItem;  /\*\*  \* **@ClassName**: ShoppingReceipt  \* **@Description**: 超市购物小票类  \* **@date** 2017年11月13日 上午10:48:20  \* Company www.igeekhome.com  \*  \*/  **public** **class** ShoppingReceipt {    /\*\*  \* **@Fields** data : 成员位置定义集合,存储所有的商品项对象  \*/  **static** ArrayList<GoodsItem> *data* = **new** ArrayList<GoodsItem>();    **public** **static** **void** main(String[] args) {  //为集合准备数据  *initData*();    //打印欢迎语句  System.***out***.println("欢迎使用超市管理系统!");    //使用死循环完成反复操作的逻辑  //for(;;) {}  **while**(**true**) {  //使用switch给出操作选择：1输入购买数量2打印小票3退出  //提示信息  System.***out***.println("请输入您要进行的操作:1输入购买数量 2打印小票 3退出");    //通过键盘录入得到数据  Scanner sc = **new** Scanner(System.***in***);  **int** optNumber = sc.nextInt();  //完成switch中三个case的逻辑  **switch** (optNumber) {  **case** 1:  //System.out.println("给所有数据量赋值");  //调用给所有数据的数量与金额赋值方法  *enterNumber*();  **break**;  **case** 2:  //调用打印购物小票的方法  System.***out***.println("打印购物小票");  **break**;  **case** 3:  System.***out***.println("欢迎下次光临");  //退出程序  System.*exit*(0);  **default**:  System.***out***.println("请输入正确的数字!");  **break**;  }  }  }  /\*\*  \* **@Title**: initData  \* **@Description**: 定义方法,向成员位置的集合赋值  \*/  **public** **static** **void** initData() {  //创建多个商品项对象  GoodsItem sls = **new** GoodsItem("少林寺核桃", "090115", 15.5, 0, "个", 0);  GoodsItem shk = **new** GoodsItem("尚康饼干", "090027", 14.5, 0, "个", 0);  //将多个商品项对象放入集合中  *data*.add(sls);  *data*.add(shk);  }      /\*\*  \* **@Title**: enterNumber  \* **@Description**: 为所有的数据赋值数量与金额  \*/  **public** **static** **void** enterNumber() {    //集合的遍历  **for** (**int** i = 0; i <*data*.size(); i++) {  //依次获取到集合中每一个元素  GoodsItem thisGoods = *data*.get(i);    //提示用户,输入该商品项的数量  //获取该商品项名称  String thisGoodsName = thisGoods.getName();  System.***out***.println("请输入"+thisGoodsName+"的购买数量");    //键盘录入接受商品数量  Scanner sc = **new** Scanner(System.***in***);  **int** thisGoodsNumber = sc.nextInt();  //根据数量计算金额金额 = 单价 \* 数量  **double** thisGoodsMoney = thisGoods.getPrice() \* thisGoodsNumber;    //为该商品的数量与金额赋值  thisGoods.setNumber(thisGoodsNumber);  thisGoods.setMoney(thisGoodsMoney);  }  }  } |

## 超市购物小票打印小票

定义打印超市购物小票方法完成小票打印功能

### 案例代码五

|  |
| --- |
| **package** com.igeek\_07;  **import** java.util.ArrayList;  **import** java.util.Scanner;  **import** com.igeek\_03.GoodsItem;  /\*\*  \* **@ClassName**: ShoppingReceipt  \* **@Description**: 超市购物小票类  \* **@date** 2017年11月13日 上午10:48:20  \* Company www.igeekhome.com  \*  \*/  **public** **class** ShoppingReceipt {    /\*\*  \* **@Fields** data : 成员位置定义集合,存储所有的商品项对象  \*/  **static** ArrayList<GoodsItem> *data* = **new** ArrayList<GoodsItem>();    **public** **static** **void** main(String[] args) {  //为集合准备数据  *initData*();    //打印欢迎语句  System.***out***.println("欢迎使用超市管理系统!");    //使用死循环完成反复操作的逻辑  //for(;;) {}  **while**(**true**) {  //使用switch给出操作选择：1输入购买数量2打印小票3退出  //提示信息  System.***out***.println("请输入您要进行的操作:1输入购买数量 2打印小票 3退出");    //通过键盘录入得到数据  Scanner sc = **new** Scanner(System.***in***);  **int** optNumber = sc.nextInt();  //完成switch中三个case的逻辑  **switch** (optNumber) {  **case** 1:  //System.out.println("给所有数据量赋值");  //调用给所有数据的数量与金额赋值方法  *enterNumber*();  **break**;  **case** 2:  //System.out.println("打印购物小票");  //调用打印购物小票的方法  *printReceipt*();  **break**;  **case** 3:  System.***out***.println("欢迎下次光临");  //退出程序  System.*exit*(0);  **default**:  System.***out***.println("请输入正确的数字!");  **break**;  }  }  }  /\*\*  \* **@Title**: initData  \* **@Description**: 定义方法,向成员位置的集合赋值  \*/  **public** **static** **void** initData() {  //创建多个商品项对象  GoodsItem sls = **new** GoodsItem("少林寺核桃", "090115", 15.5, 0, "个", 0);  GoodsItem shk = **new** GoodsItem("尚康饼干", "090027", 14.5, 0, "个", 0);  //将多个商品项对象放入集合中  *data*.add(sls);  *data*.add(shk);  }      /\*\*  \* **@Title**: enterNumber  \* **@Description**: 为所有的数据赋值数量与金额  \*/  **public** **static** **void** enterNumber() {    //集合的遍历  **for** (**int** i = 0; i <*data*.size(); i++) {  //依次获取到集合中每一个元素  GoodsItem thisGoods = *data*.get(i);    //提示用户,输入该商品项的数量  //获取该商品项名称  String thisGoodsName = thisGoods.getName();  System.***out***.println("请输入"+thisGoodsName+"的购买数量");    //键盘录入接受商品数量  Scanner sc = **new** Scanner(System.***in***);  **int** thisGoodsNumber = sc.nextInt();  //根据数量计算金额金额 = 单价 \* 数量  **double** thisGoodsMoney = thisGoods.getPrice() \* thisGoodsNumber;    //为该商品的数量与金额赋值  thisGoods.setNumber(thisGoodsNumber);  thisGoods.setMoney(thisGoodsMoney);  }  }    /\*\*  \* **@Title**: printReceipt  \* **@Description**: 打印小票方法  \*/  **private** **static** **void** printReceipt() {  //static ArrayList<GoodsItem> data = new ArrayList<GoodsItem>();  //票头  System.***out***.println(" 欢迎光临");  System.***out***.println("品名\t售价\t数量\t单位\t金额");  System.***out***.println("-------------------------------------------");  //票体  //定义变量,记录所有的商品数量  **int** totalNumber = 0;  //定义变量,记录所有的商品金额  **double** totalMoney = 0;  //遍历集合  **for** (**int** i = 0; i <*data*.size(); i++) {  //依次获取每一个商品项  GoodsItem g = *data*.get(i);  //打印商品项  System.***out***.println(g.getName()+g.getId()+"\t"+g.getPrice()+"\t"+g.getNumber()+" "+g.getUnit()+"\t"+g.getMoney());    //累加数量与金额  totalNumber += g.getNumber();  totalMoney += g.getMoney();  }  System.***out***.println("-------------------------------------------");  //票脚  System.***out***.println("共"+*data*.size()+"项商品");  System.***out***.println("共"+totalNumber+"件商品");  System.***out***.println("共"+totalMoney+"元");  System.***out***.println();  }  } |

# 面向对象的其他内容

## 一个类的定义及使用回顾

回顾定义一个类的的基本步骤和定义哪些内容

使用类映射现实生活中的事物，其功能封装为方法，属性封装为成员变量。

一个类拥有的成员包括：

成员变量

构造方法

普通方法

getters/setters

### 案例代码六

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: Person  \* **@Description**: Person类  \* **@date** 2017年11月13日 上午11:13:43  \* Company www.igeekhome.com  \*  \*/  **public** **class** Person {  // 成员变量  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name;  /\*\*  \* **@Fields** age : 年龄  \*/  **private** **int** age;  // 构造方法:构造方法不仅仅是为了给成员变量赋值,构造方法只是定义的为创建对象而产生的方法,在构造方法执行过程当中,通过其逻辑是为了给成员变量赋值,也可以有其他的逻辑存在  /\*\*  \* **@Title**: Person  \*/  **public** Person() {  }  /\*\*  \* **@Title**: Person  \* **@param** name  \* **@param** age  \*/  **public** Person(String name, **int** age) {  **this**.name = name;  **this**.age = age;  }  // 普通方法  /\*\*  \* **@Title**: eat  \* **@Description**: 吃饭的方法  \*/  **public** **void** eat() {  System.***out***.println("吃饭");  }  /\*\*  \* **@Title**: showName  \* **@Description**: 显示个人信息  \*/  **public** **void** showName() {  System.***out***.println("我的名字是:" + **this**.name + " 我的年龄是:" + **this**.age);  }  } |

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: PersonDemo  \* **@Description**: Person类的测试类  \* **@date** 2017年11月13日 上午11:17:18  \* Company www.igeekhome.com  \*  \*/  **public** **class** PersonDemo {  **public** **static** **void** main(String[] args) {  //创建对象  Person p = **new** Person("Jack",18);  //调用方法  p.eat();  p.showName();  }  } |

## Eclipse视图与颜色介绍

Eclipse的不同视图:

Package视图:为了方便开发,只显示java文件的视图

Navigator视图:硬盘上真正存储的文件视图

Outline:大纲视图,显示该类的基本成员

Outline视图中显示的成员和方法颜色表示了不同的权限修饰符:

红色:private修饰

蓝色:默认没有修饰符

黄色:protected修饰

绿色:public修饰

## 基本类型的内存解释

### 基本数据的内存图

|  |
| --- |
| C:\Users\Administrator\AppData\Roaming\Tencent\Users\1102780712\QQ\WinTemp\RichOle\HL@)06}ZPZ05QGAM53Q2OMW.png |

**基本类型赋值代码解释**

变量名 具体数值

int p = 100;

变量数据类型 赋值运算符

**基本类型赋值对应内存图**

int p = 100

栈内存栈内存

堆内存

### 案例代码七

|  |
| --- |
| **package** com.igeek\_03;  **import** com.igeek\_01.Person;  /\*\*  \* **@ClassName**: PersonDemo  \* **@Description**: Person类的测试类  \* **@date** 2017年11月13日 上午11:17:18  \* Company www.igeekhome.com  \*  \*/  **public** **class** PersonDemo {  **public** **static** **void** main(String[] args) {  //基本类型数据  **int** a = 100;    //创建对象  Person p = **new** Person("Jack",18);  //调用方法  p.eat();  p.showName();  }  } |

## 引用类型的内存解释

### C:\Users\Administrator\AppData\Roaming\Tencent\Users\1102780712\QQ\WinTemp\RichOle\V7$%CJ~8~E@IN%S3042{XW0.png引用数据内存图

**引用类型赋值代码解释**

变量名 创建对象固定关键字

Person p = new Person(); 具体数值为内存地址

变量数据类型 赋值运算符 具体数值 (0x1234)

创建对象目前固定写法

### 案例代码八

|  |
| --- |
| **package** com.igeek\_04;  **import** com.igeek\_01.Person;  /\*\*  \* **@ClassName**: PersonDemo  \* **@Description**: Person类的测试类  \* **@date** 2017年11月13日 上午11:17:18  \* Company www.igeekhome.com  \*  \*/  **public** **class** PersonDemo {  **public** **static** **void** main(String[] args) {  //引用类型数据  //创建对象  Person p = **new** Person("Jack",18);  //调用方法  p.eat();  p.showName();  }  } |

## this关键字

### this关键字概述

this一般出现在类的一个方法的定义当中，代表当前对象的引用。我们通常用其将本对象中的成员与其他作用域区分开来。

当一个类定义好之后，如果不创建对象this是没有意义的。

一个类可以有多个对象。每个对象都有各自的属性值，各个对象的方法均是类中定义的那个方法逻辑。定义中的this就代表调用方法的这个对象。

Person类

class Person{

private String name;

private int age;

public void showName() {

System.out.println(this.name);

}

set/get方法

}

Person p = new Person();

p.setName(“郭德纲”);

p.setAge(34);

p.showName(); //打印郭德纲

Person p2 = new Person();

p2.setName(“林志颖”);

p2.setAge(34);

p2.showName (); //打印林志颖

p对象 p2对象

String name; 郭德纲

int age; 34

public void showName() { System.out.println(this.name);

}

get/set方法

System.out.println(this.name);

}

get/set方法

String name; 林志颖

int age; 34

public void showName() { System.out.println(this.name);

}

get/set方法

System.out.println(this.name);

}

get/set方法

### 案例代码九

|  |
| --- |
| **package** com.igeek\_05;  /\*\*  \* **@ClassName**: Person  \* **@Description**: Person类  \* **@date** 2017年11月13日 上午11:13:43  \* Company www.igeekhome.com  \*  \*/  **public** **class** Person {  // 成员变量  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name;  /\*\*  \* **@Fields** age : 年龄  \*/  **private** **int** age;  // 构造方法:构造方法不仅仅是为了给成员变量赋值,构造方法只是定义的为创建对象而产生的方法,在构造方法执行过程当中,通过其逻辑是为了给成员变量赋值,也可以有其他的逻辑存在  /\*\*  \* **@Title**: Person  \*/  **public** Person() {  }  /\*\*  \* **@Title**: Person  \* **@param** name  \* **@param** age  \*/  **public** Person(String name, **int** age) {  **this**.name = name;  **this**.age = age;  }  // 普通方法  /\*\*  \* **@Title**: eat  \* **@Description**: 吃饭的方法  \*/  **public** **void** eat() {  System.***out***.println("吃饭");  }  /\*\*  \* **@Title**: showName  \* **@Description**: 显示个人信息  \*/  **public** **void** showName() {  System.***out***.println("我的名字是:" + **this**.name + " 我的年龄是:" + **this**.age);  }  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  /\*\*  \* **@return** the age  \*/  **public** **int** getAge() {  **return** age;  }  /\*\*  \* **@param** age the age to set  \*/  **public** **void** setAge(**int** age) {  **this**.age = age;  }  } |

|  |
| --- |
| **package** com.igeek\_05;  /\*\*  \* **@ClassName**: ThisDemo  \* **@Description**: this关键字概述  \* **@date** 2017年11月13日 下午2:07:58  \* Company www.igeekhome.com  \*  \* 使用this可以区分成员变量与局部变量,处理成员变量与局部变量同名的情况  \*  \* 如果一个方法内访问不带this的变量,则遵循以下顺序:先查找局部变量,再查找成员变量  \* 如果一个方法内访问带this的变量,则只在成员位置找  \*  \* this的意义:代表当前对象的引用  \*  \* 类的方法定义过程当中可以定义this,代表当前调用方法的对象,是一个不确定的对象.  \* 即简单记忆为哪个对象调用方法,则方法中的this代表哪个对象  \*/  **public** **class** ThisDemo {  **public** **static** **void** main(String[] args) {  //创建对象  Person p = **new** Person("郭德纲",34);  p.showName();    //创建对象  Person p2 = **new** Person("林志颖",34);  p2.showName();  }  } |

## 匿名对象

### 匿名对象的概述

匿名对象即没有引用的对象，没有名字的对象，与有名对象对比如下：

Person p = new Person();

该对象是有名字为p的一个Person对象，可以调用方法。

如： p.showName();

new Person(); 该对象是没有名字的一个对象，同样可以调用方法。

如：new Person().showName();

### 案例代码十

|  |
| --- |
| **package** com.igeek\_06;  **import** com.igeek\_05.Person;  /\*\*  \* **@ClassName**: AnonymousObjectDemo  \* **@Description**: 匿名对象的使用  \* **@date** 2017年11月13日 下午2:13:30  \* Company www.igeekhome.com  \*  \* 匿名对象：没有引用的对象，没有名字的对象  \* 有名字(引用)的对象，使用引用调用该对象的方法  \* 匿名对象创建之后，立即使用对象自身调用对象即可  \*  \* 有引用的对象，可以通过引用，反复操作同一个对象，可以重用  \* 匿名对象因为没有引用变量指向，所以一般情况下只能使用一次  \*/  **public** **class** AnonymousObjectDemo {  **public** **static** **void** main(String[] args) {  //有名字的对象  Person p = **new** Person("Rose",16);  p.showName();  Person p2 = **new** Person("Jack",18);  p2.setName("牛顿");  p2.showName();    //匿名对象  **new** Person("怀特",12).showName();  **new** Person("特朗普",64).setName("川普");  **new** Person("特朗普",64).showName();  }  } |

# 继承的概述及特点

## 组合关系

### 组合关系的概述

当一个自定义类型A的成员变量的数据类型是自定义类型B时，A类与B类叫做组合关系。如：

人Person与宠物Pet就是一种组合的关系，是一种类与类之间的设计关系。

人Person与String也是一种组合关系，只是我们习惯了String的用法。

### 案例代码十一

**Pet宠物类:**

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: Pet  \* **@Description**: Pet宠物类  \* **@date** 2017年11月13日 下午2:47:09  \* Company www.igeekhome.com  \*  \* 宠物姓名,宠物颜色,宠物种类三个成员变量  \*/  **public** **class** Pet {  /\*\*  \* **@Fields** name : 宠物姓名  \*/  **private** String name;  /\*\*  \* **@Fields** color : 宠物颜色  \*/  **private** String color;  /\*\*  \* **@Fields** kind : 宠物种类  \*/  **private** String kind;  /\*\*  \* **@Title**: Pet  \*/  **public** Pet() {  }  /\*\*  \* **@Title**: Pet  \* **@param** name  \* **@param** color  \* **@param** kind  \*/  **public** Pet(String name, String color, String kind) {  **this**.name = name;  **this**.color = color;  **this**.kind = kind;  }  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  /\*\*  \* **@return** the color  \*/  **public** String getColor() {  **return** color;  }  /\*\*  \* **@param** color the color to set  \*/  **public** **void** setColor(String color) {  **this**.color = color;  }  /\*\*  \* **@return** the kind  \*/  **public** String getKind() {  **return** kind;  }  /\*\*  \* **@param** kind the kind to set  \*/  **public** **void** setKind(String kind) {  **this**.kind = kind;  }  } |

**Person类:**

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: Person  \* **@Description**: Person类  \* **@date** 2017年11月13日 下午2:46:37  \* Company www.igeekhome.com  \*  \* 姓名,年龄,宠物三个成员变量  \* 有showInfo(显示信息)的方法  \*  \* 当Pet作为Person的成员变量时,两个数据类型的关系为组合  \*/  **public** **class** Person {  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name;  /\*\*  \* **@Fields** age : 年龄  \*/  **private** **int** age;    /\*\*  \* **@Fields** pet : 宠物  \*/  **private** Pet pet;  /\*\*  \* **@Title**: Person  \*/  **public** Person() {  }  /\*\*  \* **@Title**: Person  \* **@param** name  \* **@param** age  \* **@param** pet  \*/  **public** Person(String name, **int** age, Pet pet) {  **this**.name = name;  **this**.age = age;  **this**.pet = pet;  }  /\*\*  \* **@Title**: showInfo  \* **@Description**: 显示个人信息  \*/  **public** **void** showInfo() {  System.***out***.println("我的名字是:" + **this**.name + "，我的年龄是:" + **this**.age);  System.***out***.println("我的宠物是"+pet.getColor()+"色的"+pet.getKind()+",它叫"+pet.getName());  }  } |

**CompositionDemo 测试类**

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: CompositionDemo  \* **@Description**: 组合关系的测试类  \* **@date** 2017年11月13日 下午3:02:16  \* Company www.igeekhome.com  \*  \* 当一个自定义类型A的成员变量的数据类型是自定义类型B时，A类与B类叫做组合关系。  \*/  **public** **class** CompositionDemo {  **public** **static** **void** main(String[] args) {  //准备新创建的Person对象的属性值  String name = "Jack";  **int** age = 18;  Pet pet = **new** Pet("大黄","黄","大狼狗");    //创建Person对象  Person p = **new** Person(name, age, pet);    //调用Person对象的方法  p.showInfo();  }  } |

## 继承的概念

### 继承的概述

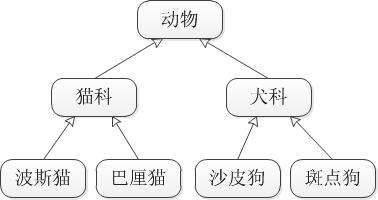
继承是面向对象的核心特性，是面向对象的学习重点。

继承是代码复用的重要方式，是类与类之间的一种关系。

从类与类之间的设计关系来看,子类必须属于父类的一种时，才会继承。

父类抽取出了共性的内容，子类可以在父类基础上扩展新的属性与行为。

子类拥有父类的所有属性与方法，无需重新定义。并且可以直接使用非私有的父类成员。

 下例展示了一个继承关系

### 案例代码十二

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: ExtendsDemo  \* **@Description**: 继承关系  \* **@date** 2017年11月13日 下午3:09:08  \* Company www.igeekhome.com  \*  \* 继承关系:类与类之间的关系  \*  \* 人:定义一些共性内容定义为父类  \*  \* 共性的成员变量:年龄,姓名  \* 共性的成员方法:吃饭,睡觉  \*  \* 程序员、老师、工人定义为人类的子类,子类可以在父类的基础上扩展新的成员变量与成员方法  \*  \* 当有了子父类关系后,子类拥有了父类的成员,可以直接访问父类的非私有成员  \* 继承通过这样的方式,提高了代码的复用性,让类与类之间产生了关系  \* 继承关系必须符合is a的关系,即我是你的一种  \*  \*/  **public** **class** ExtendsDemo {  **public** **static** **void** main(String[] args) {    }  } |

## 继承的代码格式

### 继承格式说明

使用extends关键字完成继承关系。

注意：所有的类都直接或间接继承自Object，所以如果没有继承其他类，则默认继承Object类。即Object是所有类的父类(或超类)

格式：

class 子类 extends 父类 {}

### 案例代码十三

**父类Person类：**

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Person  \* **@Description**: Person类  \* **@date** 2017年11月13日 下午3:36:21  \* Company www.igeekhome.com  \*  \* 父类-人类:  \* 年龄,姓名  \* 吃饭,睡觉  \*/  **public** **class** Person **extends** Object {  /\*\*  \* **@Fields** name : 姓名  \*/  String name;  /\*\*  \* **@Fields** age : 年龄  \*/  **int** age;    /\*\*  \* **@Title**: Person  \*/  **public** Person() {  }  /\*\*  \* **@Title**: Person  \* **@param** name  \* **@param** age  \*/  **public** Person(String name, **int** age) {  **this**.name = name;  **this**.age = age;  }    /\*\*  \* **@Title**: eat  \* **@Description**: 吃东西的方法  \*/  **public** **void** eat() {  System.***out***.println("我吃了");  }  /\*\*  \* **@Title**: sleep  \* **@Description**: 睡觉的方法  \*/  **public** **void** sleep() {  System.***out***.println("我睡了");  }  } |

**定义子类Coder：**

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Coder  \* **@Description**: 程序员类  \* **@date** 2017年11月13日 下午3:43:48  \* Company www.igeekhome.com  \*  \* 定义子类程序员类,继承人类  \* 在人类共性内容的基础上,添加编程功能  \*/  **public** **class** Coder **extends** Person {    /\*\*  \* **@Title**: code  \* **@Description**: 编写代码的方法  \*/  **public** **void** code() {  System.***out***.println("我写了一个百度!");  }  } |

**定义子类Teacher：**

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Teacher  \* **@Description**: 老师类  \* **@date** 2017年11月13日 下午3:46:15  \* Company www.igeekhome.com  \*  \* 定义子类老师类,继承人类  \* 在人类共性内容的基础上,添加教学功能  \*/  **public** **class** Teacher **extends** Person {  /\*\*  \* **@Title**: teach  \* **@Description**: 教书的方法  \*/  **public** **void** teach() {  System.***out***.println("传授知识!");  }  } |

**测试类ExtendsDemo：**

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: ExtendsDemo  \* **@Description**: 继承关系的测试类  \* **@date** 2017年11月13日 下午3:49:55  \* Company www.igeekhome.com  \*  \* 继承关系：类与类之间的关系  \*  \* 人：定义一些共性内容 定义为父类  \* 共性的成员变量：姓名、年龄  \* 共性的成员方法：吃饭、睡觉  \*  \* 程序员、老师、工人 定义为人类的子类，子类可以在父类的基础上扩展新的成员变量与方法  \*  \* 当有了子父类关系后，子类拥有了父类的成员，可以直接访问父类的非私有成员  \* 继承通过这样的方式，提高了代码的复用性，让类与类之间产生了关系  \* 继承关系必须符合is a的关系，即我是你的一种  \* 继承的格式:  \* public class 子类 extends 父类 {  \*  \* }  \*  \* 注意:  \* 所有的类(除Object类)都有父类,所有的类都直接或间接是Object的子类  \*/  **public** **class** ExtendsDemo {  **public** **static** **void** main(String[] args) {  //创建子类对象  Coder coder = **new** Coder();    //使用子类可以访问父类的非私有成员变量  coder.name = "拜伦";  coder.age = 18;  System.***out***.println(coder.name+"的年龄是:"+coder.age);  //使用子类可以访问父类的非私有成员方法  coder.eat();  coder.sleep();  //使用子类可以访问子类的自身的成员方法  coder.code();  }  } |

## 子类通过get&set方法访问父类成员变量

当父类的成员变量用private修饰后,子类无法访问父类的成员变量,但是如果父类提供了public修饰的get/set方法,则子类可以通过get/set方法,正常访问父类的成员变量。

### 代码案例十四

**父类Person类：**

|  |
| --- |
| **package** com.igeek\_04;  /\*\*  \* **@ClassName**: Person  \* **@Description**: Person类  \* **@date** 2017年11月13日 下午3:36:21  \* Company www.igeekhome.com  \*  \* 父类-人类:  \* 年龄,姓名  \* 吃饭,睡觉  \*/  **public** **class** Person **extends** Object {  //使用private修饰后,子类无法访问  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name;  /\*\*  \* **@Fields** age : 年龄  \*/  **private** **int** age;    /\*\*  \* **@Title**: Person  \*/  **public** Person() {  }  /\*\*  \* **@Title**: Person  \* **@param** name  \* **@param** age  \*/  **public** Person(String name, **int** age) {  **this**.name = name;  **this**.age = age;  }    /\*\*  \* **@Title**: eat  \* **@Description**: 吃东西的方法  \*/  **public** **void** eat() {  System.***out***.println("我吃了");  }  /\*\*  \* **@Title**: sleep  \* **@Description**: 睡觉的方法  \*/  **public** **void** sleep() {  System.***out***.println("我睡了");  }  //成员变量被private修饰,子类无法直接访问,但是可以通过get/set方法访问  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  /\*\*  \* **@return** the age  \*/  **public** **int** getAge() {  **return** age;  }  /\*\*  \* **@param** age the age to set  \*/  **public** **void** setAge(**int** age) {  **this**.age = age;  }  } |

**定义子类Coder：**

|  |
| --- |
| **package** com.igeek\_04;  /\*\*  \* **@ClassName**: Coder  \* **@Description**: 程序员类  \* **@date** 2017年11月13日 下午3:43:48  \* Company www.igeekhome.com  \*  \* 定义子类程序员类,继承人类  \* 在人类共性内容的基础上,添加编程功能  \*/  **public** **class** Coder **extends** Person {    /\*\*  \* **@Title**: code  \* **@Description**: 编写代码的方法  \*/  **public** **void** code() {  System.***out***.println("我写了一个百度!");  }  } |

**测试类ExtendsDemo：**

|  |
| --- |
| **package** com.igeek\_04;  /\*\*  \* **@ClassName**: ExtendsDemo  \* **@Description**: 继承关系的测试类  \* **@date** 2017年11月13日 下午3:49:55  \* Company www.igeekhome.com  \*  \* 当父类的成员变量用private修饰后,子类无法访问父类的成员变量,  \* 但是如果父类提供了public修饰的get/set方法,则子类可以通过get/set方法,正常访问父类的成员变量  \*/  **public** **class** ExtendsDemo {  **public** **static** **void** main(String[] args) {  //创建子类对象  Coder coder = **new** Coder();    //使用子类可以通过public修饰的get/set方法访问父类的私有成员变量  coder.setName("拜伦");  coder.setAge(18);  System.***out***.println(coder.getName()+"的年龄是:"+coder.getAge()); //使用子类可以访问父类的非私有成员方法  coder.eat();  coder.sleep();  //使用子类可以访问子类的自身的成员方法  coder.code();  }  } |

## 继承的特点

### 特点概述

Java支持单继承

Java支持多层继承

父类定义了继承树中共性内容，子类定义了该类个性内容。

在结合多态后，能使用父类时尽量使用父类，提高程序扩展性。

### 案例代码十五

**测试类：ExtendsNotesDemo**

|  |
| --- |
| **package** com.igeek\_05;  /\*\*  \* **@ClassName**: ExtendsNotesDemo  \* **@Description**: 继承特点  \* **@date** 2017年11月13日 下午4:40:08  \* Company www.igeekhome.com  \*  \* Java支持单继承  \* Java支持多层继承  \*/  **public** **class** ExtendsNotesDemo {  **public** **static** **void** main(String[] args) {    }  } |

**父类Fu1**

|  |
| --- |
| **package** com.igeek\_05;  /\*\*  \* **@ClassName**: Fu1  \* **@Description**: 父类1  \* **@date** 2017年11月13日 下午4:37:16  \* Company www.igeekhome.com  \*  \*/  **public** **class** Fu1 {  } |

**父类Fu2**

|  |
| --- |
| **package** com.igeek\_05;  /\*\*  \* **@ClassName**: Fu2  \* **@Description**: 父类2  \* **@date** 2017年11月13日 下午4:37:41  \* Company www.igeekhome.com  \*  \*/  **public** **class** Fu2 {  } |

**子类Zi1**

|  |
| --- |
| **package** com.igeek\_05;  /\*\*  \* **@ClassName**: Zi1  \* **@Description**: 子类  \* **@date** 2017年11月13日 下午4:38:32  \* Company www.igeekhome.com  \*  \* 一个子类不能拥有多个直接父类,继承只支持单继承  \* public class Zi1 extends Fu1,Fu2{  \*  \* }  \*  \* 一个子类可以有一个父类  \*/  **public** **class** Zi1 **extends** Fu1 {  } |

**孙子类Sun1**

|  |
| --- |
| **package** com.igeek\_05;  /\*\*  \* **@ClassName**: Sun1  \* **@Description**: 孙子类  \* **@date** 2017年11月13日 下午4:38:58  \* Company www.igeekhome.com  \*  \* Sun1类直接继承Zi1类,间接继承Fu1类与Object类  \* 继承存在多层继承  \*/  **public** **class** Sun1 **extends** Zi1 {  } |

# 方法重写及继承内存图解

## 方法重写的概念与格式

### 方法重写的概述

当子类继承父类后，拥有了父类的成员并可以直接调用父类非私有方法。如果子类认为父类提供的方法不够强大，子类可以按照子类自身的逻辑重新定义继承过来的父类方法，这个重新定义父类方法的过程叫做方法重写。

方法重写后，调用该方法时不再调用父类的方法，而调用子类重写后的方法。

举例：

当描述一个人类(父类)时，它具有吃饭，睡觉等功能

中国人类(子类)吃饭比较复杂，使用筷子吃饭，则可以继承原有的人类。并重写吃饭功能，在其中修改为更为复杂的用筷子吃的逻辑即可。

(注：在学习完多态和抽象类后我们会对方法重写有更深的理解)

### 方法重写的代码格式

**格式:**

定义一个与父类方法声明完全相同的方法,方法体重写

#### 代码案例十六

**测试类：OverrideDemo**

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: OverrideDemo  \* **@Description**: override测试类  \* **@date** 2017年11月13日 下午5:01:16  \* Company www.igeekhome.com  \*  \* 方法重写:  \* 当子类想加强父类继承过来的方法时,可以重写该方法  \*  \* 重写格式:  \* 定义一个与父类方法声明完全相同的方法,方法体重写  \*/  **public** **class** OverrideDemo {  **public** **static** **void** main(String[] args) {  //创建子类对象  Dog dog = **new** Dog();    //调用方法  //调用了子类重写的方法功能  dog.eat();  //调用了还是子类从父类继承下来的方法功能  dog.sleep();  }  } |

**定义父类Animal**

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: Animal  \* **@Description**: 父类Animal  \* **@date** 2017年11月13日 下午5:08:09  \* Company www.igeekhome.com  \*  \*/  **public** **class** Animal {  /\*\*  \* **@Title**: eat  \* **@Description**: 吃东西的方法  \*/  **public** **void** eat() {  System.***out***.println("吃了!");  }    /\*\*  \* **@Title**: sleep  \* **@Description**: 睡觉的方法  \*/  **public** **void** sleep() {  System.***out***.println("睡了!");  }  } |

**定义子类Dog类**

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: Dog  \* **@Description**: 子类狗类  \* **@date** 2017年11月13日 下午5:09:41  \* Company www.igeekhome.com  \*  \* 子类狗类:  \* 增强父类的吃饭方法,于是可以重写该方法  \*  \* 格式:  \* 定义一个与父类方法声明完全相同的方法,方法体重写  \*/  **public** **class** Dog **extends** Animal {    /\*\*  \* **@Title**: eat  \* **@Description**: 狗吃东西的方法  \*/  **public** **void** eat() {  System.***out***.println("我是小狗,我汪汪汪着吃骨头!");  }  } |

## Override注解解释

### Override

子类重写父类的方法时在方法上添加 @Override注解,表示该方法是子类重写父类中的方法。

### 代码案例十七

**测试类：OverrideDemo**

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: OverrideDemo  \* **@Description**: override测试类  \* **@date** 2017年11月13日 下午5:12:16  \* Company www.igeekhome.com  \*  \* 方法重写:  \* 当子类想加强父类继承过来的方法时,可以重写该方法  \*  \* 重写格式:  \* 定义一个与父类方法声明完全相同的方法,方法体重写  \*/  **public** **class** OverrideDemo {  **public** **static** **void** main(String[] args) {  //创建子类对象  Dog dog = **new** Dog();    //调用方法  //调用了子类重写的方法功能  dog.eat();  //调用了还是子类从父类继承下来的方法功能  dog.sleep();  }  } |

**定义父类Animal**

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: Animal  \* **@Description**: 父类Animal  \* **@date** 2017年11月13日 下午5:08:09  \* Company www.igeekhome.com  \*  \*/  **public** **class** Animal {  /\*\*  \* **@Title**: eat  \* **@Description**: 吃东西的方法  \*/  **public** **void** eat() {  System.***out***.println("吃了!");  }    /\*\*  \* **@Title**: sleep  \* **@Description**: 睡觉的方法  \*/  **public** **void** sleep() {  System.***out***.println("睡了!");  }  } |

**定义子类Dog类**

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: Dog  \* **@Description**: 子类狗类  \* **@date** 2017年11月13日 下午5:09:41  \* Company www.igeekhome.com  \*  \* 子类狗类:  \* 增强父类的吃饭方法,于是可以重写该方法  \*  \* 格式:  \* 定义一个与父类方法声明完全相同的方法,方法体重写  \*/  **public** **class** Dog **extends** Animal {  //方法重写的注解  /\*\*  \* **@Title**: eat  \* **@Description**: 狗吃东西的方法  \* **@see** com.igeek\_02.Animal#eat()  \*/  @Override  **public** **void** eat() {  System.***out***.println("我是小狗,我汪汪汪着吃骨头!");  }    // @Override用于限定该方法必须是重写父类的方法  // @Override  // public void eet() {  // System.out.println("我是小狗,我汪汪汪着吃骨头!");  // }  } |

## 方法重写注意事项

### 注意事项概述：

方法名称必须相同

参数列表必须相同

访问权限相同或子类方法访问权限更大(访问权限顺序public>默认)

返回值为一般必须相同(并不适用于所有情况)

### 代码案例十八

**父类Animal**

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Animal  \* **@Description**: 父类Animal  \* **@date** 2017年11月13日 下午5:08:09  \* Company www.igeekhome.com  \*  \*/  **public** **class** Animal {  /\*\*  \* **@Title**: eat  \* **@Description**: 吃东西的方法  \*/  **void** eat() {  System.***out***.println("吃了!");  }    /\*\*  \* **@Title**: sleep  \* **@Description**: 睡觉的方法  \*/  **void** sleep() {  System.***out***.println("睡了!");  }  } |

**子类Dog**

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Dog  \* **@Description**: 子类狗类  \* **@date** 2017年11月13日 下午5:09:41  \* Company www.igeekhome.com  \*  \* 方法重写的注意事项:  \* 访问权限:父类的访问权限 小于等于 子类的访问权限  \* 默认权限 < public  \*/  **public** **class** Dog **extends** Animal {    //方法重写子类权限大于或等于父类权限  //返回值类型相同  //方法名必须相同  //方法参数必须相同  /\*\*  \* **@Title**: eat  \* **@Description**: 狗吃东西的方法  \* **@see** com.igeek\_02.Animal#eat()  \*/  @Override  **public** **void** eat() {  System.***out***.println("我是小狗,我汪汪汪着吃骨头!");  }    } |

## 继承后子父类构造调用先后顺序

### 继承中子父类构造调用顺序概述

在每次创建子类对象时，我们均会先初始化父类内容，再初始化其子类本身内容。目的在于子类对象中包含了其对应的父类存储空间，便可以包含了其父类的成员，如果父类成员非private修饰，则子类可以随意使用父类成员。

代码体现在子类的构造方法调用时，一定先调用父类的构造方法。

### 代码案例十九

**测试类ConstructorDemo**

|  |
| --- |
| **package** com.igeek\_04;  /\*\*  \* **@ClassName**: ConstructorDemo  \* **@Description**: 构造方法测试类  \* **@date** 2017年11月13日 下午5:37:07  \* Company www.igeekhome.com  \*  \* 子类的构造方法会先调用父类的无参构造，从而保证父类的相关内容会先于子类内容的初识化  \* 子类就可以使用父类的内容了\*  \*/  **public** **class** ConstructorDemo {  **public** **static** **void** main(String[] args) {  //创建子类对象,验证注释部分的结论  Zi zi = **new** Zi();  }  } |

**父类Fu**

|  |
| --- |
| **package** com.igeek\_04;  /\*\*  \* **@ClassName**: Fu  \* **@Description**: Fu父类  \* **@date** 2017年11月13日 下午5:34:15  \* Company www.igeekhome.com  \*  \*/  **public** **class** Fu {  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name;  /\*\*  \* **@Title**: Fu  \*/  **public** Fu() {  // super();  System.***out***.println("父类的空参构造方法被调用了");  }  /\*\*  \* **@Title**: Fu  \* **@param** name  \*/  **public** Fu(String name) {  // super();  **this**.name = name;  System.***out***.println("父类的带参构造方法被调用了");  }  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }    } |

**子类Zi**

|  |
| --- |
| **package** com.igeek\_04;  /\*\*  \* **@ClassName**: Zi  \* **@Description**: Zi子类  \* **@date** 2017年11月13日 下午5:36:25  \* Company www.igeekhome.com  \*  \*/  **public** **class** Zi **extends** Fu {  **public** Zi() {  **super**();//这个super语句决定了子类构造方法调用父类空参构造  System.***out***.println("子类的构造方法被调用了");  }  } |

## 继承后子父类内存图

### 代码案例二十

测试类：

|  |
| --- |
| **package** com.igeek\_05;  /\*\*  \* **@ClassName**: ThisSuperDemo  \* **@Description**: 继承测试类  \* **@date** 2017年11月13日 下午6:37:11  \* Company www.igeekhome.com  \*  \* this:当前对象  \* super:父类存储空间  \*  \* Person父类  \* name,age  \*  \* Chinese子类  \* address  \*/  **public** **class** ThisSuperDemo {  **public** **static** **void** main(String[] args) {  //创建子类对象  Chinese c = **new** Chinese("Jack",18,"中南海");  //调用该类方法  System.***out***.println(c.getName()+"住在"+c.getAddress());  }  } |

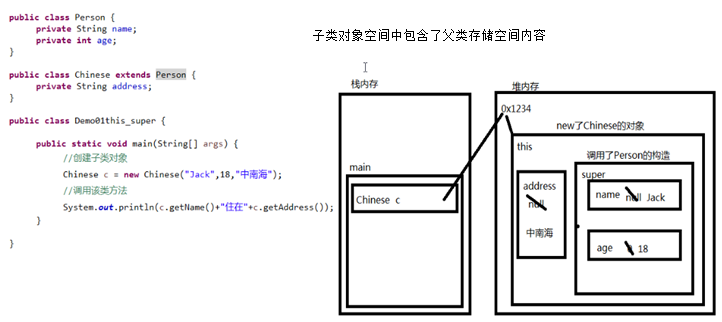
**定义父类Person**

|  |
| --- |
| **package** com.igeek\_05;  /\*\*  \* **@ClassName**: Person  \* **@Description**: Person类  \* **@date** 2017年11月13日 下午3:36:21  \* Company www.igeekhome.com  \*  \* 父类-人类:  \* 属性：name，age  \*/  **public** **class** Person **extends** Object {  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name;  /\*\*  \* **@Fields** age : 年龄  \*/  **private** **int** age;    /\*\*  \* **@Title**: Person  \*/  **public** Person() {  }  /\*\*  \* **@Title**: Person  \* **@param** name  \* **@param** age  \*/  **public** Person(String name, **int** age) {  **this**.name = name;  **this**.age = age;  }    /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  /\*\*  \* **@return** the age  \*/  **public** **int** getAge() {  **return** age;  }  /\*\*  \* **@param** age the age to set  \*/  **public** **void** setAge(**int** age) {  **this**.age = age;  }  } |

**定义子类Chinese**

|  |
| --- |
| **package** com.igeek\_05;  /\*\*  \* **@ClassName**: Chinese  \* **@Description**: Chinese子类  \* **@date** 2017年11月13日 下午6:33:43  \* Company www.igeekhome.com  \*  \* 在父类Person的基础上,多了地址属性：  \* address  \*/  **public** **class** Chinese **extends** Person{    /\*\*  \* **@Fields** address : 地址  \*/  **private** String address;  /\*\*  \* **@Title**: Chinese  \*/  **public** Chinese() {  **super**();  }  //定义三个参数的构造方法,无需了解细节,在super的讲解时会详细介绍  /\*\*  \* **@Title**: Chinese  \* **@param** name  \* **@param** age  \* **@param** address  \*/  **public** Chinese(String name, **int** age,String address) {  **super**(name, age);  **this**.address = address;  }  /\*\*  \* **@return** the address  \*/  **public** String getAddress() {  **return** address;  }  /\*\*  \* **@param** address the address to set  \*/  **public** **void** setAddress(String address) {  **this**.address = address;  }  } |

### 对象内存图



# this与super

## this与super访问普通成员

### this和super访问注意事项

调用普通成员：

this.成员变量 可以访问本类对象的成员变量

super.成员变量 可以访问父类的成员变量

this.成员方法() 可以访问本类对象的成员方法

super.成员方法() 可以访问父类的成员方法

子类方法中

访问子类自身的成员用this.

访问父类的成员super.

就近原则:

局部 > 本类成员 > 父类成员

### 代码案例二十一

**测试类ThisSuperDemo**

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: ThisSuperDemo  \* **@Description**: 继承测试类  \* **@date** 2017年11月13日 下午6:37:11  \* Company www.igeekhome.com  \*  \*/  **public** **class** ThisSuperDemo {  **public** **static** **void** main(String[] args) {  //创建子类对象,测试子类中成员的访问方法  Chinese c = **new** Chinese();    c.testAccess();  }  } |

**父类Person**

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: Person  \* **@Description**: Person类  \* **@date** 2017年11月13日 下午3:36:21  \* Company www.igeekhome.com  \*  \* 父类-人类:  \* 属性：name，age  \*/  **public** **class** Person **extends** Object {  /\*\*  \* **@Fields** name : 姓名  \*/  String name = "Rose";  /\*\*  \* **@Fields** age : 年龄  \*/  **int** age = 16;    /\*\*  \* **@Fields** i : 父类中的测试变量  \*/  **int** i = 300;    /\*\*  \* **@Title**: Person  \*/  **public** Person() {  **super**();  }  /\*\*  \* **@Title**: Person  \* **@param** name  \* **@param** age  \*/  **public** Person(String name, **int** age) {  **super**();  **this**.name = name;  **this**.age = age;  }    /\*\*  \* **@Title**: eat  \* **@Description**: 吃饭的方法  \*/  **public** **void** eat() {  System.***out***.println("吃饭");  }  } |

**子类Chinese**

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: Chinese  \* **@Description**: Chinese子类  \* **@date** 2017年11月13日 下午6:33:43  \* Company www.igeekhome.com  \*  \* 在父类Person的基础上,多了地址属性：  \* address  \*/  **public** **class** Chinese **extends** Person{    /\*\*  \* **@Fields** address : 地址  \*/  String address = "北京";    /\*\*  \* **@Fields** i : 子类中的测试变量  \*/  **int** i = 200;  /\*\*  \* **@Title**: Chinese  \*/  **public** Chinese() {  **super**();  }  //定义三个参数的构造方法,无需了解细节,在super的讲解时会详细介绍  /\*\*  \* **@Title**: Chinese  \* **@param** name  \* **@param** age  \* **@param** address  \*/  **public** Chinese(String name, **int** age,String address) {  **super**(name, age);  **this**.address = address;  }  /\*\*  \* **@Title**: eat  \* **@Description**: 子类吃饭的方法  \* **@see** com.igeek\_01.Person#eat()  \*/  @Override  **public** **void** eat() {  System.***out***.println("子类吃饭");  }    /\*\*  \* **@Title**: testAccess  \* **@Description**: 测试访问成员的方法  \*/  **public** **void** testAccess() {    //访问成员时,在不考虑private修饰的情况下  //本类对象成员用this.调用  //父类成员用super.调用  System.***out***.println(**this**.address);  System.***out***.println(**super**.name);  System.***out***.println(**super**.age);    **this**.eat();  **super**.eat();    **int** i = 100;    System.***out***.println(i);  System.***out***.println(**this**.i);  System.***out***.println(**super**.i);  }  } |

## super访问父类构造方法

### super调用父类构造方法的格式

调用构造方法：

this(其他参数) 可以访问本类其他的构造方法

super(其他参数) 可以访问父类其他的构造方法

默认子类调用父类构造方法：

子类的每个构造方法中均有默认的super(),调用父类的空参构造。手动调用父类构造会覆盖默认的super();

### 案例代码二十二

**测试类ThisSuperConstructorsDemo**

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: ThisSuperConstructorsDemo  \* **@Description**: 继承测试类  \* **@date** 2017年11月13日 下午6:37:11  \* Company www.igeekhome.com  \*  \* super在子类中调用父类的构造方法  \* this在子类中调用子类其他的构造方法  \*  \* 子类的每个构造方法中均有默认的super(),调用父类的空参构造。手动调用父类构造会覆盖默认的super();  \*/  **public** **class** ThisSuperConstructorsDemo {  **public** **static** **void** main(String[] args) {  //测试:子类调用父类的带参构造  Chinese c = **new** Chinese("杰克",18);  System.***out***.println(c.getAge()+"岁的"+c.getName());    }  } |

**父类Person类**

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: Person  \* **@Description**: Person类  \* **@date** 2017年11月13日 下午3:36:21  \* Company www.igeekhome.com  \*  \* 父类-人类:  \* 属性：name，age  \*/  **public** **class** Person **extends** Object {  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name = "Rose";  /\*\*  \* **@Fields** age : 年龄  \*/  **private** **int** age = 16;    /\*\*  \* **@Title**: Person  \*/  **public** Person() {  System.***out***.println("父类空参构造");  }  /\*\*  \* **@Title**: Person  \* **@param** name  \* **@param** age  \*/  **public** Person(String name, **int** age) {  System.***out***.println("父类带参构造");  **this**.name = name;  **this**.age = age;  }    /\*\*  \* **@Title**: eat  \* **@Description**: 吃饭的方法  \*/  **public** **void** eat() {  System.***out***.println("吃饭");  }  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  /\*\*  \* **@return** the age  \*/  **public** **int** getAge() {  **return** age;  }  /\*\*  \* **@param** age the age to set  \*/  **public** **void** setAge(**int** age) {  **this**.age = age;  }  } |

**子类Chinese**

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: Chinese  \* **@Description**: Chinese子类  \* **@date** 2017年11月13日 下午6:33:43  \* Company www.igeekhome.com  \*  \* 子类可以使用super(参数)调用父类的构造方法  \*/  **public** **class** Chinese **extends** Person{    /\*\*  \* **@Title**: Chinese  \*/  **public** Chinese() {  System.***out***.println("子类的空参构造方法");  }  /\*\*  \* **@Title**: Chinese  \* **@param** name  \* **@param** age  \* **@param** address  \*/  **public** Chinese(String name, **int** age) {  **super**(name,age);  System.***out***.println("子类的带参构造");  }  /\*\*  \* **@Title**: eat  \* **@Description**: 子类吃饭的方法  \* **@see** com.igeek\_02.Person#eat()  \*/  @Override  **public** **void** eat() {  System.***out***.println("子类吃饭");  }    } |

## this访问子类构造方法

在子类构造中使用this() 或 this(参数类型 参数值…)的方法可以调用本类中的其他构造方法。但是最终都是要调用父类的构造方法，完成父类成员的初始化。

### 案例代码二十三

**测试类：ThisSuperConstructorsDemo**

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: ThisSuperConstructorsDemo  \* **@Description**: super构造调用测试类  \* **@date** 2018年1月16日 下午7:41:25  \* Company www.igeekhome.com  \*  \* super在子类中调用父类的构造方法  \* this在子类中调用子类其他的构造方法  \* 子类构造方法中,第一行要么是super调用父类构造,要么是this调用子类构造,  \* 其最终原则就是先有父类内容,再有子类内容  \*  \* 子类的每个构造方法中均有默认的super(),调用父类的空参构造。  \* 手动调用父类构造会覆盖super();  \*/  **public** **class** ThisSuperConstructorsDemo {  **public** **static** **void** main(String[] args) {  //测试：子类调用父类的带参构造  Chinese c = **new** Chinese("杰克",18);  System.***out***.println(c.getAge()+"岁的"+c.getName());    System.***out***.println("==================");  //测试:子类调用子类的其他构造  Chinese c2 = **new** Chinese();  System.***out***.println(c2.getAge()+"岁的"+c2.getName());  }  } |

**父类Person类**

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Person  \* **@Description**: Person父类  \* **@date** 2018年1月16日 下午6:17:37  \* Company www.igeekhome.com  \*  \* 父类-人类：  \* 属性：name、age  \*/  **public** **class** Person {  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name;  /\*\*  \* **@Fields** age : 年龄  \*/  **private** **int** age;    /\*\*  \* **@Title**: Person  \*/  **public** Person() {  System.***out***.println("父类空参构造");  }  /\*\*  \* **@Title**: Person  \* **@param** name  \* **@param** age  \*/  **public** Person(String name, **int** age) {  System.***out***.println("父类带参构造");  **this**.name = name;  **this**.age = age;  }    /\*\*  \* **@Title**: eat  \* **@Description**: 吃饭的方法  \*/  **public** **void** eat(){  System.***out***.println("吃饭");  }  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  /\*\*  \* **@return** the age  \*/  **public** **int** getAge() {  **return** age;  }  /\*\*  \* **@param** age the age to set  \*/  **public** **void** setAge(**int** age) {  **this**.age = age;  }    } |

**子类Chinese**

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Chinese  \* **@Description**: Chinese子类  \* **@date** 2018年1月16日 下午6:18:54  \* Company www.igeekhome.com  \*  \* 子类可以使用super(参数)调用父类的构造方法  \*/  **public** **class** Chinese **extends** Person{    /\*\*  \* **@Title**: Chinese  \*/  **public** Chinese() {  //使用this调用本类的两个参数的构造  **this**("默认",20);  System.***out***.println("子类的空参构造方法");  }  /\*\*  \* **@Title**: Chinese  \* **@param** name  \* **@param** age  \* **@param** address  \*/  **public** Chinese(String name, **int** age) {  **super**(name, age);  System.***out***.println("子类的带参构造方法");  }  /\*\*  \* **@Title**: eat  \* **@Description**: 子类吃饭的方法  \* **@see** com.igeek\_02.Person#eat()  \*/  @Override  **public** **void** eat() {  System.***out***.println("子类吃饭");  }    } |

## 一个继承的练习

需求描述:

员工类

员工姓名,员工薪水,员工工作方法

经理类

除了员工属性还多加了奖金

工作方法为打印"管理酒店"

服务员类

属性和员工的属性相同

工作方法为打印"上菜与结账"

厨师类

属性和员工的属性相同

工作方法为打印"做饭"

共有1个经理,2个服务员,1个厨师,(所有人的属性任意定义)

让所有的员工工作

并求所有人的总收入是多少?

### 案例代码二十四

员工类：

|  |
| --- |
| **package** com.igeek\_04;  /\*\*  \* **@ClassName**: Employee  \* **@Description**: 父类员工类  \* **@date** 2017年11月13日 下午7:44:57  \* Company www.igeekhome.com  \*  \* 定义父类员工类:  \* 员工姓名,员工薪水,员工工作方法  \*/  **public** **class** Employee {  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name;  /\*\*  \* **@Fields** salary : 工作方法  \*/  **private** **double** salary;  /\*\*  \* **@Title**: Employee  \*/  **public** Employee() {  **super**();  }  /\*\*  \* **@Title**: Employee  \* **@param** name  \* **@param** salary  \*/  **public** Employee(String name, **double** salary) {  **super**();  **this**.name = name;  **this**.salary = salary;  }  /\*\*  \* **@Title**: work  \* **@Description**: 工作方法  \*/  **public** **void** work() {  System.***out***.println("work");  }  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  /\*\*  \* **@return** the salary  \*/  **public** **double** getSalary() {  **return** salary;  }  /\*\*  \* **@param** salary the salary to set  \*/  **public** **void** setSalary(**double** salary) {  **this**.salary = salary;  }  } |

经理类：

|  |
| --- |
| **package** com.igeek\_04;  /\*\*  \* **@ClassName**: Manager  \* **@Description**: 经理类  \* **@date** 2017年11月13日 下午7:48:38  \* Company www.igeekhome.com  \*  \* 定义子类经理类  \* 经理类,多加了奖金,工作方法为打印"管理酒店"  \*/  **public** **class** Manager **extends** Employee {  /\*\*  \* **@Fields** bonus : 奖金  \*/  **private** **double** bonus;  /\*\*  \* **@Title**: Manager  \*/  **public** Manager() {  **super**();  }  /\*\*  \* **@Title**: Manager  \* **@param** name  \* **@param** salary  \* **@param** bonus  \*/  **public** Manager(String name, **double** salary,**double** bonus) {  **super**(name, salary);  **this**.bonus = bonus;  }  /\*\*  \* **@Title**: work  \* **@Description**: 经理的工作方法  \* **@see** com.igeek\_04.Employee#work()  \*/  @Override  **public** **void** work() {  System.***out***.println("管理酒店");  }  /\*\*  \* **@return** the bonus  \*/  **public** **double** getBonus() {  **return** bonus;  }  /\*\*  \* **@param** bonus the bonus to set  \*/  **public** **void** setBonus(**double** bonus) {  **this**.bonus = bonus;  }  } |

服务员类：

|  |
| --- |
| **package** com.igeek\_04;  /\*\*  \* **@ClassName**: Waiter  \* **@Description**: 服务员类  \* **@date** 2017年11月13日 下午7:51:44  \* Company www.igeekhome.com  \*  \* 服务员类,工作方法为打印"上菜与结账"  \*/  **public** **class** Waiter **extends** Employee {    /\*\*  \* **@Title**: Waiter  \*/  **public** Waiter() {  **super**();  }  /\*\*  \* **@Title**: Waiter  \* **@param** name  \* **@param** salary  \*/  **public** Waiter(String name, **double** salary) {  **super**(name, salary);  }  /\*\*  \* **@Title**: work  \* **@Description**: 服务员的工作方法  \* **@see** com.igeek\_04.Employee#work()  \*/  @Override  **public** **void** work() {  System.***out***.println("上菜与结账");  }  } |

厨师类：

|  |
| --- |
| **package** com.igeek\_04;  /\*\*  \* **@ClassName**: Cook  \* **@Description**: 厨师类  \* **@date** 2017年11月13日 下午7:55:57  \* Company www.igeekhome.com  \*  \* 厨师类,工作方法为打印"做饭"  \*/  **public** **class** Cook **extends** Employee{  /\*\*  \* **@Title**: Cook  \*/  **public** Cook() {  **super**();  }  /\*\*  \* **@Title**: Cook  \* **@param** name  \* **@param** salary  \*/  **public** Cook(String name, **double** salary) {  **super**(name, salary);  }  /\*\*  \* **@Title**: work  \* **@Description**: 厨师的工作方法  \* **@see** com.igeek\_04.Employee#work()  \*/  @Override  **public** **void** work() {  System.***out***.println("做饭");  }  } |

测试类：

|  |
| --- |
| **package** com.igeek\_04;  /\*\*  \* **@ClassName**: DemoTest  \* **@Description**: 测试类  \* **@date** 2017年11月13日 下午7:58:12  \* Company www.igeekhome.com  \*  \* 员工类  \* 员工姓名,员工薪水,员工工作方法  \*  \* 经理类,多加了奖金,工作方法为打印"管理酒店"  \* 服务员类,工作方法为打印"上菜与结账"  \* 厨师类,工作方法为打印"做饭"  \*  \* 共有1个经理,2个服务员,1个厨师,(所有人的属性任意定义)  \* 让所有的员工工作  \* 并求所有人的总收入是多少?  \*/  **public** **class** DemoTest {  **public** **static** **void** main(String[] args) {  //创建四个员工  Manager manager= **new** Manager("Jack", 5000, 2000);  Waiter w1 = **new** Waiter("Rose", 3000);  Waiter w2 = **new** Waiter("Obama", 3000);  Cook c = **new** Cook("Trump", 6000);    //调用让所有的员工工作的方法  *employee\_work*(manager, w1, w2, c);    //调用方法，返回所有人的收入  **double** sum = *getSum*(manager, w1, w2, c);  System.***out***.println("sum:"+sum); }    /\*\*  \* **@Title**: employee\_work  \* **@Description**: 让所有的员工工作的方法  \* **@param** m  \* **@param** w1  \* **@param** w2  \* **@param** c  \*/  **public** **static** **void** employee\_work(Manager m,Waiter w1,Waiter w2,Cook c){  m.work();  w1.work();  w2.work();  c.work();  }    /\*\*  \* **@Title**: getSum  \* **@Description**: 计算所有人的总收入的方法  \* **@param** m  \* **@param** w1  \* **@param** w2  \* **@param** c  \* **@return**  \*/  **public** **static** **double** getSum(Manager m,Waiter w1,Waiter w2,Cook c){  **return** m.getSalary()+m.getBonus()+w1.getSalary()+w2.getSalary()+c.getSalary();  }} |

重点和总结

1、前面基础知识点的回顾

2、基本类型和引用类型的内存解释

4、面向对象中的组合和继承关系

5、父类、子类实现继承操作

6、继承中的方法重写

7、this和super关键字的作用及区别