  
目录

[1](#_Toc466894208)

[Java初级 4](#_Toc466894209)

[一、java基础 4](#_Toc466894210)

[（一）环境搭建 4](#_Toc466894211)

[（二）、基础知识 8](#_Toc466894212)

[（三）、Dos命令 8](#_Toc466894213)

[（四）首个java程序 9](#_Toc466894214)

[（五）、转意字符 11](#_Toc466894215)

[（六）、java注释 11](#_Toc466894216)

[二、标识符变量和常量 12](#_Toc466894217)

[1、数据类型 12](#_Toc466894218)

[2、(标识符)变量的命名规则 14](#_Toc466894219)

[3.常量 16](#_Toc466894220)

[4. 局部变量与实例变量(成员变量) 16](#_Toc466894221)

[三、运算符 18](#_Toc466894222)

[1、赋值运算符 18](#_Toc466894223)

[2、算术运算符 18](#_Toc466894224)

[3、关系运算符 18](#_Toc466894225)

[4、条件运算符（三目运算） 19](#_Toc466894226)

[5、 逻辑运算符 19](#_Toc466894227)

[四、分支语句 20](#_Toc466894228)

[（一）、分支结构（选择结构） 20](#_Toc466894229)

[（二）、循环结构 26](#_Toc466894230)

[五、数组 38](#_Toc466894231)

[1、数组的特性 39](#_Toc466894232)

[2、数组内存分析 39](#_Toc466894233)

[3、数组的声明和使用 40](#_Toc466894234)

[4、数组的常用算法 44](#_Toc466894235)

[5、方法 48](#_Toc466894236)

[六、面向对象之封装 继承 多态 50](#_Toc466894237)

[1、 类和对象 50](#_Toc466894238)

[2、面向对象之——封装 70](#_Toc466894239)

[3、面向对象之——继承 82](#_Toc466894240)

[4、面向对象之——多态 94](#_Toc466894241)

[七、接口 99](#_Toc466894242)

[1、接口的特点： 99](#_Toc466894243)

[2, 比较抽象类与接口 103](#_Toc466894244)

[八、内部类 104](#_Toc466894245)

[（1）、成员内部类 107](#_Toc466894246)

[（2）、静态内部类 109](#_Toc466894247)

[（3）、匿名内部类 🡪就是没有名字的内部类 110](#_Toc466894248)

[（4）、方法中的内部类 112](#_Toc466894249)

[九、异常 113](#_Toc466894250)

[1、为什么需要异常处理 113](#_Toc466894251)

[2、java处理异常的5个关键字 116](#_Toc466894252)

[3、异常的执行情况 118](#_Toc466894253)

[4、常用的异常类型 121](#_Toc466894254)

[5、finally 122](#_Toc466894255)

[6、在catch中有return，程序会不会执行 finally; 123](#_Toc466894256)

[7、多重异常 124](#_Toc466894257)

[8、throws关键字 125](#_Toc466894258)

[9、throw关键字 128](#_Toc466894259)

[10、自定义异常类 130](#_Toc466894260)

[11、小结 131](#_Toc466894261)

[十、常用类与包装类 132](#_Toc466894262)

[1、包装类中的Integer 136](#_Toc466894263)

[2、字符串 139](#_Toc466894264)

[十一、 容器 149](#_Toc466894265)

[（1） 、collection 150](#_Toc466894266)

[（2） 、set 153](#_Toc466894267)

[（3） 、List 164](#_Toc466894268)

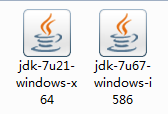
# Java初级

## 一、java基础

### （一）环境搭建

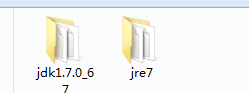
学习Java的首要任务就是进行环境的搭建和环境变量的配置

1. 首先从sun（已被oracle公司收购）公司的官网上下载jdk（根据自己电脑系统下载合适的32或64位的jdk）。

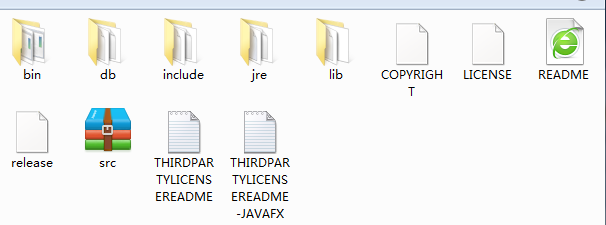


I586为32位 x64为64位

1. 双击jdk应用程序进行安装（因安装简单则不再配图）只需要一步步点击下一步即可，可以更换安装路径
2. 安装完jdk之后效果如图（此图为1.7版本jdk）

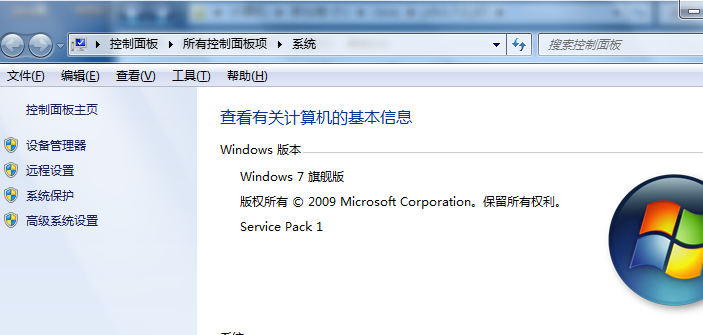


1. 双击进入jdk1.7（图为内部文件）

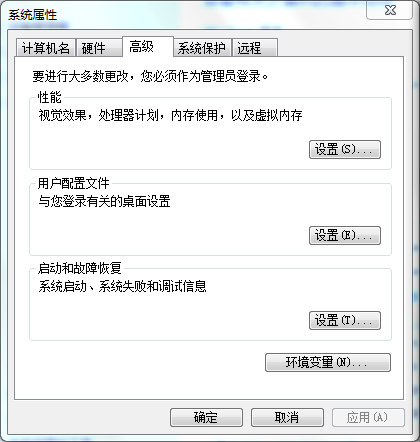


1. 配置环境变量

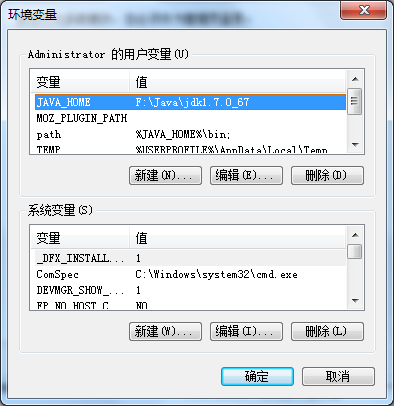
（1）、 计算机右击属性进入下图



（2）、点击高级系统设置 如 下



（3）、点击环境变量



（4）、在用户变量或系统变量内添加两个变量

单击新建按钮，在变量名内添加JAVA\_HOME

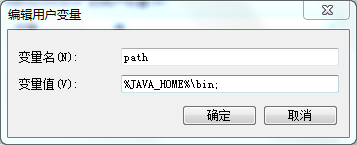
在变量值内填写jdk的安装路径（填写到jdk版本内 如下图）



单击新建按钮，在变量名内添加path变量

在变量值内填写%JAVA\_HOME%\bin;（或jdk路径到bin内 如：F:\Java\jdk1.7.0\_67\bin；）如下图

注意：路径末尾加分号



1. 测试win+r键 （或开始——运行）输入 cmd 进入命令提示窗，在命令提示窗内输入 java –version 显示如下信息则为配置成功。



### （二）、基础知识

1、Java的源文件后缀名为.java

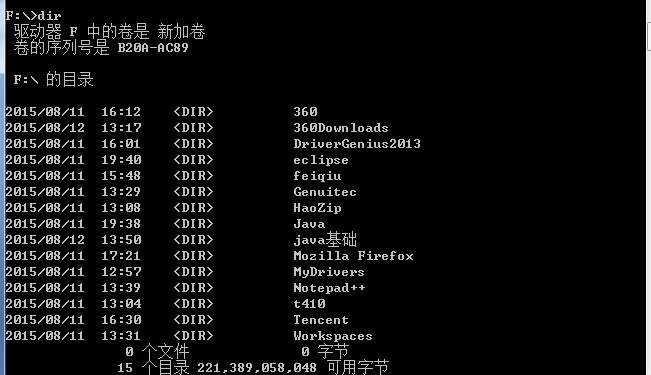
2、Java的编译后的可执行文件的后缀名为.calss

3、Java源文件 可一次编写 随处运行（各个操作系统之间）

### （三）、Dos命令

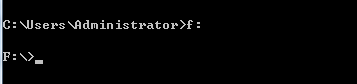
1、Windows键+R:打开运行窗口，输入cmd，进入Dos界面

2、dir:显示文件或文件夹 如图



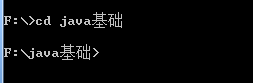
3、cd 文件夹名称:改变当前目录

（1）、切换盘符时：直接输入盘符号冒号



（2）、同盘符内切换路径：若我们想进入到上图的java基础

则使用命令 cd java基础

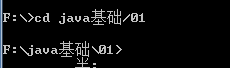


Java 基础内有个01文件夹

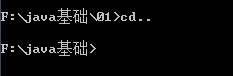
使用命令 cd 01 即可进入

也可从外层目录直接进入内层目录

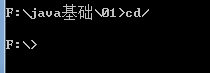
如cd java基础/01



(3)、若想退出当前目录进入上层目录 使用命令 cd..效果如图



(4)、若想直接从深层目录回到根目录 则使用cd/



4、cls:清屏

### （四）首个java程序

|  |
| --- |
| **public class Welcome{** // class 是类的意思，Welcome是类名称，要求，首字母要大写，  //程序的入口  **public static void main(String [] args){//主方法,看到小括号的，就是方法，括号里面的叫形式参数（形参）,static静态的，void的意思是没有的，无返回值, "Hello,World"实际参数（实参） String [] a，叫做String类型的数组，数组的名字叫a**  **System.out.println("Hello World!");**//println方法  **}**  **}** |

1、向控制台输出一句话

|  |
| --- |
| System.out.println("老师说：\"不会找百度，网址如下http:\\\\www.baidu.com\"");  System.out.println(19);  System.out.println("19");  System.out.println(1.5);  System.out.println('a');  System.out.println('a'+1);  System.out.println(1.4f);  System.out.println(1.8d);  System.out.println(1900l); |

Println的小括号中的数据有带双引号的字符串，单引号的字符，带f的单精度，带d的双精度，带l的长整型的数。。。。都可以输出

以上代码实现了方法的**重载**

2、一个.java的文件中可以有N多个class（类）,但**必须只能有一个public的**类.

如果出现多个，效果如下



|  |
| --- |
| public class **Welcome**{ // class 是类的意思，Welcome是类名称，要求，首字母要大写，  }  class Hello{//class 是类,Hello 是类的名称  }  class World{//class 是类,World 是类的名称  } |

**注意事项：**文件的名字**必须与公共**的类名完全相同 (所有的字母，大小写必须完全相同)

Java是对**大小**写非常敏感的

**特点:**一个class（类）编译后对应一个.class的文件

一个class(类)写在了另一个class里，这就叫**内部类**

### （五）、转意字符

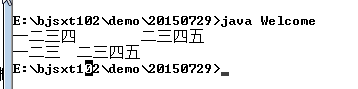
\n:换行

\t:一个制表位（大概七到八个空格），实际的运行时候，取决于前边的字符数，

如果满8个字节，\t就是八个空格,如果前边不满八个，\t被齐八个字节

|  |
| --- |
| System.out.print("一二三四\t二三四五\n");  System.out.print("一二三\t二三四五"); |

运行的效果



### （六）、java注释

**单行注释**(//):通常对一句代码进行解说明

**多行注释**(/\*开头 \*/结尾):方法或类,代码块的前 面，对方法或类进行解释说明，还可以进行代码调试

**文档注释(doc注释)**(/\*\*开头，\*/结尾):方法或类的前边 目的，生成API帮助文档这种注释主要是为支持JDK 工具 javadoc 而采用的。

## 二、标识符变量和常量

### 1、数据类型

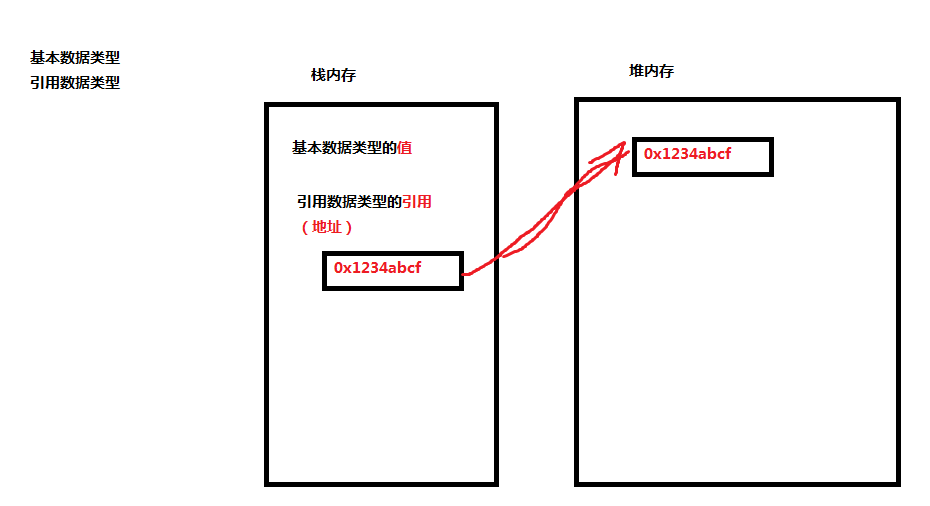
Java 数据类型被分为：基本数据类型和引用数据类型。

|  |
| --- |
| Java 中有 8 中基本数据类型：  类型 位长/b 默认值 取值范围  布尔型 (boolean） 1 false true false  字节型（byte ） 8 0 -128~127  字符型 (char) 16 ‘\u0000’ ‘\u0000’~’\uffff’即 0~65535  短整型(short) 16 0 -32768~32767  整型(int) 32 0 -231~231-1  长整型（long） 64 0 -263~263-1  单精度（float） 32 0.0 +-1.4E-45 或+-3.4028235E+38  双精度（double） 64 0.0 +-4.9E-324 或+-1.797693134862315E+308 |

基本数据类型🡪byte ,short ,int ,long,char,Boolean ,float,double

引用数据类型🡪class(类),接口(interface),数组，枚举

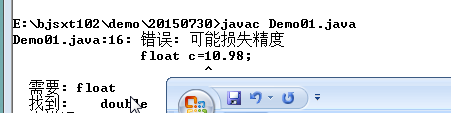
了解数据类型的目的，是告诉内存分配合的空间



每一种基本数据类型都对应一个引用的数据类型(包装类)

Java默认的整型为int 默认的浮点型为double

|  |
| --- |
| float c=10.98;  System.out.println("c="+c); |



正确写法

|  |
| --- |
| float c=10.98**F**; 或小f  System.out.println("c="+c); |

注：String 类型不是基本数据类型，它被定义为类，属于引用数据类型。

由于字符串是常用的数据类型。Java 提供了对 String 类型特殊操作，直接引

用，例如：String s="hello world"；

引用类型是一个对象类型的，它的值是指向内存空间的引用，就是地址，所

指向的内存中保存着变量所表示的一个值或一组值。很好理解吧，因为一个

对象，比如说一个人，不可能是个数字也不可能是个字符啊，所以要想找它

的话只能找它的地址了。

接下来看看基本类型和引用类型变量的不同处理吧。基本类型自然是简单，声明是自然系统就给它空间了。例如，

|  |
| --- |
| int a；  a=250； //声明变量 a的同时，系统给a分配了空间。 |

引用类型就不是了，只给变量分配了引用空间，数据空间没有分配，因为

谁都不知道数据是什么啊，整数，字符？我们看一个错误的例子：

|  |
| --- |
| MyDate today；  today.day = 4； //发生错误，因为 today 对象的数据空间未分配。 |

那我们怎么给它赋值啊？引用类型变量在声明后必须通过实例化开辟数据空间，才能对变量所指向的对象进行访问。举个例子：

MyDate today； //将变量分配一个保存引用的空间

today = new MyDate （） ； //这句话是 2 步， 首先执行 new MyDate （） ，

给 today 变量开辟数据空间，然后再执行赋值操作。

### 2、(标识符)变量的命名规则

(1)可以包含字母，数字，下划线,$

(2)不可以以数字开头

(3)不能是java关键字

**常见错误**(1)非法字符 🡪解决方案，英文字符写成了中文

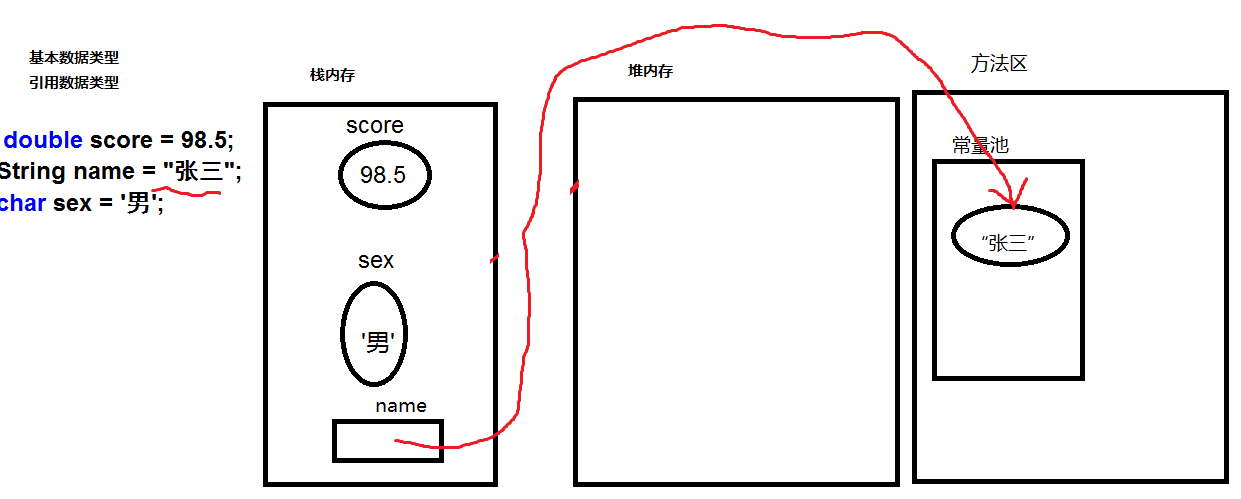
(2)未找到标识符🡪解决方案，变量名称写错，或没有该变量的名字

**变量的使用三步走**

(1)声明变量,🡪通知内存分配空间(根据数据类型) int money;

(2)赋值🡪将等号右边的值赋给左边的变量 money=1000;

(3)使用🡪通过变量的名字来使用这个变量所存储的值 money\*(1+0.05)



#### 1.声明变量

格式：类型 变量名[，变量名]=初值，… ；

赋值：[类型] 变量名=值

如：int a=5 , b=6 , c , d ;

说明：

（1） 变量名必须在作用域中是唯一的， 不同作用域中才允许相同名字的变量出现;

（2） 只要在同一代码块没有同名的变量名， 可以在程序中任何地方定义变量，一个代码块就是两个相对的“{ }”之间部分。

#### 2．变量的使用范围

每个变量的使用范围只在定义它的代码块中，包括这个代码块中包含的代码块。

在类开始处声明的变量是成员变量，作用范围在整个类；

在方法和块中声明的变量是局部变量，作用范围到它的“}”；

#### 3．变量类型的转换

Java 可以将低精度的数字赋值给高精度的数字型变量， 反之则需要强制类型转换。

强制转换格式：（数据类型）数据表达式

字节型 短整型 字符型 整型 长整型 单精度实型 双精度实型

转化规律：由低到高

变量与存储器有着直接关系，定义一个变量就是要编译器分配所需要的内存空间，分配多少空间，这就是根据我们所定义的变量类型所决定的。变量名实际上是代表所分配空间的内存首地址

### 3.常量

Java 中的常量值是用文字串表示的， 它区分为不同的类型， 如整型常量 123，字符常量‘a’， 布尔常量 true、 false 以及字符串常量“This is a constant string”。

Java 的常量用 final 说明，约定常量名一般全部使用大写字母，如果是多个单词组合在一起的，单词之间用下划线连接，常量在程序执行时不可更改。

|  |
| --- |
| 如：final int i=1;  i=i+1; //错，i 是 final（常量），不可更改值  例如：final double IP = 3.14159 D |

说明：默认类型为 64 位 double 双精度类型(D 或 d),数字后面加 F 或 f 则是

32 位 float 单精度(实数)类型

特点:常量的名称，要求全部大写字母,用于与变量区分

### 4. 局部变量与实例变量(成员变量)

1、声明的位置不同:

局部变量:方法里或代码块里

成员变量(实例变量):方法外，类里

2、局部变量与实例变量名称可以相同，局部变量更具有优先级,使用this代表的是实例变量

3、初始值: 局部变量必须赋初始值，否则程序编译出错

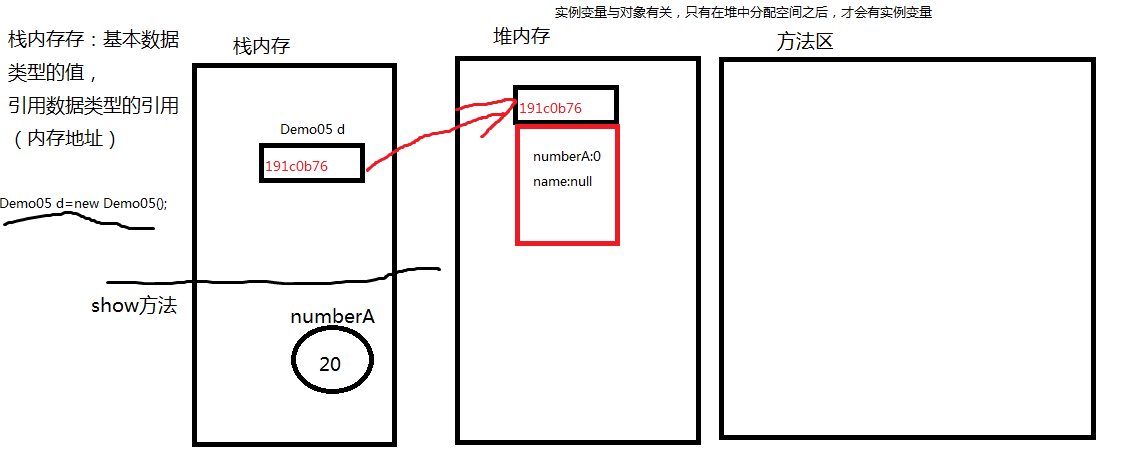
实例变量如果不赋初始值，显示默认值，int类型默认为0,引用数据类型默认为null

|  |
| --- |
| public void show(){  // int numberA=20;//局部变量  // System.out.println(this.numberA);  **int numberA;**//声明  System.out.println(numberA); //使用  } |

运行效果:



4、内存存储 :局部变量存储在栈内存，实例变量与对象有关，存储在堆内存里



## 三、运算符

### 1、赋值运算符

赋值运算符用于把一个数赋予一个变量。赋值运算符两边的类型不一致时，那么如果左侧的数据类型的级别高，则右边的数据将转换成左边的数据类型在赋予左边的变量，否则需要强制类型转换。

赋值运算符包括= 、+=、-=、\*=、%=、/=等。

### 2、算术运算符

算数运算符用于对整型数或者浮点数进行运算，java 语言中的算术运算符包括二元运算符和一元运算符。 所谓的几元运算符即参加运算的操作数的个数。

#### 1） 二元运算符

Java 的二元运算符有+（加）、-（减）、\*（乘）、/（除）、%（取余数）。

#### 2） 一元运算符

Java 的一元运算符有++（自加）、--（自减）

### 3、关系运算符

关系运算符用来比较两个值，返回布尔类型的值 true 或 false。

等于 不等于 小于 小于等于 大于等于 大于

== ！= < <= >= >

### 4、条件运算符（三目运算）

条件运算符的作用是根据表达式的真假决定变量的值。

1> 格式：条件表达式 ? 表达式 2 ：表达式 3

2> 功能：条件表达式为 true，取“表达式 2”值，为 false，取“表达式 3”的值

|  |
| --- |
| 例： int a=10, b=5 , max ;  max=a>b ? a : b ;  10  System.out.println(max) ; // 输出 10 |

### 5、 逻辑运算符

运算符 结果

~ 按位非（NOT）（一元运算）

& 按位与（AND）

| 按位或（OR）

^ 按位异或（XOR）

>> 右移

>>> 右移，左边空出的位以 0 填充 ；无符号右移

<< 左移

&= 按位与赋值

|= 按位或赋值

^= 按位异或赋值

>>= 右移赋值

>>>= 右移赋值，左边空出的位以 0 填充 ；无符号左移

<<= 左移赋值

按位非（NOT）

按位非也叫做补，一元运算符 NOT“~”是对其运算数的每一位取反。例如，数字 42，它的二进制代码为： 00101010经过按位非运算成为 1101010

## 四、分支语句

**程序的结构**

1. 顺序结构:人的成长过程就是顺序结构
2. 分支结构：人生中的Y字路口
3. 循环结构:每天来学校学习

### （一）、分支结构（选择结构）

#### 1、简单的if

|  |
| --- |
| If(条件){  如果条件为真、、、、  }  Else{  如果条件为假、、、、、、  }  2、只有 if 的语句：  If(条件）{  如果条件为真，执行。。。如果为假，不执行  } |

1. 小括号的条件为true,才会执行{}中的代码块
2. 如果{}中只包含一句代码，则{}可省略,如果{}省略，那么默认只有一句代码从属于if为true的时候去执行

|  |
| --- |
| int number=(int)(6\*Math.random());//产生0-5之间数  if(number>3)//使用关系运算符产生的比较之后的结果，这个结果一定是boolean类型  System.out.println("哈哈，我是"+number);  System.out.println("我比3大");    System.out.println("程序结束");  if(true){  System.out.println("嘿嘿");  }  boolean isTrue=false;  if(!isTrue){  System.out.println("嘻嘻 1");  }  if(isTrue==false){  System.out.println("嘻嘻 2");  }  if(isTrue=true){  System.out.println("嘻嘻 3");  }  } |

**注意事项:**

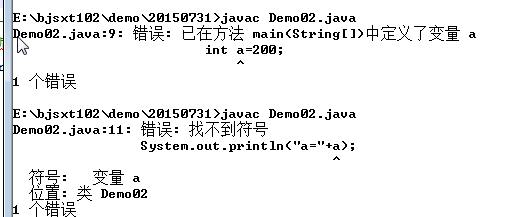
If的小括号中的条件，可以千变万化，但必须是boolean，通常这个boolean类型由关系运算符，进行比较之后的结果

|  |
| --- |
| **int a=300;**  int number=(int)(6\*Math.random());//产生0-5之间数  if(number>3){//使用关系运算符产生的比较之后的结果，这个结果一定是boolean类型  System.out.println("哈哈，我是"+number);  System.out.println("我比3大");  **int a=200;**  }  **System.out.println("a="+a);** |

将产生两个问题:

(1)如果没有声明int a=300;之前，问题是a的作用域是if的{}范围 内，出了{}找不到符号

(2)结果声明了int a=300,产生的问题是，在同一个法中有两个同名的局部变量

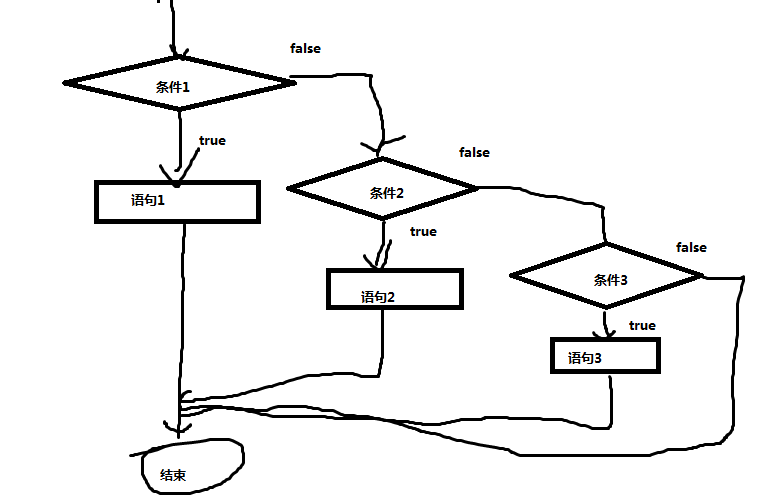


#### 2、多重if

多重if用于解决区间段的问题，先执行第一个判断，如果为true，后续所有的判断都不再进行，如果第一个为false,继续判断第二个，。。。。。。如果所的判断都不成立，将执行else部分,

Else不是必须的，可以省略

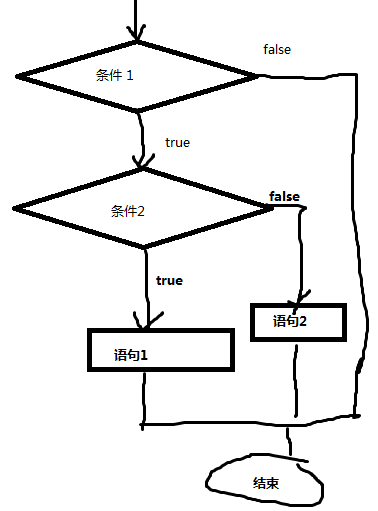
流程图



#### 3、嵌套if

第一个条件成立，才有机会判断第二个

流程图



Equals:方法

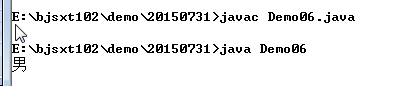
|  |
| --- |
| String sex=null;  if(**sex.equals("女")**){  System.out.println("女");  }else{  System.out.println("男");  } |

运行效果:



|  |
| --- |
| String sex=null;  if(**"女".equals(sex**)){  System.out.println("女");  }else{  System.out.println("男");  } |

运行效果:



#### 4、Switch

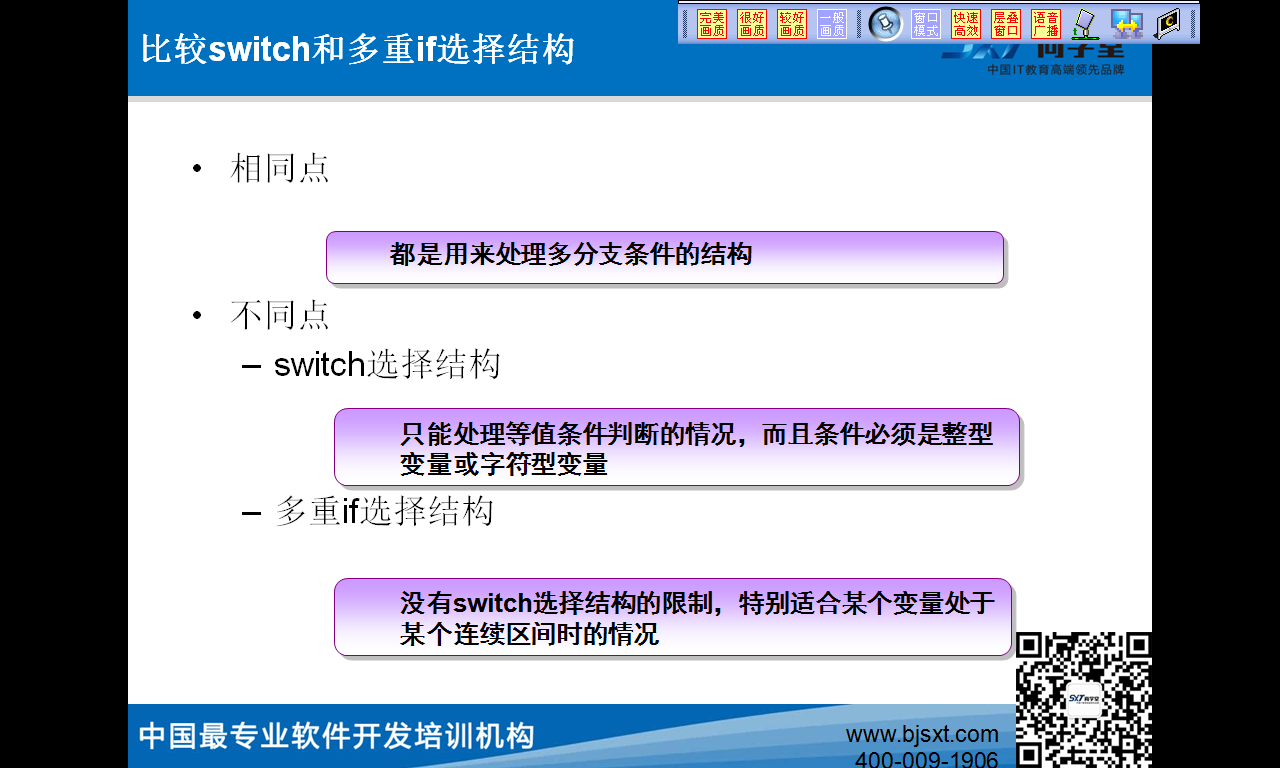
switch 语句是多分枝语句，基本语法：

|  |
| --- |
| Switch(expr){  Case value1:  Statements;  Break;  ........  Case valueN:  Statements;  Break;  Default:  Statements;  Break;  } |

4.Switch 语句中只能有一个 default 子句。

注意事项:

1. case后的常量值不能重复
2. break的作者是中断switch中代码的贯穿
3. default的位置可放前也放后，程序永远先从case执行，所有的case均不满足条件，最后去执行default
4. switch中的()里的表达式，必须是int或char，（JDK版本1.7之后，可以使用String）



### （二）、循环结构

#### 1、为什么使用循环？循环解决的问题?

循环可以解决代码中重复性的操作问题

#### 2、While与if的区别

If:做条件判断，条件成立的情况下，执行一次

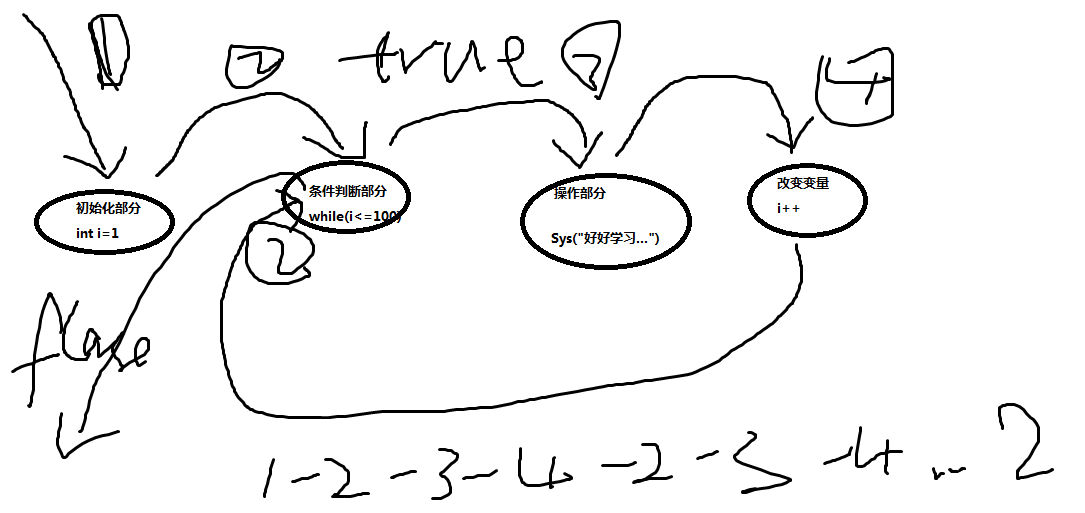
While:做条件判断，条件成立的情况下，一直执行，直到条件为false，执行结束

#### 3、while循环

|  |
| --- |
| public static void main(String [] args){  **int i=1;**//[1]初始化部分  while**(i<=100**){//[2]条件判断  System.out.println("第"+i+"遍，好好学习，天天向上");//[3]操作  **i++**;// [4]改变变量(改变循环条件)  }  System.out.println("结束了");    } |

**初始化什么，就判断什么，判断什么，就改变什么**

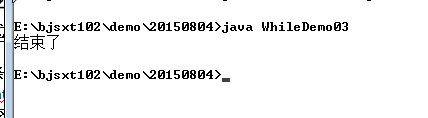
**循环的执行顺序**



**先判断，再执行，条件不成立，一次也不执行**

|  |
| --- |
| **int i=1000;//[1]初始化部分**  **while(i<=100){//[2]条件判断**  **System.out.println("第"+i+"遍，好好学习，天天向上");//[3]操作**  **i++;// [4]改变变量(改变循环条件)**  **}**  **System.out.println("结束了");** |

**运行结果**



**什么情况下会出现死循环**

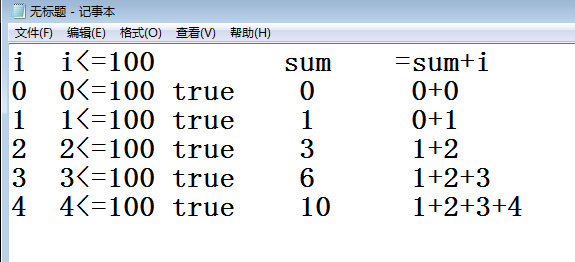
|  |
| --- |
| public static void main(String [] args){  **int i=1;**//[1]初始化部分  while**(i<=100**){//[2]条件判断  System.out.println("第"+i+"遍，好好学习，天天向上");//[3]操作  }  System.out.println("结束了");  } |

**条件判断永远成立,没有出口**

示例代码 （**0-100之间的累加求和**）

|  |
| --- |
| public class WhileTest {  public static void main(String[] args) {  **int i = 0;**  int sum = 0;  while (**i <= 100**) {  sum += i; //sum = sum+i;  **i++;**  }  System.out.println("Sum= " + sum);  }  } |

**分步演示**



**While通常用于解决循环中的次数不固定的情况的问题**

**代码举例**

|  |
| --- |
| import java.util.\*;  public class WhileDemo06{  public static void main(String [] args){  Scanner input=new Scanner(System.in);  **String answer="y";//[1]初始化部分**  **while("y".equals(answer)){//[2]操作**  **//[3]操作部分**  System.out.println("游戏开始啦！");  System.out.println("你拍一，我拍一，一个小孩织毛衣...... ！");  System.out.println("游戏结束啦 ！");    **//[4]改变部分**  **System.out.println("继续吗?");**  **answer=input.next();**  }  System.out.println("over");  }  } |

#### 4、Do-while

Do-while: 🡪 与while一样，通常用来解决循环中次数不固定的情况

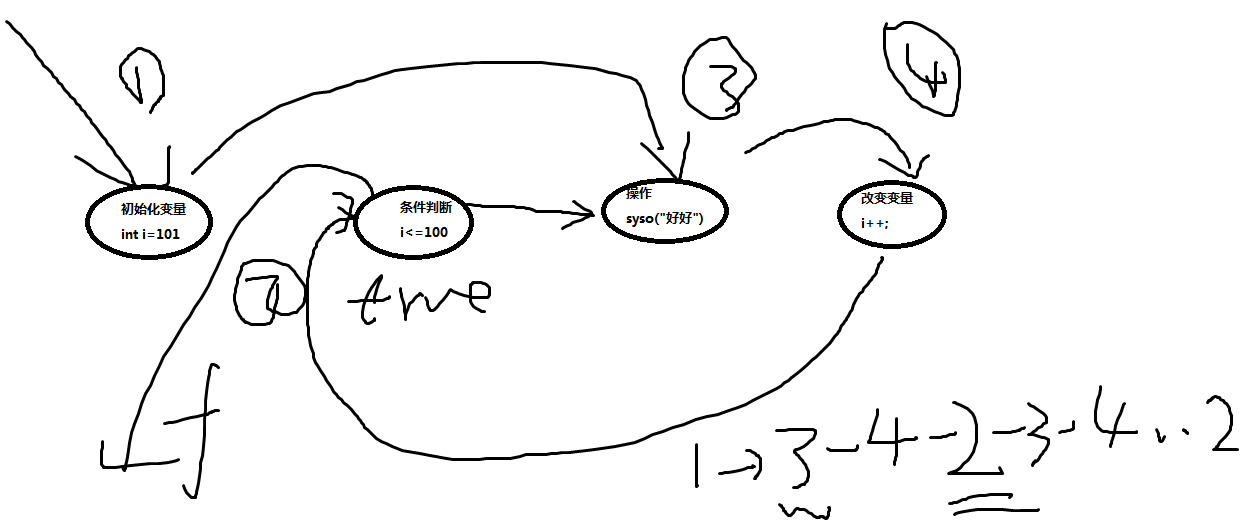
**语法**

**Do{**

**}while(条件);**

|  |
| --- |
| **public static void main(String [] args){**  **int i=101;//[1]初始化部分**  **do{**  **System.out.println("第"+i+"遍，好好学习，天天向上");//[3]操作**  **i++;// [4]改变变量(改变循环条件)**  **}while(i<=100);//[2]条件判断**  **System.out.println("结束了");**  **}** |

**执行顺序**



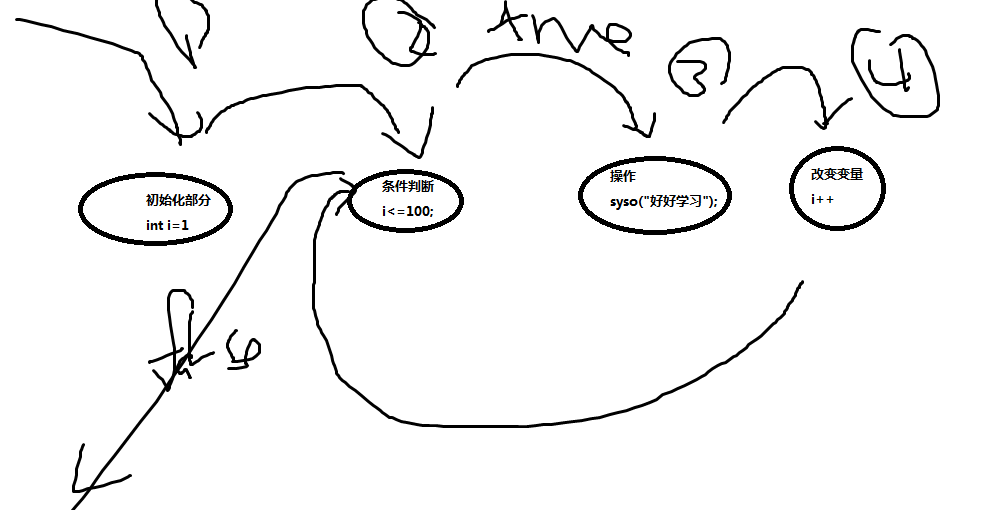
##### While与do-while的区别

1. 语法
2. 当条件不成立的时候，do-while至少执行一次,while一次也不执行
3. 当条件成立的时候，可以互换使用
4. 通常都用来解决循环中的次数固定的情况。推荐使用while, do-while了解即可

#### 5、For循环

主要解决的是循环中的次数固定的情况,当次数固定时，首先 for循环

执行顺序



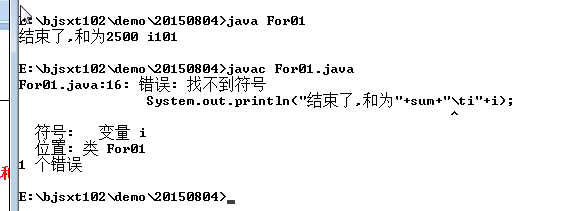
1🡪2🡪3🡪4🡪2…..2

从执行顺序上来说，for是while的变形，条件不成立时，循环一次都不执行

**常见问题 1 （变量的作用域的问题）**

|  |
| --- |
| **/\*int i=1;//计数器**  **int sum=0;**  **while(i<=100){**  **//System.out.println("第"+i+"遍，好好学习，天天向上");**  **sum+=i;**  **i+=2;// 计数器累加**  **}**  **System.out.println("结束了,和为"+sum+"\ti"+i);**  **\*/**  **int sum=0;**  **for(int i=1;i<=100;i=i+2){**  **sum+=i;**  **}**  **System.out.println("结束了,和为"+sum+"\ti"+i);** |

**运行效果**



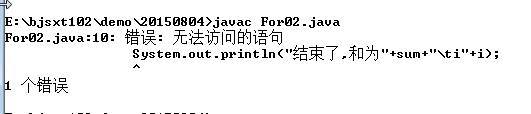
**While得到i的值是101**

**For报错，找不到符号，因为i在for中声明的，出了for的}，变量就消失了.**

**常见问题2 (死循环)**

|  |
| --- |
| **int sum=0;**  **//int i=1;//初始化部分**  **int i;//声明变量**  **for(i=1;;i=i+2){//i=1初始化**  **sum+=i;**  **}**  **System.out.println("结束了,和为"+sum+"\ti"+i);** |

**运行效果:**

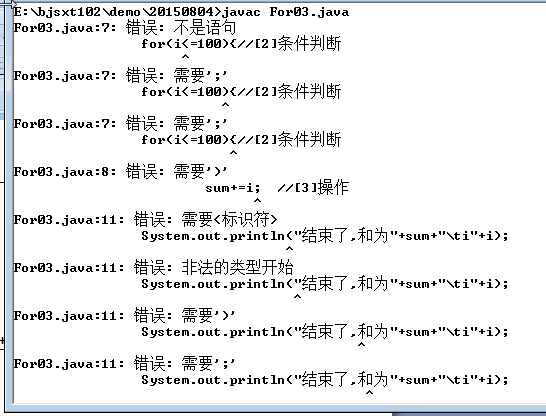


**原因是没有条件，for无条件执行，死循环，永远没有机会退出for循环，所以syso()没有机会去执行**

**常见问题3**

|  |
| --- |
| **int sum=0;**  **//int i=1;//初始化部分**  **int i=1;//[1]初始化部分**  **for(i<=100){//[2]条件判断**  **sum+=i; //[3]操作**  **i=i+2;//[4]改变**  **}**  **System.out.println("结束了,和为"+sum+"\ti"+i);** |

**运行效果:**



**在for循环中，初始化部分可以放到for外,改变部分也可以放到for的}之前,但是**

**For中的 ;号不能省**

|  |
| --- |
| **For( ; ;){**  **}** |

**经验总结**

**100以内的3的倍 数之和**

|  |
| --- |
| **int sum=0;**  **//for(int i=0;i<=100&&i%3==0;i++){**  **for(int i=0;i<=100;i++){**  **if(i%3==0){**  **sum+=i;**  **}**  **}**  **System.out.println(sum);** |

**For当中的条件，只做循环次数的判断，如果还有其它附加条件，统统 由if来完成**

#### 6、流程控制语句

**Break;中断**

1. 用在switch中，作用，停止贯穿
2. 用在循环里，作用，退出循环结构,通常与if一起使用

**Continue;继续**

只能用在物质循环里，作用，跳出本次循环，进入下一次循环

在while,do-while中continue跳到条件判断处，在for中跳到i++(迭代处\改变处)

|  |
| --- |
| public static void main(String [] args){  //求偶数之和    **int sum=0;**  **int i=0;**  **do{**  **if(i%2!=0){**  **continue;**  **}**  **sum+=i;**  **i++;**  **}while(i<=10);**  **/\*for(int i=0;i<=10;i++){**  **if(i%2!=0){**  **continue;**  **}**  **sum+=i;**  **}\*/**  **//int i=0;//初始化**  **/\*while(i<=10){**  **if(i%2!=0){**  **continue;**  **}**  **sum+=i;**  **i++;**    **}\*/**  System.out.println("和为:"+sum);  } |

**Return:返回**

**Foreach循环：🡪通常用来遍历数组或集合**

|  |
| --- |
| **//**  String str="helloWorld";  char[] ch=str.toCharArray() ;// 将字符串转为 char类型的数组  **for(char c:ch){**//每次从ch这个char类型的数组中取出一个字符，赋值给char类型的变量c  System.out.println(c);  **}** |

**本章总结**

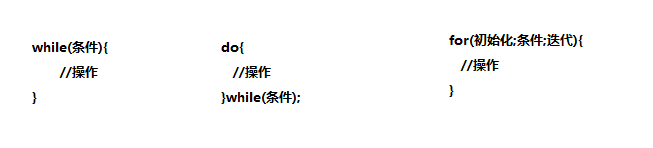
**【1】**while,do-while:次数不固定，继续吗？

for:次数固定

【2】while,for,条件不成立，一次也不执行，都是先判断，再执行

do-while:先执行，再判断，条件不成立，至少执行一次

【3】语法结构：



【4】执行顺序

While,for: 1🡪2🡪3🡪4🡪2🡪3🡪4…..2不成立，从2处退出

Do-while : 1🡪3🡪2🡪4🡪2🡪3🡪4…..2不成立，从2处退出

#### 二重循环

一个循环体中包含另外一个完成的循环结构

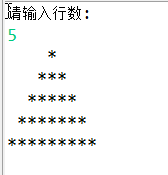
各种循环可以任意去嵌套

**特点：**外层循环执行一次，内层循环执行完成的一轮

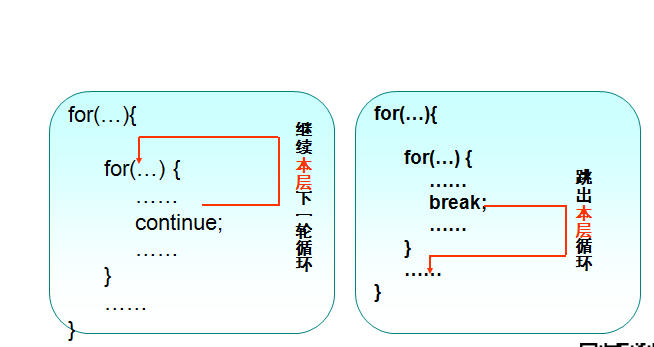
**示例:**打印各种三角形(**外层控制行数，内层控制列数**)

|  |
| --- |
| Scanner input=**new** Scanner(System.*in*);  System.*out*.println("请输入行数:");  **int** row=input.nextInt();  **for**(**int** i=1;i<=row;i++){ //行数  //空格  **for**(**int** j=row;j>i;j--){  System.*out*.print(" ");  }  //\*  **for**(**int** j=1;j<=2\*i-1;j++){  System.*out*.print("\*");  }  System.*out*.println();  } |

**运行效果图**



#### 8、二重循环中的break,和continue



## 五、数组

数组是有序数据的集合，数组中的每个元素具有相同的数组名，根据数组名

和下标来唯一确定数组中的元素。使用时要先声明后创建

### 1、数组的特性

（1）、存储的数据类型是相同的

（2）、有序的集合（容器）

（3）、 一旦声明，长度不能更改

（4）、 通过下标（索引）来访问数据

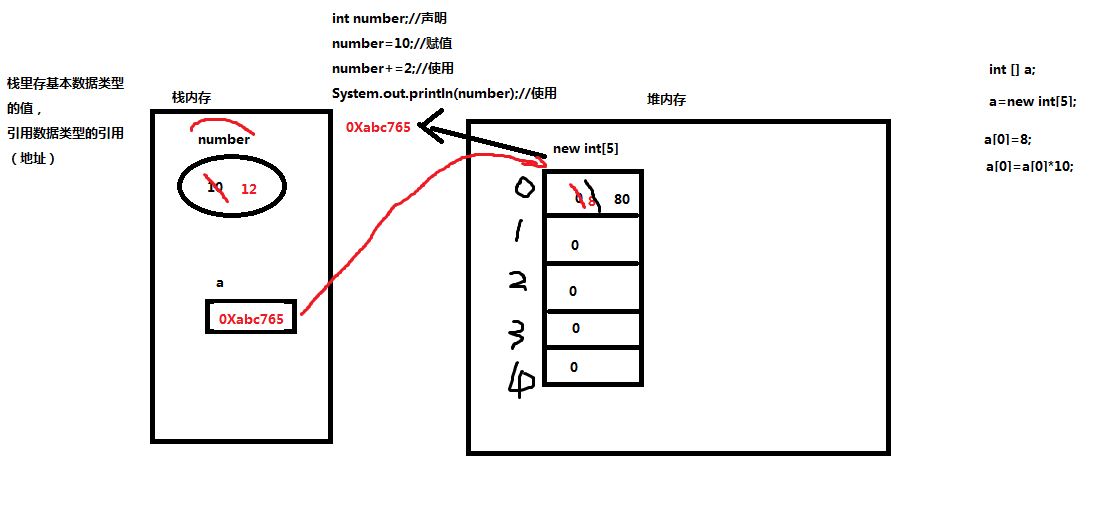
数组存储的数据类型为:（基本数据类型，引用数据类型）

数组是引用数据类型

变量：向内存划一个空间，存一个变量的值

数组: 划一连串空间，存储N多个值

### 2、数组内存分析



### 3、数组的声明和使用

#### 1） 一维数组的声明

格式： 数据类型 数组名[ ] 或 数据类型 [ ]数组名

例： int a[] ; String s[] ; char []c ;

说明： 定义数组， 并不为数据元素分配内存， 因此“[ ]”中不用指出数组中元素

个数。

#### 2 ）一维数组的创建与赋值

创建数组并不是定义数组，而是在数组定义后，为数组分配存储空间，同时

对数组元素进行初始化

（1）用运算符 new 分配内存再赋值

格式：数组名=new 数据类型[size]

|  |
| --- |
| 例：int a[] ;  a=new int[3] ; // 产生 a[0] , a[1] , a[2] 三个元素  a[0]=8 ; a[1]=8 ; a[2]=8 ; |

#### 3）直接赋初值并定义数组的大小

|  |
| --- |
| 例：int i[]={4,5,010,7,3,2,9} ;  String names[]={“张三”,”李四”,”王五”,”宋七”} ; |

#### 4）测试数组长度（补充）

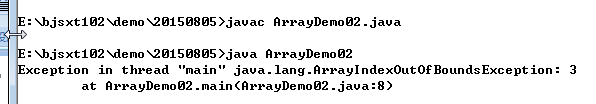
格式：数组名.length

|  |
| --- |
| char c[]={‘a’,’b’,’c’,’北’,’京’} ;  System.out.print(c.length) ; // 输出 5 |

#### 5）常见问题

**(数据下标越界)**

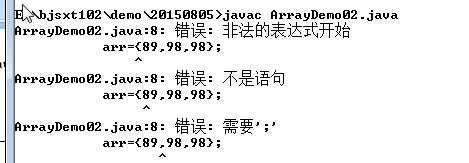
|  |
| --- |
| int [] score=new int[3];//声明并分配空间  score[0]=89;  score[1]=90;  score[2]=67;  score[3]=76; |



常见问题—》静态赋值，不能分两行写(因为没有写new)

|  |
| --- |
| **int [] arr;//声明**  **arr={89,98,98};** |

**运行效果图:**



**下面的写法, 就可以**

|  |
| --- |
| int [] array;  array=new int[]{89,90,67,89,89,78,47,38,8,876,90}; |

动态赋值🡪索引是变化的

**For中的i代表的就是数组中的下标（索引）**

|  |
| --- |
| **int** [] score=**new** **int**[5];  **for** (**int** i = 0; i < score.length; i++) {  System.*out*.println("请输入第一个学员的成绩");  score[i]=input.nextInt();  } |

数组的遍历 🡪通常会使用到foreach

|  |
| --- |
| **for** (**int** i = 0; i < score.length; i++) {  sum+=score[i];  }  System.*out*.println("for和:"+sum);  sum=0;  System.*out*.println("======================");    **for**(**int** s:score){  sum+=s;  } |

**什么情况下使用for,使用情况下使用加强for(foreach)**

在使用数组中的数组时，如果不涉及到下标（索引）的情况下，使用foreach语法更简洁

#### 6）多维数组

以二维数组为例

例：int d[][] ; // 定义一个二维数组

d=new int[3][4] ; // 分配 3 行 4 列数组内存

int a[][]=new int[2][] ;

a[0]=new int[3] ; // 第二维第一个元素指向 3 个整型数

a[1]=new int[5] ; // 第二维第一个元素指向 5 个整型数

注意：Java 可以第二维不等长

int i[][]={{0},{1,4,5},{75,6},{8,50,4,7}} ; //定义和赋初值在一起

下面数组定义正误的判断

int a[][]=new int[10,10] //错

int a[10][10]=new int[][] //错

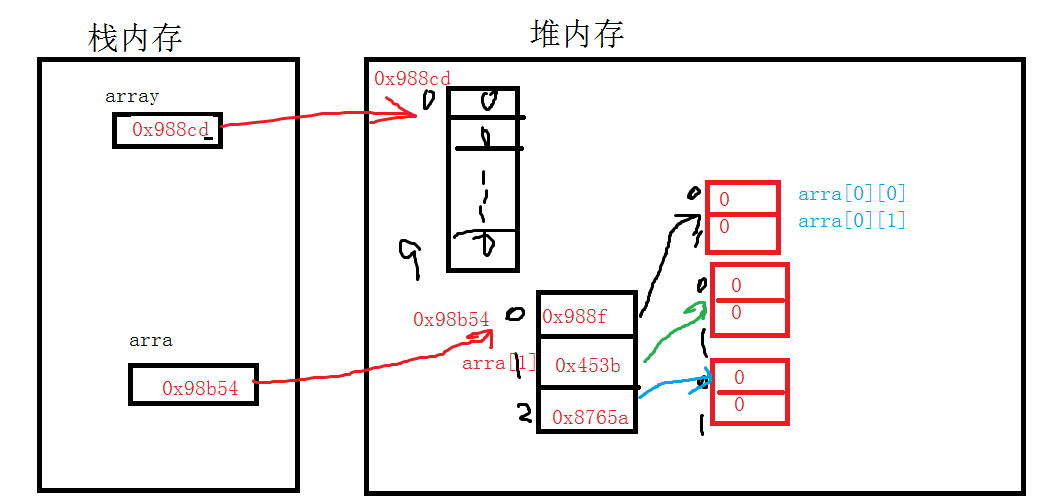
int a[][]=new int[10][10] //对

int []a[]=new int[10][10] //对

int [][]a=new int[10][10] //对

注意：java 中二维数组分配空间是第二维可以为空，但是第一维必须分配内存。

**二维数组是数组的数组**



遍历的时候使用双层for循环

外层控制行数，内层控制列数

|  |
| --- |
| for(int i=0;i<arra.length;i++){//控制的行数  for(int j=0;j<;j++){//列数  System.out.print(arra[i][j]+"\t");  }  System.out.println();//换行  } |

二维数组的列数可以任意，

|  |
| --- |
| int [] oldArray={12,34,89,90,100};  int [][] arr=new int [3][]; //行和列  arr[0]=oldArray;  arr[1]=new int[]{9,98,9};  arr[2]=new int[4];  for(int i=0;i<arr.length;i++){  for(int j=0;j<arr[i].length;j++){  System.out.print(arr[i][j]+"\t");  }  System.out.println();//换行  } |

二维数组的赋值形式

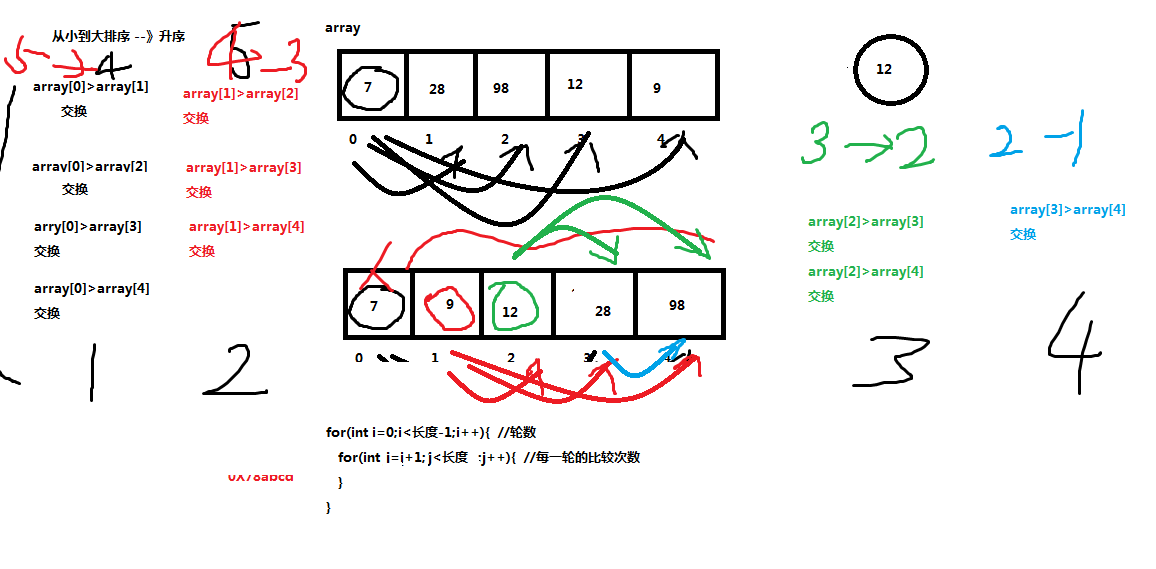
|  |
| --- |
| int [] [] arrc={{1,2,3},{90,98},{87,87,9,0}}; |

### 4、数组的常用算法

#### (1)插入算法(红色的为操作步骤)

|  |
| --- |
| import java.util.Scanner;  public class ArrayDemo03{  public static void main(String [] args){  //插入算法  **//【1】两个数组，新数组的长度是原数的长度+1，原数组一定有顺序**  int [] oldArray={21,56,87,89,90,100};//原数组  int []newArray=new int [oldArray.length+1];//新数组  //将原数组与新数组打印输出一次  System.out.println("原数组");  for(int n:oldArray){  System.out.print(n+"\t");  }    **//【2】将原数组的值赋给新数组中对应位置，以原数组为准**  for(int i=0;i<oldArray.length;i++){  newArray[i]=oldArray[i];  }    **//【3】准备要插入的数据**  Scanner input=new Scanner(System.in);  System.out.println("\n 请输入要插入的整数:");  int number=input.nextInt();  int index=newArray.length-1;//用来记录第一个比这个数大的下标  for(int i=0;i<newArray.length;i++){  if(newArray[i]>number){  index=i;//记录一下位置  break;  }  }  System.out.println("index="+index);  **//【4】移动位置**  for(int i=newArray.length-1;i>index;i--){  newArray[i]=newArray[i-1];  }  **//【5】将新值插入**  newArray[index]=number;    System.out.println("\n插入后的新数组");  for(int n:newArray){  System.out.print(n+"\t");  }    }  } |

#### (2)选择排序



|  |
| --- |
| **int** [] array={12,28,87,78,9};  System.*out*.println("之前");  System.*out*.println(Arrays.*toString*(array));  **for**(**int** i=0;i<array.length-1;i++){  **for**(**int** j=i+1;j<array.length ;j++){  **if**(**array[i]>array[j]**){  **int** temp=array[i];  array[i]=array[j];  array[j]=temp;  }  }  }  System.*out*.println("之后");  System.*out*.println(Arrays.*toString*(array)); |

**依次拿array[i]的值与array[ j]进行比较**

**Arrays.toString(数组名);将数组的内容转为字符串进地输出，使用，号分隔，放到[ ]中**

#### (3)冒泡排序

口决：

**N个数字来排队，两两相比，小靠前，外层循环N-1，内层循环N-1-i;**

|  |
| --- |
| int [] array={12,28,87,78,9};  System.out.println("之前");  System.out.println(Arrays.toString(array));  **for(int i=0;i<array.length-1;i++){**  **for(int j=0;j<array.length-1-i;j++){**  if(**array[j]>array[j+1]**){  int temp=array[j];  array[j]=array[j+1];  array[j+1]=temp;  }  **}**  **}**  System.out.println("之后");  System.out.println(Arrays.toString(array)); |

### 5、方法

在写代码的过程中，往往会用到某一段代码，这段代码经常使用到，这个时候就可以把这段代码提取出来，编写一个方法

**方法：就是一段代码，用来完成一个独立的功能，不能确定的元素，定义成方法的参数**

|  |
| --- |
| import java.util.Arrays;  public class ArrayDemo07{  /\*\*  \*/  //使用方法来解决，方法就是一段代码，这段代码完成一个独立的功能，使用的时候就调用一下  /\*\*  用于将两个数进行交换,不确定的变量使用参数来完成，  (1)用于需要排序的数组，  (2)第二个参数是下标  （3）第三个参数还是下标  \*/  **public static void change(int [] arr,int a,int b){**  int temp=arr[a];  arr[a]=arr[b];  arr[b]=temp;  **}**  //选择排序  **public static void select(int [] array){**  for(int i=0;i<array.length-1;i++){  for(int j=i+1;j<array.length ;j++){  if(array[i]>array[j]){  change(array,i,j);//方法调用  }  }  }  **}**  //冒泡排序  **public static void maoPao(int [] array){**  for(int i=0;i<array.length-1;i++){  for(int j=0;j<array.length-1-i;j++){  if(array[j]<array[j+1]){  change(array,j,j+1);//调用方法  }  }  }  **}**  **public static void main(String [] args){**    int [] array={12,28,87,78,9}; //第一个数组  int [] b=new int[10];//第二个数组  for(int i=0;i<b.length;i++){  b[i]=(int)(Math.random()\*10)+1;  }  int [] c={23,43,23};//第三个数组  //想排序  **ArrayDemo07.select(array);//选 择**  **ArrayDemo07.maoPao(b);//冒泡**  **Arrays.sort(c);**  System.out.println(Arrays.toString(array));  System.out.println(Arrays.toString(b));  System.out.println(Arrays.toString(c));      }  **}** |

因为方法被**static**所修饰，所以可以使用**类名.方法名()**的形式进行调用

## 六、面向对象之封装 继承 多态

Java 是一种面向对象的语言，这是大家都知道的，他与那些像 c 语言等面向过程语言不同的是它本身所具有的面向对象的特性——封装，继承，多态，这也就是传说中的面向对象三大特性。

### 类和对象

类：指的是一类事物

对象：这类事物中具体的某一个

类： 对象的蓝图， 生成对象的模板， 是对一类事物的描述， 是抽象的概念上的定义

对象：对象是实际存在的该类事物的每个个体，因而也称为实例

类之间的三种关系： 依赖关系 （uses-a） 聚集关系 （has-a） 继承关系 （is-a）

在 java 中，类和对象的关系就像是动物和老虎的关系一样，老虎属于动物，

老虎只是动物的一个实例。

类表示了对象的类别，是创建对象的蓝本。建立一个事物的抽象模型，本质上就是表达该事物的性质和行为。使用类来建立抽象模型，是通过在类中定义变量和方法来实现的。

类中定义的属性是一种可存储值的变量，该变量代表着事物的具体性质。类的对象所具有的行是由定义的方法来实现的。类中定义的变量和方法被称为类的成员。

对象是类的实例。对象在构造时以类为蓝本，创建对象的过程叫做实例化。对象是类所表示的抽象事物的具体实例。

面向对象：

1：将复杂的事情简单化。

2：面向对象将以前的过程中的执行者，变成了指挥者。

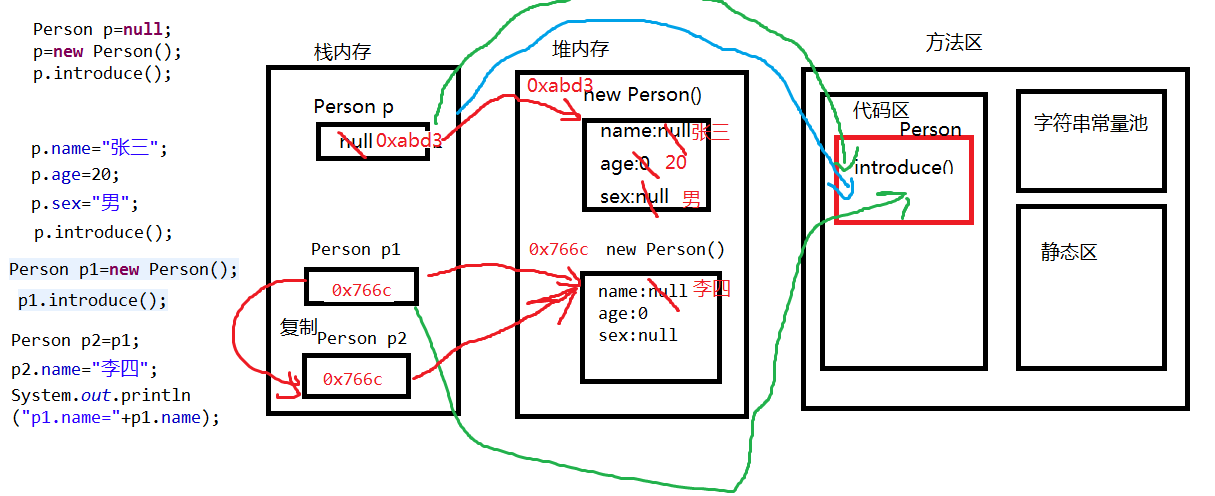
3：面向对象这种思想是符合现在人们思考习惯的一种思想。

#### （1）、定义类的步骤

|  |
| --- |
| **public** **class** Person {  **//[1]给类起名**  **//[2]类的属性**  String name;//姓名  **int** age;//年龄  String sex;//性别    **//[3]方法**  **public** **void** introduce(){  System.*out*.println("name="+name+"\tage="+age+"\tsex="+sex);    }  } |

#### （2）、创建对象的过程

内存分析图



**总结**：每个对象的属性是**独立**的，方法是**共享**的

编写类和创建对象的完整代码

|  |
| --- |
| public class Student{**//[1]类，模板**  **//[2] 成员变量，或实例变量**  String name;  int age;  String school;  String hobby;  **//[3]**  **//成员方法，或实例方法，由对象来调用**  **public void show(){**  System.out.println(name);  System.out.println("年龄:"+age);  System.out.println("就读于:"+school);  System.out.println("爱好:"+hobby);  }  //加static的方法，叫类方法，由类来调用  //主方法带static  **public static void main(String [] args){**  **//[1]创建对象 类名 对象名=new 类名();**  Student stu=new Student();  **//[2]给属性赋值**  stu.name="张浩";  stu.age=23;  stu.school="尚学堂102";  stu.hobby="看书";  **//[3]调用方法**  stu.show();  **}**  } |

#### （3）、深入类的方法

(1)如果这个方法前加static，静态方法，归类所有，与对象无关，类名.方法名()

Math.random(), Arrays.sort()…..

(2)方法前没有加static,实例方法，成员方法，归对象所有。

方法的类型:

1. 无参无返回
2. 无参有返回
3. 有参无返回
4. 有参有返回

##### （1）、构造方法

**概念：**

1. 方法的名称与类的名称相同
2. 没有任何返回值类型，也不可以写void
3. 构造方法可以重载

**构造方法的作用**

用于创建对象, new …的时候使用

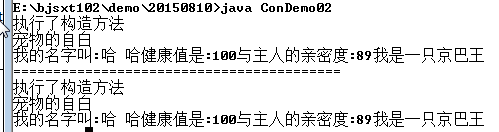
**无参构造方法，目的是在创建对象的时候给属性赋值**

|  |
| --- |
| //构造方法  **public Dog1(){**  name="哈 哈";  health=100;  love=89;  strain="京巴王";  System.out.println("执行了构造方法");  **}** |

**测试**

|  |
| --- |
| public class ConDemo02{  public static void main(String [] args){  //创建对象  **Dog1 d=new Dog1();** //带()叫方法,构造方法,  d.print();//调用方法  System.out.println("=========================================");  **Dog1 d2=new Dog1();**  d2.print();//调用方法  }  } |

**运行效果:**



**发现的问题：**创建了N多个对象，每个对象的属性值都相同，这个构造方法，不够通用，这个时候，可以使用带参的构造方法

**带参方法**

|  |
| --- |
| public Dog1(String name,int health,int love,String strain){  name=name; //**想将方法的局部变量赋给成员变量**  health=health;  love=love;  strain=strain;  } |

**测试**

|  |
| --- |
| public class ConDemo03{  public static void main(String [] args){  //创建对象  **Dog1 d=new Dog1("美美",90,10,"雪娜瑞");** //调用带参的方法的注意事项，方法参数的类型，个数，顺序必须与方法定义完全相同  d.print();//调用方法    }  } |

**运行效果:**



**原因**:局部变量与成员变量名称相同，采取的是就近原则，

name=name; 将相当于，当局部变量赋给局部变量,

而成员变量的值没有，依然采用的是默认值

可以使用**this关键字**来解决这个问题

**当局部变量与成员变量名称相同时，使用this来代表成员变量,**

**如果名称不同，this可以省略**

**修改后的代码**

|  |
| --- |
| public Dog1(String name,int health,**int qinmi,**String strain){  **this.name=name;** //将局部变量的值赋给了成员变量s  this.health=health;  **love=qinmi;**  this.strain=strain;  } |

**构造方法的特点：**

如果没有人为的去编写构造方法，那么系统默认提供一个无参的构造方法，用于创建对象

一旦手动去编写了构造方法之后，那么无参的构造方法，系统将收回，不再分配

,这个时候去调用无参构造方法创建对象，程序将编译失败

|  |
| --- |
| //创建对象  **Dog1 d=new Dog1("美美",90,10,"雪娜瑞");** //调用带参的方法的注意事项，方法参数的类型，个数，顺序必须与方法定义完全相同  d.print();//调用方法    **Dog1 d2=new Dog1();**//调用无参构造方法  d2.print(); |



##### (2)、方法的重载

(1)在同一个类里

(2)方法的名称相同

(3)参数的类型，个数，顺序不同

(4)与方法的返回值和访问修饰符无关

**注意事项**

|  |
| --- |
| //是构造方法吗?不是，因为有返回值类型，**叫普通方法，实例方法或是成员方法**  **public Dog1 Dog1(){**  //Dog1 d=new Dog1();  //return d;  **return new Dog1();**//new Dog1()创建对象，在堆中分配空间，没有赋给栈中的任对象所引用，new Dog1()叫做匿名对象  **}** |

**构造方法与普通方法的区别?**

**构造方法：**是用于创建对象的，在创建对象的时候调用

**普通方法：**是描述对象所具有的行为，在创建对象之后，使用对象名 .方法名()去调用

先使用**构造方法**创建对象，再使用对**象名去调用普通方法**

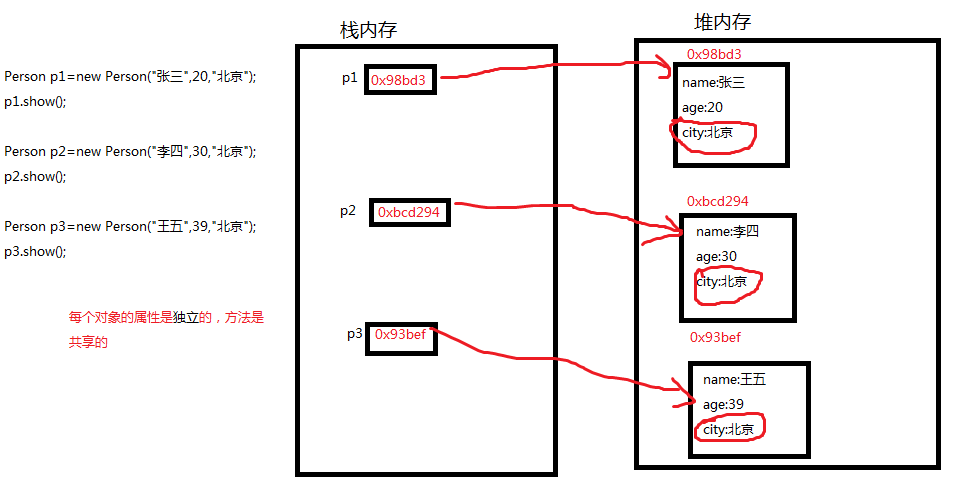
**调用一次构造方法，则创建一个新的对象**

**而普通方法可以被同一个对象调用N次**

|  |
| --- |
| **//创建对象**  **Dog1 d=new Dog1("美美",90,10,"雪娜瑞");**  **Dog1 d2=new Dog1();**  **d2**.print();  **d2**.print();  **d2**.print(); |

#### （4）、static关键字

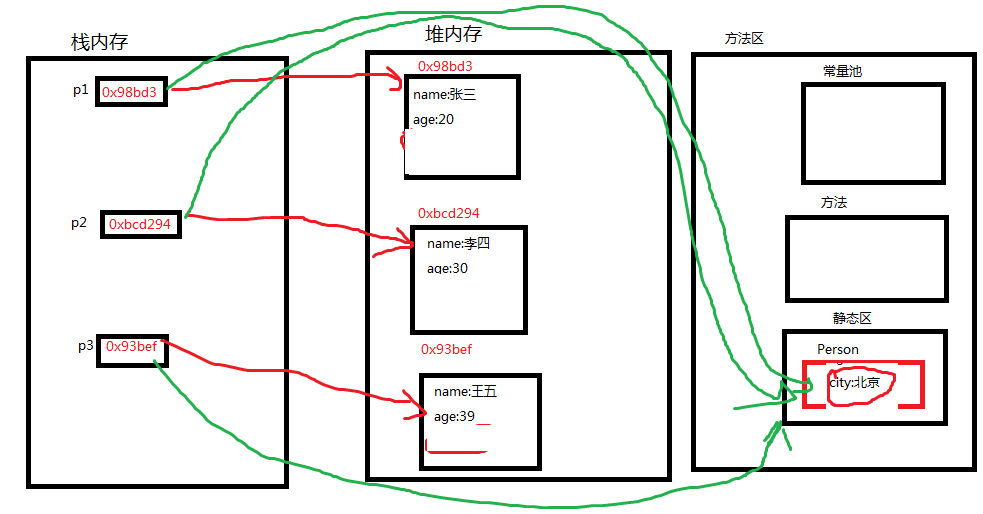
为什么需要static



根据以上代码，可以看出，city这个成员属性是每个对象都有的，而且属性的值都是相同的，如果创建N多个对象的话，会造成内存空间的大量浪费，那么这个时候，就可以考虑，让city被所有的对象所共享最好。

**使用static可以解决这个问题**

|  |
| --- |
| class Person{  String name;//名称  int age;//年龄  **static** String city;//城市  } |

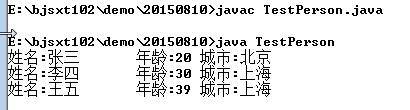


加上static之后，这个属性就被所有的对象所**共享,它将从堆中去除，放到了静态区**

**如果一个对象将值改变之后，那么所有的对象看到的都是改变后的结果**

|  |
| --- |
| Person p1=new Person("张三",20,"北京");  p1.show();    Person p2=new Person("李四",30);  **p2.city="上海";**  p2.show();    Person p3=new Person("王五",39);  p3.show(); |

**运行效果图**



**Static是修饰符**

可以修饰属性 :类属性,**不可以修饰局部变量**

可以修饰方法: 类方法

**归类所有，**当 类加载到内存时，**static修饰的方法或属性就被加载进来了**,与对象无关

所以可以使用类名.属性，或类名.方法名()

|  |
| --- |
| //类名.属性  **Person.city="上海"; //static修饰的属性**  **Person.print(); //static修饰的方法** |

是static修饰的方法或属性先产生还是对象先产生?

|  |
| --- |
| **String name;**//名称  **int age;/**/年龄  static String city;//城市  //方法  public **static** void show(){  System.out.println("姓名:"+name+"\t年龄:"+age+"\t城市:"+city);  } |

运行效果:



Name与age归对象所有，而方法show()归类所有，当类加载到内存的时候，还没有对象的产生，所以不能使用实例属性（成员属性）name和age

**总结**：在静态方法中不能使用成员属性或成员方法

This关键字：（1）局部变量与成员变量重名的问题

(2)代表当前对象

(3)可以使用this.方法名()去调用本类中的方法,this 可以省略

**在static中可以使用this关键字吗?**

**不可以：**因为this代表当前对象，static修饰的属性或方法加载到内存时，

This还没有产生

**Static与非 static的区别**



#### （5）、代码块

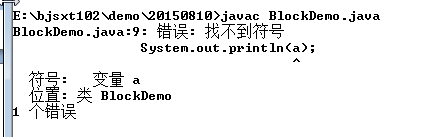
**概念:**使用”{}”括起来的一段代码

**分类：**根据位置可分类

普通代码块🡪直接在方法或语句中定义的代码块

|  |
| --- |
| public class BlockDemo{  public static void main(String [] args){//归类所有  new BlockDemo().show();  }  public void show(){//实例方法  //普通代码块-->声明在方法或语句  for( int i=0;i<5;i++){ //语句    }  //方法  **{**  **int a=10;**  **}**  **System.out.println(a);**  }  } |

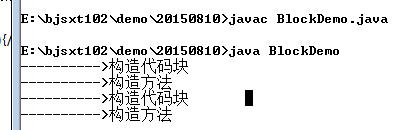
超出了作用域的范围



构造代码块🡪直接写在类中的代码块

|  |
| --- |
| public class BlockDemo{  //定义在类中的代码块，**构造代码块**  **{**  **System.out.println("---------->构造代码块");**  **}**  //构造方法  **public BlockDemo()**  {  System.out.println("---------->构造方法");  }  public static void main(String [] args){//归类所有  new BlockDemo();  new BlockDemo();  }  } |

运行效果:

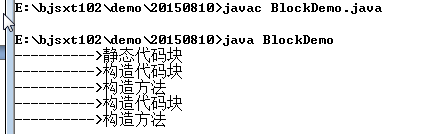


构造代码块优先于构造方法先执行，而且每创建一个对象的时候都会去执行构造代码块

**静态代码块**🡪使用static声明的代码块

|  |
| --- |
| public class BlockDemo{  //定义在类中的代码块，构造代码块  {  System.out.println("---------->构造代码块");  }  //构造方法  public BlockDemo()  {  System.out.println("---------->构造方法");  }  //静态代码块  **static{**  **System.out.println("---------->静态代码块");**    **}**  **public static void main(String [] args){//归类所有**  **new BlockDemo();**  **new BlockDemo();**  **}** |

运行效果:



静态代码块优先于构造代码块先执行，并且归类所有， 在类加载的时候执行一次，与对象无关，无论创建多少个对象，静态代码块只执行**一次。**

**作用:给静态属性赋值**

同步代码块🡪**多线程的时候会学到**

#### （6）、类的访问权限

(1)public：

对于成员来说：任何其他类都可以访问它们，不管在同一个包中还是在另外

的包中。

对于类来说： 也是一样。

(2)friendly：

对于成员来说：如果一个类的成员没有任何权限修饰，那么它门就是缺省包访问权限，用 friendly 来表示，注意 friendly 不是 Java 中的关键字，这里是个人喜欢的方式用它表示而已。同一个包内其它类可以访问，但包外就不可以。对于同一个文件夹下的、没有用 package 的 classes，Java 会自动将这些 classes 初见为隶属于该目录的 default package，可以相互调用 class 中的 friendly 成员。如以下两个 class分别在同一个文件夹的两个文件中，虽然没有引入 package，但隶属于相同的 default package。对于类来说：同一个包中的类可以用。总之，类只可以声明为 public 或者friendly。

(3)private：

对于对于成员来说：只能在该成员隶属于的类中访问。

对于类来说：类不可以声明为 private。

4)protected：

对于对于成员来说： 相同包中的类可以访问 （包访问权限） ； 基类通过 protected把基类中的成员的访问权限赋予派生类不是所有类（派生类访问权限）。对于类来说：类不可以声明为 protected

说明：

1、每个编译单元（类文件）都仅能有一个 public class

2、public class 的名称（包含大小写）必须和其类文件同名。

3、一个类文件(\*.java)中可以不存在 public class。

这种形式的存在的场景：如果我们在某个包内撰写一个 class，仅仅是为了配合同包内的其他类工作，而且我们不想再为了撰写说明文档给客户（不一定是现实意义的客户，可能是调用这个类的类）看而伤脑筋，而且有可能过一段时间之后有可能会彻底改变原有的做法，并完全舍弃旧版本，以全新的版本代替。

4、class 不可以是 private 和 protected。

5、如果不希望那个任何产生某个 class 的对象，可以将该类得所有构造函数设置成 private。但是即使这样也可以生成该类的对象，就是 class 的 static 的成员（属性和方法）可以办到。

**同包中的情况(同一个文件夹下)**

1. **本类 🡪四种访问权限均可以使用**

|  |
| --- |
| **package** cn.sxt;  **public** **class** Father {  **private** **int** age;//私有的年龄  **double** money;//默认的  **protected** String car;//受保护的  **public** String house;//公有的  **public** **void** method(){  **System.*out*.println(this.age);//private**  **System.*out*.println(this.money);//默认的**  **System.*out*.println(this.car);//protected**  **System.*out*.println(this.house);//public**  }  } |

1. **子类🡪三种**

|  |
| --- |
| **package** cn.sxt;  **public** **class** Son **extends** Father{  **public** **void** show(){  **System.*out*.println(super.age);//private //出错**  System.*out*.println(**super**.money);//默认的  System.*out*.println(**super**.car);//protected  System.*out*.println(**super**.house);//public  }  } |

1. **其它类🡪三种**

|  |
| --- |
| **public** **class** Other {  **public** **void** show(){  Father f=**new** Father();  **System.*out*.println(f.age);//private //出错**  System.*out*.println(f.car);//protected  System.*out*.println(f.house);//public  System.*out*.println(f.money);//默认的  }  } |

**不同的包的情况**

1. **子类🡪两种 protected,public**

|  |
| --- |
| **package** cn.sxt.s102;//包的声明  **import** cn.sxt.Father;//导包  **public** **class** SSon **extends** Father {  **public** **void** show(){  **System.*out*.println(super.age);//private**  **System.*out*.println(super.money);//默认的**  System.*out*.println(**super**.car);//protected  System.*out*.println(**super**.house);//public  }  } |

1. **其它类🡪有一种public**

|  |
| --- |
| **package** cn.sxt.s102;  **import** cn.sxt.Father;  **public** **class** SOther {  **public** **void** method(){  Father f=**new** Father();  **System.*out*.println(f.age);//private**  **System.*out*.println(f.car);//protected**  **System.*out*.println(f.money);//默认的**  System.*out*.println(f.house);//public  }  } |

继承关系中的构造方法执行顺序

|  |
| --- |
|  |

**执行顺序**

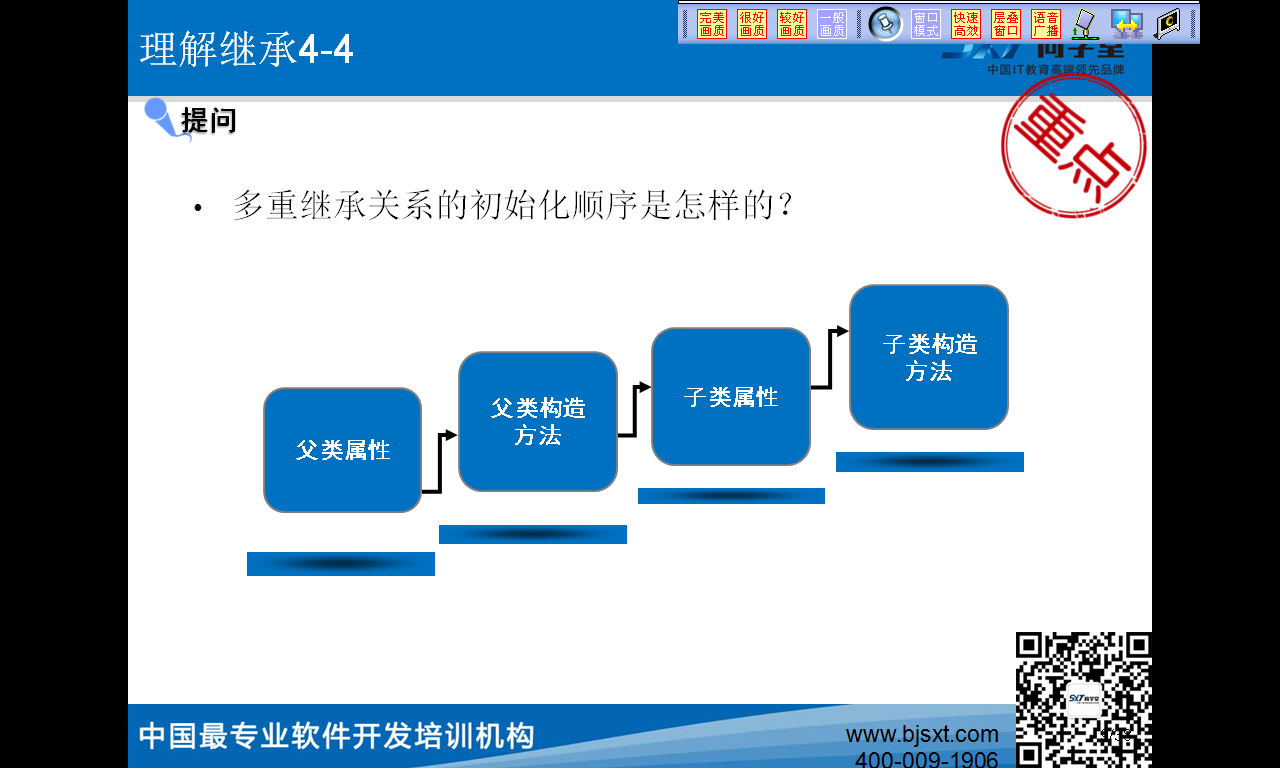
|  |
| --- |
|  |

**内存图**

|  |
| --- |
|  |

在创建子类对象的时候，先执行父类的属性赋值，再执行构造方法，将局部变量的值赋给成员变量，然后再执行子类的的属性赋值，最后执行子类构造方法，将局部变量的的值赋给成员变量。

**在创建子类对象的时候，调用了**父类的构造方法**，但是**没有创建父类的对象



小结：

**继承的作用：**实现代码的复用

**必须满足** :is …a

**继承的关键字:**extends

**子类不可以从父类继承什么?**private, 构造方法，不在同包中的默认的…

**Super关键字: 父类对象，只能出现在子类中，**

**调用父类的构造方法，必须是构造方法中 的第一句，不可以与this()一起使**

**访问权限:private ,默认，protected，public**

**同包:本类（4个），子类(3个)，其它类(3个)**

**不同包:子类(2个)，其它类(1)**

**创建子类对象的时候，构造方法的执行顺序**

父类的属性，父类的构造方法，子类的属性，子类的构造方法

### 2、面向对象之——封装

#### （1）、什么是封装

封装：顾名思义，隐藏对象的属性和实现细节，仅对外公开接口，控制在程序中属性的读和修改的访问级别；将抽象得到的数据和行为（或功能）相结合，形成一个有机的整体，也就是将数据与操作数据的源代码进行有机的结合，形成“类”，其中数据和函数都是类的成员。

封装的目的是增强安全性和简化编程，使用者不必了解具体的实现细节，而只是要通过 外部接口，一特定的访问权限来使用类的成员。

封装的大致原则:

1 把尽可能多的东西藏起来.对外提供简捷的接口.

2、把所有的属性藏起来.

3、封装好处：将变化隔离；便于使用；提高重用性；安全性。

#### （2）、为什么需要使用封装:

当给属性赋值的的时候，赋的值不合理

|  |
| --- |
| Dog1 d2=new Dog1();//调用无参构造方法  **d2.health=-1000;**  d2.print(); |

运行效果:

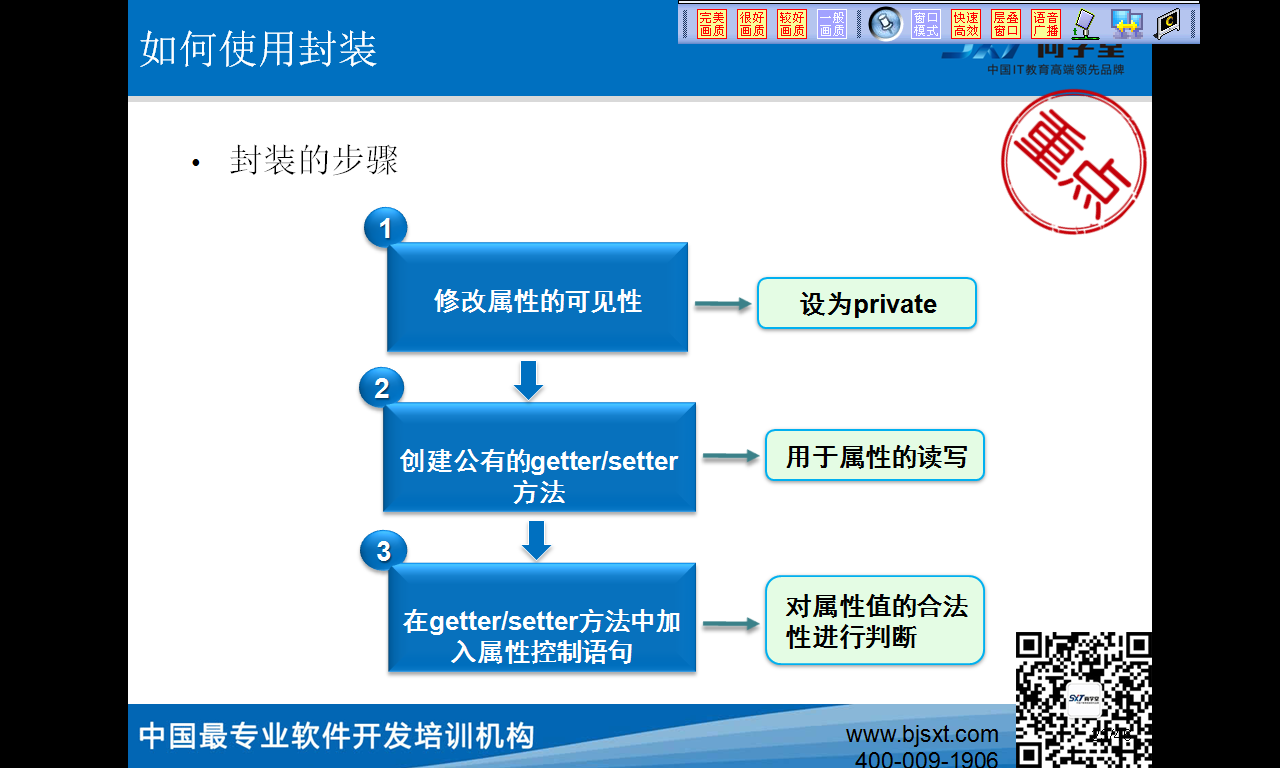


这个情况程序可以运行，但不符合实际情况，(不合理)

**使用封装的好处**

1. 隐藏了内部的实现细节
2. 可以通过公有的方法去访问
3. 可以加入控制语句
4. 方便修改

#### （3）、如何来使用封装

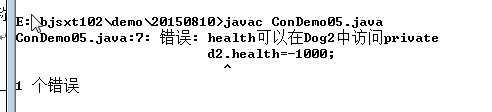


1. **将属性私有化**

|  |
| --- |
| **//[1]将属性私有化**  private String name;//昵称  private int health;//健康值  private int love;//亲密度  private String strain;//品种 |

**测试**

|  |
| --- |
| public class ConDemo05{  public static void main(String [] args){      Dog2 d2=new Dog2();//调用无参构造方法  **d2.health=-1000;**  d2.print();    }  } |



1. **提供公有的取值和赋值的方法**

|  |
| --- |
| **//[2]编写取值和赋值的方法**  public void setHealth(int health){  **//[3]加入控制语句**  if(health<0||health>100){  this.health=60;  }else{  this.health=health;//局部变量与成员变量重名，this代表成员变量  }  }  public int getHealth(){  return health;  } |

**注意事项**

|  |
| --- |
| public Dog2(String name,int health,int qinmi,String strain){  this.name=name; //将局部变量的值赋给了成员变量s  this.health=health;  love=qinmi;  this.strain=strain;  } |

**Health私有化了，通过setHealth方法进行赋值，但是在构造方法中存在了给health赋值的操作，这个操作绕过了setHealth方法，所以对不合理的值没有进行判断。**

**解决方法，将health从构造方法中删除，那么health这个值必须通过setHealth进行赋值，**

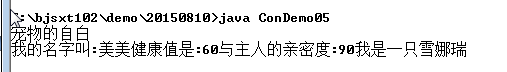
**其它的值则可通过构造方法进行赋值**

|  |
| --- |
| public Dog2(String name,int qinmi,String strain){  this.name=name; //将局部变量的值赋给了成员变量s  love=qinmi;  this.strain=strain;  } |

**测试**

|  |
| --- |
| Dog2 d2=new Dog2("美美",90,"雪娜瑞");  **d2.setHealth(-1000);**  d2.print(); |

**运行效果:**



构造方法在创建对象的时候给属性赋值了，而且只能赋一次，如果需要重新的给属性赋值的时候，还需要set赋值的方法

**Get与set必须同时存在吗?**

可以单独存在

#### （4）、this关键字

(1)代表当前对象

(2)当局部变量和成员变量重名时，this 代表的是成员变量

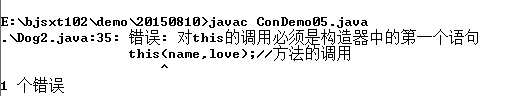
(3)调用属性和方法**，this可以省略不写**

|  |
| --- |
| **public void print(){**  **System.out.println("宠物的自白");**  **System.out.println("我的名字叫:"+this.getName()+"健康值是:"+getHealth()+"与主人的亲密度:"+this.love+"我是一只"+strain);**  **}** |

(4)调用构造方法，必须是构造方法中的第一条语句

|  |
| --- |
| public Dog2(String name,int love,String strain){  **//this(); // 代表调用无构造方法**    this.strain=strain;  **this(name,love);//方法的调用**  }  public Dog2(String name,int love){  this.name=name;  this.love=love;  } |

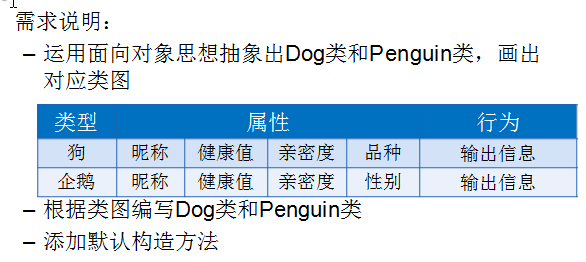
运行效果:



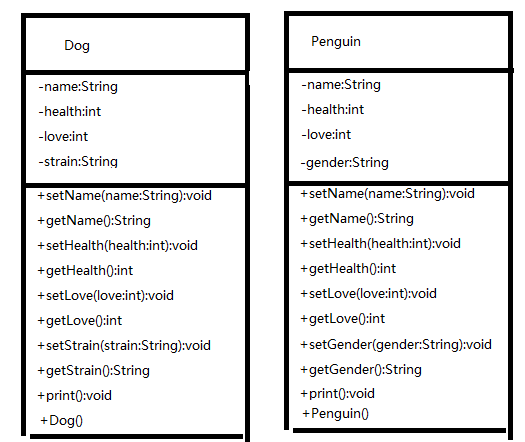
**修改代码为**

|  |
| --- |
| public Dog2(String name,int love,String strain){  **this(name,love);//方法的调用**  this.strain=strain;  }  public Dog2(String name,int love){  this.name=name;  this.love=love;  } |

**上机练习**



**类图**



根据类图去编写类

#### （5）、this 和 static 详解

this:代表对象。就是所在函数所属对象的引用。

this 到底代表什么呢？哪个对象调用了 this 所在的函数。this 就代表哪个对象。就是哪个对象的引用。

开发时，什么时候使用 this 呢？

在定义功能时，如果该功能内部使用到了调用该功能的对象。这时就用 this来表示这个对象。

this 还可以用于构造函数间的调用。

调用格式：this(实际参数)；

this 对象后面跟上 . 调用的是成员属性和成员方法(一般方法)；

this 对象后面跟上 () 调用的是本类中的对应参数的构造函数。

注意：用 this 调用构造函数，必须定义在构造函数的第一行。因为构造函数是用于初始化的，所以初始化动作一定要执行。否则编译失败。

static：关键字，是一个修饰符。用于修饰成员(成员变量和成员函数)。

特点：

1，想要实现对象中的共性数据的对象共享。可以将这个数据进行静态修饰。

2，被静态修饰的成员，可以直接被类名所调用。也就是说， 静态的成员多了

一种调用方式。类名.静态方式。

3，静态随着类的加载而加载。而且优先于对象存在。

弊端：

1，有些数据是对象特有的数据，是不可以被静态修饰的。因为那样的话， 特

有数据会变成对象的共享数据。这样对事物的描述就出了问题。所以，在定

义静态时，必须要明确，这个数据是否是被对象所共享的。

2，静态方法只能访问静态成员，不可以访问非静态成员。

因为静态方法加载时，优先于对象存在，所以没有办法访问对象中的成员。

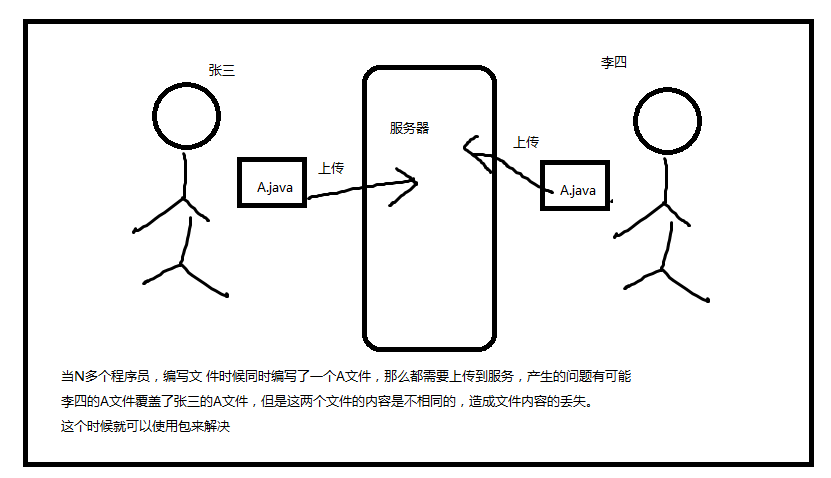
3，静态方法中不能使用 this，super 关键字。

因为 this 代表对象，而静态在时，有可能没有对象，所以 this 无法使用。

4，主函数是静态的。

#### （6）、包

为什么需要使用包

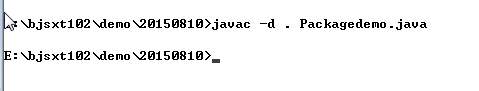


包相当于window中的文件夹，文件夹的作用是管理文件，将文件分门别类去存放。

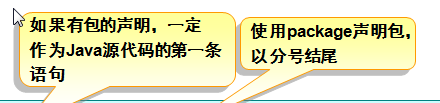
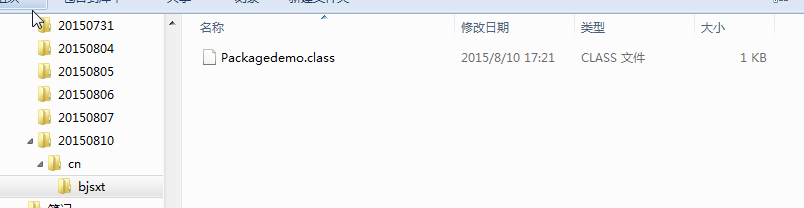
包的作用：管理类，而且还可以解决类的名称冲突的问题

**包的声明**

|  |
| --- |
| **package cn.bjsxt;//声明一个包**  public class Packagedemo{  } |



对应于window中的文件夹



Package必须是java源文件中的第一句非注释性代码，package不能再有代码，如果有只能是注释

在一个java源文件中，可以没有包的声明，但如果有包的声明，只能有一句

**包的命名规范：**

不能以圆点开头或结尾

倒置的网络域名，package cn.bjsxt.s102;

包名全部小写字母

不可以以数字开头

举例:

**包的声明**

|  |
| --- |
| **package cn.bjsxt.s102;**//声明一个包  public class Packagedemo{  public int age=10;  } |

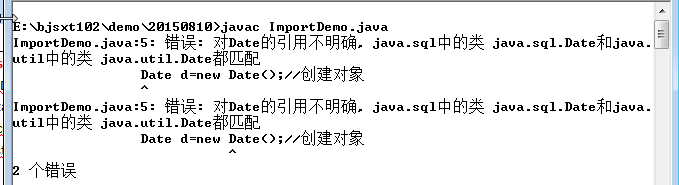
**包的使用**

|  |
| --- |
| **import cn.bjsxt.s102.Packagedemo;**//告诉应用程序，类的存放位置(.class文件的存放)  public class PDemo{  public static void main(String [] args){  Packagedemo p=new Packagedemo();  System.out.println(p.age);  }  } |

**不同包中的同名类**

|  |
| --- |
| import java.util.\*;  import java.sql.\*;  public class ImportDemo{  public static void main(String [] args){  **Date d=new Date();//创建对象**  }  } |

Util包与sql包都有Date这个类文件，那么应用程序就不知道你要使用哪个Date



如何解决，必须加上完整的包名

|  |
| --- |
| public static void main(String [] args){  **java.util.Date d=new java.util.Date();**//创建对象  } |

静态导入

静态导入前

|  |
| --- |
| **package cn.bjsxt.demo;**//声明包  public class Calc{  public static int a=20;  public static int add(int a,int b){  return a+b;  }  } |

测试

|  |
| --- |
| package cn.bjsxt;//声明包  import cn.bjsxt.demo.Calc;//导包  public class TestCalc{  public static void main(String [] args){  System.out.println**(Calc.a**);//类名.属性  System.out.println(**Calc.add(12,20**));//类名.属性  }  } |

现在使用静态导入，因为Calc这个类中的所有的属性和方法都是静态的，这个时候就可以使用静态导入

|  |
| --- |
| package cn.bjsxt;//声明包  **import static cn.bjsxt.demo.Calc.\*;//导包**  public class TestCalc{  public static void main(String [] args){  System.out.println(**a**);//类名.属性  System.out.println(**add(12,20)**);//类名.属性  }  } |

直接调用属性 或方法，类名可以省略

### 3、面向对象之——继承

#### （1）、什么是继承

**继承**是一种联结类的层次模型，并且允许和鼓励类的重用，它提供了一种明确表述共性的方法。对象的一个新类可以从现有的类中派生，这个过程称为类继承。新类继承了原始类的特性，新类称为原始类的派生类（子类），而原始类称为新类的基类（父类）。派生类可以从它的基类那里继承方法和实例变量，并且类可以修改或增加新的方法使之更适合特殊的需要。私有成员能继承，但是由于访问权限的控制，在子类中不能直接使用父类的私有成员。并且 java 中是单继承，一个子类只能有一个父类

继承中的构造方法

当生成子类对象时，Java 默认首先调用父类的不带参数的构造方法，然后执行该构造方法，生成父类的对象。接下来，再去调用子类的构造方法，生成子类的对象。【要想生成子类的对象，首先需要生成父类的对象，没有父类对象就没有子类对象。比如说：没有父亲，就没有孩子】。

如果子类使用 super()显式调用父类的某个构造方法，那么在执行的时候就会寻找与 super()所对应的构造方法而不会再去寻找父类的不带参数的构造方法。与 this 一样，super 也必须要作为构造方法的第一条执行语句，前面不能有其他可执行语句。

当两个方法形成重写关系时，可以在子类方法中通过 super.run()形式调用父类的 run()方法， 其中 super.run()不必放在第一行语句， 因此此时父类对象已经构造完毕，先调用父类的 run()方法还是先调用子类的 run()方法是根据程序的逻辑决定的。

#### （2）、如何实现继承

继承的作用：实现代码的复用

必须满足：is-a的关系:

举例:苹果是水果，三角形是几何图形，自行车是交通工具

父类： 被继承的类（基类）

子类： （派生类）

**编写父类**

|  |
| --- |
| **public class Pet {**  } |

子类继承自父类

|  |
| --- |
| public class **Dog** **extends** **Pet**{  } |

#### （3）、继承的特性

(1)单根性： 一个子类只能直接继承一个父类

|  |
| --- |
| **public** **class** C **extends** A,B {  } |

**上述代码，违背了单根性原则，**

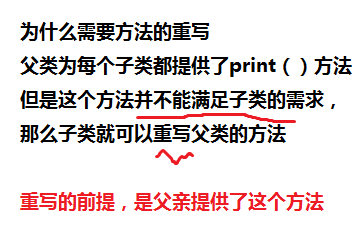
**类与类之间的继承只能是单继承，（接口可以实现多继承）**

(2)传递性：祖孙N代

B类继承自A类，C类继承自B类，可以说C继承自A，C就可以拥有A类中public的属性或方法

#### （4）、方法的重写

**为什么需要方法重写**



**什么是方法重写:**

重写的要求：

1、子类覆盖方法和父类被覆盖方法的方法返回类型，方法名称，参数列表必须相同

2、子类覆盖方法的访问权限必须大于等于父类的方法的访问权限

3、方法覆盖只能存在于子类和父类之间

4、子类覆盖方法不能比父类被覆盖方法抛出更多异常

**父类**

|  |
| --- |
| **private** **void** method1(){  System.*out*.println("父类的method1方法");  } |

**子类**

|  |
| --- |
|  |

**(如果父类的方法的访问权限是private,那么子类是不能继承的，就不能重写，只能叫与父类同名的方法)**

1. 返回值相同或者是其子类

**父类**

|  |
| --- |
| //方法，方法的返回值是Pet对象  **public** Pet pet(){  **return** **new** Pet();//new Pet()是匿名对象  } |

**子类**

|  |
| --- |
|  |

**子类**

|  |
| --- |
|  |

**子类重写父类的方法**

|  |
| --- |
|  |

**@Override:注解**

**绿色三角号:如果方法前存在这个符号，代表是方法的重写**

**Super.print();调用父类的方法**

当子类重了父类的方法之后，那么再调用的时候，调用的是子类重写之后的方法

|  |
| --- |
| //创建狗的对象  Dog d=**new** Dog("美美",90,"京巴");  Penguin p=**new** Penguin("哈哈",89,"雄");  d.print();//调用的是子类重写之后的方法  p.print(); |

**注意事项🡪父类与子类中的静态同名方法**

**父类**

|  |
| --- |
| **public** **static** **void** method2(){  System.*out*.println("父类的静态方法");  } |

**子类**

|  |
| --- |
|  |

**问:是方法重写吗? 不是：因为父类的method2归父类所有**

**子类的method2的方法归子类所有**

**小知识点:**

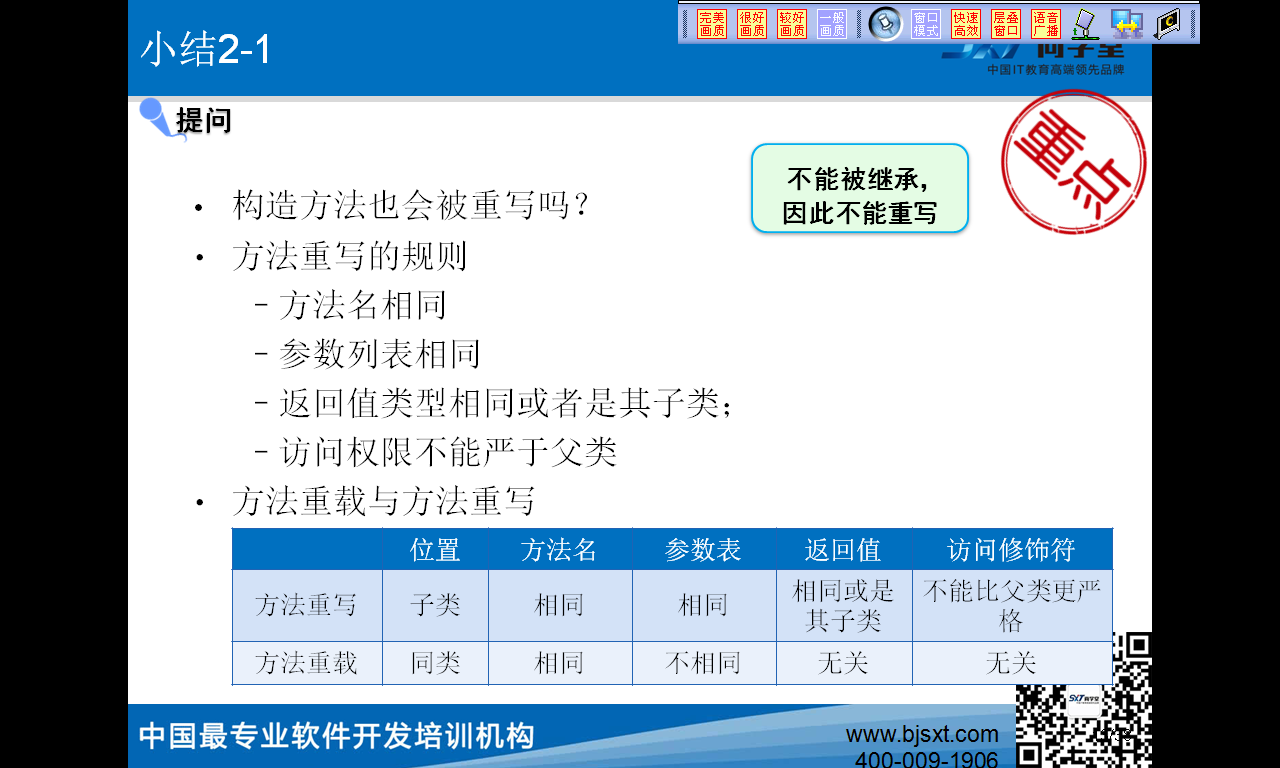
|  |
| --- |
| System.*out*.println(**this**.getName());  System.*out*.println(**super**.getName()); |

第一句：如果子类没有这个方法，就去父类调用

第二句:直接去父类调用

#### （5）、方法重写与方法重载之间的关系

重载发生在同一个类内部的两个或多个方法。重写发生在父类与子类之间。



#### （6）、final 关键字在继承中的使用

final 可以用于以下四个地方：

##### 1、定义变量，包括静态的和非静态的。

如果 final 修饰的是一个基本类型，就表示这个变量被赋予的值是不可变的，即它是个常量； 如果 final 修饰的是一个对象， 就表示这个变量被赋予的引用是不可变的，不可改变的只是这个变量所保存的引用，并不是这个引用所指向的对象，其实更贴切的表述 final 的含义的描述，那就是，如果一个变量或方法参数被 final 修饰， 就表示它只能被赋值一次， 但是 JAVA 虚拟机为变量设定的默认值不记作一次赋值。

被 final 修饰的变量必须被初始化。初始化的方式有以下几种：

1. 在定义的时候初始化。

2. 在初始化块中初始化。

3. 在类的构造器中初始化。

4. 静态变量也可以在静态初始化块中初始化。

##### 2、 定义方法。

当 final 用来定义一个方法时， 它表示这个方法不可以被子类重写， 但是它这不影响它被子类继承。

说明：

具有 private 访问权限的方法也可以增加 final 修饰，但是由于子类无法继承private 方法， 因此也无法重写它。 编译器在处理 private 方法时， 是按照 final方法来对待的，这样可以提高该方法被调用时的效率。不过子类仍然可以定义同父类中的 private 方法具有同样结构的方法， 但是这并不会产生重写的效果，而且它们之间也不存在必然联系。

##### 3、定义类。

由于 final 类不允许被继承，编译器在处理时把它的所有方法都当作 final 的，因此 final 类比普通类拥有更高的效率。final 的类的所有方法都不能被重写，但这并不表示 final 的类的属性（变量）值也是不可改变的，要想做到 final类的属性值不可改变，必须给它增加 final 修饰。

关于继承的几点注意：

a) 父类有的，子类也有

b) 父类没有的，子类可以增加

c) 父类有的，子类可以改变

d) 构造方法不能被继承

e) 方法和属性可以被继承

f) 子类的构造方法隐式地调用父类的不带参数的构造方法

g) 当父类没有不带参数的构造方法时，子类需要使用 super 来显式地调用父类的构造方法，super 指的是对父类的引用

h) super 关键字必须是构造方法中的第一行语句。

##### 4、super关键字

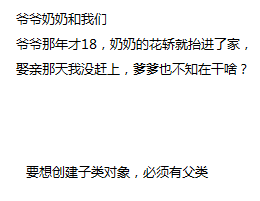
(1)super代表的是父类对象，只能出现在子类当中

(2)super可以调用父类的方法,**与顺序无关，super.方法名()**

(3)super可以调用父类的属性,**super.sex="公";,私有的除外(未分包的情况)**

(4)super可以调用父类的构造方法，必须是子类的构造方法中的第一句

**构造方法的执行顺序**



**注意事项1：🡪默认调用父类无参**

|  |
| --- |
| **public** Dog(String name,**int** love,String strain){  System.*out*.println("Dog类的有参构造方法");  } |

这句代码是子类的带参构造方法，没有指定调用父类Pet中的哪一个构造方法，那么默认调用的是父类的无参的构造方法，相当于在这个方法中写了一个super();

|  |
| --- |
| **public** Dog(String name,**int** love,String strain){  **super();**  System.*out*.println("Dog类的有参构造方法");  } |

如果使用super关键字，指定调用父类的构造方法，那么就不会去调用无参的构造方法了

|  |
| --- |
| **public** Dog(String name,**int** love,String strain){  **super**(name,love);//调用父类还有两个参数的构造方法  System.*out*.println("Dog类的有参构造方法");  } |

**注意事项2🡪this与super不能同时存在**

因为this,与 super 在调用构造方法的时候，都必须是构造方法中的第一句

**注意事项3🡪子类可以使用super调用父类的静态方法**

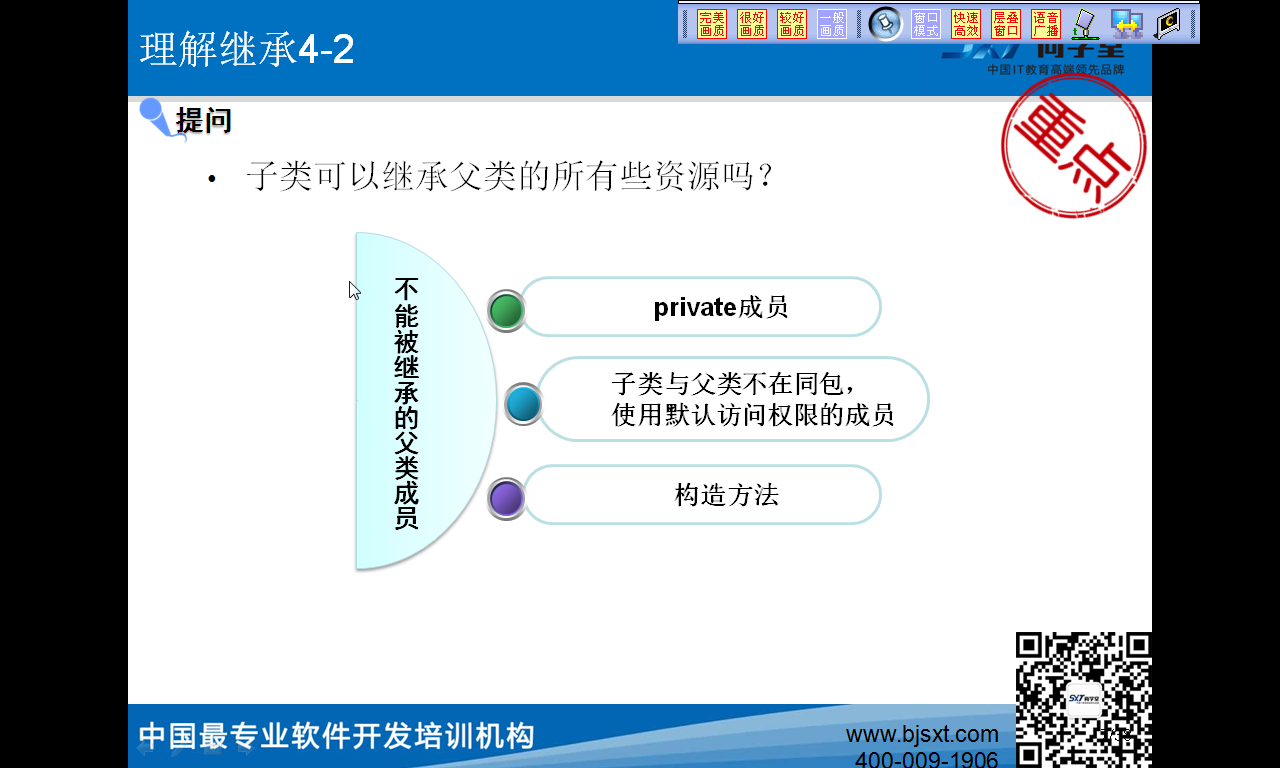
**父类**

|  |
| --- |
| public static void method(){  System.*out*.println("Pet类中的method方法");  } |

**子类**

|  |
| --- |
| **public** **void** show(){//实例方法，归对象所有  **super**.*method*();**//怎么就能调用父类的静态方法了呢？，因为创建子类对象的时候，必须先创建父类对象**  System.*out*.println("Dog为的show方法");  **super**.print();//调用父类的方法    } |

子类可以从父亲那继承所有的资源吗?



**不可以：比如 说生活中的，学历，驾驶本…**

**构造方法，**用于创建对象，父类的构造方法用于创建父类对象，子类不继承，但是，子类可以使用super关键字进行调用

#### （7）、抽象类

抽象类（abstract class）：使用了 abstract 关键字所修饰的类叫做抽象类。

抽象类无法实例化，也就是说，不能 new 出来一个抽象类的对象（实例）。

抽象方法（abstract method）：使用 abstract 关键字所修饰的方法叫做抽象方法。抽象方法需要定义在抽象类中。相对于抽象方法，之前所定义的方法叫做具体方法（有声明，有实现）。

如果一个类包含了抽象方法，那么这个类一定是抽象类。

如果某个类是抽象类，那么该类可以包含具体方法（有声明、有实现）。

如果一个类中包含了抽象方法，那么这个类一定要声明成 abstract class， 也就是说，该类一定是抽象类；反之，如果某个类是抽象类，那么该类既可以包含抽象方法，也可以包含具体方法。

无论何种情况，只要一个类是抽象类，那么这个类就无法实例化。

在子类继承父类（父类是个抽象类）的情况下，那么该子类必须要实现父类

中所定义的所有抽象方法；否则，该子类需要声明成一个 abstract class。

创建父类的对象

|  |
| --- |
| Pet p=**new** Pet("哈哈",90);  p.print(); |

从实际情况下来说，不合理，因为创建Dog的对象，我们能知道是什么样的，创建Pet类，宠物类的对象，**我们不知道具体是什么动物?**虽然程序能够运行，但在实际中不合理

可以采用抽象类来解决问题，对于父类不允许实例化对象,不能new父类

在class前加abstract关键字,这个类就变为了抽象类

|  |
| --- |
| public **abstract** class Pet {  } |

**测试类**

|  |
| --- |
|  |

**抽象类的特点:**

1. **不能创建对象（不能实例化）**

**加上abstract之后，这个类就不能创建对象，**

**问：**那么abstract的类，可以有构造方法吗?

可以，虽然不能创建对象了，但可以被子类调用

**加上abstract的方法叫抽象方法**

|  |
| --- |
| public **abstract** void show()**;** |

**抽象方法的特点:**

1. 抽象方法不能有方法体
2. 抽象方法以分号结尾
3. 含有抽象方法的类，必须是抽象类
4. 抽象类中可以有实例方法

**重写与实现**

|  |
| --- |
|  |

如果说父类中的方法是抽象方法，**那么子类必须去实现父类的抽象方法，（**除非子类也是抽象类**）**

**父类**

|  |
| --- |
|  |

**子类**

|  |
| --- |
|  |

由绿色的三角就变成了白色的空心的，这个时候叫**方法的实现**

**如果A类是抽象类，A类中有一个抽象方法method,**

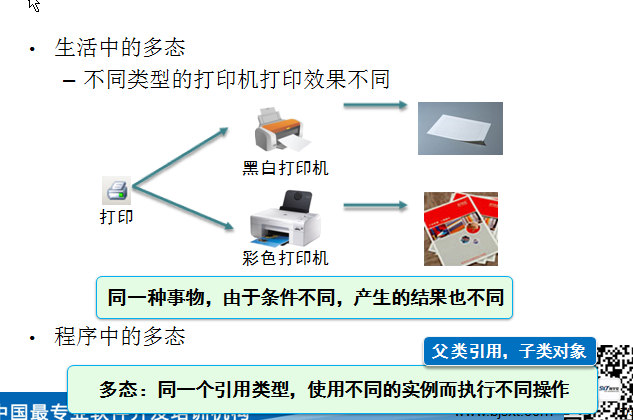
**B类继承了A类，B类是抽象类**

**C类继承B类，然后这个方法，由谁来重写 🡪C类来重写**

### 4、面向对象之——多态

多态（Polymorphism）：用我们通俗易懂的话来说就是子类就是父类（猫是动物，学生也是人），因此多态的意思就是：父类型的引用可以指向子类的对象。

##### （1）、什么是多态



**举例，程序中的多态的例子**

|  |
| --- |
| Pet d=**new** Dog("美美",90,40,"京巴王");  Pet p=**new** Penguin("丽丽",89,76,"Q妹");  Pet pig=**new** Pig("小白",40,12);  d.print();  p.print();  pig.print();//父类的方法 |

1. 左边的类型都是父类Pet类型，--🡪**同一种引用类型**
2. 右边是不同的子类对象-- -- >**不同的实例**
3. 执行的是子类重写之后的方法**，🡪不同的操作**

**实现多态的前提：继承**

**即然是继承，就有父子关系，有父子关系，就会用到重写**

**所以实现多态的两个素：**

|  |
| --- |
|  |

**使用父类作为方法的形参实现多态的好处**

|  |
| --- |
| **public** **void** feed(Pet p){//将父类作为方法的形参，实现多态  p.eat();  } |

这个方法编写完之后，更具有通用性，只需要传Pet类的任何子类都可以实现.Pet类是的抽象的，指的是一类事物，在运行的时候，只需要给这个类型中的任何一个对象就可以父类类型作为方法的返回值实现多态

作用：就是用于创建子类对象

方法的重写、重载与动态连接构成多态性。Java 之所以引入多态的概念， 原因之一是它在类的继承问题上和 C++不同，后者允许多继承，这确实给其带来的非常强大的功能，但是复杂的继承关系也给 C++开发者带来了更大的麻烦， 为了规避风险， Java 只 允许单继承， 派生类与基类间有 IS-A 的关系 （即“猫”is a “动物”）。这样做虽然保证了继承关系的简单明了，但是势必在功能上有很大的限制， 所以， Java 引入了多态性的概念以弥补这点的不足， 此外，抽象类和接 口也是解决单继承规定限制的重要手段。同时，多态也是面向对象编程的精髓所在。

在一个类中，可以定义多个同名的方法，只要确定它们的参数个数和类型不同， 这种现象称为类的多态。 类的多态性体现在两方面： 一是方法的重载上，包括成员方法和构造方法的重载；二是在继承过程中，方法的重写。

多态性是面向对象的重要特征。方法重载和方法覆写实际上属于多态性的一种体现，真正的多态性还包括对象多态性的概念。

对象多态性主要是指子类和父类对象的相互转换关系。

a) 向上类型转换（upcast）：比如说将 Cat 类型转换为 Animal 类型，即将子类型转换为父类型。对于向上类型转换，不需要显式指定。

b) 向下类型转换（downcast）：比如将 Animal 类型转换为 Cat 类型。即将父类型转换为子类型。对于向下类型转换，必须要显式指定（必须要使用强制类型转换）。

(1)自动类型转换🡪向上转型 🡪子类转父亲

**作用:实现多态，**

|  |
| --- |
| **Pet p**=new Penguin("丽丽",89,76,"Q妹"); |

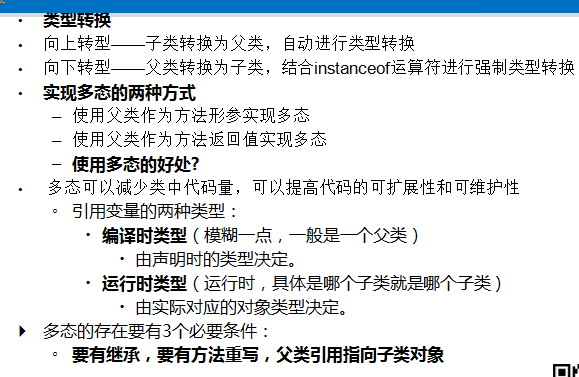
(2)强制类型转换🡪向下转型🡪父类转子类

**作用：**调用子类独有的方法

|  |
| --- |
| **public** **void** play(**Pet p**){//将父类作为方法的形参  **if** (p **instanceof** Dog) {  **Dog d=(Dog)p;/**/将父类再转换为子类  d.catchingFlyDisc();//调用子类独有的方法  }**else** **if** (p **instanceof** Penguin) {  **((Penguin)p)**.swimming();  }  } |

举例:

|  |
| --- |
| **public** **static** **void** main(String[] args) {  **Pet p=new Dog();//调用无参构造方法**  //只能调用父类中的方法-->子类重写后的方法，如果子类没有重写，调用的是父类的方法  System.*out*.println("p instanceof Pet:-->"+(p **instanceof** Pet));  System.*out*.println("p instanceof Dog:-->"+(p **instanceof** Dog));    **Dog d=new Dog();**  d.catchingFlyDisc();  //即可调用父类的方法，也可调用自己独有的方法  System.*out*.println("d instanceof Pet:-->"+(d **instanceof** Pet));  System.*out*.println("d instanceof Dog:-->"+(d **instanceof** Dog));    } |



多态的前提条件:--继承

没有继承，无从多态 --

实现多态的步骤:

1. 编写父类
2. 编写子类，继承父类，实现方法重写
3. 父类指向子类对象

如何实现多态

1. 父类作方法的参数
2. 父类作方法的返回值

网上摘抄的一段多态小总结：

1. Java 中除了 static 和 final 方法外，其他所有的方法都是运行时绑定的。

2.在派生类中，对于基类中的 private 方法，最好采用不同的名字。

3.包含抽象方法的类叫做抽象类。注意定义里面包含这样的意思，只要类中包含一个抽象方法， 该类就是抽象类。 抽象类在派生中就是作为基类的角色，为不同的子类提供通用的接口。

4.对象清理的顺序和创建的顺序相反，当然前提是自己想手动清理对象，因为大家都知道 Java 垃圾回收器。

5.在基类的构造方法中小心调用基类中被重写的方法，这里涉及到对象初始化顺序。

6.构造方法是被隐式声明为 static 方法。

7.用继承表达行为间的差异，用字段表达状态上的变化。

## 七、接口

|  |
| --- |
| //接口是类吗?不是类，因为类使用class声明,但是接口是一种数据类型,引用数据类型  **public** **interface** Lock {  } |

**注意事项:**接口的访问修饰符，只能是public 和默认的访问权限

### 1、接口的特点：

1. 接口不可以实例化。 🡪就是不能创建对象

**那么接口可有构造方法吗? 不可以（**因为构造方法要求，方法名与类名同名，因为接口不是类，所以找不到类名**）**

1. 接口中的所有的方法都是public abstract的(公共的抽象方法)

**如下代码，作用相同**

|  |
| --- |
| **void** method1();//默认为public abstract,可以省略  **public** **abstract** **void** method2(); |

1. 接口中的变量都是静态常量(**static final**),接口中的变量都默认使用static final来修饰

|  |
| --- |
| **int** *x*=1;  **public** **final** **static** **int** *y*=9; |

即然是常量就有初值，而且这个初值只能赋一次，

即然加 上static，那个这个值只能使用**接口名去访问**

1. 实现类必须实现接口中的所有的方法 **,除非实现类也是接口**

因为接口中的所有方法都是抽象方法，抽象方法必须被子类实现

**接口中的方法可以使用final修饰吗?**

**答:**不能，因为接口的中方法必须被子类实现，但是final是终止的意思，是不允许子类重写或实现的 .所以….

**接口可以使用final修饰吗 ?**

**答：**不可以，因为final是终止的，如果加 final是不能有子类的,但是，不能有子类，就无从谈重写或实现

**接口可以使用abstract修饰吗?**

**答:**因为含 有抽象方法的类必须是的抽象类，

可以这样理解，接口中的所有方法都是抽象方法，可以把接口理解为特殊抽象类

**你还可以这样理解**

**如果一个类中的所有方法都是抽象方法的话，那么就可以将这个类定义接口**

1. **接口与接口的关系🡪继承, 接口可以实现多继承**

|  |
| --- |
| //extends的意思是继承,  **public** **interface** InterfaceD **extends**  InterfaceA, InterfaceB, InterfaceC {  } |

**接口与类的关系🡪实现关系,一个类可以实现N多个接口**

**接口**

|  |
| --- |
| **public** **interface** InterfaceA {  **public** **void** methodA();  } |

|  |
| --- |
| **public** **interface** InterfaceB {  **public** String methodA(**int** a);  } |

**实现类**

|  |
| --- |
| **public** **class** ImplementClass **implements** InterfaceA, InterfaceB{  @Override  **public** **void** methodA() {  // **TODO** Auto-generated method stub  }  @Override  **public** String methodA(**int** a) {  // **TODO** Auto-generated method stub  **return** **null**;  }  } |

**实现类现时实现InterfaceA和InterfaceB,这两个接口中有同名方法，那么要求，同名方法在实现类中必须构成方法的重载**

1. 如果一个类即有继承，又实现接口，谁在前谁在后?**继承在前，实现接口在后**

这个子类必须实现父类中所有的抽象方法和接口中的所有方法

**接口是一种功能，或接口是一种能力**

**（程序员是代表一类人，只规定了具备什么样的能力，而不关心，谁具有这样的能力）**

**接口是一种规范，或者是说是一种标准，**

**（插排，两项插头或三项插头）**

**Java 语言不支持一个类有多个直接的父类(多继承),但现实例子中， 又有很多类似于多继承的例子， 比如教师， 他的父类既可以是人， 也可以是父母，**

**所以，在 java 中就用继承来填充这个空缺，java 不可以多继承， 但可以实现（implements）多个接口,间接的实现了多继承。**

**Java 接口的特征归纳：**

**1, Java 接口中的成员变量默认都是 public,static,final 类型的(都可省略)必须被显示初始化,即接口中的成员变量为常量(大写,单词之间用"\_"分隔)**

**2, Java 接口中的方法默认都是 public,abstract 类型的(都可省略),没有方体,不能被实例化**

**3, Java 接口中只能包含 public,static,final 类型的成员变量和 public,abstract类型的成员方法**

**4, 接口中没有构造方法,不能被实例化**

**5, 一个接口不能实现(implements)另一个接口,但它可以继承多个其它的接口**

**6, Java 接口必须通过类来实现它的抽象方法public class A implements B{...}**

**7, 当类实现了某个 Java 接口时,它必须实现接口中的所有抽象方法,否则这个类必须声明为抽象的**

**8, 不允许创建接口的实例(实例化),但允许定义接口类型的引用变量,该引用变量引用实现了这个接口的类的实例**

**9, 一个类只能继承一个直接的父类,但可以实现多个接口,间接的实现了多继承.**

**10、通过接口,可以方便地对已经存在的系统进行自下而上的抽象,对于任意两个类,不管它们是否属于同一个父类,只有它们存在相同的功能,就能从中抽象出一个接 口类型.对于已经存在的继承树,可以方便的从类中抽象出新的接口,但从类中抽象出新的抽象类却不那么容易,因此接口更有利于软件系统的维护与重构.对于两 个系统,通过接口交互比通过抽象类交互能获得更好的松耦合.**

**11,、接口是构建松耦合软件系统的重要法宝,由于接口用于描述系统对外提供的所有服务,因此接口中的成员变量和方法都必须是 public 类型的,确保外部使用者 能访问它们,接口仅仅描述系统能做什么,但不指明如何去做,所有接口中的方法都是抽象方法,接口不涉及和任何具体实例相关的细节,因此接口没有构造方法, 不能被实例化,没有实例变量.**

### 2, 比较抽象类与接口

**相同点**

**1, 代表系统的抽象层,当一个系统使用一颗继承树上的类时,应该尽量把引用变量声明为继承树的上层抽象类型,这样可以提高两个系统之间的送耦合**

**2, 都不能被实例化**

**3, 都包含抽象方法,这些抽象方法用于描述系统能提供哪些服务,但不提供具体的实现**

**不同点:**

**1, 在抽象类中可以为部分方法提供默认的实现,从而避免在子类中重复实现它们,这是抽象类的优势,但这一优势限制了多继承,而接口中只能包含抽象方法.由于在 抽象类中允许加入具体方法,因此扩展抽象类的功能,即向抽象类中添加具体方法,不会对它的子类造成影响,而对于接口,一旦接口被公布,就必须非常稳定,因 为随意在接口中添加抽象方法,会影响到所有的实现类,这些实现类要么实现新增的抽象方法,要么声明为抽象类**

**2, 一个类只能继承一个直接的父类,这个父类可能是抽象类,但一个类可以实现多个接口,这是接口的优势,但这一优势是以不允许为任何方法提供实现作为代价的 三, 为什么 Java 语言不允许多重继承呢?当子类覆盖父类的实例方**

**法或隐藏父类的成员变量及静态方法时,Java 虚拟机采用不同的绑定规则,假如还允许一个类 有多个直接的父类,那么会使绑定规则更加复杂,**

**结论：**

**因此,为了简化系统结构设计和动态绑定机制,Java 语言禁止多重继承.而接口中只有抽象方法,没有实例变量和静态方法,只有接口的实现类才会实现 接口的抽象方法(接口中的抽象方法是通过类来实现的),因此,一个类即使有多个接口,也不会增加 Java 虚拟机进行动态绑定的复杂度.因为 Java 虚拟机 永远不会把方法与接口绑定,而只会把方法与它的实现类绑定.四, 使用接口和抽象类的总体原则:**

**1, 用接口作为系统与外界交互的窗口站在外界使用者(另一个系统)的角度,接口向使用者承诺系统能提供哪些服务,站在系统本身的角度,接口制定系统必须实现哪 些服务,接口是系统中最高层次的抽象类型.通过接口交互可以提高两个系统之间的送耦合系统 A 通过系统 B 进行交互,是指系统 A 访问系统B 时,把引用变量声明 为系统 B 中的接口类型,该引用变量引用系统 B 中接口的实现类的实例。**

**2, Java 接口本身必须非常稳定,Java 接口一旦制定,就不允许随遇更加,否则对外面使用者及系统本身造成影响**

**3, 用抽象类来定制系统中的扩展点抽象类来完成部分实现,还要一些功能通过它的子类来实现**

## 八、内部类

Object类:

任何一个类都直接或间接继承自Object类

|  |
| --- |
| **public** **class** Person **extends** Object{  }  与  **public** **class** Person {  } |

**toString方法**

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Person p=**new** Person();//p的类型是Person ,类是对象的类型  System.*out*.println("p="+p);  System.*out*.println("p.toString()="+p.toString());  } |

总结:p默认调用了父类的toString方法,运行结果：显示的是内存地 址

|  |
| --- |
|  |

如果不能满足子类的需要的话，子类可以重写父类的方法

|  |
| --- |
| @Override  **public** String toString() {  // **TODO** Auto-generated method stub  **return** "姓名:"+name+"\t身份证号:"+**this**.getId();  } |

**运行效果:**

|  |
| --- |
|  |

**Equals方法—》Object类**

**从本质上来说也是比较内存地址是否相同，与”==”一个概念**

**重写之前 🡪”==”与equals都是用于比较内存地址**

|  |
| --- |
| Person pA=**new** Person("张一一","110110");//p的类型是Person ,类是对象的类型  Person pB=**new** Person("张一一","110110");  System.*out*.println("pA==pB"+(pA==pB));  System.*out*.println("pA.equals(pB)"+(pA.equals(pB))); |

|  |
| --- |
|  |

**重写之后 ---》用于比较对象的内容**

|  |
| --- |
| @Override  **public** **boolean** equals(Object obj) {  **if** (**this**==obj) {//如果当前对象与传入的对象地址相同，肯定指同一个对象  **return** **true**;  }  **if** (!(obj **instanceof** Person)) {//如果传入的对象不是Person的类型，  **return** **false**;  }**else**{//传入的对象是Person类型的对象，  //需要向下转型  Person p=(Person)obj;  **if** (**this**.getName().equals(p.getName())&&**this**.getId().equals(p.getId())) {  **return** **true**;  }  }  **return** **false**;  } |

**运行效果:**

|  |
| --- |
|  |

**内部类**

**一个类定义在了另一个类的内部，**

|  |
| --- |
| **public** **class** Test {🡪外部类  **class** T{ 🡪内部类    }  } |

**内部类的作用:🡪**方便访问外部类的私有属性

### （1）、成员内部类

|  |
| --- |
| **public** **class** Outer {//外部类  **private** String info="helloWorld";  **class Inner{ //成员内部类**  **public void fun(){**  **System.*out*.println(info);//内部类最大的好处就是可以访便的访问外部类私有属性**  **}**  **}**  **public** **void** show(){  **new** Inner().fun();//通过匿名对象调方法    }  } |

**测试**

|  |
| --- |
| **public** **class** Test {  **public** **static** **void** main(String[] args) {  **new** Outer().show();  }  } |

**注意事项:**

1. **外部类不能直接使用内部类的成员和方法,可以间接访问,创建对象方法**
2. 如果外部类和内部类具有相同的成员变量或方法，内部类

默认访问自己的成员变量或方法，如果要访问外部类的成员变量，需使用this关键字

|  |
| --- |
| **public** **class** Outer {//外部类  **private** String info="helloWorld";  **String name="李四";**  **class** Inner{ //成员内部类  **String name="张三 ";**  **public** **void** fun(){  **System.*out*.println(name);**  **System.*out*.println(Outer.this.name);**  //System.out.println(info);//内部类最大的好处就是可以访便的访问外部类私有属性  }  } |

**在其它的类访问内部类**

|  |
| --- |
| **//在测试类中如何访问内部类的方法**  **//内部类定义在外部类中，成为成员内部类**  **//第一种方法**  Outer o=**new** Outer();//创建外部类的对象    o.**new** Inner().fun();//创建内部类的匿名对象  **//第二种方法**  Inner i=o.**new** Inner();  i.fun();  **//第三种方法**  **new** Outer().**new** Inner().fun(); |

### （2）、静态内部类

**在内部类前加static关键字,**

**注意事项：**

静态内部类，不能访问外部类的非静态成员，因为在类加载到内存的时候，静态内部类就被加载进来了，但是这个时候还没有对象的产生，所以不能调用外部类的成员属性

|  |
| --- |
|  |

**注意事项：**如果内部类中的属性或方法使用static修饰了，那么这个内部类必须是静态内部类

|  |
| --- |
| **public** **class** Outer3 {//外部类  **static class Inner{**  //静态内部类  **static** String *name*="张三 ";  **public** **static** **void** fun(){  System.*out*.println(*name*);  }  }  } |

**静态内部类在其它类中如何访问**

|  |
| --- |
| **public** **static** **void** main(String[] args) {  **new** Outer2.Inner().fun();  }  相当于  Outer2.Inner i=**new** Outer2.Inner();  i.fun(); |

**创建内部类的匿名对象，调用方法**

因为内部类是静态的，所以使用外部类的  **类名.内部类,**但是静态内部类中的方法是非静态的，需要创建对象来方法，所以new的是内部类的对象

### （3）、匿名内部类 🡪就是没有名字的内部类

**前提：**必须有继承或实现接口

**有名字的内部类**

|  |
| --- |
| **public** **class** Outer4 {//外部类  //成员方法  **public** **void** method(){  **new** Inner().method();  }  //成员内部类  **class** Inner **extends** Father{//Inner继承了Father类,必须实现父类中的抽象方法  @Override  **public** **void** method() {  System.*out*.println("哈哈");  }  }  }  //定义一个父类  **abstract** **class** Father{  **public** **abstract** **void** method();//抽象方法  } |

**匿名内部类**

|  |
| --- |
| **package** cn.bjsxt.inner;  **public** **class** Outer5 {//外部类  //成员方法  // 因为匿名内部类是没有名字的内部类，没有名字就不能创建对象,就创建父类或接口的对象(抽象类和接口是不能创建对象的)  //在创建父类或接口的对象的时候对方法进行实现就可以了  **public** **void** method(){  **new** Father2() //创建父类或接口的对象  //代码块,在代码块中重写父类的方法  {  @Override  **public** **void** method() {  System.*out*.println("哈哈 ，嘿");  }    }.method();  }//方法的结束花括号  }  //定义一个父类  **abstract** **class** Father2{  **public** **abstract** **void** method();//抽象方法  } |

**测试类**

|  |
| --- |
| **public** **class** Test5 {  **public** **static** **void** main(String[] args) {  **new** Outer5().method();  }  } |

### （4）、方法中的内部类

**注意事项**

|  |
| --- |
| 方法内部类是指：将内部类定义在外部类的方法中。  **注意事项：**  （1）方法内部类不能在外部类的方法以外的地方使用，所以  方法内部类不能使用访问控制符和static修饰符  （2）方法内部类如果想使用方法的参数，那么参数前必须加  上final关键字 |

|  |
| --- |
| **public** **class** Outer6 {//外部类  **public** **void** show(**final** **int** **temp**){    //定义在了方法里  **class** Inner{  **public** **void** fun(){//方法中内部类的方法  System.*out*.println(**temp**);  }  }//内部类的结束  **new Inner().fun();**    }  **public** **static** **void** main(String[] args) {  **new** Outer6().show(10);  }  } |

## 九、异常

### 1、为什么需要异常处理

异常 是正常情况中特殊情况，有可能发生，也有可能不发生

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Scanner input=**new** Scanner(System.*in*);  System.*out*.println("请输入被除数:");  **int** numA=input.nextInt();  System.*out*.println("请输入除数：");  **int** numB=input.nextInt();  System.*out*.println(String.*format*("%d/%d=%d", numA,numB,numA/numB));  System.*out*.println("谢谢使用本程序");  } |

**正常情况:**

|  |
| --- |
|  |

**出现异常的时候**

|  |
| --- |
|  |

**输入的数据类型与程序期望的数据类型不一致**

**最后的一行，**告诉我们 是在程序中异常出现的行数

**第一行，**告诉我们 是异常的类型

|  |
| --- |
|  |

除数为0，在数学中，除数不能为0

**没有异常机制时，使用if-else解决问题**

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Scanner input=**new** Scanner(System.*in*);  System.*out*.println("请输入被除数:");  **int** numA;  **if** (input.hasNextInt()) {  numA=input.nextInt();  System.*out*.println("请输入除数：");  **int** numB;  **if** (input.hasNextInt()) {  numB=input.nextInt();  **if** (numB==0) {  System.*out*.println("除数不能为0");  }**else**{  System.*out*.println(String.*format*("%d/%d=%d", numA,numB,numA/numB));  System.*out*.println("谢谢使用本程序");  }  }**else**{  System.*out*.println("对不起，您输入的有误，请输入整数!");  }  }**else**{  System.*out*.println("对不起，您输入的有误，请输入整数!");  }  } |

**弊端：**

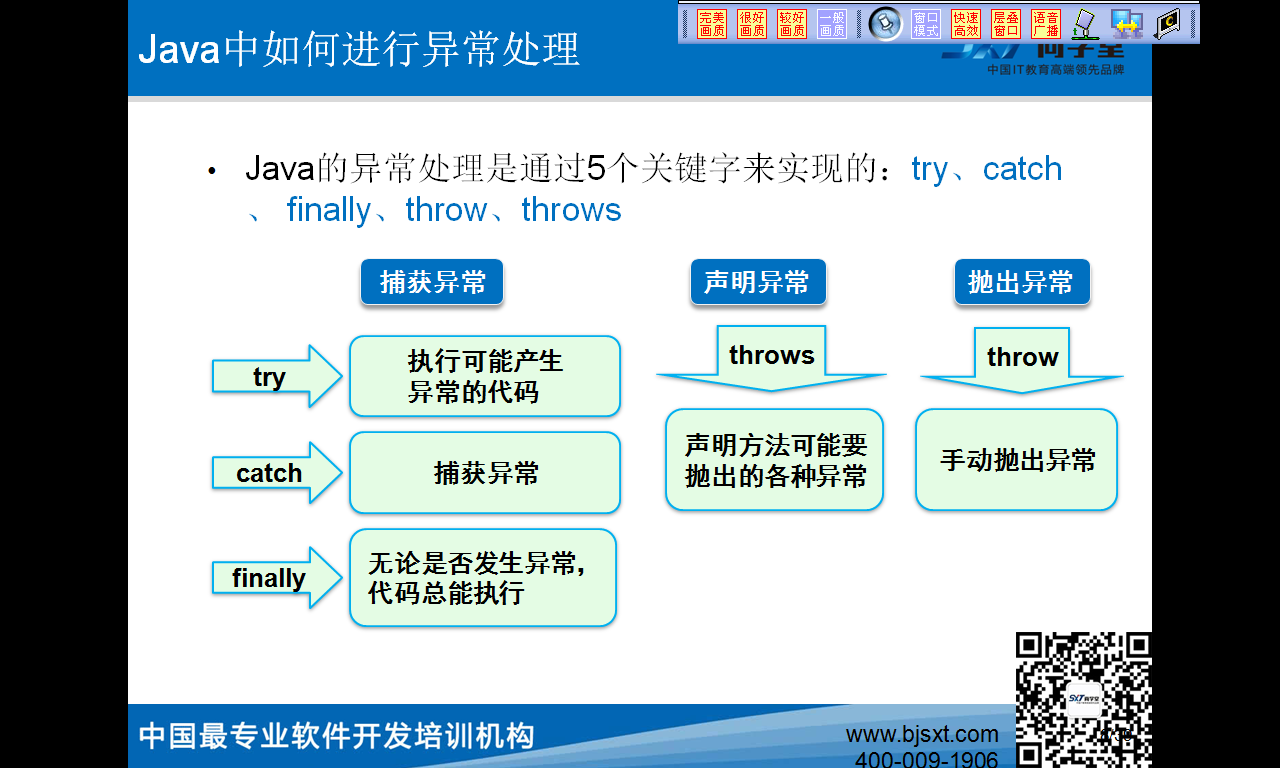
1、代码臃肿

2、程序员要花很大精力“堵漏洞”

3、程序员很难堵住所有“漏洞”

### 2、java处理异常的5个关键字

异常的组合形式



**第一种try…catch**

|  |
| --- |
| **try** {  System.*out*.println("请输入被除数:");  **int** numA=input.nextInt();  System.*out*.println("请输入除数：");  **int** numB=input.nextInt();  System.*out*.println(String.*format*("%d/%d=%d", numA,numB,numA/numB));  System.*out*.println("谢谢使用本程序");  } **catch** (Exception e) {  //e.printStackTrace();//打印异常的堆栈信息  System.*out*.println("出错了");  } |

**第二种组合try—finally**

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Scanner input=**new** Scanner(System.*in*);  **try** {  System.*out*.println("请输入被除数:");  **int** numA=input.nextInt();  System.*out*.println("请输入除数：");  **int** numB=input.nextInt();  System.*out*.println(String.*format*("%d/%d=%d", numA,numB,numA/numB));  } **finally**{  System.*out*.println("谢谢使用本程序");  }  } |

**第三种组合 try-catch..finally**

|  |
| --- |
| **try** {  System.*out*.println("请输入被除数:");  **int** numA=input.nextInt();  System.*out*.println("请输入除数：");  **int** numB=input.nextInt();  System.*out*.println(String.*format*("%d/%d=%d", numA,numB,numA/numB));  } **catch** (Exception e) {  System.*out*.println("出错了");  }**finally**{  System.*out*.println("谢谢使用本程序！");  } |

### 3、异常的执行情况

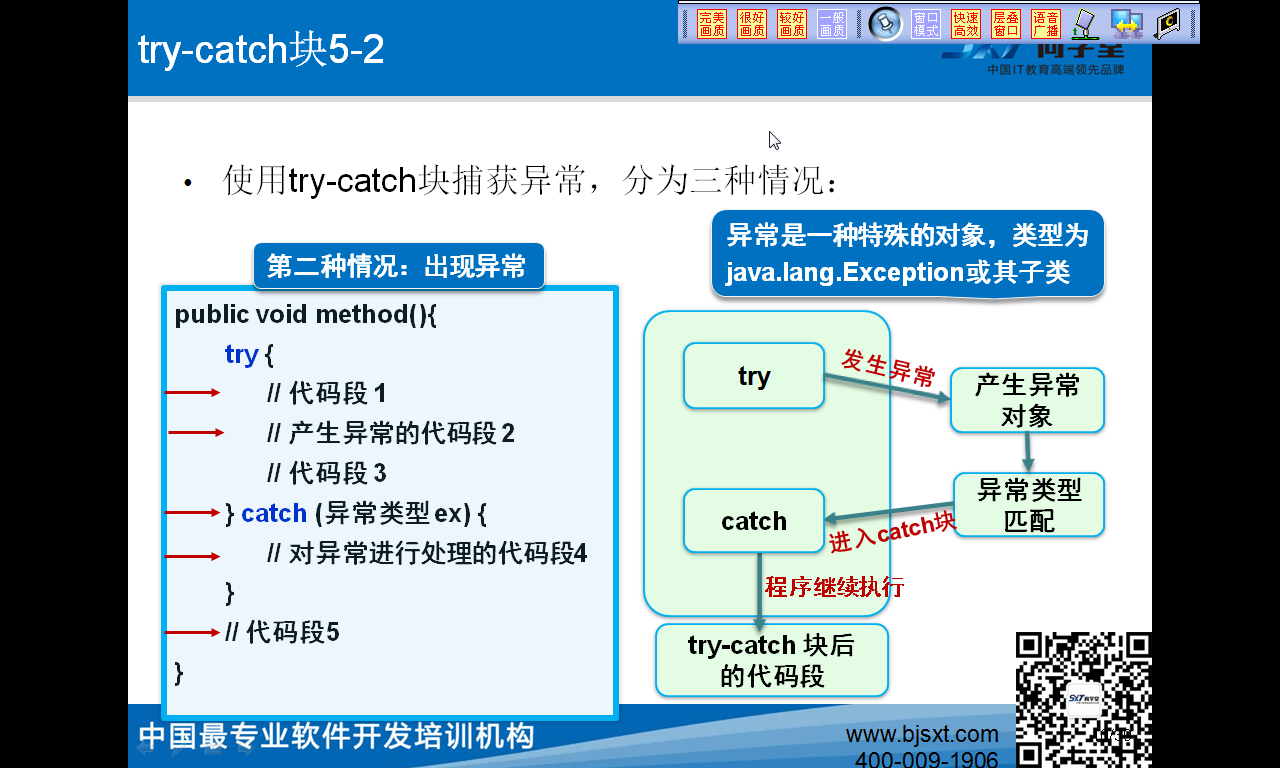
(1)正常情况



**红色的代码是程序所执行的代码**

|  |
| --- |
| **try** {  **System.*out*.println("请输入被除数:");**  **int numA=input.nextInt();**  **System.*out*.println("请输入除数：");**  **int numB=input.nextInt();**  **System.*out*.println(String.*format*("%d/%d=%d", numA,numB,numA/numB));**  } **catch** (Exception e) {  e.printStackTrace();  System.*out*.println("出错了");  }  **System.*out*.println("谢谢使用本程序");** |

**第二种情况，出现了异常，而且异常的类型 与catch所捕所的异常相匹配**

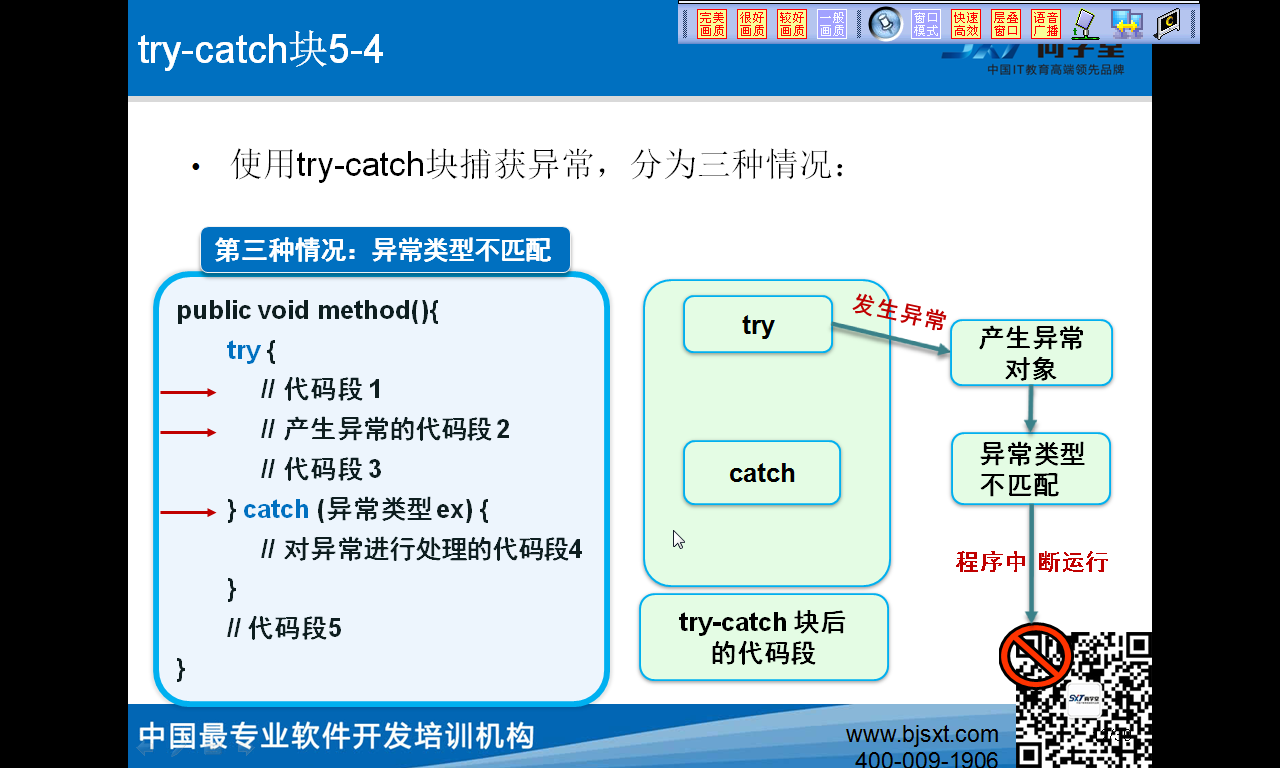


**运行效果图:**

|  |
| --- |
|  |

|  |
| --- |
| **try** {  **System.*out*.println("请输入被除数:");**  **int numA=input.nextInt();**  System.*out*.println("请输入除数：");  **int** numB=input.nextInt();  System.*out*.println(String.*format*("%d/%d=%d", numA,numB,numA/numB));  **} catch (Exception e) {**  //e.printStackTrace();  **System.*out*.println("出错了");**  }  **System.*out*.println("谢谢使用本程序");** |

**第三种情况，产生了异常，但是异常的类型与catch所捕获的异常不匹配**



**运行效果图**

|  |
| --- |
|  |

|  |
| --- |
| **try** {  **System.*out*.println("请输入被除数:");**  **int numA=input.nextInt();**  System.*out*.println("请输入除数：");  **int** numB=input.nextInt();  System.*out*.println(String.*format*("%d/%d=%d", numA,numB,numA/numB));  **} catch (ArithmeticException e) {**//算术异常  //e.printStackTrace();  System.*out*.println("除数不能为0");  }  System.*out*.println("谢谢使用本程序"); |

**异常信息的三种形式**

|  |
| --- |
| **catch** (ArithmeticException e) {//算术异常  **//[1]e.printStackTrace();**  **//[2]System.err.println("除数不能为0");**  **//[3]System.out.println(e.getMessage());**    } |

### 4、常用的异常类型



**继承关系图**

|  |
| --- |
|  |

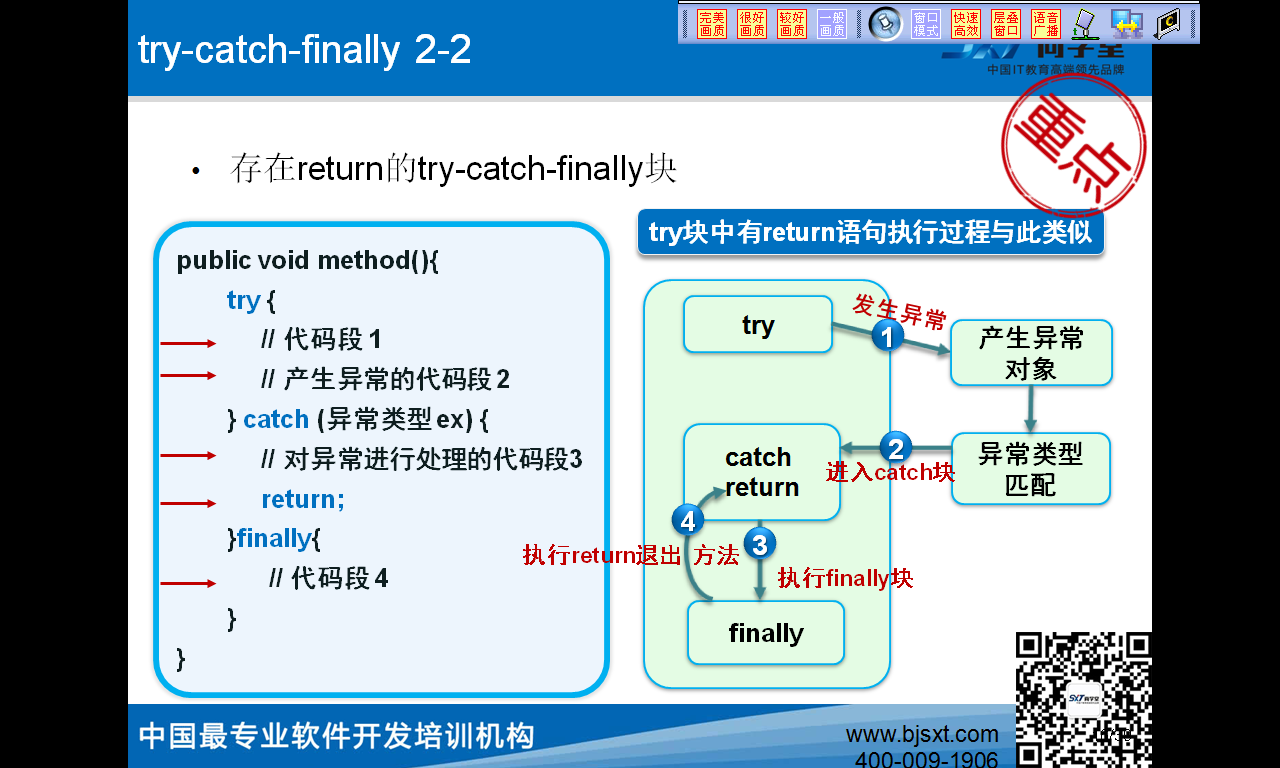
### 5、finally

**Finally:无论是否发生异常，都要执行的代码。**

**Finally唯一不执行的情况**

|  |
| --- |
| **try** {  System.*out*.println("请输入被除数:");  **int** numA=input.nextInt();  System.*out*.println("请输入除数：");  **int** numB=input.nextInt();  System.*out*.println(String.*format*("%d/%d=%d", numA,numB,numA/numB));  } **catch** (Exception e) {//算术异常  System.*out*.println("出错了!");  **System.*exit*(1); ---中断程序，退出java虚拟机**  }**finally**{  System.*out*.println("谢谢使用本程序");  } |

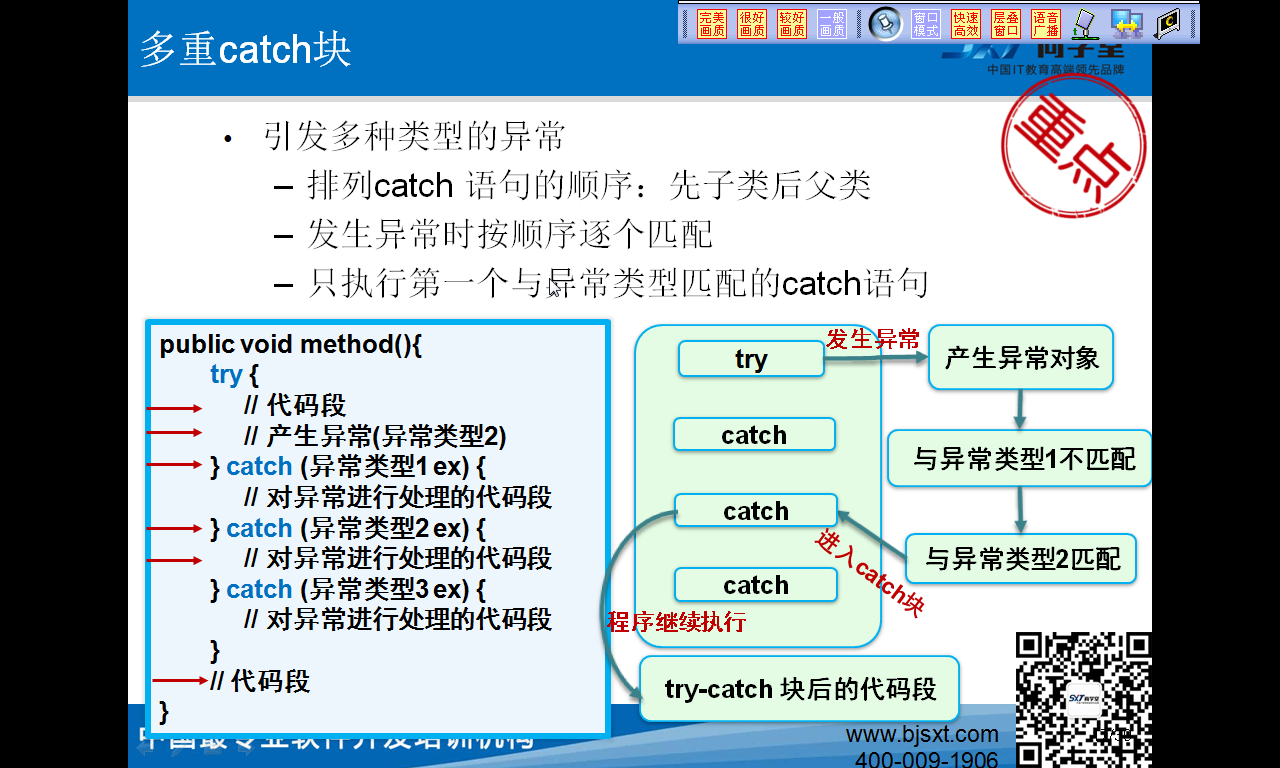
### 6、在catch中有return，程序会不会执行 finally;



代码

|  |
| --- |
| **try** {  **System.*out*.println("请输入被除数:");//[** [1]**]**  **int numA=input.nextInt();**  System.*out*.println("请输入除数：");  **int** numB=input.nextInt();  System.*out*.println(String.*format*("%d/%d=%d", numA,numB,numA/numB));  **} catch (Exception e) {//算术异常//[**[2]**]**  **System.*out*.println("出错了!");**  **return**; //[4]  }**finally**{  **System.*out*.println("谢谢使用本程序"); //[[3]]**  } |

### 7、多重异常



代码

|  |
| --- |
| **try** {  System.*out*.println("请输入被除数:");  **int** numA=input.nextInt();  System.*out*.println("请输入除数：");  **int** numB=input.nextInt();  System.*out*.println(String.*format*("%d/%d=%d", numA,numB,numA/numB));  **}catch (ArithmeticException e) {//算术异常**  **System.*out*.println("不能被0除");**  **}catch (InputMismatchException e) {**  **System.*out*.println("请输入整数");**  **}catch (Exception e) {**  **System.*out*.println("出错了！");**  }**finally**{  System.*out*.println("谢谢使用本程序");    } |

**执行顺序**，首先与第一个catch进行匹配，不匹配，再去与第二个catch进行匹配，如果所有的都不匹配就执行最后一个Exception,因为Exception是所有异常的父类

在写多重catch的时候，Exception放到最后

### 8、throws关键字

|  |
| --- |
| **public** **class** Test4 {  **public** **static** **void** div(**int** numA,**int** numB) **throws** **Exception**{  System.*out*.println(String.*format*("%d/%d=%d", numA,numB,numA/numB));  System.*out*.println("谢谢使用本程序");  }  **public** **static** **void** main(String[] args) {  **try {**  **Test4.*div*(10, 0);**  **} catch (Exception e) {**  **// TODO Auto-generated catch block**  **e.printStackTrace();**  **}**  }  /\*public static void main(String[] args) throws Exception {  Test4.div(10, 0);  }\*/  } |

在方法的定义处使用throws 声明该方法可能会产生异常(**这个异常是Exception**)， 那么该方法的调用处就必须做try-catch处理，或者继续使用throws声明

如果声明的是**RuntimeException的子类，**那个这个异常在运行时，可以处理，也可以不处理

|  |
| --- |
| **public** **class** Test5 {  **public** **static** **void** div(**int** numA,**int** numB) **throws** **InputMismatchException**{  System.*out*.println(String.*format*("%d/%d=%d", numA,numB,numA/numB));  System.*out*.println("谢谢使用本程序");  }  **public** **static** **void** main(String[] args) {  **Test5.div(10, 0);**  }  } |

**在继承中的子类父类中方法的声明时，声明异常**

**(Exception ，RuntimeException)**

**Exception:**

1. **如果父类的方法中没有使用throws声明异常，那么子类也不能声明Exception异常**

|  |
| --- |
| **public** **class** Father {  **public** **void** show() {    }  } |

**子类**

|  |
| --- |
|  |

**如果子类真的会有这样一个Exception的异常，那么要么给父类添加上异常，要么自己处理**

|  |
| --- |
| **public** **class** Father {  **public** **void** show() **throws** Exception {  }  } |

**或者**

|  |
| --- |
| **public** **class** Son **extends** Father {  @Override  **public** **void** show() {    **try** {    } **catch** (Exception e) {  // **TODO**: handle exception  }  **super**.show();  }  } |

#### （1）、RuntimeException

|  |
| --- |
| **public** **class** Father {  **public** **void** show(){    }  } |

子类

|  |
| --- |
| **public** **class** Son **extends** Father {  @Override  **public** **void** show() **throws** RuntimeException{  // **TODO** Auto-generated method stub  **super**.show();  }  } |

父类的方法如果没有声明异常，那么子类在重写方法的时候，声明的是RuntimeException或者是RuntimeException的子类的异常时，程序可以不进行处理。

### 9、throw关键字

**Exception**

|  |
| --- |
| **public** **void** setGender(String gender) **throws Exception**{  **if** ("男".equals(gender)||"女".equals(gender)) {  **this**.gender = gender;  }**else**{  **throw new Exception("性别只能是男或者女");**  }  } |

**或者**

|  |
| --- |
| **public** **void** setGender(String gender) {  **if** ("男".equals(gender)||"女".equals(gender)) {  **this**.gender = gender;  }**else**{  **try {**  **throw new Exception("性别只能是男或者女");**  **} catch (Exception e) {**  **// TODO Auto-generated catch block**  **e.printStackTrace();**  **}**  }  } |

#### （1）、RuntimeException

|  |
| --- |
| **public** **void** setGender(String gender) **throws RuntimeException** {  **if** ("男".equals(gender)||"女".equals(gender)) {  **this**.gender = gender;  }**else**{  **throw new RuntimeException("性别只能是男或者女");**  }  } |

**或者**

|  |
| --- |
| **public** **void** setGender(String gender) {  **if** ("男".equals(gender)||"女".equals(gender)) {  **this**.gender = gender;  }**else**{  **throw new RuntimeException("性别只能是男或者女");**  }  } |

**总结:**

**Throws 与throw的区别**

1. throws是声明，定义在方法名的后面(方法的外部)
2. throw是手动抛出异常，定义在方法的内部
3. 如果throw的Exception，那么必须声明或处理
4. 如果throw的RuntimeException，那么可以处理，也可以不处理

### 10、自定义异常类

自定义异常类的步骤

|  |
| --- |
|  |

自定义异常类

|  |
| --- |
| //声明一个类，继承自Exception类,  **public** **class** GenderException **extends** Exception {  **public** GenderException(String message) {  **super**(message);//调用父类带参构造方法  }  } |

如何来使用

|  |
| --- |
| **public** **void** setGender(String gender) **throws GenderException** {  **if** ("男".equals(gender)||"女".equals(gender)) {  **this**.gender = gender;  }**else**{  **throw new GenderException("性别只能是男或者女");**  }  }  **public** **static** **void** main(String[] args) {  **try** {  **new** Person().setGender("其它");  } **catch (GenderException e) {**  **System.*out*.println(e.getMessage());**  **}**  } |

### 11、小结

异常 ：运行时有可能发生，也有可能不发生的情况

三种组合形式:try…catch, try..finally, ,try..catch..finally

Finally:无论是否发生异常，都要执行的代码，只有唯一一种不执行的情况，使用

System.exit(1);

如果在catch中有return程序会怎么执行

1. 没有异常发生，执行,try-finally
2. 有异常发生，try,异常类型匹配，执行catch, 执行finally,最后执行catch中的return

异常的种类

Throwable:

Error :不能处理的，我们不管

Exception:

检查时异常🡪必须做处理的，否则编译无法通过

RuntimeException—》运行时异常，可以不处理，

在继承的关系中，父类是否声明异常，子类是否需要做处理，取决于异常的类型，

Exception就必须处理,RuntimeException可以不处理

Throws与throw:

Throws：声明，定义在方法的外边，

Throw:抛出，定义在方法的里边

自定义异常类

1. 编写类，继承Exception或RuntimeException
2. 编写构造方法

## 十、常用类与包装类

Java 为基本类型提供包装类，这使得任何接受对象的操作也可以用来操作基本类型，直接将简单类型的变量表示为一个类，在执行变量类型的相互转换时，我们会大量使用这些包装类。java 是一种面向对象语言,java 中的类把方法与数据连接在一起,并构成了自包含式的处理单元.但在 java 中不能定义基本类型(primitive type),为了能将基本类型视为对象来处理,并能连接相关的方法,java 为每个基本类型都提供了包装类,这样,我们便可以把这些基本类型转化为对象来处理了.这些包装类

有:Boolean,Byte,Short,Character,Integer,Long,Float,Void 等

值得说明的是,java 是可以直接处理基本类型的,但是在有些情况下我们需要将其作为对象来处理,这时就需要将其转化为包装类了.所有的包装类(Wrapper Class)都有共同的方法,他们是:

(1)带有基本值参数并创建包装类对象的构造函数.如可以利用 Integer 包 装

类创建对象,Integer obj=new Integer(145);

(2)带有字符串参数并创建包装类对象的构造函数.如 new Integer("-45.36");

(3)生成字符串表示法的 toString()方法,如 obj.toString().

(4)对同一个类的两个对象进行比较的 equals()方法,如 obj1.eauqls(obj2);

(5)生成哈稀表代码的 hashCode 方法,如 obj.hasCode();

(6)将字符串转换为基本值的 parseType 方法,如 Integer.parseInt(args[0]);

(7)可生成对象基本值的 typeValue 方法,如 obj.intValue();在一定的场合,运用 java 包装类来解决问题,能大大提高编程效率.

包装类的自动装箱，自动拆箱

所谓装箱，就是把基本类型用它们相对应的引用类型包起来，使它们可以具有对象的特质，如我们可以把 int 型包装成 Integer 类的对象，或者把 double包装成 Double，等等。

所谓拆箱，就是跟装箱的方向相反，将 Integer 及 Double 这样的引用类型的对象重新简化为值类型的数据

javaSE5.0 后提供了自动装箱与拆箱的功能，此功能事实上是编译器来帮您的忙，编译器在编译时期依您所编写的方法，决定是否进行装箱或拆箱动作。

自动装箱的过程：每当需要一种类型的对象时，这种基本类型就自动地封装到与它相同类型的包装中。

自动拆箱的过程：每当需要一个值时，被装箱对象中的值就被自动地提取出来，没必要再去调用 intValue()和 doubleValue()方法。

自动装箱， 只需将该值赋给一个类型包装器引用， java 会自动创建一个对象。

例如：Integer i=100;//没有通过使用 new 来显示建立，java 自动完成。

自动拆箱，只需将该对象值赋给一个基本类型即可，例如

· int i = 11;

· Integer j = i; //自动装箱

· int k = j //自动拆箱

然而在 Integer 的自动装拆箱会有些细节值得注意：

public static void main(String[] args) {

Integer a=100;

Integer b=100;

Integer c=200;

Integer d=200;

System.out.println(a==b); //1

System.out.println(a==100); //2

System.out.println(c==d); //3

System.out.println(c==200); //4

}

在 java 种，"=="是比较 object 的 reference 而不是 value，自动装箱后，abcd都是 Integer 这个 Oject，因此“==”比较的是其引用。按照常规思维，1 和 3都应该输出 false。但结果是：

true

true

false

true

结果 2 和 4， 是因为 ac 进行了自动拆箱， 因此其比较是基本数据类型的比较，就跟 int 比较时一样的，“==”在这里比较的是它们的值，而不是引用。对于结果 1，虽然比较的时候，还是比较对象的 reference,但是自动装箱时，java 在编译的时候 Integer a = 100; 被翻译成-> Integer a = Integer.valueOf(100);关键就在于这个 valueOf()的方法。

public static Integer valueOf(int i) {

final int offset = 128;

if (i >= -128 && i <= 127) { // must cache

return IntegerCache.cache[i + offset];

}

return new Integer(i);

}

private static class IntegerCache {

private IntegerCache(){}

static final Integer cache[] = new Integer[-(-128) + 127 + 1];

static {

for(int i = 0; i < cache.length; i++)

cache = new Integer(i - 128);

}

}

根据上面的 jdk 源码，java 为了提高效率，IntegerCache 类中有一个数组缓存 了值从-128 到 127 的 Integer 对象。当我们调用 Integer.valueOf（int i） 的时候， 如果 i 的值是>=-128 且<=127 时， 会直接从这个缓存中返回一个对象，否则就 new 一个 Integer 对象。

具体如下：

static final Integer cache[] = new Integer[-(-128) + 127 + 1]; //将 cache[]变成

静态

static {

for(int i = 0; i < cache.length; i++)

cache[i] = new Integer(i - 128); //初始化 cache[i]

}

这是用一个 for 循环对数组 cache 赋值，cache[255] = new Integer(255-128),也就是 newl 一个 Integer(127) ,并把引用赋值给 cache[255],好了，然后是Integer b= 127,流程基本一样，最后又到了 cache[255] = new Integer(255-128),

这一句，那我们迷糊了，这不是又 new 了一个对象 127 吗，然后把引用赋值给 cache[255]，我们比较这两个引 用（前面声明 a 的时候也有一个）,由于是不同的地址，所以肯定不会相等，应该返回 false 啊！呵呵，这么想你就错了，请注意看 for 语句给 cache[i]初始化的时候外面还一个{}呢，{}前面一个大大的 static 关键字，是静态的，那么我们就可以回想下 static 有什么特性了，只能 初始化一次，在对象间共享，也就是不同的对象共享同一个 static数据。那么当我们 Integer b = 127 的时候，并没有 new 出一个新对象来，而是共享了 a 这个对象的引用，记住，他们共享了同一个引用！！！，那么我们进行比较 a==b 时，由于是同一个对象的引用（她们在堆中的地址相同），那当然返回 true 了！！！

### 1、包装类中的Integer

类的定义

|  |
| --- |
|  |

|  |
| --- |
|  |

属性:

**最大值与最小值**

|  |
| --- |
| //[1]整型的最大值最小值  **int** max=Integer.*MAX\_VALUE*;  System.*out*.println(max);  **int** min=Integer.*MIN\_VALUE*;  System.*out*.println(min); |

构造方法

|  |
| --- |
|  |

|  |
| --- |
| //[2]构造方法  //将基本数据类型转为包装类  Integer numA=**new** Integer(21);  Integer numB=**new** Integer("21");  //再将字符串转为Integer类型的时候有一个RuntimeException  //Integer numC=new Integer("98a");  Integer numD=**new** Integer("0100101");  **//Integer numE=new Integer("0x192ef");**  System.*out*.println(numA);  System.*out*.println(numA.toString());  System.*out*.println("numB="+numB);  //System.out.println("numC="+numC);  System.*out*.println("numD="+numD);  //System.out.println("numE="+numE); |

**.toString()说明Integer类重写了父类Object的toString()方法**

方法

**[1]equals方法**

|  |
| --- |
| //equals:重写了父类Object的equals方法，用于比较对象的值是否相等  //==:用于比较内存地址(引用)  System.*out*.println("numA==numB:"+(numA==numB));  System.*out*.println("numA.equals(numB):"+(numA.equals(numB))); |

运行效果图:

|  |
| --- |
|  |

**[2]inValue:用于将Integer转为int类型**

|  |
| --- |
| **//将基本数据类型转为包装类 --》装箱**  Integer numA=**new** Integer(21);  Integer numB=**new** Integer("21");  //左边:包装类  //右边:基本数据类型  Integer numC=21;    **//intValue 将包装类型转为基本数据类型-->拆箱**  //左边:基本数据类型  //右边:numA包装类Integer类型  **int** ivalue=numA.intValue();  System.*out*.println(ivalue); |

**[3]与字符串类型之间的转换**

|  |
| --- |
| //[4]与String类型转为Integer类型。先转为int类型。int装箱为Integer类型  Integer numD=Integer.*parseInt*("21");  //将Integer转为String类型  System.*out*.println(numD.toString());//实例方法  System.*out*.println(numD+"");  System.*out*.println(Integer.*toString*(21));//类方法  System.*out*.println(Integer.*toBinaryString*(21));//转二进制  System.*out*.println(Integer.*toHexString*(21));//转十六进制  System.*out*.println(Integer.*toOctalString*(21));//转八进制 |

**[4]valueOf方法**

|  |
| --- |
| Integer numE=Integer.*valueOf*(21);  Integer numF=Integer.*valueOf*("21"); |

**小结：**

**从基本数据类型转为包装类型有几种方法**

|  |
| --- |
| **//使用构造方法**  Integer numA=**new** Integer(21);  Integer numB=**new** Integer("21");  **//直接转换**  Integer numC=21;  **//使用方法**  Integer numE=Integer.*valueOf*(21);  Integer numF=Integer.*valueOf*("21"); |

**新特性**

|  |
| --- |
| Integer a=127;  Integer b=127;  System.*out*.println("a==b:"+(a==b));  Integer aa=128;  Integer bb=128;  System.*out*.println("aa==bb:"+(aa==bb)); |

**运行效果:**

|  |
| --- |
|  |

**说明:**因为类Byte的最大值是127,所在新的特征中，127以内Integer就不开辟新的空间了，如果越 过127之后将开避新的空间保存数据

课堂练习

**包装类中的Double**

[1]求出Double中的最大值与最小值  **--属性**

[2]使用构造方法来构造一个Double类型的对象 **--构造方法**

[3]与字符串之间的相互相转 **--方法**

### 2、字符串

#### 1、string

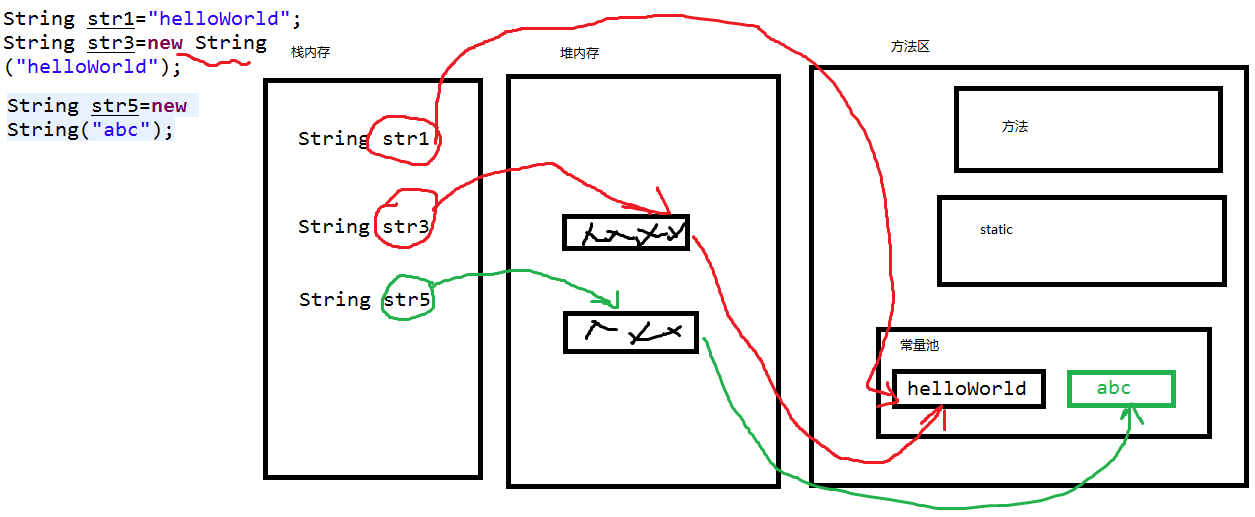
**类的结构**

|  |
| --- |
|  |

|  |
| --- |
|  |

**String是不能有子类的,是可以被序列化的**

**内存分析图**



内存图说明了:

字符串存在常量池中，而且字符串是常量，一旦创建其值**不能改变**，但是其值可以被**共享，**

字符串相当于char类型的数组

|  |
| --- |
| String str = "abc";  等效于：  char data[] = {'a', 'b', 'c'};  String str = **new String(data);** |

构造方法

有N多个重载的构造方法

String的常用方法

|  |
| --- |
| **//[1]length方法，求字符串中字符的个数**  System.*out*.println(str1.length());  **//[2]charAt(?)，求指定位置上的字符**  System.*out*.println(str1.charAt(0));  System.*out*.println(str1.charAt(str1.length()-1));  **//[3]equals方法，用于比较字符串的内容是否相同,重写了父类的Object中的equals方法**  //,==用于比较内存的地址  System.*out*.println("str1.equals(str3):"+(str1.equals(str3)));  System.*out*.println("str1==str3:"+(str1==str3)); |

**字符串的连接 concat方法**

|  |
| --- |
| //[4]concat字符串的拼接  String str1="abc";  **str1.concat("bdc");**  System.*out*.println(str1);  **String s=str1.concat("bdc");**//将产生一个新的字符串对象  **str1=str1.concat("");/**/返回的是原对象,因为参数的长度为0  System.*out*.println(s); |

**运行效果图**

|  |
| --- |
|  |

**说明:**红色的代码，连接后，并没有赋给新的对象所引用，所以str1还是原来的值

蓝色的代码，连接后，产生一个新的对象

橙色的代码，参数的长度为0，返回的是原对象

**字符串的连接🡪”+”**

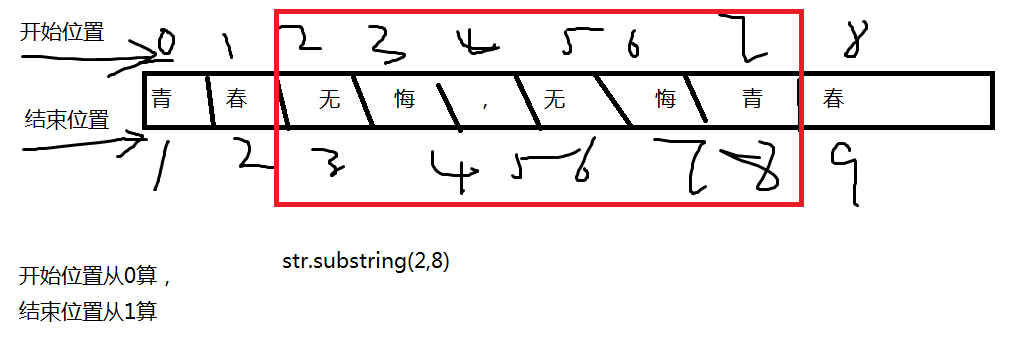
**字符串的查找 :indexOf.. lastIndexOf**

|  |
| --- |
| [5]indexOf查找指定字符/字符串在字符串中的位置  String str="String.java.txt";  **int** index1=str.indexOf('a');//char可以自动转int  **int** index2=str.indexOf(97);  **int** index3=str.indexOf("@");  **int** index4=str.indexOf("a", 12);//从第12个位置开始查  **int** index5=str.indexOf('a',18);  System.*out*.println("index1="+index1+"\tindex2="+  index2+"\tindex3="+index3+"\tindex4="+index4+"" +  "\tindex5="+index5);  //[6]lastIndexOf()查找指定字符/字符串在字符串中最后出现的位置  **int** num1=str.lastIndexOf('a');  System.*out*.println("num1="+num1); |

**字符串的截取substring**

|  |
| --- |
| String str1="青春无悔,无悔青春";  System.*out*.println(str1.substring(2,8));  System.*out*.println(str1.substring(2)); |

**分析图**



**字符串的截取与去空格 substring ,trim**

|  |
| --- |
| String str1="青春无悔,无悔青春";  System.*out*.println(str1.substring(2,8));  System.*out*.println(str1.substring(2));  String str2=" 哈 哈 ";  System.*out*.println("字符串长度:"+str2.length());  str2.trim(); //因为没有赋给对象所引用  System.*out*.println("字符串长度:"+str2.length());  str2=str2.trim();//只去字符串前后的空格  System.*out*.println("字符串长度:"+str2.length()); |

**字符串的拆分—split,结果是String类型的数组**

|  |
| --- |
| Scanner input=**new** Scanner(System.*in*);  System.*out*.println("请输入一首歌，以空格分隔");  String str=input.next();  //String str=input.nextLine();  String [] s=str.split(" ");  **for**(String st:s){  System.*out*.println(st);  } |

**其它常用方法**

|  |
| --- |
| String str="青春无悔，无悔青春highHE";  **char**[] c=str.toCharArray();//转char类型的数组  System.*out*.println(str.toUpperCase());//转大写  System.*out*.println(str.toLowerCase());//转小写  System.*out*.println(str.replace("无悔", "有悔"));//String是 CharSequence的实现类 |

#### 2、StringBuffer:

1. String的加强版，在处理字符串的连接过程中，效率比String高，因为追加字符串不会产生新的对象，只是有原有的对象的值的基础上继续追加
2. StringBuffer是一个容器，就是用来存放数据的
3. 线程安全的可变的字符串

|  |
| --- |
| StringBuffer sb = **new** StringBuffer("青春无悔");  **int** num=110;  StringBuffer sb1 = sb.append("我心永恒");  System.*out*.println(sb1);  StringBuffer sb2 = sb1.append('啊');  System.*out*.println(sb2);  StringBuffer sb3 = sb2.append(num);  System.*out*.println(sb3);  System.*out*.println("sb==sb1:"+(sb==sb1));  System.*out*.println("sb==sb2:"+(sb==sb2));  System.*out*.println("sb==sb3:"+(sb==sb3)); |

#### 3、StringBuilder

**与StringBuffer的使用方法相同，线程不安全的，单线程操作时比StringBuffer速度快**

#### 4、日期与时间

|  |
| --- |
| Date d=**new** Date();//创建一个日期对象,得到当前的日期  //System.out.println(d);  //System.out.println(d.getYear());//加横线了，说明这个方法已过时    //Calendar rightNow = Calendar.getInstance();  //父类指向子类对象  Calendar calendar=**new** GregorianCalendar();  System.*out*.println("YEAR: " + calendar.get(Calendar.*YEAR*));  System.*out*.println("MONTH: " +( calendar.get(Calendar.*MONTH*)+1));  System.*out*.println("DAY\_OF\_MONTH: " + calendar.get(Calendar.*DAY\_OF\_MONTH*));    System.*out*.println("HOUR\_OF\_DAY: " + calendar.get(Calendar.*HOUR\_OF\_DAY*));  System.*out*.println("MINUTE: " + calendar.get(Calendar.*MINUTE*));  System.*out*.println("SECOND: " + calendar.get(Calendar.*SECOND*));  System.*out*.println("MILLISECOND: " + calendar.get(Calendar.*MILLISECOND*));  StringBuffer sb=**new** StringBuffer();  sb.append(calendar.get(Calendar.*YEAR*));  sb.append("年");  sb.append(calendar.get(Calendar.*MONTH*)+1);  sb.append("月 ");  sb.append(calendar.get(Calendar.*DAY\_OF\_MONTH*)).append("日 ");  sb.append(calendar.get(Calendar.*HOUR\_OF\_DAY*)).append("时");  sb.append(calendar.get(Calendar.*MINUTE*)).append("分");  sb.append(calendar.get(Calendar.*SECOND*)).append("秒");  sb.append(calendar.get(Calendar.*MILLISECOND*)).append("毫秒");    System.*out*.println(sb.toString());  DateFormat df=DateFormat.*getDateInstance*();    System.*out*.println(df.format(d));    SimpleDateFormat sdf=**new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss.sss");  System.*out*.println(sdf.format(d)); |

**总结:**Date 用于创建一个日期时间对象

Calendar:想得到单个的年，月，日时使用比较方便

DateFormat:可以进行格式化日期，得到简单的形式

SimpleDateFormat:是DateFormat的子类，可以根据用户的需要进行不同的格式化

#### 5、Math类

|  |
| --- |
| //英文天花板,向上取大值  System.*out*.println(Math.*ceil*(90.00001));  System.*out*.println(Math.*ceil*(-90.999999));  //英文地板，向下取小值  System.*out*.println(Math.*floor*(-90.0000001));  System.*out*.println(Math.*floor*(90.999999));  //中间，四舍五入  System.*out*.println(Math.*round*(89.4900));  System.*out*.println(Math.*round*(89.54)); |

#### 6、枚举

枚举存在一组有限的值，都是public static final，默认的，不可以手动添加

枚举的结尾分号可以省略不写

枚举的使用, 枚举名.值

枚举的定义的关键字: enum

**枚举的定义:**

|  |
| --- |
| **public** **enum** Gender {//枚举不是类，但枚举是一种数据类型，引用数据类型，隐式继承自Enum  *男*,*女*  } |

**将字符串转枚举**

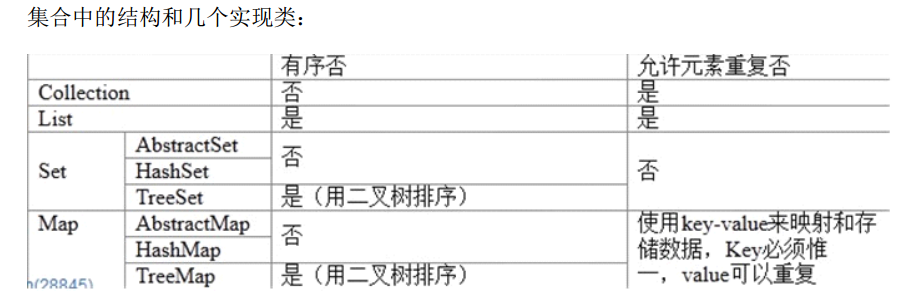
|  |
| --- |
| System.*out*.println("请输入您的性别:");  String g=input.next();//字符串  //将字符转枚举  Gender gender=Enum.*valueOf*(**Gender.class**, g); |

#### 7、file

案例

|  |
| --- |
| package cn.bjsxt.demo;  import java.io.File;  import java.io.IOException;  public class FileDemo {  public static void main(String[] args) throws IOException {  File file=new File("E:/bjsxt/java.txt");//文件对象创建完毕  //File file=new File("java.txt");//文件对象创建完毕  if (!file.exists()) {//返回值是true或者是false,存在为true,不存在为false  file.createNewFile();//创建文件的，但不能创建目录  System.out.println("创建成功");  System.out.println("==================================================");  System.out.println("相对路径:"+file.getPath());  System.out.println("绝对路径:"+file.getAbsolutePath());  System.out.println("文件:"+file.getName());  file.delete();//直接从硬盘上进行删除，而没有经过回收站  System.out.println("删除成功");  }else{  System.out.println("文件已存在!");  }  }  } |
|  |
| package cn.bjsxt.demo;  import java.io.File;  import java.io.IOException;  public class FileDemo2 {  public static void main(String[] args) {  //列出一个给定路径下的所有文件，如果有子文件有文件，继续显示  File my=new File("E:/bjsxt102");//传入一个路径  print(my);  }  //递归调用  public static void print(File file){//输出的方法,传入是File类的对象  if (file!=null) {//判断对象是否为空  if (file.isDirectory()) {//判断是否是目录  //为什么不使用list()方法，因为list方法得到的是String类型的数组，而我们需要使用递归，自己调用自己  //参数要求是File的对象，所以需要使用listFiles()方法得到File的对象数组  File [] f=file.listFiles();//得到是文件和目录的数组  for (int i = 0; i < f.length; i++) {  print(f[i]);//调用本身  }  }else{//不是目录，就是文件  System.out.println(file);//将文件输出  }  }  }  } |

## 容器



总述：List、Set、Map 是这个集合体系中最主要的三个接口。

其中 List 和 Set 继承自 Collection 接口。

Set 不允许元素重复。HashSet 和 TreeSet 是两个主要的实现类。

List 有序且允许元素重复。 ArrayList、 LinkedList 和 Vector 是三个主要的实现类。

Map 也属于集合系统， 但和 Collection 接口不同。 Map 是 key 对 value 的映射集合，其中 key 列就是一个集合。key 不能重复，但是 value 可以重复。

HashMap、TreeMap 和 Hashtable 是三个主要的实现类。

SortedSet 和 SortedMap 接口对元素按指定规则排序，SortedMap 是对 key列进行排序。

### 、collection

1.Collection 接口用于表示任何对象或元素组。想要尽可能以常规方式处理一组元素时，就使用这一接口。

操作：

#### (1) 单元素添加、删除操作：

boolean add(Object o):将对象添加给集合

boolean remove(Object o): 如果集合中有与 o 相匹配的对象，则删除对象 o

#### (2) 查询操作：

int size() ：返回当前集合中元素的数量

boolean isEmpty() ：判断集合中是否有任何元素

boolean contains(Object o) ：查找集合中是否含有对象 o

Iterator iterator() ：返回一个迭代器，用来访问集合中的各个元素

#### (3) 组操作 ：作用于元素组或整个集合

boolean containsAll(Collection c): 查找集合中是否含有集合 c 中所有元素

boolean addAll(Collection c) : 将集合 c 中所有元素添加给该集合

void clear(): 删除集合中所有元素

void removeAll(Collection c) : 从集合中删除集合 c 中的所有元素

void retainAll(Collection c) : 从集合中删除集合 c 中不包含的元素

#### (4) Collection 转换为 Object 数组 ：

Object[] toArray() ：返回一个内含集合所有元素的 arrayObject[] toArray(Object[] a) ： 返回一个内含集合所有元素的 array。 运行期返回的 array 和参数 a 的型别相同，需要转换为正确型别。

此外，您还可以把集合转换成其它任何其它的对象数组。但是，您不能直接把集合转换成基本数据类型的数组，因为集合必须持有对象。“斜体接口方法是可选的。 因为一个接口实现必须实现所有接口方法， 调用程序就需要一种途径来知道一个可选的方法是不是不受支持。如果调用一种可选方法 时，一个 UnsupportedOperationException 被抛出，则操作失败， 因为方法不受支持。此异常类继承 RuntimeException 类，避免了将所有集合操作放入 try-catch 块。”

Collection 不提供 get()方法。 如果要遍历 Collectin 中的元素， 就必须用 Iterator。

1、如何对与Collection相关的容器时行遍历？

1) 方法: 凡是与Collection接口相关的容器均有一个 iterator()方法，此方法用来反馈一个迭代器 (Iterator接口来管理)。

2) 新增的for循环

容器中的所有元素均为变为: Object类型

容器===>>可变的数组

|  |
| --- |
| //遍历  **public** **static** **void** main(String[] args) {  Collection c1 = **new** HashSet();  c1.add( (**byte**)127 );  c1.add( (**short**)32767 );  c1.add( 2147483647 );  c1.add( 9876543210L );  c1.add( 1.78f );  c1.add( Math.*PI* );  c1.add( **true** );  c1.add( 'A' );  c1.add( "Hello" );  System.*out*.println( "c1>>> " + c1 + "，共有: " + c1.size() + " 个元素。");  System.*out*.println("对以上容器进行遍历如下：");  //获取迭代器  Iterator it = c1.iterator();  //判断遍历完了吗  **while** ( it.hasNext() ){  Object elem = it.next() ; //访问元素  **if**( elem **instanceof** Byte || elem **instanceof** Short ){  it.remove(); //借助迭代器实现容器中元素的删除。  }  System.*out*.println( elem ); //显示  } //反复    System.*out*.println( "c1>>> " + c1 + "，共有: " + c1.size() + " 个元素。");  System.*out*.println("\n用新增的for循环遍历以上容器的元素如下：");  **for**(Object tem : c1 ){  System.*out*.println( tem );  }  } |

### 、set

#### 1、set

Set 接 口也是 Collection 的一种扩展，而与 List 不同的时，在 Set 中的对象元素不能重复，也就是说你不能把同样的东西两次放入同一个 Set 容器中。 它的常用具体实现有 HashSet 和 TreeSet 类。HashSet 能快速定位一个元素，但是你放到 HashSet 中的对象需要实现 hashCode()方法，它使用了前面说过的哈希码的算法。而 TreeSet 则将放入其中的元素按序存放，这就要求你放入其中的对象是可排序的， 这就用 到了集合框架提供的另外两个实用类Comparable 和 Comparator。一个类是可排序的，它就应该实现 Comparable接口。有时多个类具 有相同的排序算法，那就不需要在每分别重复定义相同的排序算法，只要实现 Comparator 接口即可。集合框架中还有两个很实用的公用 类： Collections 和 Arrays。 Collections 提供了对一个 Collection 容器进行

诸如排序、复制、查找和填充等一些非常有用 的方法，Arrays 则是对一个数组进行类似的操作。

#### 2、Comparable 接口

在“集合框架”中有两种比较接口：Comparable 接口和 Comparator 接口。像String 和 Integer 等 Java 内建类实现 Comparable 接口以提供一定排序方式，但这样只能实现该接口一次。 对于那些没有实现 Comparable 接口的类、 或者自定义的类，您可以通过 Comparator 接口来定义您自己的比较方式。

Comparable 接口

在 java.lang 包中，Comparable 接口适用于一个类有自然顺序的时候。假定对象集合是同一类型，该接口允许您把集合排序成自然顺序。

(1) int compareTo(Object o): 比较当前实例对象与对象 o，如果位于对象 o之前，返回负值，如果两个对象在排序中位置相同，则返回 0，如果位于对象 o 后面，则返回正值在 Java 2 SDK 版本 1.4 中有二十四个类实现 Comparable 接口。 下表展示了 8

种基本类型的自然排序。虽然一些类共享同一种自然排序，但只有相互可比的类才能排序。

利用 Comparable 接口创建您自己的类的排序顺序，只是实现 compareTo()方



法的问题。 通常就是依赖几个数据成员的自然排序。 同时类也应该覆盖 equals()

和 hashCode()以确保两个相等的对象返回同一个哈希码。

#### 3、SortedSet 接口

“集合框架”提供了个特殊的 Set 接口：SortedSet，它保持元素的有序顺序。

SortedSet 接口为集的视图(子集)和它的两端 （即头和尾） 提供了访问方法。当您处理列表的子集时，更改视图会反映到源集。此外，更改源集也会反映在子集上。发生这种情况的原因在于视图由两端的元素而不是下标元素 指定，所以如果您想要一个特殊的高端元素（toElement）在子集中，您必须找到下一个元素。添加到 SortedSet 实现类的元素必须实现 Comparable 接口，否则您必须给它的构造函数提供一个 Comparator 接口的实现。TreeSet 类是它的唯一一份实现。

“因为集必须包含唯一的项，如果添加元素时比较两个元素导致了 0 返回值（通过 Comparable 的 compareTo()方法或 Comparator 的 compare()方法）， 那么新元素就没有添加进去。如果两个元素相等，那还好。但如果它们不相等的话， 您接下来就应该修改比较方法， 让比较方法和 equals() 的效果一致。 ”

(1) Comparator comparator(): 返回对元素进行排序时使用的比较器， 如果使用 Comparable 接口的 compareTo()方法对元素进行比较，则返回 null

(2) Object first(): 返回有序集合中第一个(最低)元素

(3) Object last(): 返回有序集合中最后一个(最高)元素

(4)SortedSet subSet(Object fromElement, Object toElement): 返回从fromElement(包括)至 toElement(不包括)范围内元素的 SortedSet 视图(子集)

(5) SortedSet headSet(Object toElement): 返回 SortedSet 的一个视图， 其内各元素皆小于 toElement

(6) SortedSet tailSet(Object fromElement): 返回 SortedSet 的一个视图， 其内各元素皆大于或等于 fromElement

#### 4、HashSet 类

(1) HashSet(): 构建一个空的哈希集

(2) HashSet(Collection c): 构建一个哈希集，并且添加集合 c 中所有元素

(3) HashSet(int initialCapacity): 构建一个拥有特定容量的空哈希集

(4) HashSet(int initialCapacity, float loadFactor): 构建一个拥有特定容量和加载因子的空哈希集。LoadFactor 是 0.0 至 1.0 之间的一个数

#### 5、TreeSet 类

(1) TreeSet():构建一个空的树集

(2) TreeSet(Collection c): 构建一个树集，并且添加集合 c 中所有元素

(3) TreeSet(Comparator c): 构建一个树集， 并且使用特定的比较器对其元素进行排序

“comparator 比较器没有任何数据，它只是比较方法的存放器。这种对象有时称为函数对象。函数对象通常在“运行过程中”被定义为匿名内部类的一个实例。”

TreeSet(SortedSet s): 构建一个树集，添加有序集合 s 中所有元素，并且使用与有序集合 s 相同的比较器排序

1、Set接口的特性及其实现类:

1) 元素无序的。

2) 元素不重复。当添加重复的元素时，它采用屏蔽技术屏蔽掉。

3) 它的常用实现类: HashSet.

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Set s1 = **new** HashSet();    s1.add("a1");  s1.add("a5");  s1.add("a321");  s1.add("a2");  s1.add("a6");  s1.add("a231");  s1.add("a4");  s1.add("a3");  s1.add("a7");  s1.add("a123");    System.*out*.println("s1>>> " + s1 + " <<<注意: 元素是否有序？");    s1.add("a6");  **boolean** flag = s1.add("a6");  System.*out*.println("s1>>> " + s1 + " <<<注意: 元素是否重复？");  System.*out*.println( flag );  } |

2、HashSet实现类的特性:

1) 元素无序的。

2) 元素不重复。当添加重复的元素时，它采用屏蔽技术屏蔽掉。

3) 个性:

判断元素（对象）相等依赖equals()方法。同时，它还依赖hashCode()方法。

在Java中，规定: 任何一个对象均有一个 哈希编码值，它是一个int型整数，由系统自动生成。

规定: 相对象的对象，应该拥有相同的哈希编码。

结论:

凡是自定义类的对象，要判断相等，则该类必须重写 equals()方法。

结论:

凡是用HashSet实现类所构建的容器，且用来存放自定义类的对象时，则要求该类必须重写equals()和hashCode()两个方法。

|  |
| --- |
| **public** **static** **void** main(String[] args) {  // String s1 = new String("aBC");  // String s2 = new String("ABC");  //  // boolean flag = s1.equals(s2);  // if( flag ){  // System.out.println("s1和s2所引用的对象是相等的。");  // }else{  // System.out.println("s1和s2所引用的对象不相等。");  // }  //  // flag = s1 == s2 ;  // if( flag ){  // System.out.println( "s1和s2是同一个对象。");  // }else{  // System.out.println("s1和s2不是同一个对象。");  // }  //  // System.out.println( s1 + "，的哈希编码值是: " + s1.hashCode() );  // System.out.println( s2 + "，的哈希编码值是: " + s2.hashCode() );  //  // Date d1 = new Date();  // Date d2 = new Date();  //  // flag = d1.equals(d2);  // System.out.println( flag );  // System.out.println( "d1的哈希编码是: " + d1.hashCode() );  // System.out.println( "d2的哈希编码是: " + d2.hashCode() );    Set ss = **new** HashSet();  Flower f1 = **new** Flower("rose", 10) ;  Flower f2 = **new** Flower("rose", 10) ;    ss.add( f1 );  ss.add( f2 );      System.*out*.println( ss );  }  }  **class** Flower {  **private** String name;  **private** **double** price;    **public** Flower(String name, **double** price ) {  **this**.name = name;  **this**.price = price;  }    //  //    @Override  **public** **boolean** equals(Object obj) {  Flower f2 = (Flower)obj;  **if**( **this**.name.equals(f2.name) && **this**.price == f2.price ){  **return** **true**;  }**else**{  **return** **false**;  }  }    @Override  **public** String toString() {  **return** "\n花名: " + name + " 价格: " + price + " 元/支.";  }    // @Override  // public int hashCode() {  // return 989898;  // }    @Override  **public** **int** hashCode() {  Double d1 = **this**.price;  **return** **this**.name.hashCode() + d1.hashCode() ;  } |

3、treeset

1、SortedSet接口的特性及其实现类:

1) 元素自动排序(按大小).

2) 元素不重复。当添加重复的元素时，也会采用屏蔽技术屏蔽掉。

3) 它的实现类: TreeSet.

注意:

凡是排序，就必须比较大小；因此，若要比较大小，请问不同种类的数据能比较大小吗？

结论: 凡是会自动排序的容器，则其中存放的元素必须是同种类的。

|  |
| --- |
| **public** **static** **void** main(String[] args) {  SortedSet s1 = **new** TreeSet();    s1.add("a5"); // "a5" 与 "a1" 比较大小 ? 　　Unicode值 '5' 53 '1' 49  s1.add("a19");  s1.add("a321");  s1.add("a2");  s1.add("a6");  s1.add("a231");  s1.add("a4");  s1.add("a3");  s1.add("a7");  s1.add("a123");    //s1.add( (byte)127 );  // s1.add( (short)32767 );  //s1.add( 2147483647 );  //s1.add( 'A' );  //  System.*out*.println( "s1>>> " + s1 + "<<<<注意：元素是否有序？");    s1.add( "a6");  System.*out*.println("s1>>> " + s1 );  } |

2、TreeSet实现类的特性:

1) 元素自动排序(按大小).

2) 元素不重复。当添加重复的元素时，也会采用屏蔽技术屏蔽掉。

3) 个性:

判断元素相等依赖 equals()方法。但，它实质上依赖 compareTo()方法。

注意:

要排序，就必须比较大小，要比较大小，就必须有比较大小的条件。

结论:

凡是对象要比较大小，就必须有compareTo()方法。即: 该类必须实现　java.lang.Comparable接口，重写

compareTo()方法。

结论:

当容器由 TreeSet实现类所构建时，且用来存放自定义类的对象，则要求该类必须实现　Comparable接口，重写

compareTo()方法，为对象比较大小提供条件。

同时，

注意: 在重写compareTo()方法时，要对对象的所有属性一一比较大小。

|  |
| --- |
| **public** **static** **void** main(String[] args) {  SortedSet ss = **new** TreeSet();  Dog d1 = **new** Dog("AAA", **true**, 2);  Dog d2 = **new** Dog("CCC", **false**, 1);    **int** x = d1.compareTo(d2);    ss.add( d1 );  ss.add( d2 );  ss.add( **new** Dog("BBB", **true**, 4) );  ss.add( **new** Dog("EEE", **false**, 3) );  ss.add( **new** Dog("DDD", **false**, 5) );    System.*out*.println( "ss>>> " + ss );    ss.add( **new** Dog("DDD", **true**, 5) );  System.*out*.println( "ss>>> " + ss );  }  }  //自定义类  **class** Dog **implements** Comparable {    //重写此方法的目的是: 为对象比较大小提供依据。  **public** **int** compareTo(Object o ) {  Dog d2 = (Dog)o;    **int** x = **this**.name.compareTo( d2.name ) ;  Boolean b1 = **this**.sex;  Boolean b2 = d2.sex;  **int** y = b1.compareTo(b2); //(this.sex+"").compareTo( d2.sex+"" ) ;  **int** z = **this**.age - d2.age ;    **if**( x != 0 ){  **return** x;  }**else** **if**( y != 0 ){  **return** y;  }**else**{  **return** z;  }  }    **private** String name;  **private** **boolean** sex;  **private** **int** age;    **public** Dog() {  }  **public** Dog(String name, **boolean** sex, **int** age) {  **this**.name = name;  **this**.sex = sex;  **this**.age = age;  }    //  //    @Override  **public** String toString() {  **return** "\n" + name + " " + (sex ? "公":"母") + " " + age;  } |

### 、List

#### 1、list

List 接口对 Collection 进行了简单的扩充，它的具体实现类常用的有ArrayList 和 LinkedList。你可以将任何东西放到一个 List 容器中，并在需要时从中取出。ArrayList 从其命名中可以看出它是一种类似数组的形式进行存储，因此它的随机访问速度极快，而 LinkedList 的内部实现是链表，它适合于在链表中间需要频繁进行插入和删除操作。在具体应用时可以根据需要自由选择。前面说的 Iterator 只 能对容器进行向前遍历，而 ListIterator则继承了 Iterator 的思想，并提供了对 List 进行双向遍历的方法。

(1) 面向位置的操作包括插入某个元素或 Collection 的功能，还包括获取、 除去或更改元素的功能。在 List 中搜索元素可以从列表的头部或尾部开始，如果找到元素，还将报告元素所在的位置 :

void add(int index, Object element): 在指定位置 index 上添加元素 element

boolean addAll(int index, Collection c): 将集合 c 的所有元素添加到指定位置index

Object get(int index): 返回 List 中指定位置的元素

int indexOf(Object o): 返回第一个出现元素 o 的位置，否则返回-1

int lastIndexOf(Object o) ：返回最后一个出现元素 o 的位置，否则返回-1

Object remove(int index) ：删除指定位置上的元素

Object set(int index, Object element) ：用元素 element 取代位置 index 上的元素，并且返回旧的元素

(2) List 接口不但以位置序列迭代的遍历整个列表，还能处理集合的子集：

ListIterator listIterator() : 返回一个列表迭代器，用来访问列表中的元素

ListIterator listIterator(int index) : 返回一个列表迭代器，用来从指定位置index 开始访问列表中的元素

List subList(int fromIndex, int toIndex) ：返回从指定位置 fromIndex（包含）到 toIndex（不包含）范围中各个元素的列表视图

“对子列表的更改（如 add()、remove() 和 set() 调用）对底层 List 也有影响。” “ArrayList 和 LinkedList 都实现 Cloneable 接口，都提供了两个构造函数，一个无参的，一个接受另一个 Collection”

#### 2、LinkedList 类

LinkedList 类添加了一些处理列表两端元素的方法。

(1) void addFirst(Object o): 将对象 o 添加到列表的开头

void addLast(Object o)：将对象 o 添加到列表的结尾

(2) Object getFirst(): 返回列表开头的元素

Object getLast(): 返回列表结尾的元素

(3) Object removeFirst(): 删除并且返回列表开头的元素

Object removeLast():删除并且返回列表结尾的元素

(4) LinkedList(): 构建一个空的链接列表

LinkedList(Collection c): 构建一个链接列表，并且添加集合 c 的所有元素“使用这些新方法，您就可以轻松的把 LinkedList 当作一个堆栈、队列或其它面向端点的数据结构。”

#### 3、ArrayList 类

ArrayList 类封装了一个动态再分配的 Object[]数组。每个 ArrayList 对象有一个 capacity。 这个 capacity 表示存储列表中元素的数组的容量。 当元素添加到ArrayList 时，它的 capacity 在常量时间内自动增加。

在向一个 ArrayList 对象添加大量元素的程序中， 可使用 ensureCapacity 方法增加 capacity。这可以减少增加重分配的数量。

(1) void ensureCapacity(int minCapacity): 将 ArrayList 对象容量增加

minCapacity

(2) void trimToSize(): 整理 ArrayList 对象容量为列表当前大小。程序可使用

这个操作减少 ArrayList 对象存储空间。

1、List接口的特性及其常用的实现类:

1) 元素在位置上有序(即: 每个元素拥有一个位置编号，它从0开始。)

2) 它比较父接口(Collection)多了一些与位置操作的方法。

插入，获取某个位置上的元素，删除某个位置上的元素。 查询。

3) 元素可以重复。因此，重复的元素可以在不同的位置上。

4) 它的常用实现类: ArrayList 和 LinkedList 。

|  |
| --- |
| **public** **static** **void** main(String[] args) {  List alist = **new** ArrayList();    alist.add("A1");  alist.add("A6");  alist.add("A2");  alist.add("A5");  alist.add("A4");  alist.add("A3");  alist.add("A7");  System.*out*.println("alist>>> " + alist + " <<<注意: 显示的顺序与添加时的顺序有何关联？");    alist.add("A6");  alist.add("A6");  System.*out*.println("alist>>> " + alist + " <<<注意: 元素重复？");    alist.add(0, 365); //插入  System.*out*.println("alist>>> " + alist );    **int** n = (**int**)( alist.size() \* Math.*random*() );  Object obj = alist.get( n ); //获取  System.*out*.println( "第" + n + "个位置上的元素是: " + obj );    alist.remove( n ); //删除第n个位置上的元素  alist.remove("A456");  System.*out*.println( alist );    //思考: 当容器中存放的是Integer类的对象时，则 alist.remove(5) 且 容器10个元素且有 5这个元素。;    alist.set(0, "China");  System.*out*.println( alist );    //在alist容器中查找 "A6"这个元素。若存在，则反馈它的位置；反之，反馈一个负数。  **int** loc = alist.indexOf( "A6" ); //顺序查找  **if**( loc >= 0 ){  System.*out*.println("找到了，它在第" + loc + " 个位置上。");  }**else**{  System.*out*.println("找不到，结果是: " + loc );  }    loc = alist.lastIndexOf( "A6" ); //查找，并反馈它最后出现的位置；反之，反馈一个负数。  System.*out*.println( loc );    //截取部分  List subAlist = alist.subList(5, alist.size()); //将alist容器中的第5至末尾的所有元素收集起来并构成一个新容器。  System.*out*.println("\nalist>>> " + alist );  System.*out*.println("\nsubAlist>>> " + subAlist );      } |

2、ArrayList 和 LinkedList实现类的特性:

1) 元素在位置上有序(即: 每个元素拥有一个位置编号，它从0开始。)

2) 元素可以重复。因此，重复的元素可以在不同的位置上。

3) ArrayList采用队列的原理来管理元素。

因此，它在查找元素时，效率较高；而在删除和插入元素时效率较低。

LinkedList采用链表的原理来管理元素。

因此，它在查找时，效率较低；　而在插入和删除元素时效率较高。

|  |
| --- |
| **public** **static** **void** main(String[] args) {  List alist = **new** LinkedList(); //ArrayList();    **long** t1 = **new** Date().getTime();    **for**(**int** i = 0; i < 100000; i++ ){  alist.add(0, "A"+i);    **if**( (i+1)%30 == 0 ){  alist.add(0, "\n");  }  }    **long** t2 = System.*currentTimeMillis*();    System.*out*.println("alist>>> " + alist );  System.*out*.println("消耗的时间: " + (t2-t1) + "毫秒。");  } |

|  |
| --- |
| package com.bjxt.oop.d21;  /\*\*  \* //思考: 当容器中存放的是Integer类的对象时，则 alist.remove(5) 且 容器10个元素且有 5这个元素。;  \* 为什么 是移除的是第5个位置的元素 而不是移除的5这个元素  \* 这种情况下 要移除第5个元素 怎么做  \*/  public class QuTest {  public static void main(String[] args) { //对象　与 对象变量 是两个东西  Father f1 = new Father("rose", 10) ;  Father f2 = f1; // new Father("rose", 10) ;    boolean tt = f1 == f2; //判断f1变量中的值 与 f2变量中的值相等吗    System.out.println( "f1.hashCode :" + f1.hashCode() );//哈希编码是地址的10进制，重写hashcode()方法  System.out.println( "f2.hashCode :" + f2.hashCode() );  //对地址有没有影响？地址是根据哈希编码得到，还是哈希编码根据地址得到？    System.out.println("f1的地址是:"+f1.toString());  System.out.println("f1的地址是:"+f2);    Object obj;    System.out.println("f1==f2??:"+(f1==f2));  //重写hashcode()方法后 两对象的地址相同，但f1!=f2？"=="究竟比较什么？  System.out.println("fi.equals(f2)??:"+f1.equals(f2));  }  }  class Father{  private String name; //"rose" h = 1298707; hash = h;  private double price;  public Father(String name, double price ) {  this.name = name;  this.price = price;  }      public int hashCode() {  Double d1 = this.price;  return this.name.hashCode() + d1.hashCode() ;  }  } |