# Java学习学年总结

目录

[Java学习学年总结 1](#_Toc64361638)

[一，配置Java程序设计环境 1](#_Toc64361639)

[1.1 安装jdk 1](#_Toc64361640)

[1.2 设置jdk 1](#_Toc64361641)

[二，Java的基本程序设计结构 5](#_Toc64361642)

[2.1 第一个java程序 5](#_Toc64361643)

[2.2 注释 6](#_Toc64361644)

[2.3 数据类型 6](#_Toc64361645)

[2.4 变量 7](#_Toc64361646)

[2.5 运算符 8](#_Toc64361647)

[2.6 字符串 12](#_Toc64361648)

[2.7输出流和输入流 18](#_Toc64361649)

[2.8 控制流程 20](#_Toc64361650)

[2.9 大数值 25](#_Toc64361651)

[2.10数组 26](#_Toc64361652)

[三，类与对象 30](#_Toc64361653)

[3.1 面向对象设计概论 30](#_Toc64361654)

[3.2 类 31](#_Toc64361655)

[3.3方法参数 36](#_Toc64361656)

[3.4 对象的构造 40](#_Toc64361657)

[3.5 包 45](#_Toc64361658)

[四，继承 47](#_Toc64361659)

[4.1 类，超类和子类 48](#_Toc64361660)

[4.2 Object类的了解 59](#_Toc64361661)

[4.3 泛型数组列表 68](#_Toc64361662)

[4.4 对象包装器和自动装箱 70](#_Toc64361663)

[4.5 参数数量可变的方法 72](#_Toc64361664)

[4.6 枚举类 75](#_Toc64361665)

[4.7 反射 77](#_Toc64361666)

[五，接口，lambda表达式与内部类 82](#_Toc64361667)

[5.1 接口 82](#_Toc64361668)

[5.2 接口实例 87](#_Toc64361669)

[5.3 lambda表达式 90](#_Toc64361670)

[5.4 内部类 93](#_Toc64361671)

[六，异常，日志，断言 100](#_Toc64361672)

[6.1 异常概述 100](#_Toc64361673)

[6. 2 捕获异常 102](#_Toc64361674)

[6.3断言和日志 105](#_Toc64361675)

[七，注解 107](#_Toc64361676)

[7.1，什么是java注解？ 107](#_Toc64361677)

[7.2，常用的注解 107](#_Toc64361678)

[7.3，annotation架构 110](#_Toc64361679)

[7.4，annotation的组成部分 110](#_Toc64361680)

[7.5，java 自带的 Annotation 112](#_Toc64361681)

[7.6，Annotation 的作用 113](#_Toc64361682)

[八，泛型设计 116](#_Toc64361683)

[8.1 为什么要泛型设计 116](#_Toc64361684)

[8.2 定义一个简单的泛型类 116](#_Toc64361685)

[8.3 泛型方法 118](#_Toc64361686)

[8.4 类型变量的限制 120](#_Toc64361687)

[8.5 泛型代码和虚拟机 122](#_Toc64361688)

[8.6 泛型的约束和局限性 123](#_Toc64361689)

[8.7 泛型类型的继承规则 125](#_Toc64361690)

[8.8 通配符类型 125](#_Toc64361691)

[九，集合 129](#_Toc64361692)

[9.1 java集合框架 129](#_Toc64361693)

[9.2 具体的集合 133](#_Toc64361694)

[9.3 映射 142](#_Toc64361695)

[十一，输入输出流（io流） 148](#_Toc64361696)

[11.1 File类 149](#_Toc64361697)

[11.2 初探io流 153](#_Toc64361698)

[十二，网络编程 163](#_Toc64361699)

[12.1 java.net包 164](#_Toc64361700)

[12.2 网络编程入门 164](#_Toc64361701)

## 一，配置Java程序设计环境

### 1.1 安装jdk

网址：<https://www.oracle.com/java/technologies/javase-downloads.html>

**步骤：**

**•你需要的是 JDK (Java SE 8 开发包)，而不是 JRE。**

**•Windows 或 Linux: 32 位选择 x86，64 位以 x64c**

**•Linux：选择.tar.gz版本。**

**接受许可协议，然后下载文件。**

### 1.2 设置jdk

下载 JDK 之后，在 Windows 上， 启动控制面板，选择“ 系统与安全’（System and Security)，再选择 ‘ 系 统”（System)，选择高级系统设置（Advanced System Settings)(参见图 2-1 )。在系统属 性（SystemProperties) 对话框中，点击“ 高级”(Advanced) 标签页，然后点击“ 环境” (Environment) 按钮。

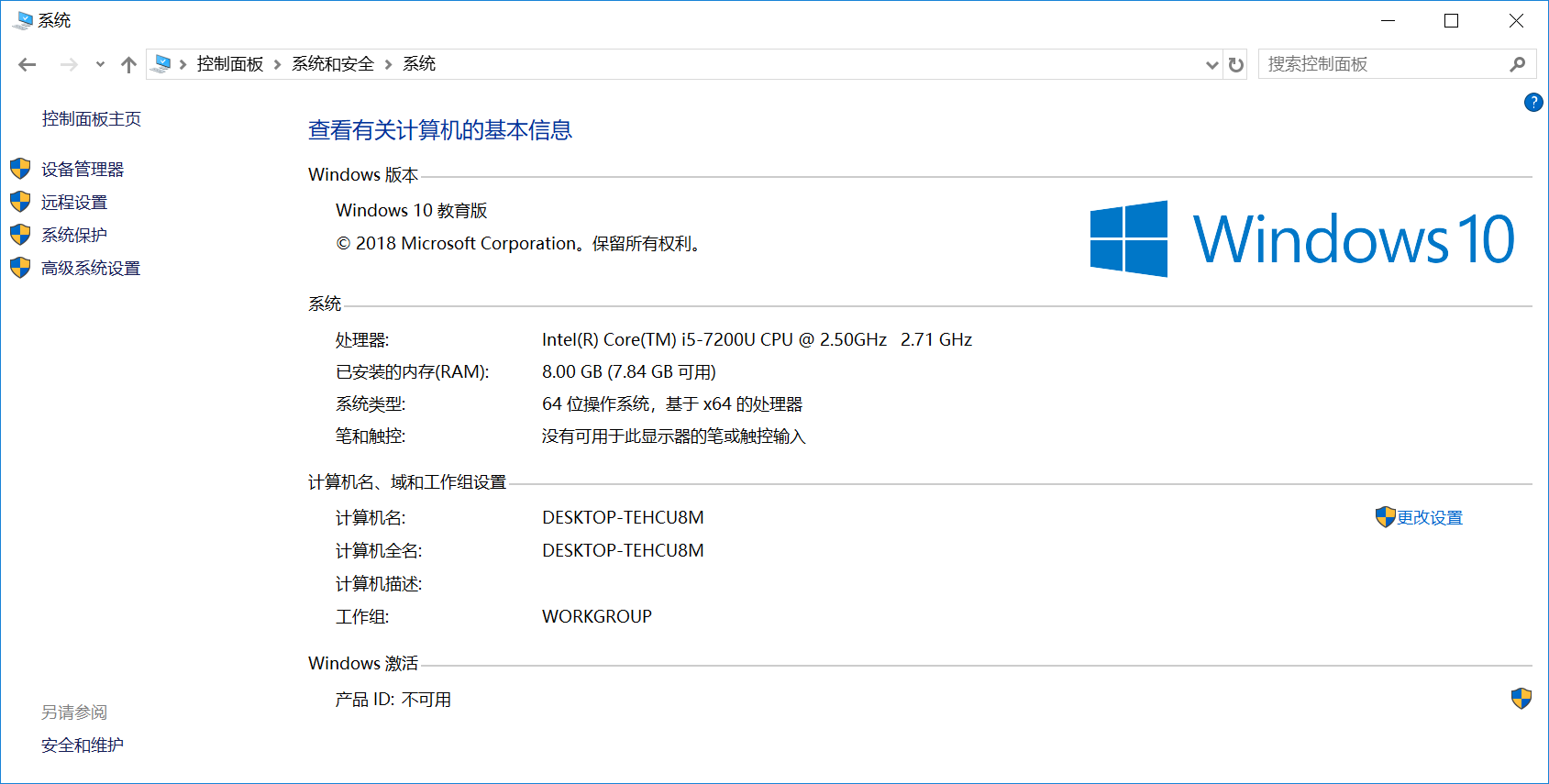
1：打开控制面板



2.选择系统和安全

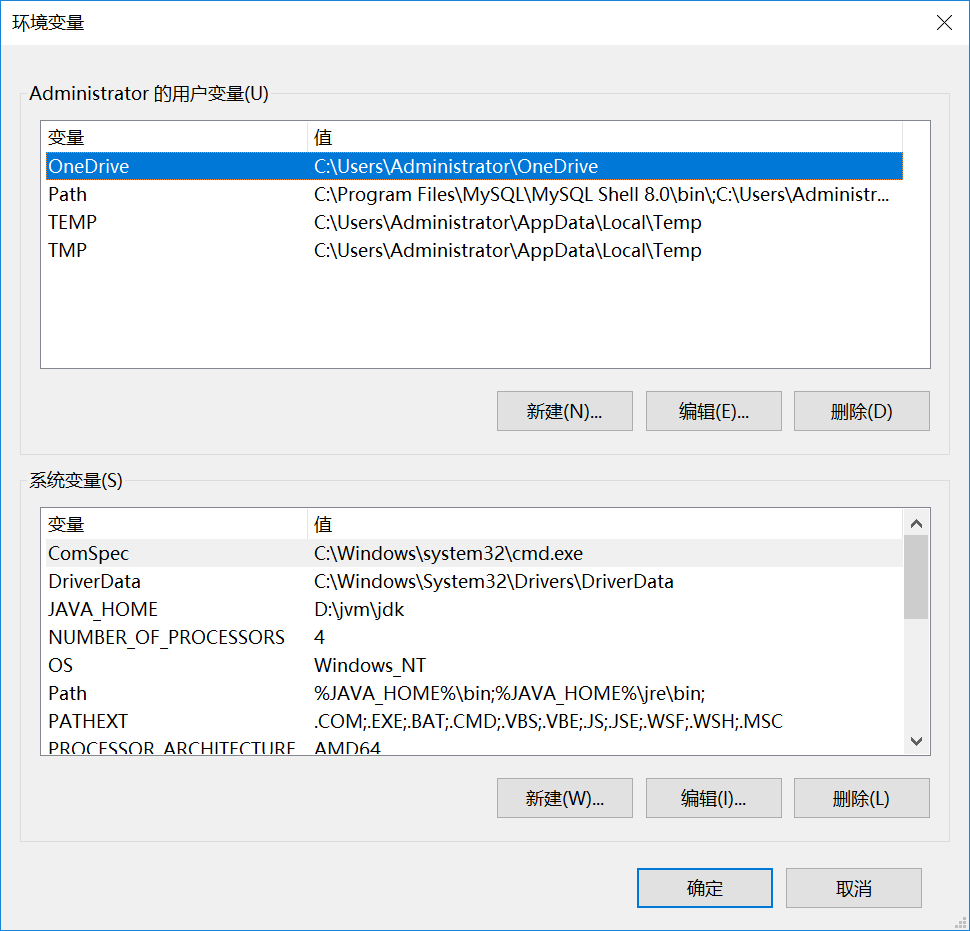


3.选择系统



4.点击高级系统设置

  
5.点击环境变量



6.设置系统变量path和Java\_home

Path: **%JAVA\_HOME%\bin;%JAVA\_HOME%\jre\bin;**

**Java\_home:自己Java储存的实际路径**

变量名：**CLASSPATH**

变量值：.;%JAVA\_HOME%\lib\dt.jar;%JAVA\_HOME%\lib\tools.jar;      //记得前面有个"."

注意：如果使用 1.5 以上版本的 JDK，不用设置 CLASSPATH 环境变量，也可以正常编译和运行 Java 程序。

7.打开终端窗口cmd输入指令：**java -version**、**java**、**javac** .出现以下信息说明环境安装成功了。



如果出现'Javac' 不是内部或外部命令，也不是可运行的程序

或批处理文件。说明安装并未成功。

## 二，Java的基本程序设计结构

### 2.1 第一个java程序

代码：

package com.dong.day1;

//包暂时不用了解，后面的笔记会详细说明

public class Welcome {

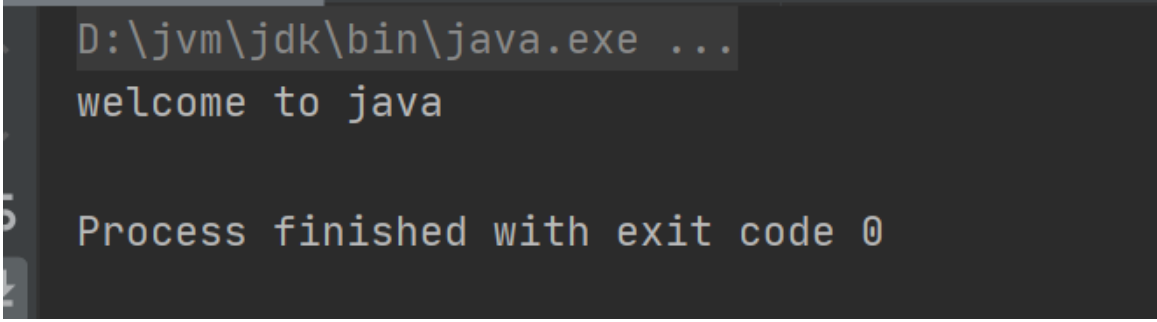
public static void main(String[] args) {

System.out.println("welcome to java");

}

}

结果：



**说明：**

（1）Java 区分大小写

（2）public成为访问修饰符，这些修饰符用于控制程序的其他部分对这段代码的访问级別

（3）为了代码能够执行，在类的源文件中必须包含一个 main 方法

（4）在 Java 中，像在 C/C++ 中一样，用大括号划分程序的各 个部分（通常称为块)

（5）System.out 还有一个 print 方法， 它在输出之后不换行。

### 2.2 注释

注释有三种形式：

（1）//： 其注释内容从 // 开始到本行结尾。

（2）/\*\*\*/：将一段比较 长的注释括起来

（3）/\*\* 开始， 以 \*/ 结束：用来自动地生成文档

### 2.3 数据类型

#### 2.3.1 整形

Long:8个字节

Short：2个字节

Int：4个字节

Byte：一个字节

长整型数值有一个后缀 L 或 1 (如 4000000000L)。十六进制数值有一个前缀 Ox 或 0X (如 OxCAFEL 八进制有一个前缀 0, 例如， 010 对应八进制中的 8。很显然， 八进制表示法比较 容易混淆， 所以建议最好不要使用八进制常数。

#### 2.3.2 浮点型

Float: 4个字节 大约保留6位小数

Double：8个字节 大约保留20个字节

float类型的数值有一个后缀 F 或 f (例如，3.14F)。没有后缀 F 的浮点数值（如 3.14 ) 默 认为double 类型。当然，也可以在浮点数值后面添加后缀 D 或 d (例如，3.14D)。

3种溢出错误提示：

•正无穷大

•负无穷大

•NaN (不是一个数字）

#### 2.3.3 Boolean型

boolean (布尔）类型有两个值：false 和 true, 用来判定逻辑条件整型值和布尔值之间不能进行相互转换。

#### 2.3.4 char（字符）型

（1）char 类型的字面量值要用单引号括起来。

### 2.4 变量

#### 2.4.1 变量的初始化

声明一个变量之后，必须用赋值语句对变量进行显式初始化（赋值）， 千万不要使用未初始化的变量。

代码：

Int a=5;

#### 2.4.2 常量

在 Java中，利用关键字 **final**指示常量，关键字 final 表示这个变量只能被赋值一次。一旦被赋值之后，就不能够再更改了。习惯上, 常量名使用全大写。

代码：

final int A=10;

说明：

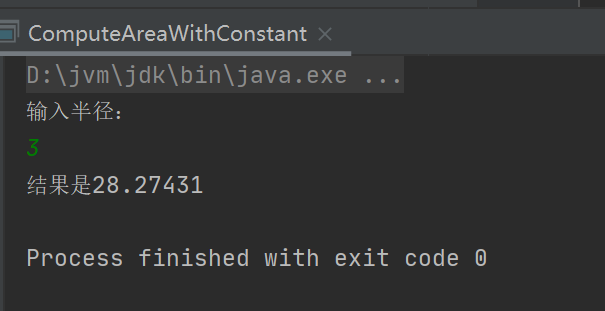
如果常量不在main方法中定义，而是在类中定义，那么这个常量被称为类常量。

程序清单1

代码：

package com.dong.day3;  
  
import java.util.Scanner;  
  
public class ComputeAreaWithConstant {  
 public static void main(String[] args) {  
 final double PI =3.14159; //常量的定义  
 Scanner scanner=new Scanner(System.*in*);  
 System.*out*.println("输入半径：");  
 double r=scanner.nextDouble();  
 double area=r\*r\*PI;  
 System.*out*.println("结果是"+area);  
 }  
}

结果：



### 2.5 运算符

在 Java中，使用算术运算符 +、-、 \*、/ 表示加、减、乘、除运算。 当参与 / 运算的两个 操作数都是整数时，表示整数除法；否则，表示浮点除法。整数的求余操作（有时称为取模) 用 ％ 表示。

需要注意， 整数被 0除将会产生一个异常， 而浮点数被 0 除将会得到无穷大或 NaN结果。

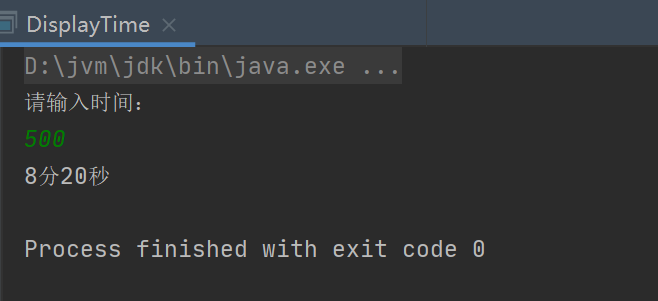
程序清单：

将秒的计量时间转化为分和秒为计量单位的时间，如500秒转化为8 分20秒

代码：

package com.dong.day3;  
  
import java.util.Scanner;  
  
public class DisplayTime {  
 public static void main(String[] args) {  
 Scanner scanner=new Scanner(System.*in*);  
 System.*out*.println("请输入时间：");  
 int second=scanner.nextInt();  
 System.*out*.println(second/60+"分"+second%60+"秒");  
 }  
}

结果：



#### 2.5.1 数学函数与常量

在 Math类中，包含了各种各样的数学函数。在编写不同类别的程序时，可能需要的函数也不同。Math类在java.lang包中，java.lang包是隐式导入的，故无需自行导入（import）

要想计算一个数值的平方根， 可以使用 sqrt 方法：

double x = 4;

double y = Math.sqrt(x);

System.out.println(y); // prints 2.0

在 Java中，没有幂运算， 因此需要借助于 Math 类的 pow方法。

pow方法有两个 double类型的参数， 其返回结果也为 double 类型。

##### Math 类提供了一些常用的三角函数：

Math,sin sin函数

Math.cos cos函数

Math.tan tan函数

Math.atan arctan函数

Math.acos arccos函数

Math.asin arcsin函数

##### 还有指数函数以及它的反函数— —自然对数以及以 10 为底的对数：

Math.exp e^x自然指数

Math.log 自然对数

Math.log10 10为底的对数

##### 取整方法

Math.ceil(x) 向上取整

Math.floor(x) 向下取整

Math.rint(x) 取x最接近的整数，如果一样接近，例如2.5就向偶数取整（2）；

Math.round(x) 如果是单精度就（int）Math.floor(x+0.5);如果是双精度就(long)Math.floor(x+0.5)

##### 最后，Java 还提供了两个用于表示 TC 和 e 常量的近似值：

Math.PI ==Π

Math.E ==e（常数）

##### 补充：了解random方法

Math.random()

例子：

a+Math.random()\*b; 取值在【a，b】

#### 2.5.2 数值类型之间的转换

•如果两个操作数中有一个是 double 类型， 另一个操作数就会转换为 double 类型。

•否则，如果其中一个操作数是 float 类型，另一个操作数将会转换为 float 类型。

•否则， 如果其中一个操作数是 long 类型， 另一个操作数将会转换为 long 类型。

•否则， 两个操作数都将被转换为 int 类型。

程序清单：

**要求：营业税值保留两位小数**

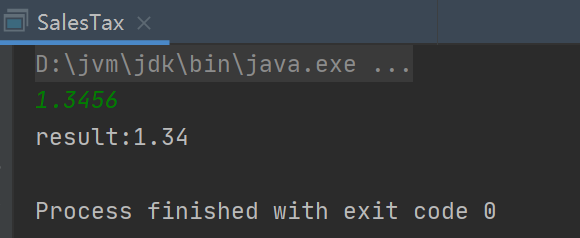
**输入：营业税值s=1.3456**

**输出：1.34**

代码：

package com.dong.day3;  
  
import java.util.Scanner;  
  
public class SalesTax {  
 public static void main(String[] args) {  
 //将营业税值保留两位小数  
 Scanner scanner=new Scanner(System.*in*);  
 double s=scanner.nextDouble();  
 System.*out*.println("result:"+(int)(s\*100)/100.0);  
 }  
}

结果：



#### 2.5.3强制转化

在上一小节中看到， 在必要的时候， int类型的值将会自动地转换为 double 类型。但另 一方面，有时也需要将 double转换成 int。 在 Java中， 允许进行这种数值之间的类型转换。 当然， 有可能会丢失一些信息。

#### 2.5.4 位运算符

& ("and")

| ("or")

~("not")

#### 2.5.5 补充：了解如何显示当前时间

可以通过调用System.currentTimeMillis()返回当前时间。

程序清单

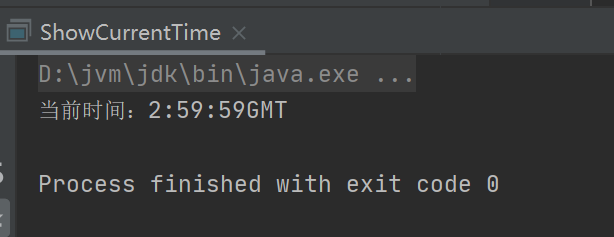
返回当前GMT时间

代码：

package com.dong.day3;  
  
public class ShowCurrentTime {  
 public static void main(String[] args) {  
 long second=System.*currentTimeMillis*(); //获取的时间是毫秒级  
 second=second/1000; //转化为秒  
 long currentSecond=second%60;   
 long totalMinute=second/60;  
 long currentMinute=totalMinute%60;  
 long totalHour=totalMinute/60;  
 long currentHour=totalHour%24;

System.*out*.println("当前时间："+currentHour+":"+currentMinute+":"+currentSecond+"GMT");  
 }  
}

结果：



### 2.6 字符串

#### 2.6.1 子串

String类的 substring方法可以从一个较大的字符串提取出一个子串。例如：

String greeting = "Hello";

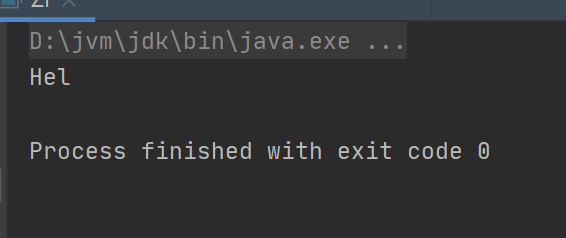
String s = greeting.substring(0, 3); //substring(参数1，参数2)，参数2是不取的位置

程序清单

代码：

package com.dong.day4;  
  
 public class Zi {  
 public static void main(String[] args) {  
 String str="HelloWorld!";  
 String str1=str.substring(0,3);  
 System.*out*.println(str1);  
 }  
}

结果：



#### 2.6.2 拼接

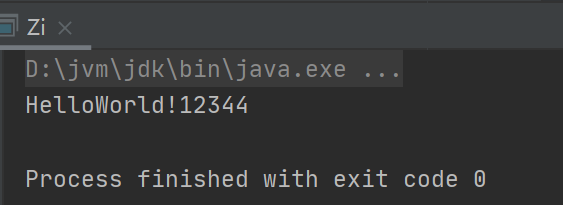
（1）与绝大多数的程序设计语言一样，Java语言允许使用 + 号连接（拼接）两个或多个字符串。

程序清单

代码：

package com.dong.day4;  
  
 public class Zi {  
 public static void main(String[] args) {  
 String str1="HelloWorld!";  
 String str2="12344";  
 System.*out*.println(str1+str2);  
 }  
}

结果：



**注意：连接时，至少有一个操作数是字符。**

（2）当将一个字符串与一个非字符串的值进行拼接时，后者被转换成字符串（在第 5 章中可 以看到，任何一个 Java对象都可以转换成字符串）

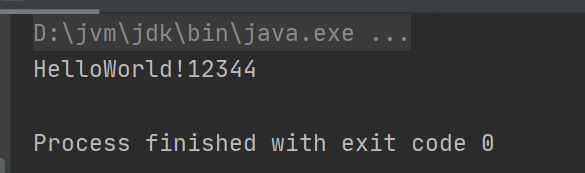
(3)可以使用concat方法连接两个字符串。

程序清单：

代码：

package com.dong.day4;  
  
 public class Zi {  
 public static void main(String[] args) {  
 String str1="HelloWorld!";  
 String str2="12344";  
 String str3=str1.concat(str2);  
 System.*out*.println(str3);  
 }  
}

结果：



#### 2.6.3 不可改字符串

String类没有提供用于修改字符串的方法.只能拼接

Greeting=”Hello”;

greeting = greeting.substring(0, 3) + "p!";

3.6.4 字符串是否相等

可以使用 equals方法检测两个字符串是否相等。对于表达式： s.equals(t) 如果字符串 s 与字符串 t 相等， 则返回 true.

一定不要使用= 运算符检测两个字符串是否相等.

#### 2.6.4 空串与 Null 串

**空串 ""**: 是长度为 0 的字符串。可以调用以下代码检查一个字符串是否为空：

if (str.length() = 0)

**NULL串：**不过，String 变量还可以存 放一个特殊的值， 名为 null, 这表示目前没有任何对象与该变量关联（关于 null 的更多信息 请参见第 4 章)。要检查一个字符串是否为 null, 要使用以下条件： if (str == null)

**注意**：有时要检查一个字符串既不是 null 也不为空串，这种情况下就需要使用以下条件： if (str != null && str.length()!= 0) 首先要检查str 不为 null

#### 2.6.5 码点与代码单元

**码点**是指一个编码表中的某个字符对应的代码值.

每个字符用16位表示**代码单元**，而辅助字符采用连续的**一对连续代单元**进行编码。

**length 方法**：返回采用 UTF-16 编码表示的给定字符串所需要的代码单元数量。

String greeting = "Hello";

int n = greeting.length() ;

**codePointCount**: 得到实际的长度，即码点数量

**length和codepointcount的区别**：对于UniCode的编码有区别，length对辅助单元的代码单元数量是2，codepointcount是1.

**.charAt(n)**：返回位置 n 的代码单元，n 介于 0 ~ s.length()-l 之间

char first = greeting.charAt(0); // first is 'H'

char last = greeting.charAt(4); // last is ’o’

#### 2.6.6 获取子串长度

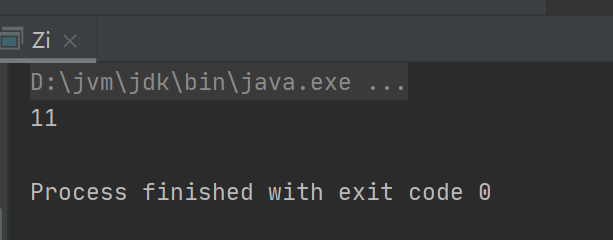
1，可以调用字符串的length()方法；此方法返回值是数字（int）

程序清单：

代码：

package com.dong.day4;  
  
 public class Zi {  
 public static void main(String[] args) {  
 String str1="HelloWorld!";  
 String str2="12344";  
 int len=str1.length();  
 System.*out*.println(len);  
 }  
}

结果：



#### 2.6.7 从字符串中获取字符

方法s.charAt(index);可以获取字符串中的字符，index是你要获取的字符的位置。

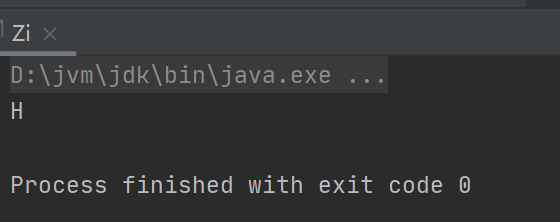
注意：下标index的取值范围：【0~s.length()-1】

程序清单：

代码

package com.dong.day4;  
  
 public class Zi {  
 public static void main(String[] args) {  
 String str1="HelloWorld!";  
 String str2="12344";  
 char str3=str1.charAt(0);  
 System.*out*.println(str3);  
 }  
}

结果：



#### 2.6.8 字符串的转换

toLowerCase() 返回一个新的字符串，其中所有字母都是小写

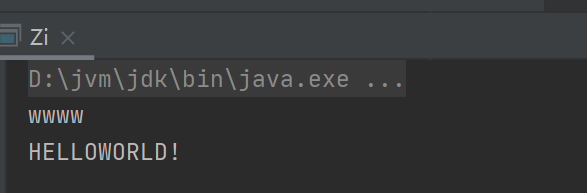
toUpperCase() 返回一个新的字符串，其中所有字母都是大写

程序清单：

代码：

package com.dong.day4;  
  
 public class Zi {  
 public static void main(String[] args) {  
 String str1="helloworld!";  
 String str2="WWWW";  
 String Str3= str2.toLowerCase();  
 String Str4=str1.toUpperCase();  
 System.*out*.println(Str3+"\n"+Str4);  
 }  
}

结果：



#### 2.6.9 字符串的比较

1,使用equals方法,返回值为boolean类型，如果值相同就返回true

例子：

If（a.equals(b)）

System.out.println(“YES”);

程序清单1：

代码：

package com.dong.day4;  
  
 public class Zi {  
 public static void main(String[] args) {  
 String str1="WWWW";  
 String str2="WWWW";  
 if(str1.equals(str2))  
 System.*out*.println("YES");  
 }  
}

结果：



2,使用a.compareTo(b)方法，返回值为整形，0代表相等，>0代表a>b;<0代表a<b;

程序清单2：

代码：

package com.dong.day4;  
  
 public class Zi {  
 public static void main(String[] args) {  
 String str1="WWWW";  
 String str2="WWWW";  
  
 if( str1.compareTo(str2)==0)  
 System.*out*.println("YES");  
 else  
 System.*out*.println("NO");  
 }  
}

结果：



### 2.7输出流和输入流

#### 2.7.1输入

**package** shuru;

**import** java.util.\*; //输入流的包

**public** **class** ShuRu {

**public** **static** **void** main(String[] args)

{

Scanner in = **new** Scanner(System.***in***); //构造Scanner，并入System.in

// get first input 输入一行

System.***out***.println("What is you name?"); //输出流

String name=in.nextLine();

// 输入整形数字

**int** age=in.nextInt();

//输出数据name和age

System.***out***.println("Hello,"+name+".next year,you'll be "+(age+1));

//输出

}

}

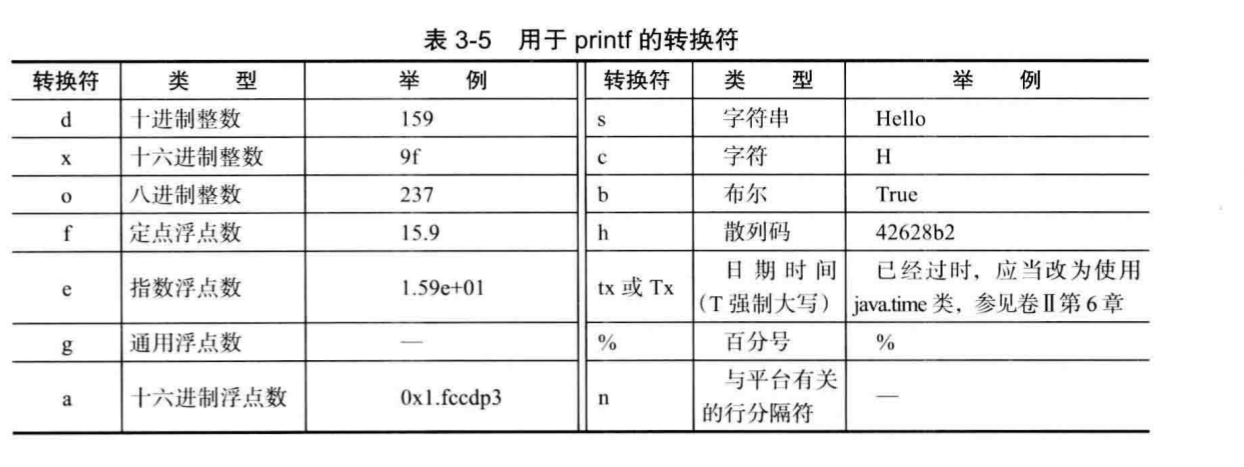
**输入的方法：**



#### 2.7.2格式输出

System.out.printf(“%.2If”, x);

格式沿用C



可以使用静态的 String.format 方法创建一个格式化的字符串， 而不打印输出

String message = String.format("Hello, %s. Next year, you'll be %d", name , age);

#### 2.7.3文件的输入和输出

（1）要想对文件进行读取，就需要一个用 File 对象构造一个 Scanner 对象，如下所示： Scanner in = new Scanner(Paths.get("niyflle.txt"), "UTF-8");

（2）要想写入文件， 就需要构造一个 PrintWriter 对象。在构造器中，只需要提供文件名： PrintWriter out = new PrintWriter(“myfile.txt", "UTF-8");

如果文件不存在，创建该文件.

Scanner(File f) 构造一个从给定文件读取数据的 Scanner。

Scanner(String data) 构造一个从给定字符串读取数据的 Scanner。

### 2.8 控制流程

#### 2.8.1 条件结构

##### 1, if条件结构

**语法格式：**

**If(true)**

**{statement}**

**或者**

**If(true)**

**{statement}**

**Else**

**{statement}**

**程序清单：**

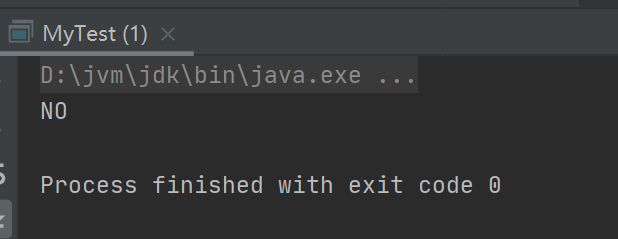
**题目：**

**判断1和2的大小，如果1>2则输出YES，否则输出NO**

**代码：**

**package com.dong.day5;  
  
public class MyTest {  
 public static void main(String[] args) {  
 if(1>2)  
 System.*out*.println("Yes");  
 else  
 System.*out*.println("NO");  
 }  
}**

**结果：**



##### 2,switch语句

**语法格式：**

**switch (表达式){**

**case 目标值1:**

**执行语句1**

**break;**

**case 目标值2:**

**执行语句2**

**break;**

**．．．．．．**

**case 目标值n:**

**执行语句n**

**break;**

**default:**

**执行语句n+1**

**break;**

**}**

**程序清单：**

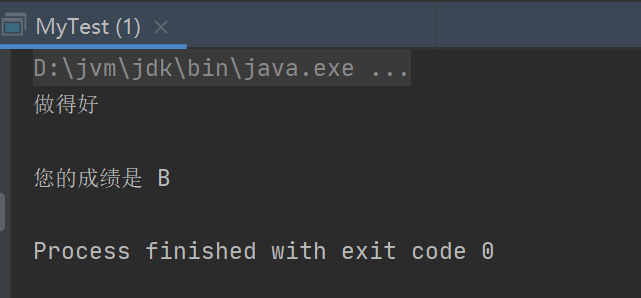
**题目：**

**已知grade为B， A~E有不同的回答**

**代码：**

**package com.dong.day5;  
  
public class MyTest {  
 public static void main(String[] args) {  
 char grade = 'B';  
  
 switch(grade)  
 {  
 case 'A' :  
 System.*out*.println("很棒！\n");  
 break;  
 case 'B' :  
 case 'C' :  
 System.*out*.println("做得好\n");  
 break;  
 case 'D' :  
 System.*out*.println("您通过了\n");  
 break;  
 case 'F' :  
 System.*out*.println("最好再试一下\n");  
 break;  
 default :  
 System.*out*.println("无效的成绩");  
 }  
 System.*out*.printf("您的成绩是 %c\n", grade );  
 }  
}**

**结果：**



#### 2.8.2循环结构

##### 1,do……while循环

**语法格式：**

**Do**

**{statement}**

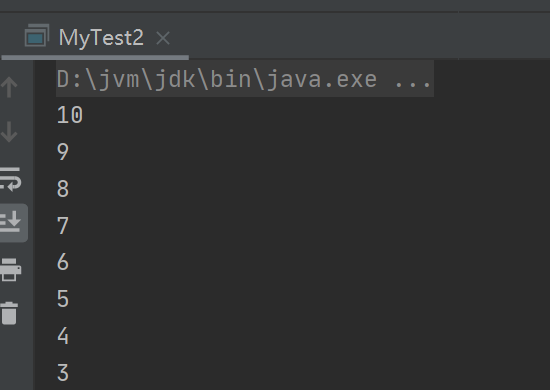
**while(true);**

**程序清单**

**代码：**

**package com.dong.day5;  
  
public class MyTest2 {  
 public static void main(String[] args) {  
 int i=10;  
 do{  
 System.*out*.println(i);  
 i--;  
 }while(i>=0);  
 }  
}**

**结果：**



##### 2，while循环

**语法格式：**

**While(true)**

**{**

**Statement;**

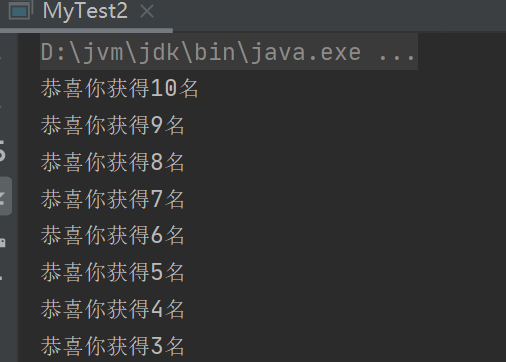
**}**

**程序清单：**

**代码：**

**package com.dong.day5;  
  
public class MyTest2 {  
 public static void main(String[] args) {  
 int i=10;  
 while(i>=0)  
 {  
 System.*out*.println("恭喜你获得"+i+"名");  
 i--;  
 }  
 }  
}**

**结果：**



##### 3，for确定循环

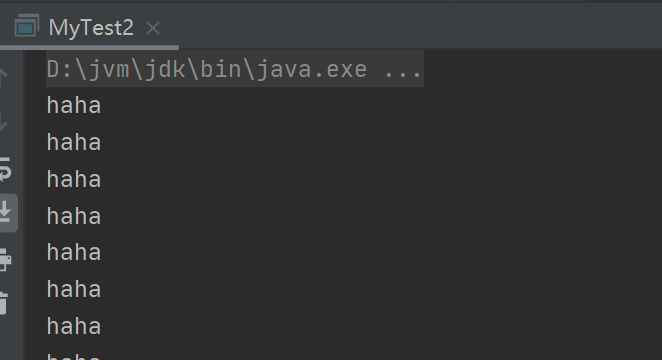
**语法格式：**

**for ( init; condition; increment ) { statement(s); }**

**程序清单：**

**package com.dong.day5;  
  
public class MyTest2 {  
 public static void main(String[] args) {  
 int i;  
 for(i=0;i<=10;i++)  
 System.*out*.println("haha");  
 }  
}**

**结果：**



### 2.9 大数值

使用java.math 包中的两个 很有用的类：Biglnteger 和 BigDecimaL 这两个类可以处理包含任意长度数字序列的数值。 Biglnteger 类实现了任意精度的整数运算， BigDecimal 实现了任意精度的浮点数运算

使用静态的 **valueOf方法**可以将普通的数值转换为大数值。

Biglnteger a = Biglnteger.valueOf(100);

遗憾的是，不能使用人们熟悉的算术运算符（如：+ 和 \*) 处理大数值。 而需要使用大数 值类中的 add(+) 和 multiply (\*)方法。

Biglnteger c = a.add(b); // c = a + b

Biglnteger d = c.multiply(b.add(Biglnteger.valueOf(2))); // d = c \* (b + 2)

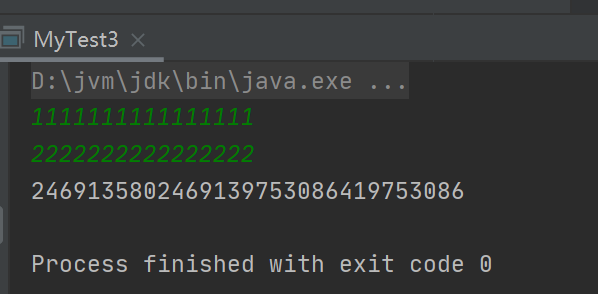
**程序清单：**

**题目：计算d=c\*(b+2)**

代码：

package com.dong.day5;  
  
import java.math.BigInteger;  
import java.util.Scanner;  
  
public class MyTest3 {  
 public static void main(String[] args) {  
 BigInteger c,b,result;  
 Scanner scanner=new Scanner(System.*in*);  
 b=scanner.nextBigInteger();  
 c=scanner.nextBigInteger();  
 result=c.multiply(b.add(BigInteger.*valueOf*(2)));  
 System.*out*.println(result);  
 }  
}

结果：



### 2.10数组

#### 2.10.1 声明数组

**语法格式：**

**数据类型[ ] 数组名=new 数据类型[大小];**

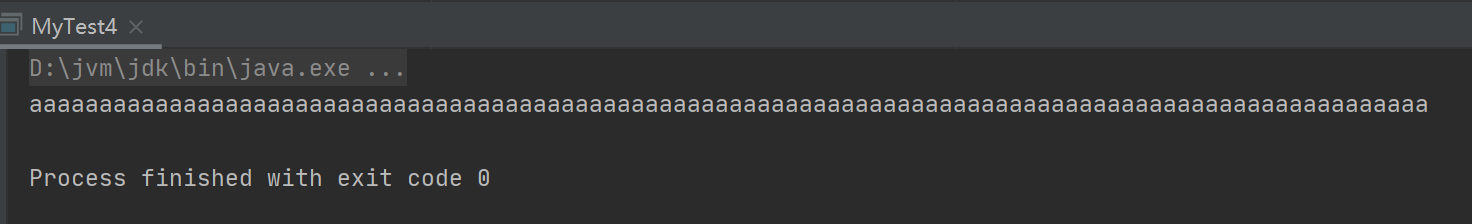
**程序清单：**

**声明一个char类型数组并输出**

**代码：**

**package com.dong.day5;  
  
public class MyTest4 {  
 public static void main(String[] args) {  
 char[]a;  
 a=new char[100];  
 for(int i=0;i<100;i++)  
 a[i]='a';  
 System.*out*.println(a);  
 }  
}**

**结果：**



#### 2.10.2 foreach循环

Java 有一种功能很强的循环结构， 可以用来依次处理数组中的每个元素（其他类型的元 素集合亦可）而不必为指定下标值而分心

**语法格式：**

**for (variable : collection) statement**

**程序清单：**

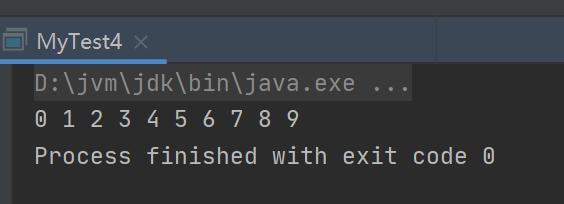
**要求：**

**使用foreach循环输出数组a中的值**

**代码：**

**package com.dong.day5;  
  
public class MyTest4 {  
 public static void main(String[] args) {  
 int[] a=new int[10];  
 for(int i=0;i<10;i++)  
 a[i]=i;  
 for(int element:a)  
 System.*out*.print(element+" ");  
 }  
}**

**结果：**



#### 2.10.3 数组拷贝

数组拷贝需要用到就需要用到Arrays类中的copyOf方法：int[] copiedLuckyNumbers = Arrays.copyOf(luckyNumbers , luckyNumbers .length);

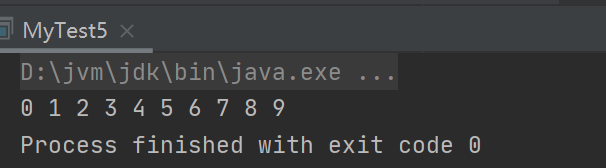
第一个参数是被拷贝数组，第 2 个参数是新数组的长度。

**程序清单：**

代码实现：

package com.dong.day5;  
  
import java.util.Arrays;  
  
public class MyTest5 {  
 public static void main(String[] args) {  
 int[]a=new int[10];  
 int[]b=new int[10];  
  
 for(int i=0;i<10;i++)  
 a[i]=i;  
 b= Arrays.*copyOf*(a,a.length);  
 for(int elem:b)  
 System.*out*.print(elem+" ");  
 }  
}

结果：



#### 2.10.4 数组排序

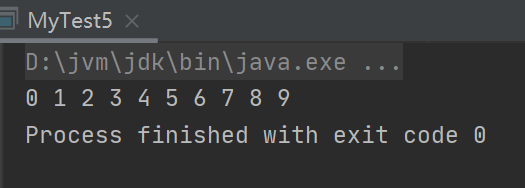
数组排序需要使用Arrays类中的sort方法：Arrays.sort(数组名);

程序清单：

代码:

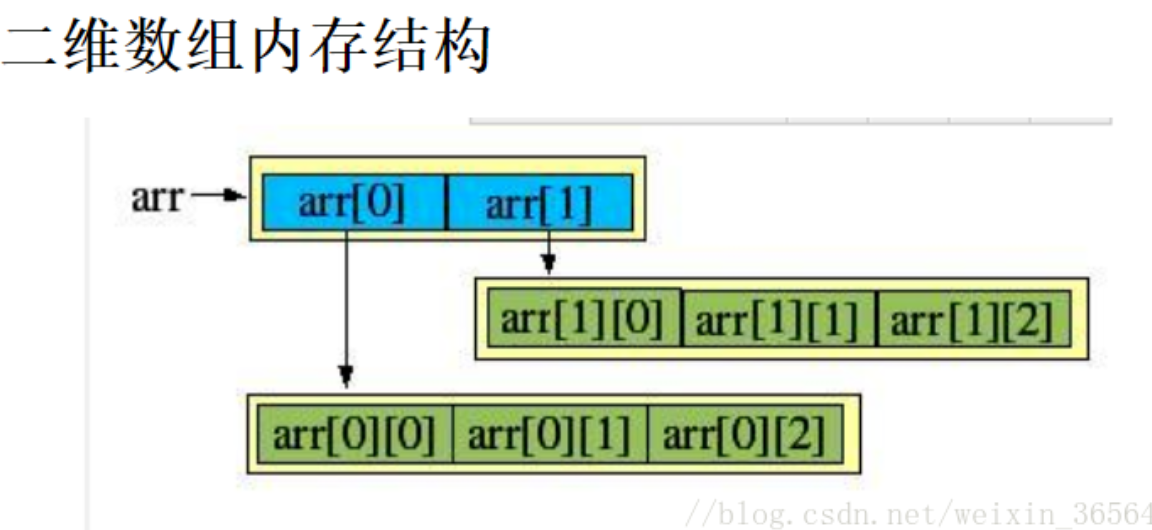
package com.dong.day5;  
  
import java.lang.reflect.Array;  
import java.util.Arrays;  
  
public class MyTest4 {  
 public static void main(String[] args) {  
 int[] a=new int[10];  
 for(int i=0;i<10;i++)  
 a[i]=10-i-1;  
 Arrays.*sort*(a);  
 for(int element:a)  
 System.*out*.print(element+" ");  
 }  
}

结果：



#### 2.10.5 多维数组（重点二维数组）

多维数组其实是一维数组的嵌套。



 两种初始化形式

    格式1: 动态初始化

数据类型 数组名 [ ][ ] = new 数据类型[m][n]  
数据类型 [ ][ ]  数组名 = new 数据类型[m][n]  
数据类型 [ ]   数组名 [ ] = new 数据类型[m][n]

举例：int [ ][ ]  arr=new  int [5][3];  也可以理解为“5行3例”

格式2: 静态初始化  
数据类型 [ ][ ]   数组名 = {{元素1,元素2....},{元素1,元素2....},{元素1,元素2....}.....};

举例：int [ ][ ]  arr={{22,15,32,20,18},{12,21,25,19,33},{14,58,34,24,66}};

静态初始化可用于不规则二维数组的初始化

**程序清单：**

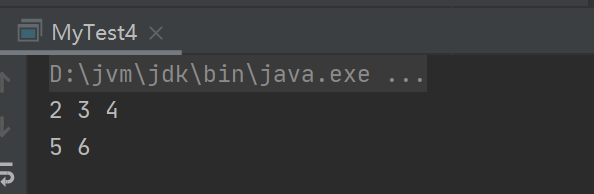
**要求：**

**初始化不规则多维数组并输出**

**代码：**

**package com.dong.day5;  
  
import java.lang.reflect.Array;  
import java.util.Arrays;  
  
public class MyTest4 {  
 public static void main(String[] args) {  
 int[][]a=new int[][]{{2,3,4},{5,6}};  
 for(int i=0;i< a.length;i++) {  
 for (int j = 0; j <a[i].length; j++)  
 System.*out*.print(a[i][j] + " ");  
 System.*out*.println();  
 }  
 }  
}**

**结果：**



## 三，类与对象

面向对象程序设计概述▲

使用预定义类 ▲

用户自定义类 ▲

静态域与静态方法 ▲

方法参数 ▲

对象构造 ▲

包 ▲

### 3.1 面向对象设计概论

**面向对象程序设计**（英语：Object-oriented programming，[缩写](https://wanweibaike.com/wiki-%E7%BC%A9%E5%86%99)：OOP）是种具有[对象](https://wanweibaike.com/wiki-%E5%AF%B9%E8%B1%A1_(%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6))概念的[程序编程典范](https://wanweibaike.com/wiki-%E7%BC%96%E7%A8%8B%E8%8C%83%E5%9E%8B)，同时也是一种程序开发的抽象方针。它可能包含[资料](https://wanweibaike.com/wiki-%E6%95%B0%E6%8D%AE)、属性（英语：[Attribute (computing)](https://en.wanweibaike.com/wiki-Attribute_(computing))）、[代码](https://wanweibaike.com/wiki-%E6%BA%90%E4%BB%A3%E7%A0%81)与[方法](https://wanweibaike.com/wiki-%E6%96%B9%E6%B3%95_(%E9%9B%BB%E8%85%A6%E7%A7%91%E5%AD%B8))。**对象**则指的是**类的实例**。**它将对象作为程序的基本单元**，将**程序**和**数据**封装其中，以**提高**软件的**重用性、灵活性和扩展性**，对象里的程序可以访问及经常修改对象相关连的资料。在面向对象程序编程里，计算机程序会被设计成彼此相关的对象。

通俗的理解：

1，面向对象程序设计可以看作一种在程序中包含各种独立而又互相调用的对象的思想。

2，OOP被理解为一种将程序分解为封装数据及相关操作的模块而进行的编程方式。

#### 3.1.1 封装性

具备[封装性](https://wanweibaike.com/wiki-封裝\_(物件導向程式設計))（Encapsulation）的面向对象编程隐藏了某一方法的具体运行步骤，取而代之的是通过消息传递机制发送消息给它。封装是通过限制只有特定类的对象可以访问这一特定类的成员，而它们通常利用接口实现消息的传入传出。

#### 3.1.2 继承性

[继承性](https://wanweibaike.com/wiki-%E7%BB%A7%E6%89%BF%E6%80%A7)（Inheritance）是指，在某种情况下，一个类会有“[子类](https://wanweibaike.com/wiki-%E5%AD%90%E7%B1%BB)”。子类比原本的类（称为父类）要更加具体化。子类会继承父类的属性（英语：[Attribute (computing)](https://en.wanweibaike.com/wiki-Attribute_(computing))）和[行为](https://wanweibaike.com/wiki-%E8%A1%8C%E4%B8%BA)，并且也可包含它们自己的。

#### 3.1.3 多态性

多态（Polymorphism）是指由继承而产生的相关的不同的类，其对象对同一消息会做出不同的响应。

#### 3.1.4 抽象性

**抽象**（Abstraction）是简化复杂的现实问题的途径，它可以为具体问题找到最恰当的类定义，并且可以在最恰当的继承级别解释问题。举例说明，莱丝在大多数时候都被当作一条狗，但是如果想要让它做牧羊犬做的事，你完全可以调用牧羊犬的方法。如果狗这个类还有**动物的父类**，那么你完全可以视莱丝为一个动物。

#### 3.1.5 为什么要面向对象编程？

**1.面向对象程序设计推广了程序的灵活性和可维护性，并且在大型项目设计中广为应用。**

**2.面向对象程序设计要比以往的做法更加便于学习，因为它能够让人们更简单地设计并维护程序，使得程序更加便于分析、设计、理解。**

### 3.2 类

**什么是类？**

类就是具备某些共同bai特征的实体的集合，它是一种抽象的数据类型，它是对所具有相同特征实体的抽象。在面向对象的程序设计语言中，类是对一类“事物”的属性与行为的抽象。

#### 3.2.1 对象

**什么是对象？**

对象是一个类的实例，是一个实体。

**举个例子：**

如果有一个类是学生类，李明是学生，李明是学生类的一个对象。

**什么是对象变量？**

对象变量是将类作为数据类型来定义的变量，注意：对象变量需要实例化后才能调用类中的方法，属性等等。

**程序清单3.1：**

**要求：**

1，定义一个学生类：类中有三公有个属性，年龄和性别和名字

2，输出对象

代码：

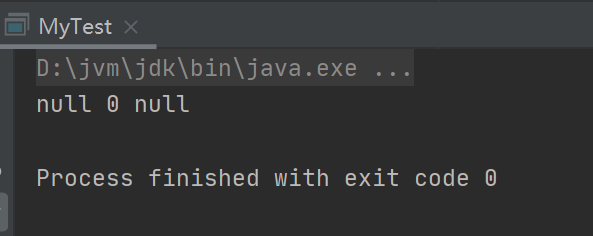
**学生类**：

package com.dong.day5.student;  
  
public class Student {  
 public String name;  
 public String sex;  
 int age;  
}

**测试类：**

package com.dong.day5.student;  
  
public class MyTest {  
 public static void main(String[] args) {  
 //定义对象  
 Student LiMing;  
 //实例化对象  
 LiMing=new Student();  
 System.*out*.println(LiMing.name+" "+LiMing.age+" "+LiMing.sex);  
 }  
}

结果：



#### 3.2.2 类之间的关系

**依赖**：如果一个类的方法操纵另一个类的对象，我们就说一个类依赖于另一个类。

**聚合**：一个 Order 对象包含一些 Item 对象。聚合关系意味着类 A 的对象包含类 B 的对象。

**继承**：类A继承了类B的功能。



#### 3.2.3 使用预定类

并不是所有 的类都具有面向对象特征。例如，Math 类。在程序中，可以使用 Math 类的方法， 如 Math, random, 并只需要知道方法名和参数（如果有的话，) 而不必了解它的具体实现过程。

#### 3.2.4 对象与对象变量

**对象**：构造对象，指定其初始状态。**在 Java 程序设计语言中， 使用构造器（constructor ) 构造新实例**，**构造器的名字应该与类名相同**。（构造器在笔记后头有讲，这里先了解）

**对象变量**：Date birthday = new Date();对象变量是birthday，对象是 new Date();

对象与对象变量之间的区别：

如Date deadline; 变量 deadline 不是一个对象， 实际上也没有引用对象。

**要想调用对象的方法有两种方法**：

1：deadline=new Date(); //用新构造的对象给变量引用

2：deadline=day; //day是一个已经引用对象的对象变量

**注意**：一个对象变量并没有实际包含一个对象，而仅仅引用一个对象。

#### 3.2.5 类的访问权限

**类的访问权限类型**：

1，public（公有）所有**包**（笔记后续有详细的讲，现在不用深究）下的类都可以调用

2，protect（保护）仅仅在和**此类**相同**包**下的类可以调用

3，private（私有）仅仅**类的自身方法**可以调用

#### 3.2.6 final实例域

private final String name; //在每 一个构造器执行之后,name的值不能再被重新设置

**final：**修饰符大都应用于基本 （primitive ) 类型域，或不可变（immutable) 类的域（如果类 中的每个方法都不会改变其对象， 这种类就是不可变的类。

**程序清单3.2：**

**要求：**

1，在程序清单3.1的基础上将年龄属性修改为final类型

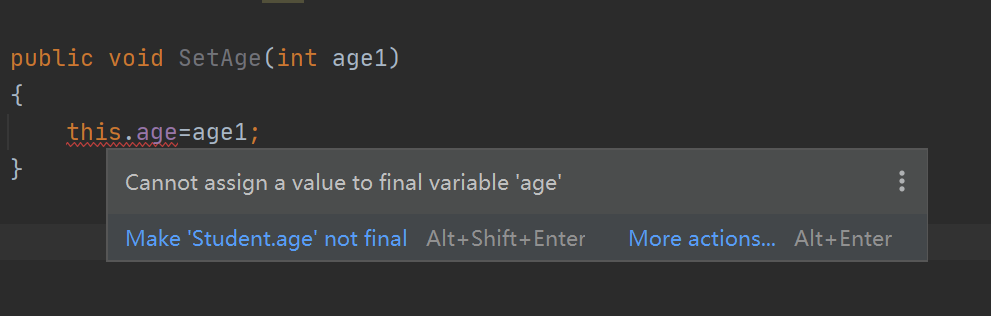
2，验证年龄属性是否能修改（通过Student类中的set方法）

代码：

学生类：

package com.dong.day5.student;  
  
public class Student {  
 public String name;  
 public String sex;  
 private final int age=12;  
   
 public void SetAge(int age1)  
 {  
 this.age=age1;  
 }  
}

验证结果：



#### 3.2.7 静态域和静态方法

##### 3.2.7.1 静态域static

如果将**域（类的属性）**定义为 static。一个类的所有对象共享一个静态域。

static方法就是没有this的方法。在static方法内部不能调用非静态方法，反过来是可以的。而且可以在没有创建任何对象的前提下，仅仅通过类本身来调用static方法。这实际上正是static方法的主要用途。

方便在没有创建对象的情况下来进行调用（方法/变量）。

**程序清单3.3**

**要求：**

1，将程序清单3.1中的学生类中的年龄属性修改为静态域

2，在学生类中书写一个静态方法初始化学生年龄为12

3，验证在静态方法中能不能调用非静态域

**代码：**

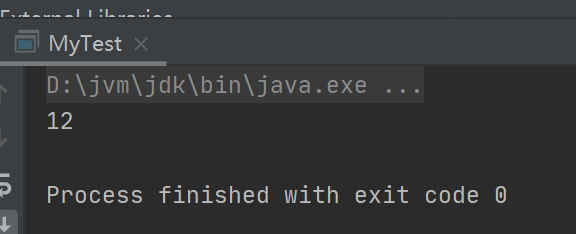
学生类：

package com.dong.day5.student;  
  
public class Student {  
 public static String *name*;  
 public String sex;  
 public static int *age*;  
  
 public static void setAge(int age) {  
 Student.*age* = age;  
 }  
}

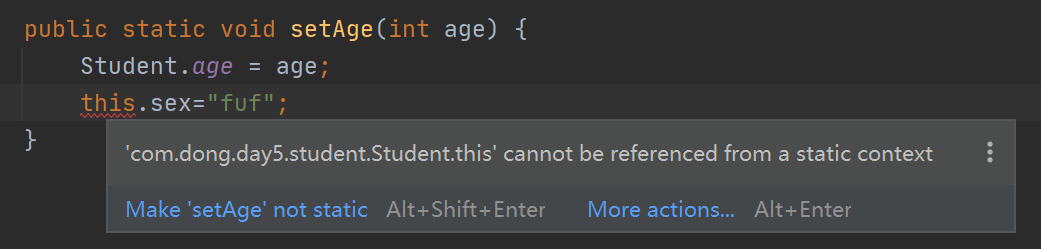
测试类：

package com.dong.day5.student;  
  
public class MyTest {  
 public static void main(String[] args) {  
 //定义对象  
  
 //实例化对象  
 Student.*setAge*(12);  
 System.*out*.println(Student.*age*);  
 }  
}

结果：



如果在静态方法中调用非静态属性会报错：



##### 3.2.7.2 静态常量

π=PI，就是一个静态常量。

System.out 也是一个静态常量。

##### 3.2.7.3 静态变量初始化的顺序

最先初始化的是 static 修饰的类属性，且只会执行一次，后面再运行这个类，类属性不会被执行其次成员属性初始化和代码块初始化处于同级，跟书写顺序有关 最后是构造方法初始化。

### 3.3方法参数

Java 程序设计语言总是采用**按值调用**。也就是说， 方法得到的是所有参数值的一个拷贝，特别是，**方法不能修改传递给它的任何参数变量的内容**。

数据类型有两种：

（1）**基本数据类型（数字、布尔值）**

例子：

public static void tripieValue(double x) // doesn't work

{

x = 3 \* x;

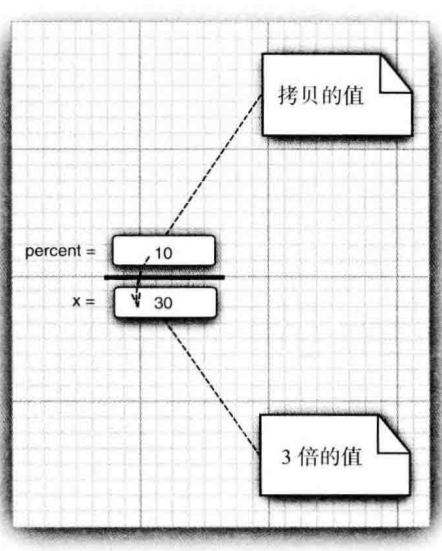
}

double percent = 10;

tripieValue(percent);

执行结果：prcent仍然是10；

结论：参数是单方面传递的。



**（2）对象引用：**

一个方法不能修改任何参数变量的值，但是对象引用就能实现。

例子：

public static void tripieSalary (Employee x) // works

{

x.raiseSa1ary(200);

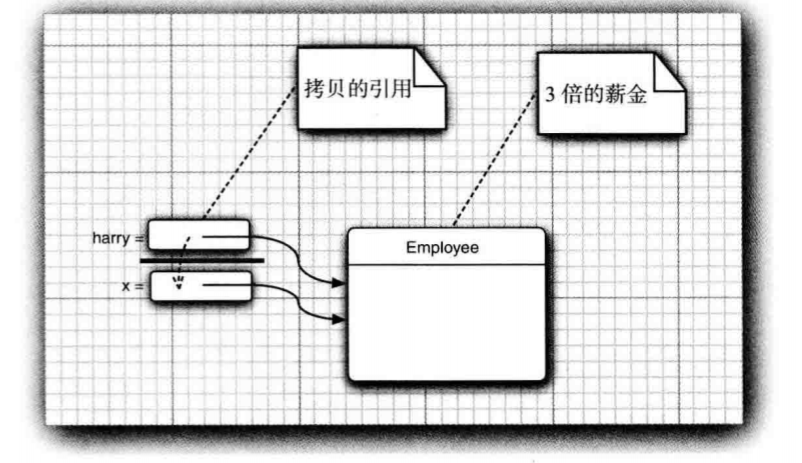
}

当调用：

harry = new Employee(. . .);

tripieSalary(harry);

就能将数值改变。



**注意**：对象引用不是C++中的引用调用。

**总结**：

•一个方法不能修改一个基本数据类型的参数（即数值型或布尔型）。

•一个方法可以改变一个对象参数的状态。

•一个方法不能让对象参数引用一个新的对象。

代码验证结论：

**package** shuru;

**import** java.time.\*;

**import** java.math.\*;

**public** **class** ShuRu

{

**public** **static** **void** main(String[] args)

{

//方法不能改变基本数据类型的参数

**int** ans=10;

*chengfa*(ans);

System.***out***.println(ans);

//方法能改变对象引用的值

Employee temp=**new** Employee("dkl",8000);

*duixianchengfa*(temp);

System.***out***.println("salary="+temp.getSalary());

//对象参数不能引用新的对象

Employee a=**new** Employee("cjh",7500);

Employee b=**new** Employee("tsq",7800);

*swap*(a,b);

System.***out***.println("name="+a.getName()+" salary="+a.getSalary());

System.***out***.println("name="+b.getName()+" salary="+b.getSalary());

}

**public** **static** **void** chengfa(**int** element)

{

element=element\*3;

}

**public** **static** **void** duixianchengfa(Employee x)

{

x.rainSalary(200);

}

**public** **static** **void** swap(Employee x,Employee y)

{

Employee t=x;

x=y;

y=t;

}

}

**class** Employee

{ //员工的对象

**private** String name; //private指只有Employee类能用这些变量

**private** **double** salary;

**public** Employee(String n,**double** s)

{//类似C++的构造函数，java中称为构造器

//构造器应该和类同名

//构造器总是伴随着 new 操作符的执行被调用，

//构造器没有返回值

name=n;

salary=s;

}

//方法

**public** String getName() // public 意味着任何类的任何方法都可以调用

{

**return** name;

}

**public** **double** getSalary()

{

**return** salary;

}

**public** **void** rainSalary(**double** lilv)

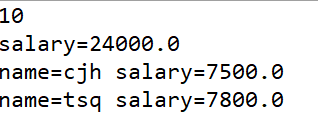
{//提高薪水

**double** rate=salary\*lilv/100;

salary+=rate;

}

}



### 3.4 对象的构造

#### 3.4.1 构造器

**什么是构造器？**

构造器就是和类名相同但无返回类型的方法。用于当前或某一对象的实例化，并将当前或某一对象返回。要得到一个类的实例时，往往是要运行其构造函数的。

**程序清单3.4**

**要求：**

1，建立一个学生类，学生类中有属性：姓名，年龄，性别

2，在学生类中建立一个构造器，构造器中有参数（姓名，年龄，性别）；

3，通过构造器构造一个李明对象，年龄12，性别男。

4，输出李明这个对象的信息

代码：

学生类：

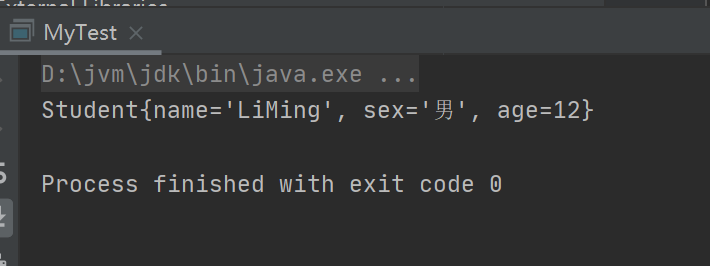
package com.dong.day5.student;  
  
public class Student {  
 private String name;  
 private String sex;  
 private int age;

//构造器  
 public Student(String name1,String sex1,int age1)  
 {  
 this.name=name1;  
 this.sex=sex1;  
 this.age=age1;  
 }  
 //将类对象以字符串的形式输出  
  
 @Override  
 public String toString() {  
 return "Student{" +  
 "name='" + name + '\'' +  
 ", sex='" + sex + '\'' +  
 ", age=" + age +  
 '}';  
 }  
}

测试类：

package com.dong.day5.student;  
  
public class MyTest {  
 public static void main(String[] args) {  
 //定义对象和实例化对象  
 Student LiMing=new Student("LiMing","男",12);  
 System.*out*.println(LiMing);  
  
 }  
}

结果：



#### 3.4.2 重载

有些类有多个构造器。

1．构造一个空对象：

StringBuilder messages = new StringBuilder();

2. 指定一个初始字符串:

StringBuilder todoList = new StringBuilder(“To do:\n")；

**定论**：有多个构造器的特征叫做**重载**。

#### 3.4.3 默认初始化

如果在构造器中没有显式地给域赋予初值，那么就会被自动地赋为默认值： 数值为 0、 布尔值为 false、 对象引用为 null。

#### 3.4.4 无参构造器

无参数构造器会将对象的值进行适当的默认值处理。

代码展示：

public Employee()

{

name = salary = 0;

hireDay = LocalDate,now();

}

如果在编写一个类时没有编写构造器， 那么系统就会提供一个无参数构造器。

注意：如果已经有构造器就不能够使用默认构造器。

#### 3.4.5 调用另外的构造器

一个类中可以有多个构造器，多个构造器间可以有联系，通过关键词this可以互相调用。

**语法格式：this(参数名,……)**

**程序清单3.5：**

**要求：**

**1，创建一个学生类，学生类中有属性：姓名，年龄，性别。有两个构造器，第一个构造器构造姓名，第二个构造器构造所有属性，但是第二个构造器需要调用第一个构造器。**

**2，创建对象李明，并输出李明对象。**

**代码：**

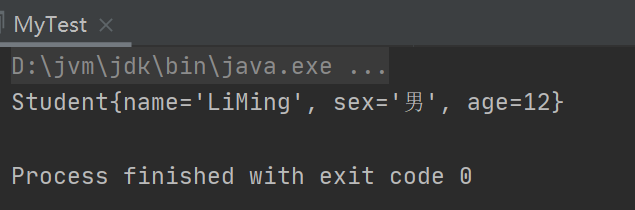
**学生类：**

**package com.dong.day5.student;  
  
public class Student {  
 private String name;  
 private String sex;  
 private int age;  
  
 //第一个构造器  
 public Student(String name1)  
 {  
 this.name=name1;  
 }  
 //第二个构造器  
 public Student(String name1,String sex1,int age1)  
 {  
 this(name1);  
 this.sex=sex1;  
 this.age=age1;  
 }  
  
 //将类对象以字符串的形式输出  
 @Override  
 public String toString() {  
 return "Student{" +  
 "name='" + name + '\'' +  
 ", sex='" + sex + '\'' +  
 ", age=" + age +  
 '}';  
 }  
}**

**测试类：**

**package com.dong.day5.student;  
  
public class MyTest {  
 public static void main(String[] args) {  
 //定义对象和实例化对象  
 Student LiMing=new Student("LiMing","男",12);  
 System.*out*.println(LiMing);  
  
 }  
}**

**结果：**



#### 3.4.6 初始化块

初始化块是类的第4种成员(成员变量、方法、构造器)，与构造器作用有些类似，用于对JAVA对象进行初始化操作

例如：

public class Animal {

//定义初始化块

{

int a = 6;

if(a>4) {

System.out.println("Animal初始化块，局部变量a的值大于4");

}

System.out.println("Animla的初始化块");

}

//再定义一个初始化块

{

System.out.println("Animal的第二个初始化块");

}

//定义构造器

public Animal() {

System.out.println("Animal的无参构造器");

}

}

测试类

public class Demo01 {

public static void main(String[] args) {

Animal a = new Animal();

}

}

上述代码输出结果为：

Animal初始化块，局部变量a的值大于4  
Animla的初始化块  
Animal的第二个初始化块  
Animal的无参构造器

**小结（初始化块和构造器的执行顺序）：**

1，JAVA创建对象时，首先执行初始化块。如果一个类中定义了多个初始化块，则按照顺序执行(这其实没有什么意义，因为初始化块都可以合并为一个)。并且，初始化块在构造器之前执行。

2，首先是各个类的静态初始化块先按照先父类后子类的顺序先后执行；然后再次由父类到子类开始依次执行它们的普通初始化块和无参构造器，若是子类初始化中需要为父类传参，那么在执行子类普通初始化内容前就要先执行父类的含参构造函数

#### 3.4.7 对象析构和finalize方法

**关键词**finalize 方法：。finalize 方法将在垃圾回收器清除对象之前调用。在实际应用中，不要依赖于使用 finalize 方法回收任何短缺的资源， 这是因为很难知道这个 方法什么时候才能够调用。

### 3.5 包

什么是包？

包中储存着每个程序员组织的类。可以方便地组织自己的代码，并将自己的代码与别人提供的代码库分开管理。

#### 3.5.1 类的导入

一个类可以使用所属包中的所有类和不同包中的公有类。

访问方式：

1，在调用类前书写完整包名

2，在程序开头：使用 import 语句，如import java.util .\*; 调用java.util包中的公有类。

#### 3.5.2 静态导入static import

要使用静态成员（方法和变量）我们必须给出提供这个静态成员的类。

　　使用**静态导入**可以使被导入类的**静态变量**和**静态方法**在当前类**直接可见**，使用这些静态成员无需再给出他们的类名。

　　静态导入也是JDK5.0引入的新特性，下面以实例来说明静态导入的用法：

程序清单3.6：

要求：

实现静态导入

代码：

**在一个包中定义一个这样的类：**

package com.example.learnjava;

public class Common

{

public static final int AGE = 10;

public static void output()

{

System.out.println("Hello World!");

}

}

**在另外一个包中定义的一个类（不使用静态导入）：**

package com.example.learnjava2;

import com.example.learnjava.Common;

public class StaticImportTest

{

public static void main(String[] args)

{

int a = Common.AGE;

System.out.println(a);

Common.output();

}

}

**在另外一个包中定义的一个类（使用静态导入）：**

静态导入的语法是：

**import static 包名.类名.静态成员变量;**

**import static 包名.类名.静态成员函数;**

**代码：**

package com.example.learnjava2;

import static com.example.learnjava.Common.AGE;

import static com.example.learnjava.Common.output;

public class StaticImportTest

{

public static void main(String[] args)

{

int a = AGE;

System.out.println(a);

output();

}

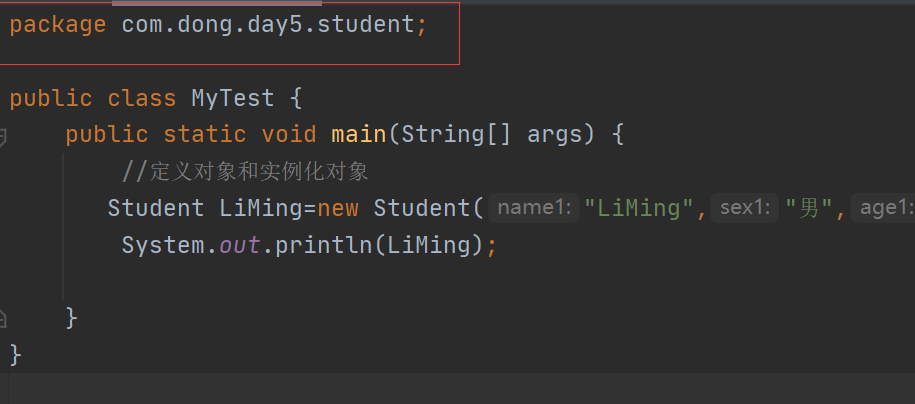
}

小结：静态导入有缺点：过度地使用静态导入会在一定程度上降低代码的可读性。

#### 3.5.3 将类放入包中

要想将一个类放人包中， 就必须将包的名字放在源文件的开头，包中定义类的代码之前.

例如：**注意红框中标记的模式和位置**



#### 3.5.4 包作用域

前面已经接触过访问修饰符 public 和 private。标记为 public 的部分可以被任意的类使 用；标记为 private 的部分只能被定义它们的类使用。如果没有指定 public 或 private , 这 个 部 分（类、方法或变量）可以被同一个包中的所有方法访问。

## 四，继承

**什么是继承？**

继承已存在的类就是复用（继承）这些类的方法和域。

**域**：设立的私有变量

### 4.1 类，超类和子类

#### 4.1.1 定义子类

**关键词extends**：表示继承，关键字 extends 表明正在构造的新类派生于一个已存在的类。

上述的**已存在的类**被定义为超类或称为父类。

**程序清单4.1:**

**要求**：

1,建立父类“动物”，建立子类“狗”，狗继承动物

2，构造一个狗的对象并输出。

代码：

**动物类：**

package com.dong.day6.JC;  
  
public class Animal {  
 String sound;  
 public Animal(String sound1)  
 {  
 this.sound=sound1;  
 }  
  
 @Override  
 public String toString() {  
 return "Animal{" +  
 "sound='" + sound + '\'' +  
 '}';  
 }  
}

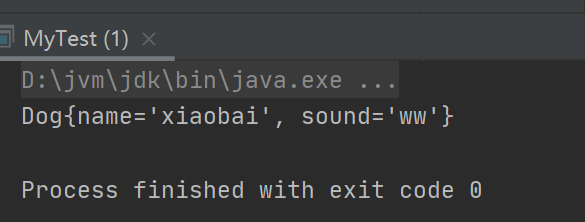
狗类：

package com.dong.day6.JC;  
  
public class Dog extends Animal{  
 String name;  
 public Dog(String name1,String sound)  
 {  
 super(sound);  
 this.name=name1;  
 }  
  
 @Override  
 public String toString() {  
 return "Dog{" +  
 "name='" + name + '\'' +  
 ", sound='" + sound + '\'' +  
 '}';  
 }  
}

测试类：

package com.dong.day6.JC;  
  
public class MyTest {  
 public static void main(String[] args) {  
 Dog XiaoBai=new Dog("xiaobai","ww");  
 System.*out*.println(XiaoBai);  
 }  
}

结果：



#### 4.1.2 覆盖方法（重写）

**问题**：并不是每一个超类的方法都是子类所需要的。

**解决方法**：覆盖此超类的方法，**关键词：super**，由于超类的域都是私有域，因此在子类中不能调用，但是存在public的方法，由于方法名在两个类中重复，于是super就用于调用超类的方法去调用超类的私有域。

**注意**：在子类中可以增加域、 增加方法或覆盖超类的方法。

**程序清单4.2：**

**要求**：

1，在清单4.1动物类中添加“发声次数”的方法，子类中对发声这个方法进行重写，然后输出。

**代码：**

动物类：

package com.dong.day6.JC;  
  
public class Animal {  
 String sound;  
 public Animal(String sound1)  
 {  
 this.sound=sound1;  
 }  
  
 public int Count()  
 {  
 return 5;  
 }  
 @Override  
 public String toString() {  
 return "Animal{" +  
 "sound='" + sound + '\'' +  
 '}';  
 }  
}

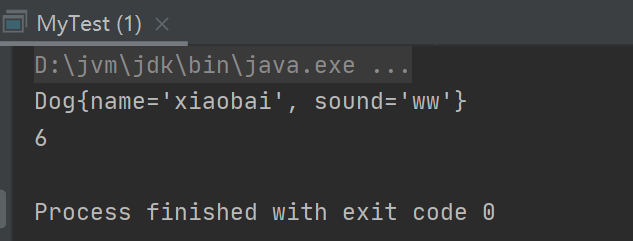
狗类：

package com.dong.day6.JC;  
  
public class Dog extends Animal{  
 String name;  
 public Dog(String name1,String sound)  
 {  
 super(sound);  
 this.name=name1;  
 }  
  
 public int Count()  
 {  
 return super.Count()+1;  
 }  
 @Override  
 public String toString() {  
 return "Dog{" +  
 "name='" + name + '\'' +  
 ", sound='" + sound + '\'' +  
 '}';  
 }  
}

测试类：

package com.dong.day6.JC;  
  
public class MyTest {  
 public static void main(String[] args) {  
 Dog XiaoBai=new Dog("xiaobai","ww");  
 System.*out*.println(XiaoBai);  
 System.*out*.println(XiaoBai.Count());  
 }  
}

结果：



#### 4.1.3 子类构造器

**问题**：子类也需要构造器，但是超类的域是私有的，子类不能直接调用？

解决办法：super(int,String……)；//此行代码写在子类构造器的首行

//super是调动子类的上一级父类的构造器

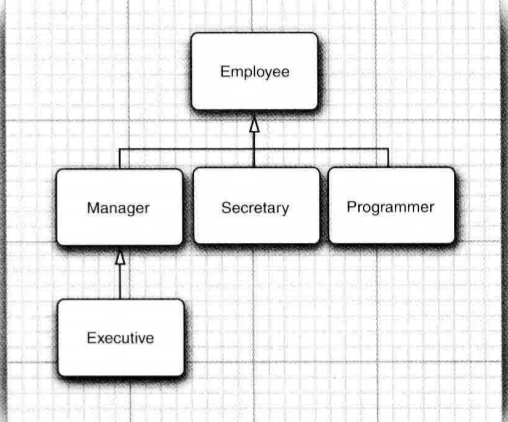
//将调用超类的拥有相应参数构造器

注意：如果子类的构造器没有显式地调用超类的构造器， 则将自动地调用超类默认（没有参数 ) 的构造器。如果超类没有无参数的构造器，java就会报错。

**程序实现请参照程序清单4.1.**

#### 4.1.4 继承的层次

继承并不仅限于一个层次。



在继承层次中， 从某个特定的类到其祖先的路径被称为该类的继承链。

**注意**：单继承就是一个类只可以继承自一个父类,多继承是指一个类可以同时继承多个父类，java不支持多继承。

#### 4.1.5 多态

1）多态性是面向对象编程的又一个重要特征，它是指在父类中定义的属性和方法被子类继承之后，可以具有不同的数据类型或表现出不同的行为，这使得同一个属性或方法在父类及其各个子类中具有不同的含义。**每个子类的对象都是超类的对象**。因此超类的对象引用可以引用其所有子类的对象。

2）对面向对象来说，多态分为编译时多态和运行时多态。编译时多态是静态的，主要是指方法的重载，运行时多态是动态的，它是通过动态绑定来实现的，也就是第一条所说的多态性。

3）[Java](http://c.biancheng.net/java/) 实现多态有 3 个必要条件：**继承、重写和向上转型。**

* **继承**：在多态中必须存在有继承关系的子类和父类。
* **重写**：子类对父类中某些方法进行重新定义，在调用这些方法时就会调用子类的方法。
* **向上转型**：在多态中需要将子类的引用赋给父类对象，只有这样该引用才既能可以调用父类的方法，又能调用子类重写父类的方法。

**程序清单4.3：**

**要求：**

验证多态性。

**代码：**

父类：

package com.dong.day6.JC;  
  
public class Animal {  
 private String sound;  
 public Animal(String sound1)  
 {  
 this.sound=sound1;  
 }  
  
 public int Count()  
 {  
 return 5;  
 }  
  
 public String getSound() {  
 return sound;  
 }  
  
 @Override  
 public String toString() {  
 return "Animal{" +  
 "sound='" + sound + '\'' +  
 '}';  
 }  
}

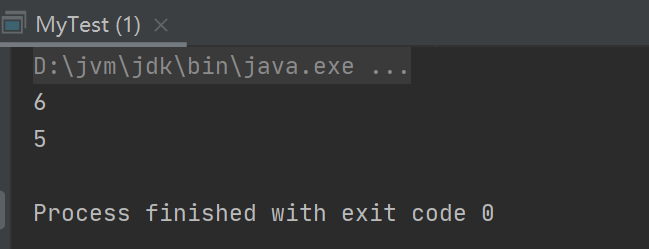
子类：

package com.dong.day6.JC;  
  
public class Dog extends Animal{  
 String name;  
 public Dog(String name1,String sound)  
 {  
 super(sound);  
 this.name=name1;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public int Count()  
 {  
 return super.Count()+1;  
 }  
 @Override  
 public String toString() {  
 return "Dog{" +  
 "name='" + name + '\'' +  
 ", sound='" + super.getSound()+ '\'' +  
 '}';  
 }  
}

测试类：

package com.dong.day6.JC;  
  
public class MyTest {  
 public static void main(String[] args) {  
 //子类的对象给父类引用，验证“所有的子类对象都是父类的对象”  
 //向上转型  
 Animal animal=new Dog("haha","ww");  
   
 System.*out*.println(animal.Count());  
 //没有向上转型  
 animal=new Animal("ww");  
 System.*out*.println(animal.Count());  
 }  
}

结果：



#### 4.1.5理解方法的调用机制

1）弄清楚如何在对象上应用方法调用非常重要。对象要调用的方法名在类和超类中可能有多个（即名字相同参数不同），**编译器会在多个相同名字的方法中筛选参数类型相同的或可转化类型的**。如果最终筛选的结果不唯一，编译器将报错。

2）如果是 private 方法、 static 方法、 final 方法，）或者构造器， 那么编译器将可以准确地知道应该调用哪个方法， 我们将这种调用方式称 为静态绑定。

3）当程序运行，并且采用动态绑定调用方法时， 虚拟机一定调用与 x 所引用对象的实际类型最合适的那个类的方法。

#### 4.1.6 阻止继承，final类或方法

1）不允许扩展的类被称为 final 类。如果 在定义类的时候使用了 final 修饰符就表明这个类是 final 类。

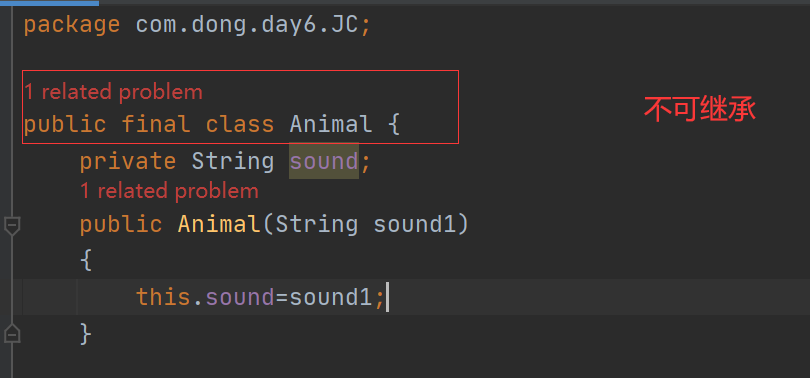
例：public final class Executive extends Manager{…………}

2）类中的特定方法也可以被声明为 final。如果这样做，子类就不能覆盖这个方法。（如果是final类，那么此类中的方法都是final方法）。

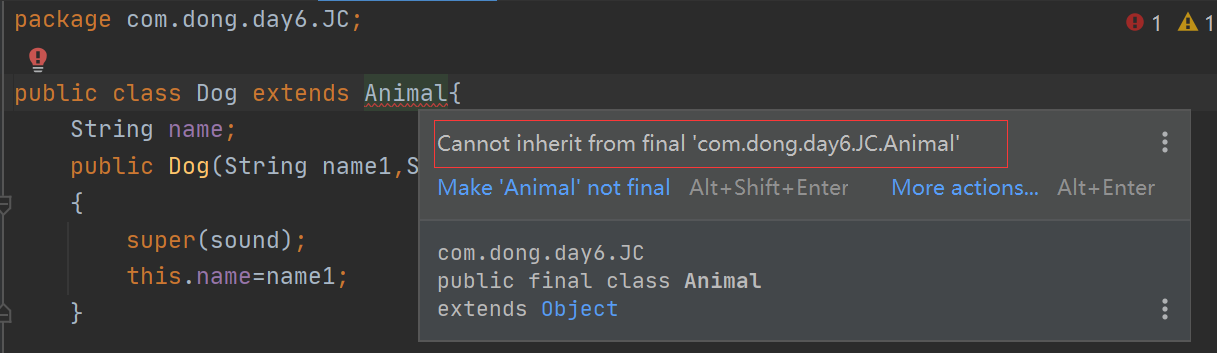
**将方法或类声明为 final 主要目的是**： 确保它们不会在子类中改变语义

举例说明：

**Final类：**



继承报错：



#### 4.1.7 抽象类

关键词：abstract

**抽象类产生的目的**：由于父类的某些方法定义不明确，就抽象处理，由其继承的子类去具体实现，如父类是动物，已知的属性有动物名字，动物年龄，但是动物的叫声是未知的，于是就可以将叫声这个方法抽象，由子类实现。

**抽象类的特点：**

1）包含一个或多个抽象方法的类本身必须被声明为抽象的。

2）抽象类不能被实例化，也就是说，如果将一个类声明为abstract, 就不能创建这个类的对 象

3）抽象类中的抽象方法在子类中都一定要去实现。

4) 抽象类除了不能实例化对象之外，类的其它功能依然存在，成员变量、成员方法和构造方法的访问方式和普通类一样。

5) 抽象类、抽象方法不能被private、final、static修饰

**程序清单4.4：**

**要求**：

1,建立抽象类动物，子类狗

2,验证抽象类的特性

代码：

动物抽象类：

package com.dong.day6.JC2;  
  
  
 public abstract class Animal {  
 String name;  
  
 public Animal(String name1)  
 {  
 this.name=name1;  
 }  
 public abstract void Sound() ;  
}

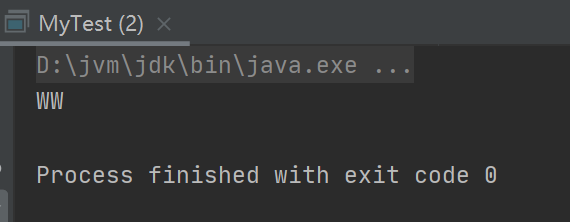
子类：

package com.dong.day6.JC2;  
  
public class Dog extends Animal{  
 private String sound;  
  
 public Dog(String name1, String sound) {  
 super(name1);  
 this.sound = sound;  
 }  
 public void Sound()  
 {  
 System.*out*.println("WW");  
 }  
}

测试类：

package com.dong.day6.JC2;  
  
public class MyTest {  
 public static void main(String[] args) {  
 //抽象类不能实例化对象  
 Animal animal=new Dog("hehe","ww");  
 animal.Sound();  
 }  
}

结果：



#### 4.1.8 强制转化

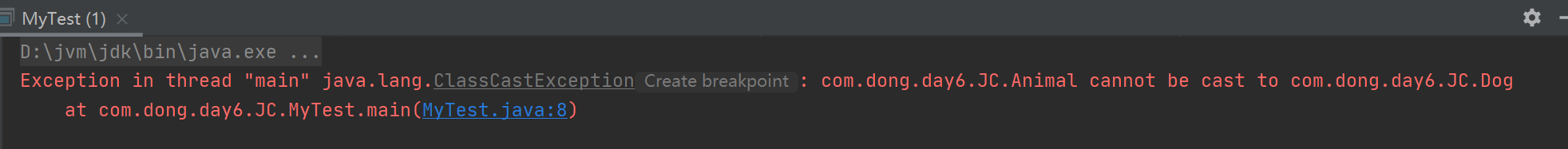
将父类的对象给子类引用就需要使用**强制转化**。这里的强制转化和前面说明的数据类型的强制转化的机制是相同的。

强制转化的三种情况：

**1，父类转化为子类**

Animal animal1=new Animal("haha");  
Dog dog1=(Dog) animal1;

**结果：**



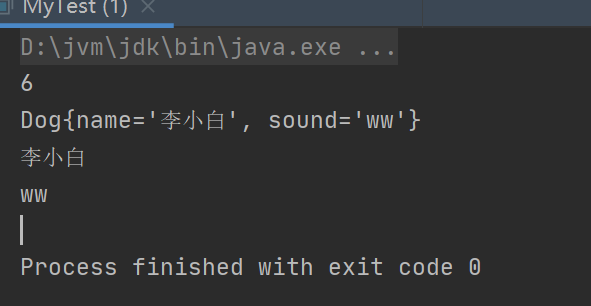
**说明：**

创建一个父类的实例，想要强制把父类对象转换成子类的，不行！通俗的想，真正当爹的永远不可能装儿子。

**2，假父类转化为子类**

package com.dong.day6.JC;  
  
public class MyTest {  
 public static void main(String[] args) {  
 //子类的对象给父类引用，验证“所有的子类对象都是父类的对象”  
 //向上转型  
 Animal animal=new Dog("李小白","ww");  
 Dog dog=(Dog)animal;  
 System.*out*.println(dog.Count());  
 System.*out*.println(dog);  
 System.*out*.println(dog.getName());  
 System.*out*.println(dog.getSound());  
  
 }  
}

**结果：**



**说明：**

父类对象引用着一个子类实例。  
Son类特有的属性**暂时**不能通过 f 来操作，因为Father类没有Son类（子类）的特有属性。  
接着创建子类对象 s，它引用的是父类对象 f 强制转换来的对象（其实就是个装爹的Son，把他强制转回了Son），这时就可以通过 s 来操作子类的特有属性了。

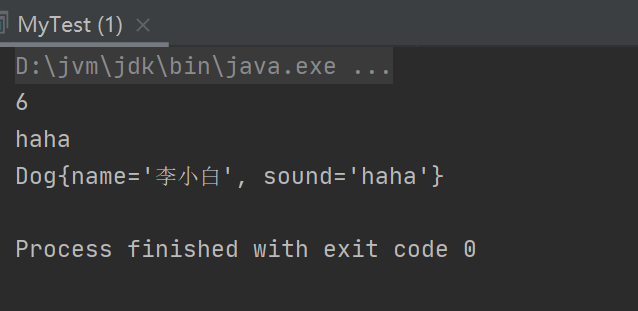
通俗的说就是儿子装爹，终究是儿子，本质没变，还是可以把他强制转回儿子的。

**第三种情况：将孩子类转化为父类**

**代码：**

package com.dong.day6.JC;  
  
public class MyTest {  
 public static void main(String[] args) {  
 //子类的对象给父类引用，验证“所有的子类对象都是父类的对象”  
 //向上转型  
 Dog dog = new Dog("李小白","haha");  
 Animal animal=(Animal)dog;  
 System.*out*.println(animal.Count()); //重写后的方法  
 System.*out*.println(animal.getSound());  
 System.*out*.println(animal);  
  
 }  
}

结果：



**分析：**

**子类转换成父类，只是子类对象的特有属性无法利用 f 操作，f 可以操作其非特有的属性（由父类继承而来的属性）。**

**通俗的说，儿子和爹的共同点——“都是人”，儿子是人是从父亲继承而来的，他们都有人的基本行为，只是儿子永远不可能和父亲平辈（子类类型转换成父类类型）。**

### 4.2 Object类的了解

Object 类是 Java 中所有类的始祖， 在 Java 中每个类都是由它扩展而来的。

表现：

1）可以使用 Object 类型的变量引用任何类型的对象

2）所有的数组类型，不管是对象数组还是基本类型的数组都扩展了 Object 类。

#### 4.2.1 equals方法

Object类中的equals方法用于检测一个对象是否等于另外一个对象。在Object类中，这个方法将判断两个对象是否具有相同的引用。如果两个对象具有相同的引用，它们一定是相等的。从这点上看，将其作为默认操作也是合乎情理的。然而，对于多数类来说，这种判断并没有什么意义。例如，采用这种方法比较两个PrintStream对象是否相等就完全没有意义。然而，经常需要检测两个对象状态的相等性，如果两个对象的状态相等，就认为这两个对象是相等的。

**下面展示equals方法运行机制**

**代码：**

public class Employee{

...

public boolean equals(Object otherObject){

//快速检查对象是否相同

if(this==otherObject) return true;

//如果引入参数为空，则必须返回false

if(otherObject==null) return false;

//如果类不匹配，它们就不能相等。

if(getClass()!=otherObject.getClass())

return false;

//现在我们知道另一个对象是非空雇员

Employee other =(Employee)otherObject;

//测试字段是否具有相同的值

return Object.equals(other.name);

&& salary==other.salary

&& hireDay.equals(other.hireDay);

}

}

**说明：**

getClass方法将返回一个对象所属的类，有关这个方法的详细内容稍后进行介绍。

Java语言规范要求equals方法具有下面的特性：

1) 自反性：对于任何非空引用x，x.equals(x)应该返回true。

2) 对称性：对于任何引用x和y，当且仅当y.equals(x)返回true，x.equals(y)也应该返回true。

3) 传递性：对于任何引用x，y和z，如果x.equals(y)返回true，y.equals(z)返回true，x.equals(z)也应该返回true。

4) 一致性：如果x和y引用的对象没有发生变化，反复调用x.equals(y)应该返回同样的结果。

5) 对于任何非空引用x，x.equals(null)应该返回false

#### 4.2.2 hashcode方法

1，hashCode方法定义在 Object 类中， 因此每个对象都有一个默认的散列码，其值为 对2，象的存储地址。

3，hashCode 方法应该返回一个整型数值（也可以是负数）。

**4，如果重新定义 equals方法，就必须重新定义 hashCode 方法。**

**程序清单4.5：**

**要求：**

输出狗类的hashcode

**代码：**

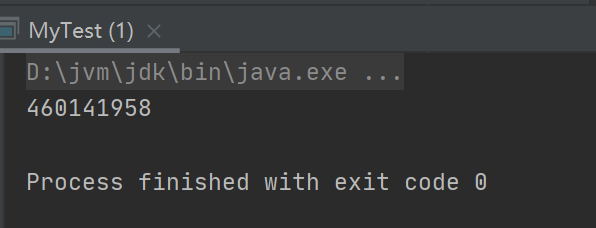
狗类：

package com.dong.day6.JC;  
  
public class Dog extends Animal{  
 String name;  
 public Dog(String name1,String sound)  
 {  
 super(sound);  
 this.name=name1;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public int Count()  
 {  
 return super.Count()+1;  
 }  
 @Override  
 public String toString() {  
 return "Dog{" +  
 "name='" + name + '\'' +  
 ", sound='" + super.getSound()+ '\'' +  
 '}';  
 }  
}

测试类：

package com.dong.day6.JC;  
  
public class MyTest {  
 public static void main(String[] args) {  
 //子类的对象给父类引用，验证“所有的子类对象都是父类的对象”  
 //向上转型  
 Dog dog = new Dog("李小白","haha");  
 System.*out*.println(dog.hashCode());  
  
 }  
}

**结果：**



#### 4.2.3 clone方法

clone意思是克隆、复制。在Java语言中，当对象调用clone()方法时，就会复制已有的对象。

**通俗的讲：**

对于任意一个对象x，表达式①x.clone != x将会是true；表达式②x.clone().getClass()==x.getClass()将会是true，但不是绝对的。通常情况下，表达式③x.clone().equals(x)将会是true，但是这也不是绝对的。

##### 4.2.3.1 Cloneable接口（看完接口后回看）

 要使类具有克隆能力时，需要实现Cloneable接口，实现它的目的是作为一个对象的一个(混入)接口，表明这个对象是允许克隆的。

**它的源码如下:**

public interface Cloneable {

}

 可以看出Cloneable是一个空接口(标记接口)，它决定了Object中受保护的clone方法的实现行为：如果一个类实现了Cloneable接口，Object的clone方法就返回这个对象的逐域拷贝，否则就抛出CloneNotSupportedException异常。如果实现了这个接口，类和它所有的超类都**无需调用构造器就可以创建对象**。下面是一个简单的clone方法应用。

**程序清单4.6：**

**要求**：

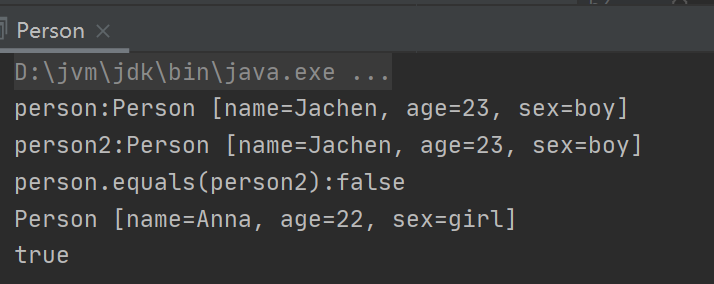
1，创建一个Person类实现Cloneable接口，重写clone方法。

2，构建两个对象person1和person2，将person1的内容克隆给person2.判断person1和person2的对象是否相同，判断person2和person1的class类是否相同

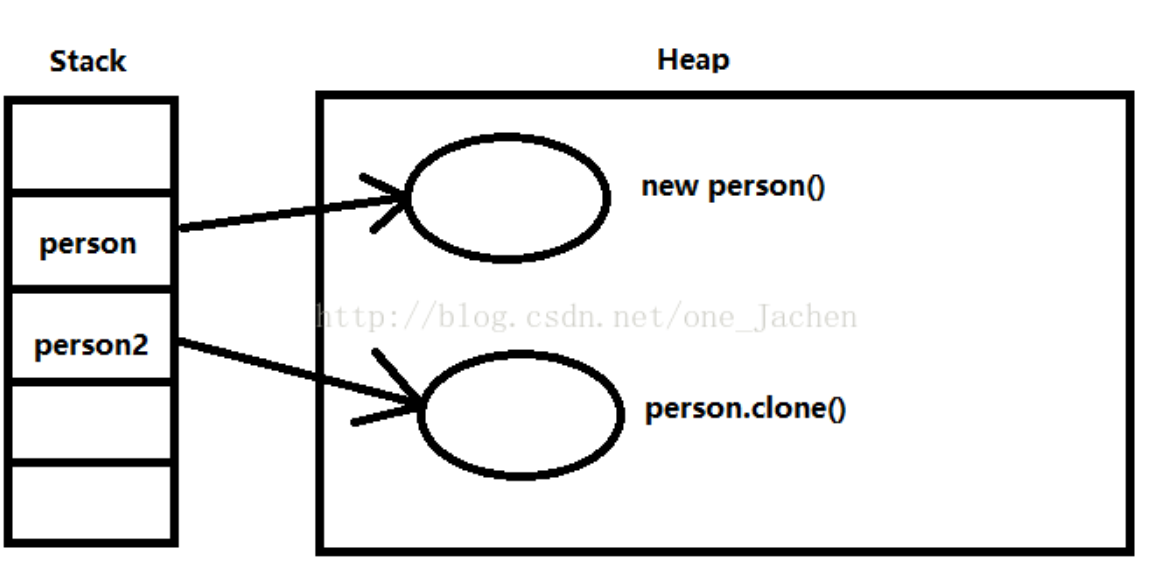
**代码**：

package com.dong.day6.JC3;  
  
public class Person implements Cloneable {  
 private String name;  
 private Integer age;  
 private String sex;  
  
 public Person() {  
 super();  
 }  
  
 public Person(String name, Integer age, String sex) {  
 super();  
 this.name = name;  
 this.age = age;  
 this.sex = sex;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public void setAge(Integer age) {  
 this.age = age;  
 }  
  
 public void setSex(String sex) {  
 this.sex = sex;  
 }  
  
 @Override  
 public String toString() {  
 return "Person [name=" + name + ", age=" + age + ", sex=" + sex + "]";  
 }  
  
 public Person clone() throws CloneNotSupportedException {  
 return (Person) super.clone();  
 }  
  
 public static void main(String[] args) {  
 Person person = new Person("Jachen", 23, "boy");  
 System.*out*.println("person:" + person);  
 try {  
 Person person2 = person.clone();  
 System.*out*.println("person2:" + person2);  
 System.*out*.println("person.equals(person2):"+ person2.equals(person));  
 person2.setName("Anna");  
 person2.setSex("girl");  
 person2.setAge(22);  
 System.*out*.println(person2);  
 System.*out*.println(person2.getClass() == person.getClass());  
 } catch (CloneNotSupportedException e) {  
 e.printStackTrace();  
 }  
 }  
}

**结果：**



**由运行结果可知，当person对象调用clone方法后复制了一个跟它“相似”的对象，可是调用equals方法后，得知他们在堆中的地址是不同的(**[**JVM中堆、栈简介**](http://blog.csdn.net/one_jachen/article/details/78157209)**)，内存图结构如下：**



##### 4.2.3.2 Object.clone()的运行效果（了解）

 根类Object中的clone()方法负责建立正确的存储容量，并通过“按位复制”将所有二进制从原始对象中复制到新对象的存储空间。也就是说，它并不只是预留存储空间以及复制一个对象--实际需要调查出欲复制的新对象准确大小，然后再复制那个对象。由于这些工作都是由根类定义的clone()方法内部代码进行的，这个过程需要用RTTI（运行时类型鉴定）来判断欲克隆对象的实际大小。采用这种方式，clone()方法便可建立起正确数量的存储空间，并对那个类型进行正确的按位复制。

        克隆过程的第一个部分通常都应该是调用super.clone()。通过进行一次准确的复制，这样做可为后续的克隆进程建立起一个良好的基础。随后，可采取另一些必要的操作，以完成最终的克隆。

        通常可在从一个能克隆的类里调用 super.clone()，以确保所有基础类行动（包括 Object.clone()）能够进行。随着是为对象内每个句柄都明确调用一个 clone()；否则那些句柄会别名变成原始对象的句柄。构建器的调用也大致相同——首先构造基础类，然后是下一个衍生的构建器……以此类推，直到位于最深层的衍生构建器。区别在于 clone()并不是个构建器，所以没有办法实现自动克隆。为了克隆，必须由自己明确进行。

        若新建一个类，它的基础类会默认为Object，并默认为不具备克隆能力。只要不明确地添加克隆能力，这种能力便不会自动产生。但我们可以在任何层添加它，然后便可从那个层开始向下具有克隆能力。

##### 4.2.3.4 深克隆

上面讲的是“浅复制”，接下来我们来讲解“深复制”。如果对象中包含的域引用了可变的对象，使用上述这种简单的clone实现（super.clone()）可能会导致灾难性的后果。

**程序清单4.7：**

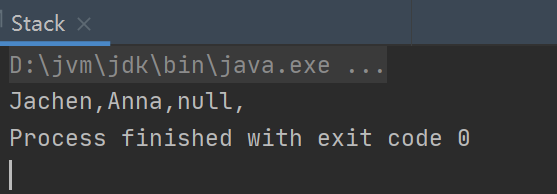
**要求：**

浅克隆的不规范设计问题

**代码：**

package com.dong.day6.JC3;  
  
  
import java.util.Arrays;  
import java.util.EmptyStackException;  
//栈类定义  
public class Stack implements Cloneable{  
  
 private Object[] elements;//用数组来存储栈的元素  
 private int size = 0;//栈中元素的个数  
 private static final int *DEFAULT\_INITIAL\_CAPACITY* = 16;//默认初始化大小为16  
  
 public Stack(){  
 this.elements = new Object[*DEFAULT\_INITIAL\_CAPACITY*];  
 }  
 //压入栈  
 public void push(Object e){  
 ensureCapacity();  
 elements[size++] = e;  
 }  
 //弹出栈信息  
 public Object pop(){  
 if(size == 0){  
 throw new EmptyStackException();  
 }  
 Object result = elements[--size];  
 elements[size] = null;  
 return result;  
 }  
 //确保栈的容量  
 private void ensureCapacity(){  
 if(elements.length == size){  
 elements = Arrays.*copyOf*(elements, 2\*size + 1);  
 }  
 }  
 //克隆栈对象  
 public Stack clone () throws CloneNotSupportedException{  
 return (Stack) super.clone();  
 }  
  
 public static void main(String[] args) throws CloneNotSupportedException {  
 Stack stack = new Stack();  
 stack.push("Jachen");  
 stack.push("Anna");  
 stack.push("Jack");  
 Stack cloneStack = stack.clone();  
 cloneStack.pop();  
  
 for(int i = 0;i<stack.size;i++){  
 System.*out*.print(stack.elements[i] + ",");  
 }  
 }  
}

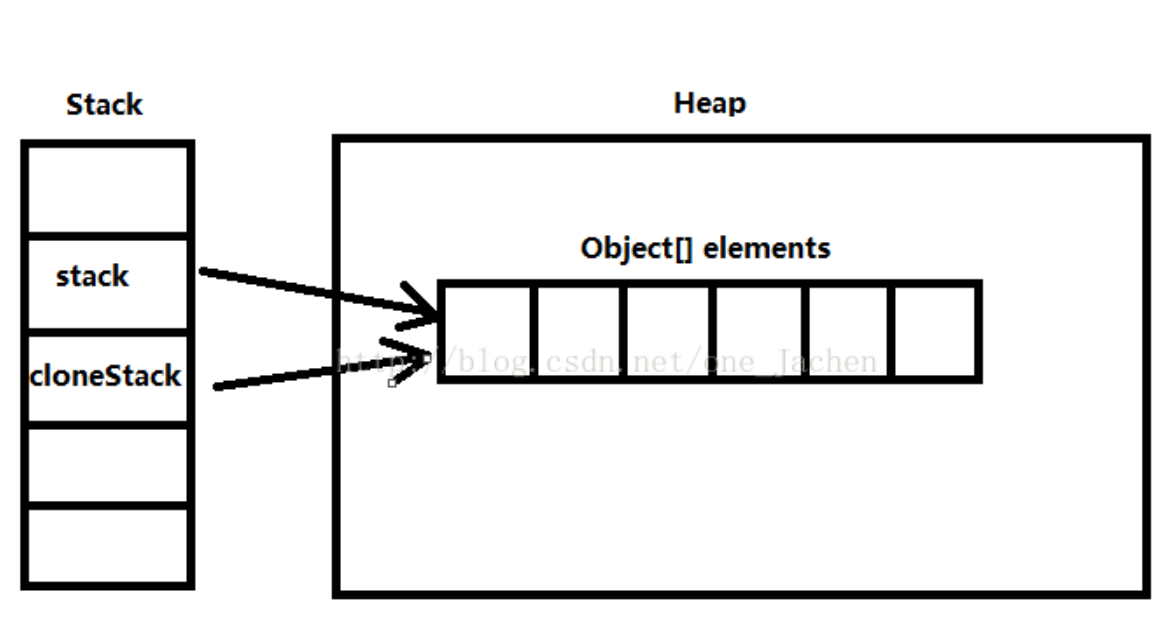
**结果**：



**分析：**

当clone方法仅仅返回super.clone()；得到这样的Stack实例：其size域具有正确的值，但是它的elements域将引用于原始Stack实例相同的数组。

**图示：**



   如果调用Stack类中唯一的构造器，这种情况就永远不会发生。实际上，**clone方法就是另一个构造器，你必须确保它不会伤害到原始的对象，并确保正确地创建被克隆对象中的约束条件。**为了使Stack类中clone方法正常工作，它必须要拷贝栈内部的信息，clone方法中代码修改如下：

public Stack clone() throws CloneNotSupportedException{  
 Stack stack = (Stack)super.clone();  
 stack.elements = elements.clone();  
 return stack;  
}

#### 4.2.3 toString方法

ToString是object类中的一个重要方法：它用于返回表示对象值的字符串，类名+散列码。

类似：

public String toString() //继承中的重写

{

return "Employee[name=" + name + ",salary: " + salary + “，hireDay=" + hireDay +“]”;

}

**结论**：toString方法是一种非常有用的调试工具。在标准类库中，许多类都定义了 toString方 法，以便用户能够获得一些有关对象状态的必要信息。

### 4.3 泛型数组列表

**问题**：在编译前数组的大小就要确定，这可能浪费储存空间，在java中如何解决？

**解决方法**：ArrayList类，此类在添加或删除元素时， 具有自动调节数组容量的功能。

**ArrayList使用方法**：ArrayList<Employee> staff = new ArrayList<Employee>(100);

//Employee是类，100是出定义空间大小

**ArrayList类中的方法：**

**1）add（）方法**

**语法格式**：

arraylist.add（int index，E element）

index是插入的位置；E是泛型，是指任意一种的类型；element是插入的数据

**2）size方法将返回数组列表中包含的实际元素数目**

**3) set方法**

**语法格式：**

ArrayList.set(index,E element)

**4)get()方法**

**语法格式：**

ArrayList.get(index)

**注意：index是从‘0’开始计数的。**

**程序清单4.8:**

**要求：**

1, 使用ArrayList类构建泛型数组列表，泛型限定为类Student

2, 向列表中添加两个对象，并输出列表中对象和对象数量

代码：

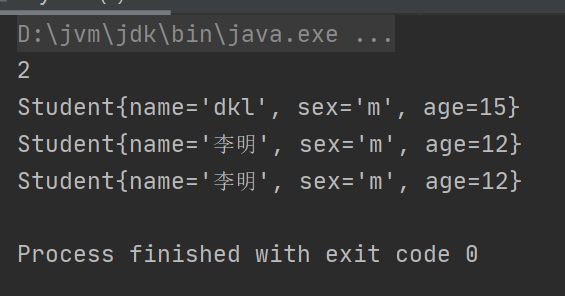
学生类：

package com.dong.day6.JC4;  
  
public class Student {  
 private String name;  
 private String sex;  
 private int age;  
  
 //构造器  
 public Student(String name, String sex, int age) {  
 this.name = name;  
 this.sex = sex;  
 this.age = age;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public String getSex() {  
 return sex;  
 }  
  
 public void setSex(String sex) {  
 this.sex = sex;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
  
 @Override  
 public String toString() {  
 return "Student{" +  
 "name='" + name + '\'' +  
 ", sex='" + sex + '\'' +  
 ", age=" + age +  
 '}';  
 }  
}

测试类：

package com.dong.day6.JC4;  
  
import java.lang.reflect.Array;  
import java.util.ArrayList;  
  
public class MyTest {  
 public static void main(String[] args) {  
 ArrayList<Student>students=new ArrayList<>(100);  
 //验证add函数  
 students.add(new Student("xwb","m",13));  
 students.add(new Student("李明","m",12));  
 //验证set函数  
 students.set(0,new Student("dkl","m",15));  
 System.*out*.println(students.size());  
 for(Student elem:students)  
 {  
 System.*out*.println(elem);  
 }  
 //验证get函数  
 System.*out*.println(students.get(1));  
  
 }  
}

结果：



### 4.4 对象包装器和自动装箱

#### 4.4.1 对象包装器

有时候，需要将int这样的基本类型转换为对象。所有的基本类型都有一个与之对应的类。通常，这些类被称为包装器（wrapper）。

这些对象包装类分别是：Integer、Long、Float、Double、Short、Byte、Character、Void和Boolean。

　　对象包装类是不可变的，即一旦构造了包装器，就不允许更改包装在其中的值。同时，对象包装器类还是final，因此不能定义它们的子类。

**例如：**如果想定义已给整型数组列表，但是尖括号中的类型参数不允许是基本类型，即不允许写成ArrayList<int>，这时，需要写成：

**ArrayList<Integer> list = new ArrayList();**

#### 4.4.2 自动装箱和自动拆箱

如果调用：

**list.add(3);**

　　编译器会把这条语句自动变成：

**list.add(Integer.valueOf(3));**

**这种变换被称为自动装箱。（autoboxing）**

对应的，当将一个Integer对象赋值给一个int值时，将会**自动地拆箱**：

　　如果这时调用：

int n = list.get(i);

　　编译器会把这条语句自动变成：

**int n = list.get(i).intValue();**

#### 4.4.3 对象包装器对象的比较

使用“==”比较两个对象包装器对象时，检测的是对象是否指向同一个存储区域。因此，下面的比较通常不会成立：

Integer a = 1000；

Integer b = 1000；

if （a == b） ...;

因此，**对两个对象包装器的比较需要调用equals方法。**

#### 4.4.4 自动装箱的空指针异常

　例如：对象包装类对象可以是null，因此，在自动拆箱后，下面的代码将抛出空指针异常

Integer n = null;

System.out.print(2\*n);

#### 4.4.5 混合对象包装器在自动拆箱时自动装换类型

　例如：执行过程是，Integer类型的值n将自动拆箱，然后将int提升为double，最后再装箱成Double

Integer n = 1;

Double x = 2.9;

if (n == x)...;

### 4.5 参数数量可变的方法

在jdk 1.5 之后 , 方法的参数可以通过省略号...来作为一部分 , 即(String ... str) , 这种形式其实是语法糖 , 实际上相当于 (String[ ] str) .

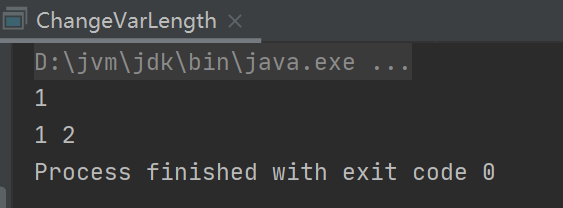
通过...这种形式的参数 , 表明这个方法可以接受任意数量的对象.

#### 4.5.1可变参数的调用：

**代码：**

public class ChangeVarLength {  
 public static void main(String[] args) {  
 *changePara*();  
 *changePara*(1);  
 *changePara*(1, 2);  
 }  
  
 public static void changePara(Object... obj) {  
 for (Object o : obj) {  
 System.*out*.println(o);  
 }  
 }  
}

结果：



#### 4.5.2 可变参数的使用原则

**1，一个方法只能有一个可变的参数,并且这个参数必须是该方法的最后一个参数**

正确形式 :

public static void test1(String ... str) { // TODO }

public static void test2(String str, String ... str1) { // TODO }

public static void test3(String str, String str1, String ... str2) { // TODO }

2，**如果重载了可变参数的方法, 优先调用匹配定长参数的方法,不定参数的那个重载方法是最后被选中的。**

3，**对于继承,子类将父类方法的参数列表数组形式改为可变参数数量属于重写,但是指向子类的父类调用该方法是形参列表由父类决定.**

**程序清单4.9：**

**要求：验证原则3**

代码：

父类：

package com.dong.day6.JC4;  
  
public class Base {  
 public void f(String[] str)  
 {  
 System.*out*.println("haha");  
 }  
}

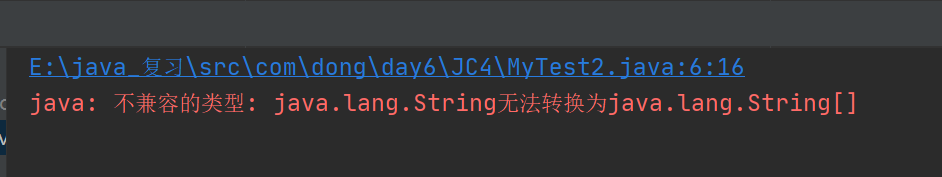
子类：

package com.dong.day6.JC4;  
  
public class Sub extends Base{  
 public void f(String...str)  
 {  
 System.*out*.println("hehe");  
 }  
}

测试类：

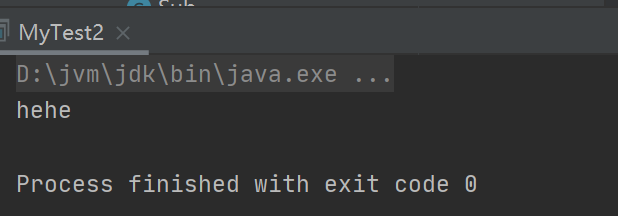
package com.dong.day6.JC4;  
  
public class MyTest2 {  
 public static void main(String[] args) {  
 Base base=new Sub();  
 base.f("Hello");  
 }  
}

结果：



**原因 : 因为父类参数列表是数组,因此传入字符串类型肯定会报错.**

**变式1：如果父类是变参，子类是数组，结果：**



调用了子类重写的方法。

**变式2：子类和父类都是变参**

代码：

父类：

package com.dong.day6.JC4;  
  
public class Base {  
 public void f(String... str)  
 {  
 System.*out*.println("haha");  
 }  
}

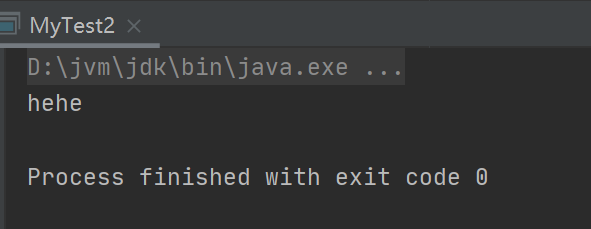
子类：

package com.dong.day6.JC4;  
  
public class Sub extends Base{  
 public void f(String...str)  
 {  
 System.*out*.println("hehe");  
 }  
}

测试类：

package com.dong.day6.JC4;  
  
public class MyTest2 {  
 public static void main(String[] args) {  
 Base base=new Sub();  
 base.f("Hello","nihao");  
 }  
}

结果：无论是传入数组还是字符串，都是调用子类重写的方法



### 4.6 枚举类

public enuni Size { SMALL , MEDIUM, LARGE, EXTRAJARGE };

这个声明定义是一个类

所有的枚举类型都是 Enum 类的子类。

继承的方法：

1）toString，name：这个方法能够返回枚举常量名。

2）values：它将返回一个包含全部枚举值的数组。代码：Week[] weeks = Week.values();

3）ordinal：返 冋 enum 声 明 中 枚 举 常 量 的 位 置， 位 置 从 0 开始计数。

4）Enum.valueOf：返回当前枚举类的name属性。

**细讲valueOf：**

**1，**该方法为静态方法，调用者为枚举类，不是枚举值；

2，该方法形参只要一个String类型的值即可，并且该String值为枚举值对应的名称，即toString()方法返回的值（比如枚举类为SPRING，对应的名称就是"SPRING"），所以肯定是需要双引号的；

3，该方法有返回值，且返回值是一个枚举值。

4.该方法输出和toString一样，toString改变返回值，该方法同时改变。

**注意**：1)枚举类的所有实例必须放在第一行显示，不需使用new，不需显示调用构造方法，每个变量都是public static final修饰的，最终以分号结束。2)枚举类的构造方法一定是private。

**代码实现：**

**package** meiju;

**public** **class** meiju {

**public** **static** **void** main(String[] args) {

//通过values()获取枚举数组

Week[] weeks = Week.*values*();

Week a=Week.***MONDAY***;

System.***out***.println(a.name());

//遍历Week枚举类

**for** (Week day : weeks) {

System.***out***.println("name:" + day.name() +

",desc:" + day.getDesc());

}

//不符合则抛出java.lang.IllegalArgumentException

System.***out***.println(Week.*valueOf*("MONDAY"));

//返回对应的name属性

System.***out***.println(Week.***FRIDAY***.toString());

//返回4，根据我们定义的次序，从0开始。如果在定义时调换FRIDAY

//的次序，返回的数字也会对应的变化

System.***out***.println(Week.***FRIDAY***.ordinal());

}

}

**enum** Week {

//本文的枚举类变量，枚举类实例，name属性指的就是MONDAY

//这类的变量

***MONDAY***(0,"星期一"), //monday是枚举类，括号中是枚举值

***TUESDAY***(1,"星期二"),

***WEDNESDAY***(2,"星期三"),

***THURSDAY***(3,"星期四"),

***FRIDAY***(4,"星期五"),

***SATURDAY***(5,"星期六"),

//最后一个类型必须要用分号结束

***SUNDAY***(6,"星期日");

**private** **int** num;

**private** String desc;

/\*\*

\* 构造方法必然是private修饰的

\* 就算不写，也是默认的

\*

\* **@param** num

\* **@param** desc

\*/

**private** Week(**int** num, String desc) {

**this**.num=num;

**this**.desc = desc;

}

**public** String getDesc() {

**return** desc;

}

**public** **int** getNum() {

**return** num;

}

/\*\*

\* 用switch重写toString方法，提高代码健壮性

\* **@return**

\*/

@Override

**public** String toString() {

//switch支持Enum类型

**switch** (**this**) {

**case** ***MONDAY***:

**return** "今天星期一";

**case** ***TUESDAY***:

**return** "今天星期二";

**case** ***WEDNESDAY***:

**return** "今天星期三";

**case** ***THURSDAY***:

**return** "今天星期四";

**case** ***FRIDAY***:

**return** "今天星期五";

**case** ***SATURDAY***:

**return** "今天星期六";

**case** ***SUNDAY***:

**return** "今天星期日";

**default**:

**return** "Unknow Day";

}

}

}

### 4.7 反射

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。要想解剖一个类,必须先要获取到该类的字节码文件对象。而解剖使用的就是Class类中的方法.所以先要获取到每一个字节码文件对应的Class类型的对象. Class对象的由来是将class文件读入内存，并为之创建一个Class对象。 **反射本质就是把java类中的各种成分映射成一个个的Java对象**

**字节码文件：**就是.class文件，.java文件编译后就会生成.class文件(即字节码文件)。

#### 4.7.1 class类

1)Class没有公共构造方法，class对象是在加载类时java虚拟机和以及通过调用类加载器中的classdefine方法自动构造的。

2) Class 类的实例(对象)表示正在运行的 Java 应用程序中的类和接口。也就是jvm中有N多的实例每个类都有该Class对象。

##### 4.7.1.1 获取class类的三种方式

假设已有Student类

**1.1 Object ——> getClass();**

**语法形式： Student.getClass();**

1.2**任何数据类型（包括基本数据类型）都有一个“静态”的class属性**1.3**通过Class类的静态方法：forName（String  className）(常用)**

**语法形式：Class.*forName*("Student");**

#### 4.7.2 反射获取构造方法并使用：Method类

注意：throws关键字对外声明该方法有可能发生异常

1.获取构造方法：

\* 1).批量的方法：

\* public Constructor[] getConstructors()：所有"公有的"构造方法

public Constructor[] getDeclaredConstructors()：获取所有的构造方法(包括私有、受保护、默认、公有)

\* 2).获取单个的方法，并调用：

\* public Constructor getConstructor(Class... parameterTypes):获取单个的"公有的"构造方法：

\* public Constructor getDeclaredConstructor(Class... parameterTypes):获取"某个构造方法"可以是私有的，或受保护、默认、公有；

调用构造方法：

Constructor-->newInstance(Object... initargs)，创建实例化对象

**package** fanshe.method;

**import** java.lang.reflect.\*;

**public** **class** MethodClass {

**public** **static** **void** main(String[] args) **throws** Exception {

//1.获取Class对象

Class stuClass = Class.*forName*("fanshe.method.Student");

//2.获取所有公有方法

System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取所有的”公有“方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

stuClass.getMethods();

Method[] methodArray = stuClass.getMethods();

**for**(Method m : methodArray){

System.***out***.println(m);

}

System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取所有的方法，包括私有的\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

methodArray = stuClass.getDeclaredMethods();

**for**(Method m : methodArray){

System.***out***.println(m);

}

System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取公有的show1()方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

Method m = stuClass.getMethod("show1", String.**class**);

System.***out***.println(m);

//实例化一个Student对象

Object obj = stuClass.getConstructor().newInstance();

m.invoke(obj, "刘德华"); //通过反射来调用方法

System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取私有的show4()方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

m = stuClass.getDeclaredMethod("show4", **int**.**class**);

System.***out***.println(m);

m.setAccessible(**true**);//解除私有限定

Object result = m.invoke(obj, 20);//需要两个参数，一个是要调用的对象（获取有反射），一个是实参

//System.out.println("返回值：" + result);

}

}

**class** Student {

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*成员方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

**public** Student(){

//System.out.println("调用了公有、无参构造方法执行了。。。");

}

**public** **void** show1(String s){

System.***out***.println("调用了：公有的，String参数的show1(): s = " + s);

}

**protected** **void** show2(){

System.***out***.println("调用了：受保护的，无参的show2()");

}

**void** show3(){

System.***out***.println("调用了：默认的，无参的show3()");

}

**private** String show4(**int** age){

System.***out***.println("调用了，私有的，并且有返回值的，int参数的show4(): age = " + age);

**return** "abcd";

}

}

**说明：**

*/\**

*\* 获取成员方法并调用：*

*\**

*\* 1.批量的：*

*\* public Method[] getMethods():获取所有"公有方法"；（包含了父类的方法也包含Object类）*

*\* public Method[] getDeclaredMethods():获取所有的成员方法，包括私有的(不包括继承的)*

*\* 2.获取单个的：*

*\* public Method getMethod(String name,Class<?>... parameterTypes):*

*\* 参数：*

*\* name : 方法名；*

*\* Class ... : 形参的Class类型对象*

*\* public Method getDeclaredMethod(String name,Class<?>... parameterTypes)*

*\**

*\* 调用方法：*

*\* Method --> public Object invoke(Object obj,Object... args):*

*\* 参数说明：*

*\* obj : 要调用方法的对象；*

*\* args:调用方式时所传递的实参；*

*):*

*\*/*

#### 4.7.3 反射获取构造方法并使用：field类

代码：

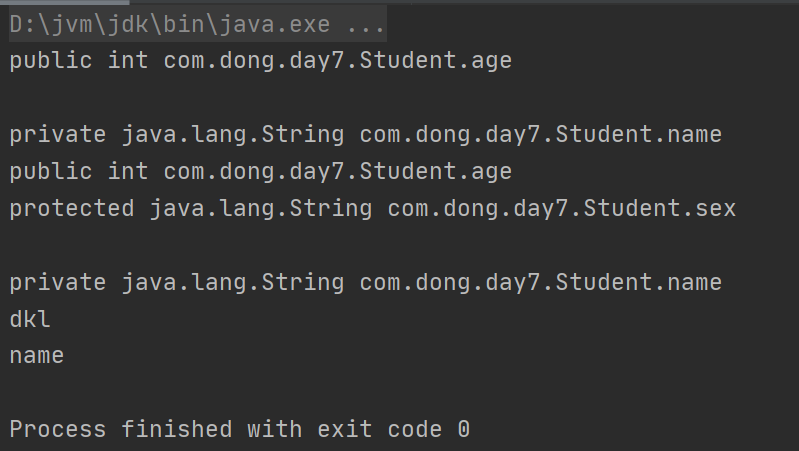
学生类：（被反射的类）

package com.dong.day7;  
  
public class Student {  
 private String name;  
 public int age;  
 protected String sex;  
  
 @Override  
 public String toString() {  
 return "Student{" +  
 "name='" + name + '\'' +  
 ", age=" + age +  
 ", sex='" + sex + '\'' +  
 '}';  
 }  
}

测试类：

package com.dong.day7;  
  
import java.lang.reflect.Field;  
  
public class MyTest {  
  
  
 public static void main(String[] args) throws ClassNotFoundException, NoSuchFieldException, IllegalAccessException, InstantiationException {  
 //获取class对象  
 Class class1=Class.*forName*("com.dong.day7.Student");  
 //获取所有公有属性  
 Field[] fields1=class1.getFields();  
 for(Field elem:fields1)  
 System.*out*.println(elem+" ");  
 System.*out*.println();  
  
 //获取所有属性（包括公有，私有，保护）  
 Field[] fields2=class1.getDeclaredFields();  
 for(Field elem:fields2)  
 System.*out*.println(elem+" ");  
 System.*out*.println();  
 //得到特定的属性  
 Field field1=class1.getDeclaredField("name");  
 System.*out*.println(field1);  
  
 //暴力访问，让某一属性可以被设置值和获取值  
 field1.setAccessible(true);  
 //对象实例化  
 Object obj=class1.newInstance();  
 //给类中的属性注入数据  
 field1.set(obj,"dkl");  
 //输出类特定属性的值  
 System.*out*.println("对象的属性值："+field1.get(obj));  
 //输出对象的名称  
 System.*out*.println("对象的名称："+field1.getName());  
 }  
}

结果：



## 五，接口，lambda表达式与内部类

### 5.1 接口

**接口概念**：在 Java 程序设计语言中， 接口不是类，而是对类的一组需求描述，这些类要遵从接口描述的统一格式进行定义。

1）接口通常以**interface**来声明。一个类通过继承接口的方式，从而来继承接口的抽象方法。

2）接口并不是类，编写接口的方式和类很相似，但是它们属于不同的概念。类描述对象的属性和方法。接口则包含类要实现的方法

3）接口无法被实例化，但是可以被实现。一个实现接口的类，必须实现接口内所描述的所有方法，否则就必须声明为抽象类。

4）函数式接口（接口中只有一个方法）

#### 5.1.1 接口与类的区别

1）接口不能用于实例化对象，但是接口可以定义引用：接口的引用指向的是实现了接口方法的类的实例化对象，定义的接口的引用可以调用实现接口的方法，实现类对象指向接口引用被称为接口的多态。

2）接口没有构造方法。

3）接口不能包含成员变量，除了 static 和 final 变量。

4）接口不是被类继承了，而是要被类实现

5）接口支持多继承。

6）接口中所有方法都是抽象方法

针对第三点发问：**为什么接口中的成员变量非得是public static final的呢？**

**答：**

首先明白一个原理，就是接口的存在意义。接口就是为了实现多继承的抽象类，是一种高度抽象的模板、标准或者说协议。规定了什么东西该是这样，如果你继承了我这接口，就必须这样。比如USB接口，就是小方口，两根电源线和两根数据线，不能多不能少。

**1.Public**

既然是公共的模板或者协议，那么如果定义成private就没有意义了，因为所有继承了你这接口的类都不能用，并且接口中的方法是不能够被具体实现的，因此，接口内部中也没有任何方法可使用。因此，为了让所有实现了该接口的类能够使用，就必须是public的。接口中定义的所有东西就应该是对所有用户开放的东西。

**2.final**

想想，如果不是final的，那么意味着每一个实现了该接口的子类都可以去修改这个变量。我们开头说了，接口就是标准规范，也改也只能是制定该接口的架构师来改，如果某类随便改的话，那么其他也继承了该接口的类就会受到影响。牵一发而动全身！！因此，既然是标准，那么就不能改，方便管理。

**3.static**

如果接口中的成员变量是非静态的，那么每一个实现了该接口的类都会有这么一个变量。那么，因为接口是多继承的，那么如果另一个接口也是有同样这样一个变量呢，那你用哪一个？所以，因为是标准，所以我规定从一开始，这个东西只能有一份，只能放在静态存储区，如果第二个接口也想同命名这么一个变量，那么存储时候就会报错，因为我静态存储区已经有一份了。你改名吧。

**归纳总结：**

* public是因为接口是标准，必须对外完全开放，自己藏着掖着没意义；
* static是因为要确保该变量只有一份，避免重名；
* final是因为接口的东西是大家共用的，不能随便修改，因此干脆不然你有修改 的权限！

#### 5.1.2 接口特性

接口中每一个方法也是隐式抽象的,接口中的方法会被隐式的指定为 **public abstract**

接口中可以含有变量，但是接口中的变量会被隐式的指定为 **public static final** 变量

接口中的方法是不能在接口中实现的，只能由实现接口的类来实现接口中的方法。

#### 5.1.3 接口与抽象类的区别

1）抽象类中的方法可以有方法体，就是能实现方法的具体功能，但是接口中的方法不行

2）抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是 **public static final** 类型的

3）一个类只能继承一个抽象类，而一个类却可以实现多个接口。

#### 5.1.4默认方法 ：default

Java8之前，接口中只能有抽象方法，但是现在已经可以实现方法了，叫做默认方法。

接口可由多个类实现，如果接口改变，类的方法也要改变，java8之前的接口操作就很麻烦，如果设计成默认方法就不用对类进行修改。

接口中可以写方法的实现，那么就会出现与父类之间进行冲突的问题：

**程序清单5.1：**

**要求：**

接口中的默认方法和父类的方法冲突验证。

**代码：**

**接口：**

package com.dong.day8;  
  
public interface MyInterface {  
 */\*\*  
 \* 默认方法  
 \** ***@return*** *\*/* default String getName () {  
 return "张三";  
 }  
  
 default String getAge () {  
 return "13";  
 }  
  
 */\*\*  
 \* 普通方法  
 \** ***@return*** *\*/* String getHome ();  
}

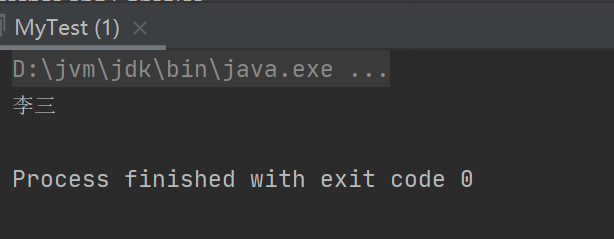
父类：

package com.dong.day8;  
  
public class MyClass {  
 public String getName()  
 {  
 return "李三";  
 }  
}

测试类：

package com.dong.day8;  
  
public class MyTest extends MyClass implements MyInterface{  
 public static void main(String[] args) {  
 MyTest myTest=new MyTest();  
 System.*out*.println(myTest.getName());  
 }  
  
 @Override  
 public String getAge() {  
 return null;  
 }  
  
 @Override  
 public String getHome() {  
 return null;  
 }  
}

结果：



**分析:**

这里有一个类优先的原则：

1）如果是接口和父类发生冲突以父类的方法优先。

2）如果是接口和接口发生冲突，就必须在实现类中覆写此方法：有两种处理方法：1.自定义2.选择其中的一种方法：

程序清单5.2：

要求：实现分析中第二种情况

代码：

public interface MyFun {

default String getName () {

return "王五";

}

}

public class My\_JAVA8\_Test implements MyInterface, MyFun{

public static void main(String[] args) {

My\_JAVA8\_Test test = new My\_JAVA8\_Test();

System.out.println(test.getName());

}

@Override

public String getHome() {

return null;

}

/\*\*

\* 此时，类会让我们自己去重写，也可以自己选择使用上面2个接口中的方法

\*/

@Override

public String getName() {

return MyFun.super.getName();

}

}

#### 5.1.5 static：静态修饰

特点：

1）静态方法不能被子接口继承

2）静态方法不能被实现该接口的类继承

3）调用形式：接口名.静态方法名()

**程序清单5.2：**

**要求：验证特点1和特点2**

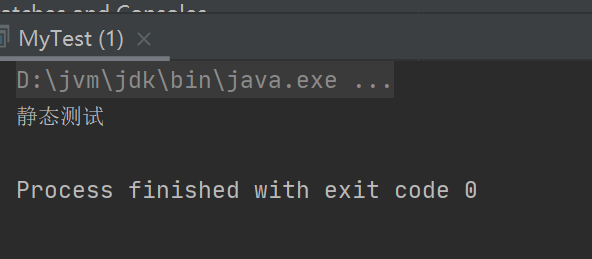
**接口**：

package com.dong.day8;  
  
public interface MyFun {  
 default String getName()  
 { return "王五";}  
  
 static void test()  
 {  
 System.*out*.println("静态测试");  
 }  
}

测试类：

package com.dong.day8;  
  
public class MyTest implements MyFun{  
 public static void main(String[] args) {  
 MyFun.*test*();  
 }  
}

结果：



### 5.2 接口实例

#### 5.2.1接口与回调

一个接口被多个类实现，为知晓是哪个类来实现接口的功能，于是出现**监控类**（调动实现接口的类的方法）。

**程序清单5.3：**

**要求：实现监控类的功能**

**代码：**

接口：

package com.dong.day8.JKL;  
  
public interface BJK {  
 public void ShiWu();  
}

实现类1：

package com.dong.day8.JKL;  
  
public class Method1 implements BJK{  
  
 @Override  
 public void ShiWu() {  
 System.*out*.println("执行方法1");  
 }  
}

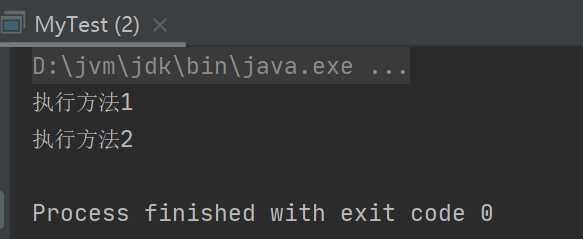
实现类2：

package com.dong.day8.JKL;  
  
public class Method2 implements BJK{  
  
 @Override  
 public void ShiWu() {  
 System.*out*.println("执行方法2");  
 }  
}

测试类：

package com.dong.day8.JKL;  
  
public class MyTest {  
 public static void main(String[] args) {  
 //构造监视器对象  
 Mymoniter mymoniter=new Mymoniter();  
 //构造实现类的对象  
 Method1 method1=new Method1();  
 Method2 method2=new Method2();  
 //方法一实现接口  
 mymoniter.setBjk(method1);  
 mymoniter.Show();  
 //方法二实现接口  
 mymoniter.setBjk(method2);  
 mymoniter.Show();  
  
 }  
}

结果：



**分析：**

MyMoniter是一个监控者，当有事件发生时，通知监控类调用具体方法解决事件。

#### 5.2.2 Comparator 接口

Comparator接口是一个用于比较的接口。该接口注重的是比较容器

Comparator接口中必须要实现的compare(T o1,T o2)，有两个参数。

**程序清单5.4：**

class StudentComparator implements Comparator<Student>{

@Override

public int compare(Student o1, Student o2) {

// TODO Auto-generated method stub

if(o1.getScore()>o2.getScore())

return -1;

else if(o1.getScore()<o2.getScore())

return 1;

else{

if(o1.getAge()>o2.getAge())

return 1;

else if(o1.getAge()<o2.getAge())

return -1;

else

return 0;

}

}

}

public class ComparableDemo02 {

public static void main(String[] args) {

// TODO Auto-generated method stub

Student stu[]={new Student("zhangsan",20,90.0f),

new Student("lisi",22,90.0f),

new Student("wangwu",20,99.0f),

new Student("sunliu",22,100.0f)};

java.util.Arrays.sort(stu,new StudentComparator());

for(Student s:stu)

{

System.out.println(s);

}

}

调用此接口排序需要两个参数：1）对象数组，2）重写的比较方法的类对象。

### 5.3 lambda表达式

#### 5.3.1函数式接口

函数式接口是只包含一个方法的接口。比如Java标准库中的java.lang.Runnable和 java.util.Comparator都是典型的函数式接口。  
java 8提供 @FunctionalInterface作为注解,这个注解是非必须的，只要接口符合函数式接口的标准（即只包含一个方法的接口），虚拟机会自动判断， 但 最好在接口上使用注解@FunctionalInterface进行声明，以免团队的其他人员错误地往接口中添加新的方法。

#### 5.3.2 lambda表达式概述

Lambda 表达式，也可称为闭包，它是推动 Java 8 发布的最重要新特性。

**Lambda 允许把函数作为一个方法的参数**。

并不是所有接口都可以使用Lambda表达式，只有函数式接口可以。函数式接口的定义，其只能有一个抽象方法，否则就不是函数时接口，就无法用Lambda表达式。

**为什么引入lambda表达式？**

Lambda表达式是一个匿名函数，Lambda是一段可以传递的代码（能够做到将代码像数据一样进行传递）。使用Lambda表达式能够写出更加简洁、灵活的代码。并且，使用Lambda表达式能够使Java的语言表达能力得到提升。

**语法格式：**

它由三个部分组成：第一部分为一个括号内用逗号分隔的形参，参数即函数式接口里面方法的参数；第二部分为一个箭头符号：->；第三部分为方法体，可以是表达式和代码块。

程序清单5.5：

要求：lambda表达式的使用

代码：

接口1：

package com.dong.day8.lambda;  
  
public interface Way1 {  
 public int f(int a,int b);  
}

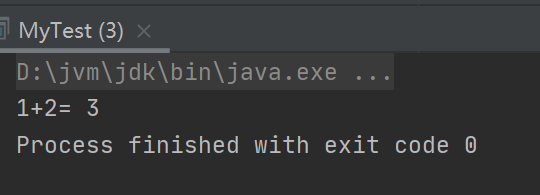
接口2：

package com.dong.day8.lambda;  
  
public interface Way2 {  
 public void print(String name);  
}

测试类：

package com.dong.day8.lambda;  
  
import com.dong.day8.JKL.Mymoniter;  
  
public class MyTest {  
 public static void main(String[] args) {  
 //加法  
 Way1 add=(int a,int b)->a+b;  
 //减法  
 Way1 sub=(int a,int b)->a-b;  
 //输出  
 Way2 output=info->{System.*out*.print(info);};  
 output.print("1+2= "+ MyTest.*operator*(1,2,add));  
  
 }  
 private static int operator(int a,int b,Way1 way1)  
 {  
 return way1.f(a,b);  
 }  
}

结果：



#### 5.3.3 lambda表达式引用方法

有时候我们不是必须要自己重写某个匿名内部类的方法，我们可以可以利用 lambda表达式的接口快速指向一个已经被实现的方法。

**语法**

​ 方法归属者::方法名 静态方法的归属者为类名，普通方法归属者为对象

**程序清单5.5**

**要求：用lambda表达式引用静态方法和非静态方法**

代码：

接口：

package com.dong.day8.lambda;  
  
public interface Way1 {  
 public int f(int a,int b);  
}

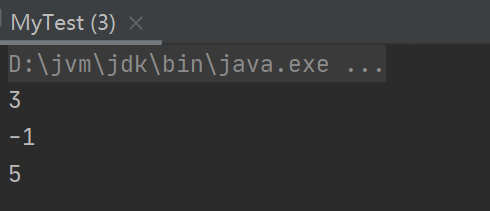
实现类：

package com.dong.day8.lambda;  
  
public class Method1 implements Way1{  
 @Override  
 public int f(int a, int b) {  
 return a-b;  
 }  
}

测试类：

package com.dong.day8.lambda;  
  
import com.dong.day8.JKL.Mymoniter;  
  
public class MyTest {  
 public static void main(String[] args) {  
 Way1 test = (a, b) -> *add*(a, b);  
 System.*out*.println(test.f(1, 2));  
  
 //非静态方法是对象名  
 Method1 method1 = new Method1();  
 Way1 test2 = method1::f;  
 System.*out*.println(test2.f(2, 3));  
 //引用静态方法  
 Way1 test3 = MyTest::*add*;  
 System.*out*.println(test3.f(2, 3));  
 }  
 public static int add(int a,int b)  
 {  
 return a+b;  
 }  
}

结果：



### 5.4 内部类

#### 5.4.1 成员内部类

定义：位于另一个类的内部。

**代码形式：**

**package** shuru;

**public** **class** ShuRu

{

**public** **static** **void** main(String[] args)

{

}

}

**class** Circle {

**private** **double** radius = 0;

**public** **static** **int** *count* =1;

**public** Circle(**double** radius) {

**this**.radius = radius;

}

**class** Draw { //内部类

**public** **void** drawSahpe() {

System.***out***.println(radius); //外部类的private成员

System.***out***.println(*count*); //外部类的静态成员

}

}

}

注意：

1）当成员内部类拥有和外部类同名的成员变量或者方法时，会发生隐藏现象，即默认情况下访问的是成员内部类的成员。如果要访问外部类的同名成员，需要以下面的形式进行访问：外部类.this.成员变量调用方法同理。

2）虽然成员内部类可以无条件地访问外部类的成员，而外部类想访问成员内部类的成员却不是这么随心所欲了。在外部类中如果要访问成员内部类的成员，必须先创建一个成员内部类的对象，再通过指向这个对象的引用来访问

**代码：实现外部类访问内部类：**

**class** Circle {

**private** **double** radius = 0;

**public** **static** **int** *count* =1;

**public** Circle(**double** radius) {

**this**.radius = radius;

getDrawInstance().drawSahpe();//创建内部类对象，然后访问

}

**private** Draw getDrawInstance()

{

**return** **new** Draw();

}

**class** Draw { //内部类

**public** **void** drawSahpe() {

System.***out***.println(radius); //外部类的private成员

System.***out***.println(*count*); //外部类的静态成员

}

}

}

注意：成员内部类是依附外部类而存在的，也就是说，如果要创建成员内部类的对象，前提是必须存在一个外部类的对象。

#### 5.4.2 隐匿内部类

##### 5.4.2.1前言：

匿名内部类在我们JAVA程序员的日常工作中经常要用到，但是很多时候也只是照本宣科地用，虽然也在用，但往往忽略了以下几点：为什么能这么用？匿名内部类的语法是怎样的？有哪些限制？

##### 5.4.2.2匿名内部类

匿名内部类可以使你的代码更加简洁，你可以在定义一个类的同时对其进行实例化。它与局部类很相似，不同的是它没有类名，如果某个局部类你只需要用一次，那么你就可以使用匿名内部类

本节包括以下几个方面：

1. 定义匿名内部类
2. 匿名内部类的语法
3. 访问作用域的局部变量、定义和访问匿名内部类成员
4. 匿名内部类实例

**1.定义匿名内部类**

**程序清单：**

**要求：定义匿名内部类**

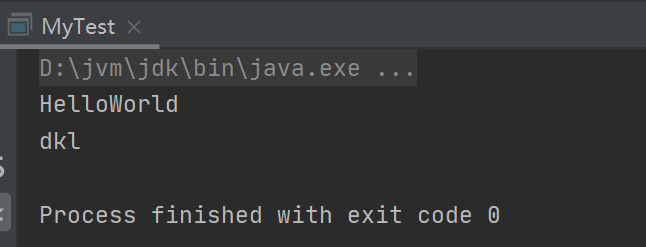
**内部类：**

package com.dong.day8.NMNBL;  
  
public class HelloWorldAnonymousClasses {  
  
 */\*\*  
 \* 包含两个方法的HelloWorld接口  
 \*/* interface HelloWorld{  
 public void greet();  
 public void greetName(String name);  
 }  
  
 //匿名内部类  
 public void SayHello()  
 {  
 HelloWorld helloWorld=new HelloWorld() {  
  
 @Override  
 public void greet() {  
 System.*out*.println("HelloWorld");  
 }  
  
 @Override  
 public void greetName(String name) {  
 System.*out*.println(name);  
 }  
 };  
 helloWorld.greet();  
 helloWorld.greetName("dkl");  
 }  
}

测试类：

package com.dong.day8.NMNBL;  
  
public class MyTest {  
 public static void main(String[] args) {  
 HelloWorldAnonymousClasses helloWorldAnonymousClasses=new HelloWorldAnonymousClasses();  
 helloWorldAnonymousClasses.SayHello();  
 }  
}

结果：



**注意**：匿名类是一个表达式，因此在定义的最后用分号";"结束。

**2.定义匿名子类**

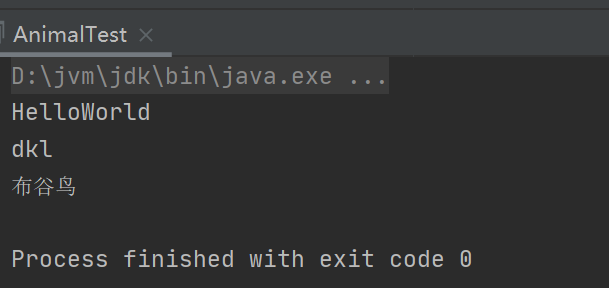
程序清单：5.7

要求：构建匿名子类，继承父类

代码：

package com.dong.day8.NMNBL;  
  
public class AnimalTest {  
 private final String name="dkl";  
 public void f()  
 {  
 System.*out*.println("HelloWorld");  
 }  
 //成员内部类  
 class Animal{  
 private String name1;  
  
 public Animal(String name1) {  
 this.name1 = name1;  
 }  
  
 public void print()  
 {  
 System.*out*.println(bird.name1);  
 }  
 }  
 //匿名子类  
 Animal bird=new Animal("布谷鸟"){  
 public void print() //重写父类方法  
 {  
 f();  
 //访问外部final修饰的成员变量和父类的变量  
 System.*out*.println(name);  
 super.print();  
 }  
 };  
 public void show()  
 {  
 bird.print();  
 }  
 public static void main(String[] args) {  
 AnimalTest animalTest=new AnimalTest() ;  
 animalTest.show();  
 }  
}

结果：



分析：

1. 操作符：new；
2. 一个要实现的接口或要继承的类，案例一中的匿名类实现了HellowWorld接口，案例二中的匿名内部类继承了Animal父类；
3. 一对括号，如果是匿名子类，与实例化普通类的语法类似，如果有构造参数，要带上构造参数；如果是实现一个接口，只需要一对空括号即可；
4. 一段被"{}"括起来类声明主体；
5. 末尾的";"号（因为匿名类的声明是一个表达式，是语句的一部分，因此要以分号结尾）

**3. 访问作用域内的局部变量、定义和访问匿名内部类成员**

(1)、匿名内部类可以访问外部内的所有成员；

(2)、匿名内部类不能访问外部类未加final修饰的变量（注意：JDK1.8即使没有用final修饰也可以访问）；

(3)、属性屏蔽，与内嵌类相同，匿名内部类定义的类型（如变量）会屏蔽其作用域范围内的其他同名类型（变量）：

(4)、匿名内部类中不能定义静态属性、方法；

(5)、匿名内部类可以有常量属性（final修饰的属性）；

(6)、匿名内部内中可以定义属性，如上面代码中的代码:private int x = 1;

(7)、匿名内部内中可以可以有额外的方法（父接口、类中没有的方法）;

(8)、匿名内部内中可以定义内部类；

(9)、匿名内部内中可以对其他类进行实例化。

#### 5.4.3 局部内部类

局部内部类是定义在一个方法或者一个作用域里面的类，它和成员内部类的区别在于局部内部类的访问仅限于方法内或者该作用域内.

**注意**：局部内部类就像是方法里面的一个局部变量一样，是不能有 public、protected、private 以及 static 修饰符的。

**程序清单5.8：**

**要求：**

**实现局部内部类和讨论相应的问题，详细请看代码**

代码：

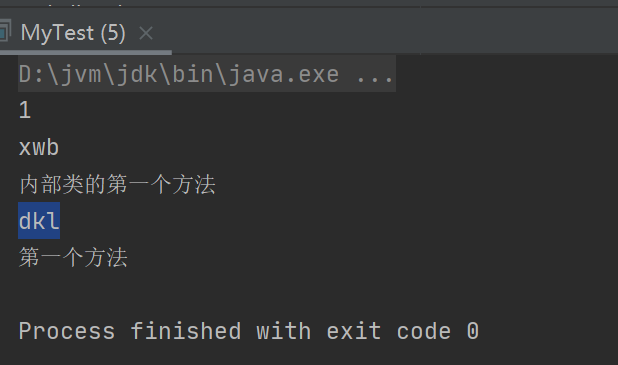
内部类：

package com.dong.day8.JBNBL;  
  
public class Student {  
 private String name;  
 private int age;  
  
 public Student(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
  
 public void method1()  
 {  
 System.*out*.println("第一个方法");  
 }  
  
 //局部内部类嵌入方法中  
 //局部内部类就类似方法中的局部变量  
 public void method2()  
 {  
 final int a=2;  
  
 class NBL{  
 private int a=1;  
 private String name="xwb";  
 public void method2()  
 {  
 //访问外部方法的变量需要在外部方法的变量前加上final  
 //满足上一个条件后，如果内部类中的变量和外部方法的变量重名，就输出内部类的变量  
 System.*out*.println(a);  
 //输出外部类的属性,(局部内部类可以直接调用外部类的属性)  
 //如果内部类的属性名和外部类的属性名同名，则输出内部类的属性值  
 System.*out*.println(name);  
 method1();  
 //如果同名后，想输出外部类的属性和方法执行下列代码  
 System.*out*.println(Student.this.name);  
 Student.this.method1();  
 }  
 public void method1()  
 {  
 System.*out*.println("内部类的第一个方法");  
 }  
 }  
 NBL nbl=new NBL();  
 nbl.method2();  
 }  
}

测试类：

package com.dong.day8.JBNBL;  
  
public class MyTest {  
 public static void main(String[] args) {  
 Student student=new Student("dkl",21);  
 student.method2();  
 }  
}

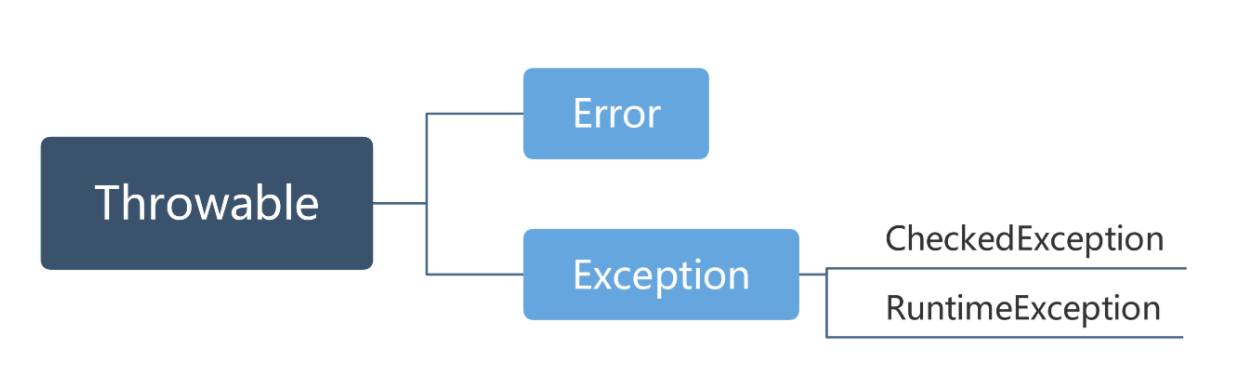
结果：



## 六，异常，日志，断言

### 6.1 异常概述

#### 6.1.1 异常分类



java.lang.Throwable是所有异常的根

java.lang.Error是错误信息

java.lang.Exception是异常信息

**Exception：**一般分为Checked异常和Runtime异常，所有RuntimeException类及其子类的实例被称为Runtime异常，不属于该范畴的异常则被称为CheckedException。

Checked异常：只有java语言提供了Checked异常，Java认为Checked异常都是可以被处理的异常，所以Java程序必须显示处理Checked异常。

对Checked异常处理方法有两种

1 当前方法知道如何处理该异常，则用try...catch块来处理该异常。

1. *// 捕获并处理这个异常*
2. public void testMethod1\_01()
3. {
4. try
5. {
6. method1();
7. }
8. catch (Exception e)
9. {
10. e.printStackTrace();
11. }
12. }

2 当前方法不知道如何处理，则在定义该方法是声明抛出该异常。

1. public void testMethod1\_03() throws Exception
2. {
3. throw new Exception(); //自定义抛出错误
4. }

**runtimeException**:

Runtime如除数是0和数组下标越界等

子类：

Java.lang.ArithmeticException  
Java.lang.ArrayStoreExcetpion  
Java.lang.ClassCastException  
Java.lang.IndexOutOfBoundsException  
Java.lang.NullPointerException

public void method3() throws RuntimeException

{

System.out.println("我是抛出运行时异常的方法");

}

**Error:**

1)当程序发生不可控的错误时，通常做法是通知用户并中止程序的执行。与异常不同的是Error及其子类的对象不应被抛出。

2)Error是throwable的子类，代表编译时间和系统错误，用于指示合理的应用程序不应该试图捕获的严重问题。

3)Error由Java虚拟机生成并抛出，包括动态链接失败，虚拟机错误等。程序对其不做处理。

#### 6.1.2抛出异常（throw）

对于已经存在的异常类，要将其抛出需要

1）找到一个合适的异常类

2）创建他的一个对象

3）抛出（throw）

#### 6.1.3 throws

方法应该在其首部用 throws 关键字声明所有可能抛出的受查异常，作用在于，告知这个方法的调用者，并交由其处理，如果这个方法的调用者不对它进行处理，将无法通过编译。而非受查异常要么不可控制（Error），要么就应该避免发生（RuntimeException）。

**注意**，throws 只是说明了发生某种异常的可能，只有在方法执行的过程中确实发生异常时，才会把相应的异常对象传递给这个方法的调用者。

#### 6.1.4 创建异常类

在程序中，可能会遇到任何标准异常类都没有能够充分地描述清楚的问题, 我们需要做的只是定义一个派生于 Exception 的类，或者派生于 Exception 子类的类, 定义的类应该包含两个构造器， **一个是默认的构造器**；另一个是带有详细描述信息的构造器.

### 6. 2 捕获异常

要想捕获一个异常，必须设置 try/catch语句块.

如果在 try语句块中的任何代码抛出了一个在 catch 子句中说明的异常类，那么:

**1)**程序将跳过 try语句块的其余代码。

**2)**程序将执行 catch 子句中的处理器代码, 如果方法中的任何代码拋出了一个在 catch 子句中没有声明的异常类型，那么这个方法 就会立刻退出

**注意**：如果调用了一个抛出受查异常的方法，那么就只能有两个选择：要么继续传递（throws），要么就地捕获并处理（try / catch）。哪种选择更好呢？一般来说，对那些知道如何处理的异常应该捕获和处理，而对不知道（可能了解的信息还不足）如何处理的异常应该继续传递。当然了，这是一个灵活的选择，重要的是对当前编写的方法功能有清晰的定位。

#### 7.2.2 捕获多个异常

在一个 try 语句块中可以捕获多个异常类型，并对不同类型的异常做出不同的处理.

可以按照下列方式为每个异常类型使用一个单独的 catch 子句

Try{

…………

}catch（异常类型）

{

}catch（异常类型）

{

}…………

在 Java SE 7中，同一个 catch 子句中可以捕获多个异常类型，前提：只有当捕获的异常类型彼此之间不存在子类关系时才需要这个特性。

捕获多个异常时，异常变量隐含为 final 变量。例如，不能在以下子句体中为 e 赋不同的值：catch (FileNotFoundException | UnknownHostException e) { . . . }

#### 6.2.3 再次抛出异常与异常链

**目的**：在 catch 子句中再次 throw 一个异常，目的在于改变异常的类型，提高灵活性

try {

// access the database

} catch (SQLException e) {

throw new ServletException("database error: " + e.getMessage());

}

还可以使用包装技术，先创建一个新的异常对象，再将原始异常设置为新异常的原因：

try {

// ...

} catch (SQLException e) {

Throwable se = new ServletException("data error");

se.initCause(e);

}

这样，当捕获到异常时，就可以通过下面的语句重新得到之前的异常：

Throwable previousException = se.getCause();

#### 6.2.4 finally

在 try 子句中发生异常后，剩余的代码就不会执行，这就给资源回收带来了问题，比如在读取一个本地文件时出现 IOException ，方法终止，但此时输出流还未被关闭。为了处理资源回收的问题，可以有两个选择：

* 使用 finally 子句，把释放资源的代码放在里面
* 使用带资源的 try 语句

不管是否有异常被捕获，finally 子句中的代码一定会被执行：

InputStream in = new FileInputStream(...);

try {

// might throw exceptions

} catch (IOException e) {

// 'catch' block is not neccessary

} finally {

in.close();

// other things that must be done

}

注意：（无法理解）

* 当 finally 块包含 return 语句时，会覆盖原本的值
* 当 finally 块中也抛出异常时，会覆盖原始的异常（如果有）

**解耦合**：

为了提高代码的清晰度，可以这么做：

InputStream in = ...;

try { // --> 确保异常被处理

try { // --> 确保资源被释放

...

} finally {

in.close();

}

} catch (IOException e) {

...

}

#### 6.2.5 try-with-resource

如果 try 块中用到的资源属于一个实现了 AutoCloseable 接口的类，这个接口有一个 close 方法，那么就可以使用带有资源的 try 语句：

try (Scanner in = new Scanner(new FileInputSream("...")),"UTF-8") {

while (in.hasNext()) {

// ...

}

}

这个 try 块退出时（不管有没有异常出现），都会自动调用 in.close()，而不需要像之前一样放在 finally 块中。 还可以指定多个资源：

try (Scanner in = new Scanner(...);

PrintWriter out = new PrintWriter(...)) {

// ...

}

使用带资源的 try 语句还有一个好处：如果 try 块和 close 方法都抛出了异常，那么 close 抛出的异常会被抑制，并自动通过 addSuppressed 方法增加到原来的异常，之后可以调用 getSuppressed 方法查看它

### 6.3断言和日志

#### 6.3.1断言的概念

断言机制允许在测试期间向代码中插入一些检查语句。当代码发布时，这些插入的检测语句将会被自动地移走。Java语言引入了关键字assert。assert有两种形式：

1）assert 条件;

和

2）assert 条件 : 表达式;

这两种形式都会对条件进行检测，如果结果为false，则抛出一个AssertionError异常。

例如：

计算：double y = Math.sqrt(x);

我们知道x不能为负数，要想使用断言，只需要简单地使用assert语句，进行测试即可：

assert x>=0;

or

assert x>=0 : x;

#### 6.3.2启用和禁用断言

在默认情况下，断言被禁用。可以在运行程序时用-enableassertions或-ea选项启用：

java -enableassertions MyApp。

可以在某个类或整个包中使用断言：

**代码**：

java -ea:MyClass //断言某个类

java -ea:com.mycompany.mylib...MyApp //断言某个包

//系统类不是由类加载器加载，所以不能用-ea 和-da,而是使用下面开关启用断言

-enablesystemassertions/-esa

**也可以使用选项-disableassertions或-da禁用某个特定类和包的断言。**

#### 6.3.3使用断言完成参数检查（不太理解）

在Java语言中，给出了3种处理系统错误的机制：

* 抛出一个异常
* 日志
* 使用断言

什么时候应该选择使用断言呢？

* 断言失败是致命的、不可恢复的错误
* 断言检查只用于开发和测试阶段

 注意：①不要在public方法里面检查参数是不是为null之类的操作

②不要用assert来检查方法操作的返回值来判断方法操作的结果

#### 6.3.4日志 Logger类

①可以很容易的取消全部日志记录，或者仅仅取消某个级别的日志，而且打开和关闭这个操作也容易

②可以很简单的禁止日志记录的输出，因此，将这些日志代码留在程序中的开销很小

③日志可以被定向到不同的处理器，例如用于在控制台显示，用于存储在文件中

④日志记录器和处理器都可以对记录进行过滤，过滤器也可以根据实际制定过滤标准，丢弃那些无用的记录项

⑤日志记录可以采用不同的方式格式化，例如：纯文本或XML

⑥应用程序可以使用多个日志记录器，它们使用类似包名的这种具有层次结构的名字

⑦在默认情况下，日志系统的配置由配置文件控制

##### 6.3.4.1基本日志

要生成简单的日志记录，可以使用全局日志记录器（global logger）并调用其info方法：

Logger.getGlobal().info("File->Open menu item selected");

也可以在适当地方调用关闭全局记录器的方法：

Logger.getGlobal.setLevel(level.OFF);

#### 6.3.4.2高级日志

1）在一个专业的应用程序中，不要将所有的日志都记录到一个全局日志记录器中，而是可以自定义日志记录器。可以调用getLogger方法创建或获取记录器：

private static final Logger myLogger = Logger.getLogger("com.company.myapp");

2）提示：未被任何变量引用的日志记录器可能会被垃圾回收。为了防止这种情况发生，要像上面的例子中一样，用一个**静态变量**存储日志记录器的一个引用。

3）记录日志的常见用途是记录那些不可预料的异常。可以使用下面两个方法提供日志记录 中包含的异常描述内容。

if (… ）

{

IOException exception = new IOException("...")； 1ogger.throwing("com•mycompany.mylib.Reader", "read", exception) ;

throw exception;

}

#### 6.3.4.3过滤器

在默认情况下， 过滤器根据日志记录的级别进行过滤。每个日志记录器和处理器都可以 有一个可选的过滤器来完成附加的过滤。另外，可以通过实现 niter 接口并定义下列方法来 自定义过滤器。

要想将一个过滤器安装到一个日志记录器或处理器中，只需要调用 setFilter 方法就可以 了。 注意，同一时刻最多只能有一个过滤器。

#### 6.3.4.4格式化器

ConsoleHandler 类和 FileHandler 类可以生成文本和 XML 格式的日志记录。

可以调用 setFormatter 方法将格式化器安装到处理器中。

#### 6.3.4.3日志配置说明

①为一个简单的应用程序，选择一个日志记录器，并把日志记录器命名为与主应用程序包一样的名字。

②默认的日志配置将级别等于或高于INFO级别的所有消息记录到控制台上。

③所有级别为INFO、WARNING、SEVERE的消息都将显示到控制台上。因此，最好只将对程序用户有意义的消息设置为这几个级别。

## 七，注解

### 7.1，什么是java注解？

（1）java注解（Annotation）又称java标注。是jdk5.0引入的注释机制。

（2）**注解（Annotation）是一种应用于类、方法、参数、变量、构造器及包声明中的特殊修饰符。它是一种 由JSR-175标准选择用来描述元数据的一种工具。**

（3）**通俗的讲就是一个辅助功能，可以传递数据，可以给编译器制定一个标准，要是你不按这个标准，就会编 译错误。**

### 7.2，常用的注解

Java 定义了一套注解，共有 7 个，3 个在 java.lang 中，剩下 4 个在 java.lang.annotation 中。

#### 2.1 作用在代码上的注释

##### 2.1.1@Override

检查该方法是否是重写方法。如果被他标注的方法不是重写其父类上的方法，就会报错。

##### 2.1.2@Deprecated

标记过时方法。如果使用该方法，会报编译警告。

##### 2.1.3@SuppressWarnings

指示编译器去忽略注解中声明的警告。

#### 2.2 作用于注解上的注释（元注解）

##### 2.2.1 @Retention

标识这个注解怎么保存，是只在代码中，还是编入class文件中，或者是在运行时可以通过反射访问。

Retention注解有一个属性value，是RetentionPolicy类型的，Enum RetentionPolicy是一个枚举类型， 这个枚举决定了Retention注解应该如何去保持，也可理解为Rentention 搭配 RententionPolicy使用。RetentionPolicy有3个值：CLASS RUNTIME SOURCE

(1)RetentionPolicy.CLASS：注解被保留到class文件，但jvm加载class文件时候被遗弃，这是默认的生命周期；

(2)RetentionPolicy.RUNTIME：注解不仅被保存到class文件中，jvm加载class文件之后，仍然存在；

(3)RetentionPolicy.SOURCE：注解只保留在源文件，当Java文件编译成class文件的时候，注解被遗弃；

###### 那怎么来选择合适的注解生命周期呢？

首先要明确生命周期长度 SOURCE < CLASS < RUNTIME ，所以前者能作用的地方后者一定也能作用。 一般如果需要在运行时去动态获取注解信息，那只能用 RUNTIME 注解，比如@Deprecated使用RUNTIME注解 如果要在编译时进行一些预处理操作，比如生成一些辅助代码（如 ButterKnife），就用 CLASS注解； 如果只是做一些检查性的操作，比如 @Override 和 @SuppressWarnings，使用SOURCE 注解。

##### 2.2.2@Target

标记这个注解应该是哪种 Java 成员。

###### 2.2.2.1@Target的属性：

**@Target(ElementType.TYPE)——接口、类、枚举、注解 @Target(ElementType.FIELD)——字段、枚举的常量 @Target(ElementType.METHOD)——方法 @Target(ElementType.PARAMETER)——方法参数 @Target(ElementType.CONSTRUCTOR) ——构造函数 @Target(ElementType.LOCAL\_VARIABLE)——局部变量 @Target(ElementType.ANNOTATION\_TYPE)——注解 @Target(ElementType.PACKAGE)——包**

##### 2.2.3@Documented（了解）

标记这些注解是否包含在用户文档中。

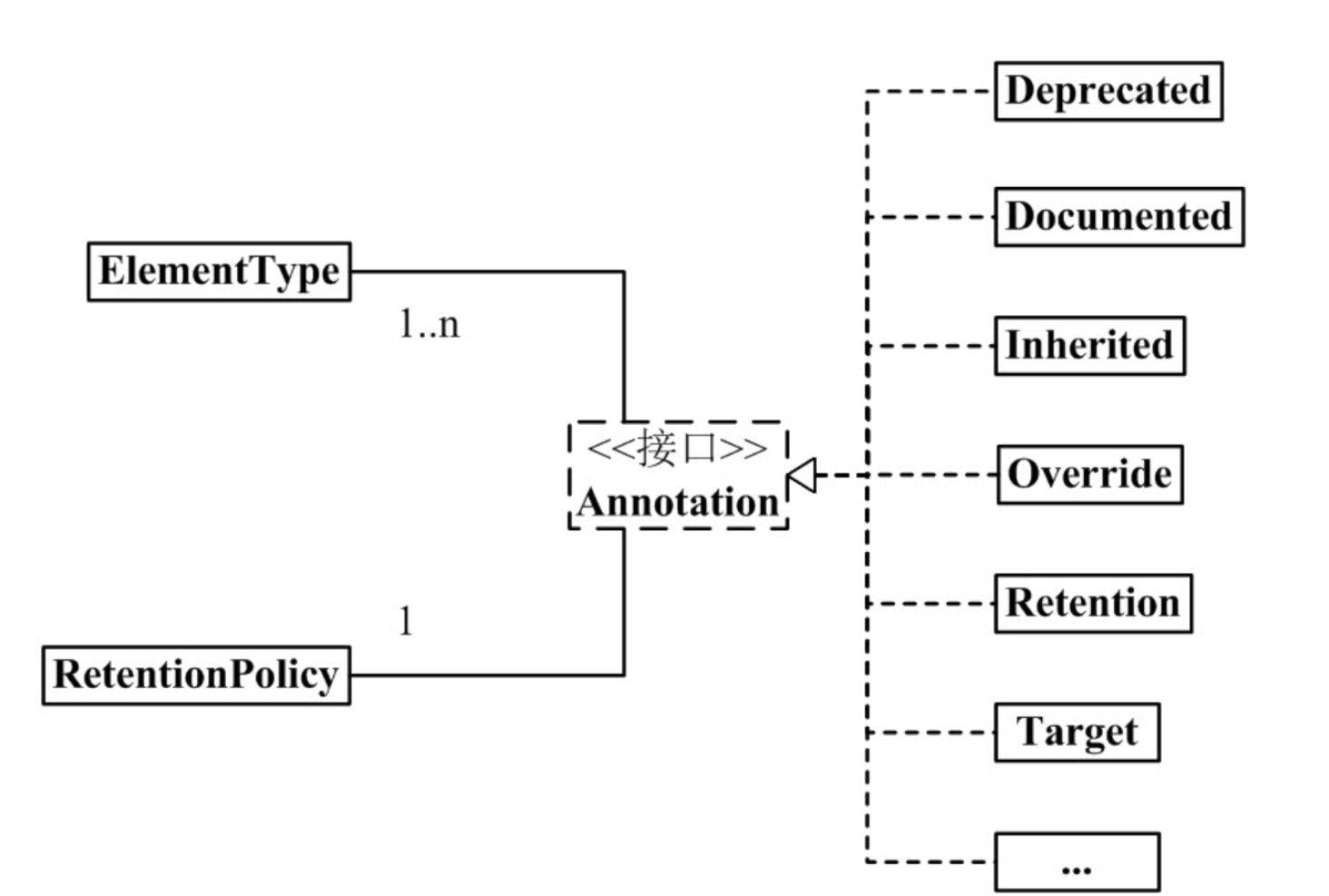
使用@Documented标注了，在生成javadoc的时候就会把@Documented注解给显示出来。

##### 2.2.4@Inherited

**标记这个注解是继承于哪个注解类。**

通俗理解（当@InheritedAnno注解加在某个类A上时，假如类B继承了A，则B也会带上该注解。）

### 7.3，annotation架构



（1）：每1个Annotation对象，都会有唯一的RetentionPolicy属性。

（2）：对于每 1 个 Annotation 对象，可以有若干个 ElementType 属性。

（3）：**Annotation 有许多实现类**：**Deprecated, Documented, Inherited, Override 等等**

### 7.4，annotation的组成部分

java Annotation 的组成中，有 3 个非常重要的主干类。

#### 4.1 主干类

（1）annotation.java

package java.lang.annotation;  
public interface Annotation {  
​  
   boolean equals(Object obj);  
​  
   int hashCode();  
​  
   String toString();  
​  
   Class<? extends Annotation> annotationType();  
}

（2）ElementType.java

package java.lang.annotation;  
​  
public enum ElementType {  
  TYPE,               /\* 类、接口（包括注释类型）或枚举声明 \*/  
​  
  FIELD,             /\* 字段声明（包括枚举常量） \*/  
​  
  METHOD,             /\* 方法声明 \*/  
​  
  PARAMETER,         /\* 参数声明 \*/  
​  
  CONSTRUCTOR,       /\* 构造方法声明 \*/  
​  
  LOCAL\_VARIABLE,     /\* 局部变量声明 \*/  
​  
  ANNOTATION\_TYPE,   /\* 注释类型声明 \*/  
​  
  PACKAGE             /\* 包声明 \*/  
}

(3)RetentionPolicy.java

package java.lang.annotation;  
public enum RetentionPolicy {  
   SOURCE,            /\* Annotation信息仅存在于编译器处理期间，编译器处理完之后就没有该Annotation信息了 \*/  
​  
   CLASS,             /\* 编译器将Annotation存储于类对应的.class文件中。默认行为 \*/  
​  
   RUNTIME            /\* 编译器将Annotation存储于class文件中，并且可由JVM读入 \*/  
}

说明：

（1）annotation就是一个接口。

（2） ElementType 是 Enum 枚举类型，它用来指定 Annotation 的类型

（3）RetentionPolicy 是 Enum 枚举类型，它用来指定 Annotation 的策略。通俗点说，就是不同 RetentionPolicy 类型的 Annotation 的作用域不同。

### 7.5，java 自带的 Annotation

#### 5.1，Annotation 通用定义

@Documented  
@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
public @interface MyAnnotation1 {  
}

说明：

上面的作用是定义一个 Annotation，它的名字是 MyAnnotation1。定义了 MyAnnotation1 之后，我们可以在代码中通过 "@MyAnnotation1" 来使用它。 其它的，@Documented, @Target, @Retention, @interface 都是来修饰 MyAnnotation1 的。

（1）**@interface**

使用 @interface 定义注解时，意味着它实现了 java.lang.annotation.Annotation 接口，即该注解就是一个Annotation。

定义 Annotation 时，@interface 是必须的。

（2）**@Documented**

类和方法的 Annotation 在缺省情况下是不出现在 javadoc 中的。如果使用 @Documented 修饰该 Annotation，则表示它可以出现在 javadoc 中。

定义 Annotation 时，@Documented 可有可无；若没有定义，则 Annotation 不会出现在 javadoc 中。

（3）**@Target(ElementType.TYPE)**

前面我们说过，ElementType 是 Annotation 的类型属性。而 @Target 的作用，就是来指定 Annotation 的类型属性。

@Target(ElementType.TYPE) 的意思就是指定该 Annotation 的类型是 ElementType.TYPE。这就意味着，MyAnnotation1 是来修饰"类、接口（包括注释类型）或枚举声明"的注解。

（4）**@Retention(RetentionPolicy.RUNTIME)**

@Retention(RetentionPolicy.RUNTIME) 的意思就是指定该 Annotation 的策略是 RetentionPolicy.RUNTIME。这就意味着，编译器会将该 Annotation 信息保留在 .class 文件中，并且能被虚拟机读取。

### 7.6，Annotation 的作用

#### 6.1，编译检查

Annotation 具有"让编译器进行编译检查的作用"。

@Deprecated 和 @Override 都具有编译检查作用。

（1）@Deprecated ：标记过时方法。如果使用该方法，会报编译警告。

（2）@Override：检查该方法是否是重写方法。如果被他标注的方法不是重写其父类上的方法，就会报错

#### 6.2，在反射中使用 Annotation

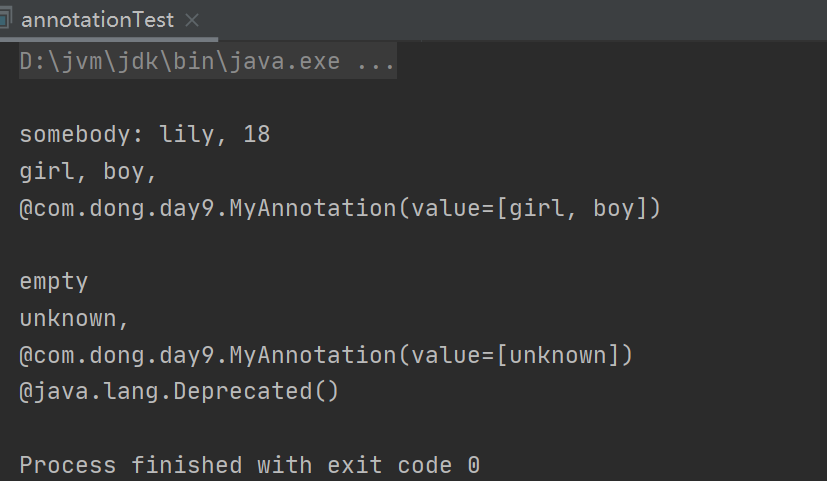
在反射的 Class, Method, Field 等函数中，有许多于 Annotation 相关的接口。

这也意味着，我们可以在反射中解析并使用 Annotation。

annotationTest.java

import java.lang.annotation.Annotation;  
import java.lang.annotation.Target;  
import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Inherited;  
import java.lang.reflect.Method;  
​  
/\*\*  
\* Annotation在反射函数中的使用示例  
\*/  
@Retention(RetentionPolicy.RUNTIME)  
@interface MyAnnotation {  
   String[] value() default "unknown";  
}  
​  
/\*\*  
\* Person类。它会使用MyAnnotation注解。  
\*/  
class Person {  
    
   /\*\*  
    \* empty()方法同时被 "@Deprecated" 和 "@MyAnnotation(value={"a","b"})"所标注  
    \* (01) @Deprecated，意味着empty()方法，不再被建议使用  
    \* (02) @MyAnnotation, 意味着empty() 方法对应的MyAnnotation的value值是默认值"unknown"  
    \*/  
   @MyAnnotation  
   @Deprecated  
   public void empty(){  
       System.out.println("\nempty");  
  }  
    
   /\*\*  
    \* sombody() 被 @MyAnnotation(value={"girl","boy"}) 所标注，  
    \* @MyAnnotation(value={"girl","boy"}), 意味着MyAnnotation的value值是{"girl","boy"}  
    \*/  
   @MyAnnotation(value={"girl","boy"})  
   public void somebody(String name, int age){  
       System.out.println("\nsomebody: "+name+", "+age);  
  }  
}  
​  
public class AnnotationTest {  
​  
   public static void main(String[] args) throws Exception {  
        
       // 新建Person  
       Person person = new Person();  
       // 获取Person的Class实例  
       Class<Person> c = Person.class;  
       // 获取 somebody() 方法的Method实例  
       Method mSomebody = c.getMethod("somebody", new Class[]{String.class, int.class});  
       // 执行该方法  
       mSomebody.invoke(person, new Object[]{"lily", 18});  
       iteratorAnnotations(mSomebody);  
        
​  
       // 获取 somebody() 方法的Method实例  
       Method mEmpty = c.getMethod("empty", new Class[]{});  
       // 执行该方法  
       mEmpty.invoke(person, new Object[]{});          
       iteratorAnnotations(mEmpty);  
  }  
    
   public static void iteratorAnnotations(Method method) {  
​  
       // 判断 somebody() 方法是否包含MyAnnotation注解  
       if(method.isAnnotationPresent(MyAnnotation.class)){  
           // 获取该方法的MyAnnotation注解实例  
           MyAnnotation myAnnotation = method.getAnnotation(MyAnnotation.class);  
           // 获取 myAnnotation的值，并打印出来  
           String[] values = myAnnotation.value();  
           for (String str:values)  
               System.out.printf(str+", ");  
           System.out.println();  
      }  
        
       // 获取方法上的所有注解，并打印出来  
       Annotation[] annotations = method.getAnnotations();  
       for(Annotation annotation : annotations){  
           System.out.println(annotation);  
      }  
  }  
}

结果：



#### 6.3，根据 Annotation 生成帮助文档

通过给 Annotation 注解加上 @Documented 标签，能使该 Annotation 标签出现在 javadoc 中。

#### 6.4，能够帮忙查看查看代码

通过 @Override, @Deprecated 等，我们能很方便的了解程序的大致结构。

另外，我们也可以通过自定义 Annotation 来实现一些功能。

## 八，泛型设计

**什么是泛型类？**

可以适应于多种类型，被称作泛型类

### 8.1 为什么要泛型设计

**原因**：泛型程序设计（Generic programming) 意味着编写的代码可以被很多不同类型的对象所重用。早期Java是使用Object来代表任意类型的，但是向下转型有强转的问题，这样程序就不太安全，有了泛型以后：

* 代码更加简洁【不用强制转换】
* 程序更加健壮【只要编译时期没有警告，那么运行时期就不会出现ClassCastException异常】
* 可读性和稳定性【在编写集合的时候，就限定了类型】

### 8.2 定义一个简单的泛型类

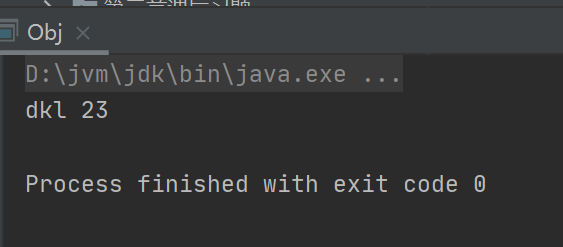
**程序清单8.1：**

**要求：定义一个简单的泛型类（具有构造器，方法和两个属性）**

**代码：**

package com.dong.day9.泛型设计;  
  
public class Obj<T> {  
 //属性  
 private T first;  
 private T second;  
  
 //构造器  
  
 public Obj(T first, T second) {  
 this.first = first;  
 this.second = second;  
 }  
 //方法  
 public T getFirst() {  
 return first;  
 }  
  
 public void setFirst(T first) {  
 this.first = first;  
 }  
  
 public T getSecond() {  
 return second;  
 }  
  
 public void setSecond(T second) {  
 this.second = second;  
 }  
  
 public static void main(String[] args) {  
 Obj<String> obj=new Obj<>("dkl","23");  
 System.*out*.println(obj.getFirst()+" "+obj.getSecond());  
  
 }  
}

结果：



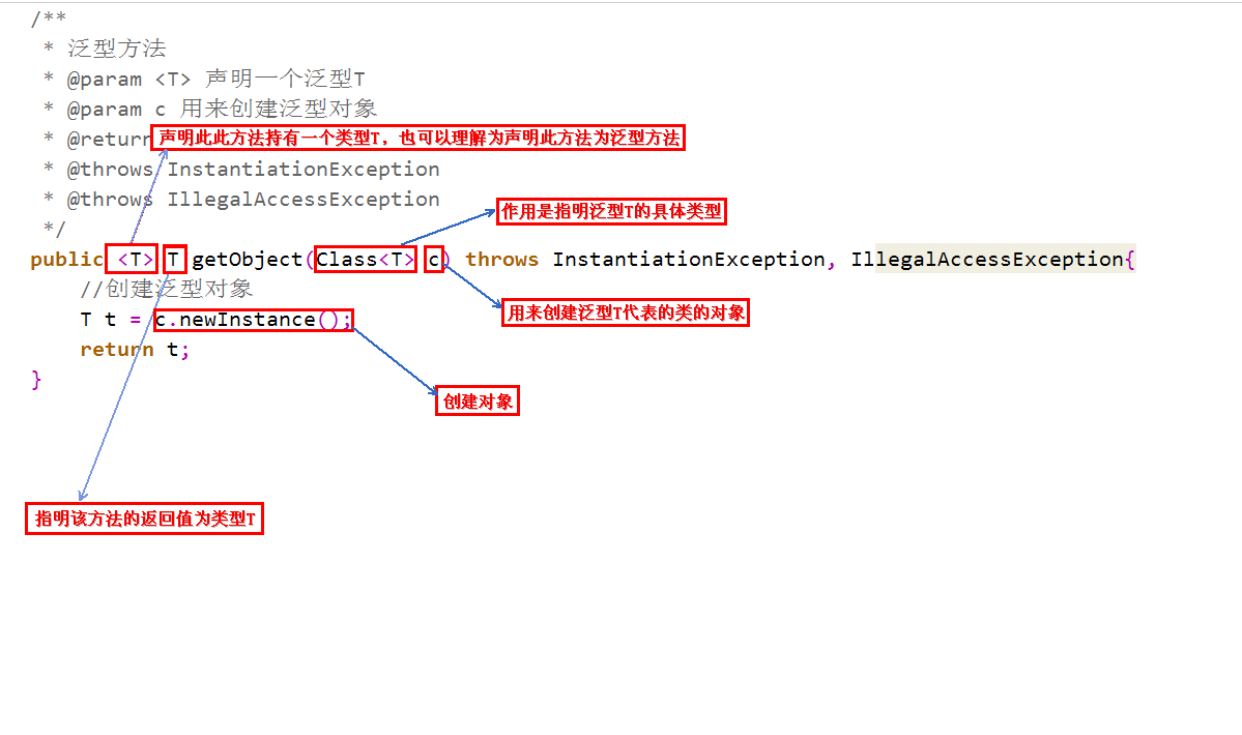
分析：**泛型设计就是将数据类型的定义时间置换到对象构建的时候。**如程序清单8.1就用

Obj<String> obj=new Obj<>("dkl","23");来限定数据类型

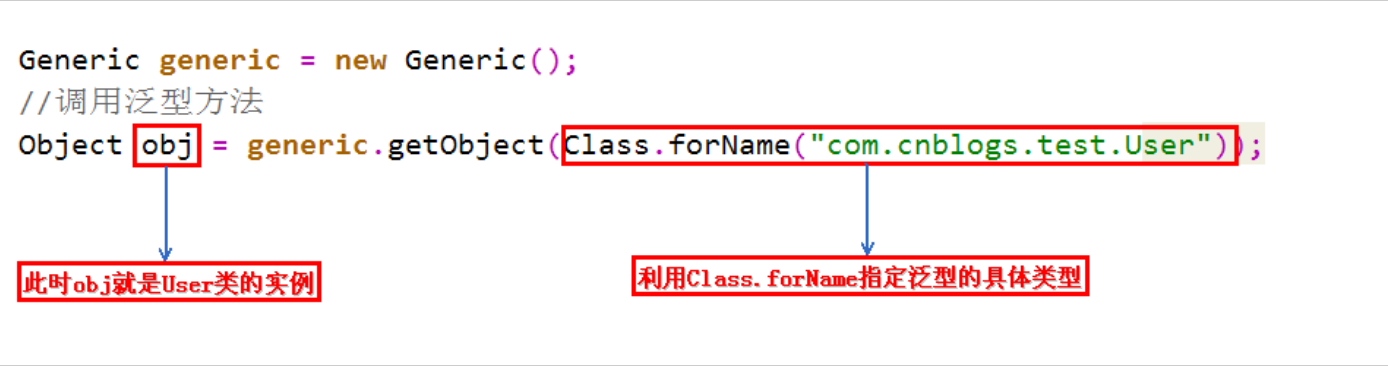
### 8.3 泛型方法

#### 8.3.1 泛型方法的定义

**泛型方法的语法：**



**泛型方法的调用：**



**分析：**

 说明一下，定义泛型方法时，必须在返回值前边加一个<T>，来声明这是一个泛型方法，持有一个泛型T，然后才可以用泛型T作为方法的返回值。如果不加<T>那么方法中的T其实就是泛型类的限定（也就是说方法的泛型类型必须和类的泛型类型相同）。

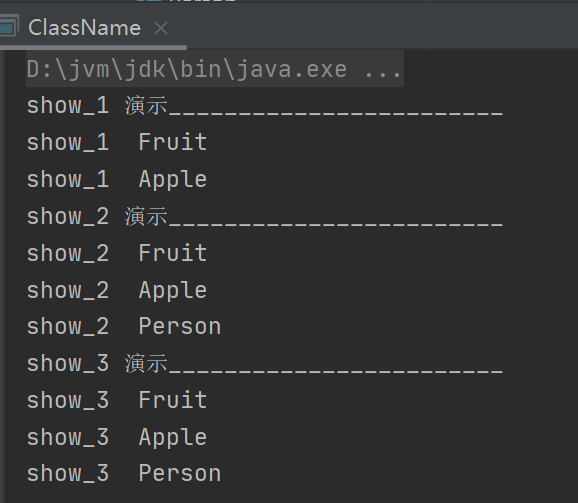
**程序清单8.2：**

**要求：验证上述分析**

**代码：**

package com.dong.day9.泛型设计;  
  
class Fruit { public String toString() { return "Fruit"; } }  
  
class Apple extends Fruit { public String toString(){ return "Apple"; } }  
  
class Person { public String toString(){ return "Person"; } }  
  
class ClassName<T> {//主类，文件名ClassName.java  
  
  
 void show\_1(T t){  
 System.*out*.println("show\_1 "+ t.toString());  
 }  
  
 <E> void show\_2(E e){  
 System.*out*.println("show\_2 "+e.toString());  
 }  
  
 <T> void show\_3(T t){  
 System.*out*.println("show\_3 "+t.toString());  
 }  
  
  
  
 public static void main(String[] args) {  
 ClassName<Fruit> o = new ClassName<Fruit>();  
 Fruit f = new Fruit();  
 Apple a = new Apple();  
 Person p = new Person();  
 System.*out*.println("show\_1 演示\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");  
 o.show\_1( f );  
 o.show\_1( a );  
// o.show\_1( p ); 这行代码是不能编译通过的。因为在  
// ClassName<Fruit>中已经限定了全局的T为Fruit，所以不能再加入Person;  
 System.*out*.println("show\_2 演示\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");  
 o.show\_2( f );  
 o.show\_2( a );  
 o.show\_2( p );  
 System.*out*.println("show\_3 演示\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");  
 o.show\_3( f );  
 o.show\_3( a );  
 o.show\_3( p );  
  
 }  
}

结果：



### 8.4 类型变量的限制

有时，类或方法需要对类型变量加以约束，使用关键词extends

Public <T extends Comparable> T min(T[] a)

上述的代码说明只能是实现comparable接口的类才能调用min方法

**注意**：1）类型变量的限定可以有多个，通过&分隔

2）在Java 的继承中，可以根据需要拥有多个接口超类型，但限定中至多有一个

如果用一个类作为限定它必须是限定列表中的第一个。

**程序清单8.3：**

**要求：**

**1，构建一个实现comparable接口的实现类**

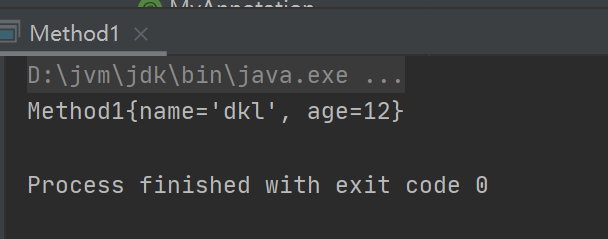
**2，构建一个泛型方法：Public <T extends Comparable> T min(T[] a)**

**3，调用这个泛型方法**

代码：

package com.dong.day9.泛型设计;  
  
public class Method1 implements Comparable{  
 String name;  
 int age;  
  
 public Method1() {  
 }  
  
 public Method1(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
  
  
 @Override  
 public int compareTo(Object o) {  
  
 return this.age-((Method1)o).age;  
  
 }  
  
 public <T extends Comparable> T min(T[] a)  
 {  
 T m=a[0];  
 for(int i=1;i<a.length;i++)  
 if(m.compareTo(a[i])>0)  
 {  
 m=a[i];  
 }  
 return m;  
 }  
  
 @Override  
 public String toString() {  
 return "Method1{" +  
 "name='" + name + '\'' +  
 ", age=" + age +  
 '}';  
 }  
  
 public static void main(String[] args) {  
 Method1[] method1= new Method1[3];  
 method1[0]=new Method1("dkl",12);  
 method1[1]=new Method1("xwb",13);  
 method1[2]=new Method1("wkw",14);  
  
 Method1 method11=new Method1();  
 System.*out*.println(method11.min(method1));  
  
  
  
 }  
}

结果：



### 8.5 泛型代码和虚拟机

1.虚拟机没有泛型类型对象—-所有的对象都属于普通类。  
2.无论何时定义一个泛型类型，都自动提供了一个相应的原始类型（raw type）。原始类型的名字就是删去类型参数后的泛型类型名。  
3.原始类型（分析的原始类型）

Pair<T>的原始类型

//擦除类型变量，并替换为限定类型（原始类型用第一个限定的类型变量来替换，无限定的变量用Object）

public class pair{

private Object first;

private Object second;

public Pair(){

first = null;

second = null;

}

public Pair(Object first, Object second){

this.first = first;

this.second = second;

}

public Object getFirst(){

return first;

}

public Object getSecond(){

return second;

}

public void setFirst(Object newValue){

first = new Value;

}

public void setSecond(Object newValue){

second= new Value;

}

}

4**.翻译泛型表达式**，如果擦除返回类型，编译器插入强制类型转换。

Pair<Employee> buddies = ...;

Employee buddy = buddies.getFirst();

擦除getFirst 方法类型后将返回一个Object类型。编译器会自动插入Employee的强制类型转换。

5**.翻译泛型方法**

public static <T extends Comparable> T min(T[] a) //是一个完整的方法族，但是擦除后只剩下一个方法:

public static Comparable min(Comparable[] a)

方法的擦除带来了两个复杂问题：

问题在于类型擦除与多态发生了冲突。要解决这个问题，就需要编译器在类中生成一个桥方法。

**桥方法**：详细看此：<https://blog.csdn.net/lcb1992/article/details/53116763>

总结：

**① 虚拟机中没有泛型，只有普通类和方法。**  
**② 在编译阶段，所有泛型类的类型参数都会被Object或者它们的限定边界来替换。(类型擦除)**  
**③ 在继承泛型类型的时候，桥方法的合成是为了避免类型变量擦除所带来的多态灾难。**

### 8.6 泛型的约束和局限性

#### 8.6.1 Jvm如何理解泛型

 事实上，JVM并不知道泛型，所有的泛型在编译阶段就已经被处理成了普通类和方法。   
    处理方法很简单，我们叫做类型变量T的**擦除(erased)**。  
    无论我们如何定义一个泛型类型，相应的都会有一个原始类型被自动提供。原始类型的名字就是擦除类型参数的泛型类型的名字。  
         如果泛型类型的类型变量没有**限定(<T>)**，那么我们就用Object作为原始类型；  
         如果有限定(<T extends XClass>)，我们就XClass作为原始类型；  
         如果有多个限定(<T extends XClass1&XClass2>)，我们就用第一个边界的类型变量XClass1类作为原始类型；

**程序清单8.6：**

**要求：泛型擦除的举例：**

**代码：**

public class Pair<T>{    
       private T first=null;    
       private T second=null;    
    
       public Pair(T fir,T sec){    
            this.first=fir;    
        this.second=sec;    
       }    
       public T getFirst(){    
             return this.first;    
       }    
       public T getSecond(){    
             return this.second;    
       }    
       public void setFirst(T fir){    
         this.first=fir;    
       }

**Jvm泛型擦除后的结果：**

public class Pair{

private Object first=null;

private Object second=null;

public Pair(Object fir,Object sec){

this.first=fir;

this.second=sec;

}

public Object getFirst(){

return this.first;

}

public void setFirst(Object fir){

this.first=fir;

}

}

#### 8.6.2运行时类型查询只适用于原始类型：

虚拟机中的对象总有一个特定的非泛型类型。因此，所有的类型查询只产生原始类型。

if (a instanceof Pair<String>) // Error

if (a instanceof Pair<T>) // Error

Pair p = (Pair<String>) a; // Warning-can only test that a is a Pair

#### 8.6.3不能创建参数化类型的数组:

例如：

Pair<String>[] stringPairs=new Pair<String>[10];

Pair<Integer>[] intPairs=new Pair<Integer>[10];

 这种写法编译器会指定一个Cannot create a generic array of Pair<String>的错误

      我们说过泛型擦除之后，Pair<String>[]会变成Pair[]，进而又可以转换为Object[];

      假设泛型数组存在，那么

Object[0]=stringPairs[0];// Ok

Object[1]=intPairs[0]; //Ok

 这就麻烦了，理论上将Object[]可以存储所有Pair对象，但这些Pair对象是泛型对象，他们的类型变量都不一

样，那么调用每一个Object[]数组元素的对象方法可能都会得到不同的记过，也许是个字符串，也许是整形，这对于

JVM可是无法预料的。

      记住： 数组必须牢记它的元素类型，也就是所有的元素对象都必须一个样，泛型类型恰恰做不到这一点。即使

Pair<String>,Pair<Integer>... 都是Pair类型的，但他们还是不一样。

### 8.7 泛型类型的继承规则

已知S,T是两个类。

1.无论 S 与 T 有什么联系，通常Pair<S>和Pair<T>没有什么关系。

2. 永远可以将参数化类型转换为一个原始类型

3. 泛型类可以扩展或实现其他的泛型类。与普通类没有区别。

### 8.8 通配符类型

当声明一个方法时，某个形参的类型是一个泛型类或泛型接口类型，但是在声明方法时，又不确定该泛型实际类型，可以考虑使用类型通配符。

看个案例：

public static void test(List c){

for (int i = 0; i < c.size(); i++) {

System.out.println(c.get(i));

}

}

上面的方法执行是没有问题的，但是此处使用 List 接口时没有传入实际类型参数，这将引起泛型警告。如何消除这个警告呢？

方法一：

public static void test1(List<Object> c){

for (int i = 0; i < c.size(); i++) {

System.out.println(c.get(i));

}

}

　这样的形参太局限了，只能传入 List<Object> 类型的实参。

方法二：定义一个泛型方法

public static <T> void test2(List<T> c){

for (int i = 0; i < c.size(); i++) {

System.out.println(c.get(i));

}

}

**该方法需要声明泛型形参T。**

方式三：使用通配符

public static void test3(List<?> c){

for (int i = 0; i < c.size(); i++) {

System.out.println(c.get(i));

}

}

**方法二和方法三有没有什么差别？**

test3方法带通配符的List仅表示它可以接受指定了任意泛型实参的List，并不能把元素加入其中，例如如下代码将会引起编译错误：

public static void test(List<?> c, String str){

c.add(str);

}

 因为我们不知道上面程序中c集合里元素的类型，所以不能向其中添加对象，除了null对象，因为它是所有引用数据类型的实例。

　　   test2方法带泛型的List，表示该集合的元素类型是T，因此允许T系列的对象加入其中，例如如下代码是可行的：

public static <T> void test(List<T> c, T t){

c.add(t);

}

即如果不涉及添加元素到带泛型的集合中，那么两种方式都可以，如果涉及到添加元素到带泛型的集合中，使用类型通配符<?>的不支持。

#### 8.8.1 通配符的上限

当直接使用 List<?> 这种形式时，即表明这个 List 集合接收泛型实参指定为任意类型的 List。但有时候不想这样，只希望接收某些类型的 List。

　　Demo：一个图形的抽象父类 Graphic，两个子类 Circle和Rectangle，想定义一个方法，可以打印不同图形的面积。

public static void printArea(List<Graphic> graphics){

for (Graphic g : graphics) {

System.out.println(g.getArea());

}

}

但是，List<Graphic> 的形参只能接收 List<Graphic>的实参，如果想要接收 List<Circle>,List<Graphic>的集合，可以使用List<?>

public static void printArea(List<?> graphics){

for (Object obj : graphics) {

Graphic g = (Graphic) obj;

System.out.println(g.getArea());

}

}

但是这样有两个问题，一个是 List<?> 可以接收任意类型，不仅仅图形，第二个是需要强制类型转换。

　　为了解决这个问题，Java 允许设定通配符的上限：

**<? extends Type>，这个通配符表示它必须是Type本身，或是Type的子类。**

**代码：**

public static void printArea(List<? extends Graphic> graphics){

for (Graphic g : graphics) {

System.out.println(g.getArea());

}

}

与前面的完全相同，因为不知道这个受限制的通配符的具体类型，所以不能把 Graphic 对象或其子类对象加入到这个泛型集合中。

**代码：**

public static void printArea(List<? extends Graphic> graphics){

graphics.add(new Circle());//编译错误，因为不知道？的具体类型，也可能是Rectangle

}

如果需要将 Graphic 对象或其子类对象加入这个泛型集合，那么就只能用泛型方法。

#### 8.8.2 设定通配符的下限

假设自己实现一个工具方法：实现将 src 集合里元素复制到 dest 集合中的功能，因为 dest 集合需要接受 src 的所有元素，所以 dest 集合元素的类型应该是 src 集合元素的父类（子类的对象也是父类的对象，所以这里dest应该是src的父类）。为了表示两个参数之间的类型依赖，考虑同时使用通配符、泛型形参类实现该方法。

**代码：**

public static <T> void copy(Collection<T> dest,Collection<? extends T> src){

for (T t : src) {

dest.add(t);

}

}

上面的方法实现了前面的功能。现在假设该方法需要一个返回值，返回最后一个被复制的元素，可以把上面方法修改如下：

**代码：**

public static <T> T copy(Collection<T> dest,Collection<? extends T> src){

T last = null;

for (T t : src) {

dest.add(t);

last = t;

}

return last;

}

表面上看这个实现没有问题，实际上有一个问题，当遍历src元素的类型是不确定（但可以肯定是T的子类），程序只能用T来笼统的表示，所以返回值类型是T。

**代码：**

public static void main(String[] args) {

ArrayList<String> src = new ArrayList<String>();

src.add("Hello");

src.add("World");

ArrayList<Object> dest = new ArrayList<Object>();

Object last = copy(dest, src);

}

**分析：**

发现返回的T是Object，也就是说，程序在复制集合元素的过程中，丢失了src集合元素的类型String。

　　对于上面的copy方法，可以这样理解两个集合参数之间的依赖关系：不管src集合元素的类型是什么，只要dest集合元素类型与src的元素类相同或是它的父类即可。

　　为了表示这种约束关系，Java允许设定通配符的下限：

**语法格式：**

**<? super Type>，这个通配符表示它必须是Type本身或是Type的父类。**

代码：

public static <T> T copy(Collection<? super T> dest,Collection<T> src){

T last = null;

for (T t : src) {

dest.add(t);

last = t;

}

return last;

}

public static void main(String[] args) {

ArrayList<String> src = new ArrayList<String>();

src.add("Hello");

src.add("World");

ArrayList<Object> dest = new ArrayList<Object>();

String last = copy(dest, src);

}

## 九，集合

### 9.1 java集合框架

Java最初版本只是为最常用的数据结构提供很少的一组类：Vector，Stack，Hashtable，bitset与Enumeration接口，Enumeration接口提供一种用于访问任何容器中各个元素的抽象机制。

#### 9.1.1 将集合的实现与接口分离

Java集合库也将接口与实现分离。

**队列接口**：可以在队列的尾部添加元素，在队列的头部删除元素，并且可以査找队列中元素的个数。

**代码：**

Public interface queue<E>

{

Void add(E element);

E remove();

Int size();

}

//通过循环数组实现queue接口

Public class circulararrayqueue<E> implement queue<E>

{

Private int head;

Private int tail;

Circulararrayqueue(int capacity){…………} //构造器

Public void add(E element) {…………}

Public E remove(){…………}

Public int size(){…………}

Private E[] elements;

}

**注意：当在程序中使用队列时，一旦构造了集合就不需要知道究竟使用了哪种实现。因此，只有在构造集合对象时，使用具体的类才有意义。**

**如：**

queue<customer> expresslane=new circulararrayqueue<>(100);

Expresslane.add(new customer(“Harry”));

#### 9.1.2 Collection接口

在 Java 类库中，集合类的基本接口是 Collection 接口，这个接口有两个基本方法：

Public interface collection<E>

{

Boolean add(E element); //向集合中添加元素，如果添加元素确实改变了集合就返回true

Iterator<E> iterator(); /\*iterator方法用于返回一个实现了Iterator接口的对象, 可以使用这个迭代器对象依次访问集合中的元素\*/

…………

}

#### 9.1.3 迭代器

**Iterator接口包含4个方法：**

**Public interface Iterator<E>**

**{**

**E next();**

**Boolean hasnext();**

**Void remove();**

**Default void foreachremaining(consumer<? Super E>action);**

**}**

通过反复调用 next 方法,可以逐个访问集合中的每个元素.

Collection<String>c=…;

Iterator<String>iter=c.iterator();

While(iter.hasnext())

{

String element=iter.next(); //重复调用next产生迭代

Do some things;

}

用for……each更简单

Collection<String>c=…;

**For(string element:c)**

**{**

**Do something with element;**

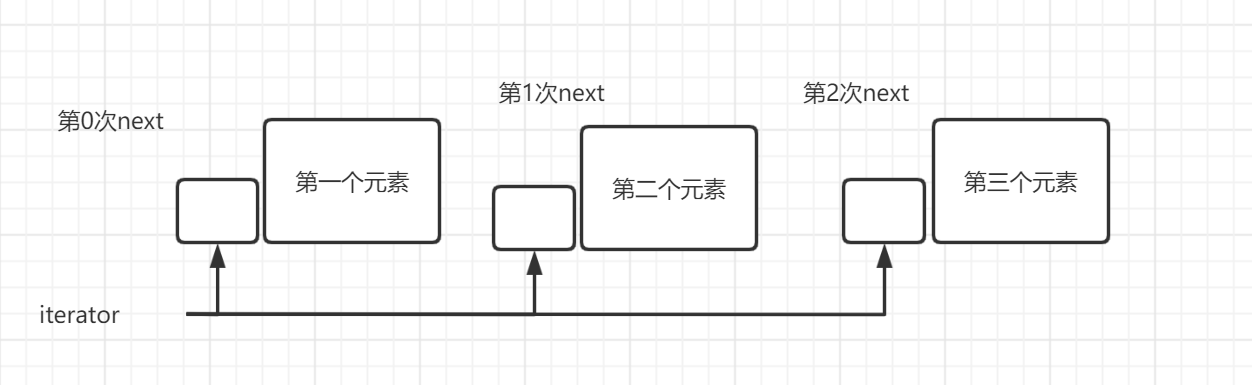
**}**

“ for each” 循环可以与任何实现了 Iterable 接口的对象一起工作, Collection 接口扩展了 Iterable接口.

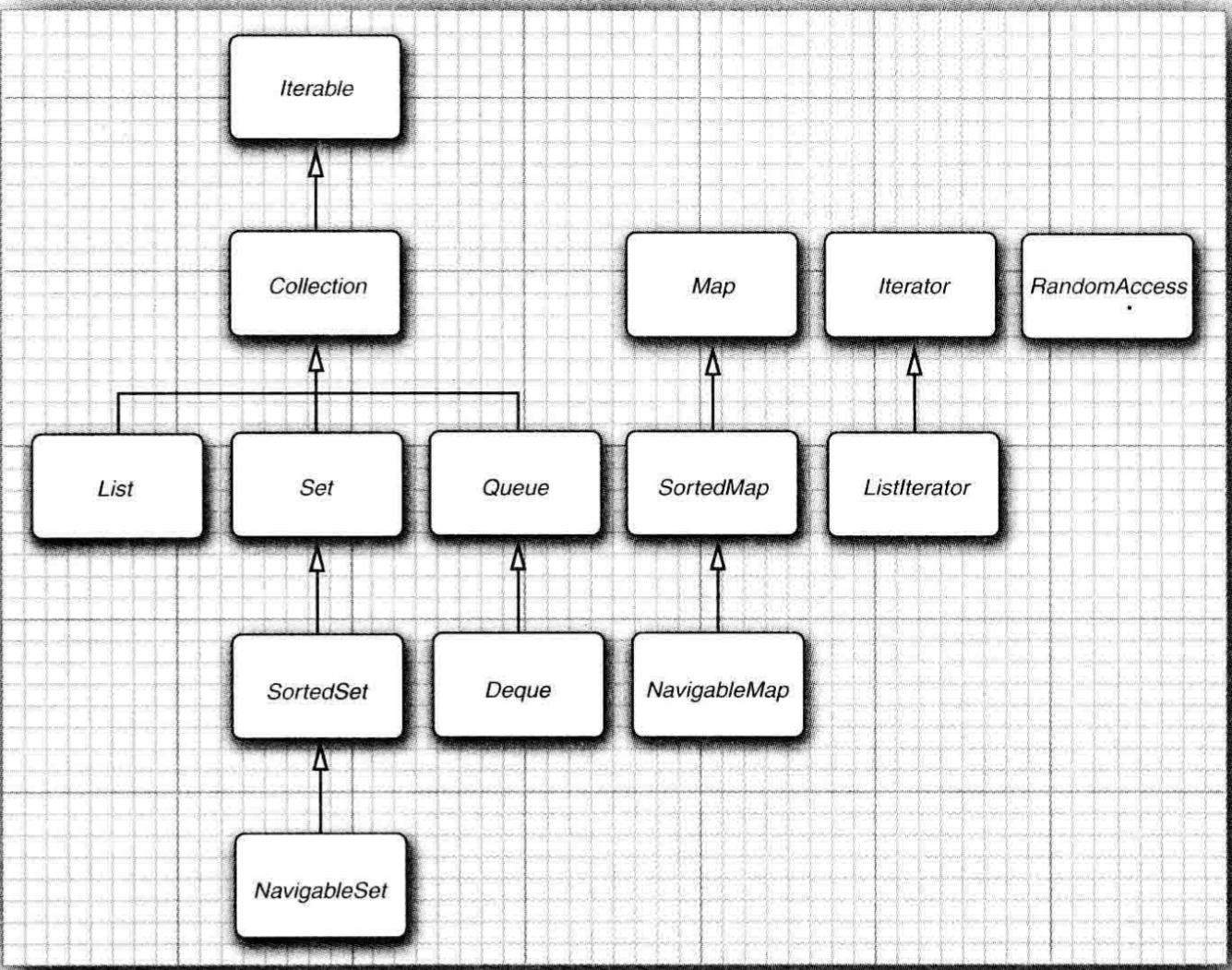
**划重点**：应该将 Java 迭代器认为是位于两个元素之间

Iterator 接口的 remove 方法将会删除上次调用 next 方法时返回的元素.

如果调用 remove 之前 没有调用 next 将是不合法的.



#### 9.1.4 集合框架中的接口



集合有两个基本接口：Collection 和 Map。

**集合的插入**：

Boolean add(E element); //collection接口

V put(K key,V value); //map接口

**集合的读取**：

要从集合读取元素， 可以用迭代器访问元素。不过，从映射中读取值则要使用 get 方法：

V get(K key);

Collection接口旗下的容器：

**1.List接口**

特点：1，加 到 容 器 中 的 特 定 位 置

2，有两种访问形式：使用迭代器访问， 或者使用一个整数索引来访问

**2 .Set 接口**

Set 接口等同于 Collection 接口，不过其方法的行为有更严谨的定义,集（set) 的 add方 法不允许增加重复的元素.

**Iterator:**

Listlterator 接口是 Iterator 的一个子接口. 它定义了一个方法用于在迭代器位置前面增加 一个元素:void add(E element)

### 9.2 具体的集合

#### 9.2.1 列表数组

##### 9.2.1.1什么是列表数组？

数组列表是一个无限长的数组，他的特点是易于查找数据，缺点是每次删除或插入一个数据后都要移动数组，会影响代码性能。

##### 9.2.1.2 ArrayList类

在java中ArrayList类是数组列表这一数据结构的工具类。现在我来介绍这个类的使用方法。

**程序清单9.1：**

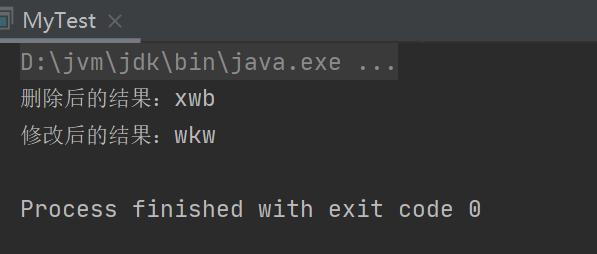
**要求：**

**1，使用ArrayList类中的add方法，set方法，remove方法**

**代码：**

package com.dong.day10.集合.链表;  
  
import java.util.ArrayList;  
  
public class MyTest {  
 public static void main(String[] args) {  
 ArrayList<String>it=new ArrayList<>();  
 //插入  
 it.add("dkl");  
 it.add(1,"xwb");  
 //删除  
 it.remove(0);  
 System.*out*.println("删除后的结果："+it.get(0));  
 //修改  
 it.set(0,"wkw");  
 System.*out*.println("修改后的结果："+it.get(0));  
  
 }  
}

结果：



#### 9.2.2 链表

在 Java 程序设计语言中，所有链表实际上都是双向链接。

Arraylist类有一个重大缺陷，从数组中删除一个元素需要付出很大的代价，于是类 LinkedList横空出现。

##### 9.2.2.1 LinkedList类的使用方法

**程序清单9.2：**

**要求：**

**1，使用add方法，remove方法，ListIterator迭代器**

**代码：**

package com.dong.day10.集合.链表;  
  
import java.util.Iterator;  
import java.util.LinkedList;  
import java.util.ListIterator;  
  
public class MyTest {  
 public static void main(String[] args) {  
 LinkedList<String> linkedList=new LinkedList<>();  
 //插入，如果没有index参数就从链表的末尾插入，如果有就插入指定的位置  
 linkedList.add("dkl");  
 linkedList.add(1,"xwb");  
 System.*out*.println("链表的数据：");  
 for(String elem:linkedList)  
 System.*out*.println(elem);  
  
 //迭代器的使用  
 System.*out*.println("迭代器输出的数据：");  
 ListIterator<String>it=linkedList.listIterator();  
 while(it.hasNext())  
 {  
 String name=it.next();  
 System.*out*.println(name);  
 }  
 System.*out*.println("逆序输出");  
 while(it.hasPrevious())  
 {  
 String name=it.previous();  
 System.*out*.println(name);  
 }  
 //删除第1个数据  
 linkedList.remove(0);  
 System.*out*.println("删除第一个数据后的第一个数据是："+linkedList.get(0));  
 }  
}

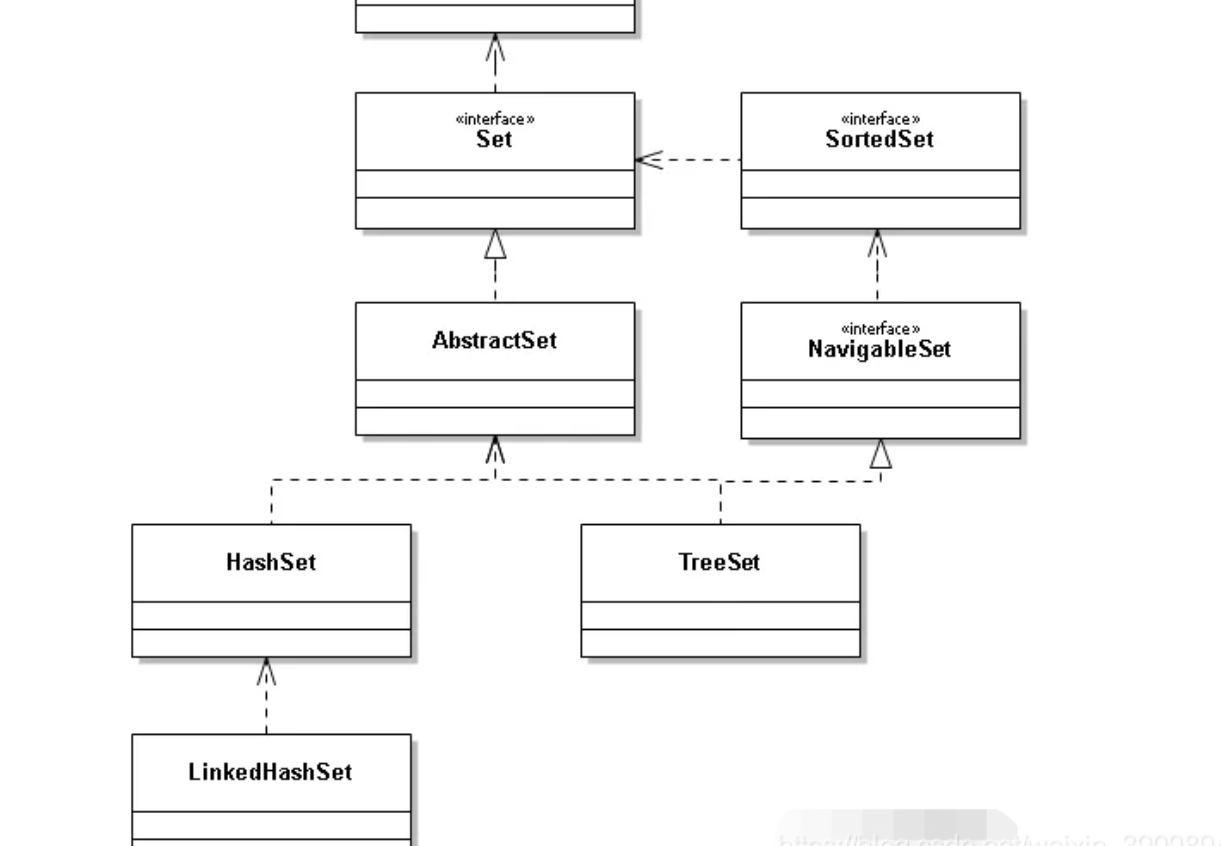
结果：



#### 9.2.3 散列集

**散列集的定义：**

HashSet是Java集合Set的一个实现类，Set是一个接口，其实现类除HashSet之外，还有TreeSet，并继承了Collection，HashSet集合很常用，同时也是程序员面试时经常会被问到的知识点，下面是结构图



##### 9.2.3.1 hashset类的使用方法

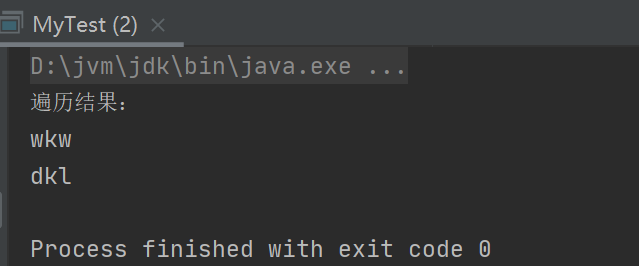
**程序清单9.3：**

**要求：remove，add，迭代器遍历set**

**代码：**

package com.dong.day10.集合.散列集;  
  
import java.util.HashSet;  
import java.util.Iterator;  
import java.util.LinkedList;  
import java.util.List;  
  
public class MyTest {  
 public static void main(String[] args) {  
 HashSet<String> hashSet=new HashSet<>();  
 hashSet.add("dkl");  
 hashSet.add("xwb");  
 hashSet.add("wkw");  
 hashSet.add("dkl");  
 List<String>list=new LinkedList<>();  
 //删除  
 hashSet.remove("xwb");  
 //采用迭代器遍历  
 Iterator<String>it=hashSet.iterator();  
 while(it.hasNext())  
 {  
 System.*out*.println( it.next());  
 }  
 }  
}

结果：



#### 9.2.4 树集

**TreeSet类**与散列集十分类似，不过它比散列集有所改进。  
树集是一个**有序集合**。可以以任意顺序将元素插入集合中。在对集合进行遍历时，每个值将自动地按照排序后的顺序呈现。当然，排序是用树结构完成的（当前实现使用的是红黑树），每次将一个元素添加到树中时，都被放置在正确的排序位置上。因此，迭代器总是以排好序的顺序访问每一个元素。

**首先，了解SortedSet接口，它里面的元素一定是有序的。  
其次，TreeSet是SortedSet接口的唯一实现类。  
最后，插入有序集合（比如Tree）的所有元素都要符合Comparable接口。**

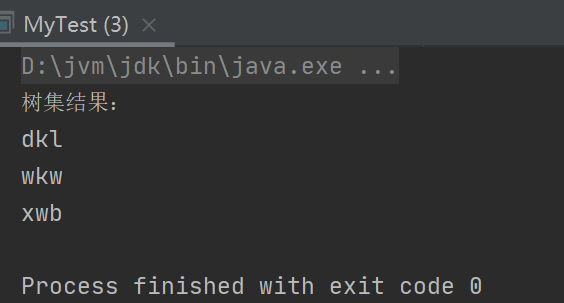
**程序清单：**

**要求：树集的简单举例**

**代码：**

package com.dong.day10.集合.树集;  
  
import java.util.SortedSet;  
import java.util.TreeSet;  
  
public class MyTest {  
 public static void main(String[] args) {  
 SortedSet<String>sorter=new TreeSet<>();  
 sorter.add("dkl");  
 sorter.add("xwb");  
 sorter.add("wkw");  
 System.*out*.println("树集结果：");  
 for(String elem:sorter)  
 System.*out*.println(elem);  
 }  
}

结果：



分析：

**将元素添加到 树（TreeSet）中要比添加到散列表（HashSet）中慢，** 但是，与将元素添加到数组或链表的正确位置上相比还是快很多的；

 TreeSet 的排序功能用到了红黑树， 因此迭代器总是以排好序的顺序访问每个元素）

##### 9.2.4.1 树集比较排序的规则是什么

 TreeSet如何知道希望元素怎样排列呢？ 在默认情况下， 树集假定插入的元素实现了 Comparable 接口， 这个接口定义了一个方法：

public **interface Comparable<T>**

{

in compareTo(T other);

}

**注意**：1，如果要插入自定义对象， 就必须通过实现 Comparable接口自定义排列顺序。

2，在Object类中，没有提供任何 compareTo 接口的默认实现 **；**

举个例子：

**（如何用部件编号对 Item 对象进行排序）**

**代码：**

**class Item implements Comparable<Item> {**

public int compareTo(Item other) //实现Comparable接口的方法

{

return partNumber - other.partNumber;

}

}

**使用Comparable接口有一个明显的缺点：**

对于一个给定的类， 只能实现这个接口一次。如果在一个集合中需要按照部件编号进行排序， 在另一个集合中 却要按照描述信息进行排序， 该怎么办呢？还有，如果需要对一个类的对象进行排序， 而这个类的创建者又没有实现 Comparable接口， 又该怎么办呢？

**解决的方法：**

通过将 Comparator 对象传递给TreeSet构造器来告诉树集使用不同的比较方法。 Comparator接口声明了一个带有两个显式参数的compare 方法:

public **interface Comparator<T>**

{

int compare(T a, T b);

}

解决上述问题的代码：

代码：

**实现Comparator接口的类**

**class ItemComparator implements Comparator<Item> {**

public int compare(Item a, Item b)

{

String desrcA = a.getDescription(); //获取需要比较的数据

String desrcB = b.getDescription();

String desrcA.compareTo(desrcB); //进行比较

}

}

**将实现类的对象传入树集：**

ItemComparator comp = new ItemComparator();

SortedSort<Item> sort = new TreeSet<>(comp)；

#### 9.2.5 队列和双端队列

**结构特征**：队列可以让人们有效地在尾部添加一个元素， 在头部删除一个元 素。有两个端头的队列， 即双端队列，可以让人们有效地在头部和尾部同时添加或删除元 素。不支持在队列中间添加元素。

Queue 普通队列

Deque 双端队列

Arraydeque 无限双端队列

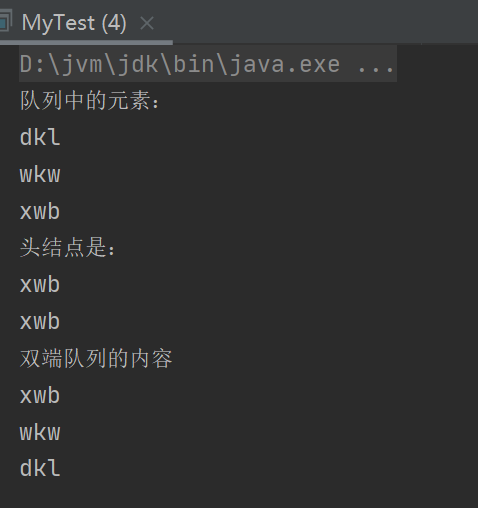
程序清单9.5：

要求：队列，双端队列的简单使用

代码：

package com.dong.day10.集合.队列;  
  
import java.util.Deque;  
import java.util.LinkedList;  
import java.util.Queue;  
  
public class MyTest {  
 public static void main(String[] args) {  
 Queue<String>queue=new LinkedList<>();  
 queue.add("dkl");  
 queue.add("wkw");  
 queue.add("xwb");  
 System.*out*.println("队列中的元素：");  
 for(String elem:queue)  
 System.*out*.println(elem);  
 //队列的数据结构不能够从中间删除节点，但是封装好的remove方法可以实现删除节点  
 queue.remove("wkw");  
 //没有带参数的remove方法默认是从头结点开始删除  
 queue.remove();  
 System.*out*.println("头结点是：");  
 System.*out*.println(queue.peek());//peek返回头结点但是不删除  
 System.*out*.println(queue.poll());//poll返回头结点并删除  
  
 Deque<String>deque=new LinkedList<>();  
  
 deque.addFirst("xwb");  
 deque.addLast("wkw");  
 deque.add("dkl"); //默认从最后一个插入  
  
 System.*out*.println("双端队列的内容");  
 for(String elem:deque)  
 System.*out*.println(elem);  
  
   
 }  
}

结果：



分析：

**offer，add区别：**

**一些队列有大小限制，因此如果想在一个满的队列中加入一个新项，多出的项就会被拒绝。**

**这时新的 offer 方法就可以起作用了。它不是对调用 add() 方法抛出一个 unchecked 异常，而只是得到由 offer() 返回的 false。**

**poll，remove区别：**

**remove() 和 poll() 方法都是从队列中删除第一个元素。remove() 的行为与 Collection 接口的版本相似，**

**但是新的 poll() 方法在用空集合调用时不是抛出异常，只是返回 null。因此新的方法更适合容易出现异常条件的情况。**

**peek，element区别：**

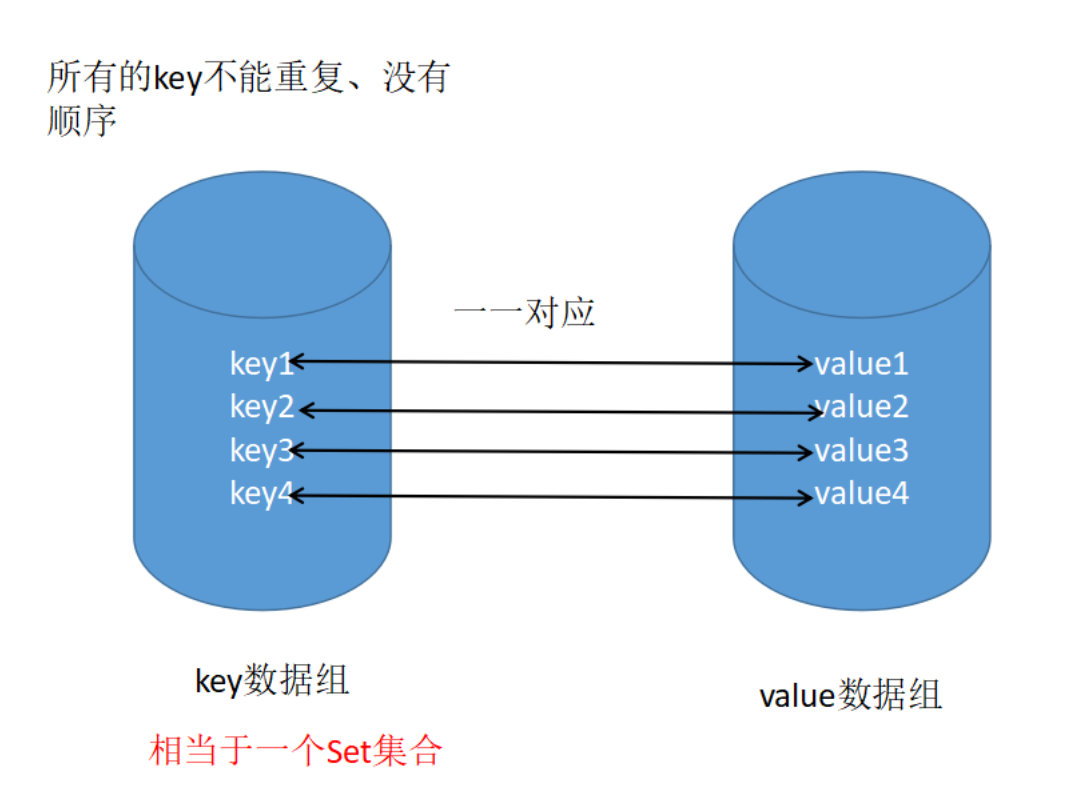
**element() 和 peek() 用于在队列的头部查询元素。与 remove() 方法类似，在队列为空时， element() 抛出一个异常，而 peek() 返回 null**

### 9.3 映射

#### 9.3.1什么是map集合？

Map用于保存具有映射关系的数据，Map集合里保存着两组值，一组用于保存Map的ley，另一组保存着Map的value。

图解：



#### 9.3.2 Map集合的作用

和查字典类似，通过key找到对应的value，通过页数找到对应的信息。用学生类来说，key相当于学号，value对应name，age，sex等信息。用这种对应关系方便查找。

#### 9.3.3 Map和set的关系

可以说关系是很密切了，虽然Map中存放的时键值对，Set中存放的是单个对象，但如果把value看做key的附庸，key在哪里，value就在哪里，这样就可以像对待Set一样来对待Map了。事实上，Map提供了一个Entry内部类来封装key-value对，再计算Entry存储时则只考虑Entry封装的key。

如果把Map集合里的所有value放在一起来看，它们又类似于一个List，元素可以重复，每个元素可以根据索引来找，只是Map中的索引不再是整数值，而是以另一个对象作为索引。

#### 9.3.4 Map的基本方法

* void clear():删除该Map对象中所有键值对；
* boolean containsKey(Object key):查询Map中是否包含指定的key值；
* boolean containsValue(Object value):查询Map中是否包含一个或多个value;
* **Set entrySet():返回map中包含的键值对所组成的Set集合，每个集合都是Map.Entry**对象。
* **Object get()：返回指定key对应的value，如果不包含key则返回null；**
* boolean isEmpty():查询该Map是否为空；
* **Set keySet():返回Map中所有key组成的集合；**
* **Collection values():返回该Map里所有value组成的Collection。**
* **Object put(Object key,Object value):添加一个键值对，如果集合中的key重复，则覆盖原来的键值对；**
* void putAll(Map m):将Map中的键值对复制到本Map中；
* **Object remove(Object key):删除指定的key对应的键值对，并返回被删除键值对的value，如果不存在，则返回null；**
* **boolean remove(Object key,Object value):删除指定键值对，删除成功返回true；**
* int size():返回该Map里的键值对个数；

#### 9.3.5 内部类：entry

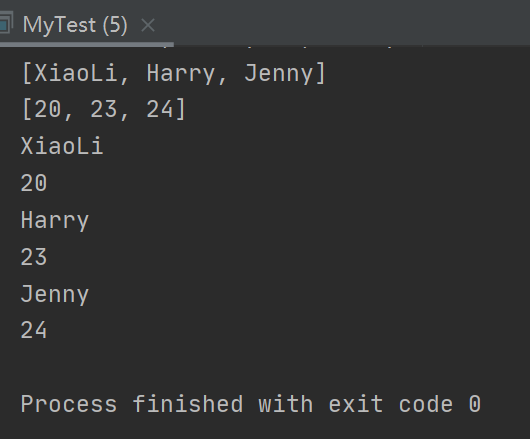
Map中包括一个内部类Entry,该类封装一个键值对，常用方法：

* Object getKey():返回该Entry里包含的key值；
* Object getvalue():返回该Entry里包含的value值；
* Object setValue(V value):设置该Entry里包含的value值，并设置新的value值。

举例：

package com.dong.day10.集合.映射;  
  
import java.util.HashMap;  
import java.util.Map;  
import java.util.Set;  
  
public class MyTest {  
 public static void main(String[] args) {  
 HashMap<String, Integer> hm = new HashMap<>();  
  
//放入元素  
 hm.put("Harry",23);  
 hm.put("Jenny",24);  
 hm.put("XiaoLi",20);  
  
 System.*out*.println(hm);//{XiaoLi=20, Harry=23, Jenny=24}  
 System.*out*.println(hm.keySet());//[XiaoLi, Harry, Jenny]  
 System.*out*.println(hm.values());//[20, 23, 24]  
  
 Set<Map.Entry<String, Integer>> entries = hm.entrySet();  
  
 for (Map.Entry<String, Integer> entry : entries) {  
 System.*out*.println(entry.getKey());  
 System.*out*.println(entry.getValue());  
 }  
 }  
}

结果：



**Java8新增的方法：**

* void forEach(BiConsumer action):遍历键值对。

**代码：**

public class MyTest {

public static void main(String[] args) {

HashMap<String, Integer> hm = new HashMap<>();

//放入元素

hm.put("Harry",23);

hm.put("Jenny",24);

hm.put("XiaoLi",20);

System.out.println(hm);//{XiaoLi=20, Harry=23, Jenny=24}

System.out.println(hm.keySet());//[XiaoLi, Harry, Jenny]

System.out.println(hm.values());//[20, 23, 24]

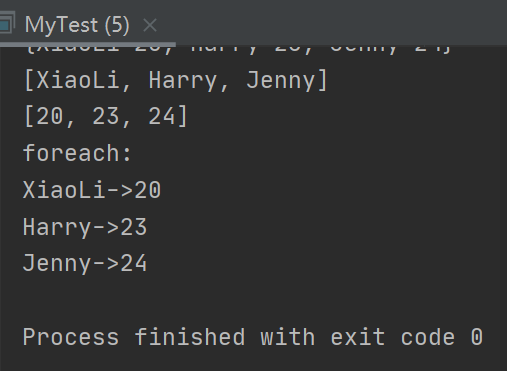
System.out.println("foreach:");

hm.forEach((key,value)-> System.out.println(key+"->"+value));

}

}

结果：



#### 9.3.6 LinkedhashMap类

和HashSet中的LinkedHashSet一样，HashMap也有一个LinkedHashMap子类，使用双向链表来维护键值对的次序，迭代顺序和插入顺序保持一致

#### 9.3.7 SortedMap接口和TreeMap实现类

正如Set接口派生出SortedSet子接口，Sorted接口有一个TreeSet实现类一样，Map接口也派生出一个SortedMap子接口，SortedMap接口也有一个TreeMap实现类。

TreeMap就是一个红黑树数据结构，每个键值对作为红黑树的一个节点。存储键值对时根据key对节点进行排序。可以保证所有键值对处于有序状态。  
和TreeSet一样，TreeMap也有自然排序和定制排序两种排序方式。

与TreeSet类似的是，TreeMap中提供了一系列根据key顺序访问键值对的方法：

**代码：**

public static void main(String[] args) {  
 TreeMap<String, Integer> stuTreeMap = new TreeMap<>(new Comparator<String>() {  
 @Override  
 public int compare(String o1, String o2) {  
 int num=o1.compareTo(o2);  
 return num;  
 }  
 });  
  
  
 stuTreeMap.put("LiMing",14);  
 stuTreeMap.put("LiMing",24);  
 stuTreeMap.put("Jenny",16);  
 stuTreeMap.put("Denny",24);  
  
 System.*out*.println(stuTreeMap);//{Denny=24, Jenny=16, LiMing=24}  
  
 System.*out*.println(stuTreeMap.firstEntry());//Denny=24  
  
}

结果：



**分析：**

* Object firstKey():返回该Map中的最小key值，如果Map为空则返回null；
* Object lastKey():返回该Map中的最大key值，如果Map为空则返回null；
* Map.Entry higherEntry(Object key):返回该Map中位于key后一位的键值对；
* Object higherKey(Object key):返回该Map中位于key后一位的key；
* Map.Entry lowerEntry(Object key):返回该Map中位于key前一位的键值对；
* Object lowererKey(Object key):返回该Map中位于key前一位的key；
* NavigableMap tailMap(Object fromkey,boolean fromInclusive,Object toKey,boolean toInclusive):返回该Map的子Map，其key范围从fromkey（是否包含取决于第二个参数）到toKey(是否包含取决于第四个参数)。

#### 9.3.8 weakedhashmap类

WeakHashMap与HashMap的用法基本相似，区别在于HashMap的key保留了对实际对象的强引用，这意味着只要该对象不销毁，该HashMap的所有key所引用的对象就不会被垃圾回收，HashMap也不会自动删除这些key所对应的键值对，但WeakHashMap的key只保留了对实际对象的弱引用，这意味着如果WeakHashMap对象的key所引用的对象没有被其他强引用变量所引用，则这些key所引用的对象可能被垃圾回收，WeakHashMap也可能自动删除这些key对应的键值对

#### 9.3.9 IdentityHashMap实现类

这个类的实现机制与HashMap基本相似，但它在处理两个key相等时比较独特：在IdentityHashMap中，当且仅当两个key严格相等(key1==key2)时，IdentityHashMap才认为两个key相等，对于普通的HashMap而言，只要key1和key2通过equals方法比较返回true，且它们的hashcode相等即可。

代码举例：

IdentityHashMap map = new IdentityHashMap<>();  
  
 map.put(new String("语文"), 89);  
 map.put(new String("语文"), 90);  
  
 map.put("java",70);  
 map.put("java",71);  
 System.out.println(map);//{java=71, 语文=90, 语文=89}

**分析：**

前面是两个对象虽然通过equal方法比较是相等的，但是通过==比较不相等，后面两个字符串在常量池中同一位置，所以使用==判断相等。

## 十一，输入输出流（io流）

io流用到的地方很多，比如上传下载，传输，设计模式等……在使用io流前需要了解的四点：

1）明确要操作的数据是数据源还是数据目的（也就是要读还是写）

**源：**

InputStream

Reader

**目的：**

OutputSream

Writer

2）明确要操作的设备上的数据是字节还是文本

**字节：**

InputStream

OutputStream

**字符：**

Reader

Writer

3）明确数据所在的具体设备

硬盘：File开头

内存：数组和字符串

键盘：System.in

网络：Socket

4）明确是否需要而外的功能（比如需要转化流，高效流）

**转换流**

InputStreamReader

OutputStreamWriter

**缓冲流**

Bufferedxxx

序列流

**序列流**

SequenceInputStream

**对象序列化**

ObjectInputStream

ObjectOutputStream

**打印流**

PrintStream

Printwriter

**保证字节原样性**

DataOutputStream

DataInputStream

### 11.1 File类

java.io.File 类是专门对文件进行操作的类，只能对文件本身进行操作，不能对文件内容进行操作。

Java文件类以抽象的方式代表文件名和目录路径名。该类主要用于文件和目录的创建、文件的查找和文件的删除等。

#### 11.1.1介绍两种构造器：

1. 通过将给定路径名字符串转换成抽象路径名来创建一个新 File 实例。

File(String pathname)

2. 通过给定的父抽象路径名和子路径名字符串创建一个新的File实例。

File(File parent, String child);

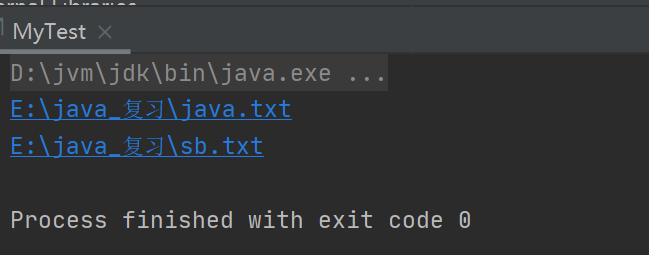
程序清单 11.1：

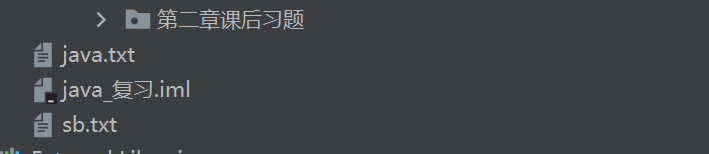
File类构造器的使用

代码：

package com.dong.day11.File;  
  
import java.io.File;  
import java.io.IOException;  
  
public class MyTest {  
  
  
 public static void main(String[] args) throws IOException {  
 File file=new File("E:/java\_复习","java.txt");  
 File file1=new File("E:/java\_复习/sb.txt");  
 file.createNewFile();  
 file1.createNewFile();  
 System.*out*.println(file.getAbsolutePath());  
 System.*out*.println(file.getAbsolutePath());  
 }  
}

结果：





**分析：**

File类new一个成员变量并不会在磁盘中建立相应的文件，要想建立文件需要用到

file.createNewFile();方法。

#### 11.1.2 File类的方法

|  |  |
| --- | --- |
| 1 | **public String getName()** 返回由此抽象路径名表示的文件或目录的名称。 |
| 2 | **public String getParent()、**  返回此抽象路径名的父路径名的路径名字符串，如果此路径名没有指定父目录，则返回 null。 |
| 3 | **public File getParentFile()** 返回此抽象路径名的父路径名的抽象路径名，如果此路径名没有指定父目录，则返回 null。 |
| 4 | **public String getPath()** 将此抽象路径名转换为一个路径名字符串。 |
| 5 | **public boolean isAbsolute()** 测试此抽象路径名是否为绝对路径名。 |
| 6 | **public String getAbsolutePath()** 返回抽象路径名的绝对路径名字符串。 |
| 7 | **public boolean canRead()** 测试应用程序是否可以读取此抽象路径名表示的文件。 |
| 8 | **public boolean canWrite()** 测试应用程序是否可以修改此抽象路径名表示的文件。 |
| 9 | **public boolean exists()** 测试此抽象路径名表示的文件或目录是否存在。 |
| 10 | **public boolean isDirectory()** 测试此抽象路径名表示的文件是否是一个目录。 |
| 11 | **public boolean isFile()** 测试此抽象路径名表示的文件是否是一个标准文件。 |
| 12 | **public long lastModified()** 返回此抽象路径名表示的文件最后一次被修改的时间。 |
| 13 | **public long length()** 返回由此抽象路径名表示的文件的长度。 |
| 14 | **public boolean createNewFile() throws IOException** 当且仅当不存在具有此抽象路径名指定的名称的文件时，原子地创建由此抽象路径名指定的一个新的空文件。 |
| 15 | **public boolean delete()**  删除此抽象路径名表示的文件或目录。 |
| 16 | **public void deleteOnExit()** 在虚拟机终止时，请求删除此抽象路径名表示的文件或目录。 |
| 17 | **public String[] list()** 返回由此抽象路径名所表示的目录中的文件和目录的名称所组成字符串数组。 |
| 18 | **public String[] list(FilenameFilter filter)** 返回由包含在目录中的文件和目录的名称所组成的字符串数组，这一目录是通过满足指定过滤器的抽象路径名来表示的。 |
| 19 | **public File[] listFiles()**   返回一个抽象路径名数组，这些路径名表示此抽象路径名所表示目录中的文件。 |
| 20 | **public File[] listFiles(FileFilter filter)** 返回表示此抽象路径名所表示目录中的文件和目录的抽象路径名数组，这些路径名满足特定过滤器。 |
| 21 | **public boolean mkdir()** 创建此抽象路径名指定的目录。 |
| 22 | **public boolean mkdirs()** 创建此抽象路径名指定的目录，包括创建必需但不存在的父目录。 |
| 23 | **public boolean renameTo(File dest)**  重新命名此抽象路径名表示的文件。 |
| 24 | **public boolean setLastModified(long time)** 设置由此抽象路径名所指定的文件或目录的最后一次修改时间。 |
| 25 | **public boolean setReadOnly()** 标记此抽象路径名指定的文件或目录，以便只可对其进行读操作。 |
| 26 | **public static File createTempFile(String prefix, String suffix, File directory) throws IOException** 在指定目录中创建一个新的空文件，使用给定的前缀和后缀字符串生成其名称。 |
| 27 | **public static File createTempFile(String prefix, String suffix) throws IOException** 在默认临时文件目录中创建一个空文件，使用给定前缀和后缀生成其名称。 |
| 28 | **public int compareTo(File pathname)** 按字母顺序比较两个抽象路径名。 |
| 29 | **public int compareTo(Object o)** 按字母顺序比较抽象路径名与给定对象。 |
| 30 | **public boolean equals(Object obj)** 测试此抽象路径名与给定对象是否相等。 |
| 31 | **public String toString()**  返回此抽象路径名的路径名字符串。 |

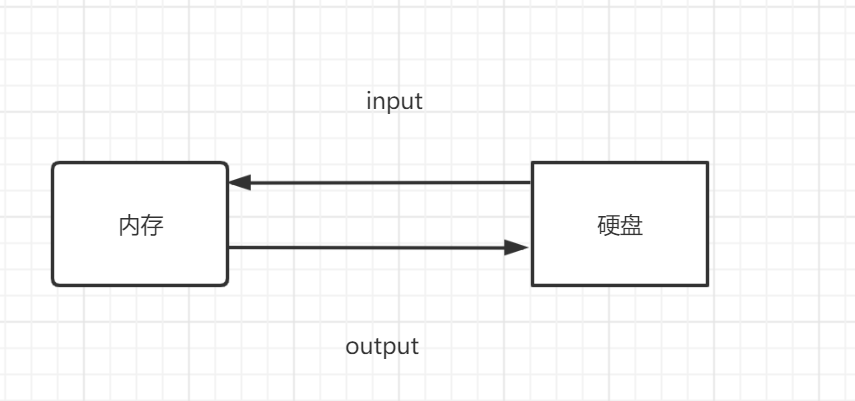
### 11.2 初探io流

我想在座各位肯定经历都过这样的场景。当你编辑一个文本文件也好用eclipse打代码也罢，忘记了ctrl+s ，在你关闭文件的哪一瞬间手残点了个不该点的按钮，但你反应过来，心早已拔凉拔凉的了。

我们把这种数据的传输，可以看做是一种数据的流动，按照流动的方向，以内存为基准，分为输入input 和输出output ，即流向内存是输入流，流出内存的输出流。

Java中I/O操作主要是指使用java.io包下的内容，进行输入、输出操作。**输入也叫做读取数据，输出也叫做作写出数据**。

图解：



#### 11.2.1 io流分类

根据数据的流向分为：**输入流** 和 **输出流**。

* **输入流** ：把数据从其他设备上读取到内存中的流。
* **输出流** ：把数据从内存 中写出到其他设备上的流。

根据数据的类型分为：**字节流** 和 **字符流**。

* **字节流** ：以字节为单位，读写数据的流。
* **字符流** ：以字符为单位，读写数据的流。

**注意：**

我们必须明确一点的是，一切文件数据(文本、图片、视频等)在存储时，都是以二进制数字的形式保存，都一个一个的字节，那么传输时一样如此。所以，字节流可以传输任意文件数据。在操作流的时候，我们要时刻明确，无论使用什么样的流对象，底层传输的始终为二进制数据。

#### 11.2.2 字节输出流（OutputStream）

java.io.OutputStream抽象类是表示**字节输出流**的所有类的**超类**（父类），将指定的字节信息写出到目的地。它定义了字节输出流的基本共性功能方法。

**OutputStream的基础方法：**

1、 public void close() ：关闭此输出流并释放与此流相关联的任何系统资源。

2、 public void flush() ：刷新此输出流并强制任何缓冲的输出字节被写出。

3、 public void write(byte[] b)：将 b.length个字节从指定的字节数组写入此输出流。

4、 public void write(byte[] b, int off, int len) ：从指定的字节数组写入 len字节，从偏移量 off开始输出到此输出流。 **也就是说从off个字节数开始读取一直到len个字节结束**

5、 public abstract void write(int b) ：将指定的字节输出流。

**FileOutputStream类：**

FileOutputStream是OutputStream的子类

FileOutoutStream类的构造方法：

1、 public FileOutputStream(File file)：根据File对象为参数创建对象。

2、 public FileOutputStream(String name)： 根据名称字符串为参数创建对象。

推荐第二种。

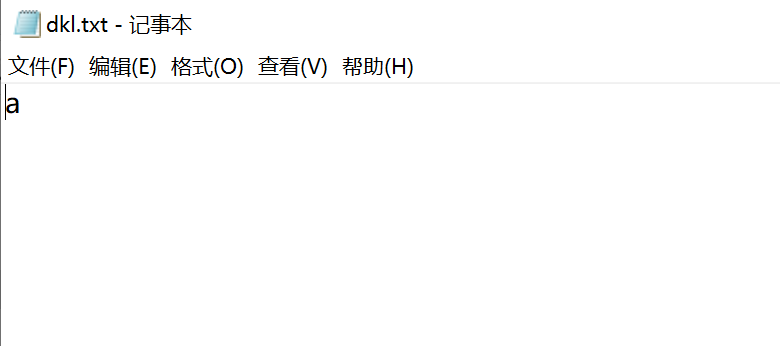
**程序清单11.2：**

**构建一个输出流**

**代码：**

package com.dong.day11.outputStream;  
  
import java.io.\*;  
  
public class MyTest {  
 public static void main(String[] args) throws IOException {  
  
 FileOutputStream outputStream=new FileOutputStream("dkl.txt");  
 outputStream.write('a'); //如果写入数字会出现乱码的问题  
 outputStream.close();  
  
 }  
}

结果：



**FileOutputStream实现追加写的效果：**

经过以上的代码测试，每次程序运行，每次创建输出流对象，都会清空目标文件中的数据。如何保留目标文件中数据，还能继续**追加新数据**呢？并且实现**换行**呢？其实很简单，这个时候我们又要再学习FileOutputStream的另外两个构造方法了，如下：

1、public FileOutputStream(File file, boolean append)

2、public FileOutputStream(String name, boolean append)

这两个构造方法，第二个参数中都需要传入一个boolean类型的值，true 表示追加数据，false 表示不追加也就是清空原有数据。这样创建的输出流对象，就可以指定是否追加续写了，至于Windows换行则是 \n\r

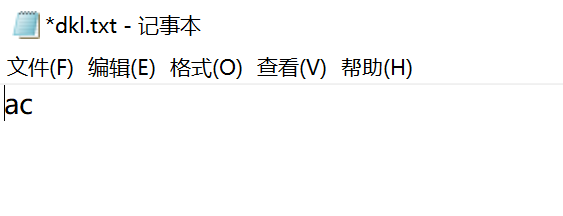
程序清单 11.3：

要求：实现追加写

代码：

package com.dong.day11.outputStream;  
  
import java.io.\*;  
  
public class MyTest {  
 public static void main(String[] args) throws IOException {  
  
 FileOutputStream outputStream=new FileOutputStream("dkl.txt",true);  
 outputStream.write('a');  
 outputStream.write('c');  
 outputStream.close();  
  
 }  
}

结果：



#### 11.2.3 字节输入流（InputStream）

java.io.InputStream抽象类是表示**字节输入流**的所有类的**超类**（父类），可以读取字节信息到内存中。它定义了字节输入流的基本共性功能方法。

**字节输入流的基本共性功能方法**:：

1、 public void close() ：关闭此输入流并释放与此流相关联的任何系统资源。

2、public abstract int read()： 从输入流读取数据的下一个字节。

3、 public int read(byte[] b)： 该方法返回的int值代表的是读取了多少个字节，读到几个返回几个，读取不到返回-1

**FileIntputStream类**

**构造方法：**

1、 FileInputStream(File file)： 通过打开与实际文件的连接来创建一个 FileInputStream ，该文件由文件系统中的 File对象 file命名。

2、FileInputStream(String name)： 通过打开与实际文件的连接来创建一个 FileInputStream该文件由文件系统中的路径名name命名。

推荐第二种。

**读取方法：**

1，**读取字节**

read方法，每次可以读取一个字节的数据，提升为int类型，读取到文件末尾，返回-1

代码：int len=read();

2，**使用字节数组读取**

read(byte[] b)，每次读取b的长度个字节到数组中，返回读取到的有效字节个数，读取到末尾时，返回-1

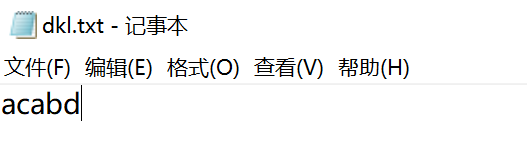
程序清单9.4：

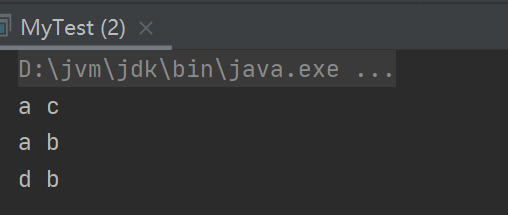
要求:采用字节数组读取dkl.txt中的数据并输出到控制台。

代码：

package com.dong.day11.InputStream;  
  
import jdk.internal.util.xml.impl.Input;  
  
import java.io.\*;  
  
public class MyTest {  
 public static void main(String[] args) throws IOException {  
 FileInputStream fis = null;  
 try {  
 fis=new FileInputStream("dkl.txt");  
 byte[]b=new byte[2];  
 while(fis.read(b)!=-1)  
 {  
 System.*out*.println((char)b[0]+" "+(char)b[1]);  
 }  
  
 }catch (Exception e)  
 {  
 e.printStackTrace();  
 }finally {  
 fis.close();  
 }  
 }  
}

结果：





#### 11.2.4 字符输入流（Reader）

字符流的由来：因为数据编码的不同，因而有了对字符进行高效操作的流对象，字符流本质其实就是基于字节流读取时，去查了指定的码表，而字节流直接读取数据会有乱码的问题。

**字符输入流的共性方法**：

1、public void close() ：关闭此流并释放与此流相关联的任何系统资源。

2、 public int read()： 从输入流读取一个字符。

3、 public int read(char[] cbuf)： 从输入流中读取一些字符，并将它们存储到字符数组 cbuf中.

**FileReader类是Reader的子类。**

#### 11.2.4 字符输出流（Writer）

**共性方法：**

1、 public void close() ：关闭此输出流并释放与此流相关联的任何系统资源。

2、 public void flush() ：刷新此输出流并强制任何缓冲的输出字节被写出。

3、 public void write(byte[] b)：将 b.length个字节从指定的字节数组写入此输出流。

4、 public void write(byte[] b, int off, int len) ：从指定的字节数组写入 len字节，从偏移量

**重点了解FileWriter类。**

**下面来看一个字符输入输出的例子：**

**程序清11.5**

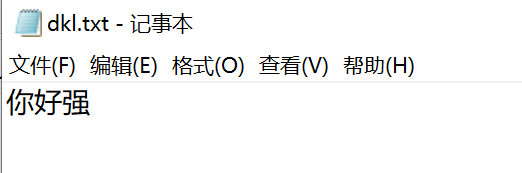
**要求：**

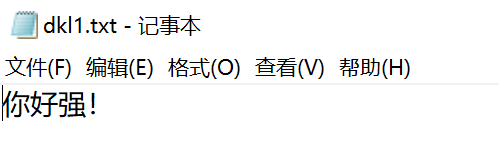
**先输入一个文本dkl.txt,写入“你好强”，在用Writer读取到控制台。**

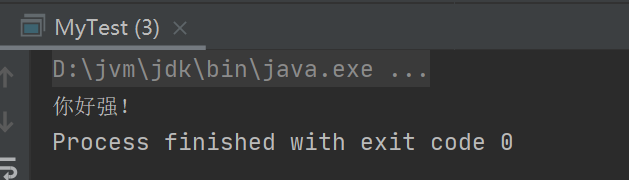
**代码：**

**package com.dong.day11.Writer;  
  
import java.io.\*;  
  
public class MyTest {  
 public static void main(String[] args) {  
 Writer writer = null;  
 Reader reader=null;  
 try {  
 writer=new FileWriter("dkl.txt");  
 reader=new FileReader("dkl1.txt");  
 writer.write("你好强");  
 int c;  
 while((c=reader.read())!=-1)  
 {  
 System.*out*.print((char)c);  
 }  
  
 } catch (IOException e) {  
 e.printStackTrace();  
 }finally {  
 try {  
 writer.close();  
 reader.close();  
 }catch (IOException e)  
 {  
 e.printStackTrace();  
 }  
 }  
  
 }  
}**

**结果：**







**注意：io异常处理**

我们在学习的过程中可能习惯把异常抛出，而实际开发中并不能这样处理，建议使用try...catch...finally 代码块

#### 11.2.5 缓冲流

首先我们来认识认识一下缓冲流,也叫高效流，是对4个FileXxx 流的“增强流”。

**缓冲流的原理：**

1、使用了底层流对象从具体设备上获取数据，并将数据存储到缓冲区的数组内。

2、通过缓冲区的read()方法从缓冲区获取具体的字符数据，这样就提高了效率。

3、如果用read方法读取字符数据，并存储到另一个容器中，直到读取到了换行符时，将另一个容器临时存储的数据转成字符串返回，就形成了readLine()功能。

也就是说在创建流对象时，会创建一个内置的默认大小的缓冲区数组，通过缓冲区读写，减少系统IO次数，从而提高读写的效率。

缓冲书写格式为BufferedXxx

**构造方法：**

字节缓冲流

* public BufferedInputStream(InputStream in) ：创建一个新的缓冲输入流，注意参数类型为**InputStream**。
* public BufferedOutputStream(OutputStream out)： 创建一个新的缓冲输出流，注意参数类型为**OutputStream**。

字符缓冲流

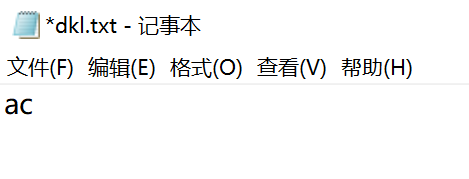
* public BufferedReader(Reader in) ：创建一个新的缓冲输入流，注意参数类型为**Reader**。
* public BufferedWriter(Writer out)： 创建一个新的缓冲输出流，注意参数类型为**Writer**。

**程序清单：11.6**

**代码：**

package com.dong.day11.outputStream;  
  
import java.io.\*;  
  
public class MyTest {  
 public static void main(String[] args) throws IOException {  
  
 FileOutputStream outputStream=new FileOutputStream("dkl.txt",true);  
 BufferedOutputStream bufferedInputStream=new BufferedOutputStream(outputStream);  
 outputStream.write('a');  
 outputStream.write('c');  
 outputStream.close();  
  
 }  
}

**结果：**



#### 11.2.5 转化流

**字符的编码和解码：**

众所周知，计算机中储存的信息都是用二进制数表示的，而我们在屏幕上看到的数字、英文、标点符号、汉字等字符是二进制数转换之后的结果。按照某种规则，将字符存储到计算机中，称为**编码** 。反之，将存储在计算机中的二进制数按照某种规则解析显示出来，称为**解码**。比如说，按照A规则存储，同样按照A规则解析，那么就能显示正确的文本符号。反之，按照A规则存储，再按照B规则解析，就会导致乱码现象。

了解两个类：

1，InputStreamReader类

InputStreamReader，是Reader的子类，从字面意思可以看出它是从字节流到字符流的桥梁。

**构造方法：**

InputStreamReader(InputStream in): 创建一个使用默认字符集的字符流。

InputStreamReader(InputStream in, String charsetName): 创建一个指定字符集的字符流。

**2，OutputStreamWriter类**

和InputStreamReader类形同。

#### 11.2.6 序列化流

(1)可以把对象写入文本文件或者在网络中传输

(2)如何实现序列化呢?

让被序列化的对象所属类实现序列化接口。

该接口是一个标记接口。没有功能需要实现。

(3)注意问题：

把数据写到文件后，在去修改类会产生一个问题。

如何解决该问题呢?

在类文件中，给出一个固定的序列化id值。

而且，这样也可以解决黄色警告线问题

(4)面试题：

什么时候序列化?

如何实现序列化?

什么是反序列化?

**1，什么是序列化？**

Java 提供了一种对象**序列化**的机制。用一个字节序列可以表示一个对象，该字节序列包含该对象的数据、对象的类型和对象中存储的属性等信息。字节序列写出到文件之后，相当于文件中**持久保存**了一个对象的信息。

**反之，该字节序列还可以从文件中读取回来，重构对象，对它进行反序列化。对象的数据、对象的类型和对象中存储的数据信息，都可以用来在内存中创建对象**

**2，ObjectOutputStream类**

java.io.ObjectOutputStream 类，将Java对象的原始数据类型写出到文件,实现对象的持久存储。

**构造方法：**

public ObjectOutputStream(OutputStream out)： 创建一个指定OutputStream的ObjectOutputStream。//序列

**一个对象要序列化的条件：**

1.该类必须实现java.io.Serializable 接口，Serializable 是一个标记接口，不实现此接口的类将不会使任何状态序列化或反序列化，会抛出NotSerializableException 。

2.该类的所有属性必须是可序列化的。如果有一个属性不需要可序列化的，则该属性必须注明是瞬态的，使用transient 关键字修饰。

**序列化的方法：**

public final void writeObject (Object obj) : 将指定的对象写出

**3.ObjectInputStream类**

ObjectInputStream反序列化流，将之前使用ObjectOutputStream序列化的原始数据恢复为对象。

**构造方法：**

public ObjectInputStream(InputStream in)：

// 创建一个指定InputStream的ObjectInputStream。

**反序列化的方法：**

public final Object readObject () : 读取一个对象

注意：对于JVM可以反序列化对象，它必须是能够找到class文件的类。如果找不到该类的class文件，则抛出一个 ClassNotFoundException 异常。

#### 11.2.7 打印流

**什么是打印流？**

平时我们在控制台打印输出，是调用print方法和println方法完成的，各位用了这么久的输出语句肯定没想过这两个方法都来自于java.io.PrintStream类吧，哈哈。该类能够方便地打印各种数据类型的值，是一种便捷的输出方式。

打印流分类：

PrintWriter //字符打印流

PrintStream//字节打印流

特点：

**A:只操作目的地,不操作数据源**

**B:可以操作任意类型的数据**

**C:如果启用了自动刷新，在调用println()方法的时候，能够换行并刷新**

**D:可以直接操作文件**

## 十二，网络编程

网络编程是指编写运行在多个设备（计算机）的程序，这些设备都通过网络连接起来。

### 12.1 java.net包

java.net 包中提供了两种常见的网络协议的支持：

* **TCP**：TCP 是传输控制协议的缩写，它保障了**两个应用程序之间**的可靠通信。通常用于互联网协议，被称 TCP / IP。
* **UDP**：UDP 是**用户数据**报协议的缩写，一个无连接的协议。提供了应用程序之间要发送的数据的数据包。

### 12.2 网络编程入门

#### 12.2.1 网络通信协议

网络通信协议：通信协议是对计算机必须遵守的**规则**，只有遵守这些规则，计算机之间才能进行**通信**。这就 好比在道路中行驶的汽车一定要遵守交通规则一样，协议中对数据的传输格式、传输速率、传输步骤等做了统一规定，通信双方必须同时遵守，最终完成数据交换。

**Tcp/ip协议**

传输控制协议/因特网互联协议( Transmission Control Protocol/Internet Protocol)，是 Internet最基本、最广泛的协议。它定义了计算机如何连入因特网，以及数据如何在它们之间传输的标准。它 的内部包含一系列的用于处理数据通信的协议，并采用了4层的分层模型，每一层都呼叫它的下一层所提供的 协议来完成自己的需求。

#### 12.2.2 软件结构

C/S结构 ：全称为Client/Server结构，是指客户端和服务器结构。常见程序有ＱＱ、迅雷等软件

B/S结构 ：全称为Browser/Server结构，是指浏览器和服务器结构。常见浏览器有谷歌、火狐等。

#### 12.2.3 协议分类

**TCP协议**

传输控制协议 (Transmission Control Protocol)。TCP协议是面向连接的通信协议，即传输数据之前， 在发送端和接收端建立逻辑连接，然后再传输数据，它提供了两台计算机之间可靠无差错的数据传输。

**三次握手定理**

第一次握手，客户端向服务器端发出连接请求，等待服务器确认。

第二次握手，服务器端向客户端回送一个响应，通知客户端收到了连接请求。

第三次握手，客户端再次向服务器端发送确认信息，确认连接

**UDP协议**

用户数据报协议(User Datagram Protocol)。UDP协议是一个面向无连接的协议。传输数据时，不需 要建立连接，不管对方端服务是否启动，直接将数据、数据源和目的地都封装在数据包中，直接发送。每个 数据包的大小限制在64k以内。它是不可靠协议，因为无连接，所以传输速度快，但是容易丢失数据。日常应 用中,例如视频会议、QQ聊天等

#### 12.2.4 网络编程三要素

**协议**

计算机网络通信必须遵守的规则

**IP地址**

指互联网协议地址（Internet Protocol Address），俗称IP。IP地址用来给一个网络中的计算机设 备做唯一的编号。假如我们把“个人电脑”比作“一台电话”的话，那么“IP地址”就相当于“电话号码”。

**端口号：**

如果说IP地址可以唯一标识网络中的设备，那么端口号就可以唯一标识设备中的进程（应用程序）了。

端口号：用两个字节表示的整数，它的取值范围是0~65535。其中，0~1023之间的端口号用于一些知名的网 络服务和应用，普通的应用程序需要使用1024以上的端口号。如果端口号被另外一个服务或应用所占用，会 导致当前程序启动失败。

#### 12.2.5 基于Socket的TCP通信程序

TCP通信能实现两台计算机之间的数据交互，通信的两端，要严格区分为客户端（Client）与服务端（Server）。

两端通信时步骤：

1. 服务端程序，需要事先启动，等待客户端的连接。

2. 客户端主动连接服务器端，连接成功才能通信。服务端不可以主动连接客户端。

**ServerSocket类**

ServerSocket 类：这个类实现了服务器套接字，该对象等待通过网络的请求

**构造方法：**

public ServerSocket(int port) ：使用该构造方法在创建ServerSocket对象时，就可以将其绑定到一个指定的端口号上，参数port就是端口号。

**成员方法：**

public Socket accept() ：侦听并接受连接，返回一个新的Socket对象，用于和客户端实现通信。该方法 会一直阻塞直到建立连接。

ServerSocket工作的四大流程：

1. **调用 ServerSocket(int port)** ：创建一个服务器端套接字，并绑定到指定端口 上。用于监听客户端的请求。
2. **调用 accept()**：监听连接请求，如果客户端请求连接，则接受连接，返回通信 套接字对象。
3. **调用 该Socket类对象的 getOutputStream() 和 getInputStream ()**：获取输出 流和输入流，开始网络数据的发送和接收。
4. **关闭ServerSocket和Socket对象**：客户端访问结束，关闭通信套接字。

程序清单：

ServerSocket ss = new ServerSocket(9999); // 1、创建服务端的ServerSocket，指明自己的端口号

Socket s = ss.accept (); // 2、调用accept()监听来自客户端的连接

InputStream in = s.getInputStream(); // 3、获取输入流，读取输入流的数据

byte[] buf = new byte[1024];

int num = in.read(buf);

String str = new String(buf,0,num); //???

System.out.println(s.getInetAddress().toString()+":"+str); //???

s.close(); // 4、回收资源

ss.close();

**Socket类（客户端类）**

该类实现客户端套接字，套接字指的是两台设备之间通讯的端点。

构造方法：

public Socket(String host, int port)

host是服务器IP地址，port是端口

成员方法：

public InputStream getInputStream()：返回此套接字的输入流

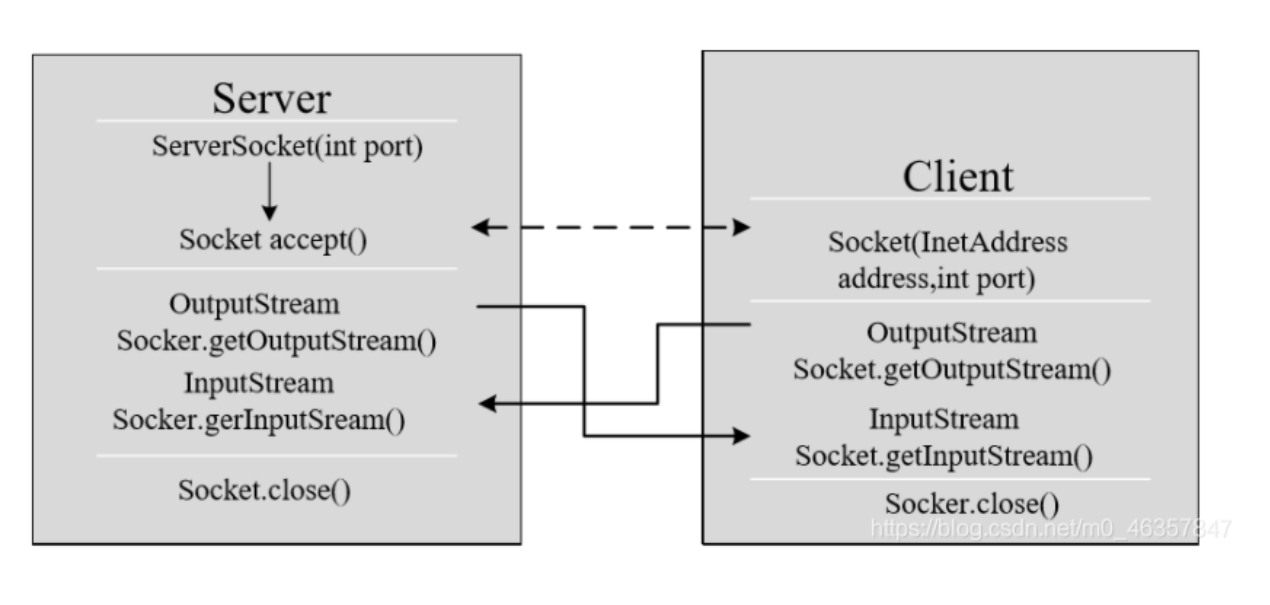
public OutputStream getOutputStream() ： 返回此套接字的输出流

public void close() ：关闭此套接字。

public void shutdownOutput() ： 禁用此套接字的输出流。

工作的四大流程：

1. 创建 Socket：根据指定服务端的 IP 地址或端口号构造 Socket 类对象。若服务器端响应，则建立客户端到服务端的通信路线。若连接失败，则会出现异常。
2. 打开连接到 Socket 的输入/出流： 使用 getInputStream()方法获得输入流，使用 getOutputStream()方法获得输出流，进行数据传输
3. 按照一定的协议对 Socket 进行读/写操作：通过输入流读取服务器放入线路的信息(但不能读取自己放入路线的信息)，通过输出流将信息写入线程
4. 关闭 Socket：断开客户端到服务器的连接，释放线路



程序清单：

Socket类的

代码：

// 构造器一

Socket(String host,int port)throws UnknownHostException,IOException

/\* 向服务器(域名是 host。端口号为port)发起TCP连接，若成功，则创建Socket对象，否则抛出异常。\*/

// 构造器二

Socket(InetAddress address,int port)throws IOException

/\* 根据InetAddress对象所表示的 IP地址以及端口号port发起连接。\*/

四大步骤：

Socket s = new Socket("192.168.40.165",9999);

// 1、创建Socket对象，指明服务端的IP和端口号

OutputStream out = s.getOutputStream(); // 2、获取一个输出流，用于输出数据

out.write("hello".getBytes()); // 3、写出数据

s.close(); // 4、回收资源

#### 12.2.6 简单的TCP程序

public class SocketTest {

/\* 客户端 \*/

@Test

public void client(){

OutputStream output = null;

Socket socket = null;

try {

InetAddress localHost = InetAddress.getByName("127.0.0.1");

socket = new Socket(localHost,8848);

output = socket.getOutputStream();

output.write("hello I'm the client".getBytes());

} catch (IOException e) {

e.printStackTrace();

} finally {

if(output != null){

try {

output.close();

} catch (IOException e) {

e.printStackTrace();

}

}

if(socket != null){

try {

socket.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

/\* 服务端 \*/

@Test

public void server() {

ServerSocket serverSocket = null;

Socket socket = null;

InputStream input = null;

ByteArrayOutputStream out = null;

try {

serverSocket = new ServerSocket(8848);

socket = serverSocket.accept();

System.out.println("client IP: " + **socket.getInetAddress()**);

input = socket.getInputStream();

/\*

一般不建议这样书写，数据传输时可能会出现乱码！！

byte[] buffer = new byte[1024];

int len;

while((len = input.read(buffer)) != -1){

String data = new String(buffer,0,len);

System.out.println(data);

}\*/

out = new ByteArrayOutputStream();

byte[] buffer = new byte[10];

int len;

while((len = input.read(buffer)) != -1){

out.write(buffer,0,len);

}

System.out.println(out.toString());

} catch (IOException e) {

e.printStackTrace();

} finally {

if(out != null){

try {

out.close();

} catch (IOException e) {

e.printStackTrace();

}

}

if(input != null){

try {

input.close();

} catch (IOException e) {

e.printStackTrace();

}

}

if(socket != null){

try {

socket.close();

} catch (IOException e) {

e.printStackTrace();

}

}

if(serverSocket != null){

try {

serverSocket.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

}