# 实战 Java 程序设计

北京尚学堂科技有限公司　编著

清華大學出版社
北京

## 内 容 简 介

这是一本既注重实战，同时也注重底层“内功”（内存分析、JVM底层、数据结构）训练的书，本书能帮助初学者打通Java编程“任督二脉”。本书集作者11年Java教学之精华，既适合初学者入门，也适合已经工作的开发者复习。

全书共分18章，内容涵盖Java开发所需的相关内容及339个案例（很多案例对于工作人员也有很大的参考价值）。书中秉承尚学堂实战化教学理念，从第一章开始介入实战项目，寓教于乐，读者可迅速进入开发者的角色。

本书适合初学者入门，也适合高等院校相关专业作为教材使用，还可作为Java程序员的参考用书。



**图书在版编目(CIP)数据**

实战Java程序设计 / 北京尚学堂科技有限公司编著. — 北京：清华大学出版社，2018
ISBN 978-7-302-48498-1

Ⅰ. ①实… Ⅱ. ①北… Ⅲ. ①Java语言—程序设计 Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字(2017)第 227453 号

**责任编辑：**杨如林
**封面设计：**杨玉兰
**版式设计：**方加青
**责任校对：**徐俊伟
**责任印制：**李红英

**出版发行：**清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
**印 装 者：**三河市君旺印务有限公司
**经　　销：**全国新华书店
**开　　本：**188mm×260mm　　**印　　张：**29.5　　**字　　数：**660 千字
**版　　次：**2018 年 6 月第 1 版　　**印　　次：**2018 年 6 月第 1 次印刷
**印　　数：**1 ～ 2000
**定　　价：**89.00 元

产品编号：076926-01

# 前　言

## 本书的特色

Java语言问世20年了，一直是世界第一编程语言，被誉为计算机界的“英语”。北京尚学堂科技11年来一直从事Java语言的相关培训教学，并且同国内外上千家企业有直接的用人合作。我们深知学员的需求是什么，企业的技术要求是什么。

**企业要求：程序员既要有实战技能，可以快速上手，同时又要拥有扎实的内功，熟悉底层原理，后劲十足。因此，在笔试和面试考查的时候也是结合“底层原理、数据结构、实战应用、设计思维”四个方面进行的。**针对这四方面的需求，我们编写了本书，这也是本书的四大特点。

**第一大特点：注重实战应用。**精心设计的案例对于工作多年的读者也有参考价值；在本书第一章就引入了游戏项目案例，让大家从一开始就能体验“编程之美”与“编程之乐”。读者在学习的第一天就可以“炫耀”一下，使自信翻倍。

**第二大特点：底层原理讲解丰富。**对于面向对象核心内容的讲解，本书深入到内存分析，让读者对于对象底层有形象的认识；对于内存管理的知识，本书也深入到JVM底层设计进行讲解。通过这些讲解，让读者不仅可以理解底层核心技术，而且对于日后的笔试与面试做到胸有成竹，为以后的工作打下更加扎实的基础。

**第三大特点：结合实战讲解数据结构和JDK源码。**本书对大多数类的讲解都深入到JDK源码，带大家学习真正高手的写法；容器一章更是深入讲解了数据结构和源码，并做到深入浅出，帮助大家修炼深厚的“编程内功”。

**第四大特点：植入设计者思维。**如果本书的读者是初学者，我们要让初学者从知其然进化到知其所以然。本书引入了设计模式与多线程架构的讲解，初学者可以通过这些内容从一开始就培养设计的思维与架构的思维，为以后的发展铺设好“高速公路”。

## 如何学习本书

本书共分18章，这里对各章节做简要说明，以方便读者概览全书。

**第1章** 讲解Java的入门知识，配置开发环境，开发第一个Java程序，开始使用Eclipse，开发自己的第一个游戏项目。

**第2章** 讲解数据类型、运算符和变量， 这是编程的基础，是程序的“砖块”。

**第3章** 讲解控制语句：条件判断结构、循环结构。控制语句是编程的基础，是程序的“混凝土”。本章是进入编程世界的门槛，需要进行大量练习。

**第4章** 讲解Java面向对象编程基础知识。本章通过类、对象、包等基本概念以及内存分析、JVM内存管理的讲解，让大家一开始就深入底层，更深刻地了解对象概念。

**第5章** 讲解Java面向对象编程的进阶知识，主要包含继承、封装、多态三大特征以及接口、抽象类、内部类等概念。

**第6章** 讲解异常机制。本章通过导引问题让大家知道为什么需要异常机制，处理异常的多种手段，以及开发中常见异常的应对方式。

**第7章** 数组。本章从底层讲解数组的本质、数组的常见使用方式，通过排序和搜索算法，既可练习数组的用法，也可学习算法知识，为应对企业笔试和面试做好准备。

**第8章** 常用类。本章讲解多种常用类的用法：包装类、字符串相关类、时间处理相关类、Math类、File类等。在讲解过程中，结合JDK源码，让大家更深刻地理解用法。

**第9章** 容器。本章讲解各种容器的用法：List、Map、Set。本章还引入数据结构的相关内容，通过源码分析让大家既学习了容器，又学习了数据结构的知识。练好了内功，应对企业面试绰绰有余。

**第10章** 输入与输出技术。本章配置了各种在工作中有参考价值的实用案例，并且讲解了在工作中常用的Apache Commons I/O工具库，还通过引入设计模式概念，让大家明白整个I/O流体系架构。

**第11章** 多线程技术。本章深入讲解了多线程的基本用法、生命周期与状态转化的知识，同时对同步机制做了深入讲解，还引入生产者与消费者模式，让大家具备架构设计的思维。此外还额外加入了定时机制与任务调度的内容。

**第12章** 网络编程。本章给出了实用价值极高的案例，让大家通过案例的学习，举一反三，就可以完成TCP、UDP的各种应用编程。

**第13章** J20飞机游戏项目。本章通过手把手教学，用一个游戏项目将前面1～12章的知识全部做了串联，寓教于乐，让大家了解项目开发的全流程。

**第14～16章** 讲解基本的Swing知识与事件模型。需要强调的是，Swing在工作中极少用到，但为了知识的完整性，这里只进行简单讲解，不作为重点。

**第17章** 反射机制。反射是Java的高级特性，在工作和学习中得到了广泛应用，掌握反射的本质及应用，非常有必要。

**第18章** 核心设计模式。GoF 23设计模式的内容庞杂，这里只选取在工作和学习中最

重要的几个设计模式进行了深入讲解，让大家从一开始就具备设计的思维。同时，这也是面试中常涉及的内容，掌握设计模式可以为你加分不少。

## 本书配套资源

**1. 视频资源库（1000G视频资源）**

读者在学习本书的过程中，可以结合附赠的《Java 300集》大型教学视频进行学习，能更好地理解本书内容，拓展Java编程视野。

**注：**《Java 300集》大型教学视频已经被北京大学计算机系教授推荐为必看视频教程。

读者可以访问网址http://www.bjsxt.com/download.html观看视频。此外，该网站还提供了很多由尚学堂老师录制的课堂教学视频，累计达1000G的视频资源，涵盖了IT行业的方方面面，内容原汁原味，课堂气氛活跃，免费共享给读者学习使用。

**2. PPT和题库（高校老师专职助手）**

为了便于院校老师使用本书，专门开放了我们现有的PPT和题库，并且可以针对不用院校的需求进行适当调整，我们还为每一位院校老师提供专职助手，有针对性地调整教学内容及考试题库，需要的老师请联系：www.bjsxt.com。

**3. 开发者常用英文词汇表（1800个开发词汇）**

由尚学堂学员孙波（现已工作）在老师指导下完成。

词汇表涵盖了日常开发中的常见词汇，通晓这些词汇，可以让你游刃有余地阅读英文文档，完成高手进阶的必由之路。

下载地址：http://www.bjsxt.com/download.html。

**4. 实例源码**

本书各章实例源码可按以下方法免费下载：

登录清华大学出版社网站（www.tup.com.cn），搜索本书后在“资源管理”中下载本书相关源码。

## 鸣　谢

本书由北京尚学堂科技教研部编写，其中主要编写者为高淇，参与本书编写工作的还有刘凯力、王焕等。

本书在出版过程中，得到了清华大学出版社栾大成、杨如林老师的大力支持，在此表示衷心的感谢。另外，本书的所有编审、发行人员为本书的出版发行付出了辛勤的劳动，在此一并致以诚挚的谢意。

我们以科学、严谨的态度，力求精益求精，但错误之处在所难免，敬请广大读者批评指正，我们将不胜感激。

教研部出版组邮箱：book@sxt.cn；高淇老师邮箱：gaoqi@sxt.cn。

作　者

实战 Java
程序设计

# 目　录

实战 Java
程序设计

# 案例目录

## 第1章　Java入门



## 第2章　数据类型和运算符



## 第3章　控制语句

## 第4章　Java面向对象编程基础



## 第5章　Java面向对象编程进阶

## 第6章 异常机制



## 第7章 数组



## 第8章 常用类

## 第9章 容器



## 第10章 输入与输出技术



## 第11章 多线程技术

## 第12章　网络编程



## 第13章　J20飞机游戏项目



## 第14章　GUI编程——Swing基础

## 第15章 事件模型



## 第16章 Swing中的其他控件



## 第17章 反射机制



## 第18章 核心设计模式

# 第3章 控制语句

从本章开始学习流程控制语句。流程控制语句是用来控制程序中各语句执行顺序的语句，可以把语句组合成能完成一定功能的小逻辑模块。程序的结构可分为三类：顺序、选择和循环。

“顺序结构”代表“先执行a，再执行b”的逻辑。例如，先找个女朋友，再给女朋友打电话；先订婚，再结婚等。

“条件判断结构”代表“如果……，则……”的逻辑。例如，如果女朋友来电，则迅速接电话；如果看到红灯，则停车。

“循环结构”代表“如果……，则再继续……”的逻辑。例如，如果没打通女朋友电话，则继续打一次； 如果没找到喜欢的人，则继续找。

前面两章讲解的程序都是顺序结构，即按照书写顺序执行每一条语句，本章研究的重点是“条件判断结构”和“循环结构”。

用这三种程序结构就能表示所有的事情，大家可以试试拆分你遇到的各种事情。实际上，任何软件和程序，小到一个练习，大到一个操作系统，本质上都是由“变量、选择语句、循环语句”组成的。

这三种基本逻辑结构是相互支撑的，它们共同构成了算法的基本结构。无论怎样复杂的逻辑结构，都可以通过它们来表达。上述三种结构组成的程序可以解决全部的问题，所以任何一种高级语言都具备上述三种结构。

本章内容是大家真正跨入编程界 “门槛”的知识，是成为“程序猿”的必由之路。 本章后面会附加大量的练习，供大家自我提升。

## 3.1 条件判断结构

在人们还不知道Java选择结构的时候，编写的程序总是从程序入口开始，顺序执行每一条语句，直到执行完最后一条语句。但是生活中经常需要进行条件判断，根据判断结果决

定是否做一件事情，这时就需要用到条件判断结构。

条件判断结构用于判断给定的条件，然后根据判断的结果来控制程序的流程。主要的条件判断结构有if结构和switch结构。if结构又可以分为if单分支结构、if-else双分支结构、if-else if-else多分支结构。

### 3.1.1 if单分支结构

if单分支语法结构如下：

```
if(布尔表达式){
    语句块
}
```

if语句对布尔表达式的值进行一次判定，若判定其值为真，则执行{}中的语句块，否则跳过该语句块，其流程图如图3-1所示。

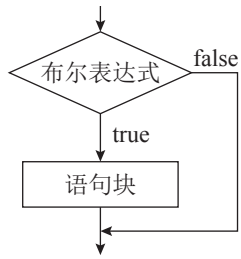



图3-1 if单分支结构流程图

【示例3-1】if单分支结构

```
public class Test1 {
  public static void main(String[ ] args) {
    //通过掷三个骰子看看今天的手气如何?
    int i = (int)(6 * Math.random()) + 1;//通过Math.random()产生随机数
    int j = (int)(6 * Math.random()) + 1;
    int k = (int)(6 * Math.random()) + 1;
    int count = i + j + k;
    //如果三个骰子之和大于15,则手气不错
    if(count > 15) {
      System.out.println("今天手气不错");
    }
    //如果三个骰子之和在10到15之间,则手气一般
    if(count >= 10 && count <= 15) { //错误写法:10<=count<=15
      System.out.println("今天手气很一般");
    }
    //如果三个骰子之和小于10,则手气不怎么样
    if(count < 10) {
      System.out.println("今天手气不怎么样");
    }
    System.out.println("得了" + count + "分");
  }
}
```

执行结果如图3-2所示。

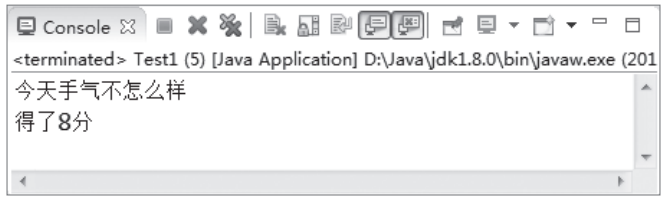



图3-2　示例3-1运行结果

**Math类的使用**

- java.lang包中的Math类提供了一些用于数学计算的方法。
- Math.random()方法用于产生一个0到1区间的double类型的随机数，但是不包括1。

```
int i = (int) (6 * Math.random()); //产生:[0,5]之间的随机整数
```

**菜鸟雷区**

- 如果if语句不写{}，则只能作用于后面的第一条语句。
- 强烈建议，任何时候都要写上{}，即使里面只有一条语句！

## 3.1.2　if-else双分支结构

if-else的语法结构如下：

```
if(布尔表达式){
    语句块1
}else{
    语句块2
}
```

当布尔表达式的值为真时，执行语句块1；否则，执行语句块2，也就是else部分。其流程图如图3-3所示。

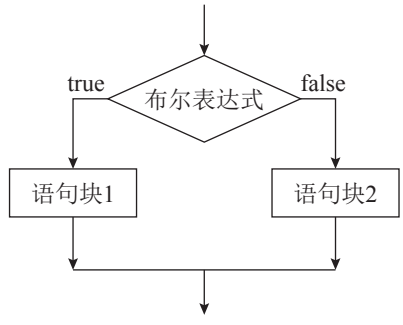



图3-3　if-else双分支结构流程图

**【示例3-2】if-else双分支结构**

```
public class Test2 {
  public static void main(String[ ] args) {
```

```
    //随机产生一个[0.0, 4.0)区间的半径,并根据半径求圆的面积和周长
    double r = 4 * Math.random();
    //Math.pow(r, 2)求半径r的平方
    double area = Math.PI * Math.pow(r, 2);
    double circle = 2 * Math.PI * r;
    System.out.println("半径为: " + r);
    System.out.println("面积为: " + area);
    System.out.println("周长为: " + circle);
    //如果面积>=周长,则输出"面积大于等于周长",否则,输出周长大于面积
    if(area >= circle) {
      System.out.println("面积大于等于周长");
    } else {
      System.out.println("周长大于面积");
    }
  }
}
```

执行结果如图3-4所示。

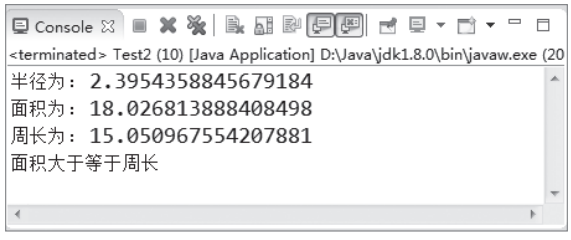



图3-4 示例3-2运行结果

条件运算符有时候可用于代替if-else，如示例3-3与示例3-4所示。

【示例3-3】if-else与条件运算符的比较：使用if-else

```
public class Test3 {
  public static void main(String[ ] args) {
    int a=2;
    int b=3;
    if (a<b) {
      System.out.println(a);
    } else {
      System.out.println(b);
    }
  }
}
```

执行结果如图3-5所示。

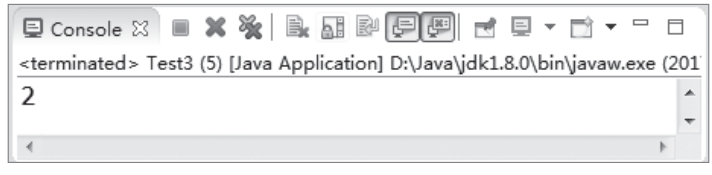



图3-5 示例3-3运行结果

【示例3-4】if-else与条件运算符的比较：使用条件运算符

```
public class Test4 {
  public static void main(String[ ] args) {
    int a=2;
    int b=3;
    System.out.println((a<b)?a:b);
  }
}
```

执行结果如图3-6所示。

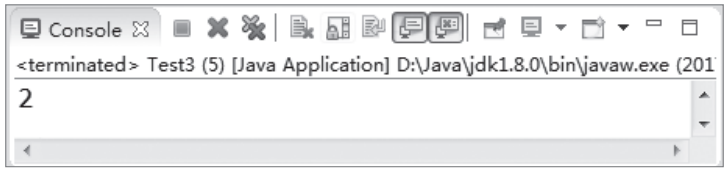



图3-6　示例3-4运行结果

## 3.1.3　if-else if-else多分支结构

if-else if-else的语法结构如下：

```
if(布尔表达式1) {
  语句块1;
} else if(布尔表达式2) {
  语句块2;
}……
else if(布尔表达式n){
  语句块n;
} else {
  语句块n+1;
}
```

当布尔表达式1的值为真时，执行语句块1；否则，判断布尔表达式2，当布尔表达式2的值为真时，执行语句块2；否则，继续判断布尔表达式3……；如果1～n个布尔表达式的值均判定为假时，则执行语句块n+1，也就是else部分。if-else if-else结构的流程图如图3-7所示。

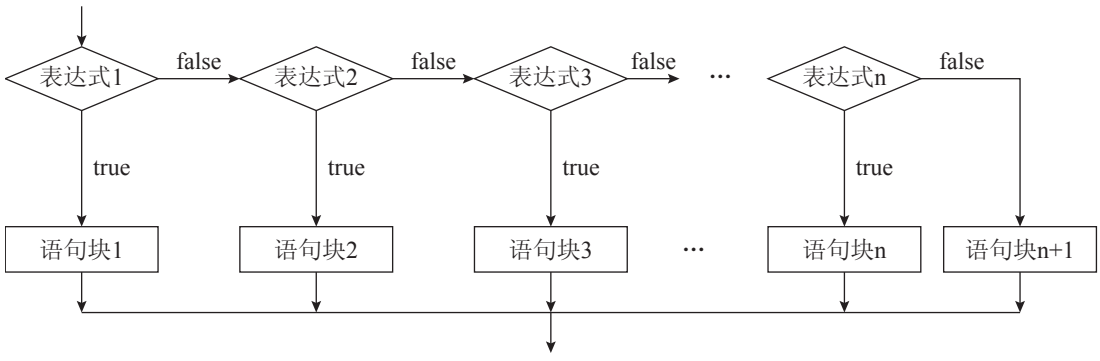



图3-7　if-else if-else多分支结构流程图

【示例3-5】if-else if-else多分支结构

```
public class Test5 {
  public static void main(String[ ] args) {
    int age = (int) (100 * Math.random());
    System.out.print("年龄是" + age + ", 属于");
    if(age < 15) {
      System.out.println("儿童, 喜欢玩! ");
    } else if(age < 25) {
      System.out.println("青年, 要学习! ");
    } else if(age < 45) {
      System.out.println("中年, 要工作! ");
    } else if(age < 65) {
      System.out.println("中老年, 要补钙! ");
    } else if(age < 85) {
      System.out.println("老年, 多运动! ");
    } else{
      System.out.println("老寿星, 古来稀! ");
    }
  }
}
```

执行结果如图3-8和图3-9所示。


图3-8 示例3-5运行结果1


图3-9 示例3-5运行结果2

**课堂练习**

仿照【示例3-5】，实现如下功能：

随机生成一个100以内的成绩，当成绩在85及以上的时候输出“等级A”；70以上到84之间输出“等级B”；60到69之间输出“等级C”；60以下输出“等级D”。

## 3.1.4 switch多分支结构

Switch语句的语法结构如下：

```
switch (表达式) {
  case 值1:
    语句序列1;
    [break];
  case 值2:
    语句序列2;
    [break];
  ……
  [default:
    默认语句;]
}
```

switch语句会根据表达式的值从相匹配的case标签处开始执行，一直执行到break语句处或者是switch语句的末尾。如果表达式的值与每个case值都不匹配，则进入default语句（如果存在default语句）。

根据表达式值的不同可以执行许多不同的操作。switch语句中case标签在JDK 1.5之前必须是整数（long类型除外）或者枚举，不能是字符串；在JDK 1.7之后允许使用字符串（String）。

**注意**

当布尔表达式是等值判断的情况，可以使用if-else if-else多分支结构或者switch结构，如果布尔表达式是区间判断的情况，则只能使用if-else if-else多分支结构。

switch多分支结构的流程图如图3-10所示。

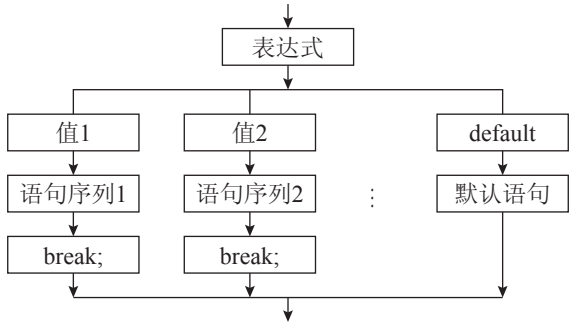



图3-10　switch多分支结构流程图

**【示例3-6】switch结构**

```
public class Test6 {
  public static void main(String[ ] args) {
    char c = 'a';
    int rand = (int) (26 * Math.random());
    char c2 = (char) (c + rand);
    System.out.print(c2 + ": ");
    switch (c2) {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
      System.out.println("元音");
      break;
    case 'y':
    case 'w':
      System.out.println("半元音");
      break;
    default:
      System.out.println("辅音");
    }
  }
}
```

执行结果如图3-11和图3-12所示。

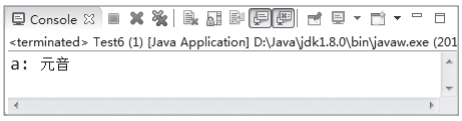



图3-11 示例3-6运行结果1

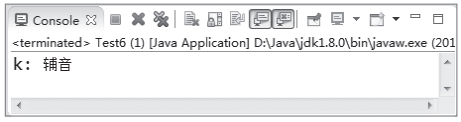



图3-12 示例3-6运行结果2

# 3.2 循环结构

循环结构分为两大类，一类是当型；另一类是直到型。

- 当型：当布尔表达式的值为true时，反复执行某语句，当布尔表达式的值为false时才停止循环，例如while与for循环。
- 直到型：先执行某语句，再判断布尔表达式，布尔表达式的值如果为true，再执行某语句。如此反复，直到布尔表达式的值为false时才停止循环，例如do-while循环。

## 3.2.1 while循环

While循环的语法结构如下：

```
while (布尔表达式) {
  循环体;
}
```

在循环刚开始时，会计算一次“布尔表达式”的值，若值为真，则执行循环体；而对于后来每一次额外的循环，都会在开始前重新计算一次。

语句中应有使循环趋向于结束的语句，否则会出现无限循环——“死”循环。

while循环结构流程图如图3-13所示。

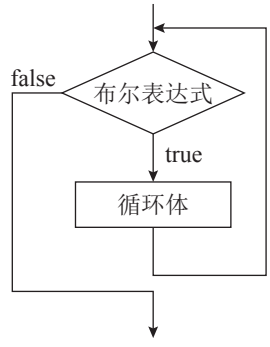



图3-13 while循环结构流程图

【示例3-7】while循环结构——求1～100的累加和

```
public class Test7 {
  public static void main(String[ ] args) {
```

```
        int  i = 0;
        int  sum = 0;
        //1+2+3+…+100=?
        while (i <= 100) {
          sum += i;//相当于sum = sum+i;
          i++;
        }
        System.out.println("Sum= " + sum);
    }
}
```

执行结果如图3-14所示。

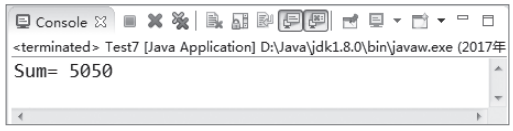



图3-14　示例3-7运行结果

## 3.2.2　do-while循环

do-while循环的语法结构如下：

```
do {
   循环体;
} while(布尔表达式) ;
```

do-while循环结构会先执行循环体，然后再判断布尔表达式的值，若值为真则执行循环体，当值为假时结束循环。do-while循环的循环体至少执行一次。do-while循环结构流程图如图3-15所示。

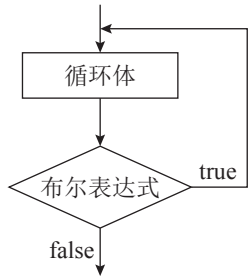



图3-15　do-while循环结构流程图

【示例3-8】do-while循环结构——求1～100的累加和

```
public class Test8 {
  public static void main(String[ ] args) {
    int i = 0;
    int sum = 0;
    do {
      sum += i; //sum = sum + i
```

```
        i++;
      } while (i <= 100);//此处的;不能省略
      System.out.println("Sum= " + sum);
    }
}
```

执行结果如图3-16所示。

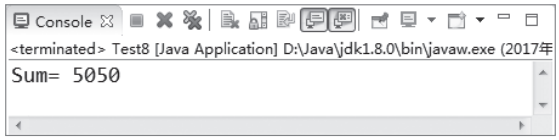



图3-16 示例3-8运行结果

【示例3-9】while与do-while的区别

```
public class Test9 {
  public static void main(String[ ] args) {
    //while循环:先判断再执行
    int a = 0;
    while (a < 0) {
      System.out.println(a);
      a++;
    }
    System.out.println("-----");
    //do-while循环:先执行再判断
    a = 0;
    do {
      System.out.println(a);
      a++;
    } while (a < 0);
  }
}
```

执行结果如图3-17所示。

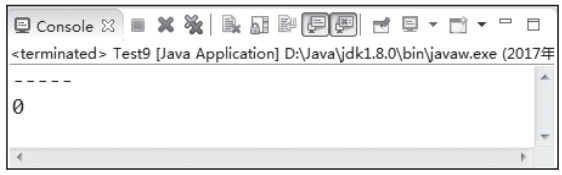



图3-17 示例3-9运行结果

从运行效果图中可以看出do-while总是保证循环体至少被执行一次。

### 3.2.3 for循环

for循环的语法结构如下：

```
for(初始表达式; 布尔表达式; 迭代因子) {
    循环体;
}
```

for循环语句是支持迭代的一种通用结构，是最有效、最灵活的循环结构。for循环在第一次反复之前要进行初始化，即执行初始表达式。随后，对布尔表达式的值进行判定，若判定结果为true，则执行循环体；否则，终止循环。最后在每一次反复的时候，进行某种形式的“步进”，即执行迭代因子。

- 初始化部分设置循环变量的初值。
- 条件判断部分为布尔表达式。
- 迭代因子控制循环变量的增减。

for循环在执行条件判定后，先执行的循环体部分，再执行步进。

for循环结构流程图如图3-18所示。

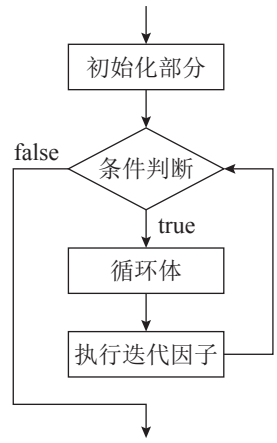



图3-18　for循环结构流程图

【示例3-10】for循环

```
public class Test10 {
  public static void main(String args[ ]) {
    int sum = 0;
    //1.求1~100之间的累加和
    for(int i = 0;i <= 100;i++) {
      sum += i;
    }
    System.out.println("Sum= " + sum);
    //2.循环输出9～1之间的数
    for(int i=9;i>0;i--){
      System.out.print(i+"、");
    }
    System.out.println();
    //3.输出90～1之间能被3整除的数
    for(int i=90;i>0;i-=3){
      System.out.print(i+"、");
    }
    System.out.println();
  }
}
```

执行结果如图3-19所示。

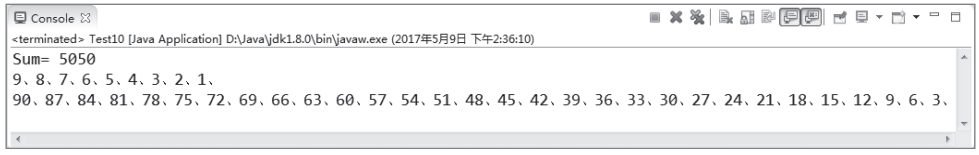



图3-19　示例3-10运行结果

Java语言中能用到逗号运算符的地方屈指可数，其中一处就是for循环的控制表达式。在控制表达式的初始化和步进控制部分，我们可以使用一系列由逗号分隔的表达式，而且那些表达式均会独立执行。

【示例3-11】逗号运算符

```
public class Test11 {
  public static void main(String[ ] args) {
    for(int i = 1, j = i + 10; i < 5; i++, j = i * 2) {
      System.out.println("i= " + i + " j= " + j);
      }
  }
}
```

执行结果如图3-20所示。

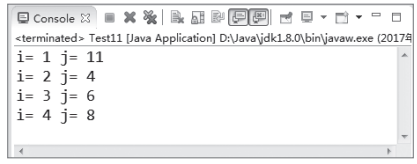



图3-20 示例3-11运行结果

- 无论在初始化还是在步进部分，语句都是顺序执行的。
- 尽管初始化部分可设置任意数量的定义，但都属于同一类型。
- 约定：只在for语句的控制表达式中写入与循环变量初始化、条件判断和迭代因子相关的表达式。

初始化部分、条件判断部分和迭代因子可以为空语句，但必须以“；”分开，如示例3-12所示。

【示例3-12】无限循环

```
public class Test12 {
  public static void main(String[ ] args) {
    for ( ; ; ) {    //无限循环: 相当于 while(true)
      System.out.println("北京尚学堂");
    }
  }
}
```

编译器将while(true)与for(;;)看作同一回事，都指的是无限循环。

在for语句的初始化部分声明的变量，其作用域为整个for循环体，不能在循环外部使用该变量，如示例3-13所示。

【示例3-13】初始化变量的作用域

```
public class Test13 {
  public static void main(String[] args) {
    for(int i = 1; i < 10; i++) {
      System.out.println(i+"、");
    }
    //编译错误,无法访问在for循环中定义的变量i
    System.out.println(i);
  }
}
```

### 3.2.4 嵌套循环

在一个循环语句内部再嵌套一个或多个循环，称为嵌套循环。while、do-while与for循环可以任意嵌套多层。

【示例3-14】嵌套循环

```
public class Test14 {
  public static void main(String args[ ]) {
    for(int i=1;i <=5;i++) {
        for(int j=1;j<=5;j++){
          System.out.print(i+"  ");
        }
      System.out.println();
    }
  }
}
```

执行结果如图3-21所示。

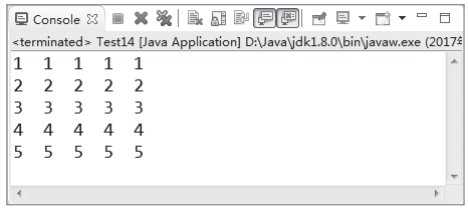



图3-21 示例3-14运行结果

【示例3-15】使用嵌套循环实现九九乘法表

```
public class Test15 {
  public static void main(String args[ ]) {
    for(int i = 1; i< 10;i++) {                //i是一个乘数
      for(int j = 1;j <= i;j++) {              //j是另一个乘数
        System.out.print(j+"*"+i+"="+(i*j<10?(""+i*j): i*j) +"");
      }
      System.out.println();
    }
  }
}
```

执行结果如图3-22所示。

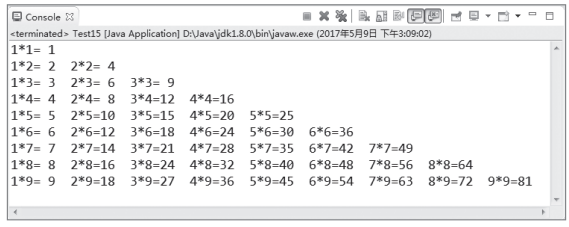



图3-22 示例3-15运行结果

**课堂练习**

- 用while循环分别计算100以内的奇数及偶数的和，并输出。
- 用while循环或其他循环输出1～1000能被5整除的数，且每行输出5个。

### 3.2.5 break语句和continue语句

在任何循环语句的主体部分，均可用break语句控制循环的流程。Break语句用于强行退出循环，不执行循环中剩余的语句。

**【示例3-16】break语句**

```
//产生100以内的随机数,直到随机数为88时终止循环
public class Test16 {
  public static void main(String[ ] args) {
    int total = 0;//定义计数器
    System.out.println("Begin");
    while (true) {
      total++;//每循环一次计数器加1
      int i = (int) Math.round(100 * Math.random());
      //当i等于88时,退出循环
      if (i == 88) {
        break;
      }
    }
    //输出循环的次数
    System.out.println("Game over,used " + total + " times.");
  }
}
```

执行结果如图3-23所示。

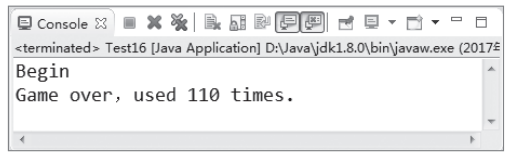



图3-23 示例3-16运行结果

continue语句用在循环语句体中，用于终止某次循环过程，即跳过循环体中尚未执行的语句，接着进行下一次是否执行循环的判定。

**注意**

- continue语句用在while、do-while循环中时，continue语句立刻跳到循环首部，越过了当前循环的其余部分。
- continue语句用在for循环中时，跳到for循环的迭代因子部分。

**【示例3-17】continue语句**

```
//把100~150之间不能被3整除的数输出,并且每行输出5个
public class Test17 {
  public static void main(String[ ] args) {
    int count = 0;//定义计数器
    for(int i = 100; i < 150; i++) {
      //如果是3的倍数,则跳过本次循环,继续进行下一次循环
      if(i % 3 == 0){
```

```
            continue;
          }
          //否则(不是3的倍数),输出该数
          System.out.print(i + "、");
          count++;//每输出一个数,计数器加1
          //根据计数器判断每行是否已经输出了5个数
          if(count % 5 == 0) {
            System.out.println();
          }
        }
      }
    }
```

执行结果如图3-24所示。

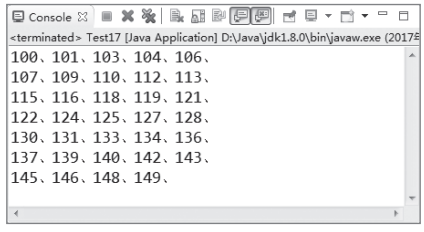



图3-24　示例3-17运行结果

## 3.2.6　带标签的break语句和continue语句

goto关键字很早就在程序设计语言中出现。尽管goto仍是Java的一个保留字，但并未在Java语言中得到正式使用，Java没有goto语句。然而，在break和continue这两个关键字身上，我们仍然能看出一些goto的影子——带标签的break语句和continue语句。

“标签”是指后面跟一个冒号的标识符，例如：“label：”。对Java来说唯一用到标签的地方是在循环语句之前，而在循环之前设置标签的唯一理由是：希望在其中嵌套另一个循环。由于break和continue关键字通常只中断当前循环，但若随同标签使用，它们就会中断到存在标签的地方。

在“goto有害”论中，最有问题的就是标签，而非goto。随着一个程序里的标签数量的增多，产生错误的概率也越来越高。但在Java环境下标签不会造成这方面的问题，因为它们的活动场所已被限定，不可能通过特别的方式到处传递程序的控制权。由此也引出了一个有趣的问题：通过限制语句的能力，反而能使一项语言特性更加有用。

【示例3-18】带标签的break语句和continue语句

```
//控制嵌套循环跳转(打印101～150之间所有的质数)
public class Test18 {
  public static void main(String args[ ]) {
    outer: for(int i = 101; i < 150; i++) {
      for(int j = 2; j < i / 2; j++) {
        if(i % j == 0){
```

```
            continue outer;
          }
        }
        System.out.print(i + "  ");
      }
    }
}
```

执行结果如图3-25所示。

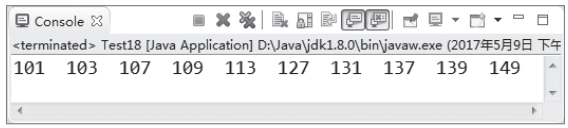



图3-25 示例3-18运行结果

## 3.3 语句块

语句块（有时叫作复合语句），是用花括号括起来的任意数量的简单Java语句。语句块确定了局部变量的作用域。语句块中的程序代码，作为一个整体，是要被一起执行的。语句块可以被嵌套在另一个语句块中，但是不能在两个嵌套的语句块内声明同名的变量。语句块可以使用外部的变量，而外部不能使用语句块中定义的变量，因为语句块中定义的变量作用域只限于语句块。

【示例3-19】语句块

```
public class Test19 {
  public static void main(String[ ] args) {
    int n;
    int a;
    {
      int k;
      int n;      //编译错误:不能重复定义变量n
    }             //变量k的作用域到此为止
  }
}
```

## 3.4 方法

方法就是一段用来完成特定功能的代码片段，类似于其他语言的函数。

方法用于定义该类或该类的实例的行为特征和功能实现。方法是类和对象行为特征的抽象。方法类似于面向过程中的函数，在面向过程中，函数是最基本的单位，整个程序由一个个函数调用组成。在面向对象编程中，整个程序的基本单位是类，方法是从属于类和对象的。

方法声明格式：

```
[修饰符1  修饰符2…]    返回值类型      方法名(形式参数列表){
  Java语句;……
 }
```

方法的调用方式：

```
对象名.方法名(实参列表)
```

方法的详细说明：

- 形式参数：在方法声明时用于接收外界传入的数据，简称形参。
- 实参：调用方法时，实际传给方法的数据。
- 返回值：方法在执行完毕后，返还给调用它的环境的数据。
- 返回值类型：事先约定的返回值的数据类型。如无返回值，必须显式指定其为void。

**【示例3-20】方法的声明及调用**

```
public class Test20 {
  /* main方法:程序的入口 */
  public static void main(String[ ] args) {
    int num1 = 10;
    int num2 = 20;
    //调用求和的方法:将num1与num2的值传给add方法中的n1与n2
    //求完和后将结果返回,用sum接收结果
    int sum = add(num1, num2);
    System.out.println("sum = " + sum);//输出:sum = 30
    //调用打印的方法:该方法没有返回值
    print();
  }
  /* 求和的方法 */
  public static int add(int n1, int n2) {
    int sum = n1 + n2;
    return sum;//使用return返回计算的结果
  }
  /* 打印的方法 */
  public static void print() {
    System.out.println("北京尚学堂...");
  }
}
```

执行结果如图3-26所示。

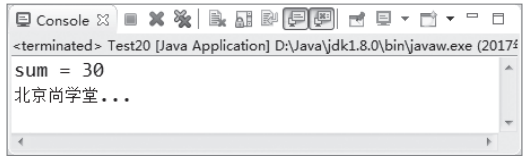



图3-26　示例3-20运行结果

**注意事项：**

- 实参的数目、数据类型和次序必须和所调用的方法声明的形式参数列表匹配。

- 注意return语句终止方法的运行并指定要返回的数据。
- Java在方法调用中传递参数时，遵循值传递的原则（传递的都是数据的副本）：
  - 基本类型传递的是该数据值的copy值。
  - 引用类型传递的是该对象引用的copy值，但指向的是同一个对象。

## 3.5 方法的重载

方法的重载（overload）是指一个类中可以定义多个方法名相同，但参数不同的方法。调用时，会根据不同的参数自动匹配对应的方法。

**菜鸟雷区**

重载的方法，实际是完全不同的方法，只是名称相同而已。

构成方法重载的条件如下：

- 不同的含义：形参类型、形参个数、形参顺序不同。
- 只有返回值不同不构成方法的重载，如int a(String str){}与void a(String str){}不构成方法重载。
- 只有形参的名称不同，不构成方法的重载，如int a(String str){}与int a(String s){}不构成方法重载。

**【示例3-21】方法重载**

```
public class Test21 {
  public static void main(String[ ] args) {
    System.out.println(add(3, 5));           //8
    System.out.println(add(3, 5, 10));       //18
    System.out.println(add(3.0, 5));         //8.0
    System.out.println(add(3, 5.0));         //8.0
                                             //我们已经见过的方法的重载
    System.out.println();                    //0个参数
    System.out.println(1);                   //参数是1个int
    System.out.println(3.0);                 //参数是1个double
  }
  /* 求和的方法 */
  public static int add(int n1, int n2) {
    int sum = n1 + n2;
    return sum;
  }
  //方法名相同,参数个数不同,构成重载
  public static int add(int n1, int n2, int n3) {
    int sum = n1 + n2 + n3;
    return sum;
  }
  //方法名相同,参数类型不同,构成重载
  public static double add(double n1, int n2) {
    double sum = n1 + n2;
    return sum;
```

```
    }
    //方法名相同,参数顺序不同,构成重载
    public static double add(int n1, double n2) {
      double sum = n1 + n2;
      return sum;
    }
    //编译错误:只有返回值不同,不构成方法的重载
    public static double add(int n1, int n2) {
      double sum = n1 + n2;
      return sum;
    }
    //编译错误:只有参数名称不同,不构成方法的重载
    public static int add(int n2, int n1) {
      double sum = n1 + n2;
      return sum;
    }
}
```

## 3.6 递归结构

递归是一种常见的解决问题的方法，即把问题逐渐简单化。递归的基本思想就是“自己调用自己”，一个使用递归技术的方法将会直接或者间接地调用自己。

利用递归可以用简单的程序来解决一些复杂的问题，例如斐波那契数列的计算、汉诺塔、快排等。

递归结构包括两个部分：

- 定义递归头：解答“什么时候不调用自身方法”。如果没有头，将陷入死循环，也就是递归的结束条件。
- 递归体：解答“什么时候需要调用自身方法”。

【示例3-22】使用递归求n!

```
public class Test22 {
  public static void main(String[ ] args) {
    long d1 = System.currentTimeMillis();
    System.out.printf("%d阶乘的结果:%s%n", 10, factorial(10));
    long d2 = System.currentTimeMillis();
    System.out.printf("递归费时:%s%n", d2-d1);  //耗时:32ms
  }
  /* 求阶乘的方法*/
  static long  factorial(int n){
    if(n==1){//递归头
      return 1;
    }else{//递归体
      return n*factorial(n-1);//n! = n * (n-1)!
    }
  }
}
```

执行结果如图3-27所示。

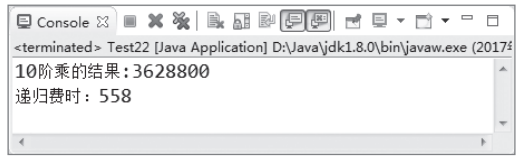
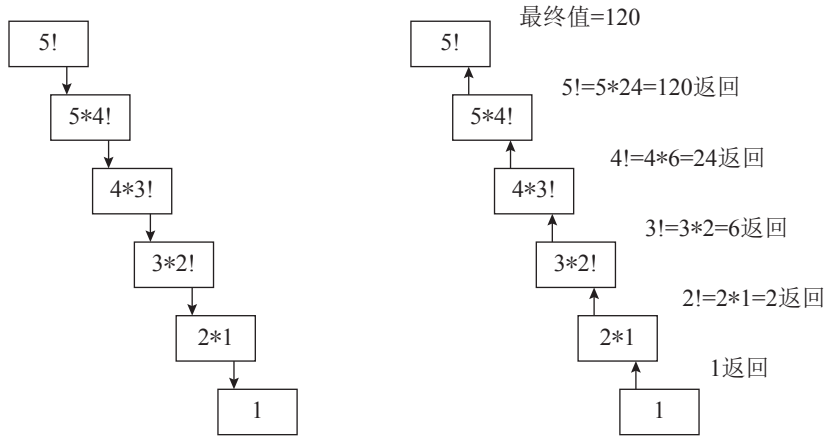



图3-27 示例3-22运行结果

递归原理分析如图3-28所示。




A.递归调用过程 B.每次递归调用的返回值

图3-28 递归原理分析图

**递归的缺陷**

简单是递归的优点之一。但是递归调用会占用大量的系统堆栈，内存耗用多，在递归调用层次多时速度要比循环慢得多，所以在使用递归方法时要慎重。

例如，上面的递归程序运行耗时558ms，如果用普通循环的话会快得多，如示例3-23所示。

**【示例3-23】使用循环求n!**

```
public class Test23 {
  public static void main(String[ ] args) {
    long d3 = System.currentTimeMillis();
    int a = 10;
    int result = 1;
    while(a > 1) {
      result *= a * (a - 1);
      a -= 2;
    }
    long d4 = System.currentTimeMillis();
```

```
        System.out.println(result);
        System.out.printf("普通循环费时:%s%n", d4 - d3);
    }
}
```

执行结果如图3-29所示。

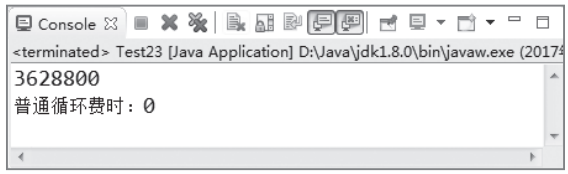



图3-29　示例3-23运行结果

**注意**

- **任何能用递归解决的问题也能使用迭代解决。**当递归方法可以更加自然地反映问题，并且易于理解和调试，并且不强调效率问题时，可以采用递归方法。
- 在要求高性能的情况下尽量避免使用递归，递归调用既花时间又耗内存。

## 本章总结

（1）从结构化程序设计角度出发，程序有三种结构：顺序结构、选择结构和循环结构。

（2）选择结构包括：

- if单选择结构、if-else双选择结构、if-else if-else多选择结构；
- switch多选择结构。

（3）多选择结构与switch的关系是当布尔表达式是等值判断的情况，可使用多重选择结构或switch结构；如果布尔表达式为区间判断的情况，则只能使用多重选择结构。

（4）循环结构包括如下两种类型。

- 当型：while与for。
- 直到型：do-while。

（5）while与do-while的区别是，在布尔表达式的值为false时，while的循环体一次也不执行，而do-while至少执行一次。

（6）break语句可以在switch语句与循环结构中使用，而continue语句只能在循环结构中使用。

（7）方法就是一段用来完成特定功能的代码片段，类似于其他语言的函数。

（8）方法的重载是指一个类中可以定义多个方法名相同，但参数不同的方法。调用时，会根据不同的参数自动匹配对应的方法。

（9）任何能用递归解决的问题也能使用迭代解决。在要求高性能的情况下尽量避免使用递归方法，递归调用既花时间又耗内存。

## 本章作业

### 一、选择题

1. 分析如下Java代码，编译运行的输出结果是（　　）（选择一项）。

```
  public static void main(String[ ] args) {
    boolean a=true;
    boolean b=false;
    if (!(a&&b)) {
      System.out.print("!(a&&b)");
    } else if (!(a||b)) {
      System.out.println("!(a||b)");
    } else {
      System.out.println("ab");
    }
}
```

A. !(a&&b)　　B. !(a||b)

C. ab　　D. !(a||b)ab

2. 下列选项中关于变量x的定义，（　　）可使以下switch语句编译通过（选择二项）。

```
  switch(x) {
    case 100 :
      System.out.println("One hundred");
      break;
    case 200 :
      System.out.println("Two hundred");
      break;
    case 300 :
      System.out.println( "Three hundred");
      break;
    default :
      System.out.println( "default");
}
```

A. double x = 100;　　B. char x = 100;

C. String x = "100";　　D. int x = 100;

3. 给定如下Java代码，编译运行的结果是（　　）（选择一项）。

```
  public class Test {
  public static void main(String[ ] args) {
    int sum=0;
    for(int i=1;i<10;i++){
      do{
        i++;
        if(i%2!=0)
          sum+=i;
      }while(i<6);
    }
    System.out.println(sum);
  }
}
```

A. 8　　B. 15

C. 24　　D. 什么也不输出

4. 以下选项中添加到代码中横线处会出现错误的是（　　）（选择二项）。

```
public class Test {
public float aMethod(float a, float b) {
  return 0;
  }_____________________________
}
```

A.
```
public float aMethod(float a,float b,float c) {
    return 0;
  }
```

B.
```
public float aMethod(float c,float d) {
    return 0;
  }
```

C.
```
public int aMethod(int a,int b) {
    return 0;
  }
```

D.
```
private int aMethod(float a,float b) {
    return 0;
  }
```

5. 以下关于方法调用的代码的执行结果是（　　）（选择一项）。

```
public class Test {
  public static void main(String args[ ]) {
    int i = 99;
    mb_operate(i);
    System.out.print(i + 100);
  }
  static void mb_operate(int i) {
    i += 100;
  }
}
```

A. 99　　B. 199

C. 299　　D. 99100

## 二、简答题

1. if多分支语句和switch语句的异同之处。
2. break和continue语句的作用。
3. 在多重循环中，如何在内层循环中使用break跳出外层循环。
4. 方法重载的定义、作用和判断依据。
5. 递归的定义和优缺点。

## 三、编码题

1. 从键盘输入某个十进制整数，转换成对应的二进制整数并输出。
2. 编程求∑1+∑2+…+∑100的值。
3. 编写递归算法程序。一列数的规则如下：1,1,2,3,5,8,13,21,34,…求数列的第40位数是多少。

# 第5章 Java面向对象编程进阶

本章重点针对面向对象编程的三大特征：继承、封装、多态进行详细讲解，另外还包括抽象类、接口、内部类等概念。很多概念对于初学者来说比较陌生，应先进行语法性质的了解，不要期望通过本章的学习就能够“搞透面向对象编程”。本章只是面向对象编程的起点，后面的所有章节都是对本章内容的应用。

**老鸟建议**

建议大家学习本章时莫停留，学完以后，迅速开展后面各章节的学习。可以这么说，后续章节的所有编程都是对“面向对象思想”的应用而已。

## 5.1 继承

继承（extends）是面向对象编程的三大特征之一，它让人们更加容易实现对已有类的扩展，更加容易实现对现实世界的建模。

### 5.1.1 继承的实现

继承让人们更加容易实现类的扩展。例如，定义了“人类”，再定义Boy类就只需要扩展“人类”即可。继承实现了代码的重用，不用再重新发明轮子（don’t reinvent wheels）。

从英文字面意思理解，extends的意思是扩展。子类是父类的扩展。现实世界中的继承无处不在。以图5-1为例，哺乳动物继承了动物，这意味着动物的特性哺乳动物都有。在编程时，如果新定义一个Student类，发现已经有Person类包含了所需的属性和方法，那么Student类只需要继承Person类即可拥有Person类的属性和方法。

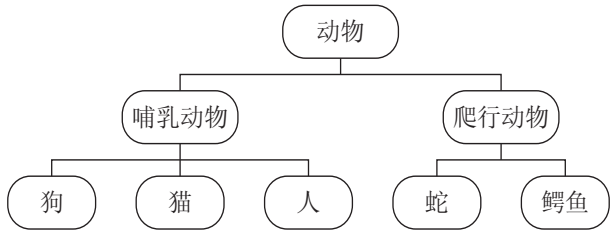



图5-1 现实世界中的继承

【示例5-1】使用extends实现继承

```
public class Test{
  public static void main(String[ ] args) {
    Student s = new Student("高淇",172,"Java");
    s.rest();
    s.study();
  }
}
class Person {
  String name;
  int height;
  public void rest(){
    System.out.println("休息一会！");
  }
}
class Student extends Person {
  String major; //专业
  public void study(){
    System.out.println("在尚学堂,学习Java");
  }
  public Student(String name,int height,String major) {
    //天然拥有父类的属性
    this.name = name;
    this.height = height;
    this.major = major;
  }
}
```

执行结果如图5-2所示。

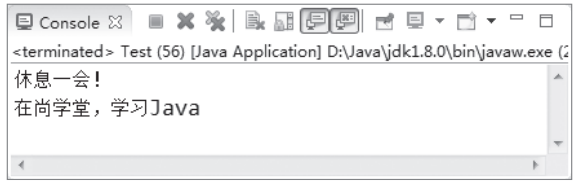



图5-2 示例5-1运行结果

## 5.1.2 instanceof运算符

instanceof是二元运算符，其左边是对象，右边是类；当对象是右边的类或子类所创建

的对象时返回true，否则返回false，如示例5-2所示。

【示例5-2】使用instanceof运算符进行类型判断

```
public class Test{
  public static void main(String[ ] args) {
    Student s = new Student("高淇",172,"Java");
    System.out.println(s instanceof Person);
    System.out.println(s instanceof Student);
  }
}
```

两条instanceof语句的输出结果都是true。

### 5.1.3 继承的使用要点

继承的使用要点如下：

（1）父类也称作超类、基类、派生类等。

（2）Java中只有单继承，没有像C++那样的多继承。多继承会引起混乱，使得继承链过于复杂，系统难以维护。

（3）Java中的类没有多继承，接口则有多继承。

（4）子类继承父类，可以得到父类的全部属性和方法（除了父类的构造器），但不见得可以直接访问（例如，父类私有的属性和方法）。

（5）如果定义一个类时，没有调用extends，则它的父类是java.lang.Object。

### 5.1.4 方法的重写

子类通过重写父类的方法，可以用自身的行为替换父类的行为。方法的重写是实现多态的必要条件。

方法的重写需要符合下面三个要点：

（1）“==”：方法名、形参列表相同。

（2）“≤”：返回值类型和声明异常类型，子类小于等于父类。

（3）“≥”：访问权限，子类大于等于父类。

【示例5-3】方法重写

```
public class TestOverride {
  public static void main(String[ ] args) {
    Vehicle v1 = new Vehicle();
    Vehicle v2 = new Horse();
    Vehicle v3 = new Plane();
    v1.run();
    v2.run();
    v3.run();
    v2.stop();
    v3.stop();
  }
```

```
}

class Vehicle { //交通工具类
  public void run() {
    System.out.println("跑....");
  }
  public void stop() {
    System.out.println("停止不动");
  }
}
class Horse extends Vehicle { //马也是交通工具
  public void run() { //重写父类方法
    System.out.println("四蹄翻飞,嘚嘚嘚...");
  }
}
class Plane extends Vehicle {
  public void run() { //重写父类方法
    System.out.println("天上飞！");
  }
  public void stop() {
    System.out.println("空中不能停,坠毁了！");
  }
}
```

执行结果如图5-3所示。

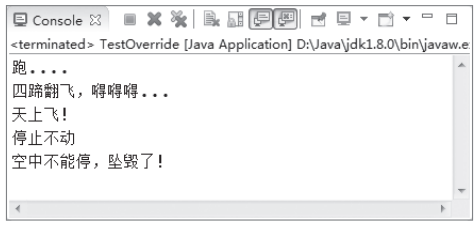



图5-3　示例5-3运行结果

## 5.2　Object类

前面学习的所有类及以后要定义的所有类都是Object类的子类，也都具备Object类的所有特性。因此，我们非常有必要掌握Object类的用法。

### 5.2.1　Object类的基本特性

Object类是所有Java类的根基类，也就意味着所有Java对象都拥有Object类的属性和方法。如果在类的声明中未使用extends关键字指明其父类，则默认继承Object类。

【示例5-4】Object类

```
public class Person {
  ...
}
```

```
//等价于:
public class Person extends Object {
  ...
}
```

## 5.2.2 toString方法

Object类中定义有public String toString()方法，其返回值是String类型。Object类中toString方法的源码如下。

```
public String toString() {
  return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

根据如上源码得知，默认会返回“类名+@+十六进制的hashcode”。在打印输出或者用字符串连接对象时，会自动调用该对象的toString()方法。

【示例5-5】重写toString()方法

```
class Person {
  String name;
  int age;
  @Override
  public String toString() {
    return name+",年龄:"+age;
  }
}
public class Test {
  public static void main(String[ ] args) {
    Person p=new Person();
    p.age=20;
    p.name="李东";
    System.out.println("info:"+p);

    Test t = new Test();
    System.out.println(t);
  }
}
```

执行结果如图5-4所示。

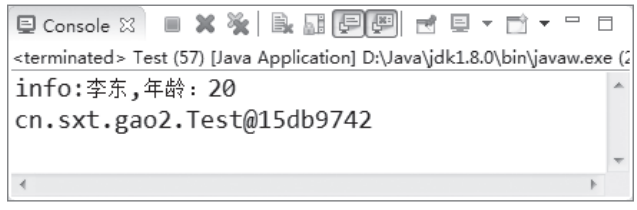



图5-4 示例5-5运行结果

### 5.2.3 ==和equals方法

“==”代表比较双方是否相同。如果是基本类型则表示值相等，如果是引用类型则表示地址相等，即是同一个对象。

Object类中定义有：public boolean equals(Object obj)方法，提供定义“对象内容相等”的逻辑。例如，公安系统中认为id相同的人就是同一个人，学籍系统中认为学号相同的人就是同一个人。

Object的equals方法默认就是比较两个对象的hashcode，是同一个对象的引用时返回true否则返回false。但是，也可以根据自己的要求重写equals方法。

【示例5-6】自定义类重写equals()方法

```
public class TestEquals {
  public static void main(String[ ] args) {
    Person p1 = new Person(123,"高淇");
    Person p2 = new Person(123,"高小七");
    System.out.println(p1==p2);           //false,不是同一个对象
    System.out.println(p1.equals(p2));    //true,id相同则认为两个对象内容相同
    String s1 = new String("尚学堂");
    String s2 = new String("尚学堂");
    System.out.println(s1==s2);           //false,两个字符串不是同一个对象
    System.out.println(s1.equals(s2));    //true,两个字符串内容相同
  }
}
class Person {
  int id;
  String name;
  public Person(int id,String name) {
    this.id=id;
    this.name=name;
  }
  public boolean equals(Object obj) {
    if(obj == null){
      return false;
    }else {
      if(obj instanceof Person) {
        Person c = (Person)obj;
        if(c.id==this.id) {
          return true;
        }
      }
    }
    return false;
  }
}
```

JDK提供的一些类，如String、Date、包装类等，重写了Object的equals方法，调用这些类的equals方法为x.equals(y)，当x和y所引用的对象是同一类对象且属性内容相等时（并不一定是相同对象）返回true，否则返回false。

## 5.3 super关键字

super是直接父类对象的引用，可以通过super来访问父类中被子类覆盖的方法或属性。

使用super调用普通方法，语句没有位置限制，可以在子类中随便调用。

若是构造器的第一行代码没有显式调用super(...)或者this(...)，Java都会默认调用super()，含义是调用父类的无参数构造器。这里的super()可以省略。

【示例5-7】super关键字的使用

```
public class TestSuper01 {
  public static void main(String[ ] args) {
    new ChildClass().f();
  }
}
class FatherClass {
  public int value;
  public void f(){
    value = 100;
    System.out.println ("FatherClass.value="+value);
  }
}
class ChildClass extends FatherClass {
  public int value;
  public void f() {
    super.f();                          //调用父类对象的普通方法
    value = 200;
    System.out.println("ChildClass.value="+value);
    System.out.println(value);
    System.out.println(super.value); //调用父类对象的成员变量
  }
}
```

执行结果如图5-5所示。

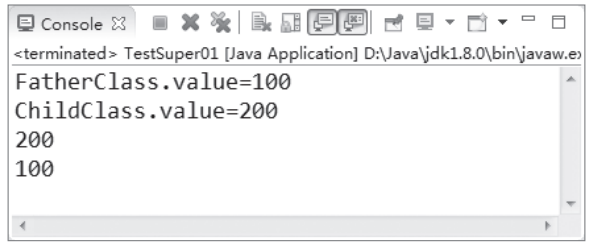



图5-5 示例5-7运行结果

### 继承树追溯

下面讲解继承树追溯的相关内容。

#### 1. 属性/方法查找顺序（例如查找变量h）

（1）在当前类中查找属性h。

（2）依次上溯每个父类，查看每个父类中是否有h，直到Object为止。

（3）如果没找到，则出现编译错误。

（4）只要找到h变量，则终止这个过程。

### 2. 构造器调用顺序

构造器的第一句总是super(…)，用来调用与父类对应的构造器。所以，流程就是：先向上追溯到Object，然后再依次向下执行类的初始化块和构造器，直到当前子类为止。

注 静态初始化块调用顺序与构造器调用顺序一样，不再重复。

【示例5-8】继承条件下构造器的执行过程

```
public class TestSuper02 {
  public static void main(String[ ] args) {
    System.out.println("开始创建一个ChildClass对象......");
    new ChildClass();
  }
}
class FatherClass {
  public FatherClass() {
    System.out.println("创建FatherClass");
  }
}
class ChildClass extends FatherClass {
  public ChildClass() {
    System.out.println("创建ChildClass");
  }
}
```

执行结果如图5-6所示。

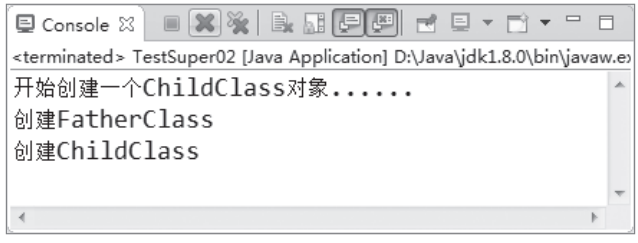



图5-6 示例5-8运行结果

## 5.4 封装

封装（encapsulation）是面向对象编程的三大特征之一。对于程序的合理封装可让外部调用更加方便，更加利于编程。同时，对于实现者来说也更加容易修正和改版代码。

### 5.4.1 封装的作用和含义

人们要看电视，只需要按一下开关和换台就可以了，没必要了解电视机的内部结构，也

没必要碰显像管。制造厂家为了方便用户使用电视，把复杂的内部构造全部封装起来，只留出简单的接口，例如电源开关。电视节目的播放在内部是如何实现的，不需要用户操心。

需要让用户知道的才暴露出来，不需要让用户知道的全部隐藏起来，这就是封装。用专业语言来表述，封装就是把对象的属性和操作结合为一个独立的整体，并尽可能隐藏对象内部的实现细节。

程序设计要追求“高内聚，低耦合”。 高内聚就是类内部的数据操作细节在其内部完成，不允许外部干涉；低耦合是指仅暴露必要的方法给外部使用，尽量方便外部调用。

在编程中，封装的优点如下。

- 提高代码的安全性。
- 提高代码的复用性。
- 高内聚：封装细节，便于修改内部代码，提高可维护性。
- 低耦合：简化外部调用，便于调用者使用，便于扩展和协作。

【示例5-9】未进行封装的代码演示

```
class Person {
  String name;
  int age;
  @Override
  public String toString() {
    return "Person [name=" + name + ", age=" + age + "]";
  }
}
public class Test {
  public static void main(String[ ] args) {
    Person p = new Person();
    p.name = "小红";
    p.age = -45;//年龄可以通过这种方式随意赋值,没有任何限制
    System.out.println(p);
  }
}
```

人们都知道，年龄不可能是负数，也不太可能超过130岁，但是如果没有进行封装的话，便可以给年龄赋值任意的整数，这显然不符合正常的逻辑人。执行结果如图5-7所示。

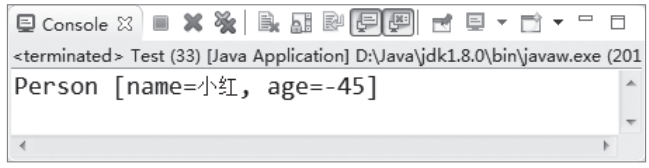



图5-7 示例5-9运行结果

再例如，如果哪天需要将Person类中age的属性修改为String类型，应该怎么办？假如只有一处使用了这个类，还比较幸运，如果有几处甚至上百处都用到了，那么修改的工作量是极大的。而封装恰恰能够解决这样的问题。如果使用了封装，修改时只需要修改Person类的setAge()方法即可，而无须修改使用了该类的客户代码。

### 5.4.2 封装的实现——使用访问控制符

Java使用访问控制符（访问权限修饰符）来控制哪些细节需要封装，哪些细节需要暴露。 Java中有4种访问控制符，分别为private、default、protected和public。它们说明了面向对象的封装性，所以要利用它们尽可能地便访问权限降到最低，从而提高安全性。

下面详细讲述访问控制符的访问权限问题，其访问权限范围如表5-1所示。

表5-1 访问控制符及其权限

| 修饰符 | 同一个类 | 同一个包中 | 子类 | 所有类 |
|---|---|---|---|---|
| private | * | | | |
| default | * | * | | |
| protected | * | * | * | |
| public | * | * | * | * |

（1）private表示私有，只有自己的类能访问。

（2）default表示没有修饰符修饰，只能访问同一个包的类。

（3）protected表示可以被同一个包的类以及其他包中的子类访问。

（4）public表示可以被该项目的所有包中的所有类访问。

下面进一步说明Java中的4种访问权限修饰符的区别：首先创建4个类：Person类、Student类、Animal类和Computer类，分别比较本类、本包、子类、其他包的区别。

public访问权限修饰符示例，如图5-8～图5-11所示。


图5-8 public访问权限——本类中访问public属性

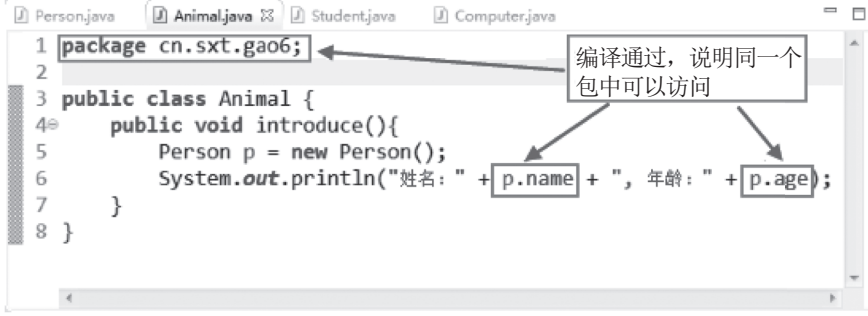



图5-9 public访问权限——本包中访问public属性

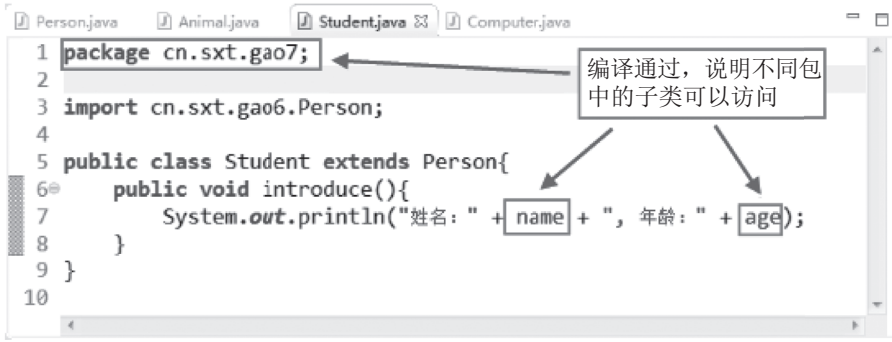



图5-10　public访问权限——不同包中的子类访问public属性

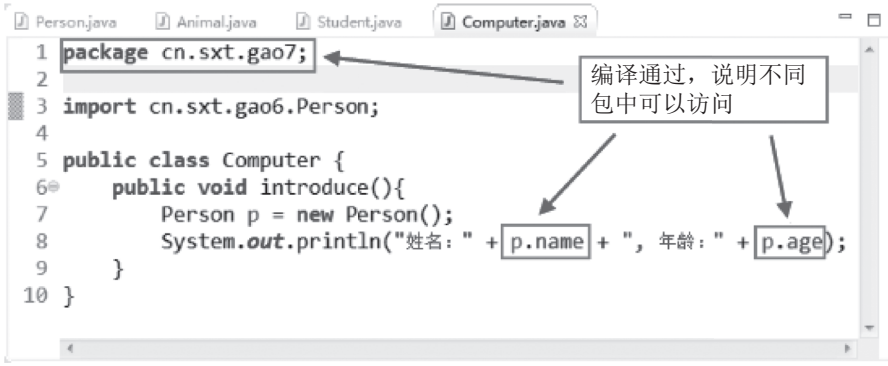



图5-11　public访问权限——不同包中的非子类访问public属性

图5-8～图5-11可以说明public访问控制符的访问权限为：该项目的所有包中的所有类。

protected访问权限修饰符示例：将Person类中的属性改为protected，其他类不修改，如图5-12和图5-13所示。

图5-12和图5-13可以说明protected修饰符的访问权限为：同一个包中的类以及其他包中的子类。

默认访问权限修饰符：将Person类中属性改为默认的，其他类不修改，如图5-14所示。


图5-12　protected访问权限——修改后的Person类

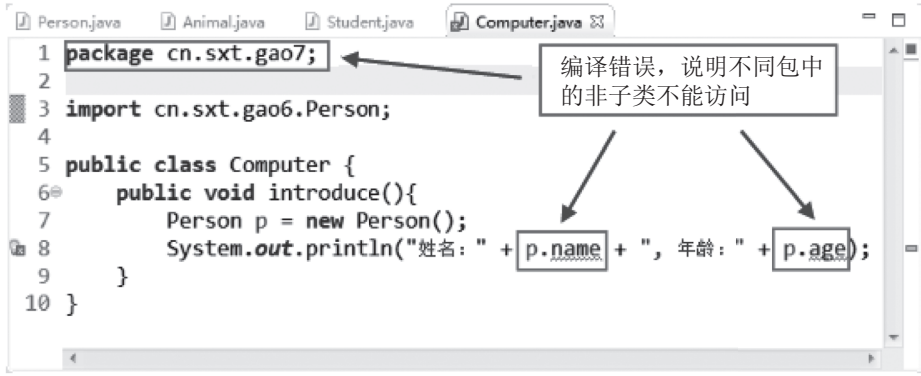



图5-13　protected访问权限——不同包中的非子类不能访问protected属性


图5-14　默认访问权限——修改后的Person类

图5-14可以说明默认修饰符的访问权限为：同一个包中的类。

private访问权限修饰符：将Person类中的属性改为private，其他类不修改，如图5-15所示。

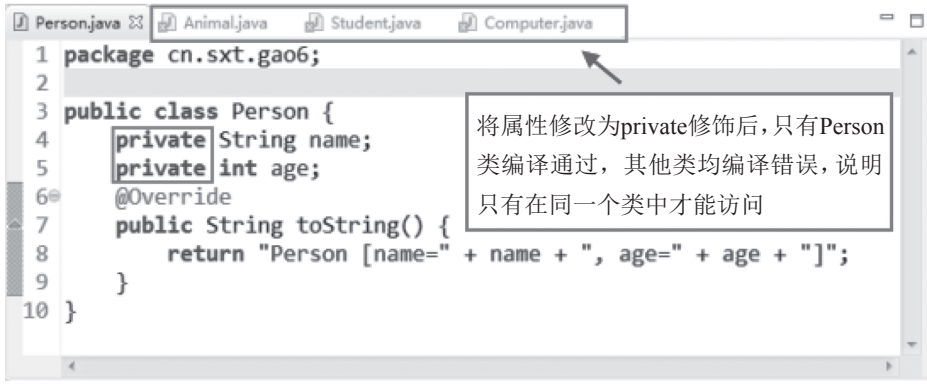



图5-15　private访问权限——修改后的Person类

图5-15可以说明private修饰符的访问权限为：同一个类。

### 5.4.3 封装的使用细节

类的属性的处理如下：

- 一般使用private访问权限。
- 提供相应的get/set方法来访问相关属性，这些方法通常是public修饰的，以提供对属性的赋值与读取操作（注意：boolean变量的get方法是is开头）。
- 一些只用于本类的辅助性方法可以用private修饰，希望其他类调用的方法用public修饰。

【示例5-10】JavaBean的封装演示

```
public class Person {
  //属性一般使用private修饰
  private String name;
  private int age;
  private boolean flag;
  //为属性提供public修饰的set/get方法
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public int getAge() {
    return age;
  }
  public void setAge(int age) {
    this.age = age;
  }
  public boolean isFlag() {//注意:boolean类型的属性get方法是is开头的
    return flag;
  }
  public void setFlag(boolean flag) {
    this.flag = flag;
  }
}
```

下面使用封装技术来解决5.4.1节中提到的年龄非法赋值问题。

【示例5-11】封装的使用

```
class Person {
  private String name;
  private int age;
  public Person() {

  }
  public Person(String name, int age) {
    this.name = name;
    //this.age = age;构造器中不能直接赋值,应该调用setAge方法
    setAge(age);
  }
```

```
    public void setName(String name) {
      this.name = name;
    }
    public String getName() {
      return name;
    }
    public void setAge(int age) {
      //在赋值之前先判断年龄是否合法
      if (age > 130 || age < 0) {
        this.age = 18;//不合法赋默认值18
      } else {
        this.age = age;//合法才能赋值给属性age
      }
    }
    public int getAge() {
      return age;
    }
    @Override
    public String toString() {
      return "Person [name=" + name + ", age=" + age + "]";
    }
  }
  public class Test2 {
    public static void main(String[ ] args) {
      Person p1 = new Person();
      //p1.name = "小红"; 编译错误
      //p1.age = -45;     编译错误
      p1.setName("小红");
      p1.setAge(-45);
      System.out.println(p1);

      Person p2 = new Person("小白", 300);
      System.out.println(p2);
    }
  }
```

执行结果如图5-16所示。

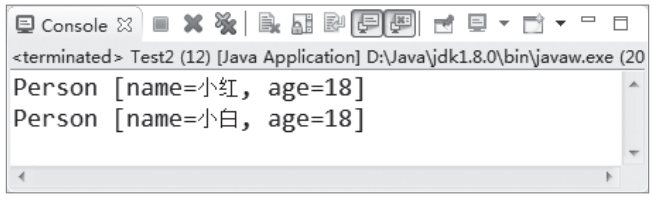



图5-16　示例5-11运行结果

## 5.5　多态

多态（polymorphism）是指同一个方法调用由于对象的不同可能会产生不同的行为。在现实生活中，同一个方法的具体实现会完全不同。例如，同样是调用人的休息方法，张三是睡觉，李四是旅游，高淇老师是敲代码，数学教授是做数学题； 同样是调用人吃饭的方

法，中国人用筷子吃饭，英国人用刀叉吃饭，印度人用手吃饭。

关于多态要注意以下3点：

（1）多态是方法的多态，不是属性的多态（多态与属性无关）。

（2）多态的存在要有3个必要条件：继承，方法重写，父类引用指向子类对象。

（3）父类引用指向子类对象后，用该父类引用调用子类重写的方法，此时多态就出现了。

【示例5-12】多态和类型转换

```
class Animal {
  public void shout() {
    System.out.println("叫了一声！");
  }
}
class Dog extends Animal {
  public void shout() {
    System.out.println("旺旺旺！");
  }
  public void seeDoor() {
    System.out.println("看门中....");
  }
}
class Cat extends Animal {
  public void shout() {
    System.out.println("喵喵喵喵！");
  }
}
public class TestPolym {
  public static void main(String[ ] args) {
    Animal a1 = new Cat(); // 向上可以自动转型
    //传的具体是哪一个类就调用哪一个类的方法。大大提高了程序的可扩展性
    animalCry(a1);
    Animal a2 = new Dog();
    animalCry(a2);//a2为编译类型,Dog对象才是运行时类型
    /*编写程序时,如果想调用运行时类型的方法,只能进行强制类型转换。
      否则通不过编译器的检查。*/
    Dog dog = (Dog)a2;//向下需要强制类型转换
    dog.seeDoor();
  }
  //有了多态,只需要让增加的这个类继承Animal类就可以了
  static void animalCry(Animal a) {
    a.shout();
  }

  /* 如果没有多态,我们这里需要写很多重载的方法。
     每增加一种动物,就需要重载一种动物的喊叫方法,非常麻烦。
  static void animalCry(Dog d) {
    d.shout();
  }
  static void animalCry(Cat c) {
    c.shout();
  }*/
}
```

执行结果如图5-17所示。

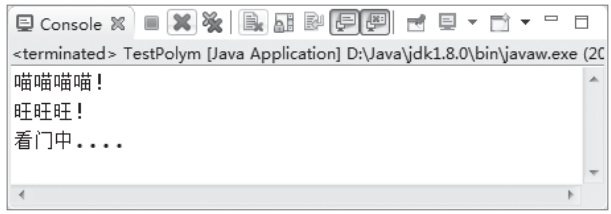



图5-17　示例5-12运行结果

示例5-12展示了多态最为多见的一种用法，即将父类引用做方法的形参，实参可以是任意的子类对象，通过不同的子类对象可实现不同的行为方式。

由此可以看出，多态的主要优势是提高了代码的可扩展性，符合开闭原则。但是多态也有弊端，就是无法调用子类特有的功能，例如，不能使用父类的引用变量调用Dog类特有的seeDoor()方法。

如果只想使用子类特有的功能，可以使用下一节所讲的方法—对象的转型。

## 5.6　对象的转型

对象的转型（casting）分为向上转型和向下转型。

父类引用指向子类对象，这个过程称为向上转型，属于自动类型转换。

向上转型后的父类引用变量只能调用它编译类型的方法，不能调用它运行时类型的方法，这时就需要进行类型的强制转换，称之为向下转型。

【示例5-13】对象的转型

```
public class TestCasting {
  public static void main(String[ ] args) {
    Object obj = new String("北京尚学堂");     //向上可以自动转型
    //obj.charAt(0) 无法调用。编译器认为obj是Object类型而不是String类型
    /* 编写程序时,如果想调用运行时类型的方法,只能进行强制类型转换。
       不然通不过编译器的检查。 */
    String str = (String) obj;                 //向下转型
    System.out.println(str.charAt(0));         //位于0索引位置的字符
    System.out.println(obj == str);            //true,它们俩运行时是同一个对象
  }
}
```

执行结果如图5-18所示。

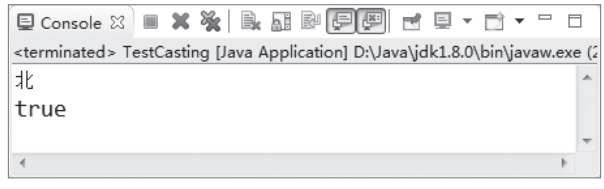



图5-18　示例5-13运行结果

在向下转型过程中，必须将引用变量转成真实的子类类型（运行时类型），否则会出现类型转换异常（ClassCastException）问题，如示例5-14所示。

【示例5-14】类型转换异常

```
public class TestCasting2 {
  public static void main(String[ ] args) {
    Object obj = new String("北京尚学堂");
    //真实的子类类型是String,但是此处向下转型为StringBuffer
    StringBuffer str = (StringBuffer) obj;
    System.out.println(str.charAt(0));
  }
}
```

执行结果如图5-19所示。

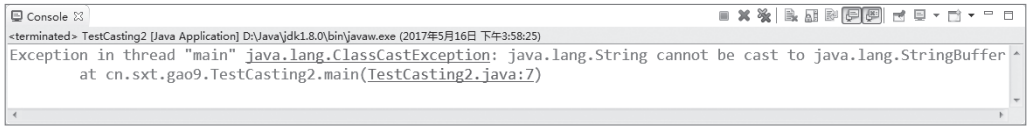



图5-19 示例5-14运行结果

为了避免出现这种异常，可以使用5.1.2节中所学的instanceof运算符进行判断，如示例5-15所示。

【示例5-15】向下转型中使用instanceof

```
public class TestCasting3 {
  public static void main(String[ ] args) {
    Object obj = new String("北京尚学堂");
    if(obj instanceof String){
      String str = (String)obj;
      System.out.println(str.charAt(0));
    }else if(obj instanceof StringBuffer){
      StringBuffer str = (StringBuffer) obj;
      System.out.println(str.charAt(0));
    }
  }
}
```

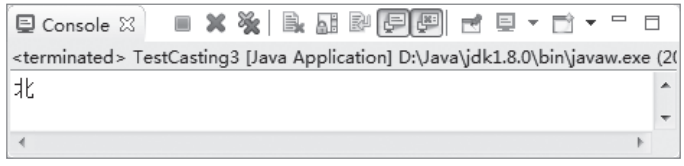



图5-20 示例5-15运行结果

## 5.7 final关键字

final关键字的作用如下：

- 修饰变量：被它修饰的变量不可改变。一旦赋予了初值，就不能被重新赋值。

```
final int MAX_SPEED = 120;
```

- 修饰方法：该方法不可被子类重写，但是可以被重载。

```
 final  void  study(){}
```

- 修饰类：修饰的类不能被继承，如Math、String等。

```
Final class A {}
```

final修饰变量的示例详见第2章示例2-9。

final修饰方法如图5-21所示。

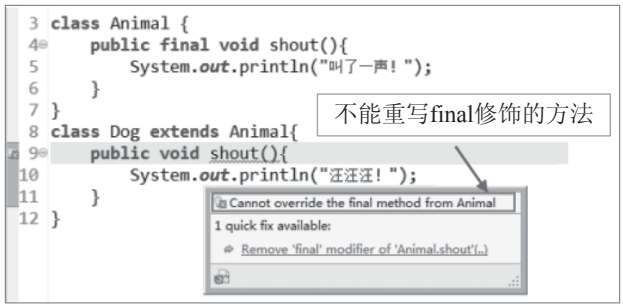



图5-21　final修饰方法

final修饰类如图5-22所示。

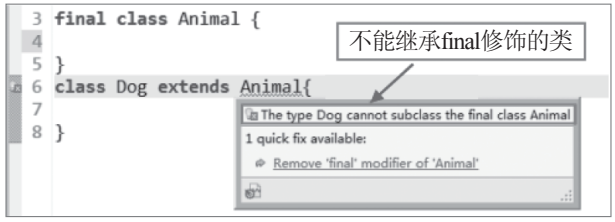



图5-22　final修饰类

## 5.8　抽象方法和抽象类

### 1. 抽象方法

使用abstract修饰的方法，没有方法体，只有声明。它定义的是一种“规范”，就是告诉子类必须要给抽象方法提供具体的实现。

### 2. 抽象类

包含抽象方法的类就是抽象类。抽象类通过abstract方法定义规范，要求子类必须定义具体实现。通过抽象类可以严格限制子类的设计，使子类之间更加通用。

【示例5-16】抽象类和抽象方法的基本用法

```
//抽象类
abstract class Animal {
  abstract public void shout();  //抽象方法
}
class Dog extends Animal {
  //子类必须实现父类的抽象方法,否则编译错误
  public void shout() {
    System.out.println("汪汪汪！");
  }
  public void seeDoor(){
    System.out.println("看门中....");
  }
}
//测试抽象类
public class TestAbstractClass {
  public static void main(String[ ] args) {
    Dog a = new Dog();
    a.shout();
    a.seeDoor();
  }
}
```

抽象类的使用要点如下：

（1）有抽象方法的类只能定义成抽象类。

（2）抽象类不能实例化，即不能用new来实例化抽象类。

（3）抽象类可以包含属性、方法和构造器，但是构造器不能用new来实例化，只能用来被子类调用。

（4）抽象类只能用来被继承。

（5）抽象方法必须被子类实现。

## 5.9 接口interface

接口就是规范，它定义的是一组规则，体现了现实世界中“如果你是……则必须能……”的思想。例如，如果你是天使，则必须能飞；如果你是汽车，则必须能跑；如果你是好人，则必须能干掉坏人；如果你是坏人，则必须欺负好人。

接口的本质是契约，就像法律一样，制定好后大家都要遵守。

面向对象的精髓是对对象的抽象，最能体现这一点的就是接口。为什么人们讨论设计模式时都只针对具备了抽象能力的语言（C++、Java、C#等），就是因为设计模式所研究的实际上就是如何合理抽象的问题。

### 5.9.1 接口的作用

**为什么需要接口？接口和抽象类的区别是什么？**

接口是比“抽象类”还“抽象”的“抽象类”，它可以更加规范地对子类进行约束，

全面、专业地实现了规范和具体实现的分离。

抽象类还提供了某些具体实现，接口不提供任何实现，接口中所有方法都是抽象方法。接口是完全面向规范的，规定了一批类具有的公共方法规范。

从接口实现者的角度看，接口定义了可以向外部提供的服务；从接口调用者的角度看，接口定义了实现者能提供的服务。

接口是两个模块之间通信的标准，是通信的规范。如果能把要设计的模块之间的接口定义好，就相当于完成了系统的设计大纲，剩下的就是添砖加瓦等具体实现了。大家在工作以后，往往就是使用“面向接口”的思想来设计系统的。

接口和实现类不是父子关系，是实现规则的关系。例如，定义一个接口Runnable，Car实现它就能在地上跑，Train实现它也能在地上跑，飞机实现它也能在地上跑。就是说，如果它是交通工具，就一定能跑，但是一定要实现Runnable接口。

普通类、抽象类和接口的区别如下。

- 普通类：具体实现。
- 抽象类：具体实现，规范（抽象方法）。
- 接口：规范。

## 5.9.2 定义和使用接口

接口的声明格式如下：

```
[访问修饰符] interface 接口名 [extends 父接口1,父接口2…]  {
  常量定义;
  方法定义;
}
```

定义接口的详细说明如下。

- 访问修饰符：只能是public或默认设置。
- 接口名：和类名采用相同的命名机制。
- extends：接口可以多继承。
- 常量：接口中的属性只能是常量，总是以public static final修饰，不写也是。
- 方法：接口中的方法只能是public abstract，即使省略，也还是public abstract。

**要点**

- 子类通过implements来实现接口中的规范。
- 接口不能创建实例，但是可用于声明引用变量类型。
- 一个类实现了接口，必须实现接口中所有的方法，并且这些方法只能是public的。
- 在JDK1.7之前的版本中，接口中只能包含静态常量和抽象方法，不能有普通属性、构造器和普通方法。
- 在JDK1.8之后的版本中，接口中包含普通的静态方法。

【示例5-17】接口的使用

```
public class TestInterface {
  public static void main(String[ ] args) {
    Volant volant = new Angel();
    volant.fly();
      System.out.println(Volant.FLY_HIGHT);

      Honest honest = new GoodMan();
      honest.helpOther();
  }
}
/*飞行接口*/
interface Volant {
  int FLY_HIGHT = 100;  //总是:public static final类型的
  void fly();            //总是:public abstract void fly();
}
/*善良接口*/
interface Honest {
  void helpOther();
}
/*Angle类实现飞行接口和善良接口*/
class Angel implements Volant, Honest{
  public void fly() {
    System.out.println("我是天使,飞起来啦! ");
  }
  public void helpOther() {
    System.out.println("扶老奶奶过马路! ");
  }
}
class GoodMan implements Honest {
  public void helpOther() {
    System.out.println("扶老奶奶过马路! ");
  }
}
class BirdMan implements Volant {
  public void fly() {
    System.out.println("我是鸟人,正在飞! ");
  }
}
```

执行结果如图5-23所示。

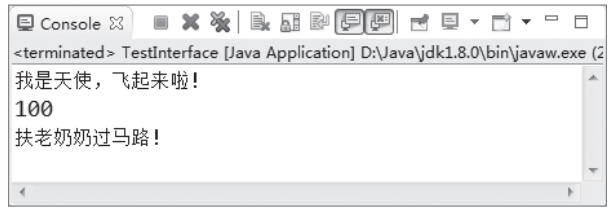



图5-23 示例5-17运行结果

## 5.9.3 接口的多继承

接口完全支持多继承。接口和类的继承类似，子接口扩展某个父接口，将会获得父接

口中所定义的一切。

【示例5-18】接口的多继承

```
interface A {
  void testa();
}
interface B {
  void testb();
}
/*接口可以多继承:接口C继承接口A和B*/
interface C extends A, B {
  void testc();
}
public class Test implements C {
  public void testc() {
  }
  public void testa() {
  }
  public void testb() {
  }
}
```

### 5.9.4 面向接口编程

面向接口编程是面向对象编程的一部分。

为什么需要面向接口编程？软件设计中最难处理的就是需求的复杂变化，需求的变化更多地体现在具体实现上。如果编程围绕具体实现来展开就会陷入“复杂变化”的汪洋大海，软件也就不能最终实现。因此，软件设计必须围绕某种稳定的东西来开展，才能以静制动，实现规范的高质量项目。

接口就是规范，就是项目中最稳定的核心。面向接口编程可以把握住真正核心的东西，使实现复杂多变的需求成为可能。

通过面向接口编程，而不是面向实现类编程，可以大大降低程序模块间的耦合性，提高整个系统的可扩展性和可维护性。

面向接口编程的概念比接口本身的概念要大得多。面向接口编程在设计阶段相对困难，因为在没有编写实现前就要想好接口，否则接口一变就乱套了，所以说设计要比实现难。

**老鸟建议**

接口语法本身非常简单，但是如何真正使用才是大学问，大家需要在后面的项目中反复使用才能体会到。学到此处，能了解基本概念，熟悉基本语法，就是“好学生”了，请继续努力！在工作后，如果大家有闲暇时间再来回顾上面的这段话，相信会有更深入的体会。

# 5.10 内部类

内部类是一种特殊的类，它指的是定义在一个类的内部的类。在实际开发中，为了方便使用外部类的相关属性和方法，通常需要定义一个内部类。

## 5.10.1 内部类的概念

一般情况下，人们把类定义成独立的单元，而在有些情况下，则把一个类放在另一个类的内部来定义，称为内部类（innerclasses）。

内部类可以使用public、default、protected 、private以及static来修饰，而外部顶级类（前面接触的类）则只能使用public和default来修饰。

**注意**

内部类只是一个编译时的概念，一旦编译成功，就会成为完全不同的两个类。对于一个名为Outer的外部类和其内部定义的名为Inner的内部类。编译完成后会出现Outer.class和Outer$Inner.class两个类的字节码文件。所以内部类是相对独立的一种存在，其成员变量/方法名可以和外部类的相同。

**【示例5-19】内部类的展示**

```
/*外部类Outer*/
class Outer {
  private int age = 10;
  public void show(){
    System.out.println(age);//10
  }
  /*内部类Inner*/
  public class Inner {
    //内部类中可以声明与外部类同名的属性与方法
    private int age = 20;
    public void show(){
      System.out.println(age);//20
    }
  }
}
```

示例5-19编译后会产生两个不同的字节码文件，如图5-24所示。

### 1. 内部类的作用

- 内部类提供了更好的封装，只能由外部类直接访问，而不允许同一个包中的其他类直接访问。
- 内部类可以直接访问外部类的私有属性，内部类被当成其外部类的成员。但外部类不能访问内部类的内部属性。
- 接口只是解决了多重继承的部分问题，而内部类使得多重继承的解决方案变得更加完整。

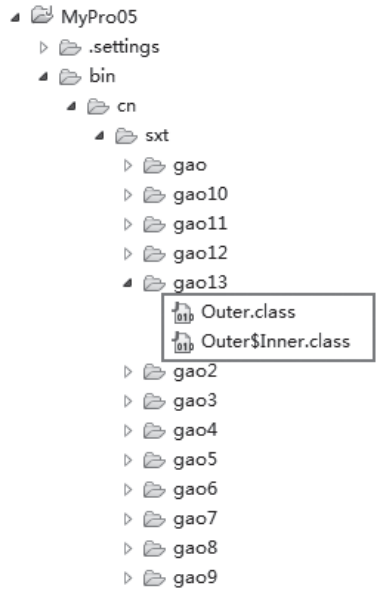



图5-24　内部类编译结果图

### 2. 内部类的使用场合

- 由于内部类具有更好的封装特性，并且可以很方便地访问外部类的属性，所以，在只为外部类提供服务的情况下可以优先考虑使用内部类。
- 使用内部类间接实现多继承。每个内部类都能独立地继承一个类或者实现某些接口，所以无论外部类是否已经继承了某个类或者实现了某些接口，都对内部类没有任何影响。

## 5.10.2　内部类的分类

在Java中内部类主要分为成员内部类（非静态内部类和静态内部类）、匿名内部类和局部内部类。

### 1. 成员内部类

成员内部类可以使用private、default、protected、public进行修饰。 类文件为：外部类$内部类.class。

1）非静态内部类（外部类里使用的非静态内部类和平时使用的其他类没什么不同）

（1）非静态内部类必须寄存在一个外部类对象里。因此，如果有一个非静态内部类对象就一定存在对应的外部类对象。非静态内部类对象单独属于外部类的某个对象。

（2）非静态内部类可以直接访问外部类的成员，但是外部类不能直接访问非静态内部类的成员。

（3）非静态内部类不能有静态方法、静态属性和静态初始化块。

（4）外部类的静态方法、静态代码块不能访问非静态内部类，包括不能使用非静态内部类定义变量，创建实例。

（5）成员变量访问要点如下。

- 内部类里的方法的局部变量：变量名。
- 内部类属性：this.变量名。
- 外部类属性：外部类名.this.变量名。

（6）内部类的访问要点如下。

- 在外部类中定义内部类：new Inner()。
- 在外部类以外的地方使用非静态内部类：Outer.Inner varname = new Outer().new Inner()。

【示例5-20】在内部类中访问成员变量

```
class Outer {
  private int age = 10;
  class Inner {
    int age = 20;
    public void show() {
      int age = 30;
      System.out.println("内部类方法里的局部变量age:" + age);       //30
      System.out.println("内部类的成员变量age:" + this.age);       //20
      System.out.println("外部类的成员变量age:" + Outer.this.age);//10
    }
  }
}
```

【示例5-21】内部类的访问

```
public class TestInnerClass {
  public static void main(String[ ] args) {
    //先创建外部类实例,然后使用该外部类实例创建内部类实例
    Outer.Inner inner = new Outer().new Inner();
    inner.show();
    Outer outer = new Outer();
    Outer.Inner inn = outer.new Inner();
    inn.show();
  }
}
```

执行结果如图5-25所示。

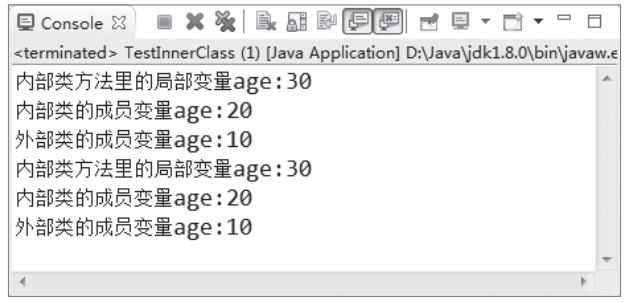



图5-25 示例5-21运行结果

2）静态内部类

（1）静态内部类的定义方式为：

```
Static class ClassName {
//类体
}
```

（2）静态内部类的使用要点如下。

- 一个静态内部类对象存在，并不一定存在对应的外部类对象，因此，静态内部类的实例方法不能直接访问外部类的实例方法。
- 静态内部类可看做是外部类的一个静态成员，因此，外部类的方法中可以通过“静态内部类.名字”的方式访问静态内部类的静态成员，通过new静态内部类()访问静态内部类的实例。

【示例5-22】静态内部类的访问

```
class Outer{
  //相当于外部类的一个静态成员
  static class Inner{
  }
}
public class TestStaticInnerClass {
  public static void main(String[ ] args) {
    //通过 new 外部类名.内部类名() 来创建内部类对象
    Outer.Inner inner =new Outer.Inner();
  }
}
```

## 2. 匿名内部类

匿名内部类适合那种只需要使用一次的类，例如键盘监听操作等。

匿名内部类的语法格式如下：

```
new 父类构造器(实参类表) \实现接口 () {
  //匿名内部类类体!
}
```

【示例5-23】匿名内部类的使用

```
this.addWindowListener(new WindowAdapter(){
  @Override
  public void windowClosing(WindowEvent e) {
    System.exit(0);
    }
  }
);
this.addKeyListener(new KeyAdapter(){
  @Override
  public void keyPressed(KeyEvent e) {
    myTank.keyPressed(e);
    }
```

```
    @Override
    public void keyReleased(KeyEvent e) {
      myTank.keyReleased(e);
      }
    }
);
```

**注意**

- 匿名内部类没有访问修饰符。
- 匿名内部类没有构造器。因为它连名字都没有又何来构造器呢。

### 3. 局部内部类

局部内部类定义在方法内部，作用域只限于本方法。

局部内部类主要是用来解决比较复杂的问题。例如想创建一个类来辅助解决问题，而又不希望这个类是公共可用的，于是就产生了局部内部类。局部内部类和成员内部类一样被编译，只是它的作用域发生了改变，只能在该方法中被使用，离开该方法就会失效。

局部内部类在实际开发中很少应用。

**【示例5-24】方法中的内部类**

```
public class Test2 {
  public void show() {
    //作用域仅限于该方法
    class Inner {
      public void fun() {
        System.out.println("helloworld");
      }
    }
    new Inner().fun();
  }
  public static void main(String[ ] args) {
    new Test2().show();
  }
}
```

执行结果如图5-26所示。

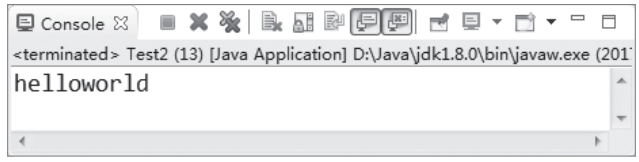



图5-26 示例5-24运行结果

## 5.11 字符串String

String是开发中最常用的类，我们不仅要掌握String类常见的方法，对于String的底层实

现也需要掌握好，不然在工作开发中很容易犯错。

### 5.11.1 String基础

String类又称为不可变字符序列。String位于java.lang包中，Java程序默认导入java.lang包下的所有类。

Java字符串是Unicode字符序列，例如字符串“Java”是由4个Unicode字符“J”“a”“v”“a”组成的。Java没有内置的字符串类型，而是在标准Java类库中提供了一个预定义的类String，每个用双引号括起来的字符串都是String类的一个实例。

【示例5-25】String类的简单使用

```
String e = ""; //空字符串
String greeting = " Hello World ";
```

Java允许使用“+”符号把两个字符串连接起来。

【示例5-26】字符串连接

```
String s1 = "Hello";
String s2 = "World! ";
String s = s1 + s2; //HelloWorld!
```

示例5-26中，“+”符号把两个字符串按给定的顺序连接在一起，并且是完全按照给定的形式。当“+”运算符两侧的操作数中有一个是字符串（String）类型，系统会自动将另一个操作数转换为字符串然后再进行连接。

【示例5-27】“+”连接符的应用

```
int age = 18;
String str = "age is" + age;  //str赋值为"age is 18"
//这种特性通常被用在输出语句中
System.out.println("age  is" + age);
```

### 5.11.2 String类和常量池

在Java的内存分析中，我们经常会听到关于“常量池”的描述，实际上常量池分为以下三种。

#### 1. 全局字符串常量池（String Pool）

全局字符串常量池中存放的内容是在类加载完成后存到String Pool中的，在每个VM中只有一份，存放的是字符串常量的引用值（在堆中生成字符串对象实例）。

#### 2. class文件常量池（Class Constant Pool）

class常量池是在编译时每个class都有的，在编译阶段，它存放的是常量（文本字符串、final常量等）和符号引用。

### 3. 运行时常量池（Runtime Constant Pool）

运行时常量池是在类加载完成之后，将每个class常量池中的符号引用值转存到运行时常量池中。也就是说，每个class都有一个运行时常量池，类在解析之后，将符号引用替换成直接引用，与全局常量池中的引用值保持一致。

【示例5-28】常量池

```
String str1 = "abc";
String str2 = new String("def");
String str3 = "abc";
String str4 = str2.intern();
String str5 = "def";
System.out.println(str1 == str3);//true
System.out.println(str2 == str4);//false
System.out.println(str4 == str5);//true
```

示例5-28经过编译后，在该类的class常量池中存放一些符号引用；当类加载之后，将class常量池中存放的符号引用转存到运行时常量池中；然后经过验证、准备阶段之后，在堆中生成驻留字符串的实例对象（也就是str1所指向的“abc”实例对象）；再将这个对象的引用存到全局String Pool中，也就是String Pool中；最后在解析阶段，要把运行时常量池中的符号引用替换成直接引用，于是直接查询String Pool，保证String Pool里的引用值与运行时常量池中的引用值一致。

回顾示例5-28，现在就很容易解释整个程序的内存分配过程了。首先，在堆中会有一个“abc”实例，全局String Pool中存放着“abc”的一个引用值。在运行第二句的时候会生成两个实例，一个是“def”的实例对象，并且在String Pool中存储一个“def”的引用值，还有一个是new出来的一个“def”的实例对象，与上面那个是不同的实例。在解析str3时查找String Pool，里面有“abc”的全局驻留字符串引用，所以str3的引用地址与之前的那个已存在的相同。str4是在运行的时候调用intern()函数，返回String Pool中“def”的引用值，如果没有就将str2的引用值添加进去。在这里，String Pool中已经有“def”的引用值了，所以返回上面在new str2的时候添加到String Pool中的 “def”引用值。最后，str5在解析时就也是指向存在于String Pool中的“def”的引用值。经过这样一分析，示例5-28的运行结果也就容易理解了。

## 5.11.3 阅读API文档

### 1. 下载API文档

（1）进入下载地址http://www.oracle.com/technetwork/java/javase/documentation/jdk8-doc-downloads-2133158.html，下载API文档，如图5-27所示。

（2）下载成功后，解压下载的压缩文件，然后打开docs、api目录下的index.html文件即可，如图5-28所示。


图5-27　API下载界面

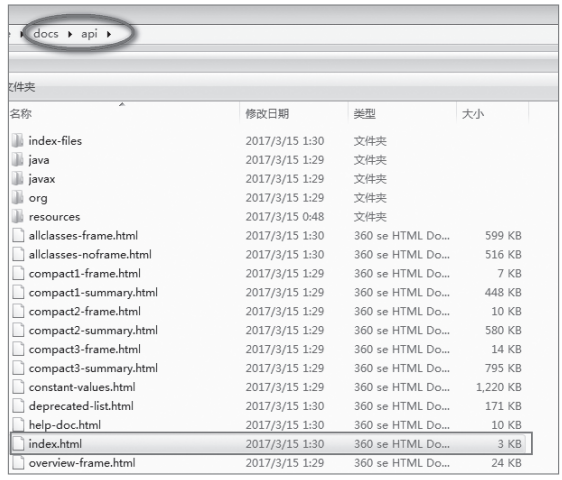



图5-28　打开index.html文件

## 2. 阅读API文档

打开的API文档如图5-29所示。

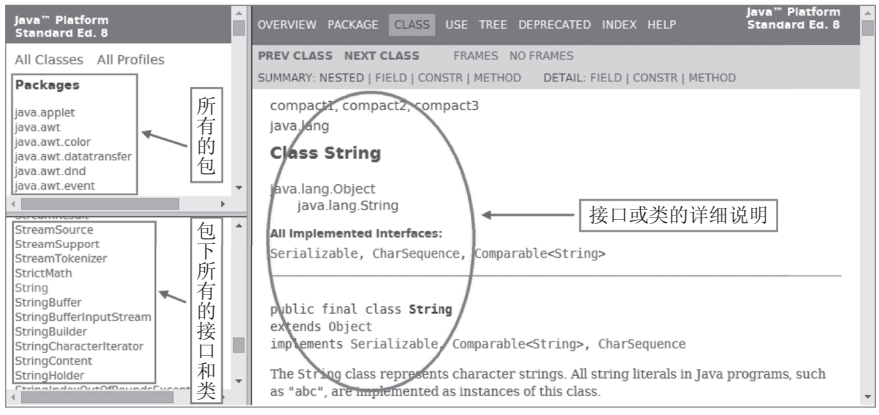



图5-29　API文档

在Eclipse窗口中将鼠标放在类或方法上，即可看到相关的注释说明，再按F2键即可将注释窗口固定，如图5-30所示。

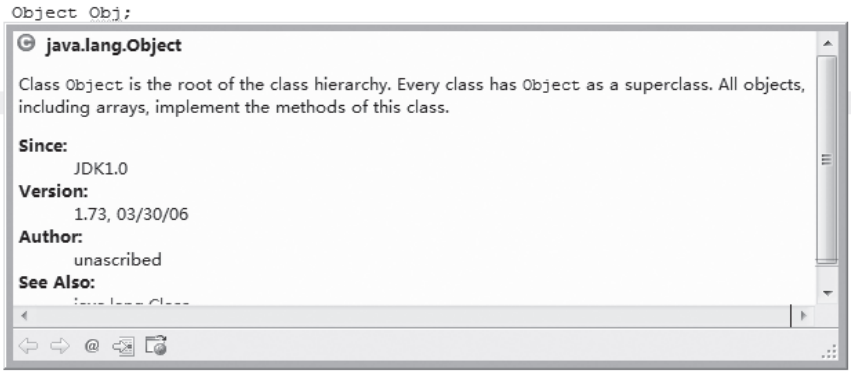



图5-30 Eclipse中的注释说明

## 5.11.4 String类的常用方法

String类是最常使用的类，程序员必须非常熟悉字符串类的方法。表5-2列出了String常用的方法。

表5-2 String类的常用方法

| 方 法 | 解 释 说 明 |
|---|---|
| char charAt(int index) | 返回字符串中第index个字符 |
| boolean equals(String other) | 如果字符串与other相等，返回true；否则，返回false |
| boolean equalsIgnoreCase(String other) | 如果字符串与other相等（忽略大小写），则返回true；否则，返回false |
| int indexOf(String str) | 返回从头开始查找第一个子字符串str在字符串中的索引位置，如果未找到子字符串str，则返回-1 |
| lastIndexOf() | 返回从末尾开始查找第一个子字符串str在字符串中的索引位置，如果未找到子字符串str，则返回-1 |
| int length() | 返回字符串的长度 |
| String replace(char oldChar，char newChar) | 返回一个新串，它是通过用 newChar 替换此字符串中出现的所有oldChar而生成的 |
| boolean startsWith(String prefix) | 如果字符串以prefix开始，则返回true |
| boolean endsWith(String prefix) | 如果字符串以prefix结尾，则返回true |
| String substring(int beginIndex) | 返回一个新字符串，该串包含从原始字符串beginIndex到串尾 |
| String substring(int beginIndex，int endIndex) | 返回一个新字符串，该串包含从原始字符串beginIndex到串尾或endIndex-1的所有字符 |
| String toLowerCase() | 返回一个新字符串，该串将原始字符串中的所有大写字母改成小写字母 |
| String toUpperCase() | 返回一个新字符串，该串将原始字符串中的所有小写字母改成大写字母 |
| String trim() | 返回一个新字符串，该串删除了原始字符串头部和尾部的空格 |

【示例5-29】String类常用方法一

```
public class StringTest1 {
  public static void main(String[ ] args) {
    String s1 = "core Java";
    String s2 = "Core Java";
    System.out.println(s1.charAt(3));              //提取下标为3的字符
    System.out.println(s2.length());               //字符串的长度
    System.out.println(s1.equals(s2));             //比较两个字符串是否相等
    System.out.println(s1.equalsIgnoreCase(s2));   //比较两个字符串(忽略大小写)
    System.out.println(s1.indexOf("Java"));        //字符串s1中是否包含Java
    System.out.println(s1.indexOf("apple"));       //字符串s1中是否包含apple
    String s = s1.replace(' ', '&');               //将s1中的空格替换成&
    System.out.println("result is :" + s);
  }
}
```

执行结果如图5-31所示。

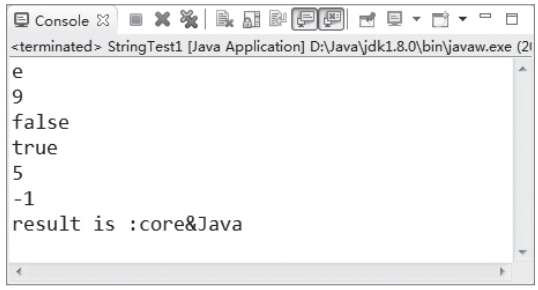



图5-31　示例5-29运行结果

【示例5-30】String类常用方法二

```
public class StringTest2 {
  public static void main(String[ ] args) {
    String s = "";
    String s1 = "How are you?";
    System.out.println(s1.startsWith("How")); //是否以How开头
    System.out.println(s1.endsWith("you"));   //是否以you结尾
    s = s1.substring(4);                      //提取子字符串:从下标为4的开始到字
                                                符串结尾为止
    System.out.println(s);
    s = s1.substring(4, 7);                   //提取子字符串:下标[4,7) 不包括7
    System.out.println(s);
    s = s1.toLowerCase();                     //转小写
    System.out.println(s);
    s = s1.toUpperCase();                     //转大写
    System.out.println(s);
    String s2 = "  How old are you!! ";
    s = s2.trim();                            //去除字符串首尾的空格。注意:中间
                                                的空格不能去除
    System.out.println(s);
    System.out.println(s2);                   //因为String是不可变字符串,所以
                                                s2不变
  }
}
```

执行结果如图5-32所示。

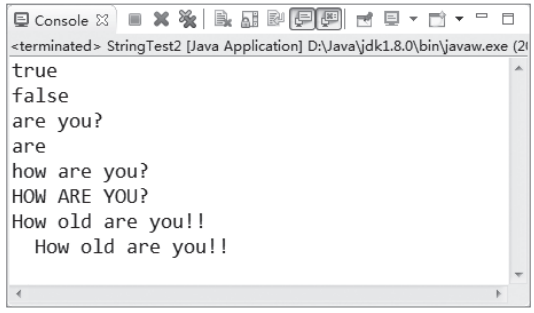



图5-32 示例5-30运行结果

## 5.11.5 字符串相等的判断

equals方法用来检测两个字符串的内容是否相等。如果字符串s和t内容相等，则s.equals(t)返回true，否则返回false。

要测试两个字符串除了大小写不同外是否相等，需要使用equalsIgnoreCase方法。

判断字符串是否相等不要使用“==”。

【示例5-31】忽略大小写的字符串比较

```
"Hello".equalsIgnoreCase("hellO");//true
```

【示例5-32】字符串的比较——“= =”与equals()方法

```
public class TestStringEquals {
  public static void main(String[ ] args) {
    String g1 = "北京尚学堂";
    String g2 = "北京尚学堂";
    String g3 = new String("北京尚学堂");
    System.out.println(g1 == g2);       //true,指向同样的字符串常量对象
    System.out.println(g1 == g3);       //false,g3是新创建的对象
    System.out.println(g1.equals(g3)); //true,g1和g3里的字符串内容是一样的
  }
}
```

执行结果如图5-33所示。

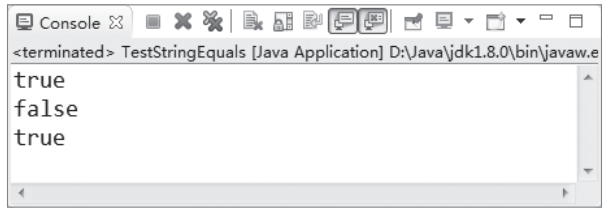



图5-33 示例5-32运行结果

示例5-32的内存分析如图5-34所示。

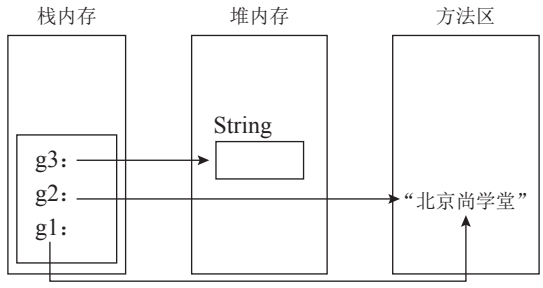



图5-34　示例5-32内存分析图

## 5.12　设计模式相关知识

面向对象基本概念学完后，大多数初学者只有一些基本的概念。如果想加深对这些概念的理解，大家可以学习一点常用的设计模式知识，体验一下面向对象编程思维和面向接口编程思维。设计模式有23种，一般只学习在工作中最常用的几种即可。

### 5.12.1　开闭原则

开闭原则（Open-Closed Principle）就是指让设计的系统对扩展开放，对修改封闭。

- 对扩展开放是指应对需求变化要灵活，在增加新功能时不需要修改已有的代码，增加新代码即可。
- 对修改关闭是指核心部分经过精心设计后，不再因为需求变化而改变。

在实际开发中，我们虽无法完全做到，但应尽量遵守开闭原则。

### 5.12.2　相关设计模式

学完本章，优秀的同学可以开展第18章核心设计模式的学习，进一步体验设计的快乐，同时加深对面向对象、面向接口编程的理解。

### 本章总结

（1）高级语言可分为面向过程和面向对象两大类。

- 面向过程与面向对象都是解决问题的思维方式，都是代码组织的方式。
- 解决简单问题可以使用面向过程的方法。
- 解决复杂问题，在宏观上使用面向对象的方法来把握，在微观上仍使用面向过程的处理方法。

（2）对象和类是从特殊到一般的关系，是从具体到抽象的关系。

（3）栈内存：

- 每个线程私有，不能实现线程间的共享。

- 局部变量放置于栈中。
- 栈由系统自动分配，速度快。栈是一个连续的内存空间。

（4）堆内存：

- 用于放置new出来的对象。
- 堆是一个不连续的内存空间，分配灵活，但速度慢。

（5）方法区：

- 被所有线程所共享。
- 用来存放程序中永远不变或唯一的内容（如类代码信息、静态变量、字符串常量）。

（6）属性用于定义该类或该类对象包含的数据或者静态属性。属性的作用范围是整个类体，Java使用默认的值对其初始化。

（7）方法用于定义该类或该类实例的行为特征和功能实现。方法是对类和对象行为特征的抽象。

（8）构造器又叫作构造方法，用于构造该类的实例。Java通过new关键字来调用构造器，从而返回该类的实例，是一种特殊的方法。

（9）垃圾回收机制：

- 程序员无权调用垃圾回收器。
- 程序员可以通过System.gc()通知垃圾回收器（Garbage Collection，GC）运行，但是Java规范并不能保证立刻运行。
- finalize方法是Java用来释放对象或资源的方法，但是应尽量少用。

（10）方法的重载是指一个类中可以定义有相同名称但参数不同的多个方法，调用时，会根据不同的参数表选择对应的方法。

（11）this关键字的作用：

- 让类中的一个方法访问该类的另一个方法或属性。
- 使用this关键字调用重载构造器，可以避免相同的初始化代码。this关键字只能在构造器中用，并且必须位于构造器的第一句。

（12）static关键字：

- 在类中，用static声明的成员变量为静态成员变量，也称为类变量。
- 用static声明的方法为静态方法。
- 可以通过对象引用或类名（不需要实例化）访问静态成员。

（13）package的作用：

- 可以解决类之间的重名问题。
- 便于管理类，令合适的类位于合适的包中。

（14）import的作用：

通过import可以导入其他包中的类，从而在本类中直接通过类名来调用这些类。

（15）super关键字的作用：

super是直接对父类对象的引用，可以通过super来访问父类中被子类覆盖的方法或属性。

（16）面向对象的三大特征为继承、封装和多态。

（17）Object类是所有Java类的根基类。

（18）访问权限控制符：范围由小到大分别是private、default、protected和public。

（19）“引用变量名 instanceof 类名”可用来判断该引用类型变量所“指向”的对象是否属于该类或该类的子类。

（20）final关键字可以修饰变量、方法和类。

（21）抽象类是一种模板模式。抽象类为所有子类提供了一个通用模板，子类可以在这个模板基础上进行扩展，使用abstract修饰。

（22）使用abstract修饰的方法为抽象方法，必须被子类实现，除非子类也是抽象类。

（23）使用interface声明接口时：

- 从接口的实现者角度看，接口定义了可以向外部提供的服务。
- 从接口的调用者角度看，接口定义了实现者能提供的服务。

（24）内部类分为成员内部类、匿名内部类和局部内部类。

（25）String位于java.lang包中，Java程序默认导入java.lang包。

（26）字符串的比较：“==”与equals()方法的区别。

## 本章作业

### 一、选择题

1. 使用权限修饰符（　　）修饰的类的成员变量和成员方法，可以被当前包中所有类访问，也可以被它的子类（同一个包以及不同包中的子类）访问（选择一项）。

A. public　　B. protected

C. Default　　D. Private

2. 以下关于继承条件下构造器执行过程的代码的执行结果是（　　）（选择一项）。

```
class Person {
  public Person() {
    System.out.println("execute Person()");
  }
}
class Student extends Person {
  public Student() {
    System.out.println("execute Student() ");
  }
}
class PostGraduate extends Student {
  public PostGraduate() {
    System.out.println("execute PostGraduate()");
  }
}
public class TestInherit {
  public static void main(String[ ] args) {
```

```
    new PostGraduate();
  }
}
```

A. execute Person()
   execute Student()
   execute PostGraduate()

B. execute PostGraduate()

C. execute PostGraduate()
   execute Student()
   execute Person()

D. 没有结果输出

3. 编译运行如下Java代码，输出结果是（　　）（选择一项）。

```
class Base {
  public void method(){
    System.out.print ("Base method");
  }
}
class Child extends Base{
  public void methodB(){
    System.out.print ("Child methodB");
  }
}
class Sample {
  public static void main(String[ ] args) {
    Base base= new Child();
    base.methodB();
  }
}
```

A. Base method

B. Child methodB

C. Base method
   Child methodB

D. 编译错误

4. 在Java中，以下哪些关于abstract关键字的说法是正确的？（　　）（选择两项）

A. abstract类中可以没有抽象方法　　B. abstract类的子类也可以是抽象类

C. abstract方法可以有方法体　　D. abstract类可以创建对象

5. 在Java接口中，下列选项中属于有效方法声明的是（　　）（选择二项）。

A. public void aMethod();　　B. final void aMethod();

C. void aMethod();　　D. private void aMethod();

## 二、简答题

1. private、default、protected、public四个权限修饰符的作用。

2. 继承条件下子类构造器的执行过程。

3. 什么是向上转型和向下转型。

4. final和abstract关键字的作用。

5. ==和equals()的联系和区别。

**三、编码题**

1. 编写应用程序，创建类的对象，分别设置圆的半径、圆柱体的高，计算并分别显示圆半径、圆面积、圆周长、圆柱体的体积。

实现思路及关键代码：

（1）编写一个圆类Circle，使该类拥有：

- 一个成员变量；

```
radius(私有,浮点型);              //存放圆的半径
```

- 两个构造器（无参、有参）；
- 三个成员方法。

```
double getArea()                  //获取圆的面积
double getPerimeter()             //获取圆的周长
void show()                       //将圆的半径、周长、面积输出到屏幕
```

（2）编写一个圆柱体类Cylinder，它继承于上面的Circle类，并拥有：

- 一个成员变量；

```
double hight(私有,浮点型);        //圆柱体的高;
```

- 构造器；
- 成员方法。

```
double getVolume()                //获取圆柱体的体积
void showVolume()                 //将圆柱体的体积输出到屏幕
```

2. 编写程序实现乐手弹奏乐器。乐手可以弹奏不同的乐器从而发出不同的声音，弹奏的乐器包括二胡、钢琴和琵琶。

实现思路：

（1）定义乐器类Instrument，包括方法makeSound()。

（2）定义乐器类的子类：二胡Erhu、钢琴Piano和小提琴Violin。

（3）定义乐手类Musician，可以弹奏各种乐器play(Instrument i)。

（4）定义测试类，给乐手不同的乐器让他弹奏。

3. 编写程序描述影、视、歌三栖艺人。需求说明：请使用面向对象的思想，设计自定义类，描述影、视、歌三栖艺人。

实现思路：

（1）分析影、视、歌三栖艺人的特性：可以演电影，可以演电视剧，可以唱歌。

（2）定义多个接口描述特性：

- 演电影的接口，方法——演电影；
- 演电视剧的接口，方法——演电视剧；
- 唱歌的接口，方法——唱歌。

（3）定义艺人类实现多个接口。

程序运行结果如图5-35所示。

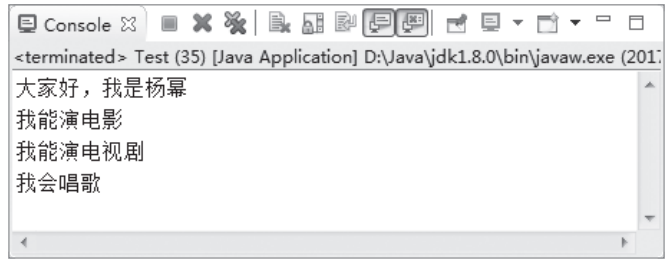



图5-35 编码题3运行结果