

北京大学信息技术系列教材

Java 程序设计

主编 蔡翠平

编著 唐大仕



清华大学出版社

北方交通大学出版社

• 北 京 •

内 容 简 介

本书详细介绍了 Java 程序设计的基本环境、概念和方法。内容分为三个部分：第一部分介绍了 Java 语言基础，包括数据、控制结构、数组、类、包、对象、接口等；第二部分介绍了 Java 深入知识，包括传值调用、虚方法调用、异常处理、工具类与算法；第三部分是 Java 的应用，包括线程、流式文件、AWT 及 Swing 图形用户界面，以及 Java 在网络、多媒体、数据库等方面的应用。

本书内容详尽，循序渐进，在介绍编程技术的同时，还着重讲解了有关面向对象程序设计的基本概念和方法。书中提供了丰富的典型实例，具有可操作性，便于读者的学习与推广应用。各章附有大量习题，便于读者思考和复习。

本书内容和组织方式立足于高校教学教材的要求，同时可作为计算机技术的培训教材，还可作为 Sun 认证考试（SCJP）的考试用书。

版权所有，翻印必究。

本书封面贴有清华大学出版社激光防伪标签，无标签者不得销售。

图书在版编目（CIP）数据

Java 程序设计/唐大仕编著. —北京：北方交通大学出版社，2003.3

（北京大学信息技术系列教材）

ISBN 7-81082-099-0

I .J… II.唐… III.Java 语言—程序设计—高等学校—教材 IV.TP312

中国版本图书馆 CIP 数据核字（2003）第 005216 号

责任编辑：谭文芳

印 刷 者：北京东光印刷厂

出版发行：北方交通大学出版社 邮编：100044 电话：010-51686045，62237564

清 华 大 学 出 版 社 邮编：100084

经 销：各地新华书店

开 本：787×1092 1/16 印张：23 字数：556 千字

版 次：2003 年 4 月第 1 版 2003 年 4 月第 1 次印刷

印 数：6000 册 定价：29.00 元

北京大学信息技术系列教材

编委会成员

主 任：蔡翠平

副主任：吕凤翥

委 员：(以姓氏笔画为序)

尹会滨 许 彦 吕凤翥 任吉治 张亦工

吴筱萌 尚俊杰 林洁梅 周宏滔 陈 虎

赵 文 赵丹群 徐尔贵 唐大仕 蔡翠平

缪 蓉 潘 曦

目 录

第 1 章	Java 语言与面向对象的程序设计	(1)
1.1	Java 语言简介	(1)
1.1.1	Java 语言出现的背景、影响及应用前景	(1)
1.1.2	Java 的特点	(2)
1.1.3	Java 和 C、C++	(4)
1.2	面向对象程序设计	(5)
1.2.1	面向对象概述	(5)
1.2.2	对象、类与实体	(6)
1.2.3	对象的状态与行为	(7)
1.2.4	对象的关系	(7)
1.2.5	面向对象的软件开发过程	(8)
	习题	(9)
第 2 章	简单的 Java 程序	(11)
2.1	Application 与 Applet 程序	(11)
2.1.1	Application 程序	(11)
2.1.2	Applet 程序	(12)
2.1.3	Java 程序的基本构成	(13)
2.2	程序的编辑、编译与运行	(15)
2.2.1	Java 工具包 JDK	(15)
2.2.2	Application 的编辑、编译与运行	(15)
2.2.3	Applet 的编辑、编译与运行	(19)
2.2.4	使用 jar 打包程序	(21)
2.3	Java 程序中的基本输入与输出	(21)
2.3.1	字符界面的输入与输出	(22)
2.3.2	Applet 图形界面输入与输出	(24)
2.3.3	Java Application 图形界面输入与输出	(26)
2.3.4	同时作为 Application 与 Applet 的程序	(27)
2.4	Java 集成开发环境	(28)
2.4.1	几种辅助工具的使用	(29)
2.4.2	几种集成工具的使用	(31)
	习题	(34)

第 3 章 数据运算、流控制和数组	(36)
3.1 数据类型、变量与常量	(36)
3.1.1 数据类型	(36)
3.1.2 标识符	(37)
3.1.3 常量	(37)
3.1.4 变量	(38)
3.1.5 程序的书写与注释	(39)
3.2 运算符与表达式	(41)
3.2.1 算术运算符	(41)
3.2.2 关系运算符	(43)
3.2.3 逻辑运算符	(43)
3.2.4 位运算符	(44)
3.2.5 赋值与强制类型转换	(48)
3.2.6 条件运算符	(49)
3.2.7 表达式及运算的优先级、结合性	(50)
3.3 流程控制语句	(50)
3.3.1 结构化程序设计的三种基本流程	(51)
3.3.2 简单语句	(51)
3.3.3 分支语句	(51)
3.3.4 循环语句	(58)
3.3.5 跳转语句	(64)
3.4 数组	(67)
3.4.1 一维数组	(67)
3.4.2 多维数组	(70)
3.4.3 System.arraycopy()方法	(73)
习题	(73)
第 4 章 类、包和接口	(75)
4.1 类、域、方法	(75)
4.1.1 定义类中的域和方法	(75)
4.1.2 构造方法与对象的创建	(77)
4.1.3 使用对象	(78)
4.1.4 方法的重载	(79)
4.1.5 this 的使用	(80)
4.2 类的继承	(81)
4.2.1 派生子类	(81)
4.2.2 域的继承与隐藏、添加	(82)
4.2.3 方法的继承、覆盖与添加	(83)
4.2.4 super 的使用	(84)

4.2.5	父类对象与子类对象的转换	(85)
4.3	包	(88)
4.3.1	package 语句	(88)
4.3.2	import 语句	(88)
4.3.3	编译和运行包中的类	(89)
4.3.4	CLASSPATH	(90)
4.4	访问控制符	(90)
4.4.1	成员的访问控制符	(90)
4.4.2	类的访问控制符	(95)
4.4.3	setor 与 getor	(96)
4.4.4	构造方法的隐藏	(96)
4.5	非访问控制符	(96)
4.5.1	static	(97)
4.5.2	final	(100)
4.5.3	abstract	(101)
4.5.4	其他修饰符	(103)
4.5.5	一个应用模型——单子	(104)
4.6	接口	(105)
4.6.1	接口的概念	(105)
4.6.2	定义接口	(106)
4.6.3	实现接口	(107)
4.6.4	对接口的引用	(108)
习题		(109)
第 5 章	深入理解 Java 语言	(111)
5.1	变量及其传递	(111)
5.1.1	基本类型变量与引用型变量	(111)
5.1.2	域变量与局部变量	(112)
5.1.3	变量的传递	(113)
5.1.4	变量的返回	(114)
5.2	多态与虚方法调用	(114)
5.2.1	上溯造型	(115)
5.2.2	虚方法调用	(116)
5.2.3	动态类型确定	(118)
5.3	对象构造与初始化	(120)
5.3.1	调用本类或父类的构造方法	(120)
5.3.2	构造方法的执行过程	(122)
5.3.3	构建方法内部调用的方法的多态性	(123)
5.4	对象清除与垃圾回收	(125)

5.4.1	对象的自动清除	(125)
5.4.2	System.gc()方法	(126)
5.4.3	finalize()方法	(126)
5.5	内部类与匿名类	(128)
5.5.1	内部类	(128)
5.5.2	方法中的内部类及匿名类	(132)
习题	(134)
第 6 章	异常处理	(135)
6.1	异常处理	(135)
6.1.1	异常的概念	(135)
6.1.2	捕获和处理异常	(137)
6.1.3	应用举例	(140)
6.2	创建用户自定义异常类	(143)
习题	(144)
第 7 章	工具类及常用算法	(146)
7.1	Java 语言基础类	(146)
7.1.1	Java 基础类库	(146)
7.1.2	Object 类	(148)
7.1.3	基本数据类型的包装类	(152)
7.1.4	Math 类	(153)
7.1.5	System 类	(154)
7.2	字符串	(155)
7.2.1	String 类	(155)
7.2.2	StringBuffer 类	(159)
7.2.3	StringTokenizer 类	(161)
7.3	集合类	(162)
7.3.1	集合与 Collection API	(162)
7.3.2	Set 接口及 HashSet 类	(163)
7.3.3	List 接口及 ArrayList, Vector 类	(164)
7.3.4	Iterator 及 Enumeration	(165)
7.3.5	Map 接口及 Hashtable 类	(167)
7.4	向量、堆栈、队列	(168)
7.4.1	Vector 向量	(168)
7.4.2	Stack 堆栈	(171)
7.4.3	LinkedList 及队列	(173)
7.5	排序与查找	(174)
7.5.1	Arrays 类	(174)

7.5.2	Collections 类	(176)
7.5.3	冒泡排序	(178)
7.5.4	选择排序	(179)
7.5.5	快速排序	(180)
7.6	遍试、迭代和递归	(183)
7.6.1	遍试	(183)
7.6.2	迭代	(185)
7.6.3	递归	(188)
	习题	(193)
第 8 章	Java 的多线程	(196)
8.1	线程及其创建	(196)
8.1.1	Java 中的线程	(196)
8.1.2	创建线程对象的两种方法	(197)
8.1.3	多线程	(199)
8.1.4	应用举例	(201)
8.2	线程的调度	(205)
8.2.1	线程的状态与生命周期	(205)
8.2.2	线程调度与优先级	(206)
8.2.3	对线程的基本控制	(208)
8.3	线程的同步与共享	(211)
8.3.1	synchronized 关键字	(211)
8.3.2	线程间的同步控制	(214)
	习题	(219)
第 9 章	流、文件及基于文本的应用	(220)
9.1	流式输入与输出	(220)
9.1.1	字节流与字符流	(220)
9.1.2	节点流和处理流	(222)
9.1.3	标准输入和标准输出	(225)
9.1.4	应用举例	(226)
9.2	文件及目录	(229)
9.2.1	文件与目录管理	(229)
9.2.2	文件输入与输出流	(231)
9.2.3	RandomAccessFile 类	(234)
9.3	基于文本的应用	(235)
9.3.1	Java Application 命令行参数	(235)
9.3.2	环境参数	(236)
9.3.3	处理 Deprecated 的 API	(237)

习题	(239)
第 10 章 图形用户界面	(240)
10.1 AWT 组件	(240)
10.1.1 图形用户界面概述	(240)
10.1.2 AWT 组件分类	(241)
10.1.3 Component 的方法	(244)
10.2 布局管理	(245)
10.2.1 FlowLayout	(245)
10.2.2 BorderLayout	(246)
10.2.3 CardLayout	(247)
10.2.4 GridLayout	(249)
10.2.5 GridBagLayout	(250)
10.2.6 通过嵌套来设定复杂的布局	(250)
10.3 事件处理	(251)
10.3.1 事件及事件监听器	(251)
10.3.2 事件监听器的注册	(253)
10.3.3 事件适配器	(258)
10.3.4 内部类及匿名类在事件处理中的应用	(259)
10.4 常用组件的使用	(262)
10.4.1 标签、按钮与动作事件	(262)
10.4.2 文本框、文本区域与文本事件	(264)
10.4.3 单、复选按钮，列表与选择事件	(265)
10.4.4 调整事件与滚动条	(269)
10.4.5 鼠标、键盘事件与画布	(271)
10.4.6 Frame 与窗口事件	(273)
10.4.7 Panel 与容器事件	(274)
10.4.8 组件事件、焦点事件与对话框	(276)
10.5 绘图、图形和动画	(277)
10.5.1 绘制图形	(277)
10.5.2 显示文字	(281)
10.5.3 控制颜色	(282)
10.5.4 显示图像	(283)
10.5.5 实现动画效果	(283)
10.6 Applet	(285)
10.6.1 Applet 的基本工作原理	(286)
10.6.2 Applet 类	(286)
10.6.3 HTML 文件参数传递	(288)
10.6.4 Applet 的局限	(289)

10.7	SwingGUI 组件	(295)
10.7.1	Swing 的特点	(295)
10.7.2	几种 Swing 组件介绍	(296)
10.8	基于 GUI 的应用程序	(299)
10.8.1	使用可视化设计工具	(299)
10.8.2	菜单的定义与使用	(299)
10.8.3	菜单、工具条及对话框的应用	(303)
	习题	(308)
第 11 章	网络、多媒体和数据库编程	(310)
11.1	Java 网络编程	(310)
11.1.1	使用 URL	(310)
11.1.2	用 Java 实现底层网络通信	(312)
11.1.3	实现多线程服务器程序	(314)
11.1.4	Java 的 Email 编程	(323)
11.2	多媒体编程	(329)
11.2.1	在 Applet 中获取声音与图像	(329)
11.2.2	Java 图像编程	(332)
11.2.3	Java 声音编程	(334)
11.3	Java 数据库编程	(335)
11.3.1	Java 访问数据库的基本步骤	(335)
11.3.2	使用 JTable 显示数据表	(339)
11.4	J2EE 及 J2ME 简介	(343)
11.4.1	J2EE 简介	(343)
11.4.2	J2ME 简介	(345)
	习题	(347)
	参考文献	(348)

前 言

在程序设计的教学中，选择一种适合的语言是十分重要的。比较多种程序设计语言，笔者认为 Java 具有如下突出的优点。

其一，Java 是面向对象的语言，与现代面向对象的设计与分析的软件工程相一致，也是当前的主流程序设计语言之一。

其二，简单易学。其中的数据类型、数据运算、程序控制结构等基本概念对于任何语言都是一致的；而其语法相对于 C++ 等语言而言更简单，更容易掌握。

其三，Java 语言就其本身而言支持一些高级特性，如多线程、异常处理、自动垃圾回收等，这些特性使 Java 成为极优秀的语言之一。

最后，Java 具有广泛的用途。Java 具有跨平台的特点，在各种平台上都有应用，它还可以有效地进行数据库、多媒体及网络的程序设计。不仅如此，Java 还与 JavaScript 十分相似，而后者可以广泛用于办公软件、网页设计、网络服务程序等方面。

综上所述，Java 是特别适合于程序设计学习的基础语言。

对于学习者而言，选择一本好的教材至关重要。现在市面上有关 Java 的书不少，但适合于教学、自学的书却不多见。笔者基于多年程序设计语言的教学经验，结合个人的软件开发实践，力图使本书突出以下特色。

1. 对 Java 语言的基础知识，包括数据类型、流程控制、类的封装与继承、多态、虚方法调用、传值调用等进行系统讲解，让学习者知其然，并知其所以然。

2. 对 Java 中的类库中的基本类，包括 Math、字符串、集合进行详细讲解，以利于学习者打下牢固的基础。

3. 对 Java 中的基本应用，包括 I/O、文本界面、图形界面等，精选大量典型而实用的例子，力图使学习者触类旁通，举一反三。

4. 对一些高级应用，如数据库编程、网络编程、多媒体编程等内容，介绍了其概念、原理，以利于学习者能了解 Java 的实际应用及最新发展。

5. 在讲解语言的同时，介绍它所采用的面向对象技术的基础理论、主要原则和思维方法，同时介绍在 Java 软件工程中常用的 UML 工具。

6. 在讲解、举例时充分考虑到各个层次的需要，力求语言简洁，内容循序渐进。同时，考虑到部分读者有参加 Sun 的 SCJP 认证考试的需要，本书提供了较多的习题。

本书在内容安排上，大致可以分为三部分：第一部分介绍了 Java 语言基础，包括数据、控制结构、数组、类、包、对象、接口等；第二部分介绍了 Java 深入知识，包括传值调用、虚方法调用、异常处理、工具类与算法；第三部分是 Java 的应用，包括线程、流式文件、AWT 及 Swing 图形用户界面，以及 Java 在网络、多媒体、数据库等方面的应用。

本教材的内容和组织方式适合作为高等学校各专业的计算机程序设计课程的教材，或者作为计算机技术的培训教材，也可以作为 Java 认证考试的考试用书。

书中存在的缺点和不足，恳请读者批评指正。

唐大仕

2003 年 4 月

于北京大学信息科学技术学院

北京大学信息技术系列教材

序 言

人类已进入21世纪，科学技术突飞猛进，知识经济初见端倪，特别是信息技术和网络技术的迅速发展和广泛应用，对社会的政治、经济、军事、科技和文化等领域产生越来越深刻的影响，也正在改变着人们的工作、生活、学习和交流方式。信息的获取、处理、交流和应用能力，已经成为人们最重要的能力之一。培养一大批掌握和应用现代信息技术和网络技术的人才，在全球信息化的发展中占据主动地位，不仅是经济和社会发展的需要，也是计算机和信息技术教育者的历史责任。

加入WTO，意味着我国要在同一个网络平台上参与国际竞争，同世界接轨。这对我们既是一个机遇，也是一个挑战。为此我们必须加强全民的信息技术教育，以提高国民的整体素质，抓住国际大环境给我国经济腾飞带来的难得机遇，迎接挑战。

教育部提出，要在全国的中小学中逐步开设信息技术必修课，从小培养获取、分析、处理、发布和应用信息的能力和素养，在条件成熟时，考虑作为普通高校招生考试的科目。国家经贸委也提出，要像抓3年改革和脱困的两大目标那样，把企业管理信息化建设作为新世纪经贸工作的历史性任务抓紧、抓好，推进企业应用计算机管理软件和网络信息技术，用3年左右的时间，在国家重点企业中建立完善的企业信息管理系统。

为了适应这个大的形势，满足各大专院校非计算机专业学生和社会各阶层从事信息技术和急需掌握信息技术人们的需要，我们组织编写了这套《北京大学信息技术系列教材》。目的是让更多的人以最快的速度掌握计算机信息技术，学会运用国际互联网络平台，不断提高自身素质和专业水平，在传统产业升级、实现跨越式发展中更好地展示自己的才能，为祖国的现代化建设服务。

本系列教材包括《计算机信息技术基础》、《计算机网络应用技术》、《办公自动化软件》、《多媒体应用技术》、《网络程序设计——ASP》、《数据库技术——SQL》、《Visual Basic程序设计》、《Visual FoxPro程序设计》、《C++语言程序设计》、《网页制作技术》、《从HTML到XML》、《计算机局域网实用技术》、《Java程序设计》等。随着信息技术的发展和读者的需要，我们还将不断对这一系列教材进行补充或增删，以期形成读者欢迎的动态系列教材。此系列教材可作为大专院校非计算机专业信息技术普及教材，也可供社会各种信息技术培训班选用。

本系列教材具有以下编写特点：

1. 适合不同层次的读者选用

此系列教材从内容上讲，跨度较大，从计算机基础知识一直到动态网站制作，这样可以满足不同领域和不同层次的读者需要，读者可以根据自己的水平像吃自助餐一样自主选用。

2. 选材超前，出版周期短

目前,计算机图书市场火爆,需求旺盛,但是,选一本合适的教材又非易事。其原因之一一是读者急需使用的高版本软件对应的书上市甚少。造成这种现象的原因有三:一是信息技术发展速度太快;二是选材没有注意超前量;三是出版周期太长。鉴于以上原因,本系列教材在内容上尽量注意超前量,如每一个软件必须选择当前最高版本。例如:动态网站制作我们选择当前流行的ASP技术和SQL网上数据库以及VB编程技术;在保证书稿质量的前提下尽量缩短出版周期。其目的都是为了适应信息技术的飞速发展,满足读者的需要。

3. 实用性强

本系列教材的主要对象是非计算机专业人员,因此,在内容上强调实用,尽量不涉及高深的与软件使用无关的理论问题。比如《多媒体应用技术》,作者着重阐述多媒体信息的获取、处理、传输、保存、制作等实用技术,不涉及多媒体的理论问题。又如《计算机局域网实用技术》,作者重点介绍局域网的构架、服务器的安装、各种网上信息服务的建立以及网络安全管理方面的内容,读者可按照书中所讲的内容自己独立构建局域网。

4. 充分体现案例教学

在本系列丛书中读者会发现,凡是操作型软件都是以一个案例为主线进行阐述,这是本系列书作者多年来在教学第一线经验的总结。案例教学引人入胜,易理解,易掌握,能使读者举一反三,技术掌握扎实。

5. 写作风格通俗易懂

介绍每一个软件开门见山,语言简明扼要,重点突出,难点翔实编写,同一功能决不重复。并在每章中附有习题,有的例题配有光盘,适合自学。

参加本系列教材编写的作者都是在大学从事信息技术课一线教学的中、青年教师,他们都有极强的敬业精神,本系列教材凝聚了他们多年丰富的教学经验和心血。

本系列教材得到了北京大学教育学院教育技术系各位老师和北京大学信息管理系余锦凤教授的支持和帮助,在此表示诚挚的感谢。

由于本系列教材从策划到出版仅仅用了不到一年的时间,编写者又都担负着繁重的教学任务,在时间紧、任务重的情况下,肯定有不少不尽人意之处,诚挚接受广大读者的批评、指正。

蔡翠平

2003年4月于北京大学

第 1 章 Java 语言与面向对象的程序设计

Java 语言是当今流行的网络编程语言，它的面向对象、跨平台、分布应用等特点给编程人员带来了一种崭新的计算概念，使 WWW 从最初的单纯提供静态信息发展到现在的提供各种各样的动态服务，产生了巨大的变化。Java 不仅能够编写小应用程序实现嵌入网页的声音和动画功能，而且还能够应用于独立的大中型应用程序，其强大的网络功能能够把整个 Internet 作为一个统一的运行平台，极大地拓展了传统单机或 Client/Server 模式应用程序的外延和内涵。自 1995 年正式问世以来，Java 已经逐步从一种单纯的计算机高级编程语言发展为一种重要的 Internet 平台，并进而引发、带动了 Java 产业的发展壮大，成为当今计算机业界不可忽视的力量和重要的发展潮流与方向。

1.1 Java 语言简介

1.1.1 Java 语言出现的背景、影响及应用前景

1991 年，SUN MicroSystem 公司的 Jame Gosling，Bill Joe 等人，为在电视机、控制烤面包箱等家用消费类电子产品上进行交互式操作而开发了一个名为 Oak（一种橡树的名字）的软件，但当时并没有引起人们的注意，直到 1994 年下半年，Internet 的迅猛发展，WWW 的快速增长，促进了 Java 语言研制的进展，使得它逐渐成为 Internet 上受欢迎的开发与编程语言，一些著名的计算机公司纷纷购买了 Java 语言的使用权，如 Microsoft，IBM，Netscape，Novell，Apple，DEC，SGI 等。因此，Java 的诞生对整个计算机产业产生了深远的影响，可以说，Java 为 Internet 和 WWW 开辟了一个崭新的时代。

Java 对传统的计算模型提出了新的挑战。业界不少人预言：“Java 语言的出现，将会引起一场软件革命。”这是因为传统的软件往往都是与具体的实现环境有关，而 Java 语言能在执行码（二进制码）上兼容，这样，以前所开发的软件就能运行在不同的机器上，只要所用的机器能提供 Java 语言解释器即可。

Java 语言将对未来软件的开发产生影响，可反映在如下几个方面。

(1) 软件的需求分析。可将用户的需求进行动态的、可视化描述，以满足设计者更加直观的要求。Java 语言不受地区、行业、部门、爱好的限制，都可以将用户的需求描述清楚。

(2) 软件的开发方法。由于 Java 语言的面向对象的特性，所以完全可以用面向对象的技术与方法来进行开发，符合最新的软件开发规范的要求。

(3) Java 语言的动态效果。就界面而言，GUI 技术达到动画效果；就数据而言，Java 能根据数据动态地提供信息。

(4) 软件最终产品。用 Java 语言开发的软件可以具有可视化、可听化、可操作化的效果，其多媒体应用也十分广泛。

(5) 其他。使用 Java 语言对开发效益、开发价值都有比较明显的影响。

正如 Java 的创始人之一 James Gosling 所说, Java 不仅仅只是 applets, 它能做任何事情: Java 不仅仅是一种程序设计语言, 更是现代化软件再实现的基础; Java 还是未来新型 OS 的核心; 将会出现 Java 芯片; Java 将构成各种应用软件的开发平台与实现环境, 是人们必不可少的开发工具。

因此, Java 语言有着广泛的应用前景, 例如:

- (1) 所有面向对象的应用开发, 包括面向对象的事件描述、处理和综合等;
- (2) 计算过程的可视化、可操作化的软件的开发;
- (3) 动态画面的设计, 包括图形图像的调用;
- (4) 交互操作的设计 (选择交互、定向交互和控制流程等);
- (5) Internet 的系统管理功能模块, 包括 Web 页面的动态设计、管理和交互操作设计等;
- (6) Intranet (企业内部网) 上的软件开发 (直接面向企业内部用户的软件);
- (7) 与各类数据库连接查询的 SQL 语句实现;
- (8) 其他应用类型的程序。

1.1.2 Java 的特点

简单地说, Java 是定位于网络计算的计算机语言, 它的几乎所有的特点也是围绕着这一中心展开并为之服务的, 这些特点使得 Java 语言特别适合用来开发网络上的应用程序。另外, 作为一种问世较晚的语言, Java 也集中体现和充分利用了若干当代软件技术新成果, 如面向对象、多线程等, 这些也都在它的特点中有所反映。Java 的特点如下。

1. 简单易学

衍生自 C++ 的 Java 语言, 出于安全稳定性的考虑, 去除了 C++ 中不容易理解和掌握的部分, 如最典型的指针操作等, 降低了学习的难度; 同时 Java 还有一个特点就是它的基本语法部分与 C 语言几乎一模一样。这样, 无论是掌握了 Java 再学 C 语言, 还是已经掌握了 C 语言再来学 Java, 都会感到易于入门。

2. 面向对象

Java 是面向对象的编程语言。面向对象技术较好地解决了当今软件开发过程中新出现的种种传统面向过程语言所不能处理的问题, 包括软件开发的规模扩大、升级加快、维护量增大, 以及开发分工日趋细化、专业化和标准化等, 是一种迅速成熟、推广的软件开发方法。面向对象技术的核心是以更接近于人类思维的方式建立计算机逻辑模型, 它利用类和对象的机制将数据与其上的操作封装在一起, 并通过统一的接口与外界交互, 使反映现实世界实体的各个类在程序中能够独立、自治、继承。这种方法非常有利于提高程序的可维护性和可重用性, 大大提高了开发效率和程序的可管理性, 使得面向过程语言难于操纵的大规模软件可以很方便地创建、使用和维护。C++ 也是面向对象的语言, 但是为了与 C 语言兼容, 其中还包含了一些面向过程的成分; Java 去除了 C++ 中非面向过程的部分, 其

程序编写过程就是设计、实现类，定义其属性、行为的过程。

3. 平台无关性

如前所述，Java 独特的运行机制使得它具有良好的二进制级的可移植性，利用 Java 语言，开发人员可以编写出与具体平台无关、普遍适用的应用程序，大大降低了开发、维护和管理开销。

4. 安全稳定

对网络上应用程序的另一个需求是较高的安全可靠。用户通过网络获取并在本地运行的应用程序必须是可信赖的，不会充当病毒或其他恶意操作的传播者而攻击用户本地的资源；同时它还应该是稳定的，轻易不会产生死机等错误，使得用户乐于使用。Java 特有的机制是其安全性的保障，同时它去除了 C++ 中易造成错误的指针，增加了自动内存管理等措施，保证了 Java 程序运行的可靠性。

5. 支持多线程

多线程是当今软件技术的又一重要成果，已成功应用在操作系统、应用开发等多个领域。多线程技术允许同一个程序有两个执行线索，即同时做两件事情，满足了一些复杂软件的需求。Java 不但内置多线程功能，而且提供语言级的多线程支持，即定义了一些用于建立、管理多线程的类和方法，使得开发具有多线程功能的程序变得简单、容易和有效。

6. 很好地支持网络编程

Java 是面向网络的语言。通过它提供的类库可以处理 TCP/IP 协议，用户可以通过 URL 地址在网络上很方便地访问其他对象。Java 的小应用程序（Applet）是动态、安全、跨平台的网络应用程序。Java Applet 嵌入 HTML 语言，通过主页发布到 Internet。网络用户访问服务器的 Applet 时，这些 Applet 从网络上进行传输，然后在支持 Java 的浏览器中运行。由于 Java 语言的安全机制，用户一旦载入 Applet，就可以放心地生成多媒体的用户界面或完成复杂的计算而不必担心病毒的入侵。虽然 Applet 可以和图像、声音、动画等一样从网络上下载，但它不同于这些多媒体的文件格式，它可以接收用户的输入，动态地进行改变，而不仅仅是动画的显示和声音的播放。

7. Java 丰富的类库

Java 提供了大量的类库以满足网络化、多线程、面向对象系统的需要。

- (1) 语言包提供的支持包括字符串处理、多线程处理、例外处理、数学函数处理等，可以用它简单地实现 Java 程序的运行平台。
- (2) 实用程序包提供的支持包括哈希表、堆栈、可变数组、时间和日期等。
- (3) 输入输出包用统一的“流”模型来实现所有格式的 I/O，包括文件系统、网络及输入/输出设备等。
- (4) 低级网络包用于实现 Socket 编程。
- (5) 抽象图形用户接口包实现了不同平台的计算机的图形用户接口部件，包括窗口、

菜单、滚动条和对话框等，使得 Java 可以移植到不同平台的机器。

(6) 网络包支持 Internet 的 TCP/IP 协议，提供了与 Internet 的接口。它支持 URL 连接，WWW 的即时访问，并且简化了用户/服务器模型的程序设计。

Java 的上述种种特性不但能适应网络应用开发的需求，而且还体现了当今软件开发方法的若干新成果和新趋势。在以后的章节里，将结合对 Java 语言的讲解，分别介绍这些软件开发方法。

1.1.3 Java 和 C、C++

对于变量声明、参数传递、操作符、流控制等，Java 使用了和 C、C++相同的传统，使得熟悉 C、C++的程序员能很方便地进行编程。同时，Java 为了实现其简单、健壮、安全等特性，也摒弃了 C 和 C++中许多不合理的内容。下面选择性地讲述几点，对于学过 C 语言或 C++语言的读者而言，起一个快速参考的作用。对于未学过 C 语言的读者，可以略过此节。

(1) 全局变量

Java 程序中，不能在所有类之外定义全局变量，只能通过在一个类中定义公用、静态的变量来实现一个全局变量。Java 对全局变量进行了更好的封装。而在 C 和 C++中，依赖于不加封装的全局变量常常会造成系统的崩溃。

(2) Goto 语句

Java 不支持 C、C++中的 Goto 语句，而是通过异常处理语句 try、catch、finally 等来代替 C、C++中用 Goto 来处理遇到错误时跳转的情况，使程序更可读且更结构化。

(3) 指针

指针是 C、C++中最灵活，也是最容易产生错误的数据类型。由指针所进行的内存地址操作常会造成不可预知的错误，同时通过指针对某个内存地址进行显示类型转换后，可以访问一个 C++中的私有成员，从而破坏了安全性，造成系统的崩溃。而 Java 对指针进行完全的控制，程序员不能直接进行任何指针操作，例如，把整数转化为指针，或者通过指针释放某一内存地址等。同时，数组作为类在 Java 中实现，很好地解决了数组访问越界这一在 C、C++中不做检查的错误。

(4) 内存管理

在 C 中，程序员通过库函数 malloc() 和 free() 来分配和释放内存，C++中则通过运算符 new 和 delete 来分配和释放内存。再次释放已释放的内存块或未被分配的内存块，会造成系统的崩溃；同样，忘记释放不再使用的内存块也会逐渐耗尽系统资源。而在 Java 中，所有的数据结构都是对象，通过运算符 new 为它们分配内存。通过 new 得到对象的处理权，而实际分配给对象的内存可能随程序运行而改变，Java 对此自动地进行管理并且进行垃圾收集，有效地防止了由于程序员的误操作而导致的错误，并且更好地利用了系统资源。

(5) 数据类型的支持

在 C、C++中，对于不同的平台，编译器为简单数据类型，如 int、float 等分别分配不同长度的字节数，例如，int 在 IBM PC 中为 16 位，在 VAX-11 中为 32 位，这导致了代码的不可移植性，但在 Java 中，对于这些数据类型总是分配固定长度的位数，如对 int 型，

它总占 32 位，这就保证了 Java 的平台无关性。

(6) 类型转换

在 C、C++ 中，由于可以通过指针进行任意的类型转换，因此常常带来不安全性；而 Java 中，系统在运行时对对象的处理要进行类型相容性检查，以防止不安全的转换。

(7) 头文件

C、C++ 中用头文件来声明类的原型及全局变量、库函数等，在大的系统中，维护这些头文件是很困难的。而 Java 不支持头文件，类成员的类型和访问权限都封装在一个类中，运行时系统对访问进行控制，防止对私有成员的操作。同时，Java 中用 `import` 语句来与其他类进行通信，以使用它们的方法。

(8) 结构和联合

C、C++ 中的结构和联合中所有成员均为公有，这就带来了安全性问题。Java 中不包含结构和联合，所有的内容都封装在类中。

(9) 预处理

C、C++ 中用宏定义来实现的代码给程序的可读性带来了困难。在 Java 中，不支持宏，它通过关键字 `final` 来声明一个常量，以实现宏定义中广泛使用的常量定义。

1.2 面向对象程序设计

Java 是面向对象的程序设计语言，面向对象的软件开发和相应的面向对象的问题求解是当今计算机技术发展的重要成果和趋势之一。本节介绍面向对象软件开发和面向对象程序设计中的基本概念和基本方法，使读者对面向对象软件开发方法的体系、原则、基本思想和特点有一定的了解。

1.2.1 面向对象概述

面向过程的程序设计是以具体的解题过程为研究和实现的主体，而面向对象的程序设计是以需解决的问题中所涉及的各种对象为主体。

在面向对象的方法学中，“对象”是现实世界的实体或概念在计算机逻辑中的抽象表示。具体地，对象是具有惟一对象名和固定对外接口的一组属性和操作的集合，用来模拟组成或影响现实世界问题的一个或一组因素。其中对象名是区别于其他对象的标志；对外接口是对象在约定好的运行框架和消息传递机制中与外界通信的通道；对象的属性表示它所处的状态；而对象的操作则用来改变对象的状态达到特定的功能。对象的最主要特点是以数据为中心，它是一个集成了数据和其上操作的独立、自恰的逻辑单位。

面向对象的问题求解就是力图从实际问题中抽象出这些封装了数据和操作的对象，通过定义属性和操作来表述它们的特征和功能，通过定义接口来描述它们的地位及与其他对象的关系，最终形成一个广泛联系的可理解、可扩充、可维护及更接近于问题本来面目的动态对象模型系统。

面向对象的程序设计将在面向对象的问题求解所形成的对象模型基础之上，选择一种面向对象的高级语言来具体实现这个模型。相对于传统的面向过程的程序设计方法，面向

对象的程序设计具有如下的优点。

(1) 对象的数据封装特性彻底消除了传统结构方法中数据与操作分离所带来的种种问题,提高了程序的可复用性和可维护性,降低了程序员保持数据与操作相容的负担。

(2) 对象的数据封装特性还可以把对象的私有数据和公共数据分离开,保护了私有数据,减少了可能的模块间干扰,达到降低程序复杂性、提高可控性的目的。

(3) 对象作为独立的整体具有良好的自恰性。即,它可以通过自身定义的操作来管理自己。一个对象的操作可以完成两类功能,一是修改自身的状态,二是向外界发布消息。当一个对象欲影响其他对象时,它需要调用其他对象自身的方法,而不是直接去改变那个对象。对象的这种自恰性能使得所有修改对象的操作都以对象自身的一部分的形式存在于对象整体之中,维护了对象的完整性,有利于对象在不同环境下的复用、扩充和维护。

(4) 在具有自恰性的同时,对象通过一定的接口和相应的消息机制与外界相联系。这个特性与对象的封装性结合在一起,较好地实现了信息的隐藏。即,对象成为一只使用方便的“黑匣子”,其中隐藏了私有数据和细节内容。使用对象时只需要了解其接口提供的功能操作即可,而不必了解对象内部的数据描述和具体的功能实现。

(5) 继承是面向对象方法中除封装外的另一个重要特性。通过继承可以很方便地实现应用的扩展和已有代码的重复使用,在保证质量的前提下提高开发效率,使得面向对象的开发方法与软件工程的新兴方法——快速原型法很好地结合在一起。

综上所述,面向对象程序设计是将数据及数据的操作封装在一起,成为一个不可分割的整体,同时,将具有相同特征的对象抽象成为一种新的数据类型——类。通过对象间的消息传递使整个系统运转。通过对象类的继承提供代码重用的有效途径。

在面向对象程序设计方法中,其程序结构是一个类的集合和各类之间以继承关系联系起来的结构,有一个主程序,在主程序中定义各对象并规定它们之间传递消息的规律。

从程序执行这一角度来看,可以归结为各对象和它们之间的消息通信。面向对象程序设计最主要的特征 is 各对象之间的消息传递和各类之间的继承。

1.2.2 对象、类与实体

对象的概念是面向对象技术的核心所在。以面向对象的观点来看,所有面向对象的程序都是由对象组成的,这些对象首先是自治、自恰的,同时它们还可以互相通信、协调和配合,从而共同完成整个程序的任务和功能。

更确切地,面向对象技术中的对象就是现实世界中某个具体的物理实体在计算机逻辑中的映射和体现。比如,电视机是一个具体存在的,拥有外形、尺寸、颜色等外部特性和开关、频道设置等实在功能的实体,而这样一个实体,在面向对象的程序中,就可以表达成一个计算机可理解、可操纵、具有一定属性和行为的对象。

类也是面向对象技术中一个非常重要的概念。简单地说,类是同种对象的集合与抽象。类是一种抽象的数据类型,它是所有具有一定共性的对象的抽象,而属于类的某一个对象则被称为是类的一个实例,是类的一次实例化的结果。如果类是抽象的概念,如“电视机”,那么对象就是某一个具体的电视机,如“我家那台电视机”。

1.2.3 对象的状态与行为

对象都具有状态和行为。

对象的状态又称为对象的静态属性，主要指对象内部所包含的各种信息，也就是变量。每个对象个体都具有自己专有的内部变量，这些变量的值标明了对象所处的状态。当对象经过某种操作和行为而发生状态改变时，具体地就体现为它的属性变量的内容的改变。通过检查对象属性变量的内容，就可以了解这个对象当前所处的状态。仍然以电视机为例，每一个电视机都具有以下这些状态信息：种类、品牌、外观、大小、颜色、是否开启、所在频道等，这些状态在计算机中都可以用变量来表示。

行为又称为对象的操作，它主要表述对象的动态属性，操作的作用是设置或改变对象的状态。比如一个电视机可以有打开、关闭、调整音量、调节亮度、改变频道等行为或操作。对象的操作一般都基于对象内部的变量，并试图改变这些变量(即改变对象的状态)。如“打开”的操作只对处于关闭状态的电视机有效，而执行了“打开”操作之后，电视机原有的关闭状态将改变。对象的状态在计算机内部是用变量来表示，而对象的行为在计算机内部是用方法来表示的。方法实际上类似于面向过程中的函数。对象的行为或操作定义在其方法的内部。

1.2.4 对象的关系

一个复杂的系统必然包括多个对象，这些对象之间可能存在的关系有三种：包含、继承和关联。

1. 包含

当对象 A 是对象 B 的属性时，称对象 B 包含对象 A。例如，每台电视机都包括一个显像管。当把显像管抽象成一个计算机逻辑中的对象时，它与电视机对象之间就是包含的关系。

当一个对象包含另一个对象时，它将在自己的内存空间中为这个被包含对象留出专门的空间，即被包含对象将被保存在包含它的对象内部，就像显像管被包含在电视机中一样，这与它是电视机组成部分的地位是非常吻合的。

2. 继承

当对象 A 是对象 B 的特例时，称对象 A 继承了对象 B。例如，黑白电视机是电视机的一种特例，彩色电视机是电视机的另一种特例。如果分别为黑白电视机和彩色电视机抽象出黑白电视机对象和彩色电视机对象，则这两种对象与电视机对象之间都是继承的关系。

实际上，这里所说的对象间的继承关系就是后面要详细介绍的类间的继承关系。作为特例的类称为子类，而子类所继承的类称为父类。父类是子类公共关系的集合，子类将在父类定义的公共属性的基础上，根据自己的特殊性特别定义自己的属性。例如，彩色电视机对象除了拥有电视机对象的所有属性之外，还特别定义了静态属性“色度”和相应的动态操作“调节色度”。

3. 关联

当对象 A 的引用是对象 B 的属性时，称对象 A 和对象 B 之间是关联关系。所谓对象的引用是指对象的名称、地址、句柄等可以获取或操纵该对象的途径。相对于对象本身，对象的引用所占用的内存空间要少得多，它只是找到对象的一条线索。通过它，程序可以找到真正的对象，并访问这个对象的数据，调用这个对象的方法。

例如，每台电视机都对应一个生产厂商，如果把生产厂商抽象成厂商对象，则电视机对象应该记录自己的生产厂商是谁，此时电视机对象和厂商对象之间就是关联的关系。

关联与包含是两种不同的关系。厂商并不是电视机的组成部分，所以电视机对象里不需要也不可能保存整个厂商对象，而只需要保存一个厂商对象的引用，例如厂商的名称。

这样，当需要厂商对象时，如当需要从厂商那里购买一个零件时，只需要根据电视机对象中保存的厂商的名字就可以方便地找到这个厂商对象。

1.2.5 面向对象的软件开发过程

面向对象的软件开发过程可以大体划分为面向对象的分析(Object Oriented Analysis, OOA)、面向对象的设计(Object Oriented Design, OOD)、面向对象的实现(Object Oriented Programming, OOP)三个阶段。

1. 面向对象的分析

面向对象的分析的主要作用是明确用户的需求，并用标准化的面向对象的模型规范地表述这一需求，最后将形成面向对象的分析模型，即 OOA 模型。分析阶段的工作应该由用户和开发人员共同协作完成。

需求分析是要抽取存在于用户需求中的各对象实体，分析、明确这些对象实体的静态数据属性和动态操作属性，以及它们之间的相互关系。更重要的是，要能够反映出由多个对象组成的系统的整体功能和状态，包括各种状态间的变迁及对象在这些变迁中的作用、在整个系统中的位置等。需求模型化方法是面向对象的分析中常用的方法。这种方法通过对需要解决的实际问题建立模型来抽取、描述对象实体，最后形成 OOA 模型，将用户的需求准确地表达出来。OOA 模型有很多种设计和表达方法，如使用较为广泛的 Coad&Yourdon 的 OOA 模型。

2. 面向对象的设计

如果说分析阶段应该明确所要开发的软件系统“干什么”，那么设计阶段将明确这个软件系统“怎么做”。面向对象的设计将对 OOA 模型加以扩展并得到面向对象的设计阶段的最终结果：OOD 模型。

面向对象的设计将在 OOA 模型的基础上引入界面管理、任务管理和数据管理三部分的内容，进一步扩充 OOA 模型。其中，界面管理负责整个系统的人机界面的设计；任务管理负责处理并行操作之类的系统资源管理功能的工作；数据管理则负责设计系统与数据库的接口。这三部分再加上 OOA 模型代表的“问题逻辑”部分，就构成了最初的 OOD 模型。

面向对象的设计还需要对最初的 OOD 模型做进一步的细化分析、设计和验证。在“问题逻辑”部分，细化设计包括对类静态数据属性的确定，对类方法(即操作)的参数、返回值、功能和功能的实现的明确规定等；细化验证主要指对各对象类公式间的相容性和一致性的验证，对各个类、类内成员的访问权限的严格合理性的验证，也包括验证对象类的功能是否符合用户的需求。

3. 面向对象的实现

面向对象的实现就是具体的编码阶段，其主要任务包括：

- (1) 选择一种合适的面向对象的编程语言，如 C++、Object Pascal、Java 等；
- (2) 用选定的语言编码实现详细设计步骤所得的公式、图表、说明和规则等对软件系统各对象类的详尽描述；
- (3) 将编写好的各个类代码模块根据类的相互关系集成；
- (4) 利用开发人员提供的测试样例和用户提供的测试样例分别检验编码完成的各个模块和整个软件系统。

综上所述，面向对象的软件开发可概括为如下过程：分析用户需求，从问题中抽取对象模型；将模型细化、设计类，包括类的属性和类间相互关系，同时考察是否有可以直接引用的已有类或部件；选定一种面向对象的编程语言，具体编码实现上一阶段类的设计，并在开发过程中引入测试，完善整个解决方案。

由于对象的概念能够以更接近实际问题的原貌和实质的方式来表述和处理这些问题，所以面向对象的软件开发方法比以往面向过程的方法有更好的灵活性、可重用性和可扩展性，使得上述“分析—设计—实现”的开发过程也更加高效、快捷。即使出现因前期工作不彻底、用户需求改动等需要反馈并修改前面步骤的情况，也能够以前工作的基础之上从容地完成，而不会陷入传统方法中不得不推翻原有设计，重新考虑数据结构和程序结构的尴尬境地。

习题

1. Java 语言有哪些主要特点？
2. 简述面向过程问题求解和面向对象问题求解的异同。试列举出面向对象和面向过程的编程语言各两种。
3. 简述对象、类和实体及它们之间的相互关系。尝试从日常接触到的人或物中抽象出对象的概念。
4. 对象有哪些属性？什么是状态？什么是行为？二者之间有何关系？设有对象“学生”，试为这个对象设计状态与行为。
5. 对象间有哪三种关系？对象“班级”与对象“学生”是什么关系？对象“学生”与对象“大学生”是什么关系？
6. 有人说“父母”和“子女”之间是继承的关系。这种说法是否正确？为什么？
7. 面向对象的软件开发包括哪些过程？OOA 模型包括哪三个层次？OOD 模型在 OOA 模型的基础上引入了哪些工作？

8. 面向对象的程序设计方法有哪些优点?

第 2 章 简单的 Java 程序

本章从介绍和分析最简单的 Java 程序例子出发，讲述开发 Java 程序的基本步骤、Java 程序的构成、基本输入输出编程及 Java 的开发工具。

2.1 Application 与 Applet 程序

根据结构组成和运行环境的不同，Java 程序可以分为两类：Java Application 和 Java Applet。简单地说，Java Application 是完整的程序，需要独立的解释器来解释运行；而 Java Applet 则是嵌在 HTML 网页（Web 页面）中的非独立程序，由 Web 浏览器内部包含的 Java 解释器来解释运行。

下面介绍两个简单的 Java 程序，并对其进行分析。

2.1.1 Application 程序

例 2-1 HelloWorldApp.java 简单的 Application 程序。

```
public class HelloWorldApp { //an application
    public static void main (String args[ ]) {
        System.out.println("Hello World!");
    }
}
```

本程序的作用是输出下面一行信息：

```
Hello World!
```

程序中，首先用保留字 `class` 来声明一个新的类，其类名为 `HelloWorldApp`，它是一个公共类(`public`)。整个类定义由大括号 `{ }` 括起来。

在该类中定义了一个 `main()` 方法，其中 `public` 表示访问权限，指明所有的类都可以使用这一方法；`static` 指明该方法是一个类方法，它可以通过类名直接调用；`void` 则指明 `main()` 方法不返回任何值。

对于一个应用程序来说，`main()` 方法是必须的，而且必须按照如上的格式来定义。Java 解释器以 `main()` 作为入口来执行程序。Java 程序中可以定义多个类，每个类中可以定义多个方法，但是最多只能有一个公共类。`main()` 方法也只能有一个，作为程序的入口。

`main()` 方法定义中，括号中的 `String args[]` 是传递给 `main()` 方法的参数，参数名为 `args`，它是类 `String` 的一个实例，参数可以为 0 个或多个，每个参数用“类名 参数名”来指定，多个参数间用逗号分隔。在 `main()` 方法的实现(大括号中)，只有一条语句：

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

它用来实现字符串的输出，这条语句实现与 C 语言中的 `printf` 语句和 C++ 中 `cout<<` 语句相同的功能。另外，“`//`”后的内容为注释。

现在我们可以运行该程序。首先把它放到一个名为 `HelloWorldApp.java` 的文件中，这里，文件名应和类名相同，因为 Java 解释器要求公共类必须放在与其同名的文件中。然后对它进行编译：

```
C:\>javac HelloWorldApp.java
```

编译的结果是生成字节码文件 `HelloWorldApp.class`。最后用 java 解释器来运行该字节码文件：

```
C:\>java HelloWorldApp
```

结果在屏幕上显示 `Hello World!`

2.1.2 Applet 程序

再来看下面的一个例子：

例 2-2 HelloWorldApplet.java 简单的 Applet 程序。

```
import java.awt.*;
import java.applet.*;

public class HelloWorldApplet extends Applet { //an applet
    public void paint(Graphics g){
        g.drawString ("Hello World!",20,20);
    }
}
```

这是一个简单的 Applet(小应用程序)。程序中，首先用 `import` 语句输入 `java.awt` 和 `java.applet` 下所有的包，使得该程序可能使用这些包中所定义的类，它类似于 C 中的 `#include` 语句。然后声明一个公共类 `HelloWorldApplet`，用 `extends` 指明它是 `Applet` 的子类。在类中，重写父类 `Applet` 的 `paint()` 方法，其中参数 `g` 为 `Graphics` 类，它表明当前作画的上下文。在 `paint()` 方法中，调用 `g` 的方法 `drawString()`，在坐标(20,20)处输出字符串"Hello World!"，其中坐标是用像素点来表示的。

这个程序中没有实现 `main()` 方法，这是 Applet 与应用程序 `Application` 的区别之一。为了运行该程序，首先我们也要把它放在文件 `HelloWorldApplet.java` 中，然后对它进行编译：

```
C:\>javac HelloWorldApplet.java
```

得到字节码文件 `HelloWorldApplet.class`。由于 Applet 中没有 `main()` 方法作为 Java 解释器的入口，必须编写 HTML 文件，把该 Applet 嵌入其中，然后用 `appletviewer` 来运行，或在支持 Java 的浏览器上运行。它的 `<HTML>` 文件如下：

```
<HTML>
<HEAD>
<TITLE> An Applet </TITLE>
</HEAD>
<BODY>
<applet code="HelloWorldApplet.class" width=200 height=40>
```

```
</applet>
</BODY>
</HTML>
```

其中, 用<Applet>标记来启动 HelloWorldApplet, code 指明字节码所在的文件, width 和 height 指明 applet 所占的大小。把这个 HTML 文件存入 Hello.html, 然后运行:

```
C:\>appletviewer Hello.html
```

这时屏幕上弹出一个窗口, 其中显示 Hello World!, 显示结果如图 2-1 所示。



图 2-1 程序的运行结果

从上述例子中可以看出, Java 程序是由类构成的, 对于一个应用程序来说, 必须有一个类中定义 main() 方法, 而对 applet 来说, 它必须作为 Applet 一个子类。在类的定义中, 应包含类变量的声明和类中方法的实现。Java 在基本数据类型、运算符、表达式、控制语句等方面与 C、C++ 基本上是相同的, 但它同时也增加了一些新的内容, 在以后的各章中将详细介绍。这里只是让读者对 Java 程序有一个初步的了解。

2.1.3 Java 程序的基本构成

一个复杂的程序可由一个至多个 Java 源程序文件构成, 每个文件中可以有多个类定义。下面的程序是一个一般的 Java 程序文件:

```
package ch02;
import java.io.*;
public class AppCharInOut
{
    public static void main(String[] args)
    {
        char c = ' ';
        System.out.print("Please input a char: ");
        try{
            c = (char) System.in.read();
        }catch(IOException e){}
        System.out.println("You have entered: " + c );
    }
}
```

从这个程序可以看出, 一般的 Java 源程序文件由以下三部分组成:

- ☞ package 语句 (0 句或 1 句);
- ☞ import 语句 (0 句或多句);
- ☞ 类定义 (1 个或多个类定义)。

其中, package 语句表示本程序所属的包。它只能有 1 句或者没有。如果有, 必须放在最前面。如果没有, 表示本程序属于默认包。

import 语句表示引入其他类的库, 便于使用。import 语句可以有 0 或多句, 它必须放在类定义的前面。

类定义是 Java 源程序的主要部分, 每个文件中可以定义若干个类。

Java 程序中定义类使用关键字 class, 每个类的定义由类头定义和类体定义两部分组成。类体部分用来定义属性和方法这两种类的成员, 其中方法类似于其他高级语言中的函数, 而属性则类似于变量。类头部分除了声明类名之外, 还可以说明类的继承特性, 当一个类被定义为是另一个已经存在的类(称为这个类的父类)的子类时, 它就可以从其父类中继承一些已定义好的类成员而不必自己重复编码。

在类体中通常有两种组成成分, 一种是域, 包括变量、常量、对象数组等独立的实体; 另一种是方法, 类似于函数的代码单元块, 这两种组成成分通称为类的成员。在上面的例子中, 类 `My Java Application` 中只有一个类成员: 方法 `main`。用来标志方法头的是一对小括号, 在小括号前面并紧靠左括号的是方法名称, 如 `main` 等; 小括号里面是该方法使用的形式参数, 方法名前面是用来说明这个方法属性的修饰符, 其具体语法规定将在后面介绍。方法体部分由若干以分号结尾的语句组成, 并由一对大括号括起, 在方法体内部不能再定义其他的方法。

同其他高级语言一样, 语句是构成 Java 程序的基本单位之一。每一条 Java 语句都由分号“;”结束, 其构成应该符合 Java 的语法规则。类和方法中的所有语句应该用一对大括号{}括起。除 package 及 import 语句之外的其他的执行具体操作的语句, 都只能存在于类的大括号之中。

比语句更小的语言单位是表达式、变量、常量和关键字等, Java 的语句就是由它们构成的。其中, 变量与常量关键字是 Java 语言语法规定的保留字, 用户程序定义的常量和变量的取名不能与保留字相同。

Java 源程序的书写格式比较自由, 如语句之间可以换行, 也可以不换行, 但养成一种良好的书写习惯比较重要。

注意: Java 是大小写严格区分的语言。书写时, 大小写不能混淆。

一个程序中可以有多个类, 但只有一个类是主类。在 `Java Application` 中, 这个主类是指包含 `main` 方法的类。在 `Java Applet` 里, 这个主类是一个系统类 `Applet` 的子类。主类是 Java 程序执行的入口点。同一个 Java 程序中定义的若干类之间没有严格的逻辑关系要求, 但它们通常是在一起协同工作的, 每一个类都可能需要使用其他类中定义的静态属性或方法。

2.2 程序的编辑、编译与运行

一般高级语言编程需要经过源程序编辑、目标程序编译生成和可执行程序运行几个过程。Java 编程也不例外，一般可以分为编辑源程序、编译生成字节码和解释运行字节码几个步骤。本节就其一般步骤进行介绍。

2.2.1 Java 工具包 JDK

Java 编程的基本工具包是 JDK (Java Development Kit)。JDK 是 Sun 公司免费提供的开发、运行 Java 程序的基本软件，它可以在 Windows 及 Unix 两种平台上使用。常用的版本是 JDK1.2.2, JDK1.3.0, JDK1.4 等。可以从 <http://java.sun.com> 网站下载较新的版本。

下载工具包，如运行 j2sdk-1_4_1-windows-i586.exe 文件，可以安装该软件，将软件安装到一定的目录，如 jdk1.4.1，此目录称为 JDK 安装目录。该目录下有几个子目录：

bin	该目录存放运行程序
demo	该目录存放一些示例文件
include	该目录存放与 C 相关的头文件
jre	该目录存放 Java 运行环境相关的文件
lib	该目录存放程序库

另外，在安装目录下还有 src.zip 文件，该压缩文件中含有 Java 库程序的源程序，有兴趣的读者可以解开此文件，阅读并学习其中的源程序。

为了使用方便，读者还应该下载 JDK 的文档，安装后，会在 jdk 目录下生成一个 doc 子目录，这里有详细的 java 文档，读者可以经常查阅。

2.2.2 Application 的编辑、编译与运行

1. 程序的编辑

Java 源程序是以 .java 为后缀的简单的文本文件，可以用各种 Java 集成开发环境中的源代码编辑器来编写，也可以用其他文本编辑工具，如 Windows 中的记事本或 DOS 中的 EDIT 软件等。

以最简单的记事本 (Notepad) 软件为例，打开记事本，输入下面一段程序（即例 2-1 所示的程序）：

```
public class HelloWorldApp { //an application
    public static void main (String args[ ]){
        System.out.println("Hello World!");
    }
}
```

在输入程序时，要注意 Java 程序是严格区分大小写的。

程序输入并修改完毕，要将此文件保存，在保存文件时要注意，文件的类型要选“所有类型”，文件名要与程序中的 public class 的类名一致，这里的文件名应为

HelloWorldApp.java。文件名的大小写最好也要保持与类名一致。

2. 程序的编译

与其他语言一样，源程序（.java 文件）要经过编译才能运行。编译的过程实际上是将 java 源程序转变为字节码（bytecode）文件。字节码文件的扩展名为.class，其中包含的是 java 虚拟机的指令。

编译可以使用 JDK 中的工具 javac.exe。在 Windows 中，该工具的使用方法如下：

(1) 进入 DOS 环境，方法是：选择【开始】菜单中的【运行】，然后键入
command 或 cmd <回车>

(2) 然后进入到存放源文件的目录（假定是 d:\tang 目录），运行
d: <回车>
cd tang <回车>

(3) 编译源程序，键入

```
c:\j2sdk1.4.1\bin\javac HelloWorldApp.java
```

这里，c:\j2sdk1.4.1 为 JDK 的安装目录，javac 为编译工具。

为了简化写 c:\j2sdk1.4.1\bin，可以先键入命令

```
Set path=c:\j2sdk1.4.1\bin;%path%
```

这样，编译命令可以直接写 javac，如：

```
javac HelloWorldApp.java
```

javac 后面可以跟 java 源程序文件名，文件名可以有多个，还可以用*及?通配符，如：

```
javac Hello*.java
```

javac 还可以跟一系列选项，其使用方法如下：

```
javac <选项> <源文件>
```

可能的选项如下。

- ☞ -g：生成所有调试信息。
- ☞ -g:none：生成无调试信息。
- ☞ -g:{lines,vars,source}：生成只有部分调试的信息。
- ☞ -nowarn：生成无警告。
- ☞ -verbose：输出关于编译器正在做的信息。
- ☞ -deprecation：输出使用了不鼓励使用 API 的源文件位置。
- ☞ -classpath：指定用户类文件的位置。
- ☞ -sourcepath：指定输入文件源文件的位置。
- ☞ -bootclasspath：跳过本地的、而使用别处的引导程序类文件。
- ☞ -extdirs：跳过本地的、而使用别处的已安装的扩展文件。
- ☞ -d：指定输出类文件的位置。
- ☞ -encoding：指定源文件使用的字符集编码。
- ☞ -source：提供指定释放的源兼容性。
- ☞ -target：生成指定虚拟机的类文件。
- ☞ -help：列出同步标准选项。

若编译不成功, javac 会提示信息, 根据此信息, 可进一步修改源程序, 再重新编译。编译成功后, javac 会产生相应的.class 文件, 这就是字节码文件。

3. 程序的运行

程序的运行就是执行.class 文件中的指令的过程。由 Java 源代码编译生成的字节码不能直接运行在一般的操作系统平台上, 而必须运行在一个称为“Java 虚拟机(JVM)”的在操作系统之外的软件平台上。在运行 Java 程序时, 首先应该启动这个虚拟机, 然后由它来负责解释执行 Java 的字节码。这样, 利用 Java 虚拟机就可以把 Java 字节码程序跟具体的软硬件平台分隔开来, 只要在不同的计算机上安装针对其特定具体平台特点的 Java 虚拟机, 就可以把这种不同软硬件平台的具体差别隐藏起来, 使得 Java 字节码程序在不同的计算机上能够面对相同的 Java 虚拟机, 而不必考虑具体的平台差别, 从而实现了真正的二进制代码级的跨平台可移植性。

JDK 提供的解释器是 java.exe。Java Application 是由若干个类定义组成的独立的解释型程序, 其中必须有一个包含 main 方法的主类。执行 Java Application 时, 需使用独立的 Java 解释器来解释执行这个主类的字节码文件。

在上面的例子中, 运行所编译好的程序, 使用命令:

```
java HelloWorldApp
```

与编译时一样, 需要提前设定好 path。

注意: 这里不能写为 HelloWorldApp.class。

对于 JDK1.2 以前的版本, 还需要设定另外一个环境变量 classpath, 设定方法类:

```
set classpath=%classpath %.;c:\jdk1.2\lib.jar
```

对于 JDK1.3 以后的版本, 则可以不设定 classpath。classpath 的作用是告诉 java 的解释器在哪里找到.class 文件及相关的库程序。

运行的效果如图 2-2 所示。

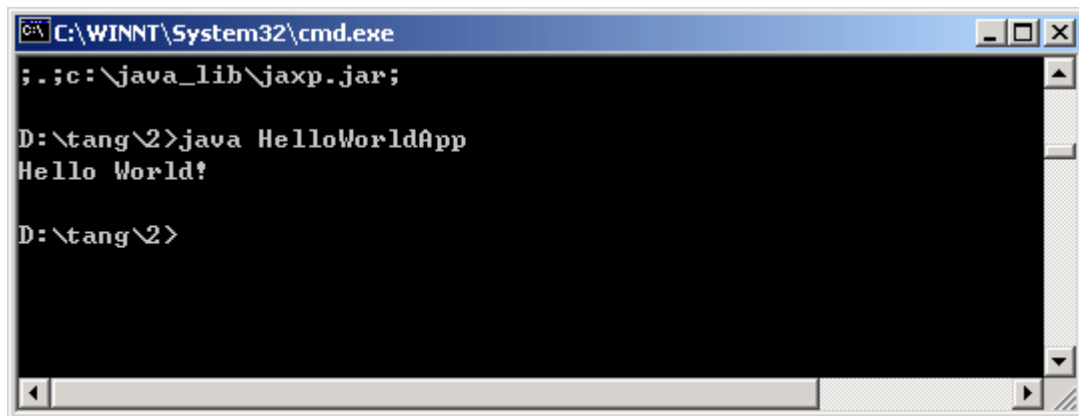


图 2-2 HelloWorld 运行结果

4. 设定 path 及 classpath

如上所述, 在编译及运行时, 经常需要设定 path 及 classpath 两个环境变量, 为了方便, 可以将以上两个命令及其他一些命令放入一个.bat 文件中, 这样只需要运行这个.bat 文件即可。

下面是一个.bat 文件的例子:

```
set ANT_HOME=c:\ant
set JAVA_HOME=c:\j2sdk1.4.1
set J2EE_HOME=c:\j2sdkee1.3
set OTHER_LIB=c:\java_lib\jaxp.jar;
set PATH=%JAVA_HOME%\bin;%J2EE_HOME%\bin;%ANT_HOME%\bin;%PATH%
set CLASSPATH=%CLASSPATH%;%JAVA_HOME%\jre\lib\rt.jar;
    %J2EE_HOME%\lib\j2ee.jar;.
set CLASSPATH=%CLASSPATH%;%OTHER_LIB%
```

还有一种设定环境变量的方法。以 Windows 2000 或 Windows XP 为例, 设定方法是: 用鼠标单击【我的电脑】, 选择【属性】, 打开【属性】对话框, 选择【高级】选项卡, 单击【环境变量】按钮, 然后在【环境变量】中进行设置即可, 如图2-3 所示。

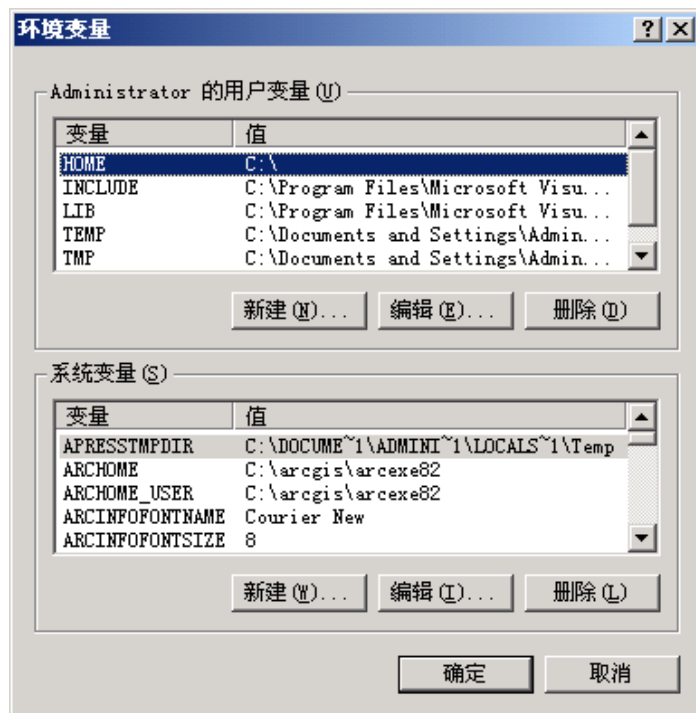


图 2-3 环境变量的设置

2.2.3 Applet 的编辑、编译与运行

Java Applet 是另一类非常重要的 Java 程序，虽然它的源代码编辑与字节码的编译生成过程与 Java Application 相同，但它却不是一类可以独立运行的程序，相反，它的字节码文件必须嵌入到另一种语言 HTML 的文件中并由负责解释 HTML 文件的 WWW 浏览器充当其解释器，来解释执行 Java Applet 的字节码程序。

HTML 是 Internet 上最广泛应用的信息服务形式 WWW 中使用的通用语言，它可以将网络上不同地点的多媒体信息有组织地呈现在 WWW 浏览器中，而 Java Applet 的作用正是进一步丰富 HTML 中的信息内容和表达方式，其中最关键的是在 WWW 中引入动态交互的内容，使得它不仅仅提供静态的信息，而且可以提供可靠的服务，从而使网络更广泛地渗入到社会生活的方方面面。在 Java 语言诞生的初期，最使人们着迷的正是 Java Applet。

1. 源程序的编辑与编译

与 Application 一样，首先用编辑器输入以下源程序：

```
import java.awt.*;
import java.applet.*;

public class HelloWorldApplet extends Applet { //an applet
    public void paint(Graphics g){
        g.drawString ("Hello World!",20,20);
    }
}
```

保存为 HelloWorldApplet.java 文件，然后用 javac 将它编译为.class 文件：

```
javac HelloWorldApplet.java
```

2. 在 HTML 文件中嵌入 Applet

Applet 的运行与 Application 的运行不同，Applet 需要嵌入 HTML 文件中，才可以执行。WWW 浏览器（如 Internet Explorer，简称 IE）可以显示 HTML 文件规定的 Web 页面，当把一个 Java Applet 程序嵌入 HTML 文件时，HTML 文件会在其 Web 页面中划定一块区域作为此 Applet 程序的显示界面。由于浏览器中也有 Java 的虚拟机，它会解释执行其中的程序。浏览器浏览这个 Applet 程序所在的 HTML 文件时，会在合适的时刻自动执行此 paint 方法，从而在屏幕上显示出程序中欲显示的信息。

HTML 是一种简单的排版描述语言，称为“超文本标记语言”，它通过各种各样的标记来编排超文本信息。在 HTML 文件中嵌入 Java Applet 同样需要通过使用一组约定好的特殊标记：<APPLET>和</APPLET>。其中，<APPLET>标记还必须包含三个参数。

(1) CODE：指明嵌入 HTML 文件中的 Java Applet 字节码文件的文件名。

(2) HEIGHT：指明 Java Applet 程序在 HTML 文件所对应的 Web 页面中占用区域的高度。

(3) WIDTH：指明 Java Applet 程序在 HTML 文件所对应的 Web 页面中占用区域的宽度。

可以看出,所谓把 Java Applet 字节码嵌入 HTML 文件,实际上只是把字节码文件的文件名嵌入 HTML 文件,而真正的字节码文件本身则通常独立地保存在与 HTML 文件相同的路径中,由 WWW 浏览器根据 HTML 文件中嵌入的名字自动去查找和执行这个字节码文件。HTML 文件可以用普通的文本编辑工具编写,并保存在 Web 服务器的合适的位置。

关于 HTML 语言的具体规则和使用方法,本书就不再介绍了,感兴趣的读者可以查看有关的参考书目或相关网站。

对于本节的例子,用任何文本编辑器编辑一个文件,文件名为 Hello.html,内容为:

```
<HTML>
<HEAD>
<TITLE> An Applet </TITLE>
</HEAD>
<BODY>
<applet code="HelloWorldApplet.class" width=200 height=40>
</applet>
</BODY>
</HTML>
```

可以用支持 Java 的浏览器(如 IE, Netscape)来打开这个文件。当 WWW 浏览器下载此 HTML 文件并显示时,它会自动下载此 HTML 中指定的 Java Applet 字节码,然后调用内置在浏览器中 Java 解释器来解释执行下载到本机的字节码程序。其效果如图 2-4 所示。

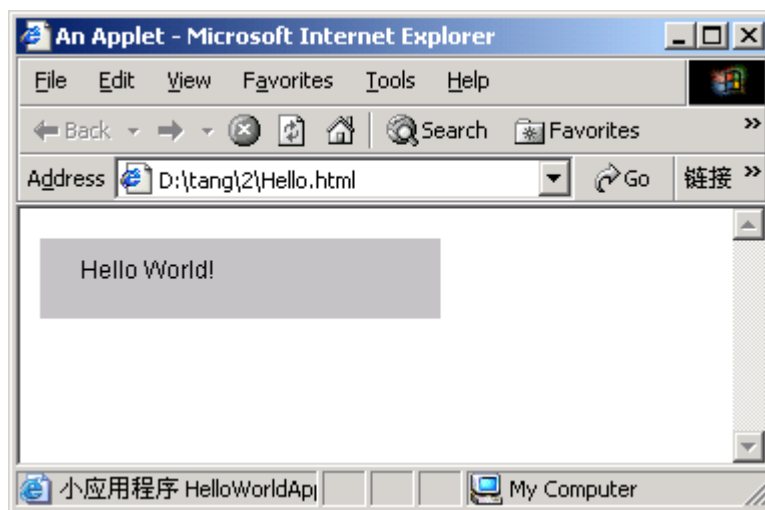


图 2-4 HelloWorld 的运行结果

在 JDK 中还提供了一个 Applet Viewer 工具,可以用来浏览 HTML 中的 Applet。如:

```
appletviewer hello.html
```

其效果如图 2-5 所示。



图 2-5 使用 AppletViewer 查看的结果

综上所述, Java Applet 是由若干个类定义组成的解释型程序, 其中必须有一个类是系统类 Applet 的子类。执行 Java Applet 时, 需先将编译生成的字节码文件嵌入 HTML 文件, 并使用内置 Java 解释器的浏览器来解释执行这个字节码文件。

2.2.4 使用 jar 打包程序

当程序很复杂时, 可以将多个.class 文件及相关的其他文件(如图像文件等)打包并压缩成一个文件, 这个文件称为 jar(Java Archive)文件。

JDK 中提供了一个工具(jar.exe)可以用来生成一个 jar 文件。例如, 以下命令将两个 class 文件存档到一个名为“classes.jar”的存档文件中:

```
jar cvf classes.jar Foo.class Bar.class
```

在使用 jar 时还可以指定一个“元信息清单文件”(manifest 文件), 它可以将元信息同时记入 jar 文件中。如, 用一个存在的清单(manifest)文件“mymanifest”将 foo/ 目录下的所有文件存档到一个名为“classes.jar”的存档文件中:

```
jar cvfm classes.jar mymanifest -C foo/
```

其中, manifest 文件的内容比较简单, 它的每一行是由一个关键字、一个冒号及一个字符串构成。例如, 为了指明 main()所在的类, 可以这样建立一个 manifest 文件, 其内容如下。

```
Manifest-Version: 1.0
```

```
Main-Class: MyClass
```

运行 jar 文件的方式是在 java 命令中用 -jar 选项, 如:

```
java -jar MyJarFile.jar
```

这时, 由于在 manifest 信息中指明了 Main-Class, 它会执行其中的主类的 main()方法。

2.3 Java 程序中的基本输入与输出

输入和输出是程序的基本功能, 本节将介绍如何编写具有基本输入与输出功能的 Java 程序, Java Application 程序输入和输出的可以是文本界面, 也可以是图形界面, Java Applet 则只能在图形界面中工作。

2.3.1 字符界面的输入与输出

所谓字符界面是指字符模式的用户界面。在字符界面中，用户用字符串向程序发出命令，传送数据，程序运行的结果也用字符的形式表达。虽然图形用户界面已经非常普及，但是在某些情况下仍然需要用到字符界面的应用程序，例如字符界面的操作系统，或者仅仅支持字符界面的终端等。

字符界面的输入输出要用到 `java.io` 包，用 `System.in` 及 `System.out` 来表示输入及输出，`System.in` 的 `read()` 方法可以输入字符，`System.out` 的 `print()` 方法可以输出一个字符串，字符串之间或字符串与其他变量间可以用加号 (+) 表示连接。`System.out` 的 `print()` 方法可以输出一个字符串并换行。如例 2-3，输入一个字符，并显示这个字符。

例 2-3 `AppCharInOut.java` 字符的输入与输出。

```
import java.io.*;

public class AppCharInOut
{
    public static void main(String[] args)
    {
        char c = ' ';
        System.out.print("Please input a char: ");
        try{
            c = (char) System.in.read();

        }catch(IOException e){}
        System.out.println("You have entered: " + c );
    }
}
```

运行结果如图 2-6 所示。

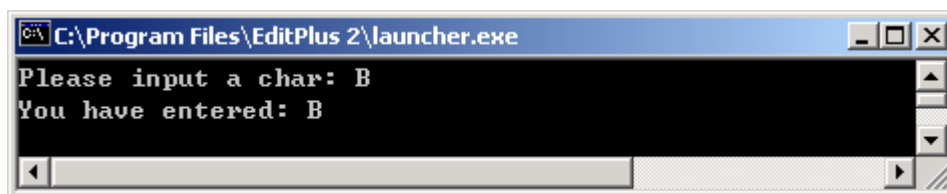


图 2-6 字符的输入与输出

该例中，要用 `try{}catch` 方式来捕获输入与输出中出现的异常。

`System.in` 的 `read()` 方法只能读入一个字符，不便于使用，例 2-4 中，将 `System.in` 进行“包装”，用它构造 (new) 出一个 `InputStreamReader` 对象，进而构造出一个 `BufferedReader` 对象，而 `BufferedReader` 对象有一个 `readLine` 方法，可用于读入一串字符。

例 2-4 AppLineInOut.java 整行的输入。

```
import java.io.*;
public class AppLineInOut
{
    public static void main(String[] args)
    {
        String s = "";
        System.out.print("Please input a line: ");
        try{
            BufferedReader in = new BufferedReader(
                new InputStreamReader( System.in ) );
            s = in.readLine();
        }catch(IOException e){}
        System.out.println("You have entered: " + s );
    }
}
```

运行结果如图 2-7 所示。

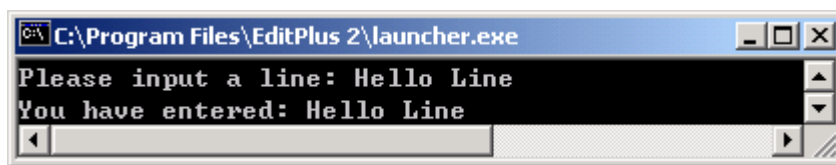


图 2-7 整行的输入

例 2-4 为输入一行文字并显示。有时，还需要将输入的字符串转成数字（如整数 `int` 或实数 `double`）。这时，可用 `Integer.parseInt` 及 `Double.parseDouble` 方法，如例 2-5 所示。

例 2-5 AppNumInOut.java 数字的输入与输出。

```
import java.io.*;
public class AppNumInOut
{
    public static void main(String[] args)
    {
        String s = "";
        int n = 0;
        double d = 0;
        try{
            BufferedReader in = new BufferedReader(
```

```
        new InputStreamReader( System.in ) );  
        System.out.print("Please input an int: ");  
        s = in.readLine();  
        n = Integer.parseInt( s );  
        System.out.print("Please input a double: ");  
        s = in.readLine();  
        d = Double.parseDouble( s );  
    }catch(IOException e){}  
    System.out.println("You have entered: " + n + " and " + d );  
}  
}
```

运行结果如图 2-8 所示。

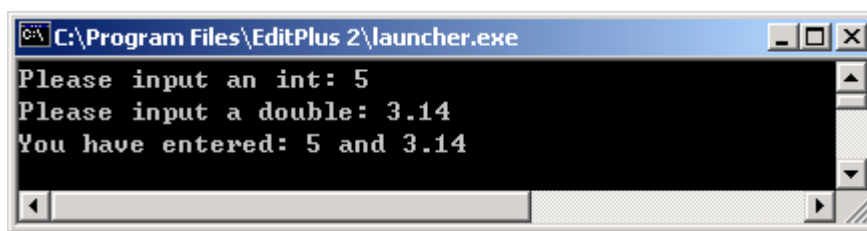


图 2-8 数字的输入与输出

2.3.2 Applet 图形界面输入与输出

Java Applet 程序需要在 WWW 浏览器中运行，而浏览器本身是图形界面的环境，所以 Java Applet 程序可以且只能在图形界面下工作。

图形界面最基本的输入与输出手段是使用文本框对象（TextField）获取用户输入的数据，使用标签对象(Label)或文本框对象输出数据，使用命令按钮(Button)来执行命令。

例 2-6 AppletInOut.java 图形界面输入与输出。

```
import java.applet.*;  
import java.awt.*;  
import java.awt.event.*;  
  
public class AppletInOut extends Applet  
{  
    TextField in = new TextField(10);  
    Button btn = new Button("求平方");  
    Label out = new Label("用于显示结果的标签");
```

```
public void init()
{
    setLayout( new FlowLayout() );
    add( in );
    add( btn );
    add( out );
    btn.addActionListener( new BtnActionAdapter() );
}

class BtnActionAdapter implements ActionListener
{
    public void actionPerformed((ActionEvent e) )
    {
        String s = in.getText();
        double d = Double.parseDouble( s );
        double sq = d * d;
        out.setText( d + "的平方是: " + sq );
    }
}
```

在本程序中，生成了一个文本框 `in` 用于输入，一个标签 `out` 用于输出，一个按钮 `btn` 用于触发命令。在 `Applet` 的 `init()`(初始化)方法中，设定布局方式为流式布局(`FlowLayout`)，然后将这三个对象加入。在程序中，还有一点很关键，就是加入一个事件的监听对象。事件监听对象是当用户单击此按钮时，事件监听对象的 `actionPerformed()`方法被调用，该方法中，通过 `getText()`方法得到用户的输入，然后用 `Double.parseDouble()`方法转为一个实数 (`double`)，再计算其平方，用 `Label` 的 `setText()`方法显示其平方值。运行结果如图 2-9 所示。

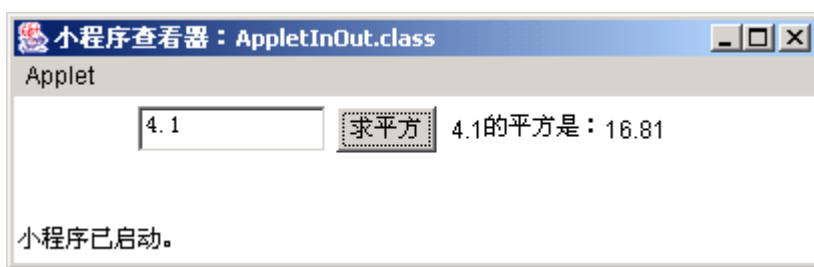


图 2-9 Applet 图形界面输入与输出

2.3.3 Java Application 图形界面输入与输出

与 Java Applet 程序不同，Java Application 程序没有浏览器提供的现成的图形界面可以直接使用，所以需要首先创建自己的图形界面。

例 2-7 AppGraphInOut.java Application 图形界面输入与输出。

```
import java.awt.*;
import java.awt.event.*;

public class AppGraphInOut
{
    public static void main( String args[] )
    {
        new AppFrame();
    }
}

class AppFrame extends Frame
{
    TextField in = new TextField(10);
    Button btn = new Button("求平方");
    Label out = new Label("用于显示结果的标签");

    public AppFrame()
    {
        setLayout( new FlowLayout() );
        add( in );
        add( btn );
        add( out );
        btn.addActionListener( new BtnActionAdapter() );
        setSize( 400,100 );
        show();
    }
}

class BtnActionAdapter implements ActionListener
{
    public void actionPerformed( ActionEvent e )
    {
        String s = in.getText();
```

```

        double d = Double.parseDouble( s );
        double sq = d * d;
        out.setText( d + "的平方是: " + sq );
    }
}

```

运行结果如图 2-10 所示。



图 2-10 Application 图形界面输入与输出

本例中，通过创建一个 `Frame`（带框架的窗口），从而创建自己的用户界面，在构造 `AppFrame` 时，设定了该 `Frame` 的大小（`setSize`），并用 `show()` 方法显示出来。

2.3.4 同时作为 Application 与 Applet 的程序

为了使同一程序既能作为 `Applet`，又能作为 `Application`，这就需要具备以下条件：

- （1）它是 `Applet` 的派生(`extends Applet`)；
- （2）它含有 `main()`，以便作为 `Application`；
- （3）在 `main` 中创建一个用户界面(如 `Frame`)，并将这个 `Applet` 加入。

例 2-8 `AppAppletInOut.java` 同时作为 `Application` 与 `Applet` 的程序。

```

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class AppAppletInOut extends Applet
{
    public static void main( String args[] )
    {
        Frame frame = new Frame();
        AppAppletInOut app = new AppAppletInOut();
        app.Init();
        frame.add( app );
        frame.setSize( 400,100 );
    }
}

```



```
        frame.show();
    }

    TextField in = new TextField(10);
    Button btn = new Button("求平方");
    Label out = new Label("用于显示结果的标签");

    public void Init()
    {
        setLayout( new FlowLayout() );
        add( in );
        add( btn );
        add( out );
        btn.addActionListener( new BtnActionAdapter() );
    }

    class BtnActionAdapter implements ActionListener
    {
        public void actionPerformed((ActionEvent e) )
        {
            String s = in.getText();
            double d = Double.parseDouble( s );
            double sq = d * d;
            out.setText( d + "的平方是: " + sq );
        }
    }
}
```

运行结果如图 2-11 所示。



图 2-11 运行结果

2.4 Java 集成开发环境

有了基本的 JDK 工具包，就可以进行 Java 程序的开发了。在实际编程时，还可以借

助一些辅助工具来加快程序的设计。这些工具有的功能简单，有的功能强大。

本节简要介绍几种简单的辅助工具及集成开发环境。

2.4.1 几种辅助工具的使用

在 Java 的辅助工具中，有许多是比较小巧的，它们的主要功能有两点：

- (1) 提供一个编辑器，能编辑 Java 程序及 HTML 文件；
- (2) 用菜单或快捷键方便地调用 Javac 及 Java, AppletViewer 来编译和运行 Java 程序。

这样的辅助工具主要有：JCreator, Editplus, Kawa, Freejava, UltraEdit 等。它们是免费软件或共享软件，可以从 Internet 下载后安装并使用。当然在安装这些软件工具之前，系统中必须首先安装 JDK。

1. JCreator, Kawa 及 Freejava

JCreator, Kawa 及 Freejava 的使用方式相似，下面以 JCreator 为例进行介绍。

JCreator 是一个非常好用的工具，有两种版本，JCreator LE 及 JCreator Pro，前者是免费软件，后者是共享软件，功能相似。但后者有一个很实用的功能，即能即时提示各个类的方法名及相关的参数，这可以大大提高程序输入的速度并减少输入的错误。

JCreator 的界面如图 2-12 所示。

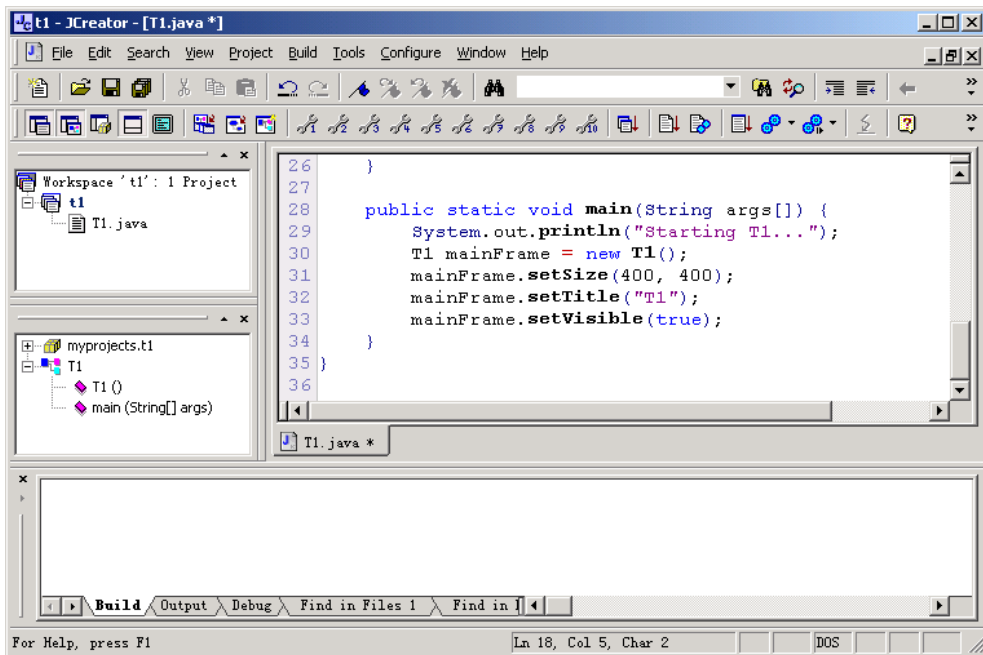


图 2-12 JCreator 的界面

界面中，左上窗格是文件列表，左下窗格是各个类的列表，中间的主要窗格是编辑窗格，下面的窗格是信息及输出窗格。

JCreator 在组织文件时，有下面三个层次：

- ✎ 文件 (File)，是各种 Java 程序及 HTML 文件等；
- ✎ 工程 (Project)，由一组相关的文件组成；
- ✎ 工作区 (Workspace)，由一组工程所组成。

JCreator 中可以用 File/New 来创建文件、工程或工作区。常用的方法是，新建一个工程，然后在工程中新建文件。

如果有已经建立好的 Java 文件，也可以从资源管理器中或【我的电脑】中将 Java 文件拖放到 JCreator，然后即可进行编辑。

JCreator 中编译和运行 Java 程序十分简单，直接点击工具栏上的编译、运行按钮



即可。这时 JCreator 会自动调用 JDK 中的 Javac 及 Java 来编译和运行 Java 程序。

2. EditPlus 和 UltraEdit

EditPlus 与 UltraEdit 的使用方式相似，下面以 EditPlus 为例进行介绍。

EditPlus 是共享软件，它的主要功能是文本编辑，对编辑 Java 程序及 HTML 网页也有较好的支持。在编辑时，对于一些重要的关键词还以醒目的颜色显示出来，这样可以使阅读程序更加方便，也有助于减少键入错误。

EditPlus 界面如图2-13 所示。左边为文件夹及文件的显示区，中间为编辑窗口，下边为信息窗口。

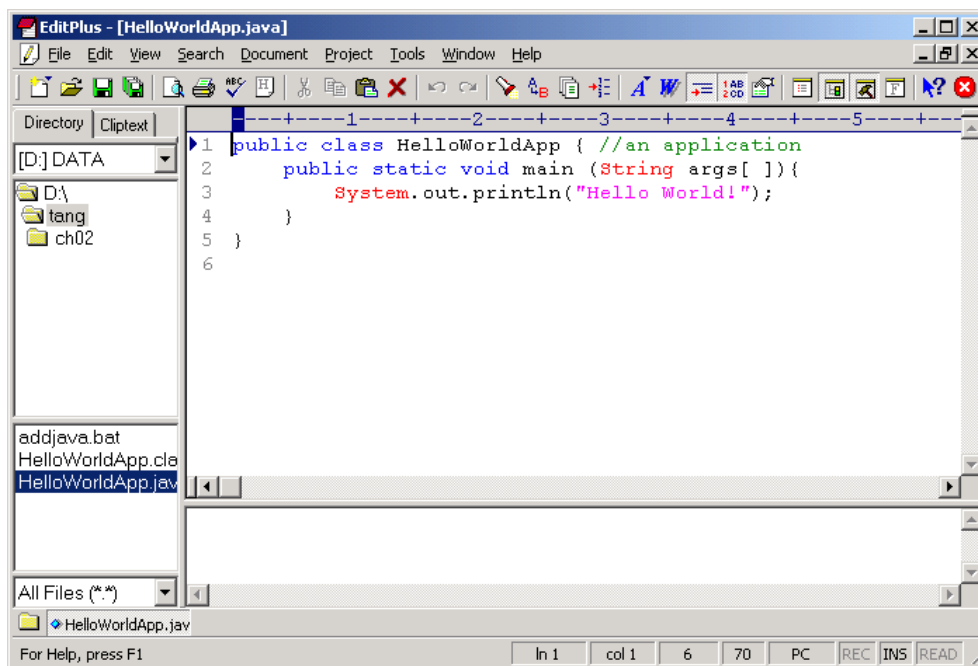


图 2-13 EditPlus 界面

为了方便在 EditPlus 中调用编译及运行功能，需要设置 User Tools(用户工具)。选择菜

单【Tools/Configure User Tools】，在弹出的对话框中，点【Add Tool】加入用户工具，如图 2-14 所示。

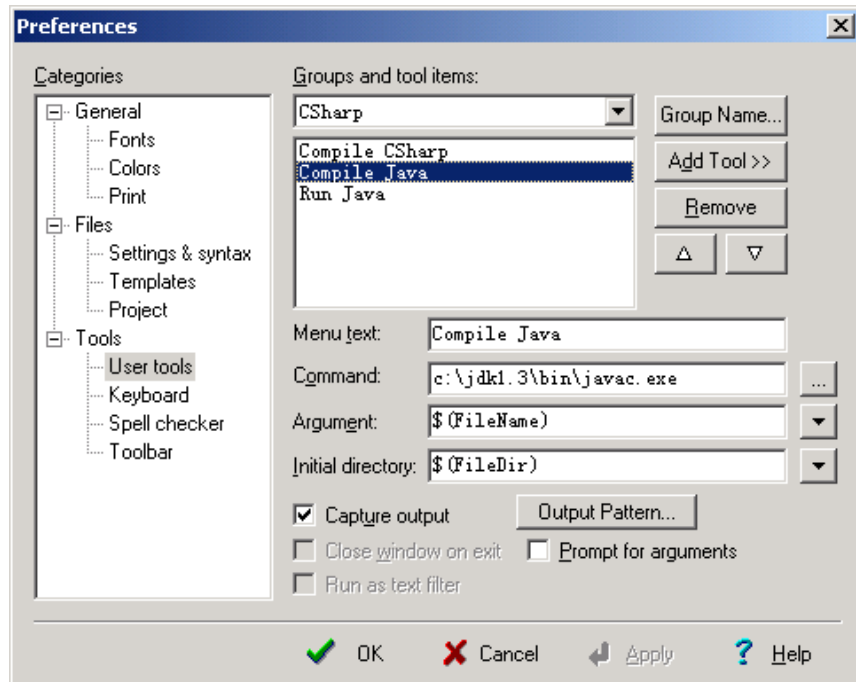


图 2-14 设置 User Tools

对于编译及运行分别设置如表 2-1 所示。

表 2-1 设置 User Tools 的值

	编 译	运 行
Menu text	Compile Java	Run Java
Command	c:\jdk1.3\bin\javac.exe	c:\jdk1.3\bin\java.exe -classpath
Argument	\$(FileName)	\$(FileNameNoExt)
Initial directory	\$(FileDir)	\$(FileDir)
Capture output	选择	不选择

设置好以后，可以按快捷键 Ctrl+1 及 Ctrl+2 来进行编译和运行。当然也可以只用 EditPlus 来编辑程序，然后在命令行状态下用 JDK 的命令进行编译和运行。

2.4.2 几种集成工具的使用

在实际的项目开发时，还经常用到一些功能更强、集成度更好的开发工具，它们集编辑、编译、调试、打包、发布等功能于一体，被称为集成开发环境（IDE）。这些 IDE 还具有可视化的界面设计、文档自动编写、辅助软件设计等功能。常用的 IDE 包括：Borland 公司出品的 JBuilder, Sun 公司出品的 Java Workshop, IBM 公司的 Visual Age for Java, Oracle 公司的 Java

Develop, 等等。另外, Symantec 公司的 Visual Café 也是著名的 Java 开发工具。

这里以 JBuilder 为例进行介绍。

1. JBuilder 的简单使用

JBuilder 将相关的文件 (Java 程序、HTML 网页、图片等) 组织成 Project (工程)。整个工程存于一个目录及其子目录下。JBuilder 中可以建立各种工程和文件, 其中包括常用的 Application 及 Applet。

在 JBuilder 的界面中, 主要有文件 (工程) 窗口、类窗口、编辑窗口, 如图 2-15 所示。

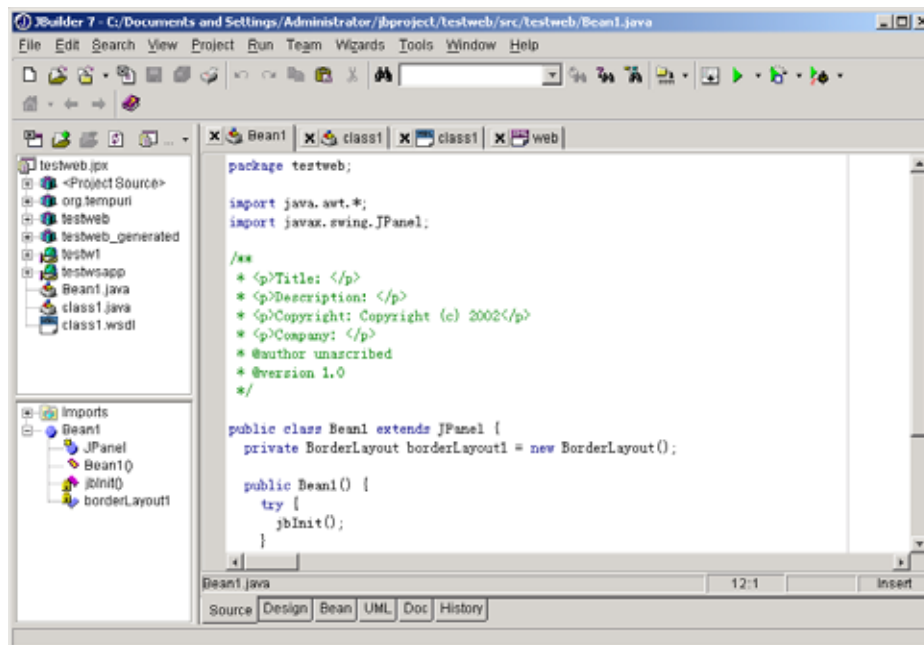


图 2-15 JBuilder 的界面

在 JBuilder 中, 文件可以有如下的多种视图。

- ☞ Source: 源程序, 可以进行源程序的编辑。
- ☞ Design: 设计视图, 可以进行图形化界面设计。
- ☞ Bean: Java Bean 视图, 可以进行 JavaBean 的设计。
- ☞ UML: UML 视图, 可以进行 UML 的设计及查看。
- ☞ Doc: 自动生成的文档视图。
- ☞ History: 历史列表。

对于初学者, 主要用到 Source 视图及 Design 视图, 分别进行代码的编写及界面的设计。其中, 可视化界面设计是十分方便的。如图 2-16 所示, 在 Design 视图中, 顶部是一排工具对象, 中部是要设计的界面, 右部是各个对象的属性的设置区。对于一个 Frame 及 Applet, 可以在其中画出各种按钮、标签、文本框等对象。

注意：为了随意定位，需要将 Frame 或 Applet 的 Layout 属性设为 null。

关于 Layout，在本书后面章节进行讲解。

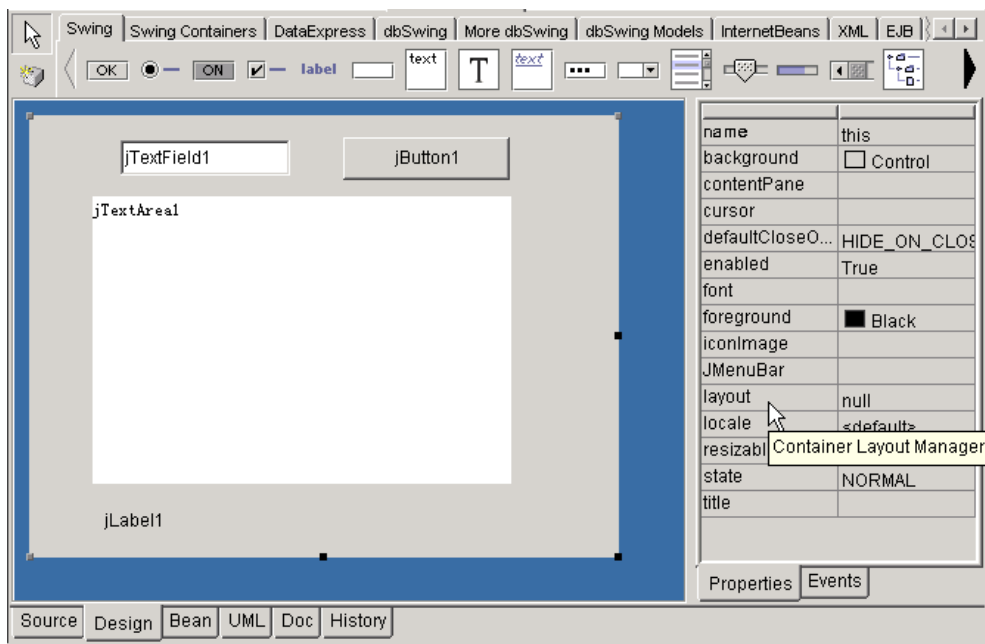


图 2-16 设计视图

2. JBuilder 的高级特性

JBuilder 将 Web 和企业应用开发与灵活高效的团队开发环境结合在一起，为开发者提供了一个端对端应用开发环境。JBuilder 支持最新的 Java 标准，它的可视化工具和向导使应用开发变得方便快捷。JBuilder 同时支持 Windows, Linux, Solaris 三种开发平台。由于 JBuilder 具有开放性、可扩展性及遵循标准等特点，使得用 JBuilder 开发符合 J2EE 标准的电子商务应用、分发 Internet 分布式的关键性企业级应用、建立互联数据库、设计数据驱动的动态网页应用，以及开发 Servlet、JSP 异常快捷，从而缩短产品推向市场的周期。JBuilder 支持最新的 Java 技术，包括 Applets, JSP/Servlets, JavaBeans, Enterprise JavaBeans, CORBA 应用。这里对 JBuilder 的一些高级特性做简单介绍，读者在今后的学习、开发过程中可以对它们有进一步的理解。

(1) 快速编程及可视化程序设计。

JBuilder 有一个可扩展的源码编辑器。它的代码向导自动访问相关 Java 对象的属性、事件、方法和包的上下文提示。同时，JBuilder 的可视化界面设计功能可以大大地缩短界面相关的程序的编写。

(2) 先进灵活的调试。

JBuilder 拥有专业化的图形界面调试，支持远程调试和多线程调试。调试器支持各种 JDK 版本，包括 J2ME, J2SE, J2EE，支持多 JDK。开发者可以在调试过程中设置条件断

点或跨进程断点。在断点处，可以检测、修改变量，同时查看多个线程，可以在本地或远程代码中单步调试定位。用调试器可以有效地排除线程死锁。在调试时可任意停止或忽略条件判断。远程调试还提供一个浏览界面，用于查看运行在多个 JVM 的所有进程，这些进程可能分布在各种操作系统之下，如 HP-UX, Tru64, AIX, 等等，支持 Java2 调试的任何环境。

(3) 开发各种动态 Web 应用。

JBuilder 向导提供的可视化工具，既可以创建客户端程序，也能创建由任意 Web 浏览器访问的纯 Java 程序。对于 Servlets，开发者可以在本地或远程的 Servlet 容器中对之进行调试，使用 Enterprise JavaBeans 进行快速开发和实施 J2EE 应用，并能与最新的 WebLogic, WebSphere 和新的 Borland AppServer 等应用平台提供无缝支持。

(4) XML 及 Web Service 的支持。

JBuilder 使用工具来定义结构和规则，数据绑定及对 XML 文档可编程的操作，还提供了对最新的 Web Service 的支持，可以方便地调用 Web Service，也可方便地将现有的程序转变为 Web Service。

(5) 创建专业数据库应用。

JBuilder 提供的 dbSwing 组件库可以很容易地开发数据库应用。开发者可以使界面对象直接与数据库相连，从而快速生成数据驱动的动态 Web 应用。

(6) 软件工程支持。

JBuilder 的集成团队开发环境非常灵活并具有可扩展的特性，简化了大型分散开发团队的代码的并发管理，方便协同开发和版本控制。另外，提供自动文档生成和 UML 支持，可以帮助开发者提高软件的开发效率。

3 . Java WorkShop 简介

Java WorkShop 是 Sun 公司的一个集成的 Java 语言开发环境，它包括以下工具。

(1) Project 管理器。Project 是应用程序或 Applet 的集合，它还包括编译、调试、运行和发布方式等信息。

(2) 源文件编辑器。源文件编辑器中可以输入源文件，并有一些实用的功能方便编辑。

(3) VisualJava 图形界面构造。本模块能可视化建造一个复杂的图形化界面。

(4) Build 管理工具。Build 管理工具是项目的编译器，在编译过程中，如果某个文件出错，会显示错误信息，并可以方便地定位出错的源文件地点。

(5) 调试器。调试器能让用户很方便地跟踪程序的执行并发现程序的错误。

习题

1. 简述 Java 编译和运行的基本方法。
2. 下载并安装 JDK 软件包，尝试查看其中的 JDK 文档。
3. 编写一个 Java Application，利用 JDK 软件包中的工具编译并运行这个程序，在屏幕上输出“Welcome to Java World!”。
4. 编写一个 Java Applet，使之能够在浏览器中显示“Welcome to Java Applet World!”

的字符串信息。

5. 编写一个 HTML 文件，将习题 4 中生成的 Applet 字节码嵌入其中，并用 WWW 浏览器观看这个 HTML 文件规定的 Web 页面。

6. 编写一个程序，能够从键盘上接收两个数字，然后计算这两个数的积。

7. 编写一个程序，从两个文本框中接收两个数字，然后计算这两个数的积。

8. 常用的集成开发工具有哪些？各有什么特点？

第 3 章 数据运算、流控制和数组

本章主要介绍编写 Java 程序必须了解的若干语言基础知识，包括数据类型、变量、常量、表达式和流程控制语句、数组等。掌握这些基础知识，是编写正确 Java 程序的前提条件。

3.1 数据类型、变量与常量

3.1.1 数据类型

在程序设计中，数据是程序的必要组成部分，也是程序处理的对象。不同的数据有不同的数据类型，不同的数据类型有不同的数据结构、不同的存储方式，并且参与的运算也不相同。

Java 中的数据类型分为两大类，一类是基本数据类型（primitive types），另一类是引用类型（reference types）。

Java 中定义了 4 类、8 种基本数据类型见表3-1。

表 3-1 Java 的基本数据类型

数据类型	关键字	占用字节数	默认数值	取值范围
布尔型	boolean	1	false	true, false
字节型	byte	1	0	-128~127
短整型	short	2	0	-32768~32767
整型	int	4	0	-2147483648~2147483647
长整型	long	8	0L	-9223372036854775808~9223372036854775807
单精度浮点型	float	4	0F	$1.4\times 10^{-45}\sim 3.4\times 10^{38}$
双精度型	double	8	0D	$4.9\times 10^{-324}\sim 1.8\times 10^{308}$
字符型	char	2	'\u0000'	'\u0000' ~'\uffff'

1．逻辑型——boolean

boolean 是用来表示布尔型(逻辑)数据的数据类型。boolean 型的变量或常量的取值只有 true 和 false 两个。其中，true 代表“真”，false 代表“假”。

2．整数型——byte, short, int, long

4 种整数型 byte（字节型），short（短整型），int（整型），long（长整型）在内存中所占长度不同，分别是 1，2，4，8 字节。Java 的各种数据类型占用固定的内存长度，与具体的软硬件平台环境无关，体现了 Java 的跨平台特性。Java 的每种数据类型都对应一个默认的数值，使得这种数据类型变量的取值总是确定的，体现了其安全性。

3．浮点型——float, double

float（单精度实数）及 double(双精度实数)在计算机中分别占 4 字节和 8 字节，它们所

表达的实数的精度和范围是不同的。

4. 字符型——char

char(字符型)是用 Unicode 编码表达的字符，在内存中占两个字节。由于 Java 的字符类型采用了一种新的国际标准编码方案——Unicode 编码，这样便于东方字符和西方字符的处理。因此，与其他语言相比，Java 处理多语种的能力大大加强。

除基本数据类型外，Java 中还存在一种引用数据类型，包括数组(array)、类(class)和接口(interface)。这些类型在以后的章节中再进行介绍。

3.1.2 标识符

任何一个变量、常量、方法、对象和类都需要有名字，这些名字就是标识符。标识符可以由编程者自由指定，但是需要遵循一定的语法规则。标识符要满足如下的规定：

- (1) 标识符可以由字母、数字和下划线(_)、美元符号(\$)组合而成；
- (2) 标识符必须以字母、下划线或美元符号开头，不能以数字开头。

在实际应用标识符时，应该使标识符能一定程度上反映它所表示的变量、常量、对象或类的意义，这样程序的可读性会更好。

同时，应注意 Java 是大小写敏感的语言。例如，class 和 Class，System 和 system 分别代表不同的标识符，在定义和使用时要特别注意这一点。

在 Java 编程时，经常遵循以下的编码习惯（虽然不是强制性的）：类名首字母应该大写；变量、方法及对象（句柄）的首字母应小写。对于所有标识符，其中包含的所有单词都应紧靠在一起，而且大写中间单词的首字母。例如，ThisIsAClassName，thisIsMethodOrFieldName。若在定义中出现了常数初始化字符，则大写所有字母。这样便可标志出它们属于编译期的常数。Java 包（Package）属于一种特殊情况，它们全都是小写字母，即便中间的单词亦是如此。

3.1.3 常量

常量是在程序运行的整个过程中保持其值不改变的量。Java 中常用的常量有布尔常量、整型常量、字符常量、字符串常量和浮点常量。

1. 布尔常量

布尔常量包括 true 和 false，分别代表真和假。

2. 整型常量

整型常量可以用来给整型变量赋值，整型常量可以采用十进制、八进制和十六进制表示。十进制的整型常量用非 0 开头的数值表示，如 100，-50；八进制的整型常量用以 0 开头的数字表示，如，017 代表十进制的数字 15；十六进制的整型常量用 0x 开头的数值表示，如 0x2F 代表十进制的数字 47。

整型常量按照所占用的内存长度，又可分为一般整型常量和长整型常量，其中一般整

型常量占用 32 位，长整型常量占用 64 位，长整型常量的尾部有一个大写的 L 或小写的 l，如 -386L，0l，7777l。

3. 浮点常量

浮点常量表示的是可以含有小数部分的数值常量。根据占用内存长度的不同，可以分为一般浮点(单精度)常量和双精度浮点常量两种。其中，单精度常量后跟一个 f 或 F，双精度常量后跟一个 d 或 D。双精度常量后的 d 或 D 可以省略。

浮点常量可以有普通的书写方法，如 3.14f，-2.17d，也可以用指数形式，如 5.3e-2 表示 5.3×10^{-2} ，123E3D 代表 123×10^3 （双精度）。

4. 字符常量

字符常量用一对单引号括起的单个字符表示，如 'A'，'1'。字符可以直接是字母表中的字符，也可以是转义符，还可以是要表示的字符所对应的八进制数或 Unicode 码。

转义符是一些有特殊含义、很难用一般方式表达的字符，如回车、换行等。为了表达清楚这些特殊字符，Java 中引入了一些特别的定义。所有的转义符都用反斜线(\)开头，后面跟着一个字符来表示某个特定的转义符，如表 3-2 所示。

表 3-2 转义符

转 义 字 符	含 义
\ddd	1 到 3 位八进制数所表示的字符(ddd)
\uxxxx	1 到 4 位十六进制数所表示的字符(xxxx)
\'	单引号字符
\"	双引号字符
\\	反斜杠字符
\r	回车
\n	换行
\f	走纸换页
\t	横向跳格
\b	退格

5. 字符串常量

字符串常量是用双引号括起的一串若干个字符(可以是 0 个)。字符串中可以包括转义符，标志字符串开始和结束的双引号必须在源代码的同一行上。

如: "Hello world\n".

3.1.4 变量

变量是在程序的运行过程中数值可变的数据，通常用来记录运算中间结果或保存数据。Java 中的变量必须先声明后使用，声明变量包括指明变量的数据类型和变量的名称，必要时还可以指定变量的初始数值。变量声明后要用分号。

如:

```
int a,b,c;
double x = 12.3;
```

例 3-1 DeclareAssign.java 声明并赋值。

```
public class DeclareAssign
{
    public static void main (String args []) {
        boolean b = true;           // 声明 boolean 型变量并赋值
        int x, y=8;                  // 声明 int 型变量
        float f = 4.5f;              // 声明 float 型变量并赋值
        double d = 3.1415;           // 声明 double 型变量并赋值
        char c;                      // 声明 char 型变量
        c = '\u0031';                // 为 char 型变量赋值
        x = 12;                      // 为 int 型变量赋值
        System.out.println("b=" + b);
        System.out.println("x=" + x);
        System.out.println("y=" + y);
        System.out.println("f=" + f);
        System.out.println("d=" + d);
        System.out.println("c=" + c);
    }
}
```

3.1.5 程序的书写与注释

Java 程序中最基本的成分是常量、变量、运算符等。除这些成分外，Java 程序中还有注释。注释虽然对程序的运行不起作用，但对于程序的易读性具有重要的作用。

Java 中可以采用三种注释方式：

- (1) // 用于单行注释。注释从//开始，终止于行尾；
- (2) /* ... */ 用于多行注释。注释从/*开始，到*/结束，且这种注释不能互相嵌套；
- (3) /** ... */ 是 Java 所特有的 doc 注释。它以/**开始，到*/结束。

其中，第 3 种注释主要是为支持 JDK 工具 javadoc 而采用的。javadoc 能识别注释中用标记@标识的一些特殊变量，并把 doc 注释加入它所生成的 HTML 文件。常用的@标记如下。

- ☞ @see: 引用其他类。
- ☞ @version: 版本信息。
- ☞ @author: 作者信息。
- ☞ @param: 参数名说明。
- ☞ @return: 说明。
- ☞ @exception: 完整类名说明。

对于有@标记的注释，javadoc 在生成有关程序的文档时，会自动地识别它们，并生成相应的文档。

例 3-2 HelloDate.java 加入了注释的程序，以用于 javadoc。

```
/** A Simple Java example program.  
 * @(#)HelloDate.java 1.5 98/07/09  
 *  
 * Displays a string and today's date.  
 * @author Bruce Eckel  
 * @author www.BruceEckel.com  
 * @version 2.0  
 */  
import java.util.*;  
public class HelloDate {  
    /** Sole entry point to class & application  
     * @param args array of string arguments  
     * @return No return value  
     * @exception exceptions No exceptions thrown  
     */  
    public static void main(String[] args) {  
        System.out.println("Hello, it's: ");  
        System.out.println(new Date());  
    }  
}
```

使用 javadoc 的命令为：

```
javadoc HelloDate.java
```

这就可以自动生成文档。生成的文档如图3-1 所示。

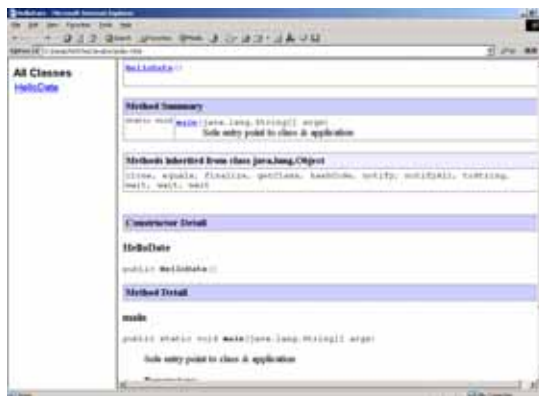


图 3-1 用 javadoc 生成的文档

3.2 运算符与表达式

运算符指明对操作数所进行的运算。按操作数的数目来分，可以有一元运算符（如++）二元运算符（如+、>）和三元运算符（如?:），它们分别对应于一个、两个和三个操作数。对于一元运算符来说，可以采用前缀表达式(如++i)和后缀表达式(如 i++)，对于二元运算符来说则采用中缀表达式(如 a+b)。按照运算符功能来分，基本的运算符有下面几类。

- (1) 算术运算符(+, -, *, /, %, ++, --)。
 - (2) 关系运算符(>, <, >=, <=, ==, !=)。
 - (3) 布尔逻辑运算符(!, &&, ||, &, |)。
 - (4) 位运算符(>>, <<, >>>, &, |, ^, ~)。
 - (5) 赋值运算符(=, 及其扩展赋值运算符如+=)。
 - (6) 条件运算符(?:)。
 - (7) 其他（包括分量运算符·，下标运算符[]，实例运算符 instanceof，内存分配运算符 new，强制类型转换运算符 (类型)，方法调用运算符 () 等)。
- 本节中主要讲述前 6 类运算符。

3.2.1 算术运算符

算术运算符作用于整型或浮点型数据，完成算术运算。

1．二元算术运算符

二元算术运算符如表 3-3 所示。

表 3-3 二元算术运算符		
运 算 符	用 法	描 述
+	op1 + op2	加
-	op1 - op2	减
*	op1 * op2	乘
/	op1 / op2	除
%	op1 % op2	取模（求余）

对取模运算符%来说，其操作数可以为浮点数，如 37.2%10=7.2。

值得注意的是 Java 对加运算符进行了扩展，使它能够进行字符串的连接，如 “abc” + “de”，得到字符串 “abcde”。

2．一元算术运算符

一元算术运算符如表 3-4 所示。

表 3-4 一元算术运算符

运 算 符	用 法	描 述
+	+ op	正值
-	- op	负值
++	++ op, op ++	加 1
--	-- op, op --	减 1

注意：++及--运算符可以置于变量前，也可以置于变量后。i++与++i，都会使 i 的值加 1，但作为表达式，i++与++i 是有区别的：

i++在使用 i 之后，使 i 的值加 1，因此执行完 i++后，整个表达式的值为 i，而 i 的值变为 i+1。

++i 在使用 i 之前，使 i 的值加 1，因此执行完++i 后，整个表达式和 i 的值均为 i+1。

对 i--与--i 来说，与上述情况一样。

下面的例子说明了算术运算符的使用。

例 3-3 ArithmeticOp.java 算术运算符的使用。

```
public class ArithmeticOp{
    public static void main( String args[] ){
        int a=5+4; //a=9
        int b=a*2; //b=18
        int c=b/4; //c=4
        int d=b-c; //d=14
        int e=-d; //e=-14
        int f=e%4; //f=-2
        double g=18.4;
        double h=g%4; //h=2.4
        int i=3;
        int j=i++; //i=4,j=3
        int k=++i; //i=5,k=5
        System.out.println("a = "+a);
        System.out.println("b = "+b);
        System.out.println("c = "+c);
        System.out.println("d = "+d);
        System.out.println("e = "+e);
        System.out.println("f = "+f);
        System.out.println("g = "+g);
        System.out.println("h = "+h);
        System.out.println("i = "+i);
    }
}
```

```
System.out.println("j = "+j);
System.out.println("k = "+k);
}
}
```

其结果为：

```
C:\>java ArithmeticOp
a = 9
b = 18
c = 4
d = 14
e = -14
f = -2
g = 18.4
h = 2.4
i = 5
j = 3
k = 5
```

3.2.2 关系运算符

关系运算符用来比较两个值，返回布尔类型的值 true 或 false。关系运算符都是二元运算符，如表 3-5 所示。

表 3-5 关系运算符

运 算 符	用 法	返回 true 的情况
>	op1 > op2	op1 大于 op2
>=	op1 >= op2	op1 大于或等于 op2
<	op1 < op2	op1 小于 op2
<=	op1 <= op2	op1 小于或等于 op2
==	op1 == op2	op1 与 op2 相等
!=	op1 != op2	op1 与 op2 不相等

Java 中，任何数据类型的数据（包括基本类型和引用类型）都可以通过==或!=来比较是否相等。关系运算的结果返回 true 或 false，而不是 C 或 C++中的 1 或 0。

关系运算符常与布尔逻辑运算符一起使用，作为流控制语句的判断条件。

例如：

```
if( a>b && b==c)
```

3.2.3 逻辑运算符

逻辑运算是针对布尔型数据进行的运算，运算的结果仍然是布尔型量，如表 3-6 所示。

表 3-6 逻辑运算符

运 算 符	运 算	用 法	描 述
&	逻辑与	op1 & op2	两操作数均为 true 时，结果才为 true
	逻辑或	op1 op2	两操作数均为 false 时，结果才为 false
!	取反	!op	与 op 的 true 或 false 相反
^	异或	op1 ^ op2	两操作数同真假时，结果才为 false
&&	简洁与	op1 && op2	两操作数均为 true 时，结果才为 true
	简洁或	op1 op2	两操作数均为 false 时，结果才为 false

! 为一元运算符，实现逻辑非。&、| 为二元运算符，实现逻辑与、逻辑或。简洁运算（&&、||）与非简洁运算（&、|）的区别在于：非简洁运算在计算左右两个表达式之后，最后才取值；简洁运算可能只计算左边的表达式而不计算右边的表达式，即对于&&，只要左边表达式为 false，就不计算右边表达式，则整个表达式为 false；对于||，只要左边表达式为 true，就不计算右边表达式，则整个表达式为 true。

下面的例子说明了关系运算符和布尔逻辑运算符的使用。

例 3-4 RelationAndConditionOp.java 关系和逻辑运算符的使用。

```
public class RelationAndConditionOp{
    public static void main( String args[] ){
        int a=25,b=3;
        boolean d=a<b; //d=false
        System.out.println("a<b = "+d);
        int e=3;
        if(e!=0 && a/e>5)
            System.out.println("a/e = "+a/e);
        int f=0;
        if(f!=0 && a/f>5)
            System.out.println("a/f = "+a/f);
        else
            System.out.println("f = "+f);
    }
}
```

其运行结果为：

```
a<b = false
a/e = 8
f = 0
```

注意：例 3-4 中，第二个 if 语句在运行时不会发生除 0 溢出的错误，因为 e!=0 为 false，所以就不需要对 a/e 进行运算。

3.2.4 位运算符

位运算符用来对二进制位进行操作，Java 中提供了如表 3-7 所示的位运算符。

表 3-7 位运算符

运 算 符	用 法	描 述
~	~ op	按位取反
&	op1 & op2	按位与
	op1 op2	按位或
^	op1 ^ op2	按位异或
>>	op1 >> op2	op1 右移 op2 位
<<	op1 << op2	op1 左移 op2 位
>>>	op1 >>> op2	op1 无符号右移 op2 位

位运算符中，除~以外，其余均为二元运算符。操作数只能为整型和字符型数据。有的符号（如&、|、^）与逻辑运算符的写法相同，但逻辑运算符的操作数为 boolean 型。

为了帮助不熟悉位运算的读者，特介绍位运算及其应用。

1. 补码

Java 使用补码来表示二进制数，在补码表示中，最高位为符号位，正数的符号位为 0，负数为 1。补码的规定如下。

（1）对正数来说，最高位为 0，其余各位代表数值本身(以二进制表示)，如+42 的补码为 00101010。

（2）对负数而言，把该数绝对值的补码按位取反，然后对整个数加 1，即得该数的补码。如，-42 的补码为 11010110（因为：00101010 按位取反 11010101，再加 1 后为 11010110）。

用补码来表示数，0 的补码是惟一的，都为 00000000（而在原码，反码表示中，+0 和 -0 的表示是不惟一的，可参见相应的书籍）。而且，可以用 111111 表示-1 的补码（这也是补码与原码和反码的区别）。

2. 按位取反运算符 ~

~是一元运算法，对数据的每个二进制位取反，即把 1 变为 0，把 0 变为 1。

例如：0010101 取反后为

1101010。

注意：~ 运算符与 - 运算符不同，~ 21 -21。

3. 按位与运算符 &

参与运算的两个值，如果两个相应位都为 1，则该位的结果为 1，否则为 0。即，0 & 0 = 0，0 & 1 = 0，1 & 0 = 0，1 & 1 = 1。

例如： 00101010

(&) 00010111

00000010

按位与可以用来对某些特定的位清零。如，对数 11010110 的第 2 位和第 5 位清零，可

让该数与 11101101 进行按位与运算。

```
例如:   11010110
(&)      11101101
         11000100
```

按位与可以用来取某个数中某些指定的位。如, 要取数 11010110 的第 2 位和第 5 位, 可让该数与 00010010 进行按位与运算。

```
例如:   11010110
(&)      00010010
         00010010
```

4. 按位或运算符 |

参与运算的两个值, 只要两个相应位中有一个为 1, 则该位的结果为 1。即: $0|0=0$, $0|1=1$, $1|0=1$, $1|1=1$ 。

```
例如:   00101010
(|)      00010111
         00111111
```

按位或可以用来把某些特定的位置 1, 如对数 11010110 的第 4、5 位置 1, 可让该数与 00011000 进行按位或运算。

```
例如:   11010110
(|)      00011000
         11011110
```

5. 按位异或运算符 ^

参与运算的两个值, 如果两个相应位相同, 则结果为 0, 否则为 1。即: $0^0=0$, $1^0=1$, $0^1=1$, $1^1=0$ 。

```
例如:   00101010
(^)      00010111
         00111101
```

按位异或可以用来使某些特定的位翻转 (求反), 如对数 11010110 的第 4、5 位求反, 可以将数与 00011000 进行按位异或运算。

```
例如:   11010110
(^)      00011000
         11001110
```

通过按位异或运算, 可以实现两个值的交换, 而不使用临时变量。例如, 交换两个整数 a、b 的值, 可通过下列语句实现:

```
a = 11010110, b = 01011001
a = a ^ b; // a = 10001111
b = b ^ a; // b = 11010110
a = a ^ b; // a = 01011001
```

6. 左移运算符 <<

用来将一个数的各二进制位全部左移若干位。例如， $a = a \ll 2$ ，使 a 的各二进制位左移 2 位，右补 0，若 $a = 00001111$ ，则 $a \ll 2 = 00111100$ 。高位左移后溢出，舍弃不起作用。

在不产生溢出的情况下，左移一位相当于乘 2，而且用左移来实现乘法比乘法运算速度要快。

7. 右移运算符 >>

用来将一个数的各二进制位全部右移若干位。例如， $a = a \gg 2$ ，使 a 的各二进制位右移 2 位，移到右端的低位被舍弃，最高位则移入原来高位的值。如， $a = 00110111$ ，则 $a \gg 2 = 00001101$ ； $b = 11010011$ ，则 $b \gg 2 = 11110100$ 。

右移一位相当于除 2 取商，而且用右移实现除法比除法运算速度要快。

8. 无符号右移运算符 >>>

用来将一个数的各二进制位无符号右移若干位，与运算符 \gg 相同，移出的低位被舍弃，但不同的是最高位补 0。如， $a = 00110111$ ，则 $a \ggg 2 = 00001101$ ； $b = 11010011$ ，则 $b \ggg 2 = 00110100$ 。

9. 不同长度的数据进行位运算

如果两个数据长度不同(如 `byte` 型和 `int` 型)，对它们进行位运算时，如 $a \& b$ ， a 为 `byte` 型， b 为 `int` 型，则系统首先会将 a 的左侧 24 位填满，若 a 为正，则填满 0，若 a 为负，则左侧填满 1。

10. 位运算的应用

位运算可以用来处理一组布尔标志。如果一个程序中有几个不同的布尔标志分别代表对象的几个性质的状态，则可以把它们放在一个变量中，通过对该变量进行位操作实现对各布尔变量的访问。下面的例子说明了这一过程。

例 3-5 BitwiseOp.java。

```
public class BitwiseOp{
    static String binary[]={ "0000", "0001", "0010", "0011",
        "0100", "0101", "0110", "0111",
        "1000", "1001", "1010", "1011",
        "1100", "1101", "1110", "1111"};

    static final int FLAG1=1; //make FLAG1 a constant(0x0001)
    static final int FLAG2=2; //make FLAG2 a constant(0x0010)
    static final int FLAG4=8; //make FLAG4 a constant(0x1000)
    public static void main( String args[] ){
```

```
        int flags=0; //clear all flags
        System.out.println("Clear all flags... flags="+binary[flags]);
        flags=flags | FLAG4; //set flag4
        System.out.println("Set flag4... flags="+binary[flags]);
        flags=flags ^ FLAG1; //revert flag1
        System.out.println("Revert flag1... flags="+binary[flags]);
        flags=flags ^ FLAG2; //revert flag2
        System.out.println("Revert flag2... flags="+binary[flags]);
        int cf1=~FLAG1;
        flags=flags & cf1; //clear flag1
        System.out.println("Clear flag1... flags="+binary[flags]);
        int f4=flags & FLAG4;
        f4=f4>>>3; //get flag4
        System.out.println("Get flag4... flag4="+f4);
        int f1=flags & FLAG1; //get flag1
        System.out.println("Get flag1... flag1="+f1);
    }
}
```

其结果为：

```
C:\>java BitwiseOp
Clear all flags... flags=0000
Set flag4... flags=1000
Revert flag1... flags=1001
Revert flag2... flags=1011
Clear flag1... flags=1010
Get flag4... flag4=1
Get flag1... flag1=0
```

3.2.5 赋值与强制类型转换

1. 赋值运算符

赋值运算符“=”把一个数据赋给一个变量，简单的赋值运算是把一个表达式的值直接赋给一个变量或对象，使用的赋值运算符是“=”，其格式如下：

变量或对象=表达式

在赋值运算符两侧的类型不一致的情况下，则需要自动或强制类型转换。变量从占用内存较少的短数据类型转化成占用内存较多的长数据类型时，可以不做显式的类型转换，Java 会自动转换；而将变量从较长的数据类型转换成较短的数据类型时，则必须做强制类型转换。强制类型的基本方式是：

(类型) 表达式

例如：

```
byte b=100;          // 自动转换
int i=b;
int i=100;           // 强制类型转换
byte b=(byte)a;
```

注意：当从其他类型转为 char 型时，必须用强制类型转换。

2. 扩展赋值运算符

在赋值符“=”前加上其他运算符，即构成扩展赋值运算符，例如： $a+=3$ 等价于 $a=a+3$ 。也就是：

```
var=var op expression
```

用扩展赋值运算符可表达为：

```
var op=expression
```

就是说，在先进行某种运算之后，再对运算的结果进行赋值。

表3-8 列出了 Java 中的扩展赋值运算符及等效的表达式。

表 3-8 扩展赋值运算

运 算 符	用 法	等效表达式
+=	$op1 += op2$	$op1 = op1 + op2$
-=	$op1 -= op2$	$op1 = op1 - op2$
*=	$op1 *= op2$	$op1 = op1 * op2$
/=	$op1 /= op2$	$op1 = op1 / op2$
%=	$op1 \% = op2$	$op1 = op1 \% op2$
&=	$op1 \& = op2$	$op1 = op1 \& op2$
=	$op1 = op2$	$op1 = op1 op2$
^=	$op1 \wedge = op2$	$op1 = op1 \wedge op2$
>>=	$op1 >> = op2$	$op1 = op1 >> op2$
<<=	$op1 << = op2$	$op1 = op1 << op2$
>>>=	$op1 >>> = op2$	$op1 = op1 >>> op2$

3.2.6 条件运算符

条件运算符 $?$ ：为三元运算符，它的一般形式为：

```
x ? y : z
```

其规则是，先计算表达式 x 的值，若 x 为真，则整个三元运算的结果为表达式 y 的值；若 x 为假，则整个运算结果为表达式 z 的值。其中， y 与 z 需要返回相同的数据类型。

例如：

```
ratio = denom==0 ? 0 : num/denom;
```

这里，如果 $denom==0$ ，则 $ratio=0$ ，否则 $ratio=num/denom$ 。

又如：

```
z = a>0 ? a : -a;    // z 为 a 的绝对值
```

```
z = a>b ? a : b;     // z 为 a、b 中较大值
```

如果要通过测试某个表达式的值来选择两个表达式中的一个进行计算时，用条件运算符来实现是一种简练的方法。这时，它实现了 if-else 语句的功能。

3.2.7 表达式及运算的优先级、结合性

表达式是由变量、常量、对象、方法调用和操作符组成的式子，它执行这些元素指定的计算并返回某个值。如，`a+b`，`c+d` 等都是表达式，表达式用于计算并对变量赋值，以及作为程序控制的条件。

在对一个表达式进行运算时，要按运算符的优先顺序从高向低进行，表 3-9 给出了 Java 中运算符的优先次序。大体上来说，从高到低是：一元运算符、算术运算、关系运算和逻辑运算、赋值运算。

表 3-9 运算符的优先级及结合性（表顶部的优先级较高）

运算符的优先级	运算符的结合性
<code>• [] ()</code>	
<code>++ -- ! ~ instanceof</code>	右
<code>new (type)</code>	右
<code>* / %</code>	左
<code>+ -</code>	左
<code>>>> >> <<</code>	左
<code>< > <= >=</code>	左
<code>== !=</code>	左
<code>&</code>	左
<code>^</code>	左
<code> </code>	左
<code>&&</code>	左
<code> </code>	左
<code>?:</code>	右
<code>= += -= *= /= %= ^=</code> <code>&= = <<= >>= >>>=</code>	右

在表达式中，可以用括号`()`显式地标明运算次序，括号中的表达式首先被计算。适当地使用括号可以使表达式的结构清晰。例如：

```
a>=b && c<d || e==f
```

可以用括号显式地写成

```
( (a<=b) && (c<d) ) || (e==f)
```

这样就清楚地表明了运算次序，使程序的可读性加强。

注意：括号的使用必须匹配。

运算符除有优先级外，还有结合性，运算符的结合性决定了并列的相同运算的先后执行顺序。大部分运算的结合性都是从左向右(称为“左结合性”)，赋值运算、条件运算则有右结合性。

3.3 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，是程序中基本又非常关键的部分。流程控制语句可以把单个的语句组合成有意义的、能完成一定功能的小逻辑模块。最主要的流程控制方式是结构化程序设计中规定的三种基本流程结构。

3.3.1 结构化程序设计的三种基本流程

任何程序都可以且只能由三种基本流程结构构成，即顺序结构、分支结构和循环结构。

顺序结构是三种结构中最简单的一种，即语句按照书写的顺序依次执行。分支结构又称为选择结构，它根据计算所得的表达式值来判断应选择执行哪一个流程的分支。循环结构则是在一定条件下反复执行一段语句的流程结构。这三种结构构成了程序局部模块的基本框架，如图 3-2 所示。

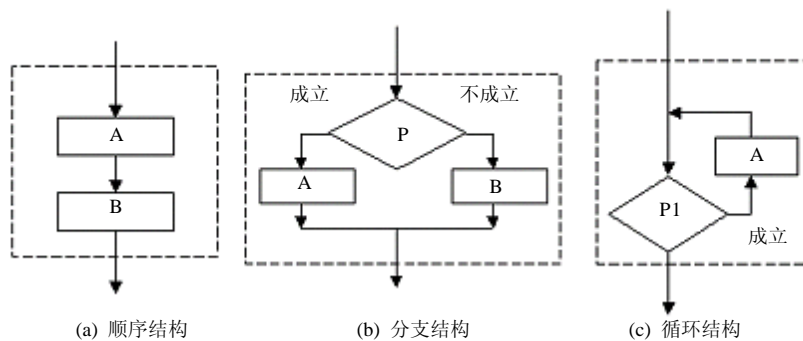


图 3-2 程序的三种流程

Java 语言虽然是面向对象的语言，但是在局部的语句块内部，仍然需要借助于结构化程序设计的基本流程结构来组织语句，完成相应的逻辑功能。Java 的语句块是由一对大括号括起的若干语句的集合。Java 中，有专门负责实现分支结构的条件分支语句和负责实现循环结构的循环语句。

3.3.2 简单语句

最简单的流程是顺序结构，在 Java 中一句一句地书写。而最简单的语句是方法调用语句及赋值语句，是在方法调用或赋值表达式后加一个分号 (;)，分别表示完成相关的任务及赋值。如：

```
System.out.println("Hello World");  
a = 3+x;  
b = a>0?a:-a;  
s = TextBox1.getText();  
d = Integer.parseInt( s );
```

3.3.3 分支语句

Java 中的分支语句有两个，一个是负责实现双分支的 if 语句，另一个是负责实现多分支的 switch 语句。

1. if 语句

if 语句的一般形式是：


```
if( 条件表达式)
    语句块;    // if 分支
else
    语句块;    // else 分支
```

其中，语句块是一条语句(带分号)或者用一对花括号{}括起来的一系列语句；条件表达式用来选择判断程序的流程走向，在程序的实际执行过程中，如果条件表达式的取值为真，则执行 if 分支的语句块，否则执行 else 分支的语句块。在编写程序时，也可以不书写 else 分支。此时，若条件表达式的取值为假，则绕过 if 分支直接执行 if 语句后面的其他语句。语法格式如下：

```
if( 条件表达式)
    语句块;    // if 分支
```

下面是一个 if 语句的简单例子，实现求某数的绝对值。

```
if( a>0 ) b=a; else b=-a;
```

例 3-6 LeapYear.java 判断闰年。

```
if(x<=0)
x=-x;
LeapYear.java
public class LeapYear{
    public static void main( String args[ ] ){
        int year=2003;
        if( (year%4==0 && year%100!=0) || (year%400==0) )
            System.out.println(year+" is a leap year.");
        else
            System.out.println(year+" is not a leap year.");
    }
}
```

该例判断某一年是否为闰年。闰年的条件是符合下面二者之一：①能被 4 整除，但不能被 100 整除；②能被 4 整除，又能被 100 整除。结果如图 3-3 所示。

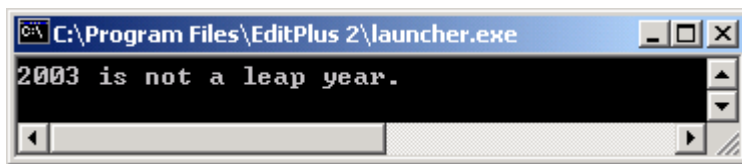


图 3-3 判断闰年

2. switch 语句

switch 语句是多分支的开关语句，一般形式是：

```
switch(表达式)
```

```
{  
    case 判断值 1: 一系列语句 1; break;  
    case 判断值 2: 一系列语句 2; break;  
    .....  
    case 判断值 n: 一系列语句 n; break;  
    default: 一系列语句 n+1  
}
```

注意：其中，表达式必须是整数型或字符类型；判断值必须是常数，而不能是变量或表达式。

switch 语句在执行时，首先计算表达式的值，这个值必须是整型或字符型，同时应与各个 case 分支的判断值的类型相一致。计算出表达式的值之后，将它先与第一个 case 分支的判断值相比较，若相同，则程序的流程转入第一个 case 分支的语句块；否则，再将表达式的值与第二个 case 分支相比较，……依此类推。如果表达式的值与任何一个 case 分支都不相同，则转而执行最后的 default 分支；在 default 分支不存在的情况下，则跳出整个 switch 语句。

注意：switch 语句的每一个 case 判断，在一般情况下都有 break 语句，以指明这个分支执行完成后，就跳出该 switch 语句。在某些特定的场合下可能不需要 break 语句，如在若干判断值共享同一个分支时，就可以实现由不同的判断语句流入相同的分支。

例 3-7 GradeLevel.java 根据考试成绩的等级打印出百分制分数段。

```
public class GradeLevel{  
    public static void main( String args[ ] ){  
        System.out.println("\n**** first situation ****");  
        char grade='C'; //normal use  
        switch( grade ){  
            case 'A' :  
                System.out.println(grade+" is 85~100");  
                break;  
            case 'B' :  
                System.out.println(grade+" is 70~84");  
                break;  
            case 'C' :  
                System.out.println(grade+" is 60~69");  
                break;  
            case 'D' :  
                System.out.println(grade+" is <60");  
                break;  
        }  
    }  
}
```

```
        default :  
            System.out.println("input error");  
        }  
    }  
}
```

3. 应用举例

例 3-8 AutoScore.java 自动出题并判分。

该程序随机产生两个数和一个运算符（即所谓“出题”），并让用户输入一个数作为答案，然后程序判断结果是否正确（即所谓“判分”），并在一个 PictureBox 中显示判断结果。如图 3-4 所示。



图 3-4 自动出题并判分

程序如下：

```
import java.awt.*;  
import java.applet.*;  
  
public class AutoScore extends Applet  
{  
    public void init()  
    {  
  
        //{{INIT_CONTROLS  
        setLayout(null);  
        setSize(380,266);  
        btnNew.setLabel("出题");
```

```
add(btnNew);
btnNew.setBackground(java.awt.Color.lightGray);
btnNew.setBounds(36,96,98,26);
button2.setLabel("判分");
add(button2);
button2.setBackground(java.awt.Color.lightGray);
button2.setBounds(216,96,94,25);
lblA.setText("text");
add(lblA);
lblA.setFont(new Font("Dialog", Font.PLAIN, 24));
lblA.setBounds(36,24,36,36);
lblOp.setText("text");
add(lblOp);
lblOp.setFont(new Font("Dialog", Font.PLAIN, 24));
lblOp.setBounds(72,24,36,36);
lblB.setText("text");
add(lblB);
lblB.setFont(new Font("Dialog", Font.PLAIN, 24));
lblB.setBounds(108,24,33,36);
label5.setText("=");
add(label5);
label5.setFont(new Font("Dialog", Font.PLAIN, 24));
label5.setBounds(168,24,34,36);
add(txtAnswer);
txtAnswer.setFont(new Font("Dialog", Font.PLAIN, 24));
txtAnswer.setBounds(216,24,85,42);
listDisp.setFont(new Font("Dialog", Font.PLAIN, 16));
add(listDisp);
listDisp.setBounds(36,144,272,106);
//}}

//{{REGISTER_LISTENERS
SymAction lSymAction = new SymAction();
btnNew.addActionListener(lSymAction);
button2.addActionListener(lSymAction);
//}}
}

//{{DECLARE_CONTROLS
```

```
java.awt.Button btnNew = new java.awt.Button();
java.awt.Button button2 = new java.awt.Button();
java.awt.Label lblA = new java.awt.Label();
java.awt.Label lblOp = new java.awt.Label();
java.awt.Label lblB = new java.awt.Label();
java.awt.Label label5 = new java.awt.Label();
java.awt.TextField txtAnswer = new java.awt.TextField();
java.awt.List listDisp = new java.awt.List(0);
//}}

class SymAction implements java.awt.event.ActionListener
{
    public void actionPerformed(java.awt.event.ActionEvent event)
    {
        Object object = event.getSource();
        if (object == btnNew)
            btnNew_ActionPerformed(event);
        else if (object == button2)
            button2_ActionPerformed(event);
    }
}

void btnNew_ActionPerformed(java.awt.event.ActionEvent event)
{
    // to do: code goes here.
    a = (int)(Math.random()*9+1);
    b = (int)(Math.random()*9+1);
    int c = (int)(Math.random()*4);
    switch( c )
    {
        case 0: op="+"; result=a+b; break;
        case 1: op="-"; result=a-b; break;
        case 2: op="*"; result=a*b;break;
        case 3: op="/"; result=a/b;break;
    }
    lblA.setText(""+a);
    lblB.setText(""+b);
    lblOp.setText(""+op);
    txtAnswer.setText("");
}
```

```
}

int a=0,b=0;
String op="";
double result=0;

void button2_ActionPerformed(java.awt.event.ActionEvent event)
{
    // to do: code goes here.
    String str = txtAnswer.getText();
    double d = Double.valueOf(str).doubleValue();
    String disp = "" + a + op + b+"="+ str + " ";
    if( d == result ) disp += "☆";
    else disp += "✕";

    listDisp.add( disp );
}

public static void main(String [] args)
{
    Frame f = new Frame();
    f.setSize( 400,300 );
    AutoScore p = new AutoScore();
    f.add ( p );
    p.init();
    p.start();
    f.setVisible( true );
}

}
```

在该程序的界面设计中,用到了如表 3-10 所示的界面对象。上面的程序代码可以用普通的文本编辑器录入,但最好使用可视化界面设计工具,如 Visual Café 或 JBuilder 等,其设计时的界面如图 3-4 所示。

表 3-10 界面对象及其属性

界面对象	名称	属性名	属性值
数 a	lblA		
运算符 op	lblOp		
数 b	lblB		
输入的数	txtAnswer	text	

续表

界面对象	名称	属性名	属性值
判分按钮	Button2	label	判分
信息显示列表框	listDisp		

注意：应将 Applet 的 Layout 属性设为 null。

程序中，产生随机数的方法是用 `Math.random()` 函数，而随机产生符号的方法是先产生一个随机数，然后根据这种数的大小得到一个符号（用 `switch` 语句实现）。

在计算正确的答案时，也用了 `switch` 语句。在判断答案是否正确时，用到了 `if` 语句。

该程序可以作为 Application，也可以作为 Applet。嵌入 Applet 的网页 `AutoScore.html` 为：

```
<HTML>
<HEAD>
<TITLE>AutoSoce 自动出题并判分</TITLE>
</HEAD>
<BODY>
<APPLET CODE="AutoScore.class" WIDTH=380 HEIGHT=266></APPLET>
</BODY>
</HTML>
```

3.3.4 循环语句

循环结构是在一定条件下反复执行某段程序的流程结构，被反复执行的程序被称为循环体。循环结构是程序中非常重要和基本的一种结构，它是由循环语句来实现的。Java 的循环语句共有三种：`for`、`while` 和 `do-while` 语句，如图 3-5 所示。

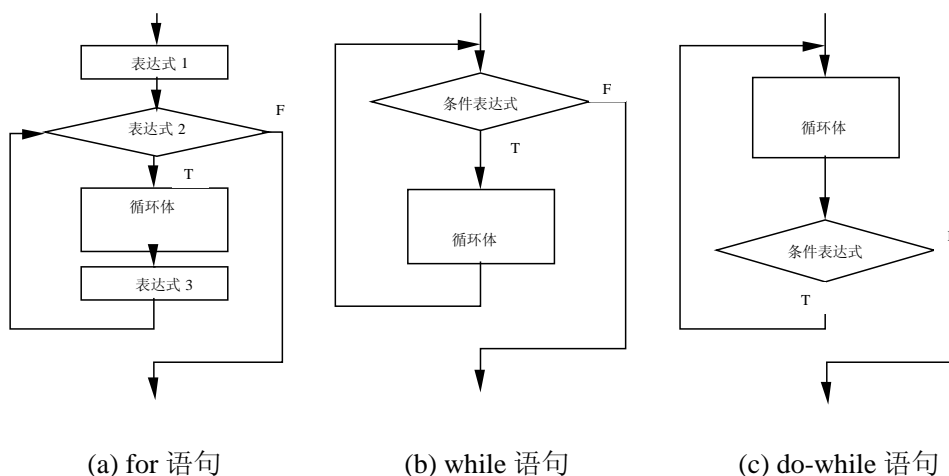


图 3-5 循环语句

三种语句在使用时，都要表达以下几个要素。

(1) 循环的初始化。

- (2) 循环的条件。
- (3) 循环体。
- (4) 循环的改变。

下面的例题是用三种方式来表达的 $1+2+3+\cdots+100$ 的循环相加的过程。

例 3-8 Sum100.java 循环语句用于求 $1+2+3+\cdots+100$ 。

```
public class Sum100{
    public static void main( String args[ ] ){
        int sum,n;

        System.out.println("\n**** for statement ****");
        sum=0;
        for( int i=1; i<=100; i++) {    // 初始化,循环条件,循环改变
            sum+=i;                      // 循环体
        }
        System.out.println("sum is "+sum);

        System.out.println("\n**** while statement ****");
        sum=0;
        n=100;                          // 初始化
        while( n>0 ){                   // 循环条件
            sum+=n;                      // 循环体
            n--;                         // 循环改变
        }
        System.out.println("sum is "+sum);

        System.out.println("\n**** do_while statement ****");
        sum=0;
        n=0;                            // 初始化
        do{
            sum+=n;                      // 循环体
            n++;                         // 循环改变
        }while( n<=100 );               // 循环条件
        System.out.println("sum is "+sum);
    }
}
```

运行结果如图 3-6 所示。

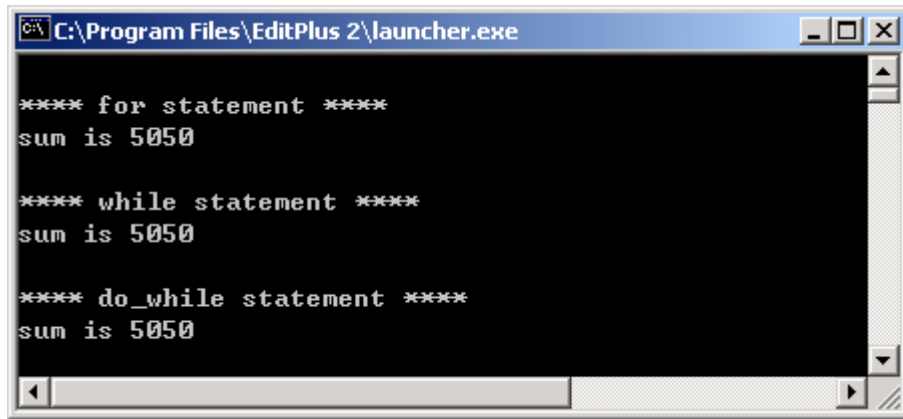


图 3-6 运行结果

可以从中比较这三种循环语句，从而在不同的场合选择合适的语句。

下面比较详细地讲解这三种循环语句的用法。

1. for 语句

for 语句是 Java 语言三个循环语句中功能较强、使用较广泛的一个，它的流程结构可参看

图3-5(a)。for 语句的一般语法格式如下：

```
for(表达式 1; 表达式 2; 表达式 3)
    循环体
```

其中，表达式 1 完成初始化循环变量和其他变量的工作；表达式 2 是返回布尔值的条件表达式，用来判断循环是否继续；表达式 3 用来修改循环变量，改变循环条件。3 个表达式之间用分号隔开。

for 语句的执行过程是首先计算表达式 1，完成必要的初始化工作；再判断表达式 2 的值，若为真，则执行循环体，执行完循环体后再返回表达式 3，计算并修改循环条件。这样一轮循环就结束了。第二轮循环从计算并判断表达式 2 开始，若表达式的值仍为真，则继续循环；否则，跳出整个 for 语句执行下面的句子。for 语句的三个表达式都可以为空，但若表达式 2 也为空，则表示当前循环是一个无限循环，需要在循环体中书写另外的跳转语句终止循环。

注意：for 循环的第 1 个表达式中，可以定义变量，这里定义的变量只在该循环体内有效。如 `for(int n=0; n<100; n++) { System.out.println(n); }`

例 3-9 Circle99.java。在 Applet 中画很多同心圆，如图3-7 所示。

```
import java.awt.*;
import java.applet.*;

public class Circle99 extends Applet
{
```

```
public void paint(Graphics g)
{
    g.drawString("circle 99", 20, 20);

    int x0 = getSize().width /2;
    int y0 = getSize().height /2;

    for( int r=0 ; r<getSize().height/2; r+=3 )
    {
        g.setColor( getRandomColor() );
        g.drawOval( x0-r,y0-r, r*2, r*2 );
    }

}

Color getRandomColor()
{
    return new Color(
        (int)( Math.random() * 255 ),
        (int)( Math.random() * 255 ),
        (int)( Math.random() * 255 )
    );
}
```

其相关的网页文件 Circle99.html 为:

```
<HTML>
<HEAD>
    <TITLE>Circle99 Example1</TITLE>
</HEAD>
<BODY>
    <H1>Circle99</H1>
    <HR>
    <P>
        <APPLET CODE="Circle99.class" WIDTH="300"
            HEIGHT="300">
        </APPLET>
    </P>
    <HR>
</BODY>
```

</HTML>

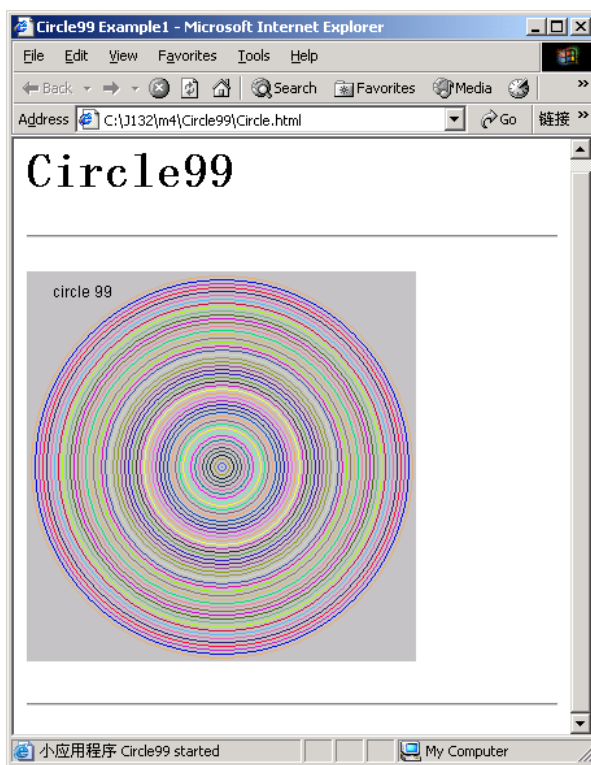


图 3-7 在 Applet 中画很多同心圆

2 . while 语句

while 语句的一般语法格式如下：

```
while( 条件表达式)  
    循环体
```

其中，条件表达式的返回值为布尔型，循环体可以是单个语句，也可以是复合语句块。

while 语句的执行过程是先判断条件表达式的值，若为真，则执行循环体，循环体执行完之后再无条件转向条件表达式做计算与判断；当计算出条件表达式为假时，跳过循环体执行 while 语句后面的语句。

例 3-10 验证“角谷猜想”。

“角谷猜想”指出：将一个自然数按以下的一个简单规则进行运算：若数为偶数，则除以 2；若为奇数，则乘 3 并加 1。将得到的数按该规则重复运算，最终可得到 1。现给定一个数 n ，试用程序来验证该过程，如图 3-8 所示。

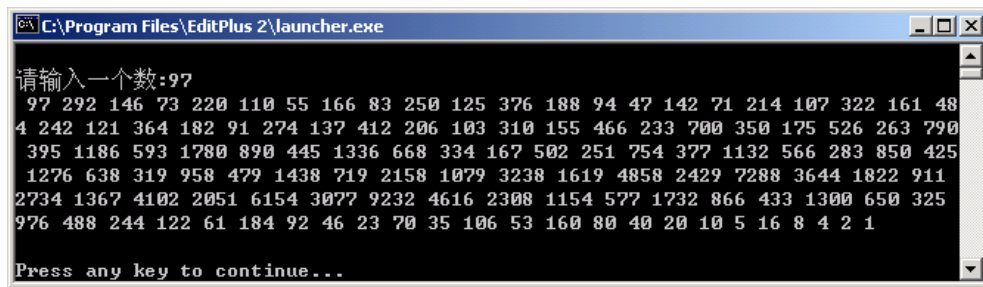


图 3-8 验证“角谷猜想”

```
import java.io.*;
class Jiaogu
{
    public static void main(String[] args)
    {
        System.out.print("\n 请输入一个数:");
        try{
            BufferedReader br =
new BufferedReader ( new InputStreamReader( System.in ) );
            String s = br.readLine();
            int a = Integer.parseInt( s );

            while( a != 1 )
            {
                System.out.print( " " +a );
                if( a%2==1 ) a = a*3+1; else a /= 2;
            }
            System.out.println(" " +a);
        }
        catch(Exception e){}
    }
}
```

3 . do-while 语句

do-while 语句的一般语法结构如下:

```
do
    循环体
while (条件表达式);
```

do-while 语句的使用与 while 语句很类似,不同的是它不像 while 语句是先计算条件表达式的值,而是无条件地先执行一遍循环体,再来判断条件表达式。若表达式的值为真,则运行循环体,否则跳出 do-while 循环,执行下面的语句。可以看出,do-while 语句的特

点是它的循环体将至少被执行一次。

例 3-11 ShowManyCharValue.java 多次输入字符，显示其 Ascii 码，直到按#结束，如图 3-9 所示。

```
import java.io.*;

class ShowManyCharValue
{
    public static void main(String[] args)
    {
        try{

            char c;
            do
            {
                System.out.println("输入字符并按回车，按#结束");
                c = (char)System.in.read();
                System.in.skip(2); //忽略回车换行
                System.out.println( c + "的 Ascii 值为:" + (int)c );
            }
            while ( c != '#' );
        }catch(Exception e){}
    }
}
```

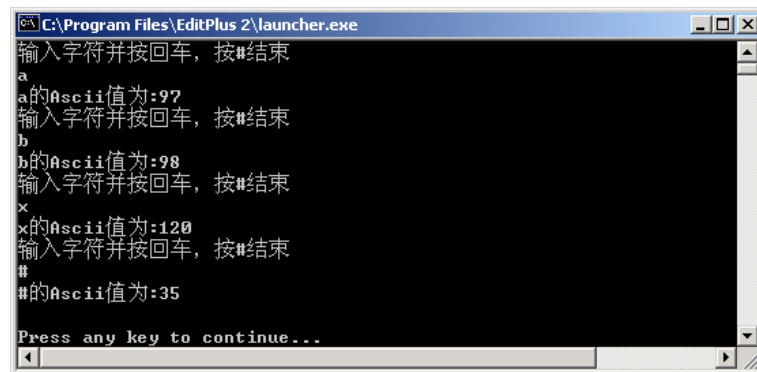


图 3-9 多次输入字符

3.3.5 跳转语句

跳转语句用来实现程序执行过程中流程的转移。前面，在 switch 语句中使用过的 break 语句就是一种跳转语句。为了提高程序的可靠性和可读性，Java 语言不支持无条件跳转的 goto 语句。Java 的跳转语句有三个：continue、break 和 return 语句。

1 . continue 语句

continue 语句必须用于循环结构中，它有两种使用形式。

一种是不带标号的 continue 语句，它的作用是终止当前这一轮的循环，跳过本轮剩余的语句，直接进入当前循环的下一轮。在 while 或 do-while 循环中，不带标号的 continue 语句会使流程直接跳转至条件表达式。在 for 循环中，不带标号的 continue 语句会跳转至表达式 3，计算修改循环变量后再判断循环条件。

另一种是带标号的 continue 语句，其格式是：

continue 标号名；

这个标号名应该定义在程序中外层循环语句的前面，用来标志这个循环结构。标号的命名应该符合 Java 标识符的规定。带标号的 continue 语句使程序的流程直接转入标号标明的循环层次。

2 . break 语句

break 语句的作用是使程序的流程从一个语句块内部跳转出来，如从 switch 语句的分支中跳出，或从循环体内部跳出。break 语句同样分为带标号和不带标号两种形式。带标号的 break 语句的使用格式是：

break 标号名；

这个标号应该标志某一个语句块。执行 break 语句就从这个语句块中跳出来，流程进入该语句块后面的语句。

不带标号的 break 语句从它所在的 switch 分支或最内层的循环体中跳转出来，执行分支或循环体后面的语句。

例 3-12 MaxDiv.java。求一个数的最大真约数。程序中从大向小进行循环，直到能整除，则用 break 退出循环，如图 3-10 所示。

```
public class MaxDiv
{
    public static void main(String[] args)
    {
        int a = 99;
        int i = a - 1;
        while(i>0){
            if( a % i == 0 ) break;
            i--;
        }
        System.out.println(a + "的最大真约数为: " + i);
    }
}
```



图 3-10 求最大真约数

3. return 语句

return 语句的一般格式是：

return 表达式；

return 语句用来使程序流程从方法调用中返回，表达式的值就是调用方法的返回值。

如果方法没有返回值，则 return 语句中的表达式可以省略。

例 3-13 Prime100Continue.java 求 100 ~ 200 间的所有素数，如图 3-11。

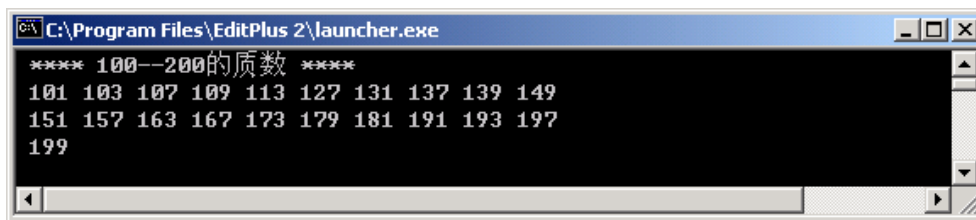


图 3-11 求 100 ~ 200 间的所有素数

```
public class Prime100Continue{
    public static void main( String args[ ] ){
        System.out.println(" **** 100--200 的质数 ****");
        int n=0;
        outer: for(int i=101;i<200;i+=2){ // 外层循环
            for(int j=2; j<i; j++){ // 内层循环
                if( i%j==0 ) // 不是质数，则继续外层循环
                    continue outer;
            }
            System.out.print(" "+i); // 显示质数
            n++; // 计算个数
            if( n<10 ) // 未满 10 个数，则不换行
                continue;
            System.out.println();
            n=0;
        }
        System.out.println();
    }
}
```

```
}
```

该例通过一个嵌套的 for 语句来实现。其中，外层循环遍历 101~200，内层循环针对一个数 i，用 2 到 i-1 之间的数去除，若能除尽，则表明不是质数，直接继续外层的下一次循环(continue outer)。

3.4 数组

数组是有序数据的集合，数组中的每个元素具有相同的数据类型，可以用一个统一的数组名和下标来惟一地确定数组中的元素。数组有一维数组和多维数组，下面分别进行介绍。

3.4.1 一维数组

1. 一维数组的定义

一维数组的定义方式为：

```
type arrayName[ ];  
或 type [ ] arrayName;
```

其中，类型(type)可以为 Java 中任意的数据类型，包括简单类型和组合类型，数组名 arrayName 为一个合法的标识符，[]指明该变量是一个数组类型变量，它可以放到数组名的前面或后面，建议读者将[]放在标识符的前面。例如：

```
int [ ] score;
```

声明了一个整型数组，数组中的每个元素为整型数据。与 C、C++不同，Java 在数组的定义中并不为数组元素分配内存，因此[]中不用指出数组中元素的个数（即数组长度），但必须为它分配内存空间，这时，要用到运算符 new，其格式如下：

```
arrayName = new type[arraySize];
```

其中，arraySize 指明数组的长度。如：

```
score = new int[100];
```

为一个整型数组分配 100 个 int 型整数所占据的内存空间。

通常，这两部分可以合在一起，格式如下：

```
type arrayName = new type [arraySize];
```

例如：

```
int [ ] score = new int[3];
```

注意：数组用 new 分配空间的同时，数组的每个元素都会自动赋一个默认值（整数为 0，实数为 0.0，字符为 '\0'，boolean 型为 false，引用型为 null）。这是因为，数组实际是一种引用型的变量，而其每个元素是引用型变量的成员变量。

2. 一维数组元素的引用

当定义了一个数组，并用运算符 `new` 为它分配了内存空间后，就可以引用数组中的每一个元素了。数组元素的引用方式为：

`arrayName[index]`

其中，`index` 为数组下标，它可以为整型常数或表达式。如 `a[3]`，`b[i]` (*i* 为整型)，`c[6*i]` 等。下标从 0 开始，一直到数组的长度减到 1。对于上面例子中的 `score` 数组来说，它有 100 个元素，分别为：

`score[0]`，`score[1]`，...，`score[99]`。

另外，与 C、C++ 中不同，Java 对数组元素要进行越界检查以保证安全性。同时，对于每个数组都有一个属性 `length` 指明它的长度，例如，`score.length` 指明数组 `score` 的长度。

例 3-14 ArrayTest.java 使用数组。

```
public class ArrayTest {  
    public static void main( String args[ ] ){  
        int i;  
        int a[ ]=new int[5];  
        for( i=0; i<5; i++ )  
            a[i]=i;  
        for( i=a.length-1; i>=0; i-- )  
            System.out.println("a["+i+"] = "+a[i]);  
    }  
}
```

运行结果如图 3-12 所示。

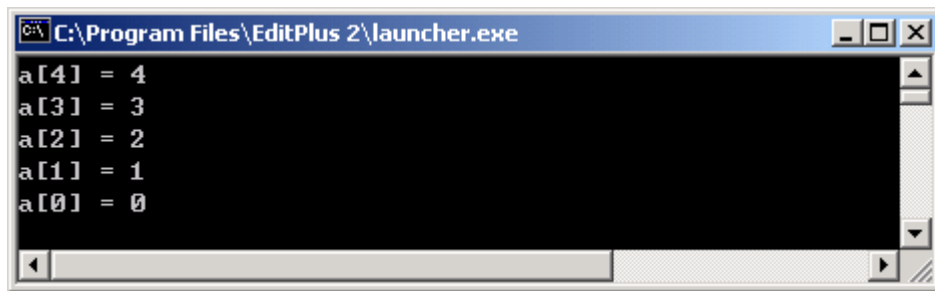


图 3-12 使用数组

该程序对数组中的每个元素赋值，然后按逆序输出。

3. 一维数组的初始化

对数组元素可以按照上述的例子进行赋值。也可以在定义数组的同时进行初始化。例如：

```
int a[ ] = {1,2,3,4,5};
```

用逗号(,)分隔数组的各个元素，系统自动为数组分配一定的空间。

与 C 中不同，这时 Java 不要求数组为静态(static)。

4. 一维数组程序举例

例 3-15 Fibonacci.java Fibonacci 数列。

Fibonacci 数列的定义为：

$$F_1 = F_2 = 1, F_n = F_{n-1} + F_{n-2} (n \geq 3)$$

程序如下：

```
public class Fibonacci{
    public static void main( String args[ ] ){
        int i;
        int f[ ]=new int[10];
        f[0]=f[1]=1;
        for( i=2; i<10; i++ )
            f[i]=f[i-1]+f[i-2];
        for( i=1; i<=10; i++ )
            System.out.println("F["+i+"]= "+f[i-1]);
    }
}
```

运行结果如图 3-13 所示。

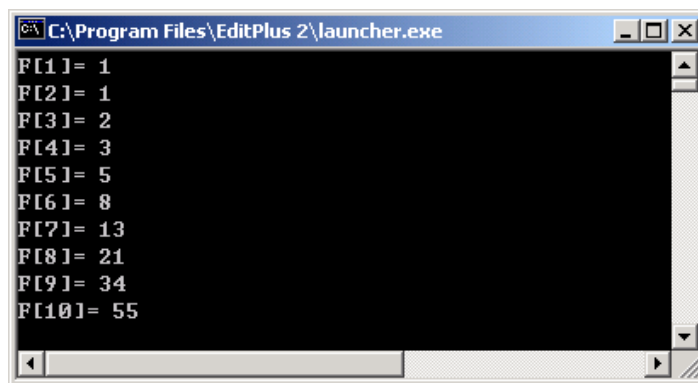


图 3-13 Fibonacci 数列

例 3-16 Rnd_36_7.java 36 选 7。随机产生 7 个数，每个数在 1~36 范围内，要求每个数不同，如图 3-14 所示。

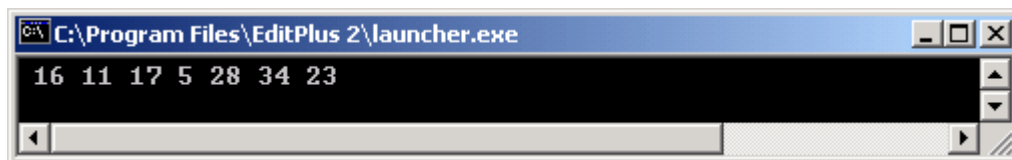


图 3-14 36 选 7

```
class Rnd_36_7
{
    public static void main(String[] args)
    {
        int a[] = new int[7];
        for( int i=0;i<a.length;i++)
        {
            one_num:
            while(true)
            {
                a[i] = (int)( Math.random()*36 ) +1;

                for( int j=0;j<i;j++ ){
                    if( a[i]==a[j] ) continue one_num;
                }
                break;
            }
        }
        for( int i=0;i<a.length;i++) System.out.print( " " + a[i] );
        System.out.println();
    }
}
```

3.4.2 多维数组

Java 中多维数组被看做数组的数组。例如，二维数组为一个特殊的一维数组，其每个元素又是一个一维数组。下面以二维数组为例来进行说明，高维数组的情况是类似的。

1. 二维数组的定义

二维数组的定义方式为：

```
type arrayName[ ][ ];
```

例如：

```
int x[ ][ ];
```

与一维数组一样，这时对数组元素也没有分配内存空间，同样要使用运算符 **new** 来分配内存，然后才可以访问每个元素。

对高维数组来说，分配内存空间有下面几种方法：

(1) 直接为每一维分配空间，如：

```
int a[ ][ ] = new int[2][3];
```

(2) 从最高维开始，分别为每一维分配空间，如：

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

```
a[0] = new int[3];
```

```
a[1] = new int[3];
```

完成方法(1)中相同的功能。

2. 二维数组元素的引用

对二维数组中每个元素，引用方式为：`arrayName[index1][index2]`

其中，`index1`、`index2` 为下标，可为整型常数或表达式，如 `a[2][3]`等。同样，每一维的下标都从 0 开始。

3. 二维数组的初始化

二维数组的初始化有以下两种方式：

(1) 直接对每个元素进行赋值；

(2) 在定义数组的同时进行初始化。如：

```
int a[][] = {{2,3}, {1,5}, {3,4}};
```

定义了一个 3×2 的数组，并对每个元素赋值。

4. 非规则矩阵数组

Java 中的多维数组实际上既是数组元素又是数组，所以每个数组的元素个数可以不一样，这样就形成了非规则矩阵的数组，又称为“锯齿状数组”。如：

```
int [] [] a = new int[4];
```

```
a[0] = new int[2];
```

```
a[1] = new int[3];
```

```
a[2] = new int[1];
```

```
a[3] = new int[9];
```

5. 二维数组举例

例 3-17 MatrixMultiply.java 两个矩阵相乘。矩阵 $A_{m \times n}$ ， $B_{n \times l}$ 相乘得到 $C_{m \times l}$ ，每个元素 $C_{ij} = a_{ik} \times b_{kj} (i=1, \dots, m, n=1, \dots, n)$ 。

```
public class MatrixMultiply{
    public static void main( String args[ ] ){
        int i,j,k;
        int a[ ][ ]={ {2,3,5}, {1,3,7} };
        int b[ ][ ]={ {1,5,2,8},{5,9,10,-3},{2,7,-5,-18} };
        int c[ ][ ]=new int[2][4];
        for( i=0; i<2; i++ ){
            for( j=0; j<4; j++ ){
                c[i][j]=0;
                for( k=0; k<3; k++ ){
```

```
        c[i][j]+=a[i][k]*b[k][j];
    }
}
System.out.println("\n*** Matrix A ***");
for( i=0; i<2; i++ ){
    for( j=0; j<3; j++ )
        System.out.print(a[i][j]+" ");
    System.out.println();
}
System.out.println("\n*** Matrix B ***");
for( i=0; i<3; i++ ){
    for( j=0; j<4; j++ )
        System.out.print(b[i][j]+" ");
    System.out.println();
}
System.out.println("\n*** Matrix C ***");
for( i=0; i<2; i++ ){
    for( j=0; j<4; j++ )
        System.out.print(c[i][j]+" ");
    System.out.println();
}
}
```

运行结果如图 3-15 所示。

```
C:\Program Files\EditPlus 2\launcher.exe

*** Matrix A ***
2 3 5
1 3 7

*** Matrix B ***
1 5 2 8
5 9 10 -3
2 7 -5 -18

*** Matrix C ***
27 72 9 -83
30 81 -3 -127
```

图 3-15 两个矩阵相乘

3.4.3 System.arraycopy()方法

System.arraycopy()方法可以用来复制数组，其格式是：

System.arraycopy(Object src, int src_position, Object dst, int dst_position, int length)

它将数组从 src 复制到 dst，复制的位置是 src 的第 src_position 个元素到 dst 的第 dst_position 位置，复制元素的个数为 length。

注意：该方法只复制元素。如果数组元素是引用型变量，则只复制引用，不复制对象实体。

习题

1. 简述 Java 程序的构成。如何判断主类？下面的程序有几处错误？如何改正，这个程序的源代码应该保存为什么名字的文件？

```
public class MyJavaClass
{
    public static void main()
    {
        System.out. println("Am I wrong?");
    }
    System.out. println("程序结束。");
}
```

2. Java 有哪些基本数据类型？写出 int 型所能表达的最大和最小数据。

3. Java 的字符采用何种编码方案？有何特点？写出五个常见的转义符。

4. Java 对标识符命名有什么规定，下面的标识符哪些是对的？哪些是错的？错在哪里？

(1) MyGame (2) _isHers (3) 2JavaProgram (4) Java-Visual-Machine (5) _\$abc

5. 什么是常量？什么是变量？字符变量与字符串常量有何不同？

6. 什么是强制类型转换？在什么情况下需要用到强制类型转换？

7. Java 有哪些算术运算符、关系运算符、逻辑运算符、位运算符和赋值运算符？试列举单目和三目运算符。

8. 编写一个字符界面的 Java Application 程序，接受用户输入的一个浮点数，把它的整数部分和小数部分分别输出。

9. 编写一个字符界面的 Java Application 程序，接受用户输入的 10 个整数，比较并输出其中的最大值和最小值。

10. 编写一个字符界面的 Java Application 程序，接受用户输入的字符，以“#”标志输入的结束；比较并输出按字典序最小的字符。

11. 结构化程序设计有哪三种基本流程？分别对应 Java 中的哪些语句？

12. 编写一个 Java 程序，接受用户输入的一个 1~12 之间的整数(如果输入的数据不满足这个条件，则要求用户重新输入)，利用 switch 语句输出对应月份的天数。

13. 在一个循环中使用 `break`, `continue` 和 `return` 语句有什么不同的效果?
14. 编写图形界面下的 Java Applet 程序, 接受用户输入的两个数据为上、下限, 然后按 10 个一行输出上、下限之间的所有素数。
15. 编写程序输出用户指定数据的所有素数因子。
16. 什么是数组? 数组有哪些特点? 在 Java 中创建数组需要使用哪些步骤? 如何访问数组的一个元素? 数组元素的下标与数组的长度有什么关系?
17. 数组元素会怎样进行默认的初始化?
18. 编程求一个整数数组的最大值、最小值、平均值和所有数组元素的和。
19. 求解“约瑟夫问题”: 12 个人排成一圈, 从 1 号报数, 凡是数到 5 的人就走出队列 (出局), 然后继续报数, 试问最后一人出局的是谁。
20. 用“埃氏筛法”求 2~100 以内的素数。2~100 以内的数, 先去掉 2 的倍数, 再去掉 3 的倍数, 再去掉 4 的倍数, ……依此类推, 最后剩下的就是素数。

第4章 类、包和接口

前面章节中，对 Java 的简单数据类型、数组、运算符和表达式及流控制方法作了详细的介绍。从本章开始，进入面向对象的编程技术，将接触到 Java 最引人入胜之处。本章介绍 Java 中面向对象的程序设计的基本方法，包括类的定义、类的继承、包、访问控制、修饰符、接口等方面的内容。

4.1 类、域、方法

4.1.1 定义类中的域和方法

Java 程序由很多类构成，类就是域和相关的方法的集合，其中域表明对象的状态，方法表明对象所具有的行为。

程序中，类的定义包括类头和类体两个步骤，其中类体用一对大括号{}括起，类体又由域(field)和方法(method)组成。

例 4-1 表示“人”的类的定义 Person。

```
class Person {  
    String name;  
    int age;  
    void sayHello(){  
        System.out.println("Hello! My name is" + name );  
    }  
}
```

类头使用关键字 class 标志类定义的开始，class 关键字后面接着用户定义的类的类名。类名的命名应符合 Java 对标识符命名的要求。

类体中包括域和方法两大部分。域和方法都是类的成员。一个类中可以定义多个域和方法。一个类可以通过 UML 图中的类图表示出来，如图 4-1 所示。类图中的类用一个矩形来表示，上部是类名，中间是域的表示，底部是方法的表示。

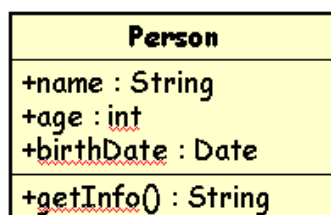


图 4-1 在 UML 图中表示的类

1. 域

域对应类的静态属性。在有些场合，域又称为域变量、属性、成员变量等。例 4-1 中有两个域，`name`（表示姓名）和 `age`（表示年龄），其类型分别是 `String` 和 `int`。域也是变量，定义方法与第 3 章中变量的定义方法相同，即：

类型名 域名；

如：

```
int age;
```

在定义域名时，还可以赋初始值。如：

```
int age=0;
```

如果不赋初始值，系统会自动赋一个默认值（数值型为 0，boolean 型为 `false`，对象型为 `null`，注意，`String` 类型是系统定义好的对象型。）

此外，定义域变量前，还可以加修饰符。有关的修饰符将在本章的后续部分中讲述。

2. 方法

方法是类的动态属性，标志了类所具有的功能和操作，用来把类和对象的数据封装在一起。Java 的方法与其他语言中的函数或过程类似，是一段用来完成某种操作的程序片断。方法由方法头和方法体组成，其一般格式如下：

```
修饰符 1 修饰符 2 ..... 返回值类型 方法名(形式参数列表) throws  
异常列表{方法体各语句;  
}
```

其中，形式参数列表的格式为：

```
形式参数类型 1 形式参数名 1, 形式参数类型 2 形式参数名 2, .....
```

小括号()是方法的标志，不能省略；方法名是标识符，要求满足标识符的规则；形式参数是方法从调用它的环境输入的数据；返回值是方法在操作完成后返还给调用它的环境的数据，返回值都有类型，若没有返回值，则使用 `void` 表示。

如，在例 4-1 中，有一个方法 `sayHello`，其定义如下：

```
void sayHello(){  
    System.out.println("Hello! My name is" + name );  
}
```

如果方法的返回类型为 `void`(没有返回值)，参数为空，方法体中有一条语句。

如果方法有返回值，则在方法体中，必须有 `return` 语句，`return` 语句后跟上返回值。如：

```
boolean isOlderThan( int anAge ){  
    boolean flg;  
    if( age > anAge ) flg = true; else flg=false;  
    return flg;  
}
```

这里的方法 `isOlderThan` 用于判断年龄是否比某个值 (`anAge`) 大。`AnAge` 是参数，返回值是 `boolean` 型。

方法定义时，修饰符和 `throws` 异常列表都可以没有，它们的详情将在后面章节中讲述。

4.1.2 构造方法与对象的创建

1. 构造方法

程序中经常需要创建对象，在创建对象的同时将调用这个对象的构造函数完成对象的初始化工作。

构造函数（**constructor**），也称构造方法，它是一种特殊的、与类同名的方法，专门用于创建对象，完成初始化工作。构造方法的特殊性主要体现在如下的几个方面。

- (1) 构造方法的方法名与类名相同。
- (2) 构造方法没有返回类型，也不能写 `void`。
- (3) 构造方法的主要作用是完成对类对象的初始化工作。
- (4) 构造方法一般不能由编程人员显式地直接调用，而是用 `new` 来调用。
- (5) 在创建一个类的新对象的同时，系统会自动调用该类的构造方法为新对象初始化。

我们知道，在声明域变量时可以为它赋初值，那么为什么还需要构造方法呢？这是因为，构造方法可以带上参数，而且构造方法还可以完成赋值之外的其他一些复杂操作。

如，可以给 `Person` 类加上一个构造方法如下：

```
Person( String n, int a ){
    name = n;
    age = a;
}
```

在该构造方法中，将给定的参数赋给域变量。

2. 默认构造方法

一般情况下，类都有一个至多个构造方法，如果在定义类对象时没有定义任何构造方法，系统会自动产生一个构造方法，称为默认构造方法（**default constructor**）。

默认构造方法不带参数，并且方法体为空。

例如，如果上面的 `Person` 类没有定义构造方法，则系统产生的默认构造方法如下：

```
Person(){}
```

如果 `class` 前面有 `public` 修饰符，则默认构造方法前面也会是 `public` 的。

3. 创建对象

Java 程序定义类的最终目的是使用它，像使用系统类一样，程序也可以继承用户自定义类或创建并使用自定义类的对象。下面讨论如何创建类的对象，即实例化对象。

创建对象前首先要声明变量，声明的方法与声明基本数据类型的变量类似，其格式为：

类名 变量名；

创建对象的一般格式为：

变量名=new 构造方法(参数)；

以上两句可以合写成一句为：

类名 变量名 = new 构造方法 (参数);

例如:

```
Person p = new Person("Liming", 20 );
```

其中, **new** 是为新建对象算符。它以类为模板, 开辟空间并执行相应的构造方法。**new** 实例化一个对象, 返回对该对象的一个引用(即该对象所在的内存地址)。

这里声明的变量, 称为对象变量, 它是引用型的变量。与基本型变量一样, 引用型变量要占据一定的内存空间, 同时, 它所引用的对象实体(也就是用 **new** 创建的对象实体)也要占据一定的空间。通常对象实体占用的内存空间要大得多, 对象是创建的具体实例。以 **Person** 类为例, 其中定义了 2 个域 (**name** 和 **age**) 和一些方法, 这些域和方法保存在一块内存中, 这块内存就是 **p** 所引用的对象所占用的内存。

变量 **p** 与它所引用的实体所占据的关系, 是一种引用关系, 可以用图 4-2 表示。实际上, **name** 又是一个引用型变量, 它所引用的实体(字符串)又会占据一定的空间。

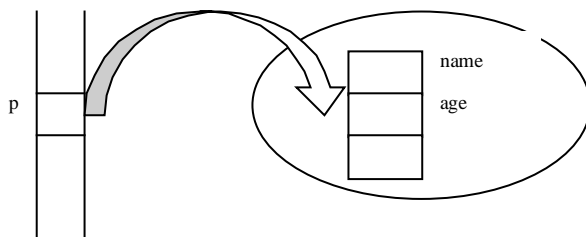


图 4-2 对象变量及其所引用的对象实体

多次使用 **new** 将多生成不同的对象, 这些对象分别对应于不同的内存空间, 它们的值是不同的, 可以完全独立地分别对它们进行操作。

4.1.3 使用对象

要访问或调用一个对象的域或方法, 需要用算符 “.” 连接 “对象” 这个对象和其域或方法。如:

```
System.out.println( p.name );  
p.sayHello();
```

由于只能通过这个对象变量来访问这个对象的域或方法, 不通过引用变量就无法访问其中的域或方法。对于访问者而言, 这个对象是封装成一个整体的, 这正体现了面向对象的程序设计的 “封装性”。

4.1.4 方法的重载

1. 方法的重载

在面向对象的程序设计语言中，有一些方法的含义相同，但带有不同的参数，这些方法使用相同的名字，这就叫方法的重载（overloading）。方法重载是实现“多态”的一种方法。

多个方法享有相同的名字，但是这些方法的参数列表必须不同，即：或者参数个数不同，或者是参数类型不同，或者参数类型的顺序不同。

注意：在这里，参数类型是关键，仅仅参数的变量名不同是不行的。方法重载时，返回值的类型可以相同，也可以不同。

例 4-2 通过方法重写分别接收一个或几个不同数据类型的数据。

```
void sayHello(){
    System.out.println("Hello! My name is " + name );
}

void sayHello( Person another ){
    System.out.println("Hello," + another.name + "! My name is " + name );
}
```

这里，两个函数都叫 sayHello，都表示问好。一个不带参数，表示向大家问好；一个带另一个 Person 对象作参数，表示向某个人问好。

在调用这两个方法时，可以不带参数，也可以带一个 Person 对象作参数。编译器会自动根据所带参数的类型来决定具体调用方法。

注意：在调用方法时，若没有找到类型相匹配的方法，编译器会找可以兼容的类型来进行调用。如，int 类型可以找到使用 double 类型参数的方法。若不能找到兼容的方法，则编译不能通过。

2. 构造方法的重载

构造方法也可以重载，要求使用不同的参数个数、不同的参数类型、不同的参数类型顺序。构造方法的重载，可以让用户用不同的参数来构造对象。

以下是 Person 的两种构造方法。

```
Person( String n, int a ){
    name = n;
    age = a;
}
```

```
Person( String n )
{
    name = n;
    age = -1;
}
```

前一个构造方法中，带有姓名及年龄信息；后一个构造方法，只有姓名信息，年龄信息未定，用一个特殊值（-1）。

4.1.5 this 的使用

在方法中，可以使用一个关键词 **this**，来表示这个对象本身。详细地说，在普通方法中，**this** 表示调用这个方法的对象；在构造方法中，**this** 表示新创建的对象。

1. 使用 **this** 来访问域及方法

在方法及构造方法中，可以使用 **this** 来访问对象的域和方法。

例如，方法 **sayHello** 中使用 **name** 和使用 **this.name** 是相同的。即：

```
void sayHello(){
    System.out.println("Hello! My name is " + name );
}
```

与

```
void sayHello(){
    System.out.println("Hello! My name is " + this.name );
}
```

的含义是相同的。

2. 使用 **this** 解决局部变量与域同名的问题

使用 **this** 还可以解决局部变量（方法中的变量）或参数变量与域变量同名的问题。如，在构造方法中，经常这样用：

```
Person( int age, String name )
{
    this.age = age;
    this.name = name;
}
```

这里，**this.age** 表示域变量，而 **age** 表示的是参数变量。

3. 构造方法中，用 **this** 调用另一构造方法

构造方法中，还可以用 **this** 来调用另一构造方法。如：

```
Person( )
{
    this( 0, "" );
}
```

```
}
```

如果，在构造方法中调用另一构造方法，则这条调用语句必须放在第一句。（关于构造方法的更复杂的问题，将在第5章中进一步讲述。）

4. 使用 this 的注意事项

在使用 this 时，要注意 this 指的是调用“对象”本身，不是指本“类定义”中看见的变量或方法。

-
- 注意：(1) 通过 this 不仅可以引用该类中定义的域和方法，还可以引用该类的父类中定义的域和方法；
- (2) 由于它指的是对象，所以 this 不能通过 this 来引用类变量 (static field) 类方法 (static method)。
-

事实上，在所有的非 static 方法中，都隐含了一个参数 this。而 static 方法中，不能使用 this。

4.2 类的继承

继承 (inheritance) 是面向对象的程序设计中最为重要的特征之一。由继承而得到的类为子类 (subclass)，被继承的类为父类或超类 (superclass)，父类包括所有直接或间接被继承的类。一个父类可以同时拥有多个子类。但由于 Java 中不支持多重继承，一个类只能有一个直接父类。父类实际上是所有子类的公共域和公共方法的集合，而每一个子类则是父类的特殊化，是对公共域和方法在功能、内涵方面的扩展和延伸。

子类继承父类的状态和行为，同时也可以修改父类的状态或重载父类的行为，并添加新的状态和行为。采用继承的机制来组织、设计系统中的类，可以提高程序的抽象程度，使之更接近于人类的思维方式，同时也通过继承能较好地实现代码重用，可以提高程序开发效率，降低维护的工作量。

Java 中，所有的类都是通过直接或间接地继承 java.lang.Object 得到的。

4.2.1 派生子类

Java 中的继承是通过 extends 关键字来实现的，在定义类时使用 extends 关键字指明新定义类的父类，就在两个类之间建立了继承关系。新定义的类称为子类，它可以从父类那里继承所有非 private 的属性和方法作为自己的成员。

通过在类的声明中加入 extends 子句来创建一个类的子类，其格式如下：

```
class SubClass extends SuperClass {  
    .....  
}
```

把 SubClass 声明为 SuperClass 的直接子类，如果 SuperClass 又是某个类的子类，则 SubClass 同时也是该类的(间接)子类。

如果没有 extends 子句，则该类默认为 java.lang.Object 的子类。因此，在 Java 中，所

有的类都是通过直接或间接地继承 `java.lang.Object` 得到的。

继承关系在 UML 图中，是用一个箭头来表示子类与父类的关系的，如图 4-3 所示。

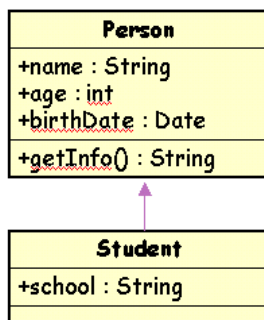


图 4-3 UML 图中继承关系的表示

类 **Student** 从类 **Person** 继承，定义如下：

```
class Student extends Person {
    //...
}
```

子类可以继承父类的所有内容。子类可以继承父类中域和方法，也可以添加域和方法，还可以隐藏或修改父类的域和方法，在继承时，受访问权限的限定，关于访问权限在 4.4 节中讲述。

4.2.2 域的继承与隐藏、添加

1. 域的继承

子类可以继承父类的所有域（只要该域没有 `private` 或 `static` 修饰）。

可见，父类的所有非私有域实际是各子类都拥有的域的集合。子类自动从父类继承域而不是把父类域的定义部分重复定义一遍，这样做的好处是减少程序维护的工作量。如 **Student** 自动具有 **Person** 的属性（`name`，`age`）。

2. 域的隐藏

子类重新定义一个与从父类那里继承来的域变量完全相同的变量，称为域的隐藏。域的隐藏在实际编程中用得较少。

3. 域的添加

在定义子类时，加上新的域变量，就可以使子类比父类多一些属性。如：

```
class Student extends Person
{
    String school;
    int score;
```

```
}
```

这里 Student 比 Person 多了两个属性：学校（school）和分数（score）。

4.2.3 方法的继承、覆盖与添加

1. 方法的继承

父类的非私有方法也可以被子类自动继承。如，Student 自动继承 Person 的方法 sayHello 和 isOlderThan。

2. 方法的覆盖

正像子类可以定义与父类同名的域，实现对父类域变量的隐藏一样，子类也可以重新定义与父类同名的方法，实现对父类方法的覆盖(Overlay)。

注意：子类在重新定义父类已有的方法时，应保持与父类完全相同的方法头声明，即应与父类有完全相同的方法名、返回值和参数类型列表。否则就不是方法的覆盖，而是子类定义自己的与父类无关的方法，父类的方法未被覆盖，所以仍然存在。

如，子类 Student 可以覆盖父类 Person 的 sayHello 方法：

```
void sayHello(){
    System.out.println("Hello! My name is " + name +
        ". My school is " + school );
}
```

可见，通过方法的覆盖，能够修改对象的同名方法的具体实现方法。

3. 方法的重载

一个类中可以有几个同名的方法，这称为方法的重载（Overloading）。同时，还可以重载父类的同名方法。与方法覆盖不同的是，重载不要求参数类型列表相同。重载的方法实际是新加的方法。

如，在类 Student 中，重载一个名为 sayHello 的方法：

```
void sayHello( Student another ){
    System.out.println("Hi!");
    if( school == another.school ) System.out.println(" Shoolmates ");
}
```

4. 方法的添加

子类可以新加一些方法，以针对子类实现相应的功能。

如，在类 Student 中，加入一个方法，对分数进行判断：

```
boolean isGoodStudent(){
    return score>=90;
```



```
}
```

4.2.4 super 的使用

Java 中除了使用 `this` 外，还有一个关键字 `super`。简单地说，`super` 是指父类，`super` 在类的继承中有重要作用。

1. 使用 `super` 访问父类的域和方法

子类自动地继承父类的属性和方法，一般情况下，直接使用父类的属性和方法，也可能以使用 `this` 来指明本对象。

注意：正是由于继承，使用 `this` 可以访问父类的域和方法。但有时为了明确地指明父类的域和方法，就要用关键字 `super`。

例如：父类 `Student` 有一个域 `age`，在子类 `Student` 中用 `age`, `this.age`, `super.age` 来访问 `age` 是完全一样的：

```
void testThisSuper(){
    int a;
    a = age;
    a = this.age;
    a = super.age;
}
```

当然，使用 `super` 不能访问在子类中添加的域和方法。

有时需要使用 `super` 以区别同名的域与方法。如，使用 `super` 可以访问被子类所隐藏了的同名变量。又如，当覆盖父类的同名方法的同时，又要调用父类的方法，就必须使用 `super`。如：

```
void sayHello(){
    super.sayHello();
    System.out.println( "My school is " + school );
}
```

从这里可以看出，即使同名，也仍然可以使用父类的域和方法，这也使得在覆盖父类的方法的同时，又利用已定义好的父类的方法。

2. 使用父类的构造方法

在严格意义上，构造方法是不能继承的。比如，父类 `Person` 有一个构造方法 `Person(String, int)`，不能说子类 `Student` 也自动有一个构造方法 `Student(String, int)`。但是，这不并不意味着子类不能调用父类的构造方法。

子类在构造方法中，可以用 `super` 来调用父类的构造方法。

```
Student(String name, int age, String school ){
    super( name, age );
}
```

```
        this.school = school;
    }
}
```

使用时，`super()`必须放在第一句。有关构造方法的更详细的讨论，参见第5章。

3. 使用 `super` 的注意事项

在使用 `super` 时，要注意 `super` 与 `this` 一样指的是调用“对象”本身，不仅是指父类中看见的变量或方法。（当然，使用 `super`，不能访问在本类定义的域和方法。）

注意：(1) 通过 `super` 不仅可以访问直接父类中定义的域和方法，还可以访问间接父类中定义的域和方法。

(2) 由于它指的是对象，所以 `super` 不能在 `static` 环境中使用，包括类变量（`static field`）、类方法（`static method`）、`static` 语句块。

4.2.5 父类对象与子类对象的转换

类似于基本数据类型数据之间的强制类型转换，存在继承关系的父类对象和子类对象之间也可以在一定条件下相互转换。父类对象和子类对象的转化需要遵循如下原则。

(1) 子类对象可以被视为其父类的一个对象，如一个 `Student` 对象也是一个 `Person` 对象。

(2) 父类对象不能被当做其某一个子类的对象。

(3) 如果一个方法的形式参数定义的是父类对象，那么调用这个方法时，可以使用子类对象作为实际参数。

(4) 如果父类对象引用指向的实际是一个子类对象，那么这个父类对象的引用可以用强制类型转换成子类对象的引用。

例 4-3 `Student.java` 继承。

```
class Person {
    String name;
    int age;

    static int x;

    Person( String n, int a ){
        name = n;
        age = a;
    }

    Person( String n ){
        name = n;
        age = -1;
    }
}
```

```
Person( int age, String name )
{
    this.age = age;
    this.name = name;
}

Person( ){
    this( 0, "" );
}

void sayHello(){
    System.out.println("Hello! My name is " + name );
}

void sayHello( Person another ){
    System.out.println("Hello," + another.name + "!
    My name is " + name );
}

boolean isOlderThan( int anAge ){
    boolean flg;
    if( age > anAge ) flg = true; else flg=false;
    return flg;
}

public static void main(String[] args)
{
    System.out.println("Hello World!");
}

class Student extends Person
{
    String school;
    int score;
    // void sayHello(){
    // System.out.println("Hello! My name is " + name + ".
    // My school is " + school );
    //}
```

```
void sayHello( Student another ){
    System.out.println("Hi!");
    if( school == another.school ) System.out.println
        (" Shoolmates ");
}

boolean isGoodStudent(){
    return score>=90;
}

void testThisSuper(){
    int a;
    a = age;
    a = this.age;
    a = super.age;
}

void sayHello(){
    super.sayHello();
    System.out.println( "My school is " + school );
}

Student(String name, int age, String school ){
    super( name, age );
    this.school = school;
}

Student(){}

public static void main( String [] args )
{
    Person p = new Person( "Liming", 50 );
    Student s = new Student( "Wangqiang", 20, "PKU" );
    Person p2 = new Student( "Zhangyi", 18, "THU" );
    Student s2 = (Student) p2;
}
}
```

4.3 包

由于 Java 编译器为每个类生成一个字节码文件，且文件名与 `public` 的类名相同，因此同名的类有可能发生冲突。为了解决这一问题，Java 提供包(package)来管理类名空间。包实际上提供了一种命名机制和可见性限制机制。

包是一种松散的类的集合，一般不要求处于同一个包中的类有明确的相互关系，如包含、继承等，但是由于同一包中的类在默认情况下可以互相访问，所以为了方便编程和管理，通常把需要在一起工作的类放在一个包里。

4.3.1 package 语句

`package` 语句作为 Java 源文件的第一条语句，指明该文件中定义的类所在的包。它的格式为：

```
package pkg1[.pkg2[.pkg3...]];
```

Java 编译器把包对应于文件系统的目录管理。例如，在名为 `myPackage` 的包中，所有类文件都存储在目录 `myPackage` 下。同时，`package` 语句中，用 “.” 来指明目录的层次，例如：

```
package java.awt.image;
```

指定这个包中的文件存储在目录 `java/awt/image` 下。

包层次的根目录是由环境变量 `CLASSPATH` 来确定的。

在简单情况下，Java 源文件默认为 `package` 语句，这时称为无名包 (unnamed package)。无名包不能有子包。

注意：包及子包的定义，实际上是为了解决名字空间、名字冲突，它与类的继承没有关系。事实上，一个子类与其父类可以位于不同的包中。

Java 的 JDK 提供的包包括：`java.applet`, `java.awt`, `java.awt.image`, `java.awt.peer`, `java.io`, `java.lang`, `java.net`, `java.util`, `javax.swing`, `sun.tools.debug` 等。

每个包中都包含了许多有用的类和接口。用户也可以定义自己的包来实现自己的应用程序。在实际应用中，一种常见的做法是将包命名在组织机构之下，如 `com.sun.xxxxx`, `org.w3c.xxxx` 等，这样能更好地解决名字空间的问题。

4.3.2 import 语句

为了能使用 Java 中已提供的类，需要用 `import` 语句来引入所需要的类。`import` 语句的格式为：

```
import package1[.package2...]. (classname | *);
```

其中，`package1[.package2...]` 表明包的层次，与 `package` 语句相同，它对应于文件目录，`classname` 则指明所要引入的类，如果要从一个包中引入多个类，则可以用星号(*)来代替。例如：

```
import java.awt.*;
```

```
import java.util.Date;
```

Java 编译器为所有程序自动引入包 `java.lang`，因此不必用 `import` 语句引入它包含的所有类，但是若需要使用其他包中的类，必须用 `import` 语句引入。

注意：使用星号(*)只能表示本层次的所有类，不包括子层次下的类。

例如，经常需要用两条 `import` 语句来引入两个层次的类：

```
import java.awt.*;
import java.awt.event.*;
```

另外，在 Java 程序中使用类的地方，都可以指明包含它的包，这时就不必用 `import` 语句引入该类了。只是这样要敲入大量的字符。在一定意义上，使用 `import` 是为了使书写更方便。

如果引入的几个包中包括名字相同的类，则当使用该类时，必须指明包含它的包，使编译器能够载入特定的类。例如，类 `Date` 包含在包 `java.util` 中，可以用 `import` 语句引入它以实现它的子类 `myDate`：

```
import java.util.*;
class myDate extends Date{
.....
}
```

也可以直接写全其类名类：

```
class myDate extends java.util.Date{
.....
}
```

两者是等价的。

4.3.3 编译和运行包中的类

前面所举的很多例子中，没有用到 `package` 语句，即把文件中所有类都放在默认的无名包中，它对应于当前工作目录，编译和运行都比较简单。对于有 `package` 语句的情况，编译和运行时情况稍复杂一些，初学者经常发现解释器常常返回“can't find class”（找不到类）。下面介绍正确的使用方法。

当程序中用 `package` 语句指明一个包，这时，当在编译时，产生的字节码文件（.class 文件）需要放到相应的目录下，可以手工建立子目录，再将.class 文件复制到相应目录下。实际上在编译时，使用 `javac` 可以将.class 文件放入到相应的目录，只需要使用一个命令选项 `-d` 来指明包的根目录即可。

例 4-4 设包的根目录为 `d:\tang\ch04`。在 `d:\tang\ch04\pk` 目录下有文件 `TestPkg.java`，其内容为：

```
package pk;
class TestPkg
{
```

```
public static void main( String [] args){  
    System.out.println( "Test Package Ok." );  
}  
}
```

在编译时，可以用

```
javac -d d:\tang\ch04 d:\tang\ch04\pk\TestPkg.java
```

如果当前目录在 d:\tang\ch04，可以用以下命令：

```
javac -d . pk\*.java
```

其中，“.”表示当前目录。

运行该程序，需要指明含有 main 的类名：

```
java pk.TestPkg
```

注意：在运行时，指明的是类名 (pk.TestPkg)，不是文件名(pk\TestPkg.class)。在运行时，还会涉及 classpath 的问题，见 4.3.4 节。

4.3.4 CLASSPATH

在编译和运行程序中，经常要用到多个包，怎样指明这些包的根目录呢？简单地说，包层次的根目录是由环境变量 CLASSPATH 来确定的。具体操作有两种方法。

一是在 java 及 javac 命令行中，用-classpath 选项来指明，如：

```
java -classpath d:\tang\ch04;c:\java\classes;. pk.TestPkg
```

二是设定 classpath 环境变量，用命令行设定环境变量，如：

```
Set classpath= d:\tang\ch04;c:\java\classes;.
```

在 Windows 中还可以按第 2 章中的办法设定环境变量。

4.4 访问控制符

在使用类及其成员时，有时为了更好地控制类及其域、方法的存取权限，更好地实现信息的封装与隐藏，需要用到 public/private/protected 等表示访问控制(Access Control)的修饰符。

4.4.1 成员的访问控制符

类的成员变量和方法都有访问权限的控制。由于类中封装了数据和代码，包中封装了类和其他的包，所以 Java 提供了对类成员在四种范围中的访问权限的控制，这四种范围包括：同一个类中、同一个包中、不同包中的子类、不同包中的非子类。访问权限则包括 private, protected, public 和默认。表4-1 列出了在不同范围中的访问权限。

表 4-1 访问权限 (Yes 表示可以访问)

	同一个类中	同一个包中	不同包中的子类	不同包中的非子类
private	Yes			
默认	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes

1 . private

类中限定为 **private** 的成员（域或方法）只能被这个类本身访问，即私有访问控制。它的声明如下：

```
private privateVar;
private privateMethod ( [paramlist] ){
.....
}
```

注意：同一个类的不同对象可以访问对方的 **private** 成员变量或调用对方的 **private** 方法，这是因为访问保护是控制在类的级别上，而不是对象的级别上。

例如，在以下程序中，三种对域 **a** 的访问都是可以的。

```
class TestPrivate
{
    private int a;
    void m(){
        int i = a;
        int j = this.a;
        int k = new TestPrivate().a;
    }
}
```

由于 **private** 域或方法只能被这个类本身所访问，所以 **private** 域或方法不可能被子类所继承。当我们说“子类自动继承所有域和方法”时，是指非 **private** 的域或方法。

2 . 默认访问控制

类中的成员默认访问控制符时，称为默认访问控制。默认访问控制的成员可以被这个类本身和同一个包中的类所访问，即包访问控制。

也有人将默认访问控制与 C++ 的 **friendly** 相类比，但要注意要 Java 中没有 **friendly** 这个关键词。

3 . protected

类中限定为 **protected** 的成员可以被这个类本身、它的子类(包括同一个包中及不同包中的子类)及同一个包中所有其他的类访问。它的声明如下：


```
protected protectedVar;  
protected protectedMethod ( [paramlist] ){  
    .....  
}
```

Java 中还有一种访问控制符为 `private protected`，它限定能被本类及其子类可以访问，而包中的其他非子类的类不能访问。

4 . public

类中限定为 `public` 的成员可以被所有的类访问。它的声明如下：

```
public publicVar;  
public PublicMethod ( [paramlist]){  
    .....  
}
```

5 . 应用举例

这里通过具体的例子来说明上述访问权限。源文件有两个，其中一个为 `Original.java`；另一个为 `Access Control.java`。

例 4-5 `Original.java` 有关访问控制的练习。

```
package p1;  
  
public class Original{  
    int n_friendly = 1;  
    private int n_private = 2;  
    protected int n_protected = 3;  
    public int n_public = 4;  
    void Access(){  
        System.out.println("**** In same class,  
        you can access ...");  
        System.out.println("friendly member "+n_friendly);  
        System.out.println("private member "+n_private);  
        System.out.println("protected member "+n_protected);  
        System.out.println("public member "+n_public);  
    }  
}  
  
class Derived extends Original{  
    void Access(){
```

```
System.out.println("**** 相同包的子类 ****");
System.out.println("friendly member "+n_friendly);
// System.out.println("private member "+n_private);
// 不能访问
System.out.println("protected member "+n_protected);
System.out.println("public member "+n_public);

Original o = new Original();
System.out.println("**** 相同包的子类的其他对象 ****");
System.out.println("friendly member "+o.n_friendly);
// System.out.println("private member "+o.n_private);
// 不能访问
System.out.println("protected member "+o.n_protected);
System.out.println("public member "+o.n_public);
}
}

class SamePackageClass{
    void Access(){
        Original o = new Original();
        System.out.println("**** 相同包的其他类 ****");
        System.out.println("friendly member "+o.n_friendly);
        // System.out.println("private member "+o.n_private);
        //不能访问
        System.out.println("protected member "+o.n_protected);
        System.out.println("public member "+o.n_public);
    }
}

class AccessControl{
    public static void main( String args[ ] ){
        Original o = new Original();
        o.Access();
        Derived d = new Derived();
        d.Access();
        SamePackageClass s = new SamePackageClass();
        s.Access();
    }
}
```

另一个源文件为 AccessControl.java。

```
package p2;

class Derived extends p1.Original{
    void Access(){
        System.out.println("**** 不同包中的子类 ****");
        // System.out.println("friendly member "+n_friendly);
        // 不能访问
        // System.out.println("private member "+n_private);
        // 不能访问
        System.out.println("protected member "+n_protected);
        // 子类可以访问父类
        System.out.println("public member "+n_public);

        p1.Original o = new p1.Original();
        System.out.println("**** 访问在不同包中的父类 ****");
        // System.out.println("friendly member "+o.n_friendly);
        // 不能访问
        // System.out.println("private member "+o.n_private);
        // 不能访问
        // System.out.println("protected member "+o.n_protected);

        // 不能访问
        System.out.println("public member "+o.n_public);
    }
}

class AnotherPackageClass{
    void Access(){
        p1.Original o = new p1.Original();
        System.out.println("**** 另一包中的其他类 ****");
        // System.out.println("friendly member "+o.n_friendly);
        // 不能访问
        // System.out.println("private member "+o.n_private);
        // 不能访问
        // System.out.println("protected member "+o.n_protected);
        // 不能访问
        System.out.println("public member "+o.n_public);
    }
}
```

```
    }  
}  
  
public class AccessControl{  
    public static void main( String args[ ] ){  
        Derived d = new Derived();  
        d.Access();  
        AnotherPackageClass a = new AnotherPackageClass();  
        a.Access();  
    }  
}
```

将两个源文件都放在当前目录下，可以用以下命令编译：

```
javac -d . *.java
```

运行 java p1.AccessControl 结果如图4-4 所示。

```
C:\WINNT\System32\cmd.exe  
D:\tang\ch04\AccessControl>javac -d . *.java  
  
D:\tang\ch04\AccessControl>java p1.AccessControl  
**** In same class, you can access ...  
friendly member 1  
private member 2  
protected member 3  
public member 4  
**** 相同包的子类 ****  
friendly member 1  
protected member 3  
public member 4  
**** 相同包的子类的其他对象 ****  
friendly member 1  
protected member 3  
public member 4  
**** 相同包的其他类 ****  
friendly member 1  
protected member 3  
public member 4  
D:\tang\ch04\AccessControl>
```

图 4-4 有关访问控制

运行 java p2.AccessControl 结果如图4-5 所示。

```
C:\WINNT\System32\cmd.exe  
D:\tang\ch04\AccessControl>javac -d . *.java  
  
D:\tang\ch04\AccessControl>java p2.AccessControl  
**** 不同包中的子类 ****  
protected member 3  
public member 4  
**** 访问在不同包中的父类 ****  
public member 4  
**** 另一包中的其他类 ****  
public member 4  
D:\tang\ch04\AccessControl>
```

图 4-5 有关访问控制的运行结果

4.4.2 类的访问控制符

在定义类时，也可以用访问控制符。类的访问控制符或者为 `public`，或者默认。若使用 `public`，其格式为：

```
public class 类名{  
    ...  
}
```

如果类用 `public` 修饰，则该类可以被其他类所访问；若类默认访问控制符，则该类只能被同包中的类访问。

4.4.3 setor 与 getor

在 Java 编程中，有一种常见的做法，是将所有的或部分的域用 `private` 修饰，从而更好地将信息进行封装和隐藏。在这样的类中，用 `setXXXX` 和 `getXXXX` 方法对类的属性进行存取，分别称为 `setor` 与 `getor`。这种方法有以下优点。

- (1) 属性用 `private` 更好地封装和隐藏，外部类不能随意存取和修改。
- (2) 提供方法来存取对象的属性，在方法中可以对给定的参数的合法性进行检验。
- (3) 方法可以用来给出计算后的值。
- (4) 方法可以完成其他必要的工作（如清理资源、设定状态，等等）。
- (5) 只提供 `getXXXX` 方法，而不提供 `setXXXX` 方法，可以保证属性是只读的。

例如：在类 `Person` 中将域 `age` 以 `set` 和 `get` 方法提供。

```
class Person2  
{  
    private int age;  
    public void setAge( int age ){  
        if (age>0 && age<200) this.age = age;  
    }  
    public int getAge(){  
        return age;  
    }  
}
```

4.4.4 构造方法的隐藏

对于构造方法，也可用访问控制符修饰符。若构造方法默认访问控制符，则可访问包；若为 `public`，则所有地方都可访问；若构造方法声明为 `private`，则其他类不能生成该类的一个实例。`private` 的构造方法用在一些特殊场合，本章及后面的章节有这样的例子，这里就不赘述。

4.5 非访问控制符

类、域、方法可以拥有若干修饰符，包括访问控制符和非访问控制符。4.4 节介绍了访问控制符，本节将讨论非访问控制符。Java 的常用非访问控制符如，`static`, `final`, `abstract` 等，也可用于对类、成员进行修饰。表 4-2 列出了它们的含义及能修饰的成分。

表 4-2 非访问控制符

非访问控制符	基本含义	修饰类	修饰成员	修饰局部变量
<code>static</code>	静态的、非实例的、类的	只可以修饰内部类	Yes	
<code>final</code>	最终的、不可改变的	Yes	Yes	Yes
<code>abstract</code>	抽象的、不可实例化的	Yes	Yes	

4.5.1 `static`

在类中声明一个域或方法时，可以用 `static` 进行修饰。格式如下：

```
static type classVar;  
static returnType classMethod ( [paramlist] ){  
.....  
}
```

分别声明了类域和类方法。如果在声明时不用 `static` 修饰，则声明实例变量和实例方法。

1. 类域

用 `static` 修饰符修饰的域仅属于类的静态域，又称为静态量、类域、类变量。与此相对，不用 `static` 修饰的域称为实例变量、实例域。

静态域最本质的特点是：它们是类的域，不属于任何一个类的具体对象实例。它不保存在某个对象实例的内存区间中，而是保存在类的内存区域的公共存储单元。换句话说，对于该类的任何一个具体对象而言，静态域是一个公共的存储单元，任何一个类的对象访问它，取到的都是相同的数值；同样任何一个类的对象去修改它，也都是在对同一个内存单元进行操作。

类变量可以通过类名直接访问，也可以通过实例对象来访问，两种方法的结果是相同的。如我们用到的 JDK 中的 `System` 类的 `in` 和 `out` 对象，就是属于类的域，直接用类名来访问，即 `System.in` 和 `System.out`。

又如，在类 `Person` 中可以定义一个类域为 `totalNum`：

```
class Person {  
    static long totalNum;  
    int age;  
    String Name;  
}
```

`totalNum` 代表人类的总人数，它与具体对象实例无关。可以有两种方法来访问：`Person.totalNum` 和 `p.totalNum` (假定 `p` 是 `Person` 对象)。

2. 类方法

用 `static` 修饰符修饰的方法仅属于类的静态方法，又称为类方法。与此相对，不用 `static` 修饰的方法，则为实例方法。类方法的本质是该方法是属于整个类的，不是属于某个实例的。

声明一个方法为 `static` 有以下几重含义。

(1) 非 `static` 的方法是属于某个对象的方法，在这个对象创建时，对象的方法在内存中拥有自己专用的代码段。而 `static` 的方法是属于整个类的，它在内存中的代码段将随着类的定义而进行分配和装载，不被任何一个对象专有。

(2) 由于 `static` 方法是属于整个类的，所以它不能操纵和处理属于某个对象的成员变量，而只能处理属于整个类的成员变量，即 `static` 方法只能处理 `static` 域或调用 `static` 方法。

(3) 类方法中，不能访问实例变量。在类方法中不能使用 `this` 或 `super`。

(4) 调用这个方法时，应该使用类名直接调用，也可以用某一个具体的对象名。

例如：前面章节用到的方法 `Math.random()`，`Integer.parseInt()` 等就是类方法，直接用类名进行访问。

例 4-6 StaticAndInstanceTest.java 使用 `static`。

```
class StaticAndInstance{
    static int classVar;
    int instanceVar;
    static void setClassVar( int i ){
        classVar = i;
        // instanceVar = i;    // 不能在类方法中存取实例变量
    }
    static int getClassVar(){
        return classVar;
    }
    void setInstanceVar( int i ){
        classVar = i;          // 可以在实例方法中存取类域
        instanceVar = i;
    }
    int getInstanceVar(){
        return instanceVar;
    }
}

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

```

        StaticAndInstance m1 = new StaticAndInstance();
        StaticAndInstance m2 = new StaticAndInstance();
        m1.setClassVar( 1 );
        m2.setClassVar( 2 );
        System.out.println("m1.classVar   =  "+m1.getClassVar()+"
        m2.classVar = "+m2.getClassVar());
        m1.setInstanceVar( 11 );
        m2.setInstanceVar( 22 );
        System.out.println("m1.InstanceVar
        = "+m1.getInstanceVar()+" m2.InstanceVar
        = "+m2.getInstanceVar());
    }
}

```

运行结果为:

```

m1.classVar = 2  m2.classVar = 2
m1.InstanceVar = 11  m2.InstanceVar = 22

```

从类成员的特性可以看出, 可用 `static` 来定义全局变量和全局方法, 这时由于类成员仍然封装在类中, 与 C、C++ 相比, 可以限制全局变量和全局方法的使用范围而防止冲突。

由于可以从类名直接访问类成员, 所以访问类成员前不需要对它所在的类进行实例化。作为程序入口的 `main()` 方法必须要用 `static` 来修饰, 也是因为 Java 运行时系统在开始执行一个程序前, 并没有生成类的一个实例, 它只能通过类名来调用 `main()` 方法作为程序的入口。

3. 静态初始化器

静态初始化器是由关键字 `static` 引导的一对大括号 `{}` 括起的语句组。它的作用与类的构造方法有些相似, 都是用来完成初始化的工作, 但是静态初始化器在三点上与构造方法有根本的不同。

(1) 构造方法是对每个新创建的对象初始化, 而静态初始化器是对类自身进行初始化。

(2) 构造方法是在用 `new` 运算符产生新对象时由系统自动执行; 而静态初始化器一般不能由程序来调用, 它是在所属的类加载入内存时由系统调用执行。

(3) 不同于构造方法, 静态初始化器不是方法, 没有方法名、返回值和参数列表。

(4) 同 `static` 方法一样, 静态初始化器不能访问实例域和实例方法。

例如, 可以在 `Person` 类中加入静态初始化器, 如:

```

class Person {
    static long totalNum;
    static {
        totalNum = (long)52e8;
        System.out.println("人类总人口"+ totalNum );
    }
}

```



```
    }  
}
```

如果有多个 `static{}` 程序段，则它们在类的初始化时，会依次执行。

4.5.2 final

1. final 类

如果一个类被 **final** 修饰符所修饰和限定，说明这个类不能被继承，即不可能有子类。

被定义为 **final** 的类通常是一些有固定作用、用来完成某种标准功能的类。如 Java 系统定义好的用来实现网络功能的 **InetAddress**, **Socket** 等类都是 **final** 类。将一个类定义为 **final** 则可以将它的内容、属性和功能固定下来，与它的类名形成稳定的映射关系，从而保证引用这个类时所实现的功能的正确无误。

2. final 方法

final 修饰符所修饰的方法，是不能子类所覆盖的方法。如果类的某个方法被 **final** 修饰符所限定，则该类的子类就不能再重新定义与此方法同名的自己的方法，这样就固定了这个方法所对应的具体操作，可以防止子类对父类关键方法的错误的重定义，保证了程序的安全性和正确性。

注意：所有已被 **private** 修饰符限定为私有的方法，以及所有包含在 **final** 类中的方法，都被默认为是 **final** 的。因为这些方法不可能被子类所继承，所以都不可能被重载，自然都是最终的方法。

3. final 域及 final 局部变量

final 域、**final** 局部变量，它们的值一旦给定，就不能更改。大体上说，**final** 域、**final** 局部变量是只读量，它们能且只能被赋值一次，而不能赋值多次。

一个域被 **static final** 两个修饰符所限定时，它实际的含义就是常量，如 **Integer.MAX_VALUE**(表示最大整数)、**Math.PI**(表示圆周率)就是这种常量。在程序中，通常用 **static** 与 **final** 一起使用来指定一个常量。

在定义 **static final** 域时，若不给定初始值，则按默认值进行初始化（数值为 0，boolean 型为 **false**，引用型为 **null**）。

在定义 **final** 域时，若不是 **static** 的域，则必须且只能赋值一次，不能缺省。这种域的赋值的方式有两种：一是在定义变量时赋初始值，二是在每一个构造函数中进行赋值。

在定义 **final** 局部变量(方法中的变量)时，也必须且只能赋值一次。它的值可能不是常量，但它的取值在变量存在期间不会改变。

例 4-7 TestFinal.java 使用 final。

```
public final class TestFinal{
    public static int totalNumber= 5 ;
    public static final int id;

    public TestFinal(){
```

```
        // 在构造方法中对声明为 final 的变量 id 赋值
        id = ++totalNumber;
    }

    public static void main(String[] args) {
        TestFinal t = new TestFinal();
        System.out.println(t.id);
        final int i = 10;
        final int j;
        j = 20;
        //j = 30;           // 非法
    }
}
```

4.5.3 abstract

1. abstract 类

凡是用 **abstract** 修饰符修饰的类被称为抽象类。抽象类就是没有具体对象的概念类。

由于抽象类是所有子类的公共属性的集合，所以使用抽象类的一大优点就是可以充分利用这些公共属性来提高开发和维护程序的效率。这种把各类的公共属性从它们各自的类定义中抽取出来形成一个抽象类的组织方法显然比把公共属性保留在具体类中的方法要方便得多。

例如，Java 中的 **Number** 类就是一个抽象类，它只表示数字这一抽象概念，只有当它作为整数类 **Integer** 或实数类 **Float** 等的父类时才有意义，定义一个抽象类的格式如下：

```
abstract class abstractClass{
    .....
}
```

由于抽象类不能被实例化，因此下面的语句会产生编译错误：

```
new abstractClass(); //abstract class can't be instantiated
```

抽象类不能用 **new** 来实例化。但是抽象类可以有构造函数，构造函数可以被子类的构造函数所调用。

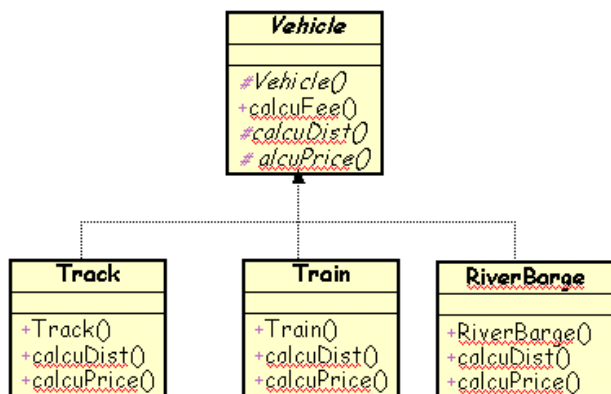


图 4-6 抽象类及其子类

又如,图 4-6 中的抽象类 *Vehicle*(抽象类名在 UML 图中用斜体表示),它的子类 *Track*, *Train*, *RiverBarge* 则可以是可实例化的类。

由于抽象类是需要继承的,所以 `abstract` 类不能用 `final` 来修饰。

抽象类的子类还可以是 `abstract` 类,但只有非 `abstract` 的类才能被实例化。

2. `abstract` 方法

被 `abstract` 所修饰的方法叫抽象方法,抽象方法的作用在为所有子类定义一个统一的接口。对抽象方法只需声明,而不需实现,即用分号 (;) 而不是用 {}, 格式如下:

```
abstract returnType abstractMethod( [paramlist] );
```

抽象类中可以包含抽象方法,也可以不包含 `abstract` 方法。但是,一旦某个类中包含了 `abstract` 方法,则这个类必须声明为 `abstract` 类。

抽象方法在子类中必须被实现,否则子类仍然是 `abstract` 的。

例 4-8 `AbstractTest.java` 使用 `abstract`。

```

abstract class C{
    abstract void callme();
    void metoo(){
        System.out.println("Inside C's metoo() method");
    }
}

class D extends C{
    void callme(){
        System.out.println("Inside D's callme() method");
    }
}
  
```

```
public class AbstractTest{  
    public static void main( String args[ ] ){  
        C c = new D();  
        c.callme();  
        c.metoo();  
    }  
}
```

运行结果为：

```
Inside D's callme() method  
Inside C's metoo() method
```

该例中，首先定义了一个抽象类 C，其中声明了一个抽象方法 callme()，然后定义它的子类 D，并覆盖方法 callme()。最后，在类 Abstract 中，生成类 D 的一个实例，并把它的引用返回到 C 型变量 C 中。

在使用 abstract 注意要以下几点。

注意：(1) abstract 不能与 final 并列修饰同一类。

(2) abstract 不能与 private, static, final 或 native 并列修饰同一方法。

(3) abstract 方法必须位于 abstract 类中。

4.5.4 其他修饰符

在 Java 中还有其他一些修饰符，如 volatile, native, synchronized 等，这里仅作简单介绍，读者可以参考相关的文献。

1 . volatile 易失域

如果一个域被 volatile 修饰符所修饰，说明这个域可能同时被几个线程所控制和修改，即这个域不仅仅被当前程序所掌握，在运行过程中可能存在其他未知的程序操作来影响和改变该域的取值。在使用当中应该特别留意这些影响因素。

通常，volatile 用来修饰接受外部输入的域。如，表示当前时间的变量将由系统的后台线程随时修改，以保证程序中取到的总是最新的当前的系统时间，所以可以把它定义为易失域。

2 . native 本地方法

native 修饰符一般用来声明用其他语言书写方法体并具体实现方法功能的特殊的方法，这里的其他语言包括 C, C++, FORTRAN, 汇编等。由于 native 的方法的方法体使用其他语言在程序外部写成，所以所有的 native 方法都没有方法体，而用一个分号代替。

在 Java 程序里使用其他语言编写的模块作为类方法，其目的主要有两个：充分利用已经存在的程序功能模块和避免重复工作。

但是，在 Java 程序中使用 native 方法时应该特别注意。由于 native 方法对应其他语言

书写的模块是以非 Java 字节码的二进制代码形式嵌入 Java 程序的,而这种二进制代码通常只能运行在编译生成它的平台之上,所以整个 Java 程序的跨平台性能将受到限制或破坏,因此使用这类方法时应特别谨慎。

3. synchronized 同步方法

如果 synchronized 修饰的方法是一个类的方法(即, static 的方法),那么在被调用执行前,将把系统类 Class 中对应当前类的对象加锁。如果 synchronized 修饰的是一个对象的方法(未用 static 修饰的方法),则这个方法在被调用执行前,将把当前对象加锁。synchronized 修饰符主要用于多线程共存的程序中的协调和同步。详细的内容将在以后的线程同步中介绍。

4.5.5 一个应用模型——单子

模型(pattern)是面向对象程序设计中在众多类似的应用中抽象出的类之间的关系。这里介绍一种模型是单子(Singleton)。单子是指在某个类只有一个实例,调用者可以获得该实例,并且这个实例是惟一的。

实现这种模式有一个方法,就是将该类的构造函数设定为 private,使得外部调用者不能直接用 new 来进行创建其实例。然后在该类中,用 static 的域来存放该类的惟一实例,并将该实例以 public 方法向外进行公开。这里利用了 private, public, static 等修饰符来实现这一模式。具体例子见例 4-9。

例 4-9 TestSingle.java 单子模式。

```
class Single{
    private static Single onlyone = new Single();
    private String name;
    public static Single getSingle() {
        return onlyone;
    }
    private Single() {}
}

public class TestSingle{
    public static void main(String args[]) {
        Single s1 = Single.getSingle();
        Single s2 = Single.getSingle();
        if (s1==s2){
            System.out.println("s1 is equals to s2!");
        }
    }
}
```

4.6 接口

4.6.1 接口的概念

Java 中的接口(interface)在语法上有些相似于类,它定义了若干个抽象方法和常量,形成一个属性集合,该属性集合通常对应于某一组功能,其主要作用是可以帮助实现类似于类的多重继承的功能。

所谓多重继承,是指一个子类可以有一个以上的直接父类,该子类可以继承它所有直接父类的成员。Java 不支持多重继承,而是用接口(interface)实现比多重继承更强的功能。编程者可以把用于完成特定功能的若干属性组织成相对独立的属性集合,凡是需要实现这种特定功能的类,都可以继承这个属性集合并在类内使用它,这种属性集合就是接口。

前面在图形界面程序中使用的 `ActionListener` 就是系统定义的接口,它代表了监听并处理动作事件的功能,其中包含了一个抽象的方法:

```
public void actionPerformed(ActionEvent e);
```

所有希望能够处理动作事件(如单击按钮、在文本框中回车等)的类都必须具有 `ActionListener` 接口定义的功能。具体地说,就是必须实现这个接口,覆盖 `actionPerformed()` 方法。

接口就是方法定义和常量值的集合。它的作用主要体现在下面几个方面。

- (1) 通过接口可以实现不相关类的相同行为,而不需要考虑这些类之间的层次关系。
- (2) 通过接口可以指明多个类需要实现的方法。
- (3) 通过接口可以了解对象的交互界面,而不需了解对象所对应的类。

例如,要实现一个堆栈,其中的每个单元可以是任意类型的对象,如整数、实数、字符串,或是一个表格等。在堆栈的处理过程中,对每个单元的对象都会有一些相同的处理方法。一种方法是找到这些类的一个共同的父类并在其中实现相同的处理方法,但是由于这些类在本质上是没有任何相关关系的,因此通过继承的方法来实现堆栈单元是不可行的。另一种方法,即通过接口来实现。在接口中定义这些类共同的行为,然后由每个类分别实现这些行为。同时,因为接口也是一种引用数据类型,可以把堆栈单元的类型作为接口,然后就可以通过它调用各个不同类对象的相应方法。

需要特别说明的是,Java 中一个类获取某一接口定义的功能,并不是通过直接继承这个接口中的属性和方法来实现的。因为接口中的属性都是常量,接口中的方法都是没有方法体的抽象方法。也就是说,接口定义的仅仅是实现某一特定功能的一组功能的对外接口和规范,而并没有真正地实现这个功能,这个功能的真正实现是在“继承”这个接口的各个类中完成的,要由这些类来具体定义接口中各抽象方法的方法体。因而在 Java 中,通常把对接口功能的“继承”称为“实现(implements)”。

总之,接口把方法的定义和对它的实现区分开来。同时,一个类可以实现多个接口来达到实现与“多重继承”相似的目的。

Java 通过接口使得处于不同层次、甚至互不相关的类可以具有相同的行为。例如,在图 4-7 中,接口 `Flyable` (可飞)具有 `tackoff()`, `fly()`, `land()` 等方法,它可以被 `Airplane`, `Bird`,

Superman 等类来实现，而这些类并没有继承关系体系，也不一定处于同样的层次上。

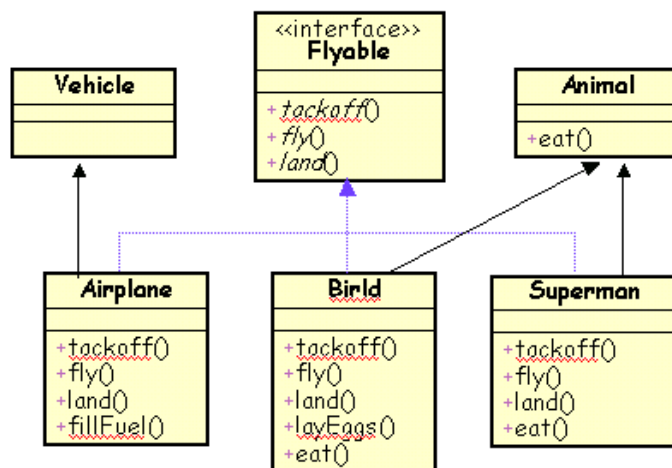


图 4-7 接口的实现

4.6.2 定义接口

Java 中声明接口的语法如下：

```

[public]interface 接口名[extends 父接口名列表]
{
    // 接口体
    // 常量域声明
    [public][static][final]域类型域名=常量值;
    // 抽象方法声明
    [public][abstract]返回值方法名(参数列表)[throw 异常列表];

```

从上面的语法规则可以看，定义接口与定义类非常相似。

其中，interface 前的修饰符可以为 public 或默认，其含义与类的访问控制相似。通常接口名以 able 或 ible 结尾，表明接口能完成一定的行为。

域及方法的定义的修饰符与类的定义相似，但也有不同之处。

(1) 接口中的域实际上是常量，域前面即使省略修饰符，仍然默认为 public static final。

(2) 接口中的方法都是抽象方法，不能有实现体；方法前面即使省略修饰符，仍然默认为 public abstract。

事实上，在定义接口时，一般都省略域和方法的修饰符。

与类相似，接口也具有继承性。定义一个接口时可以通过 extends 关键字声明该新接口是某个已经存在的父接口的派生接口，它将继承父接口的所有属性和方法。与类的继承不同的是，一个接口可以有一个以上的父接口，它们之间用逗号分隔，形成父接口列表。新接口将继承所有父接口中的属性和方法。

例 4-10 给出一个接口的定义。

```
interface Collection {  
    int MAX_NUM=100;  
    void add (Object obj);  
    void delete (Object obj);  
    Object find (Object obj);  
    int currentCount ();  
}
```

该例定义了一个名为 **Collection** 的接口，其中声明了一个常量和四个方法。这个接口可以由队列、堆栈、链表等类来实现。

4.6.3 实现接口

接口的声明仅仅给出了抽象方法，相当于程序开发早期的一组协议，而具体地实现接口所规定的功能，则需某个类为接口中的抽象方法书写语句并定义实在的方法体，称为实现这个接口。

一个类要实现接口时，请注意以下问题。

-
- 注意：**(1) 在类的声明部分，用 **implements** 关键字声明该类将要实现哪些接口。
- (2) 如果实现某接口的类不是 **abstract** 的抽象类，则在类的定义部分必须实现指定接口的所有抽象方法。如果其直接或间接父类中实现了接口，父类所实现的接口中的所有抽象方法都必须有实在的方法体。也就是说，非抽象类中不能存在抽象方法。
- (3) 一个类在实现某接口的抽象方法时，必须使用完全相同的方法头。否则，只是在重载一个新的方法，而不是实现已有的抽象方法。
- (4) 接口的抽象方法的访问限制符都已指定为 **public**，所以类在实现方法时，必须显式地使用 **public** 修饰符，否则将被系统警告为缩小了接口中定义的方法的访问控制范围。
- (5) 一个类只能有一个父类，但是它可以同时实现若干个接口。一个类实现多个接口时，在 **implements** 子句中用逗号分隔。这种情况下如果把接口理解成特殊的类，那么这个类利用接口实际上就获得了多个父类，即实现了多重继承。
-

下面在类 **FIFOQueue** 中实现上面所定义的接口 **Collection**：

```
class FIFOQueue implements collection{  
    void add ( Object obj ){  
        .....  
    }  
    void delete( Object obj ){  
        .....  
    }  
}
```

```
Object find( Object obj ){
.....
}
int currentCount{
.....
}
}
```

例 4-11 TestInterface.java 使用接口。

```
interface Runner { public void run();}

interface Swimmer {public void swim();}

abstract class Animal {abstract public void eat();}

class Person extends Animal implements Runner,Swimmer {
    public void run() { System.out.println("run"); }
    public void swim() { System.out.println("swim"); }
    public void eat() { System.out.println("eat"); }
}

public class TestInterface{
    public static void main(String args[]){
        TestInterface t = new TestInterface();
        Person p = new Person();
        t.m1(p);
        t.m2(p);
        t.m3(p);
    }
    public void m1(Runner f) { f.run(); }
    public void m2(Swimmer s) {s.swim();}
    public void m3(Animal a) {a.eat();}
}
```

4.6.4 对接口的引用

接口可以作为一种引用类型来使用。任何实现该接口的类的实例都可以存储在该接口类型的变量中，通过这些变量可以访问类所实现的接口中的方法。**Java** 运行时系统动态地确定该使用哪个类中的方法。

把接口作为一种数据类型可以不需要了解对象所对应的具体的类，而着重于它的交互

界面，仍以前面所定义的接口 `Collection` 和实现该接口的类 `FIFOQueue` 为例，下例中，以 `Collection` 作为引用类型来使用。

```
class InterfaceType{
    public static void main( String args[] ){
        Collection c = new FIFOQueue();
        .....
        add( obj );
        .....
    }
}
```

习题

1. 使用抽象和封装有哪些好处？
2. 编写一个 Java 程序片断定义一个表示学生的类 `student`，包括域（学号、班号、姓名、性别、年龄）和方法（获得学号、获得班号、获得性别、获得年龄、修改年龄）。
3. 为 `student` 类定义构造方法初始化所有的域，增加一个方法 `public String loString()` 把 `StdIdent` 类对象的所有域信息组合成一个字符串。编写 `Application` 程序检验新增的功能。
4. 什么是最终类？如何定义最终类？试列举最终类的例子。
5. 如何定义静态域？静态域有什么特点？如何访问和修改静态域的数据？
6. 什么是静态初始化器？它有什么特点？与构造方法有什么不同？
7. 如何定义方法？在面向对象程序设计中方法有什么作用？
8. 什么是抽象方法？它有何特点？如何定义抽象方法？如何使用抽象方法？
9. 如何定义静态方法？静态方法有何特点？静态方法处理的域有什么要求？
10. 什么是访问控制符？有哪些访问控制符？哪些可以用来修饰类？哪些用来修饰域和方法？试述不同访问控制符的作用。
11. 修饰符是否可以混合使用？混合使用时需要注意什么问题？
12. 什么是继承？什么是父类？什么是子类？继承的特性给面向对象编程带来什么好处？什么是单重继承？什么是多重继承？
13. 如何定义继承关系？为“学生”类派生出“小学生”、“中学生”、“大学生”、“研究生”四个类，其中“大学生”类再派生出“一年级学生”、“二年级学生”、“三年级学生”、“四年级学生”四个子类，“研究生”类再派生出“硕士生”和“博士生”两个子类。
14. “子类的域和方法的数目一定大于等于父类的域和方法的数目”，这种说法是否正确？为什么？
15. 什么是域的隐藏？
16. 什么是方法的覆盖？方法的覆盖与域的隐藏有何不同？与方法的重载有何不同？
17. 解释 `this` 和 `super` 的意义和作用。
18. 父类对象与子类对象相互转化的条件是什么？如何实现它们的相互转化？

19. 构造方法是否可以被继承？是否可以被重载？试举例说明。
20. 什么是包？它的作用是什么？
21. 如何创建包？在什么情况下需要在程序里创建包？
22. 如何引用包中的某个类？如何引用整个包？如果编写 Java Applet 程序时想把整个 java.applet 包都加载，该怎样实现？
23. CLASSPATH 是有关什么的环境变量？它如何影响程序的运行？如何设置和修改这个环境变量？
24. 什么是接口？为什么要定义接口？接口与类有何异同？如何定义接口？使用什么关键字？
25. 一个类如何实现接口？实现某接口的类是否一定要重载该接口中的所有抽象方法？

第5章 深入理解 Java 语言

第4章介绍了Java语言中面向对象的基本概念和相关的语法规则，有了这些基础，就可以编写完整的Java程序了。本章介绍Java语言中一些更深入的特性，通过本章的学习可以让读者对Java语言有进一步理解。对于时间不太充裕的读者，可以略过此章，而不会对后面各章的理解带来太大的影响；也可以在学过后面几章后，再回过头来学习本章。

5.1 变量及其传递

5.1.1 基本类型变量与引用型变量

Java中的变量，可分为基本类型变量（primitive）和引用型变量（reference）两种。基本类型变量包括8种类型：char, byte, short, int, long, float, double, boolean。引用类型包括对象、接口、数组。

基本类型变量与引用型变量在内存中的存储方式是不同的，基本类型的值直接存于变量中；而引用型的变量则不同，除要占据一定的内存空间外，同时，它所引用的对象实体（也就是用new创建的对象实体）也要占据一定的空间。通常对象实体占用的内存空间要大得多。

例 5-1 MyDate.java 基本类型变量与引用型变量的区别。

```
public class MyDate {
    private int day = 12;
    private int month = 6;
    private int year = 1900;
    public MyDate(int y, int m, int d) {
        year = y;
        month = m;
        day = d;
    }
    void addYear()
    {
        year++;
    }
    public void display() {
        System.out.println(year + "-" + month + "-" + day);
    }
    public static void main(String[] args) {
```

```
MyDate m, n;  
m = new MyDate(22, 9, 2003);  
n = m;  
n.addYear();  
m.display();  
n.display();  
}  
}
```

以 `MyDate` 类为例，其中定义了 3 个域（`day`，`month`，`year`）和一些方法，这些域和方法保存在一块内存中，这块内存就是 `m` 所引用的对象所占用的内存。变量 `m`，`n` 与它所引用的实体所占据的关系，是一种引用关系，可以用图 5-1 表示。引用型变量保存的实际上是对象在内存的地址，也称为对象的句柄。

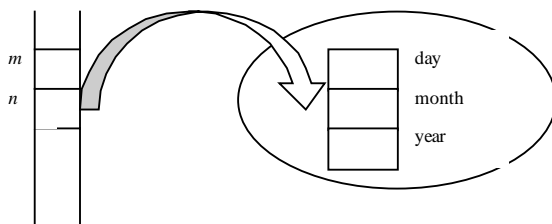


图 5-1 引用型变量与对象实体的关系

在 `m`，`n` 两个变量中，保存的是所引用的对象的地址。当调用 `n.addYear()` 方法，是将它引用的对象的 `year` 域加 1，由于 `m`，`n` 两个变量引用的是同一变量，所以它相当于 `m.addYear`。并且 `m.display` 与 `n.display` 方法的显示结果是一样的。

由于一个对象实体可能被多个变量所引用，在一定意义上就是一个对象有多个别名，通过一个引用可以改变另一个引用所指向的对象实体的内容。

5.1.2 域变量与局部变量

域变量与方法中的局部变量也是有区别的。

从语法形式上看，域变量是属于类或接口的，而局部变量是在方法中定义的变量或方法的参变量；域变量可以被 `public`，`private`，`static` 等词修饰，而局部变量则不能被访问控制符及 `static` 修饰；域变量及局部变量都可以被 `final` 修饰。

从变量在内存中的存储方式上看，域变量是对象的一部分，而对象是存在于堆中的，而局部变量是存在于栈中的。

从变量在内存中的存在时间上看，域变量是对象的一部分，它随着对象的创建而存在，而局部变量随着方法的调用而产生，随着方法调用结束而自动消失。

域变量及局部变量还有一个重要区别，域变量如果没有赋初值，则会自动以该类型的

默认值(0,false,null 等)赋值(有一种例外情况,被 final 但没有被 static 修饰的域变量必须显式地赋值);而局部变量则不会自动赋值,必须显式地赋值后才能使用。

例 5-2 LocalVarAndMemberVar.java 域变量与局部变量 class LocalVarAndMemberVar。

```
{
    int a;
    void m(){
        int b;
        System.out.println(a);    // a 的值为 0
        System.out.println(b);    // 编译不能通过
    }
}
```

类似地,数组的元素可认为是数组的成员,所以当数组用 new 创建并分配空间后,每一个元素会自动地赋值为默认值。

5.1.3 变量的传递

Java 中调用对象的方法时,需要进行参数的传递。在传递参数时,Java 遵循的是值传递,也就是说,当调用一个方法时,是将表达式的值复制给形式参数。

例 5-3 TransByValue.java 变量的传递。

```
public class TransByValue {
    private static int a;
    public static void main (String[] args) {
        int a = 0;
        modify (a); System.out.println(a);    // 输出 0

        int [] b = new int [1];
        modify(b);
        System.out.println(b[0]);              // 输出 1
    }

    public static void modify (int a) {
        a++;
    }
    public static void modify (int[] b) {
        b[0] ++;
        b = new int[5];
    }
}
```

在这个例子中，第 1 个 `modify()` 方法的参数是基本数据类型 (`int`)，第 2 个 `modify()` 方法的参数是引用数据类型 (数组类型 `int[]`)。

在调用第 1 个 `modify()` 方法时，将静态变量 `a` 的值 0 复制并传递给 `modify()` 方法的局部变量 `a` (参数变量也是局部变量)，局部变量 `a` 增加 1，然后返回 (这时局部变量 `a` 消失)，而静态变量 `a` 的值并没有受到影响，所以仍为 0。

在调用第 2 个 `modify()` 方法时，将 `main()` 方法中的局部变量 `b` 的值复制给 `modify()` 方法的局部变量 (参变量) `b`，由于 `b` 是引用型数据，这里复制不是对象实体 (位于堆中的数组元素) 本身，而是对象的地址 (即引用)，所以在 `modify()` 方法中 `b` 访问的是同一对象实体，数组的第 0 个元素增加 1，后来 `modify()` 中的局部变量 `b` 引用了一个新的数组，但这不影响 `main()` 中的 `b`，`main()` 中的 `b` 指向的仍是原来的数组。由于原来的数组的第 0 个元素已被增加到了 1，所以显示的值是 1。

从这个例子可以看出，Java 中的参数都是按值传递的，但对于引用型变量，传递的值是引用值，所以方法中对数据的操作可以改变对象的属性。

可以这样说，Java 中通过引用型变量这一概念，代替了其他语言中的指针，并且更安全。

5.1.4 变量的返回

与变量的传递一样，方法的返回值可以是基本类型，也可以是引用类型。返回值是基本类型的情形比较好理解，这里谈一下引用类型的返回。

如果一个方法返回一个引用类型，由于引用类型是一个引用，所以它可以存取对象实体。如：

```
Object GetNewObject()  
{  
    Object obj = new Object();  
    return obj;  
}
```

在调用方法时，可以这样用：

```
Object p = GetNewObject();
```

由于 Java 中的所有东西都是句柄 (引用)，而且由于每个对象都是在内存堆中创建的——只有不再需要的时候，才会当做垃圾收集掉，所以不必关心在需要一个对象的时候它是否仍然存在，因为系统会自动处理这一切。程序可在需要的时候创建一个对象，而且不必关注那个对象的传输机制的细节，只需简单地传递或返回引用即可。

5.2 多态与虚方法调用

对于面向对象的程序设计语言，多态性 (Polymorphism) 是第三种最基本的特征 (前两种是数据抽象和继承)。

所谓多态，是指一个程序中相同的名字表示不同的含义的情况。面向对象的程序中多态的情况有多种，简单情况下，可以通过子类对父类方法的覆盖 (override) 实现多态，也可

以利用重载(overload)在同一个类中定义多个同名的不同方法。

在面向对象的程序中，多态还有更为深刻的含义，就是动态绑定（dynamic binding），也称虚方法调用(virtual method invoking)，它能够使对象所编写的程序，不用做修改就可以适应于其所有的子类，如在调用方法时，程序会正确地调用子对象的方法。由此可见，多态的特点大大提高了程序的抽象程度和简洁性，更重要的是，它最大限度地降低了类和程序模块之间的耦合性，提高了类模块的封闭性，使得它们不需了解对方的具体细节，就可以很好地共同工作。这个优点对于程序的设计、开发和维护都有很大的好处。

本节介绍多态及其实现中的一些概念及相关问题。

5.2.1 上溯造型

类似于基本数据类型数据之间的强制类型转换，存在继承关系的父类对象引用和子类对象引用之间也可以在一定条件下相互转换。父类对象和子类对象的转化需要注意如下原则。

-
- 注意：
- (1) 子类对象可以被视为其父类的一个对象。如一个 Student 对象也是一个 Person 对象。
 - (2) 父类对象不能被当做其某一个子类的对象。
 - (3) 如果一个方法的形式参数定义的是父类对象，那么调用这个方法时，可以使用子类对象作为实际参数。
 - (4) 如果父类对象引用指向的实际是一个子类对象，那么这个父类对象的引用可以用强制类型转换转化成子类对象的引用。
-

其中，把派生类型当做它的基本类型处理的过程，又称为“Upcasting”（上溯造型），这一点具有特别重要的意义。我们知道，对于类有继承关系的一系列类而言，将派生类的对象当做基础类的一个对象对待，这意味着我们只需编写单一的代码，只与基础类打交道，而忽略派生类型的特定细节，这样的代码更易编写和理解。此外，若通过继承增添了一种新类型，新类型会像在原来的类型里一样正常地工作，使程序具备了扩展性。

假设我们用 Java 写了这样一个函数：

```
void doStuff(Shape s) {  
    s.erase();  
    // ...  
    s.draw();  
}
```

这个函数可与任何“几何形状”(Shape)对象做参数，所以完全独立于它要描绘(draw)和删除(erase)的任何特定类型的对象。如果我们在其他一些程序里使用 doStuff()函数：

```
Circle c = new Circle();  
Triangle t = new Triangle();  
Line l = new Line();  
doStuff(c);
```

```
doStuff(t);  
doStuff(l);
```

那么对 `doStuff()` 的调用会自动良好地工作, 无论对象的具体类型是什么。`doStuff()` 需要 `Shape` (形状) 对象句柄的函数一个 `Circle` (圆) 句柄传递给一个本来期待 `Shape` (形状) 句柄的函数。由于圆是一种几何形状, 所以 `doStuff()` 能正确地进行处理, 不会造成错误。

5.2.2 虚方法调用

在使用上溯造型的情况下, 子类对象可以当做父类对象, 对于重载或继承的方法, Java 运行时系统根据调用该方法的实例的类型来决定选择哪个方法调用。对子类的一个实例, 如果子类重载了父类的方法, 则运行时, 系统调用子类的方法, 如果子类继承了父类的方法(未重载), 则运行时, 系统调用父类的方法。

请看看 `doStuff()` 里的代码:

```
s.draw();
```

在 `doStuff()` 的代码里, 尽管没有做出任何特殊指示, 采取的操作也是完全正确和恰当的。我们知道, 为 `Circle` 调用 `draw()` 时执行的代码与为一个 `Square` 或 `Line` 调用 `draw()` 时执行的代码是不同的。在调用 `draw()` 时, 根据 `Shape` 句柄当时所引用对象的实际类型, 会相应地采取正确的操作。在这里因为当 Java 编译器为 `doStuff()` 编译代码时, 它并不知道自己要操作的准确类型是什么。但在运行时, 却会根据实际的类型调用正确的方法。对面向对象的程序设计语言来说, 这种情况就叫“多态性”(Polymorphism)。用以实现多形性的方法叫虚方法调用, 也叫“动态绑定”, 编译器和运行期系统会负责动态绑定的实现。

有些语言要求我们用一个特殊的关键字来表明虚方法调用(动态绑定), 如在 C++ 中, 这个关键字是 `virtual`。在 Java 中, 我们则完全不必添加关键字, 所有的非 `final` 的方法都会自动地进行动态绑定。

如果一个方法声明成 `final`, 它能防止覆盖该方法, 同时也告诉编译器不需要进行动态绑定。这样一来, 编译器就可为 `final` 方法调用生成效率更高的代码。

如果一个方法是 `static` 或 `private` 的, 它不能被子类所覆盖, 自然也是 `final` 的, 它也不存在虚方法调用的问题。

例 5-4 TestVirtualInvoke.java 虚方法调用。

```
class TestVirtualInvoke  
{  
    static void doStuff( Shape s ){  
        s.draw();  
    }  
  
    public static void main( String [] args ){  
        Circle c = new Circle();  
        Triangle t = new Triangle();
```

```
        Line l = new Line();
        doStuff(c);
        doStuff(t);
        doStuff(l);
    }
}

class Shape
{
    void draw(){ System.out.println("Shape Drawing"); }
}

class Circle extends Shape
{
    void draw(){ System.out.println("Draw Circle"); }
}

class Triangle extends Shape
{
    void draw(){ System.out.println("Draw Three Lines"); }
}

class Line extends Shape
{
    void draw(){ System.out.println("Draw Line"); }
}
```

程序的运行结果为：

```
Draw Circle
Draw Three Lines
Draw Line
```

用虚方法调用，可以实现运行时的多态，它体现了面向对象程序设计中的代码复用性。已经编译好的类库可以调用新定义的子类的方法而不必重新编译，而且如果增加几个子类的定义，只需分别用 **new** 生成不同子类的实例，会自动调用不同子类的相应方法。

5.2.3 动态类型确定

1 . instanceof 运算符

由于基本类型的变量可以引用子类型的对象，所以经常需要在运行时判断所引用的对象的实际类型，Java 中提供了一个运算符 `instanceof`，它的基本用法是：

变量 instanceof 类型

该表达式的结果是 boolean 值。

注意：如果变量与类型相同，或者是类型的子类型，则结果就是 `true`。

例 5-5 InstanceOf.java 使用 instanceof。

```
class InstanceOf
{
    public static void main(String[] args)
    {
        Object [] things = new Object[3];
        things[0] = new Integer(4);
        things[1] = new Double(3.14);
        things[2] = new Double(2.09);
        double s = 0;
        for( int i=0; i<things.length; i++ ){
            if( things[i] instanceof Integer )
                s += ((Integer)things[i]).intValue();
            else if( things[i] instanceof Double )
                s += ((Double)things[i]).doubleValue();
        }
        System.out.println("sum=" + s);
    }
}
```

在该例中，一个 `Object` 的数组的各个元素可以有不同的类型，用 `instanceof` 进行判定后，根据其不同的类型进行不同的处理。

2 . Class 类

除了用 `instanceof` 对运行时的类进行判断外，Java 中的对象可以通过 `getClass()` 方法来获得运行时的信息。`getClass()` 是 `java.lang.Object` 的方法，而 `Object` 是所有类的父类，所以任何对象都可以用 `getClass()` 方法，该方法的返回结果是一个 `Class` 对象。通过 `Class` 对象可以获得更为详细的关于对象的域、方法等方面的信息。

这种获得运行时的对象信息的方法又叫反射(reflection)。

例 5-6 RunTimeClassInfo.java 获得运行时的类信息。

```
import java.lang.reflect.*;

class RunTimeClassInfo
{
    public static void main(String[] args)
    {
        Object obj = new java.awt.Color(1,1,1);
        Class cls = obj.getClass();
        System.out.println( "类名:" + cls.getName() );
        Field [] fields = cls.getFields();
        for( int i=0; i<fields.length; i++ ){
            Field f = fields[i];
            System.out.println( "域:" + f.getName() + ":" + f );
        }
        Method [] methods = cls.getMethods();
        for( int i=0; i<methods.length; i++ ){
            Method m = methods[i];
            System.out.println( "方法: " + m.getName() + ":" + m );
        }
    }
}
```

例 5-6 中, 通过 `getClass()` 方法得到对象的运行时的类信息, 即一个 `Class` 类的对象, 它的 `getFields()` 及 `getMethods()` 方法能进一步获得其详细信息, 如图 5-2 所示。

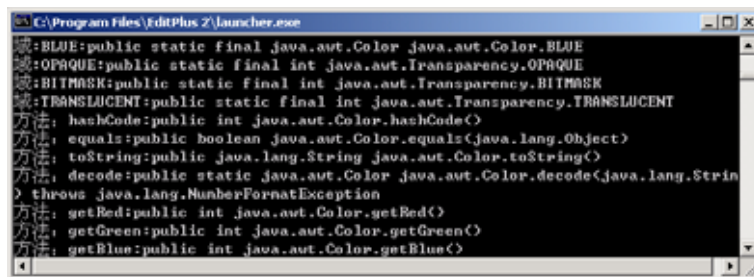


图 5-2 获得运行时的类信息

5.3 对象构造与初始化

5.3.1 调用本类或父类的构造方法

在第 4 章, 已经了解, 构造方法从语法的角度上看是可以重载的, 即有多个同名的构造方法, 但是不能继承, 因为继承意味与父类的构造方法要同名, 但显然子类的构造方法不可能与父类的构造方法同名。

但是构造方法不能继承, 这并不意味着不能调用别的构造方法。事实上, 在构造方法中, 一定要调用本类或父类的构造方法 (除非它是 `Object` 类, 因为 `Object` 类没有父类)。

具体做法可以选择以下三种之一。

(1) 使用 `this` 来调用本类的其他构造方法。

(2) 使用 `super` 来调用父类的构造方法。与普通方法中的 `super` 含义不同, `super` 指其直接父类的构造方法, 不能指间接父类的构造方法。

(3) 既不用 `this`, 也不用 `super`, 则编译器会自动加上 `super()`, 即调用父类的不带参数的构造函数。

注意: 调用 `this` 或 `super` 的构造方法的语句必须放在第一条语句, 并且最多只有一条这样的语句, 不能既调用 `this`, 又调用 `super`。

对于上面提到的第三种做法要引起读者的特别注意。

例如以下一段程序:

```
class A
{
    A(int a){}
}

class B extends A
{
    B(String s){} // 编译不能通过
}
```

其中, `B` 的一个构造 `B(String)` 的方法体中, 没有 `this` 及 `super`, 所有编译器会自动加上 `super()`, 也就是说它相当于:

```
B(String s) { super(); }
```

但由于其直接父类 `A` 没有一个不带参数的方法, 所有编译不能通过。解决这个问题的办法有多种, 如:

(1) 在 `B` 的构造方法中, 加入调用父类已有的构造方法, 如 `super(3)`;

(2) 在 `A` 中加入一个不带参数的构造方法, 如 `A()`;

(3) 去掉 `A` 中全部的构造方法, 则编译器会自动加入一个不带参数的构造方法, 称为默认构造方法。

例 5-7 ConstructCallThisAndSuper.java 在构造方法中使用 this 及 super。

```
class ConstructCallThisAndSuper
{
    public static void main(String[] args)
    {
        Person p = new Graduate();
    }
}

class Person
{
    String name;
    int age;
    Person(){}
    Person( String name, int age ){
        this.name=name; this.age=age;
        System.out.println("In Person(String,int)");
    }
}

class Student extends Person
{
    String school;
    Student(){
        this( null, 0, null );
        System.out.println("In Student()");
    }
    Student( String name, int age, String school ){
        super( name, age );
        this.school = school;
        System.out.println("In Student(String,int,String)");
    }
}

class Graduate extends Student
{
    Graduate(){
        System.out.println("In Graduate()");
    }
}
```

```
}  
}
```

在该例中, 构造一个 Graduate 对象时, 首先调用编译器自动加入的 super(); 所以进入到 Student(); 而 Student()调用 Student(String, int, String); 而 Student(String, int, String)再调用 Person(String, int); Person(String, int)中会调用自动加入的 super(), 即 Object()。以下各调用完成后, 才依次返回, 显示的结果如图5-3 所示。

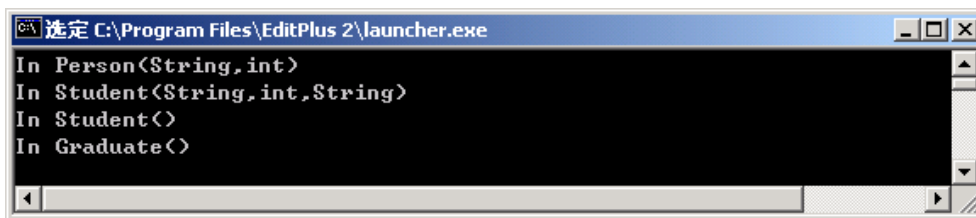


图 5-3 在构造方法中使用 this 及 super

在构造方法中调用 this 及 super 或自动加入的 super, 最终保证了任何一个构造方法都要调用父类的构造方法, 而父类的构造方法又会再调用其父类的构造方法, 直到最顶层的 Object 类。这是符合面向对象的概念的, 因为必须令所有父类的构造方法都得到调用, 否则整个对象的构建就可能不正确。

5.3.2 构造方法的执行过程

对于一个复杂的对象, 构造方法的执行过程遵照下面的步骤。

(1) 调用本类的构造或父类的构造方法。这个步骤会不断重复下去, 直到抵达最深一层的派生类。

(2) 按声明顺序调用成员初始化模块。即, 执行域的初始化赋值。

(3) 执行构造方法中的各语句。

构建器调用的顺序是非常重要的。首先调用父类的构造方法, 保证其基础类的成员得到正确的初始化并执行相关的语句, 然后对本对象的域进行初始化, 最后才是执行构造方法中的相关语句。

例 5-8 ConstructSequence.java 构造方法的执行过程。

```
class ConstructSequence  
{  
    public static void main(String[] args){  
        Person p = new Student("李明", 18, "北大");  
    }  
}  
  
class Person
```



```
{
    String name="未命名";
    int age=-1;
    Person( String name, int age ){
        System.out.println( "开始构造 Person(),此时 this.name="
            +this.name+ ",this.age="+ this.age );
        this.name=name; this.age=age;
        System.out.println( "Person()构造完成,此时 this.name="
            +this.name +",this.age="+ this.age );
    }
}

class Student extends Person
{
    String school="未定学校";
    Student( String name, int age, String school ){
        super( name, age );
        System.out.println( "开始构造 Student(),此时 this.name="+ this.
            name+ ",this.age="+this.age+ ",this.school="+ this.school );
        this.school = school;
        System.out.println( "Student()构造完成,此时 this.name=" + this.
            name+ ",this.age="+this.age+ ",this.school="+ this.school );
    }
}
```

其运行结果如图 5-4 所示,从本例中可以清楚地看到,先进行父类的构造,再进行本类的成员赋值,和最后执行构造方法中的语句的三大步骤。

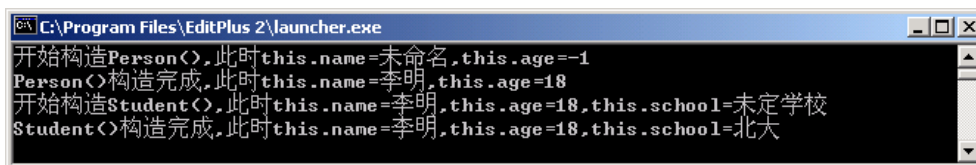


图 5-4 构造方法的执行过程

5.3.3 构建方法内部调用的方法的多态性

在构造子类的一个对象时,子类构造方法会调用父类的构造方法,而如果父类的构造方法中如果调用了对象的其他方法,如果所调用的方法被子类所覆盖的话,它可能实际上调用的是子类的方法,这是由动态绑定(虚方法调用)所决定的。从语法上来说,这是正

确的，但有时却会造成事实上的不合理，所以在构造方法中调用其他方法时要谨慎。

例 5-9 ConstructInvokeMetamorph.java 构建方法内部调用的方法的多态性。

```
class ConstructInvokeMetamorph
{
    public static void main(String[] args){
        Person p = new Student("李明", 18, "北大");
    }
}

class Person
{
    String name="未命名";
    int age=-1;
    Person( String name, int age ){
        this.name=name; this.age=age;
        sayHello();
    }
    void sayHello(){
        System.out.println( "我是一个人，我名叫: " + name + ",年龄为: "
            + age );
    }
}

class Student extends Person
{
    String school="未定学校";
    Student( String name, int age, String school ){
        super( name, age );
        this.school = school;
    }
    void sayHello(){
        System.out.println( "我是学生，我名叫: " + name + ",年龄为: "
            + age + ",学校在: " + school );
    }
}
```

运行结果如图5-5 所示。



图 5-5 构建方法内部调用的方法的多态性

在上面的例子中，在构造方法中调用一个动态绑定的方法（虚方法调用）sayHello()，这时，会使用那个方法被覆盖的定义，而这时对象尚未完全构造好，因为这时 school 尚未赋值。

注意：这样的错误会很轻易地被人忽略，而且要花很长的时间才能找出这种错误码，应对这个情况提高警惕。

因此，设计构建器时一个特别有效的规则是用尽可能简单的方法使对象进入就绪状态，如果可能，避免调用任何方法。在构建器内惟一能够安全调用的是在基础类中具有 final 属性的那些方法（也适用于 private、static 方法，它们自动具有 final 属性）。这些方法不能被覆盖，所以不会出现上述潜在的问题。

5.4 对象清除与垃圾回收

Java 中的对象使用 new 进行创建，而由系统自动进行对象的清除，清除无用对象的过程，称为垃圾回收（garbage collection）。本节介绍对象清除的相关概念。

5.4.1 对象的自动清除

Java 中与 C++ 等语言相比，其最大的特色之一就是：无用的对象由系统自动进行清除和内存回收的过程，编程者可以不关心如何回收及何时回收对象。这样大大减轻了编程者的负担，而且大大降低了由于对象提前回收或忘记回收带来的潜在错误。

对象的回收是由 Java 虚拟机的垃圾回收线程来完成的。该线程对无用对象在适当的时机进行回收。那么，Java 是如何知道一个对象是无用的呢？这里的关键是，系统中的任何对象都有一个引用计数，一个对象被引用 1 次，则引用计数为 1，被引用 2 次，则引用计数为 2，依次类推，当一个对象的引用计数被减到 0 时，说明该对象可以回收。如下面一段程序：

```
String method(){
    String a,b;
    a = new String("hello world");
    b = new String("game over");
    System.out.println(a+b+"ok");
    a = null;
    a = b;
    return a;
}
```

在程序中创建了两个对象实体，字符串"hello world"与"game over"，一开始分别被 *a* 和 *b* 所引用，后来 *a*=null；执行后，则字符串"hello world"对象不再被引用，其引用计数减到 0，可以被回收；执行 *a*=*b* 后，则字符串"game over"对象的引用计数增加到 2，而当方法执行完成后，*a*,*b* 两个变量都消失，后一对象的引用计数也变为 0。如果方法的结果赋值给其他变量，则字符串"game over"对象的引用计数增加到 1；如果方法的结果不赋值给其他变量，则这两个字符串都不再被引用，可以被回收。

5.4.2 System.gc()方法

System 类有一个 static 方法，叫 `System.gc()`，它可以要求系统进行垃圾回收。但是它仅仅是“建议”系统进行垃圾回收，而没有办法强制系统进行垃圾回收，也无法控制怎样进行回收，以及何时进行回收。

5.4.3 finalize()方法

Java 中，对象的回收是由系统进行的，但有一些任务需要在回收时进行。例如，清理一些非内存资源，关闭打开的文件，等等。这可以通过覆盖 `Object` 的 `finalize()` 方法来实现。因为系统在回收时会自动调用对象的 `finalize()` 方法。

`finalize()` 方法的形式是：

```
protected void finalize()throws Throwable
```

一般说来，子类的 `finalize()` 方法中应该调用父类的 `finalize()` 方法，以保证父类的清理工作能正常进行。

许多情况下，清除并不是个问题，只需让垃圾收集器去处理。但是如果使用 `finalize()` 方法，就必须特别谨慎，并做周全的考虑。由于不可能确切知道何时会开始垃圾回收，也不能决定垃圾收集器回收对象的顺序，所以除内存的回收以外，其他任何资源都最好不要依赖垃圾收集器进行回收，应该制作自己的清除方法，而且不要依赖 `finalize()`。

例 5-10 TestCleanUp.java 清除。

```
class TestCleanUp{
    public static void main(String[] args) {
        PolyLine x = new PolyLine(47);
        try {
            // Code and exception handling...
        } finally {
            x.cleanup();
        }
    }
}

class Shape {
    Shape(int i) {
```

```
        System.out.println("Shape constructor");
    }
    void cleanup() {
        System.out.println("Shape cleanup");
    }
}

class Line extends Shape {
    private int start, end;
    Line(int start, int end) {
        super(start);
        this.start = start;
        this.end = end;
        System.out.println("Drawing a Line: " +
            start + ", " + end);
    }
    void cleanup() {
        System.out.println("Erasing a Line: " +
            start + ", " + end);
        super.cleanup();
    }
}

class PolyLine extends Shape {
    private Line[] lines = new Line[10];
    PolyLine(int i) {
        super(i + 1);
        for(int j = 0; j < 10; j++)
            lines[j] = new Line(j, j*j);
        System.out.println("PolyLine constructor");
    }
    void cleanup() {
        System.out.println("PolyLine.cleanup()");
        for(int i = 0; i < lines.length; i++)
            lines[i].cleanup();
        super.cleanup();
    }
}
```

本例中，有 Shape, Line, PolyLine 等几个类，每个类都有一个方法 cleanup() 来负责清理工作，子类的 cleanup() 还会调用父类的 cleanup() 方法，一个类还有对其中所包含的对象调用 cleanup() 方法。自己的清除方法中，必须注意对基础类及成员对象清除方法的调用顺序，通常首先完成本类有关的工作，然后调用父类清除方法。

在 main() 中，可看到两个新关键字：try 和 finally (第 6 章会详细讲解)。在这里，finally 从句表明“无论会发生什么事情，总是为 x 调用 cleanup()”。事实上，将清理工作放在 finally 从句是一种常见的做法。

5.5 内部类与匿名类

本节介绍内部类 (Inner Class) 及匿名类 (Anonymous Class)。简单地说，内部类是定义在其他类中的类，内部类的主要作用是将逻辑上相关联的类放到一起；而匿名类是一种特殊的内部类，它没有类名，在定义类的同时，就生成该对象的一个实例，由于不会在其他地方用到该类，所以不用取名字。

5.5.1 内部类

在 Java 中，可将一个类定义置入另一个类定义中，这就叫做“内部类”。利用内部类，可对那些逻辑上相互联系的类进行分组，并可控制一个类在另一个类里的“可见性”。

1. 内部类的定义和使用

定义内部类是简单的，将类的定义置入一个用于封装它的类内部即可。

注意：内部类不能与外部类同名 (那样的话，编译器无法区分内部类与外部类)，如果内部类还有内部类，内部类的内部类不能与它的任何一层外部类同名。

在封装它的类的内部使用内部类，与普通类的使用方式相同，在其他地方使用内部类时，类名前要冠以其外部类的名字才能使用，在用 new 创建内部类时，也要在 new 前面冠以对象变量。如例 5-11 所示。

例 5-11 TestInnerUse.java 使用内部类。

```
class TestInnerUse{
    public static void main( String[] args ){
        Parcel p = new Parcel();
        Parcel.Contents c = p.new Contents(33);
        Parcel.Destination d = p.new Destination( "Hawii" );
        p.setValue( c, d );
        p.ship();
    }
}
```

```
class Parcel {
    private Contents c;
    private Destination d;
    class Contents {
        private int i;
        Contents( int i ){ this.i = i; }
        int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) {label = whereTo;}
        String readLabel() { return label; }
    }
    void setValue( Contents c, Destination d ){
        this.c =c; this.d = d;
    }
    void ship(){
        System.out.println( "运输"+ c.value() +"到"+ d.readLabel() );
    }
    public void testShip() {
        c = new Contents(22);
        d = new Destination("Tanzania");
        ship();
    }
}
```

该例中，在 `Parcel` 类中定义内部类 `Contents` 及 `Destination`，在 `Parcel` 类使用 `Contents` 及 `Destination` 类与其他类没有区别（见 `ship()` 方法及 `testShip()` 方法）。而在其他类中（见 `main()` 方法），在用类名和 `new` 运算符前分别要冠以外部类的名字及外部对象名。

2. 在内部类中使用外部类的成员

内部类与类中的域、方法一样是外部类的成员，所以在内部类中可以直接访问外部类的其他域及方法，即使它们是 `private` 的。这也是使用内部类的一个好处。

如果内部类中与外部类有同名的域或方法，可以使用冠以外部类名 `this` 来访问外部类中的同名成员。

例 5-12 TestInnerThis.java 在内部类中使用 this。

```
public class TestInnerThis
{
```

```
        public static void main(String args[]){
            A a = new A();
            A.B b = a.new B();
            b.mb(333);
        }
    }

    class A
    {
        private int s = 111;
        public class B {
            private int s = 222;
            public void mb(int s) {
                System.out.println(s);           // 局部变量 s
                System.out.println(this.s);       // 内部类对象的属性 s
                System.out.println(A.this.s);     // 外层类对象属性 s
            }
        }
    }
}
```

在该例中，分别访问了局部变量、内部类对象的属性、外层类对象属性。程序的结果为 333，222，111。

3. 内部类的修饰符

内部类与类中的域、方法一样是外部类的成员，它的前面也可以访问控制符及其他修饰符。内部类可用的修饰符比外部类的修饰符更多。例如，外部类不能使用 `protected`，`private`，`static` 等修饰，而内部类可以。

访问控制符包括 `public`，`protected`，默认及 `private`，其含义与域前面的访问控制符一样。

内部类前面用 `final` 修饰，表明该内部类不能被继承；内部类前面用 `abstract` 修饰，表明该内部类不能被实例化。

内部类前面用 `static` 修饰，则表明该内部类实际是一种外部类，因为它们的存在不依赖于外部类的一个具体实例。`Static` 内部类与普遍的内部类有较大的不同：由于 `static` 的内部类使用在 `static` 环境，`static` 环境在使用时要遵循以下规则：

- (1) 实例化 `static` 内部类时，在 `new` 前面不需要用对象变量；
- (2) `static` 内部类中不能访问其外部类的非 `static` 域及方法，即只能访问 `static` 成员。
- (3) `static` 方法中不能访问非 `static` 的域及方法，也不能不带前缀地 `new` 一个非 `static` 的内部类。

根据以上规则，不难看出在例 5-13 中的被注释起来的错误行。

例 5-13 TestInnerStatic.java 静态内部类。

```
class TestInnerStatic
{
    public static void main(String[] args)
    {
        A.B a_b = new A().new B(); // ok
        A a = new A();
        A.B ab = a.new B();

        Outer.Inner oi = new Outer.Inner();
        //Outer.Inner oi2 = Outer.new Inner(); //!!!error
        //Outer.Inner oi3 = new Outer().new Inner(); //!!! error
    }
}

class A
{
    private int x;
    void m(){
        new B();
    }
    static void sm(){
        //new B(); // error!!!!
    }
    class B
    {
        B(){ x=5; }
    }
}

class Outer
{
    static class Inner
    {
    }
}
```

5.5.2 方法中的内部类及匿名类

1. 方法中的内部类

在一个方法中，也可以定义类，这种类称为方法中的内部类。

例 5-14 TestInnerInMethod.java 方法中的内部类。

```
class TestInnerInMethod
{
    public static void main(String[] args)
    {
        Object obj = new Outer().makeTheInner(47);
        System.out.println("Hello World!" + obj.toString() );
    }
}

class Outer
{
    private int size = 5;
    public Object makeTheInner( int localVar )
    {
        final int finalLocalVar = 99;
        class Inner {
            public String toString() {
                return ( " InnerSize: " + size +
                    // " localVar: " + localVar +    // Error!
                    " finalLocalVar: " + finalLocalVar
                );
            }
        }
        return new Inner();
    }
}
```

在方法中定义内部类时，要注意以下几点。

-
- 注意：(1) 同局部变量一样，方法中的内部类前面不能用 `public`，`private`，`protected` 修饰，也不能用 `static` 修饰，但可以被 `final` 或 `abstract` 修饰。
- (2) 方法中的内部类，可以访问其外部类的成员；若是 `static` 方法中的内部类，可以访问外部类的 `static` 成员。
-

-
- (3) 方法中的内部类中,不能访问该方法的局部变量,除非是 final 的局部变量。
 - (4) 方法中定义的类,在其他地方使用时,没有类的名字,正像上面的例子中一样,只能用其父类(例中是用 Object)来引用这样的变量。
-

2. 匿名类

在类及其方法中,可以定义一种匿名类,匿名类有以下几个特点。

(1) 这种类不取名字,而直接用其父类的名字或者它所实现的接口的名字。

(2) 类的定义与创建该类的一个实例同时进行,即类的定义前面有一个 new。不使用关键词 class,同时带上()表示创建对象。也就是说,匿名类的定义方法是:

```
new 类名或接口名 () { ..... }
```

(3) 类名前面不能有修饰符。

(4) 类中不能定义构造方法,因为它没有名字。也正是这个原因,在构造对象时,也不能带参数,因为默认构造方法不能带参数。

上面的例子用匿名类可以改写成如例 5-15 的形式。

例 5-15 TestInnerAnonymous.java 匿名类。

```
class TestInnerAnonymous
{
    public static void main(String[] args)
    {
        Object obj = new Outer().makeTheInner(47);
        System.out.println("Hello World!" + obj.toString() );
    }
}

class Outer
{
    private int size = 5;
    public Object makeTheInner( int localVar )
    {
        final int finalLocalVar = 99;
        return new Object() {
            public String toString() {
                return ( " InnerSize: " + size +
                    " finalLocalVar: " + finalLocalVar
                );
            }
        };
    }
}
```

```
}
```

通过这个例子也可以看出，匿名类可以简化程序的书写。匿名类主要使用在那些需要扩展某个类或实现某个接口做参数的地方，在后面的章节中讲解图形化界面与事件监听时，会大量地用到匿名类。

习题

1. 什么是多态？面向对象程序设计为什么要引入多态的特性？使用多态有什么优点？
2. 虚方法调用有什么重要作用？具有什么修饰符的方法不能够使用虚方法调用？
3. 用默认构建器（空自变量列表）创建两个类：**A** 和 **B**，令它们自己声明自己。从 **A** 继承一个名为 **C** 的新类，并在 **C** 内创建一个成员 **B**。不要为 **C** 创建一个构建器。创建类 **C** 的一个对象，并观察结果。
4. 创建 **Rodent**（啮齿动物）：**Mouse**（老鼠），**Gerbil**（鼯鼠），**Hamster**（大颊鼠）等的一个继承分级结构。在基础类中，提供适用于所有 **Rodent** 的方法，并在派生类中覆盖它们，从而根据不同类型的 **Rodent** 采取不同的行动。创建一个 **Rodent** 数组，在其中填充不同类型的 **Rodent**，然后调用自己的基础类方法，看看会有什么情况发生。
5. Java 中怎样清除对象？能否控制 Java 中垃圾回收的时间？
6. 内部类与外部类的使用有何不同？
7. 怎样使用匿名类的对象？
8. 方法中定义的内部类是否可以存取方法中的局部变量？

第6章 异常处理

6.1 异常处理

6.1.1 异常的概念

异常(Exception)又称为例外、差错、违例等,是特殊的运行错误对象,对应着 Java 语言特定的运行错误处理机制。由于 Java 程序是在网络环境中运行的,安全成为需要首先考虑的重要因素之一。为了能够及时有效地处理程序中的运行错误,Java 中引入了异常和异常类。作为面向对象的语言,异常与其他语言要素一样,是面向对象规范的一部分。

1. Java 中的异常处理

捕获错误最理想的时间是在编译期间,并且最好在试图运行程序以前。然而,并非所有错误都能在编译期间侦测到。有些问题必须在运行期间解决,例外是在程序运行过程中发生的异常事件,比如除 0 溢出、数组越界、文件未找到等,这些事件的发生将阻止程序的正常运行。为了加强程序的健壮性,程序设计时,必须考虑到可能发生的异常事件并做出相应的处理。

在一些传统的语言(如 C 语言中),通过使用 if 语句来判断是否出现了例外,同时,调用函数通过被调用函数的返回值感知在被调用函数中产生的例外事件并进行处理。全程变量 ErrNo 常常用来反映一个异常事件的类型。但是,这种错误处理机制会导致不少问题,如:

- (1) 正常处理程序与异常处理程序的代码同样地处理,程序的可读性大幅度降低;
- (2) 每次调用一个方法时都进行全面、细致的错误检查,程序的可维护性大大降低;
- (3) 由谁来处理错误的职责不清,以致于造成大量的潜伏的问题,等等。

为了解决这些问题,Java 通过面向对象的方法来处理异常。

在一个方法的运行过程中,如果发生了异常,则这个方法生成代表该异常的一个对象,并把它交给运行时系统,运行时系统寻找相应的代码来处理这一异常。我们把生成异常对象并把它提交给运行时系统的过程称为抛出(throw)异常。运行时系统在方法的调用栈中查找,从生成异常的方法开始进行回溯,直到找到包含相应异常处理的方法为止,这一个过程称为捕获(catch)一个异常。

Java 的这种机制的另一项好处就是能够简化错误控制代码。编程者不用检查一个特定的错误,然后在程序的多处地方对其进行控制。此外,也不需要方法调用的时候检查错误(因为保证有人能捕获这里的错误)。这样可有效减少代码量,并将那些用于描述具体操作的代码与专门纠正错误的代码分隔开。一般情况下,用于读取、写入和调试的代码会变得更富有条理。

由于异常控制是由 Java 编译器进行实施的,对于编程者而言,使用这种控制却是相当简单的。

2 . Throwable 与 Exception

Java 中定义了很多异常类,每个异常类都代表了一种运行错误,类中包含了该运行错误的信息和处理错误的方法等内容。Java 的异常类都是 `java.lang.Throwable` 的子类。它派生了两个子类: `Error` (错误) 和 `Exception` (违例)。其中 `Error` 类,由系统保留;而 `Exception` 类则供应用程序使用。其中:

Error: JVM 系统内部错误、资源耗尽等严重情况,由系统保留;

Exception: 其他因编程错误或偶然的外在因素导致的一般性问题,例如:对负数开平方根;空指针访问;试图读取不存在的文件网络连接中断。

一般所说的异常,都是指 `Exception` 及其子类。因为,应用程序不处理 `Error` 类。

同其他的类一样, `Exception` 类有自己的方法和属性。它的构造函数有两个:

☞ `public Exception();`

☞ `public Exception(String s);`

第二个构造函数可以接受字符串参数传入的信息,该信息通常是对该例外所对应的错误的描述。

`Exception` 类从父类 `Throwable` 那里还继承了若干方法,其中常用的有如下两种。

(1) `public String toString ();` `toString()` 方法返回描述当前 `Exception` 类信息的字符串。

(2) `public void printStackTrace ();` `printStackTrace()` 方法没有返回值,它的功能是完成一个打印操作,在当前的标准输出(一般就是屏幕显示)上打印输出当前例外对象的堆栈使用轨迹,也即程序先后调用并执行了哪些对象或类的哪些方法,使得运行过程中产生了这个例外对象。

3 . 系统定义的异常

JDK 中已经定义了若干 `Exception` 的子类。其中分为 `RuntimeException` 及非 `RuntimeException`。如图 6-1 所示。

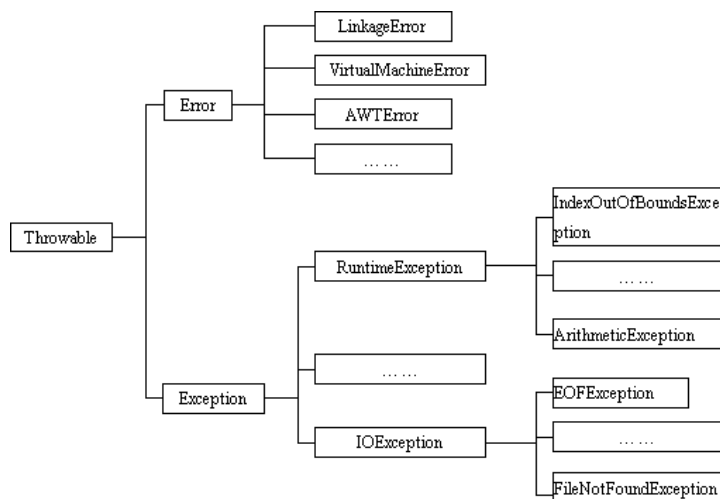


图 6-1 异常及其分类

`RuntimeException` 表明是一种设计或实现时的问题，例如 `IndexOutOfBoundsException`(下标超界)、`ArithmeticException`(算术运算异常，如整数除法中，除数为 0)。这种异常是可以通过适当的编程进行避免的，如下标就不应该超界。由于这类异常应该由程序避免，所以从语法的角度看，Java 不要求捕获这类异常。

除 `RuntimeException` 以外的异常，经常是在程序运行过程中由环境原因造成的异常，如网络地址不能打开，文件未找到、读写异常等。这类异常必须由程序进行处理，否则编译不能通过。

4. 用户自定义的异常

由用户自定义的异常，是由 `Exception` 或其子类所派生出来的类，用于处理与具体应用相关的异常。下面会详细讲解。

6.1.2 捕获和处理异常

Java 中的异常处理机制可以概括成以下几个步骤。

(1) Java 程序的执行过程中如出现异常，会自动生成一个异常类对象，该异常对象将被提交给 Java 运行时系统，这个过程称为抛出异常。抛出异常也可以由程序来强制进行。

(2) 当 Java 在运行时系统接收到异常对象，会寻找能处理这一异常的代码并把当前异常对象交给其处理，这一过程称为捕获(catch)异常。

(3) 如果 Java 运行时系统找不到可以捕获异常的方法，则运行时系统将终止，相应的 Java 程序也将退出。

1. 抛出异常

Java 程序在运行时如果引发了一个可识别的错误，就会产生一个与该错误相对应的异常类的对象，这个过程被称为异常的抛出。根据异常类的不同，抛出异常的方法也不同。

(1) 系统自动抛出的异常。所有的系统定义的运行异常都可以由系统自动抛出。

(2) 语句抛出的异常。用户程序自定义的异常不可能依靠系统自动抛出，而必须借助于 `throw` 语句来定义何种情况算是产生了此种异常对应的错误，并应该抛出这个异常类的新对象。用 `throw` 语句抛出例外对象的语法格式为：

`throw` 异常对象；

使用 `throw` 语句抛出例外时应注意如下两个问题。

注意：(1) 一般这种抛出异常的语句应该被定义为在满足一定条件时执行，例如把 `throw` 语句放在 `if` 语句的 `if` 分支中，只有当一定条件得到满足，即用户定义的逻辑错误发生时才执行。

(2) 含有 `throw` 语句的方法，或者调用其他类的有异常抛出的方法时，必须在方法头定义中增加 `throws` 异常类名列表中，如下所示：

修饰符 返回类型 方法名(参数列表) throws 异常类名列表

```
{  
      
}
```

这样做主要是为了通知所有欲调用此方法的方法。由于该方法包含 `throw` 语句，所以要准备接受和处理它在运行过程中可能会抛出的异常。如果方法中的 `throw` 语句不止一个，方法头的异常类名列表也不止一个，应该包含所有可能产生的异常。

2. 捕获异常

当一个异常被抛出时，应该有专门的语句来接收这个被抛出的异常对象，这个过程被称为捕获异常或捕捉异常。当一个异常类的对象被捕捉或接收后，用户程序就会发生流程的跳转，系统中止当前的流程而跳转至专门的异常处理语句块，或直接跳出当前程序和 Java 虚拟机回到操作系统。

在 Java 程序里，异常对象是依靠以 `catch` 语句为标志的异常处理语句块来捕捉和处理的。异常处理语句块又称为 `catch` 语句块，其格式如下：

```
try{
    语句组
} catch(异常类名 异常形式参数名){
    异常处理语句组;
} catch(异常类名 异常形式参数名){
    异常处理语句组;
} catch(异常类名 异常形式参数名){
    异常处理语句组;
} finally{
    异常处理语句组;
}
```

其中，`catch` 语句可以有一个或多个，而且至少要有一个 `catch` 语句或 `finally` 语句。

Java 语言还规定，每个 `catch` 语句块都应该与一个 `try` 语句块相对应，这个 `try` 语句块用来启动 Java 的异常处理机制，可能抛出异常的语句，包括 `throw` 语句、调用可能抛出异常方法的方法调用语句，都应该包含在这个 `try` 语句块中。

`catch` 语句块应该紧跟在 `try` 语句块的后面。当 `try` 语句块中的某条语句在执行时产生了一个异常时，此时被启动的异常处理机制会自动捕捉到它，然后流程自动跳过产生例外的语句后面的所有尚未执行语句，而转至 `try` 块后面的 `catch` 语句块，执行 `catch` 块中的语句。

3. 多异常的处理

`catch` 块紧跟在 `try` 块的后面，用来接收 `try` 块可能产生的异常，一个 `catch` 语句块通常会用同种方式来处理它所接收到的所有异常，但是实际上一个 `try` 块可能产生多种不同的异常，如果希望能采取不同的方法来处理这些例外，就需要使用多异常处理机制。

多异常处理是通过在一个 `try` 块后面定义若干个 `catch` 块来实现的，每个 `catch` 块用来接收和处理一种特定的异常对象。

当 `try` 块抛出一个异常时，程序的流程首先转向第一个 `catch` 块，并审查当前异常对象可否为这个 `catch` 块所接收。能接收是指异常对象与 `catch` 的参数类型相匹配，即以下三种情况之一。

- ✎ 异常对象与参数属于相同的例外类。
- ✎ 异常对象属于参数例外类的子类。
- ✎ 异常对象实现了参数所定义的接口。

如果 `try` 块产生的异常对象被第一个 `catch` 块所接收, 则程序的流程将直接跳转到这个 `catch` 语句块中, 语句块执行完毕后就退出当前方法, `try` 块中尚未执行的语句和其他的 `catch` 块将被忽略。如果 `try` 块产生的异常对象与第一个 `catch` 块不匹配, 系统将自动转到第二个 `catch` 块进行匹配。如果第二个仍不匹配, 就转向第三个, ……直到找到一个可以接收该异常对象的 `catch` 块, 即完成流程的跳转。

如果所有的 `catch` 块都不能与当前的异常对象匹配, 则说明当前方法不能处理这个异常对象, 程序流程将返回到调用该方法的上层方法。如果这个上层方法中定义了与所产生的异常对象相匹配的 `catch` 块, 流程就跳转到这个 `catch` 块中, 否则继续回溯更上层的方法。如果所有的方法中都找不到合适的 `catch` 块, 则由 Java 运行系统来处理这个异常对象。此时通常会中止程序的执行, 退出虚拟机返回操作系统, 并在标准输出上打印相关的异常信息。

在另一种完全相反的情况下, 假设 `try` 块中所有语句的执行都没有引发异常, 则所有的 `catch` 块都会被忽略而不予执行。

在设计 `catch` 块处理不同的异常时, 一般应注意如下问题。

注意: (1) `catch` 块中的语句应根据异常的不同而执行不同的操作, 比较通用的操作是打印异常和错误的相关信息, 包括异常名称、产生异常的方法名等。

(2) 由于异常对象与 `catch` 块的匹配是按照 `catch` 块的先后排列顺序进行的, 所以在处理多异常时应注意认真设计各 `catch` 块的排列顺序。一般地, 将处理较具体和较常见的异常的 `catch` 块应放在前面, 而可以与多种异常相匹配的 `catch` 块应放在较后的位置。若将子类异常的 `catch()` 句放在父类的后面, 则编译不能通过。

4. finally 语句

捕获异常时, 还可以使用 `finally` 语句。`finally` 语句为异常处理提供一个统一的出口, 使得在控制流转到程序的其他部分以前(即使有 `return`, `break` 等语句), 能够对程序的状态作统一的管理。不论在 `try` 代码块中是否发生了异常事件, `finally` 块中的语句都会被执行。

`finally` 语句是任选的, `try` 后至少要有一个 `catch` 或一个 `finally`。

`finally` 语句经常用于对一些资源做清理工作, 如关闭打开的文件。

5. 在覆盖的方法中声明异常

在子类中, 如果要覆盖父类的一个方法, 若父类中的方法声明了 `throws` 异常, 则子类的方法也可以 `throws` 异常。

注意: 子类方法中不能抛出比父类更多种类的异常, 也不能抛出比父类更一般的异常; 换句话说, 子类方法抛出的异常只能是父类方法抛出的异常的同类或子类。

例如, 下面 B1 是正确的, B2 则不能通过编译。

```
import java.io.*;

class A {
    public void methodA() throws IOException {
        //.....
    }
}

class B1 extends A {
    public void methodA() throws FileNotFoundException {
        //.....
    }
}

class B2 extends A {
    public void methodA() throws Exception { // Error!
        // .....
    }
}
```

6.1.3 应用举例

例 6-1 ExceptionIndexOutOf.java 使用 try。

```
public class ExceptionIndexOutOf{
    public static void main(String[] args)    {
        String friends[]={ "lisa","bily","kessy"};
        try {
            for(int i=0;i<5;i++) {
                System.out.println(friends[i]);
            }
        } catch (java.lang.ArrayIndexOutOfBoundsException e) {
            System.out.println("index err");
        }
        System.out.println("\nthis is the end");
    }
}
```

程序的运行结果如图6-2 所示。

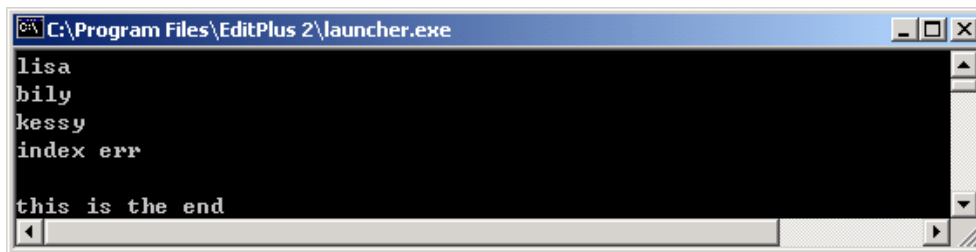


图 6-2 使用 try 的运行结果

例 6-2 ExceptionSimple.java 使用 try{}catch...finally 语句。

```
class ExceptionSimple
{
    int a = 10;
    public static void main(String[] args)
    {
        int a = 0;
        try
        {
            a= Integer.parseInt( "2" );
            a /= 0;
            // 注意：整数除以 0,会产生异常,但 0.0/0=NaN FPN/0=正无穷,-FPN/0=负无穷,
            catch(ArithmeticException ea )
            { System.out.println("ea:" +ea); }
            catch(NumberFormatException en )
            { System.out.println("en:" +en); }
            catch(NullPointerException ep )
            { System.out.println("ep:" +ep); }
            catch(IndexOutOfBoundsException eb )
            { System.out.println("eb:" +eb); }
            catch(Exception e) { System.out.println("e:" + e); }
            //先 catch 子类 Exception,后 catch 父类
            finally{ System.out.println("finally executed.");}
            System.out.println("Hello World!" +a );
        }
    }
}
```

例 6-3 ExceptionForIO.java 在该例中 IO 异常必须被捕获,否则编译不能通过。这是因为 read()等方法 throws 了 IOException。

```
import java.io.*;
```

```
public class ExceptionForIO{
    public static void main(String[] args){
        try{
            FileInputStream in=new FileInputStream("myfile.txt");
            int b;
            b = in.read();
            while(b!= -1) {
                System.out.print((char)b);
                b = in.read();
            }
            in.close();
        }catch (IOException e) {
            System.out.println(e);
        }finally {
            System.out.println(" It'always come here!");
        }
    }
}
```

例 6-4 ExceptionTrowsToOther.java 在子程序中未处理的异常通过 **throws** 语句进行声明，将异常的处理交给调用者进行捕获和处理。

```
import java.io.*;

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

        try{
            System.out.println("====Before====");
            readFile();
            System.out.println("====After====");
        }catch(IOException e){ System.out.println(e); }

    }

    public static void readFile()throws IOException {
        FileInputStream in=new FileInputStream("myfile.txt");
        int b;
        b = in.read();
        while(b!= -1) {
            System.out.print((char)b);
        }
    }
}
```

```
        b = in.read();
    }
    in.close();
}
}
```

6.2 创建用户自定义异常类

系统定义的异常主要用来处理系统可以预见的较常见的运行错误,对于某个应用所特有的运行错误,则需要编程人员根据程序的特殊逻辑在用户程序里自己创建用户自定义的异常类和异常对象。这种用户自定义异常主要用来处理用户程序中特定的逻辑运行错误。

用户自定义异常用来处理程序中可能产生的逻辑错误,使得这种错误能够被系统及时识别并处理,而不致扩散产生更大的影响,从而使用户程序更为强健,有更好的容错性能,并使整个系统更加安全稳定。

创建用户自定义异常时,一般需要完成如下的工作。

(1) 声明一个新的异常类,使之以 `Exception` 类或其他某个已经存在的系统异常类或用户异常类为父类。

(2) 为新的异常类定义属性和方法,或重载父类的属性和方法,使这些属性和方法能够体现该类所对应的错误的信息。

只有定义了异常类,系统才能识别特定的运行错误,才能及时地控制和处理运行错误,所以定义足够多的异常类是构建一个稳定完善的应用系统的重要基础之一。

例 6-5 `MyException.java` 用户定义的异常类。

```
class MyException extends Exception {
    private int idnumber;
    public MyException(String message, int id) {
        super(message);
        this.idnumber = id;
    }
    public int getId() {
        return idnumber;
    }
}

public class Exce6_6{
    public void regist(int num) throws MyException {
        if(num < 0) {
```

```
        System.out.println("登记号码" + num );
        throw new MyException("号码为负值, 不合理",3);
    }
}

public void manager() {
    try {
        regist(-100);
    } catch (MyException e) {
        System.out.println("登记失败, 出错种类" + e.getId());
    }
    System.out.println("本次登记操作结束");
}

public static void main(String args[]){
    Exce6_6 t = new Exce6_6();
    t.manager();
}
}
```

如图6-3所示, 本程序中, 定义了一个异常类 `MyException`, 用于描述数据取值范围错误信息。

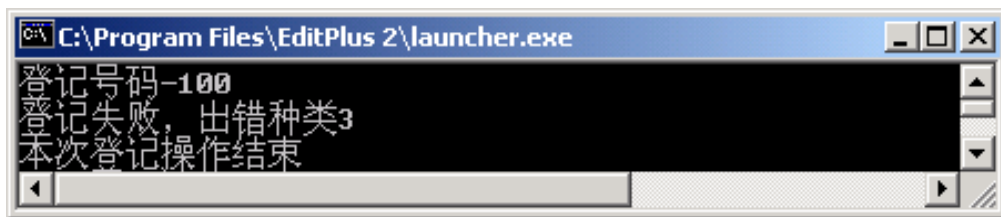


图 6-3 用户定义的异常类

习题

1. 异常可以分成几类?
2. 用 `main()` 创建一个类, 令其抛出 `try` 块内的 `Exception` 类的一个对象。为 `Exception` 的构建器赋予一个字符串参数。在 `catch` 从句内捕获异常, 并打印出字符串参数。添加一个 `finally` 从句, 并打印一条消息。
3. 用 `extends` 关键字创建自己的异常类。为这个类写一个构建器, 令其采用 `String` 参数, 并随同 `String` 句柄把它保存到对象内。写一个方法, 令其打印出保存下来的 `String`。

创建一个 `try-catch` 从句，练习实际操作新异常。

4. 写一个类，并在一个方法抛出一个异常。试着在没有异常规范的前提下编译它，观察编译器会报告什么。接着添加适当的异常规范。在一个 `try-catch` 从句中尝试自己的类及它的异常。

第 7 章 工具类及常用算法

本章首先介绍 Java 编程中经常要使用的结构和工具类,包括 Java 的语言基础类库,包括 Object, Math 和字符串。然后讨论一些常用数据结构的面向对象的实现,包括集合、列表、向量、链表、堆栈和队列。这些工具将为读者的实际应用开发提供方便。同时,本章还将介绍一些常用算法,如排序、查找、遍试、迭代和递归等。

7.1 Java 语言基础类

7.1.1 Java 基础类库

Java 程序设计就是定义类的过程,但是 Java 编程时还需要用到大量的系统定义好的类,即 Java 基本类库中的类。类库是 Java 语言的重要组成部分。Java 语言由语法规则和类库两部分组成,语法规则确定 Java 程序的书写规范;类库,或称为运行时库,则提供了 Java 程序与运行它的系统软件(Java 虚拟机)之间的接口,它是 Java 编程的 API(application program interface),它可以帮助开发者方便、快捷地开发 Java 程序。

JDK 中提供的基础类库又称为 JFC (Java Foundation Class Library),其中包含多个包,每个包中都有若干个具有特定功能和相互关系的类和接口。下面列出了一些经常使用的包及相关的类。

1 . java.lang 包

java.lang 包是 Java 语言的核心类库,包含了运行 Java 程序必不可少的系统类,如基本数据类型、基本数学函数、字符串处理、线程、异常处理类等。每个 Java 程序运行时,系统都会自动地引入 java.lang 包,所以这个包的加载是默认的。

2 . java.io 包

java.io 包是 Java 语言的标准输入/输出类库,包含了实现 Java 程序与操作系统、用户界面及其他 Java 程序做数据交换所使用的类,如基本输入/输出流、文件输入/输出流、过滤输入/输出流、管道输入/输出流、随机输入/输出流等。凡是需要完成与操作系统有关的较底层的输入输出操作的 Java 程序,都要用到 java.io 包。

3 . java.util 包

java.util 包包括了 Java 语言中的一些低级的实用工具,如处理时间的 Date 类、处理变长数组的 Vector 类、实现栈和杂凑表的 Stack 类和 HashTable 类等,通过使用它们,开发者可以更方便快捷地编程。

4 . java.awt 包

java.awt 包是 Java 语言用来构建图形用户界面(GUI)的类库,它包括了许多界面元素和资源,主要在三个方面提供界面设计支持:低级绘图操作,如 Graphics 类等;图形界面组件和布局管理,如 Checkbox 类、Container 类、LayoutManager 接口等;以及界面用户交互控制和事件响应,如 Mouse Event 类。利用 java.awt 包,开发人员可以很方便地编写出美观、方便、标准化的应用程序界面。

5 . java.applet 包

java.applet 包是用来实现运行于 Internet 浏览器中的 Java Applet 的工具类库,它仅包含少量几个接口和一个非常有用的类: java.applet.Applet。

6 . java.net 包

java.net 包是 Java 语言用来实现网络功能的类库。由于 Java 语言还在不停地发展和扩充,它的功能,尤其是网络功能,也在不断地扩充。目前已经实现的 Java 网络功能主要有:底层的网络通信,如实现套接字通信的 Socket 类、ServerSocket 类;编写用户自己的 Telnet, FTP, 邮件服务等实现网上通信的类;用于访问 Internet 上资源和进行 CGI 网关调用的类,如 URI 等。利用 java.net 包中的类,开发者可以编写自己的具有网络功能的程序。

7 . 其他包

Java 中还有其他包,如 java.corba 包将 CORBA(common object request broker architecture, 一种标准化接口体系)嵌入到 Java 环境中,使得 Java 程序可以存取、调用 CORBA 对象,并与 CORBA 对象共同工作。这样,Java 程序就可以方便、动态地利用已经存在的由 Java 或其他面向对象语言开发的部件,简化软件的开发。

Java.rmi 包用来实现 RMI(remote method invocation, 远程方法调用)功能。利用 RMI 功能,用户程序可以在远程计算机(服务器)上创建对象,并在本地计算机(客户机)上使用这个对象。

Java.security 包提供了更完善的 Java 程序安全性控制和管理,利用它们可以对 Java 程序加密,也可以把特定的 Java Applet 标记为“可信赖的”,使它能够具有与 Java Application 相近的安全权限。

Java.sql 包是实现 JDBC(Java database connection)的类库。利用这个包可以使 Java 程序具有访问不同种类的数据库的功能,如 Oracle, Sybase, DB2, SQLServer 等。只要安装了合适的驱动程序,同一个 Java 程序不需修改就可以存取、修改这些不同的数据库中的数据。JDBC 的这种功能,再加上 Java 程序本身具有的平台无关性,大大拓宽了 Java 程序的应用范围,尤其是商业应用的适用领域。

使用 JDK 类库中类的基本方法是分创建类的对象或者从类进行派生。本书中会介绍其中一些类的基本使用方法,更详细的文档参见 JDK 的 API 文档。JDK 的 API 文档可以从 java.sun.com 网站下载,安装后,打开 index.html 即可,如图 7-1 所示,有包、类、属性、

The screenshot shows a web browser displaying the Java API documentation for `java.lang.Object`. The left sidebar lists various Java packages and classes, including `Boolean`, `Byte`, `Character`, `ClassLoader`, `Compiler`, `Double`, `Float`, `InheritableThreadLocal`, `Integer`, `Long`, `Math`, `Number`, `Object`, `Package`, `Process`, and `Runtime`.

Constructor Summary

Constructor	Description
<code>Object()</code>	

Method Summary

Modifier	Method	Description
protected	<code>clone()</code>	Creates and returns a copy of this object.
boolean	<code>equals(Object obj)</code>	Indicates whether some other object is "equal to" this one.
protected void	<code>finalize()</code>	Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class	<code>getClass()</code>	Returns the runtime class of an object.
int	<code>hashCode()</code>	Returns a hash code value for the object.
void	<code>notify()</code>	Wakes up a single thread that is waiting on this object's monitor.
void	<code>notifyAll()</code>	Wakes up all threads that are waiting on this object's

7.1.2 Object 类

Object 类包含了所有 Java 类的公共属性，其中较主要的有如下一些方法。

- (1) `protected Object clone();` // 生成当前对象的一个备份，并返回这个复制对象
- (2) `public boolean equals(Object obj);` // 比较两个对象是否相同，是则返回 `true`
- (3) `public final Class getClass();` // 获取当前对象所属的类信息，返回 `Class` 对象
- (4) `protected void finalize();` // 定义回收当前对象时所需完成的清理工作
- (5) `public String toString();` // 返回当前对象本身的有关信息，按字符串对象返回
- (6) `public final void notify();` // 唤醒线程
- (7) `public final void notifyAll();` // 唤醒所有等待此对象的线程
- (8) `public final void wait()throws InterruptedException;` // 等待线程

其中, `getClass()`、`finalize()` 分别在 5.2 及 5.4 中进行了阐述, 还有 `notify()`, `notifyAll()`, `wait()` 用于多线程处理中的同步, 将在第 8 章中详细讲述。

这里介绍其他几个常用方法。

1. equals()方法与==运算符

equals()方法用来比较两个对象是否相同，如果相同，则返回 true，否则返回 false。

如果一个类没有覆盖 equals()方法，那么它的“相等”意味着两个引用相等，即它们引用的是同一个对象。这时，equals()方法的结果与相等运算符(==)的结果相同。

注意：==运算符可用于基本数据类型（判断数据是否相等），也可用于引用类型。当用于引用类型时，表示是否引用同一个对象（判断句柄是否相等）。

由于 JDK 中的许多类在实现时都已经覆盖了 equals()方法，这时它判断的是两个对象状态和功能上的相同，而不是引用上的相同。这时两个对象“相等”意味着：首先是两个对象类型相同，然后是对象状态和功能上的相同。

例如：

```
Integer one = new Integer (1),  anotherOne = new Integer (1);
if (one.equals (anotherOne))
    System.out.println ("objects are equal");
```

例中，equals()方法返回 true，因为对象 one 和 anotherOne 包含相同的整数值 1，虽然它们在内存的位置并不相同。

注意：在实际应用中，区分 equals()方法与==运算符是十分重要的。例如，判断两个字符串是否相等，实际上是判断内容是否相等，就应该用 equals 方法，而不是 ==。

例 7-1 TestEqualsString.java 有关字符串的相等。

```
public class TestEqualsString {
    public static void main(String[] args) {
        String name1 = new String("LiMing");
        String name2 = new String("LiMing");
        System.out.println( name1==name2 );
                                // 两个对象的引用，不相等

        System.out.println( name1.equals(name2) );
                                // 内容，相等

        String name3 = "LiMing";
        String name4 = "LiMing";
        System.out.println( name3==name4 );
                                // 相同常量的引用，相等

        System.out.println( name3.equals(name4) );
                                // 内容，相等
    }
}
```

```
}
```

程序中使用了==及 equals，要注意对于字符串常量，Java 是被当做对象的，但与 new 创建的对象有些不同，主要是对于==的情况。

程序产生的结果是：

```
false
true
true
true
```

例 7-2 MyDate.java 有关 equals。

```
class MyDate {
    int day,month,year;
    public MyDate(int i,int j,int k) {
        day = i;
        month = j;
        year = k;
    }
}

class MyOkDate extends MyDate{
    public MyOkDate(int i,int j,int k ){
        super(i,j,k);
    }
    public boolean equals( Object obj ){
        if( obj instanceof MyOkDate ){
            MyOkDate m = (MyOkDate)obj;
            if(m.day==day && m.month==month && m.year==year )
                return true;
        }
        return false;
    }
}

public class TestEqualsObject{
    public static void main(String[] args) {
        MyDate m1 = new MyDate(24, 3, 2001);
        MyDate m2 = new MyDate(24, 3, 2001);
        System.out.println( m1.equals(m2) );    // 不相等,显示 false
    }
}
```

```

        m1 = new MyOkDate( 24, 3, 2001 );
        m2 = new MyOkDate( 24, 3, 2001 );
        System.out.println( m1.equals(m2) );    // 相等,显示 true
    }
}

```

在该程序中,对于 `MyDate` 类,没有覆盖 `equals()` 方法,而对于 `MyOkDate` 类,覆盖了 `equals()` 方法,所以程序的显示结果不同,分别为 `false` 及 `true`。

2. toString()

`toString()` 方法用来返回对象的字符串表示,可以用于显示一个对象。例如:

```
System.out.println ( Thread.currentThread ().toString () );
```

可以显示当前的线程。

事实上, `System.out.println()` 方法,如果带一个对象做参数,则自动调用对象的 `toString()` 方法。另外,在字符串的加号运算符,如果连接的是对象,也会自动调用 `toString()` 方法,由于 `toString()` 的广泛应用,所以在自定义的类中,最好覆盖 `toString ()` 方法。

例 7-3 MyDate.java 使用 toString()方法。

```

class MyDate {
    int day,month,year;
    public MyDate(int i,int j,int k) {
        day = i;
        month = j;
        year = k;
    }
}

class MyOkDate extends MyDate{
    public MyOkDate(int i,int j,int k ){
        super(i,j,k);
    }
    public String toString(){
        return year + "-" + month + "-" + day;
    }
}

public class TestToString{
    public static void main(String[] args) {
        MyDate m1 = new MyDate(24, 3, 2001);
        MyDate m2 = new MyOkDate(24, 3, 2001);
    }
}

```

```
        System.out.println( m1 );    // 显示 MyDate@132f0db
        System.out.println( m2 );    // 显示 2001-3-24
    }
}
```

例 7-3 中, MyDate 类没有覆盖 toString()方法, 所以 m1 显示的结果是类名@内存地址。而 MyOkDate 类由于覆盖了 toString()方法, 其对象 m2 显示出来的信息更有意义。

7.1.3 基本数据类型的包装类

Java 的基本数据类型用于定义简单的变量和属性将十分方便, 但为了与面向对象的环境一致, Java 中提供了基本数据类型的包装类 (wrapper), 它们是这些基本类型的面向对象的代表。与 8 种基本数据类型相对应, 基本数据类型的包装类也有 8 种, 分别是: Character, Byte, Short, Integer, Long, Float, Double, Boolean。

这几个类有以下共同特点。

- (1) 这些类都提供了一些常数, 以方便使用, 如 Integer.MAX_VALUE (整数最大值), Double.NaN (非数字), Double.POSITIVE_INFINITY (正无穷) 等。
- (2) 提供了 valueOf(String), toString(), 用于从字符串转换及或转换成字符串。
- (3) 通过 xxxxValue()方法可以得到所包装的值, Integer 对象的 intValue()方法。
- (4) 对象中所包装的值是不可改变的 (immutable)。要改变对象中的值只有重新生成新的对象。
- (5) toString(), equals()等方法进行了覆盖。

除了以上特点外, 有的类还提供了一些实用的方法以方便操作。例如, Double 类就提供了更多的方法来与字符串进行转换。

例 7-4 DoubleAndString.java 练习 double 与 String 之间相互转换的方法。

```
class DoubleAndString
{
    public static void main(String[] args)
    {
        double d; String s;

        //double 转成 string 的几种方法
        d=3.14159;
        s = "" + d;
        s = Double.toString( d );
        s = new Double(d).toString();
        s = String.valueOf( d );
    }
}
```

```
// String 转成 double 的几种方法
s = "3.14159";
try{
    d = Double.parseDouble( s );
    d = new Double(s).doubleValue();
    d = Double.valueOf( s ).doubleValue();
}
catch(NumberFormatException e )
{
    e.printStackTrace();
}
}
```

7.1.4 Math 类

Math 类用来完成一些常用的数学运算，它提供了若干实现不同标准数学函数的方法。这些方法都是 static 的类方法，所以在使用时不需要创建 Math 类的对象，而直接用类名做前缀，就可以很方便地调用这些方法。

下面简单列出了 Math 类的主要属性和方法：

- ☞ public final static double E; // 数学常量 e
- ☞ public final static double PI; // 圆周率常量
- ☞ public static double abs(double a); // 绝对值
- ☞ public static double exp(double a); // 参数次幂
- ☞ public static double floor(double a); // 不大于参数的最大整数
- ☞ public static double IEEE remainder(double f1, double f2); // 求余
- ☞ public static double log(double a); // 自然对数
- ☞ public static double max(double a, double b); // 最大值
- ☞ public static float min(float a, float b); // 最小值
- ☞ public static double pow(double a, double b); // 乘方
- ☞ public static double random(); // 产生 0 和 1(不含 1)之间的伪随机数
- ☞ public static double rint(double a); // 四舍五入
- ☞ public static double sqrt(double a); // 平方根
- ☞ public static double sin(double a); // 正弦
- ☞ public static double cos(double a); // 余弦
- ☞ public static double tan(double a); // 正切
- ☞ public static double asin(double a); // 反正弦
- ☞ public static double acos(double a); // 反余弦
- ☞ public static double atan(double a); // 反正切

例 7-5 TestMath.java 使用 Math 类。

```
public class TestMath{  
    public static void main (String args[])  
    {  
        System.out.println("Math.ceil(3.1415)= " +  
            Math.ceil(3.1415));  
        System.out.println("Math.floor(3.1415)=" +  
            Math.floor(3.1415));  
        System.out.println("Math.round(987.654)= " +  
            Math.round(987.654));  
        System.out.println("Math.max(-987.654,301)=" +  
            Math.max(-987.654,301));  
        System.out.println("Math.min(-987.654,301)=" +  
            Math.min(-987.654,301));  
        System.out.println("Math.PI=" + Math.PI);  
        System.out.println("Math.E=" + Math.E);  
    }  
}
```

运行结果如图 7-2 所示。

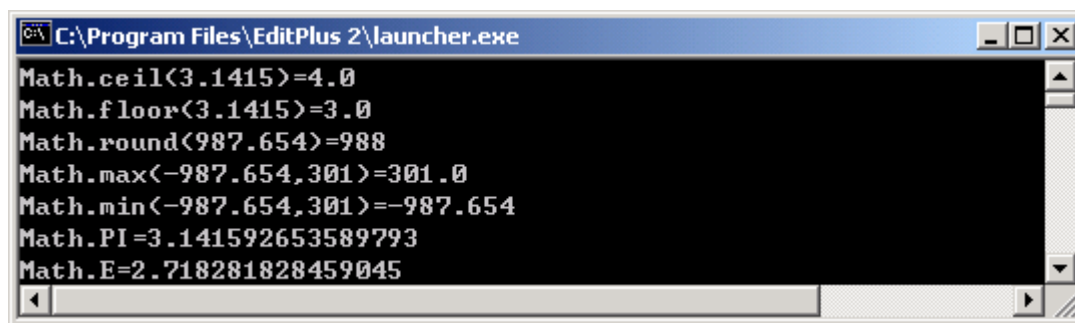


图 7-2 使用 Math 类的运行结果

7.1.5 System 类

System 是一个功能强大、非常有用的特殊的类，它提供了标准输入/输出、运行时的系统信息等重要工具。这个类不能实例化，即不能创建 System 类的对象，所以它所有的属性和方法都是 static 的，引用时以 System 为前缀即可。

1. 用 System 类获取标准输入与输出

System 类的属性有如下三种：系统的标准输入、标准输出和标准错误输出。

☞ public static InputStream in;


```
✎ public static PrintStream out;
```

```
✎ public static PrintStream err;
```

通过使用这三个属性，Java 程序就可以从标准输入读入数据并向标准输出写出数据：

```
char c=System.in.read();           // 从标准输入读入一个字节的信  
                                   息并  
                                   返回给一个字符变量
```

```
System.out.println("Hello! "); // 向标准输出字符串
```

通常情况下，标准输入指的是键盘，标准输出和标准错误输出指的是屏幕。

2. 用 System 类的方法获取系统信息，完成系统操作

System 类提供了一些用来与运行 Java 的系统进行交互操作的方法，利用它们可以获取 Java 解释器或硬件平台的系统参量信息，也可以直接向运行系统发出指令来完成操作系统级的系统操作。下面列出了部分常用的 System 类方法。

(1) public static long current TimeMillis(); // 获取自 1970 年 1 月 1 日零时至当前系统时刻的微秒数，通常用于比较两事件发生的先后时间差。

(2) public static void exit(int status); // 在程序的用户线程执行完之前，强制 Java 虚拟机退出运行状态，并把状态信息 status 返回给运行虚拟机的操作系统。

(3) public static void gc(); // 强制调用 Java 虚拟机的垃圾回收功能。收集内存中已丢失的垃圾对象所占用的空间，使之可以被重新加以利用。这在第 5 章中有介绍。

(4) public static Properties getProperties(); // 得到系统的属性。

7.2 字符串

字符串是字符的序列，在 Java 中，字符串无论是常量还是变量，都是用类的对象来实现的。程序中需要用到的字符串可以分为两大类，一类是创建之后不会再做修改和变动的字符串常量；另一类是创建之后允许再做更改和变化的字符串。前者是 String 类，后者是 StringBuffer 类。

7.2.1 String 类

字符串常量用 String 类的对象表示。在前面的程序中，已多次使用了字符串常量。这里首先强调一下字符串常量与字符常量的不同。字符常量是用单引号括起的单个字符。而字符串常量是用双引号括起的字符序列。在 Java 中，对于所有用双引号括起的字符串常量(又称做字符串字面常数)都被认为是对象。

本节讨论 String 对象的创建、使用和操作。

1. 创建 String 对象及赋值

在创建 String 对象时，通常需要向 String 类的构造函数传递参数来指定所创建的字符串的内容。下面简单列出 String 类的构造函数及其使用方法。

(1) public String(); 用来创建一个空的字符串。

(2) `public String(String value)`; 利用一个已经存在的字符串创建一个新的 `String` 对象。该对象的内容与给出的字符串一致。这个字符串可以是另一个 `String` 对象，也可以是一个用双引号括起的直接常量。

(3) `public String(StringBuffer buffer)`; 利用一个已经存在的 `StringBuffer` 对象为新建的 `String` 对象初始化。`StringBuffer` 对象代表内容、长度可改变的字符串变量，将在 7.2.2 节介绍。

(4) `public String(char value[])`; 利用已经存在的字符数组的内容初始化新建的 `String` 对象。

了解了 `String` 类的构造函数之后，再来看几个创建 `String` 对象的例子。创建 `String` 对象与创建其他类的对象一样，分为对象的声明和对象的创建两步。这两步可以分成两个独立的语句，也可以在一个语句中完成。

在 Java 中，还有一种非常特殊而常用的创建 `string` 对象的方法。这种方法直接利用双引号括起的字符串为新建的 `String` 对象赋值：

```
String s="ABC";
```

下面的例子演示了几种常见的字符串赋值的方法。

例 7-6 StringAssign.java 给 String 变量赋值。

```
class StringAssign
{
    public static void main(String[] args)
    {
        // 几种常见的字符串赋值的方法
        String s;

        // 直接赋值
        s = "Hello";
        s = new String( "Hello" );

        // 使用 StringBuilder
        s = new String( new StringBuffer( "Hello" ) );
        s = new StringBuffer( "Hello").toString();

        // 对象转为字符串
        s = new Object().toString();
        s = "" + new Object();
    }
}
```

2. 字符串的长度

☞ `public int length();`

求字符串的长度。用它可以获得当前字符串对象中字符的个数。如，字符串“Hello!”的长度为 6。

需要注意的是在 Java 中，因为每个字符是占用 16 个比特的 Unicode 字符，所以汉字与英文或其他符号相同，也只用一个字符表示就足够了。

3. 判断字符串的前缀和后缀

☞ `public boolean startsWith(String prefix);`

☞ `public boolean endsWith(String suffix);`

这两个方法可以分别判断当前字符串的前缀和后缀是否是指定的字符子串。区分字符串的前缀及后缀在某些情况下是非常有用的操作。

`startsWith` 和 `endsWith` 这两个方法的一个突出优点是限制所判断的前缀、后缀的长度。

4. 字符串中单个字符的查找

☞ `public int indexOf(int ch);`

☞ `public int indexOf(int ch, int fromIndex);`

上述两个方法查找当前字符串中某特定字符出现的位置。第一个方法查找字符 `ch` 在当前字符串中第一次出现的位置，即从头向后查找，并返回字符 `ch` 出现的位置。如果找不到则返回 -1。

5. 字符串中子串的查找

☞ `public int indexOf(String str);`

☞ `public int indexOf(String str, int fromIndex);`

☞ `public int lastIndexOf(String str);`

☞ `public int lastIndexOf(String str, int fromIndex);`

在字符串中查找字符子串与在字符串中查找单个字符非常相似，也有以上四种可供选用的方法，它就是把查找单个字符的四个方法中的指定字符 `ch` 换成了指定字符子串 `str`。

6. 比较两个字符串

☞ `public int compareTo(String anotherString);`

☞ `public boolean equals(Object anObject);`

☞ `public boolean equalsIgnoreCase(String anotherString);`

`String` 类中有三个方法可以比较两个字符串是否相同。方法 `equals` 是覆盖 `Object` 类的方法，它将当前字符串与方法的参数列表中给出的字符串相比较，若两字符串相同，则返回 `true`，否则返回 `false`。方法 `equalsIgnoreCase` 与方法 `equals` 的用法相似，只是它比较字符串时将不计字母大小写的差别。

比较字符串的另一个方法是 `compareTo()`，这个方法将当前字符串与一个参数字符串相比较，并返回一个整型量。如果当前字符串与参数字符串完全相同，则 `compareTo()` 方法返回 0；如果当前字符串按字母序大于参数字符串，则 `compareTo()` 方法返回一个大于 0 的整数；反之，若 `compareTo()` 方法返回一个小于 0 的整数，则说明当前字符串按字母序小于参数字符串。

7. 求字符及子串

```
✎ public char charAt(int index);  
✎ public String substring( int startIndex, int endIndex );  
✎ public String substring( int startIndex );
```

其中，`index` 是位置，位置是基于 0 的，即首字符是第 0 个字符。`substring()` 是求子串的方法，`startIndex` 是子串的起始位置，`endIndex` 是子串的结束位置（但是不包括 `endIndex` 的字符）。若缺省 `endIndex` 则从 `startIndex` 一直到结束。

8. 其他操作

```
✎ public String concat(String str); // 连接字符串  
✎ public String trim(); // 去掉字符串前后的空格  
✎ public String toUpperCase(); // 转成大写  
✎ public String toLowerCase(); // 转成小写  
✎ String replace(char oldChar, char newChar); // 替换字符串中的字符。
```

注意：在使用 `String` 类时，`String` 对象是不可变对象（immutable）。

`String` 字符串一经创建，无论其长度还是内容，都不能再更改了。`String` 类的各种操作，包括连接、替换、转换大小写等，都是返回一个新的字符串对象，而原字符串的内容并没有改变。这一点特别重要。

例 7-7 TestStringMethod.java 使用 `String`。

```
class TestStringMethod  
{  
    public static void main(String[] args)  
    {  
        String s = new String( "Hello World" );  
  
        System.out.println( s.length() );  
        System.out.println( s.indexOf('o') );  
        System.out.println( s.indexOf("He") );  
        System.out.println( s.startsWith("He") );  
        System.out.println( s.equals("Hello world") );  
        System.out.println( s.equalsIgnoreCase("Hello world") );  
    }  
}
```

```
System.out.println( s.compareTo("Hello Java") );
System.out.println( s.charAt(1) );
System.out.println( s.substring(0,2) );
System.out.println( s.substring(2) );
System.out.println( s.concat("!!!") );
System.out.println( s.trim() );
System.out.println( s.toUpperCase() );
System.out.println( s.toLowerCase() );
System.out.println( s.replace('o', 'x' ) );

System.out.println( s );    //注意, s 本身没有改变
    }
}
```

运行结果如图 7-3 所示。

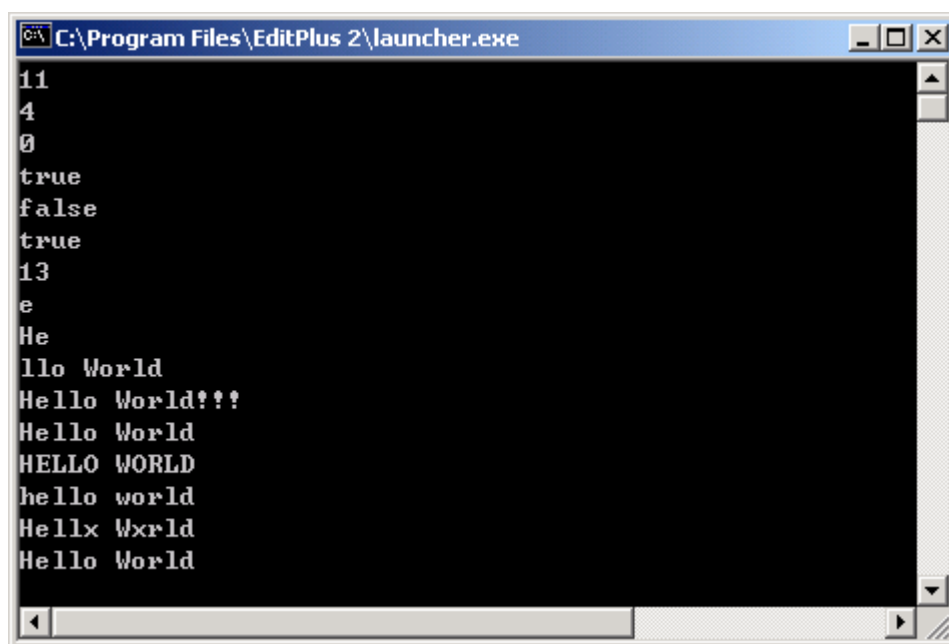


图 7-3 使用 String 类的运行结果

7.2.2 StringBuffer 类

Java 中用来实现字符串的另一个类是 StringBuffer 类,与实现字符串常量的 String 类不同, StringBuffer 对象的内容是可以修改的字符串。

1. 创建 StringBuffer 对象

由于 `StringBuffer` 表示的是可扩充、修改的字符串，所以在创建 `StringBuffer` 类的对象时并不一定要给出字符串初值。`StringBuffer` 类的构造函数有以下几个：

- ☞ `public StringBuffer();`
- ☞ `public StringBuffer(int length);`
- ☞ `public StringBuffer(String str);`

第一个函数创建了一个空的 `StringBuffer` 对象，第二个函数给出了新建的 `StringBuffer` 对象的长度，第三个函数则利用一个已经存在的字符串 `String` 对象来初始化 `StringBuffer` 对象。

2. 字符串变量的扩充、修改与操作

`StringBuffer` 类有两组用来扩充其中所包含的字符的方法，分别是：

- ☞ `public StringBuffer append(参数对象类型参数对象名);`
- ☞ `public StringBuffer insert(int 插入位置, 参数对象类型参数对象名);`

`append` 方法将指定的参数对象转化成字符串，附加在原 `stringBuffer` 字符串对象之后，而 `insert` 方法则在指定的位置插入给出的参数对象所转化而得的字符串。附加或插入的参数对象可以是各种数据类型的数据，如 `int`, `double`, `char`, `String` 等。

3. StringBuffer 与 String 的相互转化

由 `String` 对象转成 `StringBuffer` 对象，是创建一个新的 `StringBuffer` 对象，如：

```
String s = "Hello";
StringBuffer sb = new StringBuffer( s );
```

由 `StringBuffer` 转为 `String` 对象，则可以用 `StringBuffer` 的 `toString()` 方法，如：

```
StringBuffer sb = new StringBuffer();
String s = sb.toString();
```

4. 有关字符串的连接运算符（字符串加法+）

字符串是经常使用的数据类型，为了编程方便，Java 编译系统中引入了字符串的加法(+)和赋值(+=)。字符串的连接运算符 (+)，在 Java 中是一种十分特殊的运算符，因为它可以两个字符串连接字符串；如果字符串与一个对象相连，Java 还会自动将对象转成字符串（调用对象的 `toString()` 方法）；如果字符串与基本数据类型相连，则基本数据类型也转化成字符串。

Java 中的字符串 (+) 可以认为是为了方便程序的书写而设立的。

事实上，Java 编译器将这种运算符都转成 `StringBuffer` 的 `append()` 方法。如：

```
String s = "abc" + foo + "def" + 3.14 + Integer.toString(47);
```

7.2.3 StringTokenizer 类

java.util.StringToken 类提供了对字符串进行解析和分割的功能。比如，要对一条语句进行单词的区分，就可以用到该类。

StringTokenizer 的构造方法有：

- ☞ StringTokenizer(String str);
- ☞ StringTokenizer(String str, String delim);
- ☞ StringTokenizer(String str, String delim, boolean returnDelims);

其中，str 是要解析的字符串，delim 是含有分隔符的字符串，returnDelims 表示是否将分隔符也作为一个分割串。

该类的重要方法有：

- ☞ public int countTokens(); // 分割串的个数
- ☞ public boolean hasMoreTokens(); // 是否还有分割串
- ☞ public String nextToken(); // 得到下一分割串

例 7-8 TestStringTokenizer.java 使用 StringTokenizer。

```
import java.util.*;  
class TestStringTokenizer  
{  
    public static void main(String[] args)  
    {  
        StringTokenizer st = new StringTokenizer("this is a test");  
        while (st.hasMoreTokens()) {  
            System.out.println(st.nextToken());  
        }  
    }  
}
```

运行结果如图 7-4 所示。

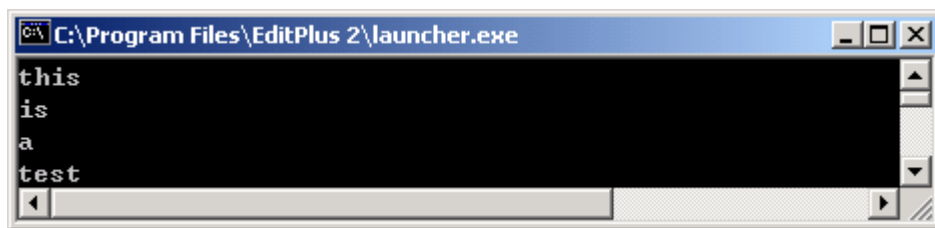


图 7-4 使用 StringTokenizer 的运行结果

7.3 集合类

7.3.1 集合与 Collection API

1. Collection API

集合是一系列对象的聚集(Collection)。集合在程序设计中是一种重要的数据结构。Java 中提供了有关集合的类库称为 Collection API。

在一定意义上, Java 中数组就是一种集合, 但数组是 Java 语言的一个组成部分, 而 Collection API 是一组类库。数组的内容已在第 3 章中介绍, 这里仅谈谈 Collection API 中的相关接口和类。

集合实际上是用一个数组代表一组对象, 在集合中的每个对象称为一个元素。与数组不同的是, 集合中的元素都是对象, (基本数据类型则要使用其包装类才能作为集合的元素。在集合中的各个元素的具体类型可以不同, 但一般说来, 它们都是由相同的类派生出来的(而这一点并不难做到, 因为 Java 中的所有类都是 Object 的子类)。在从集合中检索出各个元素时, 常常要根据其具体类型不同而进行相应的强制类型转换。

Collection API 中的接口和类主要位于 java.util 包中。其中, 最基本的接口是 Collection, 它将一组对象以集合元素的形式组织到一起, 在其子接口中分别实现不同的组织方式。Collection 的子接口有以下两种。

☞ Set (集): 不记录元素的保存顺序, 且不允许有重复元素;

☞ List (列表): 记录元素的保存顺序, 且允许有重复元素。

Set 接口的重要实现类有 HashSet(哈希集)。List 接口的重要实现类有 ArrayList, Vector, LinkedList。

它们的关系如图7-5 所示。

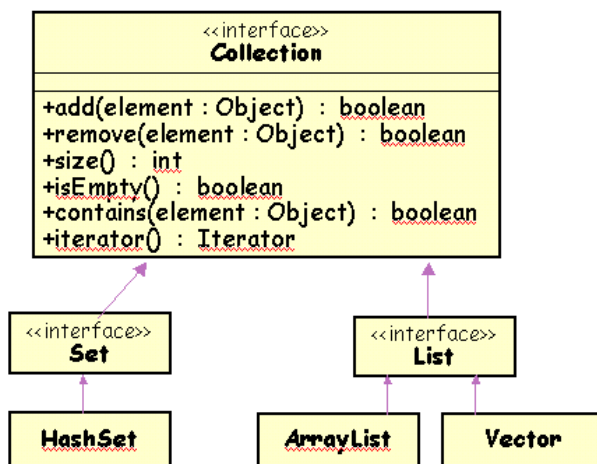


图 7-5 Collection 及其子接口 Set 和 List

2. Collection 及其方法

Collection 接口中重要的方法有：

- ☞ `public boolean add(Object o);` // 加入元素
- ☞ `public boolean remove(Object o);` // 移除元素
- ☞ `public void clear();` // 清除所有元素
- ☞ `public boolean contains(Object o);` // 判断是否包含某元素
- ☞ `public int size();` // 元素个数
- ☞ `public boolean isEmpty();` // 判断是否为空
- ☞ `public Iterator iterator();` // 得到迭代器

7.3.2 Set 接口及 HashSet 类

Set 接口是 Collection 的子接口，HashSet 是实现 Set 接口的一个类。

Set 表示的是不重复元素的集合，所谓不重复，是指两个对象不满足 `a.equals(b)`。在这个意义上，Set 相当于数学中的“集合”的概念。

Set 中可以包含 null 对象，但最多只能有一个 null 对象。

例 7-9 TestHashSet.java 使用 HashSet。

```
import java.util.*;

public class TestHashSet {
    public static void main(String[] args) {
        HashSet h = new HashSet();
        h.add("1st");
        h.add("2nd");
        h.add("3rd");
        h.add("4th");
        h.add("5th");
        h.add(new Integer(6));
        h.add(new Double(7.0));

        h.add("2nd");           // 重复元素，未被加入
        h.add(new Integer(6));  // 重复元素，未被加入

        m1(h);
    }
    public static void m1(Set s){
        System.out.println(s);    // 调用了其 toString()方法,注意显
                                   // 示时,元素无顺序
    }
}
```

```
}
```

程序的显示结果，如图 7-6 所示。

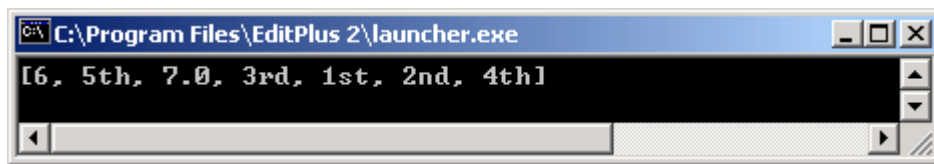


图 7-6 使用 HashSet

可以看出，Set 中的两个特点：

- (1) 元素的顺序与加入时的顺序没有关系，因为 Set 中元素的顺序是无意义的；
- (2) 相等的元素没有被加入，因为 Set 中元素是不能重复的。

7.3.3 List 接口及 ArrayList, Vector 类

List 接口是 Collection 的子类，它表示对象可重复的集合。ArrayList(数组列表), Vector (向量) 是 List 的两个重要实现。ArrayList, Vector 实际上是 Java 中的“动态数组”。我们知道，数组在用 new 创建后，其大小 (length) 是不能改变的，而 ArrayList 及 Vector 中的数组元素的个数 (size()) 是可以改变的，元素可以加入及移除，所以说是“动态数组”。

ArrayList 与 Vector 基本上是相当的，只是 Vector 是线程安全的，因为 Vector 是 synchronized 的，而 ArrayList 则不是。(关于 synchronized，见第 8 章)。

例 7-10 TestArrayList.java 使用 ArrayList。

```
import java.util.*;

public class TestArrayList{
    public static void main(String[] args) {
        ArrayList h = new ArrayList();
        h.add("1st");
        h.add("2nd");
        h.add(new Integer(3));
        h.add(new Double(4.0));
        h.add("2nd");           // 重复元素, 加入
        h.add(new Integer(3));  // 重复元素, 加入
        m1(h);
    }
    public static void m1(List s){
        System.out.println(s);
    }
}
```

本应用程序的运行结果如图 7-7 所示。

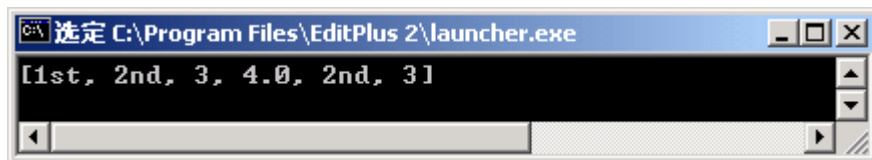


图 7-7 使用 ArrayList 的运行结果

可以看出, List 有两个特点:

- (1) 元素的顺序是有意义的;
- (2) 元素是可以重复的。

由于 ArrayList、Vector 类的元素是有顺序的, 所以除了实现 Collection 接口中的各方法外, 还实现了有关元素的位置(索引)的方法。以 Vector 为例, 其中比较常用的有:

- ✎ public int indexOf(Object elem); // 查找某个元素
- ✎ public Object elementAt(int index); // 获取某个位置的元素
- ✎ public Object set(int index, Object element); // 设定某个位置的元素
- ✎ public void removeElementAt(int index); // 移除某个位置的元素
- ✎ public void insertElementAt(Object obj, int index); // 在某个位置插入元素

7.3.4 Iterator 及 Enumeration

存取集合中的元素可以有多种方法。对于 ArrayList 及 Vector 类, 它们的元素与位置(索引)有关, 可以用与位置相关的方法来取得元素。除此之外, 所有的 Collection 都可以用 Iterator(迭代器)来列举元素, Vector 等类还可以用 Enumeration(枚举器)来列举元素。Iterator 和 Enumeration 都是列举器, 但 Iterator 的方法中还有 remove() 可用于移除对象, 所以 Iterator 的功能更强, 使用更方便。

用 Iterator 接口是一种常用方法。对于实现 Collection 接口的类, 都可以使用 Iterator 接口。事实上, Collection 接口的方法 iterator() 返回的就是 Iterator。

Iterator 的方法有三个:

- ✎ public boolean hasNext(); // 是否还有下一元素
- ✎ public Object next(); // 得到下一元素
- ✎ public remove(); // 移除当前元素

需要注意的是, 为了获取第一个元素, 也必须用 next() 方法。

对于 List, 可以通过其方法 listIterator() 来得到一个 ListIterator 接口, 该接口是 Iterator 接口的子接口, 它具有双向检索的功能, 并能存取到元素的索引。ListIterator 除了具有以上三个方法外, 还具有以下方法:

- ✎ boolean hasPrevious(); // 是否有前一个元素
- ✎ Object previous(); // 得到前一个元素
- ✎ void add(Object o); // 在当前位置前面加入一个元素
- ✎ void set(Object o); // 替换当前位置的元素
- ✎ int nextIndex(); // 下一元素的索引
- ✎ int previousIndex(); // 前一元素的索引

对于有些类，还可以使用 Enumeration 来列举对象。例如，Vector 类的 elements()方法可以返回一个 Enumeration 接口。Enumeration 接口有两个方法：

- ☞ boolean hasMoreElements(); // 还有元素
- ☞ Object nextElement(); // 下一元素

例 7-11 TestListAllElements.java 用几种不同方法列出向量中的所有元素。

```
import java.util.*;

public class TestListAllElements{

    public static void main(String[] args) {
        Vector h = new Vector();
        h.add("1st");
        h.add("2nd");
        h.add("3rd");
        h.add("4th");
        h.add("5th");
        printAll((Object)h);
        printAll((Collection)h);
        printAll((List)h);
        printAll((Vector)h);
        printAllGetByIndex( h );

    }

    public static void printAll(Object s){
        System.out.println(s);
    }

    public static void printAll( Collection s ){
        Iterator it = s.iterator();
        while( it.hasNext() ){
            System.out.println( it.next() );
        }
    }

    public static void printAll( List s ){
        ListIterator it = s.listIterator();
        while( it.hasNext() ){
            System.out.println( it.next() );
        }
        while( it.hasPrevious() ){
            System.out.println( it.previous() );
        }
    }
}
```

```

    }
    public static void printAll( Vector s ){
        Enumeration em = s.elements();
        while(em.hasMoreElements()) {
            System.out.println(em.nextElement());
        }
    }
    public static void printAllGetByIndex( List s ){
        int size = s.size();
        for( int i=0; i<size; i++ ){
            System.out.println(s.get(i));
        }
    }
}
}

```

7.3.5 Map 接口及 Hashtable 类

Map (映射) 接口提供了一组“关键字-值”的集合。这种集合可以从三种角度来查看，一是关键字的集合，二是值的集合，三是“关键字-值”的集合，它们分别由以下三个方法来实现：

- ☞ public Set keySets();
- ☞ public Collection values();
- ☞ public Set entrySet()。

Map 接口的重要实现类有：Hashtable (哈希表)，Properties (属性组) 等。其中，Properties 是 Hashtable 的子类，它的关键字及值都限于 String 类。

Hashtable 中的每个关键字的元素都要求实现 equals() 方法及 hashCode() 方法，以使对象的区分成为可能。Hashtable 可用于实现“字典”的概念，即每个单词是“关键字”，对应于单词的注解就是“值”。

Hashtable 的重要方法有：

- ☞ public void put(Object key, Object value); // 放入一对“关键字-值”
- ☞ public Object get(Object key); // 检索一个值
- ☞ public Object remove(Object key); // 移除一个元素
- ☞ public Enumeration keys(); // 得到键的枚举器

例 7-12 TestHashtable.java 使用 Hashtable。

```

import java.util.*;
class TestHashtable

```

```
{  
    public static void main( String[] args){  
        Hashtable ht = new Hashtable();  
        ht.put("one", new Integer(1));  
        ht.put("two", new Integer(2));  
        ht.put("three", new Integer(3));  
        ht.put("four", new Integer(4));  
        ht.put("five", new Integer(5));  
  
        Enumeration em = ht.keys();  
        while( em.hasMoreElements() ){  
            Object key = em.nextElement();  
            Object value = ht.get( key );  
            System.out.println( " "+key+"="+value);  
        }  
    }  
}
```

运行结果如图 7-8 所示。

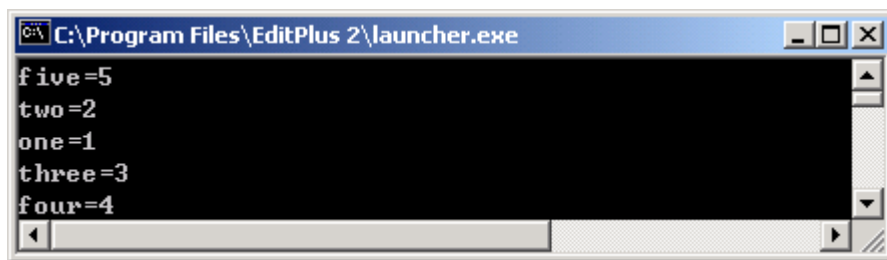


图 7-8 Hashtable 的运行结果

7.4 向量、堆栈、队列

7.4.1 Vector 向量

向量(Vector)是 java.util 包(java.util 包是专门保存各种常用工具类的类库)提供的一个工具类。它对应于类似数组的顺序存储的数据结构,但是具有比数组更强大的功能。

它是允许不同类型元素共存的变长数组。每个 Vector 类的对象可以表达一个完整的数据(每个数据用某种类型的对象表达)序列。Vector 类的对象不但可以保存顺序的一系列数据,而且还封装了许多有用的方法来操作和处理这些数据。另外,Vector 类对象所表达的序列中元素的个数是可变的,即 Vector 实现了变长数组。

相对于数组,Vector 可以追加对象元素数量,可以方便地修改和维护序列中的对象,

所以比较适合在如下的情况中使用。

- (1) 需要处理的对象数目不定，序列中的元素都是对象，或可以表示为对象。
- (2) 需要将不同类的对象组合成一个数据系列。
- (3) 需要做频繁的对象序列中元素的插入和删除。
- (4) 经常需要定位序列中的对象或其他查找操作。
- (5) 在不同的类之间传递大量的数据。

`Vector` 类的方法相对于数组要多一些，但是使用这个类也有一定的局限性。例如，其中的对象不能是简单数据类型等。

一般在下述情况下，使用数组比较合适。

- (1) 序列中的元素是简单数据类型的数据。
- (2) 序列中元素的数目相对固定，插入、删除和查找操作较少。

1. 创建向量类的对象

`Vector` 类有三个构造函数。这里介绍最复杂的一个。

```
public Vector(int initCapacity, int capacityIncrement);
```

这个构造函数有两个形式参数：`initCapacity` 表示刚创建时 `Vector` 序列包含的元素数目；`capacityIncrement` 表示如果需要向 `Vector` 序列中追加元素，那么需一次性地追加多少个。下面的语句就利用这个构造函数创建了一个向量序列：

```
Vector MyVector=new Vector(100, 50);
```

这个语句创建的 `MyVector` 向量序列初始有 100 个元素的空间，以后一旦使用殆尽则以 50 为单位递增，使序列中元素的个数变化成 150, 200, ... 在创建 `Vector` 序列时不需要指明序列中元素的类型，可在使用时再确定。

2. 向向量序列中添加元素

向向量序列中添加元素的方法有两种：`addElement()` 方法将新元素添加在向量序列的尾部，`insertElement()` 方法将新元素插入在序列的指定位置处。

```
addElement(Object obj);
```

```
insertElement(Object obj, int index);
```

其中，`Obj` 是加入到向量序列中的对象，`index` 是插入的位置(0 为第一个位置)。

3. 修改或删除向量序列中的元素

可以使用如下方法修改或删除向量序列中的元素。

(1) `void setElementAt(Object obj, int index)`；将向量序列 `index` 位置处的对象元素设置成为 `obj`，如果这个位置原来有元素则被覆盖。

(2) `boolean removeElement(Object obj)`；删除向量序列中第一个与指定的，`obj` 对象相同的元素，同时将后面元素前提补上空位。

(3) `void removeElementAt(int index)`；删除 `index` 指定位置处的元素，同时将后面的元素向前提。

(4) `void removeAllElements()`；清除向量序列中的所有元素。

4. 查找向量序列中的元素

常用于查找向量序列中某元素的方法如下：

(1) `Object elementAt(int index)`；返回指定位置处的元素。

注意：由于返回的是 `Object` 类型的对象，在使用之前通常需要进行强制类型转换将返回的对象引用转换成 `Object` 类的某个具体子类的对象。

(2) `int indexOf(Object obj, int start_index)`；从指定的 `start_index` 位置开始向后搜索，返回所找到的第一个与指定对象 `obj` 相同的元素的下标位置。若指定对象不存在，则返回 -1。

(3) `int lastIndexOf(Object obj, int start_index)`；从指定的 `start_index` 位置开始向前搜索，返回所找到的第一个与指定对象 `obj` 相同的元素的下标位置。若指定对象不存在，则返回 -1。

例 7-13 PhotoAlbum.java 用一个向量来表示相册。

```
import java.util.*;

class Photo {
    private int PhotoNumber;

    Photo(int i) {
        PhotoNumber = i;
    }

    public String toString() {
        return "This is Photo #" + PhotoNumber;
    }
}

class Printer {
    static void printAll(Enumeration e) {
        while(e.hasMoreElements())
            System.out.println(e.nextElement());
    }
}

public class PhotoAlbum {
    public static void main(String[] args) {
        Vector v = new Vector();
        for(int i = 0; i < 3; i++)
            v.addElement(new Photo(i));
        Printer.printAll(v.elements());
    }
}
```



```
}  
}
```

本例中，用一个向量来表示相册中的多张照片，如图 7-9 所示。

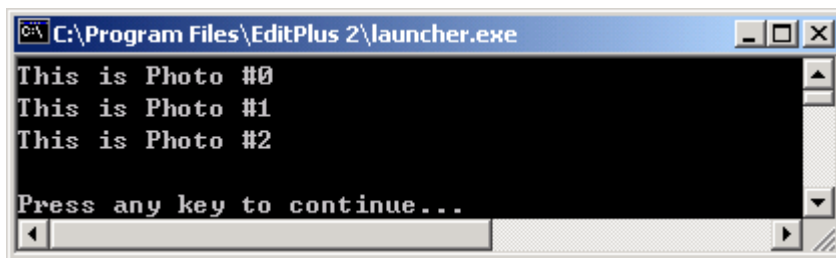


图 7-9 用一个向量来表示相册

7.4.2 Stack 堆栈

堆栈(Stack)又称为栈，也是线性数据结构，并且是遵循“后进先出”(Last In First Out, LIFO)原则的重要线性数据结构。在 Java 中，Stack 是 java.util 包中专门用来实现栈的工具类。

栈只能在一端输入输出，它有一个固定的栈底和一个浮动的栈顶。栈顶可以理解为一个永远指向栈最上面元素的指针。向栈中输入数据的操作成为“压栈”，被压入的数据保存在栈顶，并同时使栈顶指针上浮一格。从栈中输出数据的操作称为“弹栈”，被弹出的总是栈顶指针指向的位于栈顶的元素。如果栈顶指针指向了栈底，则说明当前的堆栈是空的。

Stack 是 Java 用来实现栈的工具。Stack 类是 Vector 的子类，它具有 Vector 的所有方法，同时，Stack 类能实现堆栈操作的方法如下。

(1) 构造函数。

public Stack(): 是栈类惟一的构造函数，创建堆栈时可以直接调用它。

(2) 压栈与弹栈操作。

public Object push(Object item): 将指定对象压入栈中。

Public Object pop(): 将堆栈最上面的元素从栈中取出，并返回这个对象。

(3) 检查堆栈是否为空

public boolean empty(): 若堆栈中没有对象元素，则此方法返回 true，否则返回 false。

注意：压入堆栈和弹出堆栈的都是 Object 对象或是 Object 子类的对象，而不是基本数据类型的数据，所以在上述的例子中堆栈中保存的都是 Integer 类的对象(实际上，一个堆栈里可以保存不同类的对象)。

堆栈最大的特点就是“后进先出”。例如，假设压栈的数据依次为 1, 2, 3, 4, 5, 则弹栈的顺序为 5, 4, 3, 2, 1。压栈和弹栈操作也可以交叉进行，如弹出几个数据后又压入几个数据等。

例 7-14 Stacks.java 使用 Stack。

```
import java.util.*;  
public class Stacks {  
    static String[] months = {  
        "January", "February", "March", "April",  
        "May", "June", "July", "August", "September",  
        "October", "November", "December" };  
    public static void main(String[] args) {  
        Stack stk = new Stack();  
        for(int i = 0; i < months.length; i++)  
            stk.push(months[i] + " ");  
        System.out.println("stk = " + stk);  
        System.out.println("popping elements:");  
        while(!stk.empty())  
            System.out.println(stk.pop());  
    }  
}
```

运行结果如图 7-10 所示。

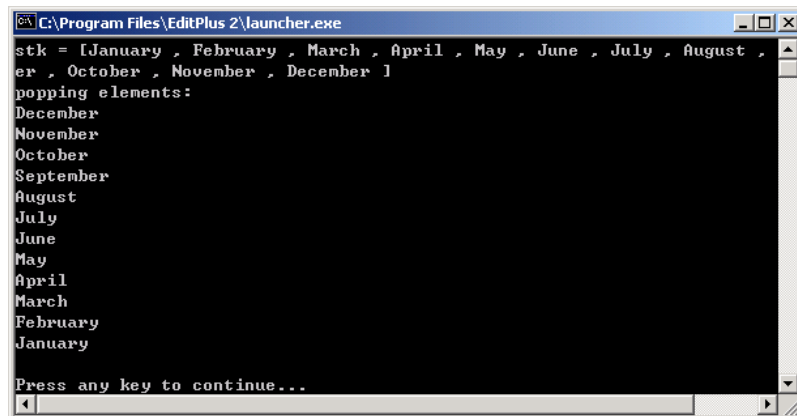


图 7-10 使用 Stack 的运行结果

7.4.3 LinkedList 及队列

队列(Queue),也是重要的线性数据结构。队列遵循“先进先出”(First In First Out, FIFO)的原则,固定在一端输入数据(称为加队),另一端输出数据(称为减队)。可见队列中数据的插入和删除都必须在队列的头尾处进行,而不能像链表一样直接在任何位置插入和删除数据。

计算机系统的很多操作都要用到队列这种数据结构。例如,当需要在只有一个CPU的计算机系统中运行多个任务时,因为计算机一次只能处理一个任务,其他的任务就被安排在一个专门的队列中排队等候。任务的执行,按“先进先出”的原则进行。另外,打印机缓冲池中的等待作业队列、网络服务器中待处理的客户机请求队列,也都是使用队列数据结构的例子。

队列可以用链表(LinkedList)类来实现。与Vector相似,LinkedList类实现了List接口,也是一种线性结构。与Vector不同,LinkedList类提供了在线性序列的头和尾的操作。利用LinkedList,可以实现栈、队列等。

例 7-15 TestQueue.java 用 LinkedList 来实现 Queue。

```
import java.util.*;

class TestQueue
{
    public static void main(String[] args)
    {
        Queue q = new Queue();
        for( int i=0; i<5; i++ )
            q.enqueue( ""+i );
        while( ! q.isEmpty() )
            System.out.println( q.dequeue() );
    }
}

class Queue extends LinkedList
{
    void enqueue( Object obj ){
        addLast( obj );
    }

    Object dequeue(){
        return removeFirst();
    }

    public boolean isEmpty(){
        return super.isEmpty();
    }
}
```

```
    }  
}
```

运行结果如图7-11 所示。

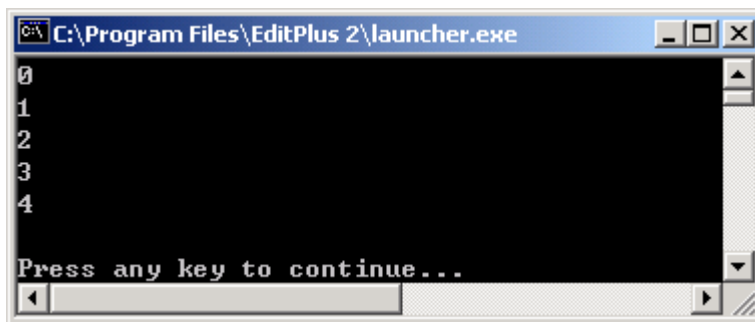


图 7-11 用 LinkedList 来实现 Queue

在例 7-15 中定义的队列类 Queue 是链表 LinkedList 类的子类。Queue 类中定义了三个方法 enqueue(), dequeue() 和 isEmpty() 分别实现加队、减队和判断队列是否为空的操作。该例先创建一个空队列, 加入若干数据, 然后再从队列中顺序取出。

7.5 排序与查找

排序是将一个数据序列中的各个数据元素根据某种大小规则进行从大到小(称为降序)或从小到大(称为升序)排列的过程。查找则是从一个数据序列中找到某个元素的过程。考虑到执行的效率, 人们提出了很多排序算法及查找算法。本节先介绍 JDK 中已经实现的排序及查找的 Arrays 类和 Collections 类, 然后介绍几种实用的排序及查找的算法。

7.5.1 Arrays 类

Arrays 类是用于对数组进行排序和搜索的类。Arrays 类为所有基本数据类型的数组提供了 sort() 和 binarySearch(), 它们亦可用于 String 和 Object。例 7-16 显示出如何排序和搜索一个字节数组 (其他所有基本数据类型都是类似的) 及一个 String 数组。

例 7-16 TestArraysSort.java 使用 Arrays 的 sort()。

```
// Testing the sorting & searching in Arrays  
import java.util.*;  
  
public class TestArraysSort {  
    static Random r = new Random();  
    static String ssource =  
        "ABCDEFGHJKLMNOPQRSTUVWXYZ" +  
        "abcdefghijklmnopqrstuvwxyz";
```

```
static char[] src = ssource.toCharArray();
// Create a random String
public static String randString(int length) {
    char[] buf = new char[length];
    int rnd;
    for(int i = 0; i < length; i++) {
        rnd = Math.abs(r.nextInt()) % src.length;
        buf[i] = src[rnd];
    }
    return new String(buf);
}
// Create a random array of Strings:
public static
String[] randStrings(int length, int size) {
    String[] s = new String[size];
    for(int i = 0; i < size; i++)
        s[i] = randString(length);
    return s;
}
public static void print(byte[] b) {
    for(int i = 0; i < b.length; i++)
        System.out.print(b[i] + " ");
    System.out.println();
}
public static void print(String[] s) {
    for(int i = 0; i < s.length; i++)
        System.out.print(s[i] + " ");
    System.out.println();
}
public static void main(String[] args) {
    byte[] b = new byte[15];
    r.nextBytes(b); // Fill with random bytes
    print(b);
    Arrays.sort(b);
    print(b);
    int loc = Arrays.binarySearch(b, b[10]);
    System.out.println("Location of " + b[10] +
        " = " + loc);
    // Test String sort & search:
```

```
String[] s = randStrings(4, 10);  
print(s);  
Arrays.sort(s);  
print(s);  
loc = Arrays.binarySearch(s, s[4]);  
System.out.println("Location of " + s[4] +  
    " = " + loc);  
}  
}
```

运行结果如图 7-12 所示。

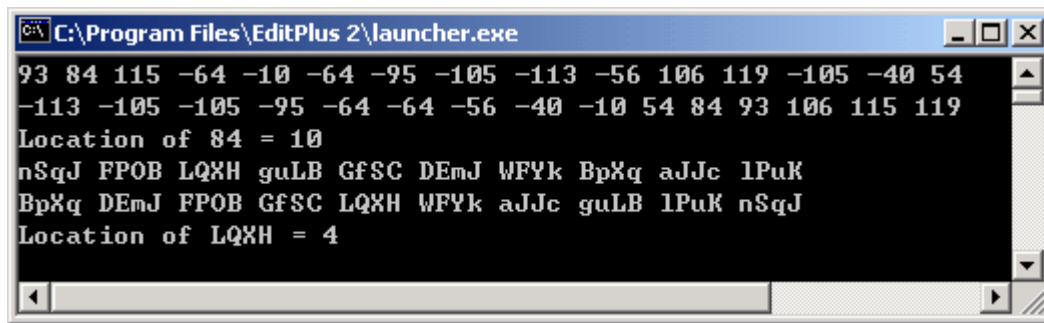


图 7-12 使用 Arrays 的 sort()

类的第一部分包含了用于产生随机字符串对象的实用工具，可供选择的随机字母保存在一个字符数组中。randString()返回一个任意长度的字符串；而 readStrings()创建随机字符串的一个数组，同时给定每个字符串的长度及希望的数组大小。两个 print()方法简化了对示范数组的显示。在 main()中，Random.nextBytes()用随机选择的字节填充数组自变量（没有对应的 Random 方法用于创建其他基本数据类型的数组）。获得一个数组后，便可发现为了执行 sort()或者 binarySearch()，只需发出一次方法调用即可。

注意：若在执行一次 binarySearch()之前不调用 sort()，便会发生不可预测的行为，其中甚至包括无限循环。

对 String 的排序和搜索是相似的，但在运行程序的时候，会注意到一个有趣的现象：排序遵守的是字典顺序，亦即大写字母在字符集中位于小写字母的前面。这是因为字符串的比较方式决定的，在编程时，也可以提供一个 Comparator 来表示比较方式，具体做法可参考 7.5.2 节。

7.5.2 Collections 类

java.util.Collections 类是一个针对 List 具有排序、查找、反序等功能的类。关于查找、排序的方法有：

```
☞ public static void sort(List list);
```

- ✎ `public static void sort(List list, Comparator c);`
- ✎ `public static int binarySearch(List list, Object key);`
- ✎ `public static int binarySearch(List list, Object key, Comparator c);`

这几个方法均为 `static` 方法。`sort` 为排序，`binarySearch` 为二分法查找。参数的含义如下。

- ✎ `list`: 要进行查找、排序的 `List`。对于 `binarySearch` 要求 `list` 已排好序。
- ✎ `key`: 要查找的对象
- ✎ `Comparator c`: 比较器。

在使用时要注意，在排序和查找时，大小关系的规则是靠以下两种方法之一来提供的。

(1) 若不提供 `Comparator`，则 `List` 中的对象必须实现 `java.lang.Comparable` 接口，`Comparable` 接口只有一个方法：

```
int compareTo( Object obj);
```

它根据大小关系返回正数、0、负数；若比 `obj` 大，返回正数；若相等，返回 0；若比 `obj` 小，返回负数。

(2) 提供 `Comparator`。其 `java.util.Comparator` 有两个方法：

- ✎ `int compare(Object o1, Object o2);` // 根据大小关系返回正数、0、负数
- ✎ `boolean equals(Object obj);` // 判断是否相等

例 7-17 TestCollectionsSort.java 排序。

```
import java.util.*;

class TestCollectionsSort
{
    public static void main(String[] args)
    {
        Vector school = new Vector();
        school.addElement( new Person("Li",23));
        school.addElement( new Person("Wang",28));
        school.addElement( new Person("Zhang",21));
        school.addElement( new Person("Tang",19));
        school.addElement( new Person("Chen",22));
        System.out.println( school );

        Collections.sort( school, new PersonComparator() );
        System.out.println( school );

        int index = Collections.binarySearch(
            school, new Person("Li",23), new
                PersonComparator() );
        if( index >=0 )
            System.out.println( "Found:" + school.
```

```
        elementAt( index ));
    else
        System.out.println( "Not Found!" );
    }
}

class Person
{
    String name;
    int age;
    public Person( String name, int age){
        this.name=name;
        this.age=age;
    }
    public String toString(){
        return name+": "+age;
    }
}

class PersonComparator implements Comparator
{
    public int compare( Object obj1, Object obj2 ){
        Person p1 = (Person)obj1;
        Person p2 = (Person)obj2;
        if( p1.age > p2.age ) return 1;
        else if(p1.age<p2.age) return -1;
        return 0;
    }
}
```

7.5.3 冒泡排序

冒泡排序算法的基本思路是把当前数据序列中的各相邻数据两两比较，发现任何一对数据间不符合要求的升序或降序关系则立即调换它们的顺序，从而保证相邻数据间符合升序或降序的关系。以升序排序为例，经过从头至尾的一次两两比较和交换(称为“扫描”)之后，序列中最大的数据被排到序列的最后。这样，这个数据的位置就不需要再变动了，因此就可以不再考虑这个数据，而对序列中的其他数据重复两两比较和交换的操作。

第二次扫描之后会得到整个序列中次大的数据并将它排在最大数据的前面和其他所有数据的后面，这也是它的最后位置，尚未排序的数据又减少了一个。依此类推，每一轮扫

描都将使一个数据就位并使未排序的数据数目减一，所以经过若干轮扫描之后，所有的数据都将就位，未排序数据数目为零，而整个冒泡排序就完成了。

例 7-18 BubbleSort.java 冒泡法排序(从小到大)。冒泡法排序对相邻的两个元素进行比较，并把小的元素交换到前面。

```
public class BubbleSort{
    public static void main( String args[ ] ){
        int i,j;
        int a[ ]={30,1,-9,70,25};
        int n=a.length;
        for( i=1; i<n; i++)
            for( j=0; j<n-i; j++ )
                if( a[j]>a[j+1]){
                    int t=a[j];
                    a[j]=a[j+1];
                    a[j+1]=t;
                }
        for( i=0; i<n; i++ )
            System.out.println(a[i]+" ");
    }
}
```

运行结果如图 7-13 所示。

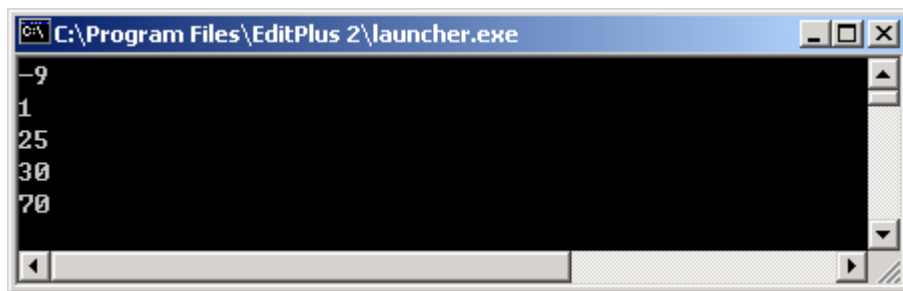


图 7-13 冒泡法排序

7.5.4 选择排序

选择排序的基本思想是从中选出最小值，将它放在前面第 0 位置；然后在剩下的数中选择最小值，将它放在前面第 0 位置，依次类推。

例 7-19 SelectSort.java 选择法排序。

```
public class SelectSort{
```

```
public static void main( String args[ ] ){
    int i,j;
    int a[ ]={30,1,-9,70,25};
    int n=a.length;
    for( i=0; i<n-1; i++)
        for( j=i+1; j<n; j++ )
            if( a[i]>a[j] ){
                int t=a[i];
                a[i]=a[j];
                a[j]=t;
            }
        for( i=0; i<n; i++ )
            System.out.println(a[i]+" ");
    }
}
```

运行结果如图 7-14 所示。

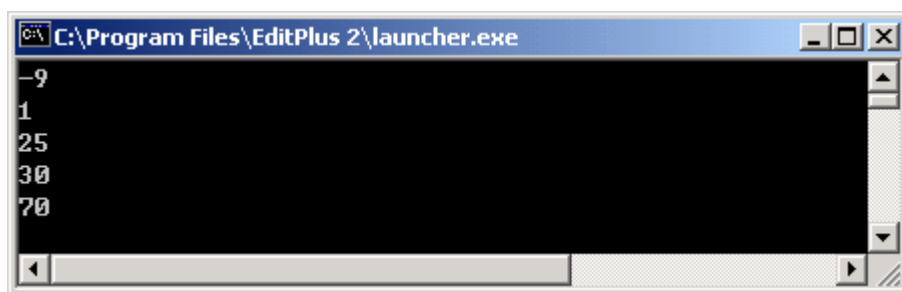


图 7-14 选择法排序

7.5.5 快速排序

快速排序的效率最高，其具体方法较复杂，读者可以参考数据结构方面的书籍，这里给出一个 Java 的程序实现。

例 7-20 SortVector.java 快速排序。

```
import java.util.*;

interface Compare {
    boolean lessThan(Object lhs, Object rhs);
    boolean lessThanOrEqual(Object lhs, Object rhs);
}

class SortVector extends Vector {
```

```
private Compare compare; // To hold the callback
public SortVector(Compare comp) {
    compare = comp;
}
public void sort() {
    quickSort(0, size() - 1);
}
private void quickSort(int left, int right) {
    if(right > left) {
        Object o1 = elementAt(right);
        int i = left - 1;
        int j = right;
        while(true) {
            while(compare.lessThan(
                elementAt(++i), o1))
                ;
            while(j > 0)
                if(compare.lessThanOrEqual(
                    elementAt(--j), o1))
                    break; // out of while
            if(i >= j) break;
            swap(i, j);
        }
        swap(i, right);
        quickSort(left, i-1);
        quickSort(i+1, right);
    }
}
private void swap(int loc1, int loc2) {
    Object tmp = elementAt(loc1);
    setElementAt(elementAt(loc2), loc1);
    setElementAt(tmp, loc2);
}

class QuickSortTest {
    static class StringCompare implements Compare {
        public boolean lessThan(Object l, Object r) {
            return ((String)l).toLowerCase().compareTo(
```

```
        ((String)r).toLowerCase()) < 0;
    }
    public boolean
    lessThanOrEqual(Object l, Object r) {
        return ((String)l).toLowerCase().compareTo(
            ((String)r).toLowerCase()) <= 0;
    }
}
public static void main(String[] args) {
    SortVector sv =
        new SortVector(new StringCompare());
    sv.addElement("d");
    sv.addElement("A");
    sv.addElement("C");
    sv.addElement("c");
    sv.addElement("b");
    sv.addElement("B");
    sv.addElement("D");
    sv.addElement("a");
    sv.sort();
    Enumeration e = sv.elements();
    while(e.hasMoreElements())
        System.out.println(e.nextElement());
}
}
```

运行结果如图 7-15 所示。

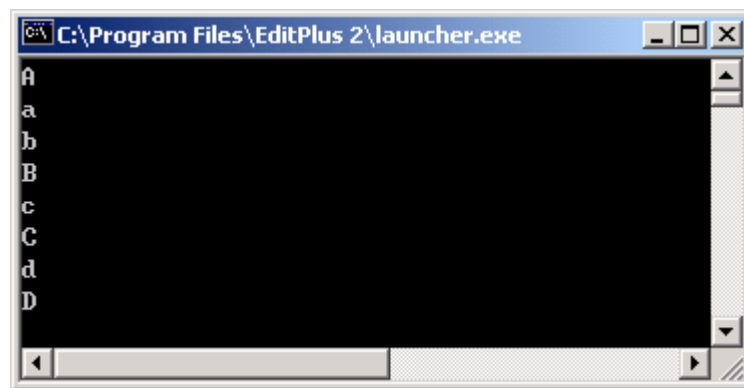


图 7-15 快速排序

7.6 遍试、迭代和递归

本节介绍在程序设计中常用的几种算法，包括遍试、迭代和递归，这些算法属于“通用算法”，它们在解决许多问题中都有应用。

7.6.1 遍试

程序中有一类问题，就是求解满足某种条件的值。大多数问题的求解没有直接的计算公式，但如果在有限的范围内，可以对所有的值都进行试验和判断，从而找到满足条件的值。在本章中，称这种算法为“遍试”。

例 7-21 All_153.java 求三位的水仙花数。所谓三位的水仙花数是指这样的三位数：其各位数字的立方和等于其自身，如 $153 = 1^3 + 5^3 + 3^3$ 。

```
public class All_153
{
    public static void main(String args[]){
        for( int a=1; a<=9; a++ )
            for( int b=0; b<=9; b++ )
                for( int c=0; c<=9; c++ )
                    if( a*a*a+b*b*b+c*c*c == 100*a+10*b+c )
                        System.out.println( 100*a+10*b+c );
    }
}
```

该例中，针对三个数字进行三重循环，如果相关的数满足条件则显示出来，如图 7-16 所示。

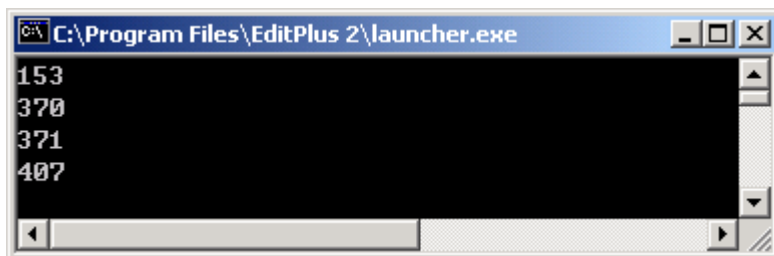


图 7-16 求三位的水仙花数

例 7-22 All_628.java 求 9999 以内的完全数。所谓完全数是指这样的自然数：它的各个约数（不包括该数自身）之和等于该数自身。如 $28=1+2+4+7+14$ 就是一个完全数。

```
class All_628
{
    public static void main( String[] args){
        for( int n=1; n<9999; n++)
            if( n==divsum(n) ) System.out.println(n);
    }
    public static int divsum( int n ){
        int s = 0;
        for( int i=1; i<n; i++ )
            if( n%i == 0 ) s+=i;
        return s;
    }
}
```

在该例中，两次用到了“遍试”的方法。

在主程序中，为了找到满足条件的数，对 1~999 之间的所有数都进行试验和判断，看它是否等于其约数和，若相等，则显示出来。

在求约数和的函数 `divsum` 中，事先不知道谁是约数，于是从 1 到 $n-1$ 都进行判断，检验其是否满足条件 $n\%i==0$ ；若满足，则说明它是约数，将它加入总和中。

运行结果如图 7-17 所示。

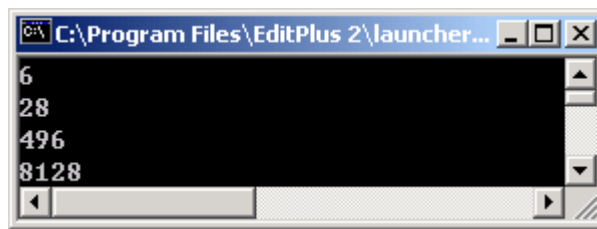


图 7-17 求 9999 以内的完全数

例 7-23 All_220.java 求 1000 以内的“相亲数”。所谓相亲数是指这样的一对数：甲数的约数之和等于乙数，而乙数的约数之和等于甲数。

```
class All_220
{
    public static void main( String[] args){
        for( int n=1; n<9999; n++){
            int s = divsum(n);
            if( n<s && divsum(s)==n )
                System.out.println(n + "," + s);
        }
    }
}
```

```

    }
}
public static int divsum( int n ){
    int s = 0;
    for( int i=1; i<n; i++ )
        if( n%i == 0 ) s+=i;
    return s;
}
}

```

运行结果如图 7-18 所示。

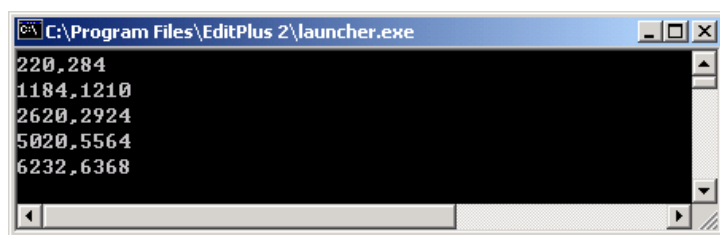


图 7-18 求 1000 以内的“相亲数”

7.6.2 迭代

迭代也是程序设计中的常用算法。迭代，实际上是多次利用同一公式进行计算，每次将计算的结果再代入公式进行计算。迭代在数值计算、分形理论及计算机艺术等领域都有广泛的用途。本节通过实例介绍这种算法。

例 7-24 Sqrt.java 自编一个函数求平方根。

```

public class Sqrt
{
    public static void main(String args[]){
        System.out.println( sqrt( 2.0 ) );
        System.out.println( Math.sqrt(2.0) );
    }

    static double sqrt( double a ){
        double x=1.0;
        do{
            x = ( x + a/x ) /2;
        }while( Math.abs(x*x-a)/a > 1e-6 );
    }
}

```

```

        return x;
    }
}

```

上述公式的直观解释是取 $1 \sim a$ 之间的一个值（这里取 1）作为 f ，然后求 f 与 a/f 之间的算术平均值作为新的 f 。由于平方根总位于 f 与 a/f 之间，这样多次迭代运算就可以逼近平方根，运行结果如图 7-19 所示。

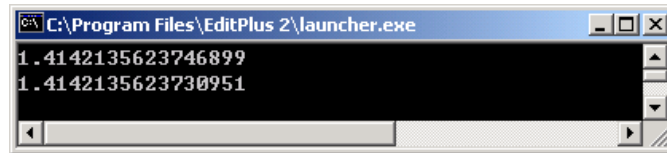


图 7-19 迭代法求平方根

事实上，上述方法是求方程 $x^2 - a = 0$ 的根的方法。这是牛顿迭代法的一个具体应用。设方程 $f(x) = 0$ 。已知在根附近的值 x_0 ，可以用以下迭代公式来逼近真实的根：

$$x_{n+1} = x_n - f(x_n) / f'(x_n)$$

其几何意义如图 7-20 所示。

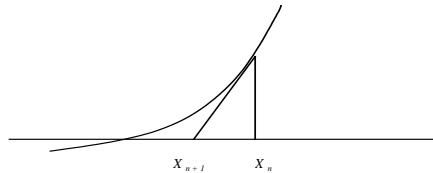


图 7-20 牛顿迭代法求方程的根

例 7-25 Julia.java 利用迭代公式求 Julia 集。Julia 集是分形理论中的一种基本图形，如图 7-21 所示。

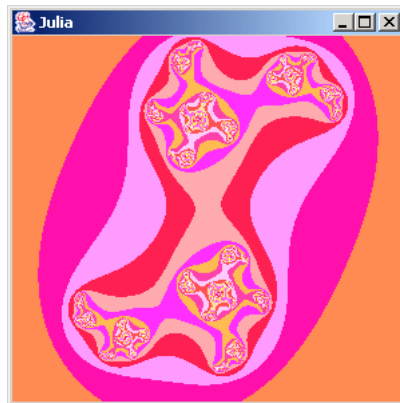


图 7-21 Julia 集


```
import java.awt.*;

class Julia
{
    public static void main( String []largs ){
        Frame frm = new Frame("Julia");
        frm.setSize( 300, 300 );
        frm.show();
        Julia p = new Julia( frm );
        p.drawJulia();
    }

    private Frame frm;
    private Graphics graphics;
    private int width;
    private int height;

    public Julia(Frame frm)
    {
        graphics = frm.getGraphics();
        width = frm.getSize().width;
        height = frm.getSize().height;
    }

    public void drawJulia(){
        //Scale (-1.5, 1.5)-(-1.5, -1.5)

        final double a = 0.5;    //c=a+bi 为 Julia 集的参数
        final double b = 0.55;

        for( double x0 = -1.5; x0 < 1.5; x0+=0.01 )
            for( double y0 = -1.5; y0 < 1.5; y0+=0.01 ){
                double x=x0, y=y0;
                int n;
                for( n = 1; n<100; n++ ){
                    double x2 = x * x - y * y + a;
                    double y2 = 2 * x * y + b;
                    x = x2;
                    y = y2;
                    if( x * x + y * y > 4 ) break;
                }
            }
    }
}
```

```

        pSet(x0, y0, n ); //按 n 值来将(x,y)点进行着色
    }
}

public void pSet(double x, double y, int n){
    graphics.setColor( new Color( n* 0xff8855 ) );
    graphics.drawLine(
        (int)(x*width/3 + width/2),
        (int)(y*height/3 + height/2 ),
        (int)(x*width/3 + width/2),
        (int)(y*height/3 + height/2 )
    );
}
}
}

```

7.6.3 递归

简单地说，递归(recursive)就是一个过程调用过程本身。在递归调用中，一个过程执行的某一步要用到它自身的上一步(或上几步)的结果。

递归是常用的编程技术，其基本思想就是“自己调用自己”，一个使用递归技术的方法即是直接或间接地调用自身的方法。递归方法实际上体现了“依此类推”、“用同样的步骤重复”这样的思想，它可以用简单的程序来解决某些复杂的计算问题，但是运算量较大。

递归调用在完成阶乘运算、级数运算、幂指数运算等方面特别有效。递归分为两种类型，一种是直接递归，即在过程中调用过程本身；一种是间接递归，即间接地调用一个过程。例如，第一个过程调用了第二个过程，而第二个过程又回过头来调用第一个过程。

递归方法解决问题时划分为两个步骤：首先是求得范围缩小的同性质问题的结果，然后利用这个已得到的结果和一个简单的操作求得问题的最后解答。这样一个问题的解答将依赖于一个同性质问题的解答，而解答这个同性质的问题实际就是用不同的参数(体现范围缩小)来调用递归方法自身。

在执行递归操作时，Java 把递归过程中的信息保存在堆栈中。如果无限循环地递归，或者递归次数太多，则产生“堆栈溢出”错误。

例 7-26 Fac.java 用递归方法求阶乘。利用的公式是 $n! = n \times (n-1)!$ 。该公式将 n 的阶乘归结到 $(n-1)$ 的阶乘。

```

public class Fac
{
    public static void main(String args[])
    {
        System.out.println("Fac of 5 is " + fac( 5 ) );
    }
}

```

```

static long fac( int n ){
    if( n==0 || n==1) return 1;
    else return fac(n-1) * n;
}
}

```

例 7-27 Fibonacci.java 求菲波那契(Fibonacci)数列的前 10 项。已知该数列的前两项都为 1，即 $F(1)=1, F(2)=1$ ；而后面各项满足： $F(n)=F(n-1)+F(n-2)$ 。

```

public class Fibonacci
{
    public static void main(String args[])
    {
        System.out.println("Fibonacci(10) is " + fib(10) );
    }
    static long fib( int n ){
        if( n==0 || n==1) return 1;
        else return fib(n-1) + fib(n-2);
    }
}

```

以上方法是用递归方法来实现的，运行结果如图 7-22 所示。可以看出，用递归方法，程序结构简单、清晰。

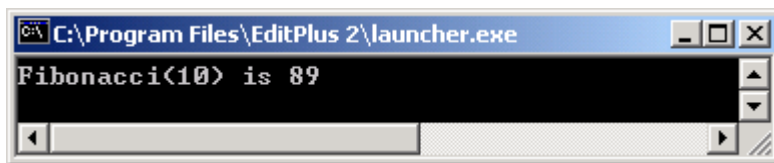


图 7-22 Fibonacci 数列

例 7-28 VonKoch.java 画 Von_Koch 曲线。该曲线可用递归方法画出，如图 7-23 所示。

```

import java.awt.*;

public class VonKoch
{
    public static void main( String []larges ){
        Frame frm = new Frame("VonKoch");
        frm.setSize( 400, 200 );
        frm.setBackground( Color.lightGray );
    }
}

```

```
        frm.show();
        VonKoch p = new VonKoch( frm );
        p.drawVonKoch( 8, p.width );
    }
    private Frame frm;
    private Graphics graphics;
    private int width;
    private int height;
    private double th, curx, cury;
    private final double PI = Math.PI;
    private final double m = 2*(1 + Math.cos(85 * PI/180));

    public VonKoch(Frame frm){
        graphics = frm.getGraphics();
        width = frm.getSize().width;
        height = frm.getSize().height;
    }

    void drawVonKoch( int n, double d ){
        if( n == 0 ){
            double x = curx + d * Math.cos(th * PI / 180);
            double y = cury + d * Math.sin(th * PI / 180);
            drawLineTo (x, y);
            return;
        }
        drawVonKoch( n - 1, d / m );
        th = th + 85;
        drawVonKoch( n - 1, d / m );
        th = th - 170;
        drawVonKoch( n - 1, d / m );
        th = th + 85;
        drawVonKoch( n - 1, d / m );
    }

    void drawLineTo( double x, double y ){
        graphics.drawLine( (int)curx, (int)cury, (int)x, (int)y );
        curx=x;
        cury=y;
    }
}
```

}

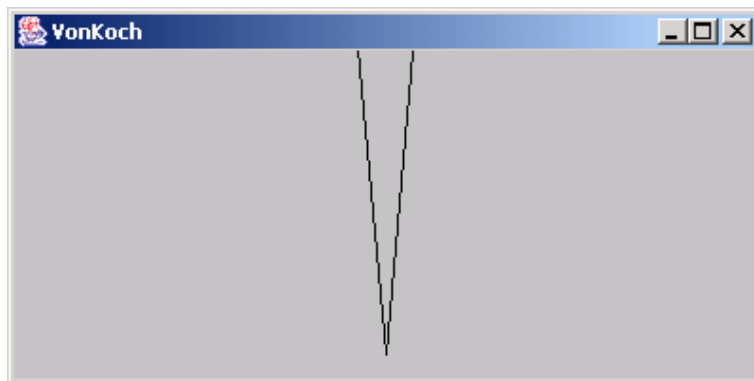
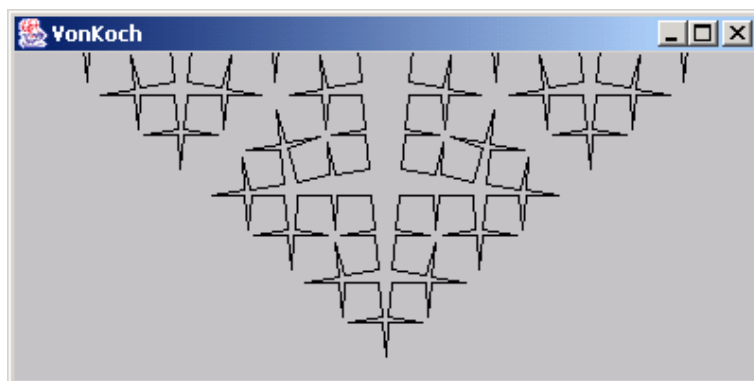
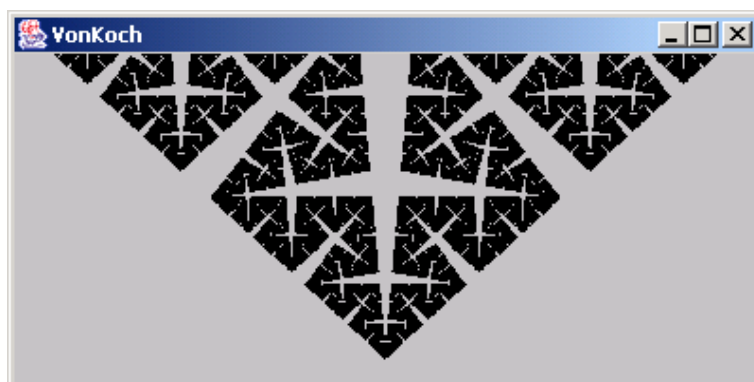
(a) $n=1$ (b) $n=4$ (c) $n=8$

图 7-23 Von_Koch 曲线

例 7-28 CelayTree.java 用计算机生成 Cayley 树。它由 Y 型树多次递归生成，如图 7-24 所示。

```
import java.awt.*;
```

```
public class CelayTree
{
    public static void main( String []largs ){
        Frame frm = new Frame("CelayTree");
        frm.setSize( 400, 400 );
        frm.setBackground( Color.lightGray );
        frm.show();
        CelayTree p = new CelayTree( frm );
        p.drawCelayTree( 10, 200, 400, 100, -Math.PI/2 );
    }
    private Frame frm;
    private Graphics graphics;
    private int width;
    private int height;
    private final double PI = Math.PI;
    private final double th1 = 30 * PI / 180;
    private final double th2 = 20 * PI / 180;
    private final double per = 0.7;
    private final double per1 = 0.7;
    private final double per2 = 0.6;

    public CelayTree(Frame frm){
        graphics = frm.getGraphics();
        width = frm.getSize().width;
        height = frm.getSize().height;
    }

    void drawCelayTree(int n,
        double x0, double y0, double leng, double th)
    {
        if( n==0 ) return;

        double x1 = x0 + leng * Math.cos(th);
        double y1 = y0 + leng * Math.sin(th);

        drawLine(x0, y0, x1, y1);
    }
}
```

```
drawCelayTree( n - 1, x1, y1, per * leng, th + th1 );
drawCelayTree( n - 1, x1, y1, per * leng, th - th2 );
}

void drawLine( double x0, double y0, double x1, double y1 ){
graphics.drawLine( (int)x0, (int)y0, (int)x1, (int)y1 );
}

}
```

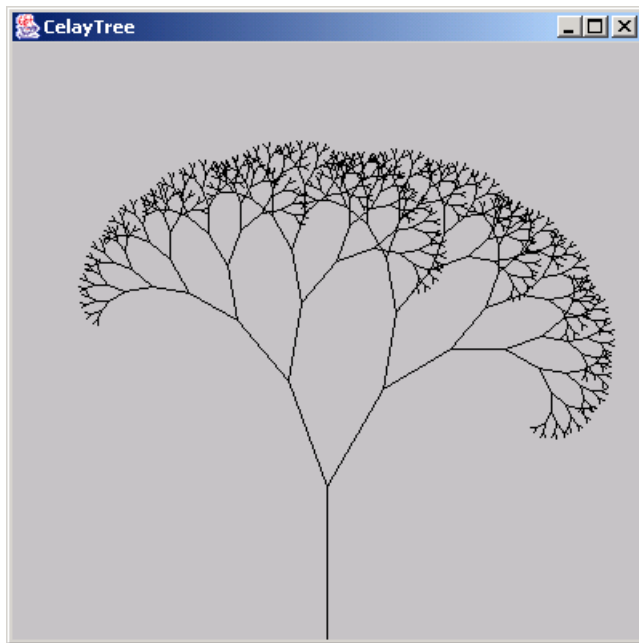


图 7-24 Cayley 树

习题

1. 在所有的 Java 系统类中, Object 类有什么特殊之处? 它在什么情况下使用?
2. 数据类型包装类与基本数据类型有什么关系?
3. Math 类用来实现什么功能? 设 x , y 是整型变量, d 是双精度型变量, 试书写表达式完成下面的操作:
 - (1) 求 x 的 y 次方;
 - (2) 求 x 和 y 的最小值;
 - (3) 求 d 取整后的结果;
 - (4) 求 d 的四舍五入后的结果;
 - (5) 求 $\text{atan}(d)$ 的数值。
4. Math.random()方法用来实现什么功能? 下面的语句起到什么作用?

```
(int)(Math.random()*6)+1
```

5. 编程生成 100 个 1~6 之间的随机数, 统计 1~6 之间的每个数出现的概率; 修改程序, 使之生成 1 000 个随机数并统计概率; 比较不同的结果并给出结论。

6. 什么是字符串? Java 中的字符串分为哪两类?

7. 编写 Applet 程序, 接受用户输入的一个字符串和一个字符, 把字符串中所有指定的字符删除后输出。

8. 编程判断一个字符串是否是回文。

9. String 类的 concat()方法与 StringBuffer 类的 append()方法都可以连接两个字符串, 它们之间有何不同?

10. 什么是递归方法? 递归方法有哪两个基本要素? 编写一个递归程序求一个一维数组所有元素的乘积。

11. 你了解几种排序算法? 它们各自有什么优缺点? 分别适合在什么情况下使用?

12. 向量与数组有何不同? 它们分别适合于什么场合?

13. Java 中有几种常用的集合类及其区别如何? 怎样获取集合中的各个元素?

14. 队列和堆栈各有什么特点?

15. 求解“鸡兔同笼问题”: 鸡和兔在一个笼里, 共有腿 100 条, 头 40 个, 问鸡兔各有几只。

16. 求解“百鸡问题”。已知公鸡每只 3 元, 母鸡每只 5 元, 每 3 只小鸡 1 元。用 100 元钱买 100 只鸡, 问每种鸡应各买多少。

17. 求四位的水仙花数。即满足这样条件的四位数: 各位数字的 4 次方和等于该数自身。

18. 求 1000 以内的“相亲数”。所谓相亲数是指这样的一对数: 甲数的约数之和等于乙数, 而乙数的约数之和等于甲数。

19. “哥德巴赫猜想”指出, 每个大于 6 的偶数, 都可以表示为两个素数的和。试用程序将 6~100 内的所有偶数都表示为两个素数的和。

20. 菲波那契(Fibonacci)数列的第一项是 0, 第二项是 1, 以后各项都是前两项的和, 试用递归算法和非递归算法各编写一个程序, 求菲波那契数列第 N 项的值。

21. 用迭代法编写程序用于求解立方根。

22. 用迭代法编写程序用于求解以下方程:

$$x^2 + \sin x + 1.0 = 0$$

在 -1 附近的一个根。

23. 作出对应于不同 c 值的 Julia 集的图。

$$c = -1 + 0.005i$$

$$c = -0.2 + 0.75i$$

$$c = 0.25 + 0.52i$$

$$c = 0.5 + 0.55i$$

24. 求“佩尔不定方程”的最小正整数解:

$$x^2 - Dy^2 = 1$$

其中, D 为某个给定的常数。令 $D=92$, 求其解。再令 $D=29$, 求其解。这里假定已知

其解都在 10 000 以内。

25. 从键盘上输入 10 个整数, 并放入一个一维数组中, 然后将其前 5 个元素与后 5 个元素对换, 即: 第 1 个元素与第 10 个元素互换, 第 2 个元素与第 9 个元素互换,……, 第 5 个元素与第 6 个元素互换。分别输出数组原来各元素的值和对换后各元素的值。

26. 有一个 $n \times m$ 的矩阵, 编写程序, 找出其中最大的那个元素所在的行和列, 并输出其值及行号和列号。

第 8 章 Java 的多线程

Java 语言的最大特色之一，就是在语言的级别支持多线程，本章介绍多线程机制及其应用。

8.1 线程及其创建

以往开发的程序大多是单线程的，即一个程序只有一条从头至尾的执行路线。然而，现实世界中的很多过程都具有多种途径同时运作，例如生物的进化，就是多种因素共同作用的结果。再如服务器，可能需要同时处理多个客户机的请求，等等。

多线程是指同时存在几个执行体，按几条不同的执行路线共同工作的情况。Java 语言的一个重要功能特点就是内置对多线程的支持，它使得编程人员可以很方便地开发出具有多线程功能。能同时处理多个任务的功能强大的应用程序。

8.1.1 Java 中的线程

1. 程序、进程与线程

程序是一段静态的代码，它是应用软件执行的蓝本。

进程是程序的一次动态执行过程，它对应从代码加载、执行到执行完毕的一个完整过程，这个过程也是进程本身从产生、发展到消亡的过程。作为执行蓝本的同一段程序，可以被多次加载到系统的不同内存区域分别执行，形成不同的进程。

线程是比进程更小的执行单位。一个进程在其执行过程中，可以产生多个线程，形成多条执行线索。每条线索，即每个线程也有它自身的产生、存在和消亡的过程，也是一个动态的概念。我们知道，每个进程都有一段专用的内存区域，并以 PCB（进程控制块）作为它存在的标志，与此不同的是，线程间可以共享相同的内存单元(包括代码与数据)，并利用这些共享单元来实现数据交换、实时通信与必要的同步操作。

2. Thread 类及 Runnable 接口

可以将一个线程理解成以下三个部分的组合，如图 8-1 所示。

- (1) CPU：虚拟的 CPU，专门用于执行该线程的任务。
- (2) Code：代码，即线程中要执行的指令，在程序中表现为特定的方法。
- (3) Data：数据，即线程中要处理的数据，在程序中表现为变量。

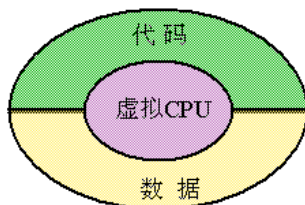


图 8-1 线程的构成

在 Java 中，由 `java.lang.Thread` 类来实现线程的概念，其中：

虚拟的 CPU，由 `java.lang.Thread` 类封装和虚拟；CPU 所执行的代码，在构造 `Thread` 类时，传递给 `Thread` 类对象。CPU 所处理的数据，在构造 `Thread` 类时，传递给 `Thread` 类对象。

为了构造一个 `Thread` 类，可以向 `Thread` 的构造函数传递一个 `Runnable` 对象，这个对象就是线程所需要的代码和数据的封装。

`Runnable` 对象是指实现了 `java.lang.Runnable` 接口的任何对象，`Runnable` 接口只有一个方法：

```
public void run();
```

这个 `run()` 方法实际上就是线程所要执行的代码。

要启动线程中的代码，只需要执行 `Thread` 类的 `start()` 方法，则系统会在可能的情况下去执行 `run()` 方法中所规定的代码。

8.1.2 创建线程对象的两种方法

创建线程对象，要传递代码与数据，而传递代码与数据有两种方法，一是通过继承 `Thread` 类，一是向 `Thread` 类传递一个 `Runnable` 对象。

1. 通过继承 `Thread` 类创建线程

从 `Thread` 派生出一个新类，在其中加入属性及方法，同时覆盖 `run()` 方法。当创建这样一个派生类的新对象后，使用 `start()` 方法，即可启动该线程。

例 8-1 `TestThread1.java` 简单的线程。

```
public class TestThread1 {  
    public static void main(String args[]){  
        Thread t = new MyThread();  
        t.start();  
    }  
}  
  
class MyThread extends Thread {  
    public void run() {
```

```
        for(int i=0;i<100;i++) {  
            System.out.print ( " " + i);  
        }  
    }  
}
```

在程序中通过继承，定义了 `MyThread` 类。用 `new` 创建了一个线程实例，用 `start()` 方法进行启动，程序结果如图 8-2 所示。

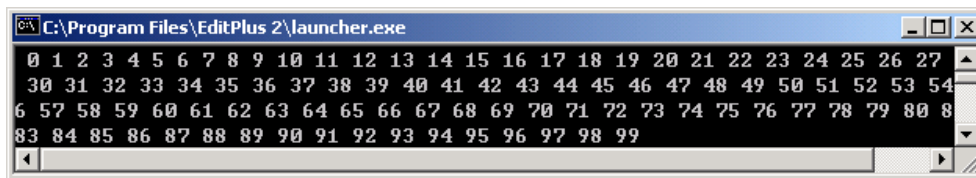


图 8-2 通过继承 `Thread` 类创建线程

2. 通过向 `Thread()` 构造方法传递 `Runnable` 对象来创建线程

通过向 `Thread()` 构造方法传递 `Runnable` 对象来创建线程，是第二种方法。

例 8-2 `TestThread2.java` 实现 `Runnalbe`。

```
public class TestThread2 {  
    public static void main(String args[]) {  
        MyTask mytask = new MyTask(100);  
        Thread thread = new Thread(mytask);  
        thread.start();  
    }  
}  
  
class MyTask implements Runnable {  
    private int n;  
    public MyTask(int n){  
        this.n = n;  
    }  
    public void run() {  
        for(int i=0; i<n; i++) {  
            System.out.println(" " + i);  
            try{  
                Thread.sleep(500);  
            }catch( InterruptedException e ){}  
        }  
    }  
}
```

```
}  
}
```

运行结果如图 8-3 所示。

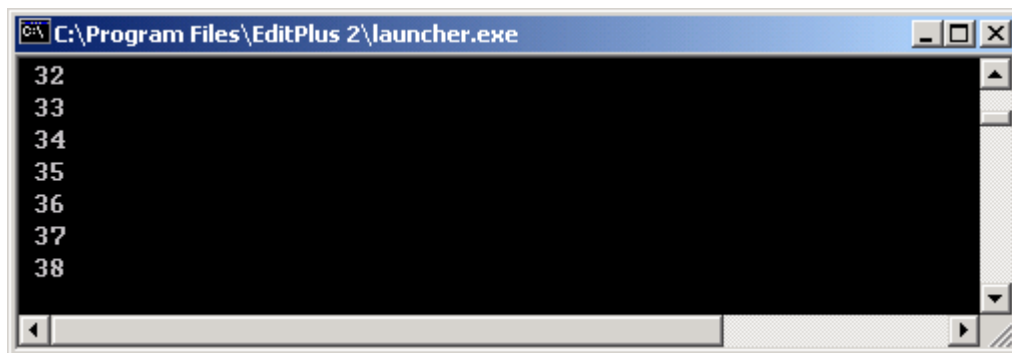


图 8-3 实现 Runnable

该程序定义了一个类 `MyTask`，它实现了 `Runnable` 接口，也就是说，它含有 `public void run()` 方法。在创建线程时，先创建一个 `MyTask` 对象，再将此对象作为 `Thread` 构造方法的参数。

注意：(1) `run()` 方法规定了线程要执行的任务，但一般不是直接调用 `run()` 方法，而是通过线程的 `start()` 来启动线程。

(2) 程序中调用了 `Thread` 类的一个 `static` 方法：`sleep()`，它表示线程等待（休眠）一定的时间（单位为毫秒），但休眠用的时间一般不会严格等于所给定的时间。休眠结束后，系统发出一个 `InterruptedException` 异常，来告诉休眠的结束。

3. 两种方法的比较

直接继承 `Thread` 类，这种方法的特点是：编写简单，可以直接操纵线程；但缺点也是明显的，因为若继承 `Thread` 类，就不能再从其他类继承。

使用 `Runnable` 接口，这种方法的特点是：可以将 `Thread` 类与所要处理的任务的类分开，形成清晰的模型；还可以从其他类继承。

另外，若直接继承 `Thread` 类，在类中 `this` 即指当前线程；若使用 `Runnable` 接口，要在类中获得当前线程，必须使用 `Thread.currentThread()` 方法。

8.1.3 多线程

一个程序中可以有多个线程，多个线程可以同时运行。多个线程中可以共享代码及数据。如在例 8-3 中，共建了 7 个线程，前 6 个线程共享相同的代码，其中 `t1`、`t2`、`t3` 共享了对象 `r1`，`t4`、`t5`、`t6` 共享了对象 `r2`；程序中还建立了另一个线程，该线程无穷循环地显示当前的时间。其中，使用了 `SimpleDateFormat` 类及 `Date` 类来获得时间。从该程序中可以看出几个线程同时运行的情况。

例 8-3 TestThread3.java 多线程。

```
import java.util.*;
import java.text.*;
public class TestThread3 {
    public static void main(String args[]) {
        Runner r1 = new Runner(1);
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r1);
        Thread t3 = new Thread(r1);
        Runner r2 = new Runner(2);
        Thread t4 = new Thread(r2);
        Thread t5 = new Thread(r2);
        Thread t6 = new Thread(r2);
        Timer timer = new Timer();
        Thread t7 = new Thread(timer);
        t1.start();
        t2.start();
        t3.start();
        t4.start();
        t5.start();
        t6.start();
        t7.start();
    }
}

class Runner implements Runnable {
    int id;
    Runner(int id){
        this.id = id;
    }
    public void run() {
        int i=0;
        while( true ){
            i++;
            System.out.println("ID: " + id + " No. " + i);
            try{ Thread.sleep(300); } catch( InterruptedException
            e ){}
        }
    }
}
```

```
    }  
}  
  
class Timer implements Runnable {  
    public void run(){  
        while(true){  
            System.out.println( new SimpleDateFormat().  
                format( new Date()));  
            try{ Thread.sleep(1000); } catch( InterruptedException  
                e ){}  
        }  
    }  
}
```

运行结果如图 8-4 所示。

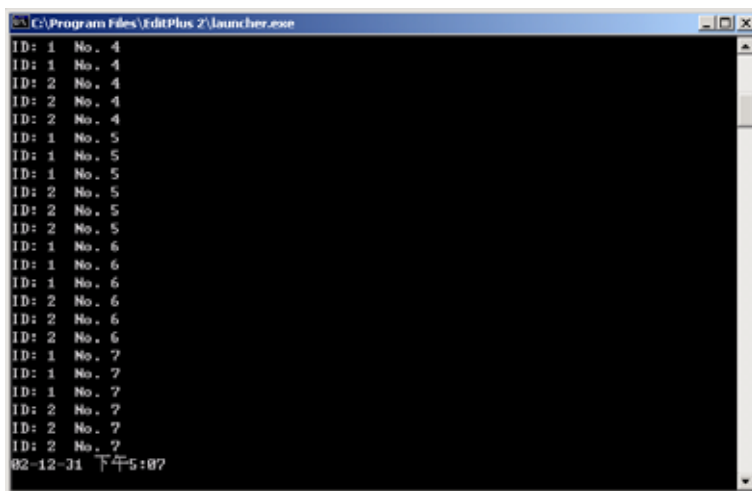


图 8-4 多线程

8.1.4 应用举例

下面举一个实际运用的例子。程序中有多个线程，每个线程在不同的时间在不同的地方画一些图形。

例 8-4 ThreadDraw.java 多线程绘图。

```
import java.awt.*;  
import java.applet.*;  
import java.awt.event.*;
```

```
public class ThreadDraw extends Applet
{
    MovingShape [] shapes;

    public void init()
    {
        //{{INIT_CONTROLS
        setLayout(null);
        setSize(426,266);
        //}}

        shapes = new MovingShape[ 10 ];
        for( int i=0; i<shapes.length; i++ )
            shapes[i] = new MovingShape(this);
    }

    public void start()
    {
        for( int i=0; i<shapes.length; i++ );
        // shapes[i].start(); // it started when it's constructed
        super.start();
    }

    public void stop()
    {
        for( int i=0; i<shapes.length; i++ )
            shapes[i].suspend(); // deprecated
        super.stop();
    }

    public void destroy()
    {
        for( int i=0; i<shapes.length; i++ )
            shapes[i].stop(); // deprecated
        super.destroy();
    }

    public static void main(String [] args) // 加入main,使之能当
    Application应用
```



```

    {
        Frame f = new Frame();
        f.setSize(450,300 );
        ThreadDraw p = new ThreadDraw();
        f.add ( p );
        f.setVisible( true );
        p.init();
        p.start();
        f.addWindowListener( new WindowAdapter(){
            public void windowClosing(WindowEvent e)
            { System.exit(0); }
        });
    }
    //{{DECLARE_CONTROLS
    //}}
}

class MovingShape extends Thread
{

    boolean bContinue = false;
    private int size=100;
    private int speed=10;
    private Color color;
    private int type;
    private int x,y,w,h,dx,dy;
    protected java.awt.Component app;

    MovingShape( java.awt.Component app )
    {
        this.app = app;
        x = (int)(Math.random() * app.getSize().width);
        y = (int)(Math.random() * app.getSize().height);
        w = (int)(Math.random() * size );
        h = (int)(Math.random() * size );
        dx = (int)(Math.random() * speed );
        dy = (int)(Math.random() * speed );
        color = new Color (
            (int)(Math.random()*128+128),

```

```
(int)(Math.random()*128+128),
(int)(Math.random()*128+128) );
type = (int)(Math.random() * 3 );
bContinue = true;
this.start(); // 注意,线程刚构造出来,这里就立即进行启动
}

public void run()
{
    while( true ){
        x += dx;
        y += dy;
        if( x<0 || x+w>app.getSize().width ) dx = -dx;
        if( y<0 || y+h>app.getSize().height ) dy = -dy;
        Graphics g = app.getGraphics();

        switch( type ){

        case 0:
            g.setColor(color);
            g.fillRect( x,y,w,h );
            g.setColor( Color.black );
            g.drawRect( x,y,w,h );
            break;

        case 1:
            g.setColor(color);
            g.fillOval( x,y,w,h );
            g.setColor( Color.black );
            g.drawOval( x,y,w,h );
            break;

        case 2:
            g.setColor(color);
            g.fillRoundRect( x,y,w,h,w/5,h/5 );
            g.setColor( Color.black );
            g.drawRoundRect( x,y,w,h,w/5,h/5 );
            break;

        }

        // System.out.println(x+", "+y+", "+w+", "+h+": "+type+", "
        +dx+", " +dy);
    }
}
```

```
        try{ Thread.sleep(130); } catch( InterruptedException e ){}  
    }  
}  
  
}
```

运行结果如图 8-5 所示。

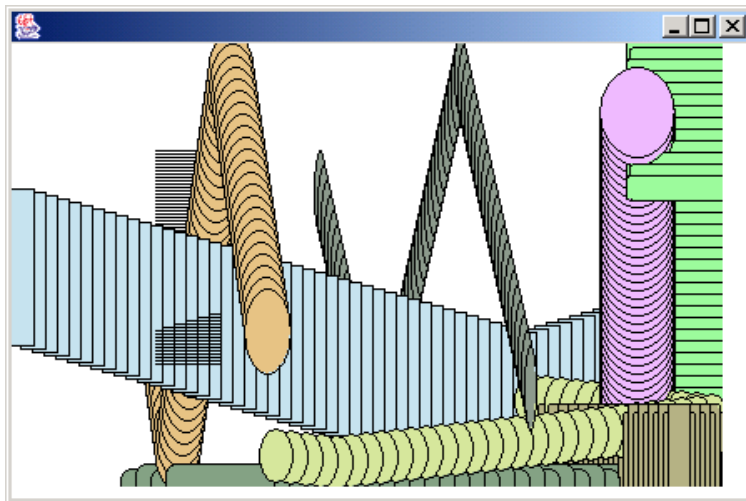


图 8-5 多线程绘图

8.2 线程的调度

8.2.1 线程的状态与生命周期

每个 Java 程序都有一个默认的主线程，对于 Application，主线程是 main()方法执行的线索；对于 Applet，主线程指挥浏览器加载并执行 Java Applet。要想实现多线程，必须在主线程中创建新的线程对象。Java 语言使用 Thread 类及其子类的对象来表示线程，新建的线程在它的一个完整生命周期中通常要经历如下的五种状态。

(1) 新建(New)。当一个 Thread 类或其子类的对象被声明并创建时，新生的线程对象处于新建状态。此时它已经有了相应的内存空间和其他资源，并已被初始化。

(2) 就绪(Runnable)。处于新建状态的线程被启动后，将进入线程队列排队等待 CPU 时间片，此时它已经具备了运行的条件。一旦轮到它来享用 CPU 资源时，就可以脱离创建它的主线程独立，开始自己的生命周期。另外，原来处于阻塞状态的线程被解除阻塞后也将进入就绪状态。

(3) 运行(Running)。当就绪状态的线程被调度并获得处理器(CPU)资源时，便进入运行状态。每一个 Thread 类及其子类的对象都有一个重要的 run()方法，当线程对象被调度执行时，它将自动调用本对象的 run()方法，从第一句开始顺次执行。run()方法定义了这一类线

程的操作和功能。

(4) 阻塞(Blocked)。一个正在执行的线程在某些特殊情况下，如被人为挂起或需要执行费时的输入、输出操作时，将让出 CPU 资源并暂时中止自己的执行，进入阻塞状态。阻塞时，它不能进入排队队列。只有当引起阻塞的原因被消除后，线程才可以转入就绪状态，重新进到线程队列中排队等待 CPU 资源，以便从原来中止处开始继续运行。

(5) 终止(Dead)。处于死亡状态(终止状态)的线程不具有继续运行的能力。线程死亡的原因有两个：一个是正常运行的线程完成了它的全部工作，即执行完了 run() 方法的最后一个语句并退出；另一个是线程被提前强制性地终止，如通过执行 destroy() 终止线程。

线程在各个状态之间的转化及线程生命周期的演进（如图 8-6 所示）是由系统运行的状况、同时存在的其他线程和线程本身的算法所共同决定的。在创建和使用线程时应注意利用线程的方法宏观地控制这个过程。

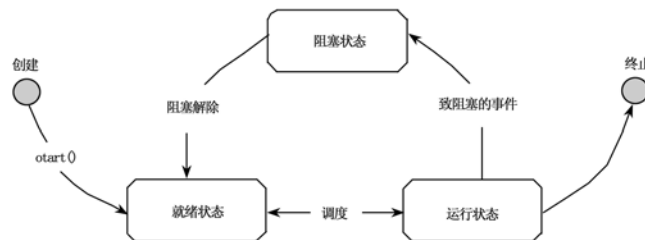


图 8-6 线程的状态

8.2.2 线程调度与优先级

处于就绪状态的线程首先进入就绪队列排队等候处理器资源。同一时间在就绪队列中的线程可能有多个，它们各自任务的轻重缓急程度不同。为了体现上述差别，使工作安排得更加合理，多线程系统会给每个线程自动分配一个线程的优先级(Priority)，任务较紧急重要的线程，其优先级就较高，相反则较低。

同时处于就绪状态的线程，优先级高的，有优先被调度的权利。相同优先级，一般遵循先到先服务的原则。但是，这种情况不一定十分严格，因为实际的执行过程，与 Java 虚拟机的实现情况有关。

Thread 类有三个有关线程优先级的静态常量：MIN_PRIORITY，MAX_PRIORITY，NORM_PRIORITY。其中，MIN_PRIORITY 代表最小优先级，通常为 1；MAX_PRIORITY 代表最高优先级，通常为 10；NORM_PRIORITY 代表普通优先级，默认数值为 5。

对应一个新建线程，系统会遵循如下的原则为其指定优先级。

(1) 新建线程将继承创建它的父线程的优先级。父线程是指执行创建新线程对象语句的线程，它可能是程序的主线程，也可能是某一个用户自定义的线程。

(2) 一般情况下，主线程具有普通优先级。

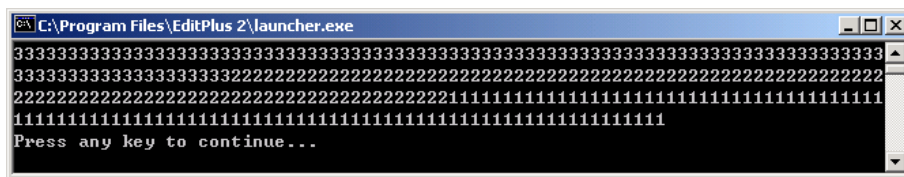
另外，用户可以通过调用 Thread 类的方法 setPriority() 来修改系统自动设定的线程优先级，使之符合程序的特定需要。

例 8-5 TestThreadPriorit.java 设定不同的优先级。读者可以发现，设定优先级会影响显示的结果。

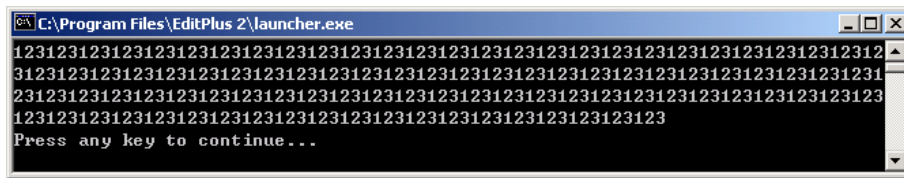
```
public class TestThreadPriority {
    public static void main(String args[]) {
        Thread t1 = new Thread( new Runner(1) );
        Thread t2 = new Thread( new Runner(2) );
        Thread t3 = new Thread( new Runner(3) );
        t1.setPriority( Thread.MIN_PRIORITY );
        t2.setPriority( Thread.NORM_PRIORITY );
        t3.setPriority( Thread.MAX_PRIORITY );
        t1.start();
        t2.start();
        t3.start();
    }
}

class Runner implements Runnable {
    int id;
    Runner(int id){
        this.id = id;
    }
    public void run() {
        for(int i=0; i<100; i++ ){
            if( i % 100 == 0 ) System.out.print("\r");
            Thread.currentThread().yield();
            System.out.print(id);
        }
    }
}
```

运行结果如图 8-7 所示。



(a) 按程序中设定不同的优先级



(b) 若三个线程具有相同的优先级

图 8-7 不同优先级对线程的影响

8.2.3 对线程的基本控制

1. 结束线程

在 JDK1.0 中, 可以用 Thread 对象的 stop() 方法来停止线程。但在以后的 JDK 版本中, stop() 已经不再主张用了。现在一般采取给线程设定一个标记变量的方法来决定线程是否应该终止。

例 8-6 ThreadTerminateByFlag.java 通过标记变量来决定是否结束线程。

```
import java.util.*;

public class ThreadTerminateByFlag {
    public static void main(String args[]) {
        Timer timer = new Timer();
        Thread thread = new Thread( timer );
        thread.setName( "Timer" );
        thread.start();
        for( int i=0; i<100; i++ ){
            System.out.print("\r" + i );
            try{
                Thread.sleep(100);
            }catch( InterruptedException e ){}
        }
        timer.stopRun();
    }
}

class Timer implements Runnable {
    boolean flg = true;
    public void run() {
        while(flg){
```

```

        System.out.print( "\r\t" + new Date() + "..." );
        try{
            Thread.sleep(1000);
        }catch( InterruptedException e ){}
    }
    System.out.println( "\n" + Thread.currentThread().
        getName() + " Stop" );
}
public void stopRun(){
    flg = false;
}
}
}

```

本例中，在这里的线程中，通过设定变量 flg 为 false 来结束循环。

2. 设定线程的优先级

设定线程的优先级可以使用 Thread 对象的 setPriority(int priority)方法。

3. 暂时阻止线程的执行

在 JDK1.0 中，可以用 Thread 对象的 suspend()方法来暂时阻止线程的执行，用 resume()方法来恢复线程的执行，不过这两个方法已经不主张用了。目前常用的方法如下。

(1) sleep()方法。用 Thread.sleep(long millisecond)来挂起线程的执行。sleep()方法可以给优先级较低的线程以执行的机会。

其基本使用方法是：

```

try{
    Thread.sleep( 1000 );
} catch( InterruptedException e ){
    //...
}

```

(2) join()方法。调用某 Thread 对象的 join()方法，可以将一个线程加入到本线程中，本线程的执行会等待另一线程执行完毕。

join()方法，可以不带参数。可以带上一个 long，表示等待的最长时间。

其基本使用方法是：

```

Thread t; // t 是另一线程
try{
    t.join();
} catch( InterruptedException e ){
    //...
}

```

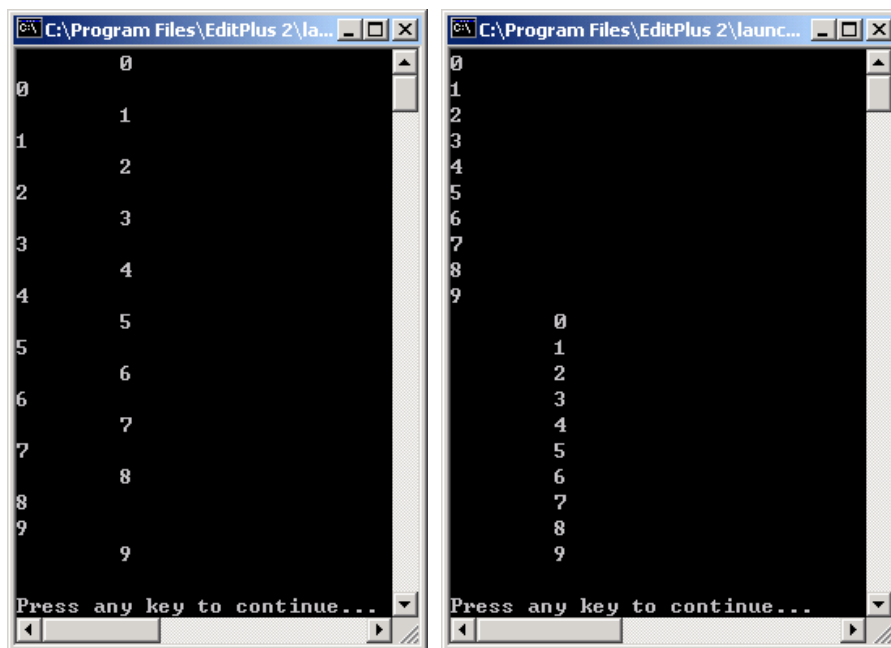
例 8-7 ThreadJoin.java 使用 join。

```
import java.util.*;

public class ThreadJoin {
    public static void main(String args[]) {
        Runner r = new Runner();
        Thread thread = new Thread( r );
        thread.start();
        //try{
            //thread.join();
        //}catch(InterruptedException e){}
        for( int i=0; i<10; i++ ){
            System.out.println("\t" + i );
            try{
                Thread.sleep(100);
            }catch( InterruptedException e ){}
        }
    }

    class Runner implements Runnable {
        public void run() {
            for( int i=0; i<10; i++ ){
                System.out.println( i );
                try{
                    Thread.sleep(100);
                }catch( InterruptedException e ){}
            }
        }
    }
}
```

程序结果如图 8-8 所示。程序中若没有用 join(), 则两个线程同时运行; 若将 join()加上, 则主程序的线程会等待 Runner 的线程执行完毕后再进行。



(a) 不使用 join()

(b) 使用 join()

图 8-8 使用 join 对程序的影响

(3) yield()方法。Thread 对象的 yield()方法，可以给其他线程以执行的机会。如果没有其他可运行的线程，则该方法不产生任何作用。

8.3 线程的同步与共享

8.3.1 synchronized 关键字

前面所提到的线程都是独立的，而且异步执行，也就是说每个线程都包含了运行时所需要的数据或方法，而不需要外部的资源或方法，也不必关心其他线程的状态或行为。但是经常有一些同时运行的线程需要共享数据，例如一个线程向文件写数据，而同时另一个线程从同一文件中读取数据，因此就必须考虑其他线程的状态与行为，这时就需要实现同步来得到预期结果。

例 8-8 SyncCounter1.java 两线程共享同一资源。

```
class SyncCounter1
{
    public static void main(String[] args){
        Num num = new Num();
        Thread counter1 = new Counter(num);
        Thread counter2 = new Counter(num);
```

```
        for( int i=0; i<10; i++ ){
            num.testEquals();
            try{
                Thread.sleep(100);
            }catch(InterruptedException e){
            }
        }
    }
}

class Num
{
    private int x=0;
    private int y=0;
    void increase(){
        x++;
        y++;
    }
    void testEquals(){
        System.out.println( x + "," + y + " : " + (x==y));
    }
}

class Counter extends Thread
{
    private Num num;
    Counter( Num num ){
        this.num = num;
        this.start();
    }
    public void run(){
        while(true){
            num.increase();
        }
    }
}
```

程序的运行结果如图 8-9 所示。

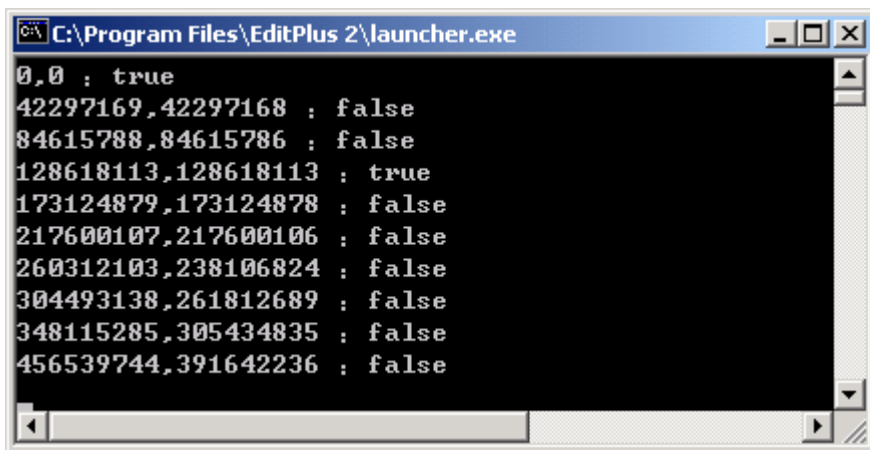


图 8-9 两线程共享同一资源带来的问题

该例中，Counter 线程操作对象 num，线程中调用 increase()方法，使 x，y 同时增加。在 main()中调用 testEquals()方法以检查 x，y 是否相等。从图中的结果可以看出大部分时间 x，y 并不相等。原因何在呢？

在这里，问题的关键在于有两个线程同时操作同一个对象。在线程执行时，可以出现这样的情况，当一个线程执行了 x++语句尚未执行 y++语句时，系统调度到另一个线程执行 x++及 y++，这时就会出现 x 多加一次的情况。由于转换线程的调度是不能预料的，所以出现了 x，y 不相等的情况。

这种由于多线程同时操作一个对象引起的现象，称为该对象不是线程安全的。为了多线程机制能够正常运转，需要采取一些措施来防止两个线程访问相同的资源的冲突，特别是在关键的时期。为防止出现这样的冲突，只需在线程使用一个资源时为其加锁即可。访问资源的第一个线程加上锁以后，其他线程便不能再使用那个资源，除非第一个线程被解锁。

对一种特殊的资源——对象中的内存——Java 提供了内建的机制来防止它们的冲突。这就是对于相关的方法使用关键字 synchronized。在任何时刻，只能有一个线程调用特定对象的一个 synchronized 方法（尽管那个线程可以调用多个对象的同步方法）。将上例中的 increase()、testEquals()设定成 synchronized 方法：

```
synchronized void increase(){
    x++;
    y++;
}

synchronized void testEquals(){
    System.out.println( x + "," + y + " : " + (x==y));
}
```

这样，程序的运行结果就是正确的了。如图 8-10 所示。

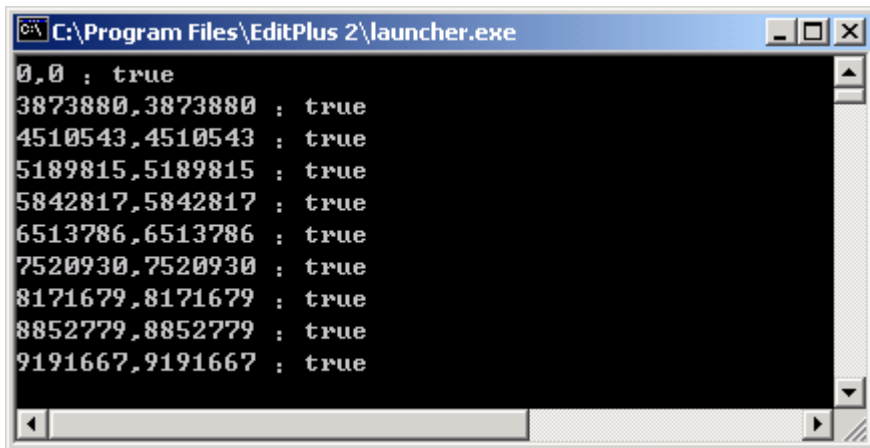


图 8-10 加上 synchronized 后的结果

每个对象都包含了一把锁（也叫做“监视器”），它自动成为对象的一部分（不必为此写任何特殊的代码）。调用任何 `synchronized` 方法时，对象就会被锁定，不可再调用那个对象的其他任何 `synchronized` 方法，除非第一个方法完成了自己的工作，并解除锁定。在上面的例子中，如果为一个对象调用 `increase()`，便不能再为同样的对象调用 `increase()` 或 `testEquals()`，除非 `increase()` 完成并解除锁定。因此，一个特定对象的所有 `synchronized` 方法都共享着一把锁，而且这把锁能防止多个方法对通用内存同时进行写操作（比如同时有多个线程）。

每个类也有自己的一把锁（作为类的 `Class` 对象的一部分），所以 `synchronized static` 方法可在一个类的范围内被相互间锁定起来，防止与 `static` 数据的接触。注意如果想保护其他某些资源不被多个线程同时访问，可以强制通过 `synchronized` 方法访问那些资源。

由于通常将数据元素设为从属于 `private`（私有），然后只通过方法访问那些内存，所以只需将一个特定的方法设为 `synchronized`（同步的），便可有效地防止冲突。

但对于非 `private` 域，则不能保证这一点（因为它可能直接访问，而不通过 `synchronized` 方法）。

`synchronized` 关键字除了可用在方法名前面外，还可以用在方法的内部，其格式如下：

```
synchronized( 对象 ){  
    ... 语句  
}
```

表示在这语句执行期间，给某对象上锁。这里，对象可以用 `this`。

8.3.2 线程间的同步控制

线程间的同步控制是多线程系统中的一个重要问题。下面以“生产者—消费者问题”这个一般性的模型来讨论关于线程的同步问题。

系统中使用某类资源的线程称为消费者，产生或释放同类资源的线程称为生产者，例如在一个 Java 的应用程序中，生产者线程向文件中写数据，消费者从文件中读数据，这样，

在这个程序中同时运行的两个线程共享同一个文件资源。

为了更简化讨论，在下面的例子中，生产者产生从 0~9 的整数，将它们存储在名为“CubbyHole”的对象中并打印出来。然后调用 sleep()方法使生产者线程在一个随机产生的 0~100 秒的时间段内休息。

例 8-9 生产者消费者问题。

```
class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        for (int i = 0; i <10; i++) {
            cubbyhole.put(i);
            System.out.println("Producer #" + this.number +
                " put: " + i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) {
            }
        }
    }
}
```

消费者线程则不断地从 CubbyHole 对象中取这些整数：

```
class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
```

```

        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumer #" + this.number +
                               " got: " + value);
        }
    }
}

```

在这里，假定 **CubbyHole** 的定义如下：

```

class CubbyHole
{
    private int seq;
    public synchronized int get() {
        return seq;
    }
    public synchronized void put(int value) {
        seq = value;
    }
}

```

在例 8-9 中，生产者与消费者通过 **CubbyHole** 对象来共享数据。但是我们发现不论是生产者线程还是消费者线程都无法保证生产者每产生一个数据，消费者就能及时取得该数据并且只取一次。通过 **CubbyHole** 中的 **put()** 和 **get()** 方法只能保证较低层次的同步，让我们来看一看可能发生的情况。

第一种情况是如果生产者比消费者快，那么在消费者来不及取前一个数据之前，生产者又产生了新的数据，于是，消费者很可能就会跳过前一个数据，这样就会有下面的结果：

```

. . .
Consumer #1 got: 3
Producer #1 put: 4
Producer #1 put: 5
Consumer #1 got: 5
. . .

```

第二种情况则反之，当消费者比生产者快时，消费者可能两次取同一个数据，可能会产生下面的输出结果：

```

. . .
Producer #1 put: 4
Consumer #1 got: 4
Consumer #1 got: 4
Producer #1 put: 5
. . .

```

上面两种输出结果都不是我们所希望的那样：生产者写一个数，消费者就取这个数。像这种异步执行的多线程，由于希望同时进入同一对象中而发生错误结果的情况称为竞争条件(race condition)。

为了避免上述情况发生，就必须使生产者线程向 **CubbyHole** 中存储数据和消费者线程从 **CubbyHole** 中取数据同步起来。为达到这一目的，程序中采用了两种结构：监视器，**notify()** 和 **wait()** 方法。下面分别介绍。

(1) 监视器 (monitors)

像 **CubbyHole** 这样被多个需同步的线程共享的对象称为条件变量(condition variable); 这里的条件变量就相当于一个监视器，Java 语言正是通过使用监视器来实现同步。一个 **monitor** 就相当于一个只能容纳一个线程的小盒子，在一段特定时间内只能有一个线程进入 **monitor**，而其他的线程将被暂停直到当前线程离开这个小盒子。

生产者—消费者问题中的类 **CubbyHole** 中提供了两个同步的方法：**put()**方法用来改变 **CubbyHole** 中的数据，**get()**方法则用来取数据。这样，系统就把每个 **CubbyHole** 类的实例与一个 **monitor** 相对应。

例 8-10 改进后的 **CubbyHole** 类。

```
class CubbyHole {
    private int seq;
    private boolean available = false;

    public synchronized int get() {
        while (available == false) {
            try {
                wait(); // waits for notify() call from Producer
            } catch (InterruptedException e) {
            }
        }
        available = false;
        notify();
        return seq;
    }

    public synchronized void put(int value) {
        while (available == true) {
            try {
                wait(); // waits for notify() call from consumer
            } catch (InterruptedException e) {
            }
        }
    }
}
```

```
        seq = value;
        available = true;
        notify();
    }
}
```

类 `CubbyHole` 有两个变量：`seq` 是 `CubbyHole` 中当前内容，布尔变量 `available` 指示当前内容是否可以取出。只有当 `available` 是 `true` 时，消费者才能取数据。为实现两线程同步，必须保证：

其一，消费者接收数据的前提是 `available` 为 `true`，即数据单元内容不空；

其二，生产者发送数据的前提是数据单元内容为空。

在例 8-10 中，通过调用对象的 `notify()` 和 `wait()` 方法来保证 `CubbyHole` 中的每个数据只被读取一次。

(2) `notify()` 方法。

在 `get()` 方法返回前调用 `notify()`，它用来选择并唤醒等候进入监视器的线程。如果消费者线程调用了 `get()` 方法，那么在整个执行过程中它将占据 `monitor`，在 `get()` 方法结束前调用 `notify()` 方法来唤醒处于等待状态的生产者线程。这样，生产者线程就占据了 `monitor` 并继续执行。

`put()` 方法与 `get()` 相似，在一个线程结束前唤醒另一个。

与 `notify()` 方法类似的还有 `notifyAll()` 方法，不同的是它唤醒所有等待的线程，这些线程中的一个经过竞争进入 `monitor`，其他的继续等待。

(3) `wait()` 方法。

`wait()` 方法使当前线程处于等待状态，直到别的线程调用 `notify()` 方法来通知它。`get()` 方法包含了一个 `while` 循环，结束条件是 `available` 为 `true`。如果 `available` 为 `false` 的话，消费者就知道生产者还没有产生新的数据，将继续等待。`while()` 循环中调用 `wait()` 方法，等待生产者线程发送消息。当 `put()` 方法调用 `notify()` 时，消费者线程被唤醒并继续 `while()` 循环。`put()` 方法中的 `wait()` 方法作用相同。

可能出现的问题是：在 `get()` 方法的开始，`available` 为 `false`，消费者线程必须等待生产者发送数据，那么，如果消费者占据 `monitor`，生产者如何发送呢？同样，若数据未被消费者取走，而生产者占据 `monitor`，消费者怎样获得数据呢？在 Java 中是这样处理的：当一个线程进入等待状态后，`monitor` 就会自动释放，而当它被唤醒后，该线程又占据 `monitor`。这样就使等待的线程有机会进入 `monitor`。

下面来看一看主程序及输出结果：

```
class ProducerConsumerTest {
    public static void main(String args[]) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);
        p1.start();
        c1.start();
    }
}
```



```
    }  
}
```

输出结果如下：

```
Producer    #1    put: 0  
Consumer    #1    got: 0  
Producer    #1    put: 1  
Consumer    #1    got: 1  
Producer    #1    put: 2  
Consumer    #1    got: 2  
Producer    #1    put: 3  
Consumer    #1    got: 3  
Producer    #1    put: 4  
Consumer    #1    got: 4  
Producer    #1    put: 5  
Consumer    #1    got: 5  
Producer    #1    put: 6  
Consumer    #1    got: 6  
Producer    #1    put: 7  
Consumer    #1    got: 7  
Producer    #1    put: 8  
Consumer    #1    got: 8  
Producer    #1    put: 9  
Consumer    #1    got: 9
```

由输出结果，可以看到生产者和消费者两个线程实现了同步。

注意：由于系统资源有限，程序中多个线程互相等待对方资源，而在得到对方资源前不会释放自己的资源，造成都想得到资源而又都得不到，线程不能继续运行，这就是死锁问题。有关死锁等问题，读者可以参考有关书籍。

习题

1. 程序中怎样创建线程？
2. 程序中怎样控制线程？
3. 多线程之间怎样进行同步？
4. 编写一个程序，用一个线程显示时间，一个线程用来计算（如判断一个大数是否是质数），当质数计算完毕后，停止时间的显示。

第9章 流、文件及基于文本的应用

与外部设备和其他计算机进行交流的输入、输出操作，尤其是对磁盘的文件操作，是计算机程序重要而必备的功能。本章中介绍流式输入与输出及文件处理，并介绍基于文本的应用的程序中的一些问题。

9.1 流式输入与输出

为进行数据的输入、输出操作，Java 中把不同的输入、输出源(键盘、文件、网络连接等)抽象表述为“流”(stream)。有两种基本的流：输入流和输出流。

(1) 输入流：只能从中读取数据，而不能向其写出数据。

(2) 输出流：只能向其写出数据，而不能从中读取数据。

流实际上指在计算机的输入与输出之间运动的数据的序列，流序列中的数据既可以是未经加工的原始二进制数据，也可以是经一定编码处理后符合某种格式规定的特定数据，如字符流序列、数字流序列等。

java.io 包中定义了多个类（抽象的或者具体的）来处理不同性质的输入、输出流。

9.1.1 字节流与字符流

按处理数据的类型，流可以分为字节流与字符流，它们处理的信息的基本单位分别是字节(byte)与字符(char)。输入字节流的类为 `InputStream`，输出字节流为 `OutputStream`，输入字符流为 `Reader`，输出字符流为 `Writer`，如表 9-1 所示。这四个类是抽象类，其他的输入输出流类都是它们的子类。

表 9-1 字节流与字符流

	字节流	字符流
输入	<code>InputStream</code>	<code>Reader</code>
输出	<code>OutputStream</code>	<code>Writer</code>

1. `InputStream` 类

`InputStream` 类最重要的方法是读数据的 `read()` 方法。`read()` 方法功能是逐字节地以二进制的原始方式读入数据，它有三种形式：

☞ `public int read();`

☞ `public int read(byte b[]);`

☞ `public int read(byte[] b, int off, int len);`

第一个 `read()` 方法从输入流的当前位置处读入一个字节(8 位)的二进制数据，然后以此数据为低位字节，配上一个全零字节合成为一个 16 位的整型量(0~255)后返回给调用此方法的语句。如果输入流的当前位置没有数据，则返回-1。

第二个和第三个 `read()` 方法从输入流的当前位置处连续读入多个字节保存在参数指定

的字节数组 `b[]` 中，同时返回所读到的字节的数目。第三个方法，`len` 指定要读取的字节个数，`off` 指定在数组的存放位置。

在流的操作过程中，都有一个“当前位置”的概念。每个流都有一个位置指针，它在流刚被创建时产生并指向流的第一个数据，以后的每次读操作都是在当前位置指针处执行；伴随着流操作的执行，位置指针自动后移，指向下一个未被读取的数据。位置指针决定了 `read()` 方法将在输入流中读到哪个数据。

`InputStream` 方法还有如下几种：

- ☞ `public long skip(long n);` // 使位置指针从当前位置向后跳过 n 个字节
- ☞ `public void mark();` // 在当前位置指针处做一个标记
- ☞ `public void reset();` // 将位置指针返回到标记的位置
- ☞ `public boolean markSupported();` // 是否支持标记操作
- ☞ `public int available();` // 流中有多少字节可读
- ☞ `public void close();` // 关闭流，并断开外设数据源的连接，释放占用的系统资源

2 . `OutputStream` 类

`OutputStream` 类的重要方法是 `write()`，它的功能是将字节写入流中，`write()` 方法有三种形式：

- ☞ `public void write (int b);` // 将参数 `b` 的低位字节写入到输出流
- ☞ `public void write (byte b[]);` // 将字节数组 `b[]` 中的全部字节顺序写入到输出流
- ☞ `public void write(byte[] b, int off, int len);` // 将字节数组 `b[]` 中从 `off` 开始的 `len` 个字节写入到流中

`Output` 的另外两个方法是 `flush()` 及 `close()`。

- ☞ `public void flush ();`
- ☞ `public void close();`

`flush()` 方法刷新缓冲区的内容。对于缓冲流式输出来说，`write()` 方法所写的数据并没有直接传到与输出流相连的外部设备上，而是先暂时存放在流的缓冲区中，等到缓冲区中的数据积累到一定的数量，再统一执行一次向外部设备的写操作把它们全部写到外部设备上。这样处理可以降低计算机对外部设备的读写次数，大大提高系统的效率。但是在某些情况下，缓冲区中的数据不满时就需要将它写到外部设备上，此时应使用强制清空缓冲区并执行外部设备写操作的 `flush()` 方法。

`Close()` 方法关闭输出流。当输出操作完毕时，应调用下面的方法来关闭输出流与外设的连接并释放所占用的系统资源。

3 . `Reader` 类

`Reader` 类与 `InputStream` 类相似，都是输入流，但差别在于 `Reader` 类读入的是字符 (`char`)，而不是字节。

`Reader` 的重要方法是 `read()`，有三种形式：

- ☞ `public int read();`
- ☞ `public int read(char b[]);`

☞ `public int read(char[] b, int off, int len);`

其中，第一个方法，将读入的字符转为整数返回。若不能读到字符，返回-1。后两个方法读入字符放入数组中。

Reader 的方法还有：

☞ `public long skip(long n);` // 使位置指针从当前位置向后跳过 *n* 个字节

☞ `public void mark();` // 在当前位置指针处做一个标记

☞ `public void reset();` // 将位置指针返回到标记的位置

☞ `public boolean markSupported();` // 是否支持 mark 操作

☞ `public int available();` // 流中有多少字节可读

☞ `public void close();` // 关闭流，并断开与对外部设备数据源的连接，释放占用的系统资源。

4. Writer 类

Writer 类与 OutputStream 类相似，都是输出流，但差别在于 Writer 类写出的是字符(char)，而不是字节。Writer 的方法有：

☞ `public void write (int b);` // 将参数 *b* 的低两字节写入到输出流

☞ `public void write (char b[]);` // 将字符数组 *b[]* 中的全部字节顺序写入到输出流

☞ `public void write(char[] b, int off, int len);` // 将字节数组 *b[]* 中从 *off* 开始的 *len* 个字节写入到流中

☞ `public void write(String s);` // 将字符串写入流中

☞ `public void write(String s, int off, int len);` // 将字符串写入流中, *off* 为位置, *len* 为长度

☞ `public void flush ();` // 刷新流

☞ `public void close();` // 关闭流

9.1.2 节点流和处理流

按照流是否直接与特定的地方（如磁盘、内存、设备等）相连，分为节点流与处理流两类。

（1）节点流(Node Stream): 可以从或向一个特定的地方(节点)读写数据。如文件流 `FileReader`。

（2）处理流(Processing Stream): 是对一个已存在的流的连接和封装，通过所封装的流的功能调用实现数据读、写功能。处理流又称为过滤流，如缓冲处理流 `BufferedReader`。

节点流与处理流的关系，如图 9-1 所示。节点流直接与节点（如文件）相连，而处理流对节点流或其他处理流进一步进行处理（如缓冲、组装成对象，等等）。

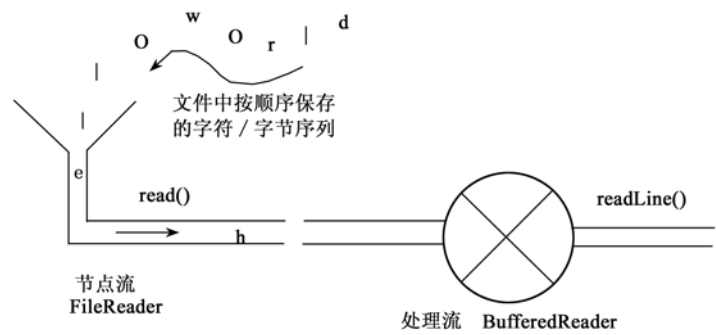


图 9-1 节点流与处理流的关系

处理流的构造方法总是要带一个其他的流对象作参数。如：

```
BufferedReader in = new BufferedReader(new FileReader(file));  
  
BufferedReader in2 =  
    new BufferedReader(  
        new (InputStreamReader(  
            new FileInputStream(file)))
```

一个流对象经过其他流的多次包装，称为流的链接，如图 9-2 所示。

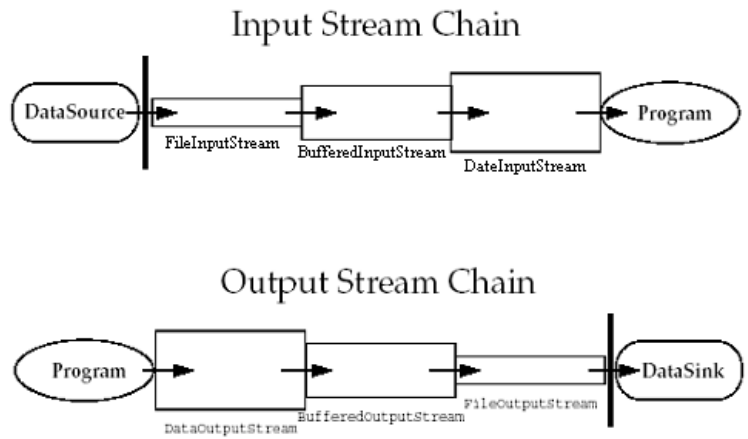


图 9-2 流的链接

常用的节点流和处理流分别如表 9-2、表 9-3 所示。

表 9-2 常用的节点流

节点类型	字节流	字符流
File 文件	FileInputStream FileOutputStream	FileReader FileWriter
Memory Array 内存数组	ByteArrayInputStream ByteArrayOutputStream	CharArrayReader CharArrayWriter
Memory String 字符串		StringReader StringWriter
Pipe 管道	PipedInputStream PipedOutputStream	PipedReader PipedWriter

表 9-3 常用的处理流

处 理 类 型	字 节 流	字 符 流
Buffering 缓冲	BufferedInputStream BufferedOutputStream	BufferedReader BufferedWriter
Filtering 过滤	FilterInputStream FilterOutputStream	FilterReader FilterWriter
Converting between bytes and character 字节流转为字符流		InputStreamReader OutputStreamWriter
Object Serialization 对象序列化	ObjectInputStream ObjectOutputStream	
Data conversion 基本数据类型转化	DataInputStream DataOutputStream	
Counting 行号处理	LineNumberInputStream	LineNumberReader
Peeking ahead 可回退流	PushbackInputStream	PushbackReader
Printing 可显示处理	PrintStream	PrintWriter

基本输入、输出流是定义基本的输入、输出操作的抽象类，在 Java 程序中真正使用的是它们的子类，对应于不同数据源和输入、输出任务，以及不同的输入、输出流。其中较常用的有：过滤输入、输出流 `FilterInputStream` 和 `FilterOutputStream` 两个抽象类，又分别派生出 `DataInputStream`、`DataOutputStream` 等子类。过滤输入、输出流的主要特点是在输入、输出数据的同时能对所传输的数据做指定类型或格式的转换，即可实现对二进制字节数据的理解和编码转换。文件输入、输出流 `FileInputStream` 和 `FileOutputStream` 主要负责完成对本地磁盘文件的顺序读写操作。管道输入、输出流 `PipedInputStream` 和 `PipedOutputStream` 负责实现程序内部的线程间通信或不同程序间的通信。字节数组流 `ByteArrayInputStream` 和 `ByteArrayOutputStream` 可实现与内存缓冲区的同步读写。顺序输入流 `SequenceInputStream` 可以把两个其他的输入流首尾相接，合并成一个完整的输入流，等等。

从抽象类 `Reader` 和 `Writer` 中也派生出一些子类，这些子类使 `InputStream` 和 `OutputStream` 的以字节为单位输入、输出转换为以字符为单位输入、输出，使用起来比 `InputStream` 和 `OutputStream` 要方便很多。

数据输入、输出流 `DataInputStream` 和 `DataOutputStream` 分别是过滤输入、输出流 `FilterInputStream` 和 `FilterOutputStream` 的子类。过滤输入、输出流的最主要作用就是在数据源和程序之间加一个过滤处理步骤，对原始数据做特定的加工、处理和变换操作。数据输入、输出流 `DataInputStream` 和 `DataOutputStream` 由于分别实现了 `DataInput` 和 `DataOutput` 两个接口中定义的独立于具体机器的带格式的读写操作，从而实现了对于不同类型数据的读写。

`DataInputStream` 流中定义了多个针对不同类型数据的读方法，如 `readByte()`，`readBoolean()`，`readShort()`，`readChar()`，`readInt()`，`readLong()`，`readFloat()`，`readDouble()`，`readLine()` 等。同样，`DataOutputStream` 中也定义了多个针对不同类型数据的写操作，如 `writeByte()`，`writeBoolean()`，`writeShort()`，`writeChar()`，`writeInt()`，`writeLong()`，`writeFloat()`，`writeDouble()`，`writeChars()` 等。每个方法都含有一个不同类型的参数，用来指定写入输出流的数据内容。这里省略了参数没有写出。

特别值得一提的是，能将字节流转为字符流的类是 `InputStreamReader` 及

OutputStreamWriter。

如果需要从与本地机器不同的字符编码(encoding)格式的文件中读取数据(例如,从网络上的一台基于不同平台的机器上读取文件),可以在构造 `InputStreamReader` 对象时显式指定其字符编码,例如:

```
ir = new InputStreamReader(System.in, "ISO8859_1")
```

Java 的输入、输出类库中包含的流类很多,这里只作了一个简要的介绍。更详细的内容可以参看 JDK 文档。

9.1.3 标准输入和标准输出

前面曾提到,当 Java 程序需要与外部设备等外界数据源做输入、输出的数据交换时,它需要首先创建一个输入或输出类的对象来完成对这个数据源的连接。例如,当 Java 程序需要读写文件时,它需要先创建文件输入或文件输出流类的对象。除文件外,程序也经常使用字符界面的标准输入、输出设备进行读写操作。

计算机系统都有默认的标准输入设备和标准输出设备。对一般的系统,标准输入通常是键盘,标准输出通常是显示器屏幕。Java 程序使用字符界面与系统标准输入、输出间进行数据通信,即从键盘读入数据,或向屏幕输出数据,是十分常见的操作,为此而频频创建输入、输出流类对象将很不方便。为此,Java 系统事先定义好两个流对象,分别与系统的标准输入和标准输出相联系,它们是 `System.in` 和 `System.out`。

`System` 是 Java 中一个功能很强大的类,利用它可以获得很多 Java 运行时的系统信息。`System` 类的所有属性和方法都是静态的,即调用时需要以类名 `System` 为前缀。`System.in` 和 `System.out` 就是 `System` 类的两个静态属性,分别对应了系统的标准输入和标准输出。

1. 标准输入

Java 的标准输入 `System.in` 是 `InputStream` 类的对象,当程序中需要从键盘读入数据的时候,只需调用 `System.in` 的 `read()` 方法即可。

在使用 `System.in.read()` 方法读入数据时,需要注意如下几点。

注意:(1) `System.in.read()` 语句必须包含在 `try` 块中,且 `try` 块后面应该有一个可接收 `IOException` 例外的 `catch` 块。

- (2) 执行 `System.in.read()` 方法将从键盘缓冲区读入一个字节的的数据,然而返回的却是 16 位的整型量,需要注意的是只有这个整型量的低位字节是真正输入的数据,其高位字节是全零。
 - (3) 另外,作为 `InputStream` 类的对象,`System.in` 只能从键盘读取二进制的的数据,而不能把这些信息转换为整数、字符、浮点数或字符串等复杂数据类型的量。
 - (4) 当键盘缓冲区中没有未被读取的数据时,执行 `System.in.read()` 将导致系统转入阻塞(block)状态。在阻塞状态下,当前流程将停留在上述语句位置且整个程序被挂起,等待用户输入一个键盘数据后,才能继续运行下去;所以程序中有时利用 `System.in.read()` 语句来达到暂时保留屏幕的目的。
-

为了使用方便，经常将 `System.in` 用各种处理流进行封装处理，如：

```
BufferedReader br = new BufferedReader( new
InputStreamReader(System.in));
Br.readLine();
```

2. 标准输出

Java 的标准输出 `System.out` 是打印输出流 `PrintStream` 类的对象。`PrintStream` 是过滤输出流类 `FilterOutputStream` 的一个子类，其中定义了向屏幕输送不同类型数据的方法 `print()` 和 `println()`。

(1) `println()` 方法。`println()` 方法有多种重载形式，概括起来可表述为：

```
public void println(类型变量或对象);
```

`println()` 的作用是向屏幕输出其参数指定的变量或对象，然后再换行，使光标停留在屏幕下一行第一个字符的位置。如果 `println()` 方法的参数为空，则将输出一个空行。

`println()` 方法可输出多种不同类型的变量或对象，包括 `boolean`, `double`, `float`, `int`, `long` 类型的变量及 `Object` 类的对象。由于 Java 中规定子类对象作为实际参数可以与父类对象的形式参数匹配，而 `Object` 类又是所有 Java 类的父类，所以 `println()` 实际可以通过重载实现对所有类对象的屏幕输出。

(2) `print()` 方法。`print()` 方法的重载情况与 `println()` 方法完全相同，也可以实现在屏幕上输出不同类型的变量和对象的操作。不同的是，`print()` 方法输出对象后并不附带一个回车，下一次输出时，将输出在同一行中。

9.1.4 应用举例

例 9-1 FileCopyByChar.java 复制文件并显示文件，将每个字符读入，并写入另一个文件，同时显示出来。

```
import java.io.*;
public class FileCopyByChar {
    public static void main(String[] args) {
        try {
            FileReader input = new
            FileReader("FileCopyByChar.java");
            FileWriter output = new FileWriter("temp.txt");
            int c = input.read();
            while ( c != -1 ) {
                output.write(c);
                System.out.print( (char) c );
                c = input.read();
            }
            input.close();
```



```
        output.close();
    } catch (IOException e) {
        System.out.println(e);
    }
}
}
```

例 9-2 FileCopyByLine.java 复制文件并显示文件，将每个字符读入，并写入另一个文件，同时显示出来。这里用了 **BufferedReader** 及 **BufferedWriter**，前面一个类的重要方法是 **readLine()**，它读入一行字符。

```
import java.io.*;

public class FileCopyByLine {
    public static void main(String[] args) {
        try {
            FileReader input = new FileReader
                ("FileCopyByLine.java");
            BufferedReader br = new BufferedReader(input);
            FileWriter output = new FileWriter("temp.txt");
            BufferedWriter bw = new BufferedWriter(output);

            String s = br.readLine();
            while ( s!=null ) {
                bw.write(s);
                bw.newLine();
                System.out.println(s);
                s = br.readLine();
            }
            br.close();
            bw.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

例 9-3 CopyFileAddLineNumber.java 读入一个 java 文件，将每行中的注释去掉，并加上行号，写入另一文件。

```
import java.io.*;

public class CopyFileAddLineNumber {
```

```
public static void main (String[] args) {
    String infname = "CopyFileAddLineNumber.java";
    String outfname = "CopyFileAddLineNumber.txt";
    if( args.length >= 1 ) infname = args[0];
    if( args.length >= 2 ) outfname = args[1];

    try {
        File fin = new File(infname);
        File fout = new File(outfname);

        BufferedReader in = new BufferedReader(new
            FileReader(fin));
        PrintWriter out = new PrintWriter(new
            FileWriter(fout));

        int cnt = 0;                                // 行号
        String s = in.readLine();
        while ( s != null ) {
            cnt ++;
            s = deleteComments(s);                  // 去掉以//开始的注释
            out.println(cnt + ": \t" + s );          // 写出
            s = in.readLine();                        // 读入
        }
        in.close();                                  // 关闭缓冲读入流及
                                                    // 文件读入流的连接

        out.close();

    } catch (FileNotFoundException e1) {
        System.err.println("File not found!" );
    } catch (IOException e2) {
        e2.printStackTrace();
    }
}

static String deleteComments( String s )
                                                    // 去掉以//开始的注释
{
    if( s==null ) return s;
    int pos = s.indexOf( "//" );
    if( pos<0 ) return s;
```

```
        return s.substring( 0, pos );
    }
}
```

9.2 文件及目录

9.2.1 文件与目录管理

Java 支持文件管理和目录管理，它们都是由专门的 `java.io.File` 类来实现。`File` 类也在 `java.io` 包中，但它不是 `InputStream` 或者 `OutputStream` 的子类，因为它不负责数据的输入、输出，而是专门被用来管理磁盘文件和目录。

每个 `File` 类的对象表示一个磁盘文件或目录，其对象属性中包含了文件或目录的相关信息，如名称、长度、所含文件个数等，调用它的方法则可以完成对文件或目录的常用管理操作，如创建、删除等。

1. 创建 `File` 类的对象

每个 `File` 类的对象都对应了系统的一个磁盘文件或目录，所以创建 `File` 类对象时需指明它所对应的文件或目录名。`File` 类共提供了三个不同的构造函数，以不同的参数形式灵活地接收文件和目录名信息。

(1) `File(String path)`。这里的字符串参数 `path` 指明了新创建的 `File` 对象对应的磁盘文件或目录名及其路径名。`path` 参数也可以对应磁盘上的某个目录，如“`c: \myProgram\Java`”或“`myProgram\Java`”。

注意：不同的操作系统所使用的目录分隔符是不一样的。例如，DOS，Windows 系统使用反斜线，而 Unix 系统却使用正斜线。

为了使 Java 程序能在不同的平台间平滑移植，可以借助于 `System` 类的一个静态方法，能得到当前系统规定的目录分隔符：

```
String sep = System.getProperty( "file.separator" );
```

(2) `File(String path, String name)`。第二个构造函数有两个参数，第一个参数 `path` 表示所对应的文件或目录的绝对或相对路径，第二个参数 `name` 表示文件或目录名。将路径与名称分开的好处是相同路径的文件或目录可共享同一个路径字符串，管理、修改都较方便。

(3) `File(File dir, String name)`。第三个构造函数使用另一个已经存在的代表某磁盘目录的 `File` 对象作为第一个参数，表示文件或目录的路径，第二个字符串参数表述文件或目录名。

2. 获取文件或目录属性

一个对应于某磁盘文件或目录的 `File` 对象一经创建，就可以通过调用它的方法来获得该文件或目录的属性。其中，较常用的方法如下。

(1) 判断文件或目录是否存在。

```
public boolean exists(); // 若文件或目录存在, 则返回 true; 否则返回 false
```

(2) 判断是文件还是目录。

```
public boolean isFile(); // 若对象代表有效文件, 则返回 true
```

```
public boolean isDirectory(); // 若对象代表有效目录, 则返回 true
```

(3) 获取文件或目录名称与路径。

```
public String getName(); // 返回文件名或目录名
```

```
public String getPath(); // 返回文件或目录的路径
```

(4) 获取文件的长度:

```
public long length(); // 返回文件的字节数
```

(5) 获取文件读写属性:

```
public boolean canRead(); // 若文件为可读文件, 则返回 true, 否则返回 false
```

```
public boolean canWrite(); // 若文件为可写文件, 返回 true, 否则返回 false
```

(6) 列出目录中的文件:

```
public String[] list(); // 将目录中所有文件名保存在字符串数组中返回
```

(7) 比较两个文件或目录:

```
public boolean equals(File f); // 若两个 File 对象相同, 则返回 true
```

3. 文件或目录操作

File 类中还定义了一些对文件或目录进行管理、操作的方法, 常用的有如下几种。

(1) 重命名文件。

```
public boolean renameTo(File newFile); // 将文件重命名成 newFile 对应的文件名
```

(2) 删除文件。

```
public void delete(); // 将当前文件删除
```

(3) 创建目录。

```
public boolean mkdir(); // 创建当前目录的子目录
```

字符作为文件名, 输出这个文件的有关信息。

例 9-4 ListAllFiles.java 递归地列出某目录下的所有文件。

```
import java.io.*;

class ListAllFiles
{
    public static void main(String[] args){
        ListFiles( new File( "d:\\tang" ));
    }

    public static void ListFiles( File dir ){
        if( !dir.exists() || ! dir.isDirectory() ) return;
    }
}
```

```

String [] files = dir.list();
for( int i=0; i<files.length; i++){
    File file = new File( dir, files[i] );
    if( file.isFile() ){
        System.out.println(
            dir + "\\\" + file.getName() + "\\t\" +
            file.length() );
    }else{
        System.out.println(
            dir + "\\\" + file.getName() + "\\t<dir>\" );

        ListFiles( file ); //对于子目录,进行递归调用
    }
}
}
}
}

```

运行结果如图 9-3 所示。

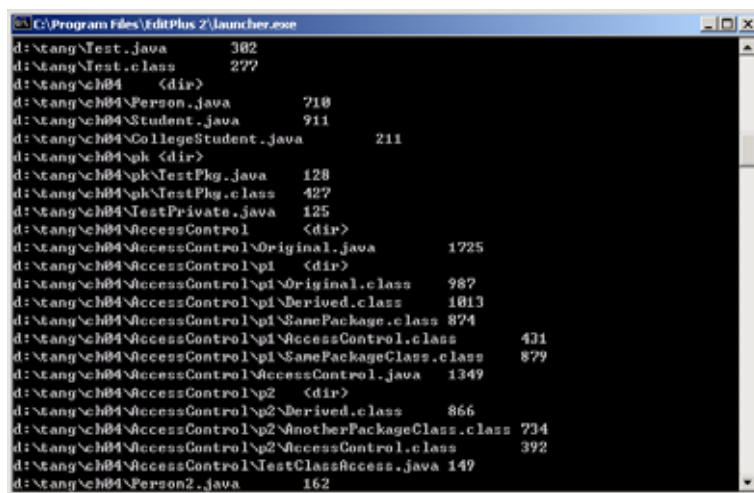


图 9-3 递归地列出某目录下的所有文件

9.2.2 文件输入与输出流

使用 File 类,可以方便地建立与某磁盘文件的连接,了解它的有关属性并对其进行一定的管理性操作。但是,如果希望从磁盘文件读取数据,或者将数据写入文件,还需要使用文件输入、输出流类 FileInputStream 和 FileOutputStream。

利用文件输入、输出流完成磁盘文件的读写一般应遵循如下的步骤。

1. 利用文件名字符串或 File 对象创建输入输出流对象

以 `FileInputStream` 为例，它有两个常用的构造函数。

(1) `FileInputStream(String FileName);` // 利用文件名(包括路径名)字符串创建从该文件读入数据的输入流

(2) `FileInputStream(File f);` // 利用已存在的 `File` 对象创建从该对象对应的磁盘文件中读入数据的文件输入流

注意：无论哪个构造函数，在创建文件输入或输出流时都可能因给出的文件名不对或路径不对，或文件的属性不对等，不能读出文件而造成错误，此时系统会抛出异常 `FileNotFoundException`，所以创建文件输入、输出流并调用构造函数的语句应该被包括在 `try` 块中，并有相应的 `catch` 块来处理它们可能产生的异常。

2. 从文件输入、输出流中读写数据

从文件输入、输出流中读写数据有两种方式，一是直接利用 `FileInputStream` 和 `FileOutputStream` 自身的读写功能；二是以 `FileInputStream` 和 `FileOutputStream` 为原始数据源，再套接上其他功能较强大的输入、输出流完成文件的读写操作。

`FileInputStream` 和 `FileOutputStream` 自身的读写功能是直接从父类 `InputStream` 和 `OutputStream` 那里继承来的，并未加任何功能的扩充和增强，如前面介绍过的 `read()`、`write()` 等方法，都只能完成以字节为单位的原始二进制数据的读写。`read()` 和 `write()` 的执行还可能因 IO 错误导致抛出 `IOException` 异常对象，在文件尾执行 `read()` 操作时将导致阻塞。

为了能更方便地从文件中读写不同类型的数据，一般都采用第二种方式，即以 `FileInputStream` 和 `FileOutputStream` 为数据源完成与磁盘文件的映射连接后，再创建其他流类的对象从 `FileInputStream` 和 `FileOutputStream` 对象中读写数据。一般较常用的是过滤流的两个子类 `DataInputStream` 和 `DataOutputStream`，甚至还可以进一步简化为如下写法：

```
File MyFile=new File("MyTextFile");
DataInputStream din=new DataInputStream(new FileInputStream(MyFile));
DataOutputStream dour=new DataOutputStream(new FileOutputStream(MyFile));
```

例 9-5 `FileDisplay.java` 显示文本内容。例中用到了 `FileDialog` 来让用户选择文件，如图 9-4 所示。

```
import java.io.*;
import java.awt.*;

public class FileDisplay{
    public static void main(String args[]){
        Frame f = new Frame("test for filedialog");
        TextArea text = new TextArea(40,40);
        f.add( text );
        f.setSize(600,500);
```

```

f.setVisible(true);

FileDialog fd=new FileDialog(f,"文件对话框",FileDialog.
LOAD);
fd.setVisible(true);
String fpath=fd.getDirectory();
String fname=fd.getFile();
String si=fpath+fname;

File file = new File(si);
try {
    BufferedReader in = new BufferedReader(new
    FileReader(file));
    String s;
    s = in.readLine();
    while ( s != null ) {
        s = in.readLine();
        text.append( s + "\n" );
    }
    in.close();
} catch (IOException e2) {
    e2.printStackTrace();
}
}
}

```

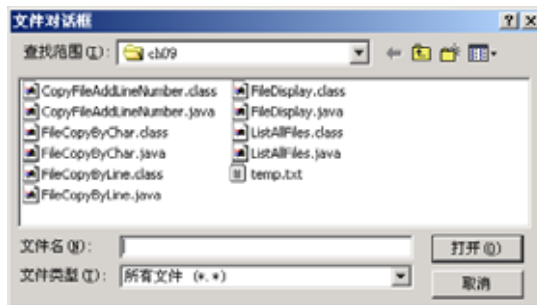


图 9-4 使用 FileDialog

9.2.3 RandomAccessFile 类

FileInputStream 和 FileOutputStream 实现的是对磁盘文件的顺序读写，而且读和写要

分别创建不同的对象。相比之下，Java 中还定义了另一个功能更强大，使用更方便的类——`RandomAccessFile`，它可以实现对文件的随机读写操作。

1. 创建 `RandomAccessFile` 对象

`RandomAccessFile` 类有两个构造函数：

```
RandomAccessFile(String name, String mode);
```

```
RandomAccessFile(File f, String mode);
```

无论使用哪个创建 `RandomAccessFile` 对象，都要求提供两种信息：一个作为数据源的文件，以文件名字符串或文件对象的方式表述；另一个是访问模式字符串，它规定了 `RandomAccessFile` 对象可以用何种方式打开和访问指定的文件。访问模式字符串 `mode` 有两种取值：“r”代表了以只读方式打开文件；“rw”代表以读写方式打开文件，这时用一个对象就可以同时实现读写两种操作。

创建 `RandomAccessFile` 对象时，可能产生两种异常：当指定的文件不存在时，系统将抛出 `FileNotFoundException`；若试图用读写方式打开只读属性的文件或出现了其他输入、输出错误，则会抛出 `IOException` 异常。下面是创建 `RandomAccessFile` 对象例句：

```
File BankMegFile=new File("BankFile.txt");
RandomAccessFile MyRAF=new RandomAccessFile(BankMegFile, "rw"); //读写方式
```

2. 对文件位置指针的操作

与前面的顺序读写操作不同，`RandomAccessFile` 实现的是随机读写，即可以在文件的任意位置执行数据读写，而不一定要从前向后操作。要实现这样的功能，必须定义文件位置指针和移动这个指针的方法。`RandomAccessFile` 对象的文件位置指针遵循如下的规律。

- (1) 新建 `RandomAccessFile` 对象的文件位置指针位于文件的开头处。
- (2) 每次读写操作之后，文件位置指针都相应后移读写的字节数。
- (3) 利用 `getPointer()`方法可获取当前文件位置指针从文件头算起的绝对位置。

```
public long getPointer();
```

- (4) 利用 `seek()`方法可以移动文件位置指针。

```
public void seek(long pos);
```

这个方法将文件位置指针移动到参数 `pos` 指定的从文件头算起的绝对位置处。

- (5) `length()`方法将返回文件的字节长度。

```
public long length();
```

根据 `length()`方法返回的文件长度和位置指针相比较，可以判断是否读到了文件尾。

3. 读操作

与 `DataInputStream` 相似，`RandomAccessFile` 类也实现了 `DataInput` 接口，即它也可以用多种方法分别读取不同类型的数据，具有比 `FileInputStream` 更强大的功能。`RandomAccessFile` 的读方法主要有：`readBealoon()`，`readChar()`，`readInt()`，`readLong()`，`readFloat()`，`readDouble()`，`readLine()`，`readUTF()`等。`readLine()`从当前位置开始，到第一个“\n”为止，读取一行文本，它将返回一个 `String` 对象。其他方法前面已经介绍过，不再赘述。

4. 写操作

在实现了 `DataInput` 接口的同时, `RandomAccessFile` 类还实现了 `DataOutput` 接口, 这就使它具有了与 `DataOutputStream` 类同样强大的含类型转换的输出功能。`RandomAccessFile` 类包含的写方法主要有: `writeBealoon()`, `writeChar()`, `writeInt()`, `writeLong()`, `writeFloat()`, `writeDouble()`, `writeLine()`, `writeUTF()`等。其中 `writeUTF()`方法可以向文件输出一个字符串对象。

注意: `RandomAccessFile` 类的所有方法都有可能抛出 `IOException` 异常, 所以利用它实现文件对象操作时应把相关的语句放在 `try` 块中, 并配上 `catch` 块来处理可能产生的异常对象。

并不是所有的 Java 程序都可以处理文件并执行文件操作。由于 Java Applet 程序通常是从网络上下载到本地运行的, 不可知也不可控, 所有 Java 的安全机制禁止 Java Applet 程序访问和存取本地文件, 以避免对本地硬盘的可能的攻击。如果试图在 Java Applet 程序中使用文件操作, 则将引发 Java 的安全异常。

9.3 基于文本的应用

基于文本的应用程序, 经常与字符串(`String`, `StringBuffer`)及文件(`File`)、流(`InputStream`, `OutputStream`, `Reader`, `Writer`)等相关。许多内容在前面的章节中已经讲过, 这里介绍基于文本的应用中的几个问题。

9.3.1 Java Application 命令行参数

Java Application 是用命令行来启动执行的, 命令行参数就成为向 Java Application 传入数据的常用而有效的手段。

在启动 Java 应用程序时可以一次性地向应用程序中传递 0 到多个参数——命令行参数。

命令行参数使用格式如下:

```
java 类名 参数 参数 ...
```

参数之间用空格隔开, 如果某个参数本身含有空格, 则可以将参数用一对双引号引起来。

命令行参数被系统以 `String` 数组的方式传递给应用程序中的 `main` 方法, 由参数 `args` 接收:

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

例 9-6 TestCommandLine.java 使用命令行参数。

```
public class TestCommandLine{
    public static void main(String[] args) {
        for ( int i = 0; i < args.length; i++ ) {
            System.out.println("args[" + i + "] = " + args[i]);
        }
    }
}
```

```
    }  
  }  
}
```

运行时，使用

```
java TestCommandLine lisa "bily" "Mr Brown"
```

运行结果如图 9-5 所示。

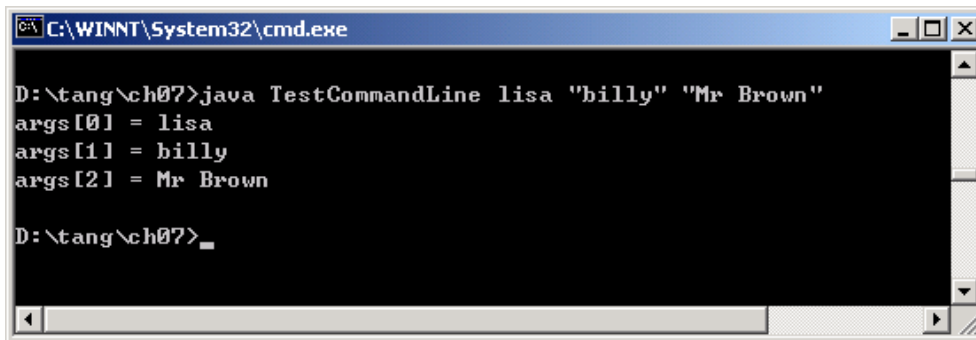


图 9-5 使用命令行参数

9.3.2 环境参数

程序运行时，常常要使用环境参数来决定程序的不同表现。这里介绍一下相关的概念，以及如何在程序中获取环境参数。

环境参数是设于运行环境中的一些信息，其形式是：

环境参数名 = 参数值

在 Windows 平台下，设定环境参数，可以在【我的电脑】中选择【属性】通过【高级】对环境变量进行设置。

在 DOS 平台下，设定环境参数，用 Set 命令，即

Set 环境参数名 = 参数值

另外，在运行 Java 时，也可以设定环境变量：

java -D 环境变量名=值 类名 命令行参数

在 Java 编程中，用 System.getProperties()或 System.getProperty()方法来获得环境参数。

例 9-7 TestProperties.java 获得环境参数。

```
import java.util.Properties;  
import java.util.Enumeration;  
  
public class TestProperties {  
    public static void main(String[] args) {  
        Properties ps = System.getProperties();  
        Enumeration pn = ps.propertyNames();
```

```

        while ( pn.hasMoreElements() ) {
            String pName = (String) pn.nextElement();
            String pValue = ps.getProperty(pName);
            System.out.println(pName + "----" + pValue);
        }
    }
}

```

程序中的 Properties 是 Hashtable 的子类。

运行时，可以用

```
java -DmyProperty=MyValue TestProperties
```

运行时，显示很多环境变量，其中包括 Java 虚拟机提供的一些参数，DOS 环境提供的一些参数，以及在 Java 命令行上给定的参数，运行结果如图 9-6 所示。

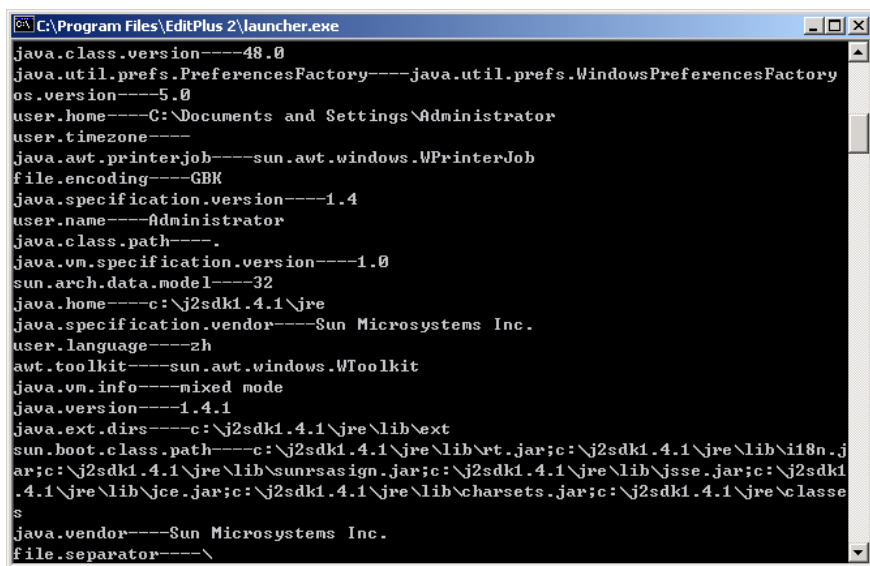


图 9-6 获得环境参数

9.3.3 处理 Deprecated 的 API

由于 JDK 版本的变化，有一部分类、方法或属性在新的版本中不提倡使用或者不能使用，这种情况称为 Deprecated 的类、方法或属性。

例 9-8 TestDeprecated1.java 处理 Deprecated 的 API。

```

public class TestDeprecated1{
    public static void main(String args[]){
        String f;
    }
}

```

```
f = System.getenv("java.class.path");  
System.out.println(f);  
}  
}
```

该程序中使用了 `System.getenv()` 方法, 在新版的 JDK 中, 已不再使用, 所以编译时会提出警告, 若要查看 Deprecation 的详细信息, 在编译时, 可加上 `-deprecation` 选项。含有 Deprecated 类或方法的程序在运行时会抛出异常, 结果如图9-7 所示。

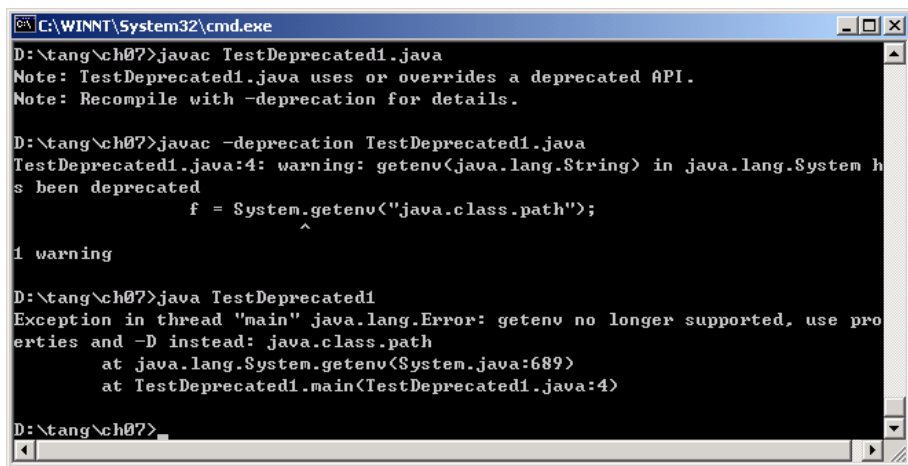


图 9-7 处理 Deprecated 的 API

要解决这类问题, 就要用新的 API 代替原来的方法。这里可以用 `System.getProperty()` 方法, 如下所示:

```
public class TestDeprecated2{  
    public static void main(String args[]){  
        String f;  
        f = System.getProperty("java.class.path");  
        System.out.println(f);  
    }  
}
```

习题

1. 字节流与字符流有什么差别?
2. 节点流与处理流有什么差别?
3. 输入流与输出流各有什么方法?
4. 怎样进行文件及目录的管理?

5. 编写一个程序，从命令上行接收两个实数，计算其乘积。
6. 编写一个程序，从命令行上接收两个文件名，比较两个文件的长度及内容。
7. 编写一个程序，能将一个 **Java** 源程序中的空行及注释去掉。

第 10 章 图形用户界面

图形用户界面是程序与用户交互的方式，利用它可以接受用户的输入并向用户输出程序运行的结果。本章将介绍图形用户界面的(GUI)基本组成和主要操作，包括 AWT 组件、布局管理、事件处理、绘制图形、显示动画、使用 Swing 组件等，在本章的最后还介绍了基于 GUI 的应用程序的一般建立方法，包括使用菜单、工具栏、剪贴板等。

10.1 AWT 组件

10.1.1 图形用户界面概述

设计和构造用户界面，是软件开发中的一项重要工作。用户界面是计算机用户与计算机系统交互的接口，用户界面功能是否完善，使用是否方便，将直接影响到用户对应用软件的使用。图形用户界面(graphics user interface, GUI)，使用图形的方式借助菜单、按钮等标准界面元素和鼠标操作，帮助用户方便地向计算机系统发出命令，启动操作，并将系统运行的结果同样以图形的方式显示给用户。图形用户界面画面生动、操作简便，已经成为目前几乎所有应用软件的既成标准。所以，学习设计和开发图形用户界面是十分重要的。

简单地说，图形用户界面就是一组图形界面成分和界面元素的有机组合，这些成分和元素之间不但外观上有着包含、相邻、相交等物理关系，内在的也有包含、调用等逻辑关系，它们互相作用、传递消息，共同组成一个能响应特定事件、具有一定功能的图形界面系统。

Java 语言中，处理图形用户界面的类库主要是 java.awt 包和 javax.swing 包。

AWT 是 abstract window toolkit(抽象窗口工具集)的缩写。“抽象窗口”使得开发人员所设计的界面独立于具体的界面实现。也就是说，开发人员用 AWT 开发出的图形用户界面可以适用于所有的平台系统。当然，这仅是理想情况。实际上 AWT 的功能还不是很完全，Java 程序的图形用户界面在不同的平台上(例如，在不同的浏览器中)可能会出现不同的运行效果，如窗口大小、字体效果将发生变化等。

Javax.swing 包是 JDK1.2 以后版本所引入的图形用户界面类库，swing 是功能强大的 Java 的基础类库(JFC)的一部分，其中定义的 SwingGUI 组件相对于 java. awt 包的各种 GUI 组件增加了许多功能。

设计和实现图形用户界面的工作主要有以下几点。

- (1) 创建组件 (Component)：创建组成界面的各种元素，如按钮、文本框等。
- (2) 指定布局 (Layout)：根据具体需要排列它们的位置关系。
- (3) 响应事件 (Event)：定义图形用户界面的事件和各界面元素对不同事件的响应，从而实现图形用户界面与用户的交互功能。

本章中将对组件、布局、事件等进行讲解，读者可以据此编制一些图形用户界面的程序。在实际开发过程中，经常借助各种具有可视化图形界面设计功能的软件，如 Jbuilder, Visual Café, Visual Age for Java, Java Workshop 等，这些工具软件有助于提高界面设计的效率。

10.1.2 AWT 组件分类

Java 中构成图形用户界面的各种元素，称为组件（Component）。Java 程序要显示的 GUI 组件都是抽象类 `java.awt.Component` 或 `java.awt.MenuComponent` 的子类。`MenuComponent` 是与菜单相关的组件，将在 10.8 节中介绍，这里介绍 `Component` 类。

组件分为容器（Container）类和非容器类组件两大类。容器本身也是组件，但容器中可以包含其他组件，也可以包含其他容器。非容器类组件的种类较多，如按钮（Button），标签（Label），文本类组件 `TextComponent` 等。

容器又分为两种顶层容器和非顶层容器两大类。顶层容器是可以独立的窗口，顶层容器的类是 `Window`，`Window` 的重要子类是 `Frame` 和 `Dialog`。非顶层容器，不是独立的窗口，它们必须位于窗口之内，非顶层容器包括 `Panel` 及 `ScrollPane` 等，`Panel` 的重要子类是 `Applet`。其中，`Panel` 和 `Applet` 的容器都是无边框的；`ScrollPane` 一组是可以自动处理滚动操作的容器；`Window`，`Frame`，`Dialog` 和 `FileDialog` 是一组大都含有边框，并可以移动、放大、缩小、关闭的功能较强的容器。

AWT 组件的分类，可以用图 10-1 来表示。相关类的继承关系如图 10-2 所示。

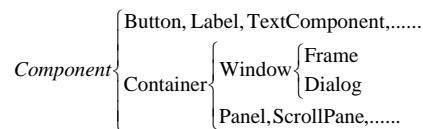


图 10-1 AWT 组件的分类



图 10-2 Component 类的继承关系

1. Container 类

Contain 类的特点是：容器中可以容纳其他组件，使用 `add()` 方法，可以将其他对象加入到容器中，加入到容器中后，组件的位置和尺寸由布局管理器决定（关于布局管理器在 10.2 节中详细介绍）。如果要人工控制组件在容器中的大小位置，可取消布局管理器，即用方法 `setLayout(null)`，然后使用 `Component` 类的下述成员方法：

```
setLocation()  
setSize()  
setBounds()
```

来设定其大小及位置。

下面介绍 Container 的两个重要子类：Frame 及 Panel。

(1) Frame 类具有以下特点：

- ✎ Frame 类是 Window 类的直接子类；
- ✎ Frame 对象显示效果是一个“窗口”，带有标题和尺寸重置角标；
- ✎ 默认初始化为不可见的，可使用 setVisible(true)方法使之变为可见；
- ✎ 默认的布局管理器是 BorderLayout，可使用 setLayout()方法改变其默认布局管理器。

(2) Panel 类具有以下特点：

- ✎ Panel 不是顶层窗口，它必须位于窗口或其他容器之内；
- ✎ Panel 提供可以容纳其他组件，在程序中经常用于布局和定位；
- ✎ 默认的布局管理器是 FlowLayout，可使用 setLayout()方法改变其默认布局管理器；
- ✎ Panel 可以采用和所在容器不同的布局管理器。

例 10-1 TestFrame.java 使用 Frame。

```
import java.awt.*;  
public class TestFrame {  
    public static void main( String args[]) {  
        Frame f = new Frame("My First Test");  
        f.setSize( 170,100);  
        f.setBackground( Color.blue);  
  
        f.setVisible( true);  
    }  
}
```

运行结果如图 10-3 所示。

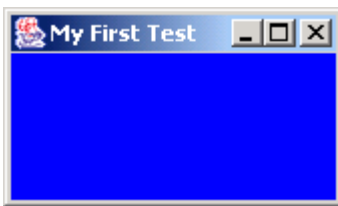


图 10-3 使用 Frame 的运行结果

例 10-2 TestFrameWithPanel.java 在 Frame 中加入 Panel。

```
import java.awt.*;
```



```
public class TestFrameWithPanel {  
    public static void main(String args[]) {  
        Frame f = new Frame("MyTest Frame");  
        f.setSize(300,200);  
        f.setLocation( 500, 400 );  
        f.setBackground(Color.blue);  
  
        Panel pan = new Panel();  
        pan.setSize(150,100);  
        pan.setLocation( 50, 50 );  
        pan.setBackground(Color.green);  
  
        Button b = new Button("ok");  
        b.setSize( 80,20 );  
        b.setLocation( 50,50 );  
        b.setBackground(Color.red);  
  
        f.setLayout(null);          // 取消默认布局管理器  
        pan.setLayout( null );  
        pan.add(b);                 // 面板上加入按钮  
        f.add(pan);                 // 窗体上加入面板  
  
        f.setVisible(true);  
    }  
}
```

运行结果如图 10-4 所示。

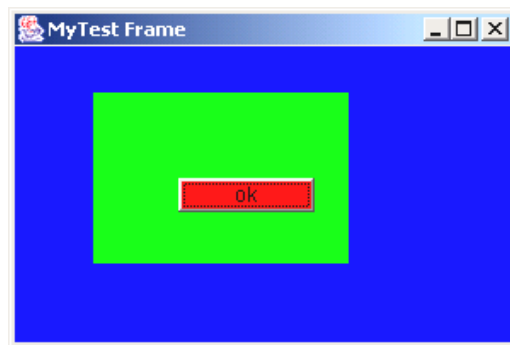


图 10-4 在 Frame 中加入 Panel

2. 非 Container 类组件

非 Container 类组件, 又称为控制组件, 与容器不同, 它里面不再包含其他组件。控制组件的作用是完成与用户的交互, 包括接收用户的一个命令(如按钮), 接收用户的一个文本或选择输入, 向用户显示一段文本或一个图形, 等等。常用的控制组件有以下几种。

命令类: 按钮 Button;

选择类: 单选按钮、复选按钮 Checkbox、列表框 List、下拉框 Choice;

文字处理类: 文本框 TextField、文本区域 TextArea。

使用控制组件, 通常需要如下的步骤。

(1) 创建某控制组件类的对象, 指定其大小等属性。

(2) 使用某种布局策略, 将该控制组件对象加入到某个容器中的某指定位置处。

(3) 将该组件对象注册给它所能产生的事件对应的事件监听者, 重载事件处理方法, 实现利用该组件对象与用户交互的功能。

10.1.3 Component 的方法

Component 类是所有组件和容器的抽象父类, 其中定义了一些每个容器和组件都可能用到的方法, 较常用的方法如下。

(1) `public void add(PopupMenu popup);` // 在组件上加入一个弹出菜单, 当用户用鼠标右键单击组件时将弹出这个菜单

(2) `public Color getBackground();` // 获得组件的背景色

(3) `public Font getFont();` // 获得组件使用的字体

(4) `public Color getForeground();` // 获得组件的前景色

(5) `public Graphics getGraphics();` // 获得在组件上绘图时需要使用的 Graphics 对象

(6) `public void repaint(int x, int y, int width, int height);` // 以指定的坐标点(x, y)为左上角, 重画组件中指定宽度(width)、指定高度(height)的区域

(7) `public void setBackground(Color c);` // 设置组件的背景色

(8) `public void setEnabled(boolean b);` // 设置组件的使能状态。参数 b 为 true 则组件使能, 否则组件不使能。只有使能状态的组件才能接受用户输入并引发事件

(9) `public void setFont(Font f);` // 设置组件使用的字体

(10) `public void setSize(int width, int height);` // 设置组件的大小

(11) `public void setVisible(boolean b);` // 设置组件是否可见的属性。参数 b 为 true 时, 组件在包括它的容器可见时也可见; 否则组件不可见

(12) `public void setForeground(Color c);` // 设置组件的前景色

(13) `public void requestFocus();` // 使组件获得注意的焦点

由于 Component 是其他组件类的父类, 所以以上所有方法都可以应用到其他各种组件中。对于具体的组件, 还有相应的方法, 读者可以查看 JDK 文档。

10.2 布局管理

在 Java 的 GUI 界面设计, 布局控制是相当重要的一环。将一个组件加入容器中时, 布局控制决定了所加入的组件的大小和位置。如果将一个容器的布局管理器设为 `null`, 即用方法 `setLayout(null)`, 则要设定容器中每个对象的大小和位置。而布局管理器能自动设定容器中的组件的大小和位置, 当容器改变大小时, 布局管理器能自动地改变其中组件的大小和位置。

Java.awt 包中共定义了五种布局管理器, 每个布局管理器对应一种布局策略, 分别是 `FlowLayout`, `BorderLayout`, `CardLayout`, `GridLayout` 和 `GridBagLayout`。下面将详细讨论这几种布局管理器。

10.2.1 FlowLayout

`FlowLayout` 是容器 `Panel` 和它的子类 `Applet` 默认使用的布局管理器, 如果不专门为 `Panel` 或 `Applet` 指定布局管理器, 则它们就使用 `FlowLayout`。

`FlowLayout` 对应的布局策略非常简单。遵循这种策略的容器将其中的组件按照加入的先后顺序从左向右排列, 一行排满之后就下转到下一行继续从左至右排列, 每一行中的组件都居中排列; 在组件不多时, 使用这种策略非常方便, 但是当容器内的 GUI 元素增加时, 就显得高低参差不齐。

对于使用 `FlowLayout` 的容器, 加入组件使用如下简单的命令:

```
add(组件名);
```

设定一个容器的布局管理器, 可以使用 `setLayout()` 方法, 如:

```
setLayout(new FlowLayout());
```

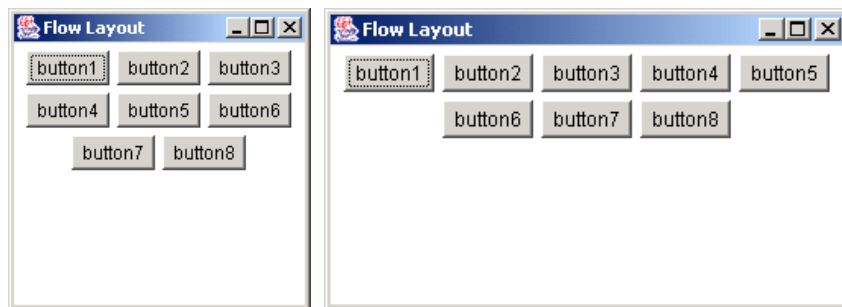
`FlowLayout` 的构造方法有三种形式:

☞ `FlowLayout()`;

☞ `FlowLayout(int align)`;

☞ `FlowLayout(int align, int hgap, int vgap)`;

参数 `align` 指定每行组件的对齐方法, 可以取三个静态常量 `LEFT`, `CENTER`, `RIGHT` 之一, 默认为 `CENTER`。`hgap` 及 `vgap` 指组件间的横纵间距, 默认为 5 个像素。如图 10-5 所示。



(a)

(b)

图 10-5 FlowLayout (a)、(b)是当 Form 在不同宽度时的布局

例 10-3 TestFlowLayout.java 使用 FlowLayout。

```
import java.awt.*;

public class TestFlowLayout {
    public static void main(String args[]) {
        Frame f = new Frame("Flow Layout");
        Button[] buttons = new Button[8];
        for( int i=0; i<buttons.length; i++)
            buttons[i] = new Button( "button"+(i+1) );

        f.setLayout(new FlowLayout(FlowLayout.LEFT, 10, 20 ));

        for( int i=0; i<buttons.length; i++)
            f.add( buttons[i] );

        f.setSize(200,200);
        // f.pack();
        f.setVisible(true);
    }
}
```

10.2.2 BorderLayout

BorderLayout 是容器 Frame 和 Dialog 默认使用的布局管理器。

BorderLayout 也是一种简单的布局策略，它把容器内的空间简单地划分为东、西、南、北、中五个区域，每加入一个组件都应该指明把这个组件加在哪个区域中。

分布在北部和南部区域的组件将横向扩展至占据整个容器的长度，分布在东部和西部的组件将伸展至占据容器剩余部分的全部宽度，最后剩余的部分将分配给位于中央的组件。如果某个区域没有分配组件，则其他组件可以占据它的空间。例如，如果北部没有分配组件，则西部和东部的组件将向上扩展到容器的最上方，如果西部和东部没有分配组件，则位于中央的组件将横向扩展到容器的左右边界。如图 10-6 所示。

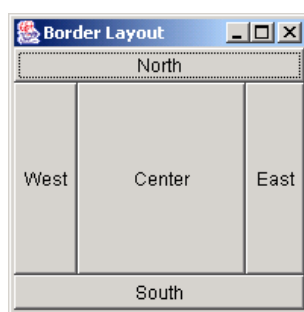


图 10-6 BorderLayout

BorderLayout 的构造方法有两种形式：

☞ BorderLayout();

☞ BorderLayout(int hgap, int vgap);

其中，hgap 及 vgap 指组件间的横纵间距，默认为 0 个像素。

只能指定五个区域位置。如果一个区域加入的组件超过多个，则只能显示该区域的最后加入的一个对象。如果容器中需要加入超过五个组件，就必须使用容器的嵌套或改用其他的布局策略。

例 10-4 TestBorderLayout.java 使用 BorderLayout。

```
import java.awt.*;

public class TestBorderLayout {

    public static void main(String args[]) {

        Frame f = new Frame("Border Layout");

        f.setLayout(new BorderLayout());
        f.add(new Button("North"), BorderLayout.NORTH);
        f.add(new Button("South"), BorderLayout.SOUTH);
        f.add(new Button("East"), BorderLayout.EAST);
        f.add(new Button("West"), BorderLayout.WEST);
        f.add(new Button("Center"), BorderLayout.CENTER);
        //f.add(new Button("Center"), "Center");//注意大小写
        //f.add("Center",new Button("Center")) );

        f.setSize(200,200);
        f.setVisible(true);

    }

}
```

10.2.3 CardLayout

使用 CardLayout 的容器表面上可以容纳多个组件，但是实际上，在同一时间容器只能从这些组件中选出一个来显示，就像一叠“扑克牌”每次只能显示最上面的一张一样，这个被显示的组件将占据所有的容器空间。

CardLayout 的构造方法有两种形式：

☞ CardLayout ();

☞ CardLayout (int hgap, int vgap);

其中，hgap 及 vgap 指组件间的横纵间距，默认为 0 个像素。如图 10-7 所示。

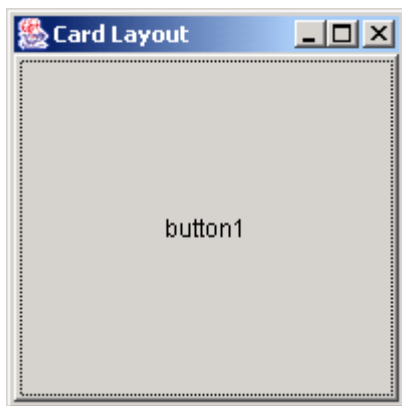


图 10-7 CardLayout

使用 CardLayout 的一般步骤如下：

- (1) 创建 CardLayout 对象作为布局管理器：MyCard=newCardLayout();
- (2) 使用容器的 setLayout()方法为容器设置布局管理器：setLayout(MyCard);
- (3) 调用容器的方法 add()将组件加入容器，同时为组件分配一个字符串的名字，以便布局管理器根据这个名字调用显示这个组件。

add(字符串, 组件);

- (4) 调用 CardLayout 的方法 show(), 根据字符串名字显示一个组件：show(容器名, 字符串); 或按组件加入容器的顺序显示组件，如 first(容器名)方法显示第一个组件，last(容器名)方法显示最后一个组件等。

例 10-5 TestCardLayout.java 使用 CardLayout。

```
import java.awt.*;
import java.awt.event.*;
public class TestCardLayout {
    public static void main(String args[]) {
        final Frame f = new Frame("Card Layout");
        final CardLayout cl = new CardLayout();

        Button[] buttons = new Button[8];
        for( int i=0; i<buttons.length; i++){
            buttons[i] = new Button( "button"+(i+1) );
            buttons[i].addMouseListener( new MouseAdapter(){
                public void mouseClicked( MouseEvent e ){
                    if( e.getModifiers() == InputEvent.BUTTON1_MASK )
                        cl.next( f );
                    else
                        cl.previous( f );
                }
            });
        }
    }
}
```

```

    }
    } );
}

f.setLayout(cl );
for( int i=0; i<buttons.length; i++)
    f.add( ""+(i+1), buttons[i] );

f.setSize(200,200);
f.setVisible(true);
}
}

```

程序中,按钮加入了事件处理,单击鼠标左键到下一组件,单击鼠标右键到前一组件。

10.2.4 GridLayout

GridLayout 是使用较多的布局管理器,其基本布局策略是把容器的空间划分成若干行乘若干列的网格区域,组件就位于这些划分出来的小格中。**GridLayout** 比较灵活,划分多少网格由程序自由控制,而且组件定位也比较精确。

使用 **GridLayout** 布局管理器的一般步骤如下。

(1) 创建 **GridLayout** 对象作为布局管理器。指定划分网格的行数和列数,并使用容器的 **setLayout()** 方法为容器设置这个布局管理器:

```
setLayout(new GridLayout(行数, 列数))
```

(2) 调用容器的方法 **add()** 将组件加入容器。组件填入容器的顺序将按照第一行第一个、第一行第二个,……,第一行最后一个、第二行第一个,……,最后一行最后一个进行。每网格中都必须填入组件,如果希望某个网格为空白,可以为它加入一个空的标签

add(new Label()), 如图 10-8 所示。

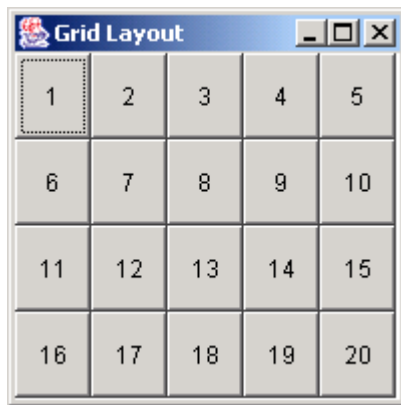


图 10-8 GridLayout

例 10-6 TestGridLayout. Java 使用 GridLayout。

```
import java.awt.*;

public class TestGridLayout {
    public static void main(String args[]) {
        Frame f = new Frame("Grid Layout");
        Button[] buttons = new Button[20];
        for( int i=0; i<buttons.length; i++)
            buttons[i] = new Button( ""+(i+1) );

        f.setLayout(new GridLayout(4,5));

        for( int i=0; i<buttons.length; i++)
            f.add( buttons[i] );

        f.setSize(200,200);
        f.setVisible(true);
    }
}
```

10.2.5 GridBagLayout

GridBagLayout 是五种布局策略中使用最复杂、功能最强大的一种，它是在 GridLayout 的基础上发展而来。因为 GridLayout 中的每个网格大小相同，并且强制组件与网格大小也相同，从而使得容器中的每个组件也都是相同的大小，显得很不自在，而且组件加入容器也必须按照固定的行列顺序，因此不够灵活。在 GridBagLayout 中，可以为每个组件指定其包含的网格个数，可以保留组件原来的大小，可以以任意顺序随意加入容器的任意位置，从而可以真正自由地安排容器中每个组件的大小和位置。但由于 GridBagLayout 的使用较复杂，限于篇幅，不再举例，读者可以查看 JDK 文档。

10.2.6 通过嵌套来设定复杂的布局

由于某一个布局管理器的布局能力有限，在设定复杂布局时，程序会采用容器嵌套的方法，即把一个容器当做一个组件加入另一个容器，这个容器组件可以有自己的组件和自己的布局策略，使整个容器的布局达到应用的需求。如图 10-9 所示。



图 10-9 嵌套的布局

例 10-7 NestedContainer.java 嵌套的布局。

```
import java.awt.*;

public class NestedContainer {
    public static void main(String args[]) {
        Frame f = new Frame("Nested Container");
        Label b0 = new Label("Display Area");
        Panel p = new Panel();

        p.setLayout(new GridLayout(2,2));
        Button b1 = new Button("1");
        Button b2 = new Button("2");
        Button b3 = new Button("3");
        Button b4 = new Button("4");
        p.add(b1);      p.add(b2);
        p.add(b3);      p.add(b4);

        f.add(b0, BorderLayout.NORTH);
        f.add(p, BorderLayout.CENTER);

        f.pack();
        f.setVisible(true);
    }
}
```

程序中, Frame 使用了 BorderLayout, 而其中一个组件 Panel 对象 p 使用了 GridLayout, p 容纳了四个按钮。

10.3 事件处理

10.3.1 事件及事件监听器

Java 中的图形用户界面中, 对于用户的鼠标、键盘操作发生反应, 就必须进行事件处理。这些鼠标、键盘操作等统称为事件 (Event)。对这些事件作出响应的程序, 称为事件处理器 (Event handler)。

1. 事件类 AWTEvent

在 java.awt.event 包中, 有相应的类来表达事件, 如 KeyEvent 及 MouseEvent 等。这些事件类都是从 AWTEvent 类派生而来的。事件类之间的继承关系如图 10-10 所示。

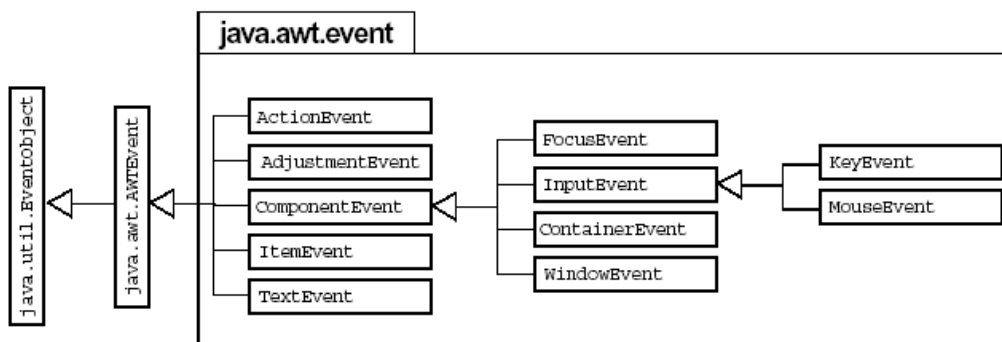


图 10-10 事件类之间的继承关系

事件类中包含有事件相关的信息，最重要的有：

- (1) 事件源（即产生事件的组件），可能通过 `getSource()` 来得到；
- (2) 事件的具体情况，如 `MouseEvent` 的 `getX()`、`getY()` 方法得到鼠标的坐标，`KeyEvent` 的 `getKeyChar()` 得到当前的字符等。

2. 事件处理器 `AWTEventListener`

事件处理器(Event handler)是对事件进行处理的程序，在编程时通过实现事件监听器(Event Listener)来实现对事件的处理。事件监听器是一些事件的接口，这些接口是 `java.awt.AWTEventListener` 的子类。接口中含有相关的方法，如：`MouseMotionListener` 是对鼠标移动事件的处理的接口，它含有两个重要的方法：

- ✧ `void mouseDragged(MouseEvent e);` // 处理鼠标拖动的方法
- ✧ `void mouseMoved(MouseEvent e);` // 处理鼠标移动的方法

在这些方法中，都带一个事件对象作为参数，如 `MouseMotionListener` 的两个方法都带 `MouseEvent` 参数。程序中可以根据这个参数可以得到有关事件的详细信息。

各个事件及其相关方法，列于表 10-1 中。

表 10-1 各个事件及其相关方法

事件类型	接口	接口中的方法
Action 行动事件	ActionListener	actionPerformed(ActionEvent)
Item 条目事件	ItemListener	itemStateChanged(ItemEvent)
Mouse 鼠标事件	MouseListener	mousePressed(MouseEvent) mouseReleased(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mouseClick(MouseEvent)
Mouse Motion 鼠标 移动事件	MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)

续表

事件类型	接口	接口中的方法
Key 键盘事件	KeyListener	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
Focus 焦点事件	FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)
Adjustment 调整	AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)
Component 组件事件	ComponentListener	componentMoved(ComponentEvent) componentHidden(ComponentEvent) componentResized(ComponentEvent) componentShown(ComponentEvent)
Window 窗口事件	WindowListener	windowClosing(WindowEvent) windowOpened(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent)
Container 容器事件	ContainerListener	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
Text 文本事件	TextListener	textValueChanged(TextEvent)

10.3.2 事件监听器的注册

Java 中处理事件的基本方法，是事件监听器的注册，也就是将组件对象与监听器对象相联系。

1. 注册事件监听器

注册事件监听器只需要使用组件对象的 `addXXXXEvent` 方法，它可以指明该对象感兴趣的事件监听器（即实现了某个 `AWTEventListener` 子接口的对象）。这样，当事件源发生了某种类型的事件时，则触发事先已注册过的监听器中相应的处理程序。

例 10-8 TestActionEvent.java 使用 ActionEvent。

```
import java.awt.*;
import java.awt.event.*;

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

```

        Frame f = new Frame("Test");
        Button b = new Button("Press Me!");
        f.add(b);

        Monitor bh = new Monitor();
        b.addActionListener(bh);

        f.pack();
        f.setVisible(true);
    }
}

class Monitor implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("a button has been pressed");
    }
}

```

运行结果如图 10-11 所示。

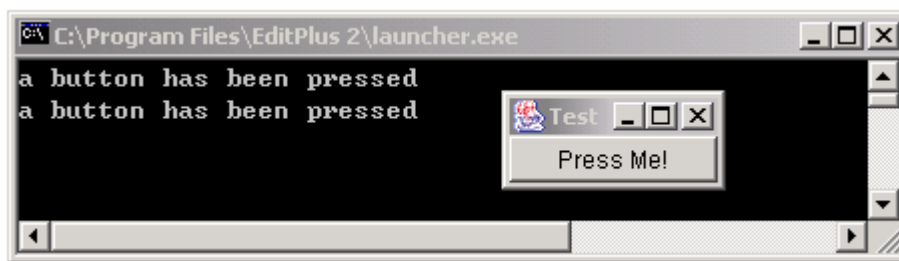


图 10-11 使用 ActionEvent

2. 一个对象注册多个监听器

一般情况下，事件源可以产生多种不同类型的事件，因而可以注册(触发)多种不同类型的监听器。不同对象所能注册的事件监听器参见表 10-2。

表 10-2 不同组件所能注册的事件监听器

组 件	Act	Adj	Cmp	Cnt	Foc	Itm	Key	Mou	MM	Txt	Win
Button	Y		Y		Y		Y	Y	Y		
Canvas			Y		Y		Y	Y	Y		
Checkbox			Y		Y	Y	Y	Y	Y		
CheckboxMenuItem						Y					
Choice			Y		Y	Y	Y	Y	Y		

续表

组 件	Act	Adj	Cmp	Cnt	Foc	Itm	Key	Mou	MM	Txt	Win
Component			Y		Y		Y	Y	Y		
Container			Y	Y	Y		Y	Y	Y		
Dialog			Y	Y	Y		Y	Y	Y		Y
Frame			Y	Y	Y		Y	Y	Y		Y
Label			Y		Y		Y	Y	Y		
List	Y		Y		Y	Y	Y	Y	Y		
MenuItem	Y										
Panel			Y	Y	Y		Y	Y	Y		
Scrollbar		Y	Y		Y		Y	Y	Y		
ScrollPane			Y	Y	Y		Y	Y	Y		
TextArea			Y		Y		Y	Y	Y	Y	
TextField	Y		Y		Y		Y	Y	Y	Y	
Window			Y	Y	Y		Y	Y	Y		Y

其中：

Act: Action 行动事件。

Adj: Adjustment 调整。

Cmp: Component 组件事件。

Cnt: Container 容器事件。

Foc: Focus 焦点事件。

Itm: Item 条目事件。

Key: Key 键盘事件。

Mou: Mouse 鼠标事件。

MM: Mouse Motion 鼠标移动事件。

Txt: Text 文本事件。

Win: Window 窗口事件。

例 10-9 TestMultiListener.java 使用多个事件监听器。

```

import java.awt.*;
import java.awt.event.*;

public class TestMultiListener {
    public static void main(String args[]) {
        Frame f = new Frame("Test");
        TextField msg = new TextField(20);

        Monitor1 m1 = new Monitor1(f);
        Monitor2 m2 = new Monitor2(f, msg);
    }
}

```

```
f.addWindowListener(m1);
f.addMouseMotionListener(m2);

f.add( msg, BorderLayout.SOUTH );
f.setSize(200,160);
f.setVisible(true);
}
}

class Monitor1 implements WindowListener {
    Monitor1( Frame f){
        this.f = f;
    }
    private Frame f;
    public void windowClosing(WindowEvent e){System.exit(0);}
    public void windowOpened(WindowEvent e){}
    public void windowIconified(WindowEvent e){}
    public void windowDeiconified(WindowEvent e){}
    public void windowClosed(WindowEvent e){}
    public void windowActivated(WindowEvent e){}
    public void windowDeactivated(WindowEvent e){}
}

class Monitor2 implements MouseMotionListener {
    Monitor2( Frame f, TextField msg ){
        this.msg = msg;
        this.f = f;
    }
    private TextField msg;
    private Frame f;
    private boolean bDragged = false;
    public void mouseMoved( MouseEvent e ){
        msg.setText( "MouseMoved: " + e.getX() + ", " + e.getY() );
        if ( bDragged){
            f.setCursor( new Cursor( Cursor.DEFAULT_CURSOR ) );
            bDragged = false;
        }
    }
    public void mouseDragged( MouseEvent e ){
```

```

msg.setText( "MouseDraged: " + e.getX() + ", " + e.getY() );
if( ! bDragged ) {
    f.setCursor( new Cursor( Cursor.CROSSHAIR_CURSOR ) );
    bDragged = true;
}
f.getGraphics().drawLine( e.getX(), e.getY(), e.getX(),
    e.getY());
}
}

```

运行结果如图 10-12 所示。

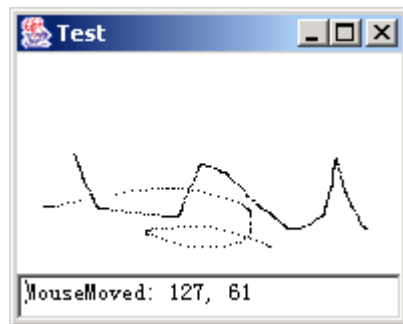


图 10-12 使用多个事件监听器

3. 多个对象注册到一个监听器

一个事件源组件上可以注册多个监听器，针对同一个事件源的同一种事件也可以注册多个监听器，一个监听器可以被注册到多个不同的事件源上。如图 10-13 所示。

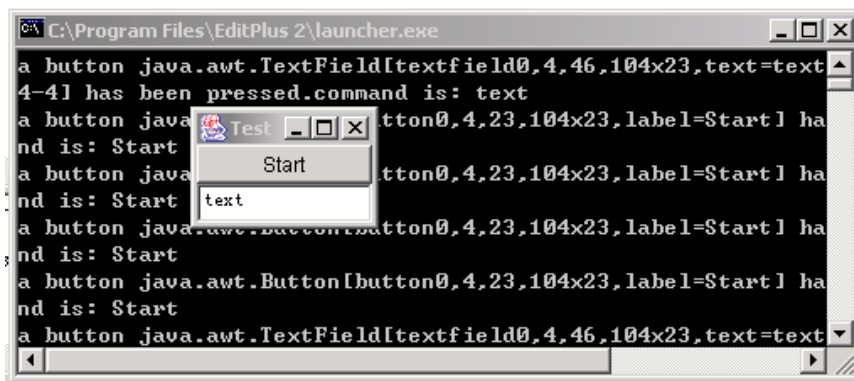


图 10-13 多个组件使用相同的监听器

例 10-10 TestMultiObjectOneListener.java 多个组件使用相同的监听器。

```
import java.awt.*;
```

```
import java.awt.event.*;

public class TestMultiObjectOneListener {
    public static void main(String args[]) {
        Frame f = new Frame("Test");
        Button b1 = new Button("Start");
        TextField b2 = new TextField("text");

        Monitor3 bh = new Monitor3();
        b1.addActionListener(bh);
        b2.addActionListener(bh);
        f.add(b1, "North");
        f.add(b2, "Center");
        f.pack();
        f.setVisible(true);
    }
}

class Monitor3 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("a button " + e.getSource()
            + " has been pressed." +
            "command is: " + e.getActionCommand()
        );
    }
}
```

10.3.3 事件适配器

用实现接口的方法，必须实现接口所规定的方法，如 `WindowListener` 有 7 个方法，即使一些方法不做任何事情，也得书写。为简化编程，针对一些事件监听器接口定义了相应的实现类——事件适配器类（`Adapter`），在适配器类中，实现了相应监听器接口中所有的方法，但不做任何事情。

事件适配器包括如下几种：

- (1) `ComponentAdapter`(组件适配器);
- (2) `ContainerAdapter`(容器适配器);
- (3) `FocusAdapter`(焦点适配器);
- (4) `KeyAdapter`(键盘适配器);
- (5) `MouseAdapter`(鼠标适配器);
- (6) `MouseMotionAdapter`(鼠标运动适配器);

(7) WindowAdapter(窗口适配器)。

在定义监听器类时就可以继承事件适配器类，并只重写所需要的方法。

例 10-11 TestWindowAdapter.java 使用事件适配器。

```
import java.awt.*;
import java.awt.event.*;

public class TestWindowAdapter {
    public static void main(String args[]) {
        Frame f = new Frame("Test");
        TextField msg = new TextField(20);
        Monitor1 m1 = new Monitor1(f);
        f.addWindowListener(m1);
        f.add( msg, BorderLayout.SOUTH );
        f.setSize(200,160);
        f.setVisible(true);
    }
}

class Monitor1 extends WindowAdapter {
    Monitor1( Frame f){
        this.f = f;
    }
    private Frame f;
    public void windowClosing(WindowEvent e){
        System.exit(0);
    }
}
```

10.3.4 内部类及匿名类在事件处理中的应用

在Java事件处理程序中,由于与事件相关的事件监听器的类经常局限于一个类的内部,所以经常使用内部类。而且定义后内部类在事件处理中的使用就实例化一次,所以经常使用匿名类。

例 10-12 TestInnerListener.java 内部类作为事件监听器。

```
import java.awt.*;
import java.awt.event.*;

public class TestInnerListener {
    Frame f = new Frame("内部类测试");
```

```
TextField tf = new TextField(30);

public TestInnerListener(){
    f.add(new Label("请按下鼠标左键并拖动"), "North");
    f.add(tf, "South");

    f.setBackground(new Color(120,175,175));
    f.addMouseMotionListener(new InnerMonitor());
    f.addMouseListener(new InnerMonitor());
    f.setSize(300, 200);
    f.setVisible(true);
}

public static void main(String args[]) {
    Object t = new TestInnerListener();
}

private class InnerMonitor implements MouseMotionListener,
MouseListener {
    public void mouseDragged(MouseEvent e) {
        String s = "鼠标拖动到位置 (" + e.getX() + ", " + e.getY()
            + ")";
        tf.setText(s);
    }

    public void mouseEntered(MouseEvent e) {
        String s = "鼠标已进入窗体";
        tf.setText(s);
    }

    public void mouseExited(MouseEvent e) {
        String s = "鼠标已移出窗体";
        tf.setText(s);
    }

    public void mouseMoved(MouseEvent e) { }
    public void mousePressed(MouseEvent e) { }
    public void mouseClicked(MouseEvent e) { }
    public void mouseReleased(MouseEvent e) { }
}
```

```
}
```

运行结果如图 10-14 所示。

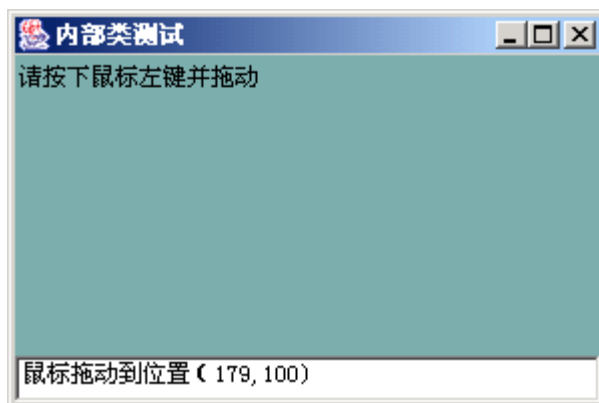


图 10-14 内部类作为事件监听器

例 10-13 TestAnonymous.java 匿名类作为事件监听器。

```
import java.awt.*;
import java.awt.event.*;

public class TestAnonymous {
    Frame f = new Frame("匿名内部类测试");
    TextField tf = new TextField(30);

    public TestAnonymous(){
        f.add(new Label("请按下鼠标左键并拖动"), "North");
        f.add(tf, "South");

        f.addMouseMotionListener(new MouseMotionListener(){
            public void mouseDragged(MouseEvent e) {
                String s = "鼠标拖动到位置 (" + e.getX() + ", "
                    + e.getY() + ")";
                tf.setText(s);
            }
            public void mouseMoved(MouseEvent e) { }
        });
        f.addWindowListener( new WindowAdapter() {
            public void windowClosing( WindowEvent e ){
                System.exit(0);
            }
        })
    }
}
```

```
    });  
    f.setSize(300, 200);  
    f.setVisible(true);  
}  
public static void main(String args[]) {  
    TestAnonymous t = new TestAnonymous();  
}  
}
```

运行结果如图 10-15 所示。

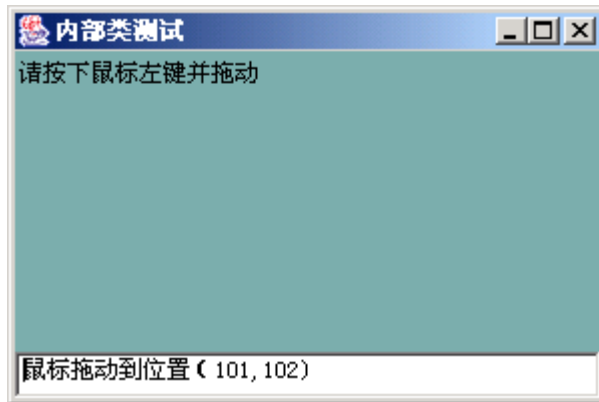


图 10-15 匿名类作为事件监听器

10.4 常用组件的使用

下面将从创建、常用方法和事件响应几个方面逐一介绍常用的 GUI 组件和容器。

10.4.1 标签、按钮与动作事件

1. 标签(Label)

标签是用户不能修改只能查看其内容的文本显示区域，它起到信息说明的作用，每个标签用一个 Label 类的对象表示。

(1) 创建标签。创建标签对象时应同时说明这个标签中的字符串：

```
Label prompt=new Label("请输入一个整数：");
```

(2) 常用方法。setText(新字符串) 设置标签上显示的文本；getText()方法来获得它的文本内容。

(3) 产生事件。标签不能接受用户的输入，所以一般不处理特定的事件，当然它还可以处理许多普通的事件，如 mouse 事件等。

2. 按钮(Button)

按钮一般对应一个事先定义好的功能操作，并对应一段程序。当用户点击按钮时，系统自动执行与该按钮相联系的程序，从而完成预先指定的功能。

(1) 创建。下面的语句用来创建按钮，传递给构造函数的字符串参数指明了按钮上的标签：

```
Button enter=new Button("操作");
```

(2) 常用方法。`getLabel()`方法可以返回按钮标签字符串。

`setLabel(String s)`方法可以把按钮的标签设置为字符串 `s`。

(3) 产生事件。按钮可以引发动作事件，当用户单击一个按钮时就引发了一个动作事件，希望响应按钮引发的动作事件的程序必须把按钮注册给实现了 `ActionListener` 接口的动作事件监听者，同时，为这个接口的 `actionPerformed(ActionEvent e)`方法书写方法体。在方法体中，可以调用 `e.getSource()`方法来获取引发动作事件的按钮对象引用，也可以调用 `e.getActionCommand()`方法来获取按钮的标签或事先为这个按钮设置的命令名。

3. 动作事件(ActionEvent)

`ActionEvent` 类只包含一个事件，即执行动作事件 `ACTION_PERFORMED`。`ACTION_PERFORMED` 是引发某个动作执行的事件。

能够触发这个事件的动作包括如下几种。

- (1) 点击按钮。
- (2) 双击一个列表中的选项。
- (3) 选择菜单项。
- (4) 在文本框中输入回车。

`ActionEvent` 类的重要方法有如下几种。

(1) `public String getActionCommand()`方法。

这个方法返回引发事件的动作的命令名，这个命令名可以通过调用 `setAction-Command()`方法指定给事件源组件，也可以使用事件源的默认命令名。

按钮的默认命令名就是按钮的标签。使用 `getActionCommand()`方法可以区分产生动作命令的不同事件源，使 `actionPerformed()`方法对不同事件源引发的事件区分对待处理(区分事件的事件源也可以使用 `getSource()`方法，但是这样一来处理事件的代码就与 GUI 结合得过于紧密，对于小程序尚可接受，对于大程序则不提倡)。

(2) `public int getModifiers()`方法。

如果发生动作事件的同时用户还按了 `Ctrl`, `Shift` 等功能键，则可以调用这个事件的 `getModifiers()`方法来获得和区分这些功能键，实际上就是把一个动作事件再细分成几个事件，把一个命令细分成几个命令。将 `getModifiers()`方法的返回值与 `ActionEvent` 类的几个静态常量 `ALT_MASK`, `CTRL_MASK`, `SHIFT_MASK`, `META_MASK` 相比较，就可以判断用户按下了哪个功能键。

10.4.2 文本框、文本区域与文本事件

1. 文本事件(TextEvent)

TextEvent 类只包含一个事件，即代表文本区域中文本变化的事件 TEXT_VALUE_CHANGED，在文本区域中改变文本内容。例如，删除字符、键入字符都将引发这个事件。这个事件比较简单，不需要特别判断事件类型的方法和常量。

2. 文本框与文本域(TextField 与 TextArea)

Java 中用于文本处理的基本组件有两种：单行文本框 TextField 和多行文本区域 TextArea，它们都是 TextComponent 的子类。

(1) 创建。

在创建文本组件的同时可以指出文本组件中的初始文本字符串，如下面的语句创建了一个 10 行 45 列的多行文本区域：

```
TextArea textArea1=new TextArea(10, 45);
```

而创建能容纳 8 个字符，初始字符串为“卡号”的单行文本框可以使用如下的语句：

```
TextField name=new TextField("卡号", 8);
```

(2) 常用方法。

- ✎ getText(); // 获得其中的文字
- ✎ setText(String); // 设置其中的文字
- ✎ setEditable(false) ; // 将文本区域设为不能编辑的
- ✎ isEditable(); // 判断当前的文本区域是否可以被编辑
- ✎ select(int start, int end); // 将根据指定的起止位置选定一段文本
- ✎ selectAll(); // 将选定文本区域中的所有文本
- ✎ setSelectionStart(); 和 setSelectionEnd(); // 分别指定选定文本的起、止位置
- ✎ getSelectionStart(); 和 getSelectionEnd(); // 将获得选定文本的起、止位置
- ✎ getSelectedText(); // 实现希望获得选定文本的具体内容

除了继承 TextComponent 类的方法，TextField 还定义了一些自己的特殊方法。例如，某些场合下(如输入密码时)希望文本区域中的内容不如实回显在屏幕上，可以调用如下的方法：

```
TextField tf=new TextField("输入密码");  
tf.setEchoChar('*');
```

这样，TextField 中的每个字符(无论中西文)都被回显成一个星号'*'，保证看不到其中的实际字符。另外，TextField 还定义了 echoCharIsSet()方法确认当前文本框是否处于不回显状态；getEchoChar()方法获得当前文本框不回显的屏蔽字符。

除了继承 TextComponent 类的方法，TextArea 也定义了两个特殊的方法 append(Strings) 和 insert(Strings, int index)。append(Strings)方法在当前文本区域已有文本的后面添加字符串参数 s 指定的文本内容；insert(String s, int index)方法将字符串 s 插入到已有文本的指定序号处。

(3) 事件响应。

`TextField` 和 `TextArea` 的事件响应首先由它们的父类 `TextComponent` 决定, 所以先讨论 `TextComponent` 的事件响应。`TextComponent` 可以引发一种事件: 当用户修改文本区域中的文本, 如做文本的添加、删除、修改等操作时, 将引发 `TextEvent` 对象代表的文本改变事件。在此基础上 `TextField` 还比 `TextArea` 多产生一种事件, 当用户在文本框中按回车键时, 将引发代表动作事件的 `ActionEvent` 事件, `TextArea` 却不能产生 `ActionEvent` 事件, 也没有 `addActionListener()` 这个方法。

如果希望响应上述两类事件, 则需要把文本框加入实现了 `TextListener` 接口的文本改变事件监听者和实现了 `ActionListener` 接口的动作事件监听者:

```
textField1.addTextListener(this);
```

```
textField1.addActionListener(this);
```

在监听者内部分别定义响应文本改变事件和动作事件的方法:

```
public void textValueChanged(TextEvent e);
```

```
public void actionPerformed(ActionEvent e);
```

就可以响应文本框引发的文本改变事件和动作事件。对于文本改变事件, 调用方法 `e.getSource()` 可以获得引发该事件的文本框的对象引用。调用这个文本框的方法, 可以获得改变后的文本内容:

```
String afterChange=((TextField)e.getSource()).getText();
```

对于动作事件, 同样可以通过调用 `e.getSource()` 方法获得用户输入回车的那个文本框的对象引用。

10.4.3 单、复选按钮, 列表与选择事件

1. 选择事件(ItemEvent)

`ItemEvent` 类只包含一个事件, 即代表选择项的选中状态发生变化的事件 `ITEM_STATE_CHANGED`。引发这类事件的动作包括:

- (1) 改变列表类 `List` 对象选项的选中或不选中状态。
- (2) 改变下拉列表类 `Choice` 对象选项的选中或不选中状态。
- (3) 改变复选按钮类 `Checkbox` 对象的选中或不选中状态。
- (4) 改变检测盒菜单项 `CheckboxMenuItem` 对象的选中或不选中状态。

`ItemEvent` 类的主要方法有以下几种。

```
(1) public ItemSelectable getItemSelectable();
```

此方法返回引发选中状态变化事件的事件源, 例如选项变化的 `List` 对象或选中状态变化的 `Checkbox` 对象, 这些能引发选中状态变化事件的都是实现了 `ItemSelectable` 接口的类的对象, 包括 `List` 对象、`Choice` 对象、`Checkbox` 对象等。`getItemSelectable()` 方法返回的就是这些类的对象引用。

```
(2) public Object getItem();
```

此方法返回引发选中状态变化事件的具体选择项, 例如用户选中的 `Choice` 中的具体

item, 通过调用这个方法可以知道用户选中了哪个选项。

(3) `public int getStateChange();`

此方法返回具体的选中状态变化类型, 它的返回值在 `ItemEvent` 类的几个静态常量列举的集合之内。

`ItemEvent.SELECTED`: 代表选项被选中。

`ItemEvent.DESELECTED`: 代表选项被放弃不选。

2. 复选按钮(Checkbox)

(1) 创建。

复选按钮又称为检测盒, 用 `Checkbox` 类的对象表示。创建复选按钮对象时可以同时指明其文本说明标签, 这个文本标签简要地说明了检测盒的意义和作用。

```
Checkbox backg=new Checkbox("背景色");
```

(2) 常用方法。

每个复选按钮都只有两种状态: 被用户选中的 `check` 状态, 未被用户选中的 `uncheck` 状态, 任何时刻复选按钮都只能处于这两种状态之一。查看用户是否选择了复选按钮, 可以调用 `Checkbox` 的方法 `getState()`, 这个方法的返回值为布尔量。若复选按钮被选中, 则返回 `true`, 否则返回 `false`。调用 `Checkbox` 的另一个方法 `setState()` 可以用程序设置是否选中复选按钮。

(3) 事件响应。

当用户点击检测盒使其选中状态发生变化时就会引发 `ItemEvent` 类代表的选择事件。如果这个检测盒已经用如下的语句:

```
backg.addItemListener(this);
```

把自身注册给 `ItemEvent` 事件的监听者 `ItemListener`。

则系统会自动调用这个 `ItemListener` 中的方法:

```
public void itemStateChanged(ItemEvent e)
```

响应复选按钮的状态改变。所以实际实现了 `ItemListener` 接口的监听者, 例如包容复选按钮的容器, 需要具体实现这个方法。这个方法的方法体通常包括这样的语句: 调用选择事件的方法 `e.getItemSelectable()` 获得引发选择事件的事件源对象引用, 再调用 `e.getState()` 获取选择事件之后的状态。也可以直接利用事件源对象自身的方法进行操作。

注意: `getItemSelectable()` 方法的返回值是实现了 `Selectable` 接口的对象, 需要把它强制转化成真正的事件源对象类型。

总的来说, 需要响应复选按钮事件的情况不多, 通常只需要知道一个确切的时刻。例如, 用户单击某个按钮的时刻、复选按钮所处的最终状态等。这可以通过调用复选按钮自身的方法很方便地获得。

3. 单选按钮组(CheckboxGroup)

(1) 创建。

`Checkbox` 只能提供“二选一”的机制。要想实现“多选一”, 可以选择单选按钮组。

单选按钮组是一组 `Checkbox` 的集合，用 `CheckboxGroup` 类的对象表示。每个 `Checkbox` 对应一种可能的取值情况。

把 `CheckboxGroup` 加入容器时需要把其中的每个单选按钮逐个加入到容器中，而不能使用 `CheckboxGroup` 对象一次性地加入。

(2) 常用方法。

单选按钮的选择是互斥的，即当用户选中了组中的一个按钮后，其他按钮将自动处于未选中状态。调用 `CheckboxGroup` 的 `getSelectedCheckbox()` 方法可以获知用户选择了哪个按钮，这个方法返回用户选中的 `Checkbox` 对象，再调用该方法 `getLabel()` 就可以知道用户选择了什么信息。同样，通过调用 `CheckboxGroup` 的 `setSelectedCheckbox()` 方法，可以在程序中指定单选按钮组中的那个按钮。

另外，也可以直接使用按钮组中的 `Checkbox` 单选按钮的方法。

(3) 事件响应。

`CheckboxGroup` 类不是 `java.awt.*` 包中的类，它是 `Object` 的直接子类，所以按钮组不能响应事件，但是按钮组中的每个按钮可以响应 `ItemEvent` 类的事件。由于单选按钮组中的每个单选按钮都是 `Checkbox` 的对象，它们对事件的响应与检测盒对事件的响应相同。

4. Choice(下拉列表)

(1) 创建。

下拉列表也是“多选一”的输入界面。与单选按钮组利用单选按钮把所有选项列出的方法不同，下拉列表的所有选项被折叠收藏起来，只显示最前面的或被用户选中的一个。

如果希望看到其他的选项，只需单击下拉列表右边的下三角按钮就可以“下拉”出一个罗列了所有选项的长方形区域。

创建下拉列表包括创建和添加选项两个步骤——创建下拉列表、为下拉列表加入选项。

(2) 常用方法。

下拉列表的常用方法包括获得选中选项的方法、设置选中选项的方法、添加和去除下拉列表选项的方法。

`getSelectedIndex()` 方法将返回被选中的选项的序号(下拉列表中第一个选项的序号为 0，第二个选项的序号为 1，依此类推)。`getSelectedItem()` 方法将返回被选中选项的标签文本字符串。`select(int index)` 方法和 `select(String item)` 方法使程序选中指定序号或文本内容的选项。`add(String item)` 方法和 `insert(String item, int index)` 方法分别将新选项 `item` 加在当前下拉列表的最后或指定的序号处。`remove(int index)` 方法和 `remove(String item)` 方法把指定序号或指定标签文本的选项从下拉列表中删除。`removeAll()` 方法将把下拉列表中的所有选项删除。

(3) 事件响应。

下拉列表可以产生 `ItemEvent` 代表的选择事件。如果把选项注册给实现了接口 `ItemListener` 的监听者，即使用 `addItemListener()`，则当用户单击下拉列表的某个选项做出选择时，系统自动产生一个 `ItemEvent` 类的对象包含这个事件的有关信息，并把该对象作为实际参数传递给被自动调用的监听者的选择事件响应方法。

```
public void itemStateChanged(ItemEvent e);
```

在这个方法里，调用 `e.getItemSelectable()` 就可以获得引发当前选择事件的下拉列表事

件源，再调用此下拉列表的有关方法，就可以得知用户具体选择了哪个选项。

```
String selectedItem = ((Choice)c.getItemSelectable()).  
getSelectedItem();
```

这里对 `e.getItemSelectable()` 方法的返回值进行了强制类型转换，转换成 `Choice` 类的对象引用后方可调用 `Choice` 类的方法。

5. List(列表)

(1) 创建。

列表也是列出一系列的选择项供用户选择，但是列表可以实现“多选多”，即允许复选。在创建列表时，同样应该将它的各项选择项(称为列表项 `Item`)加入到列表中去，如下面的语句：

```
MyList=new List(4, true);  
MyList.add("北京");  
MyList.add("上海");
```

将创建一个包括两个地址选项的列表，`List` 对象的构造函数的第一个参数表明列表的高度，可以一次同时显示几个选项，第二个参数表明列表是否允许复选，即同时选中多个选项。

(2) 常用方法。

如果想获知用户选择了列表中的哪个选项，可以调用 `List` 对象的 `getSelectedItem()` 方法，这个方法返回用户选中的选择项文本。与单选按钮不同的是，列表中可以有复选和多选，所以 `List` 对象还有一个方法 `getSelectedItems()` 方法，该方法返回一个 `String` 类型的数组，里面的每个元素是一个被用户选中的选择项，所有的元素包括了所有被用户选中的选项。

除了可以直接返回被选中的选项的标签字符串，还可以获得被选中选项的序号。在 `List` 里面，第一个加入 `list` 的选项的序号是 0，第二个是 1，依此类推。`getSelectedIndex()` 方法将返回被选中的选项的序号，`getSelectedIndexes()` 方法将返回由所有被选中的选项的序号组成的整型数组。

`select(int index)` 和 `deselect(int index)` 方法可以使指定序号处的选项被选中或不选中；`add(String item)` 方法和 `add(String item, int index)` 方法分别将标签为 `item` 的选项加入列表的最后面或加入列表的指定序号处；`remove(String item)` 方法和 `remove(int index)` 方法与之相反，将拥有指定标签的选项或指定序号处的选项从列表中移出。这两个方法使得程序可以动态调整列表所包含的选择项。

(3) 事件响应。

列表可以产生两种事件：当用户单击列表中的某一个选项并选中它时，将产生 `ItemEvent` 类的选择事件；当用户双击列表中的某个选项时，将产生 `ActionEvent` 类的动作事件。

如果程序希望对这两种事件都做出响应，就需要把列表分别注册给 `ItemEvent` 的监听者 `ItemListener` 和 `ActionEvent` 的监听者 `ActionListener`。

```
MyList.addItemListener(this);
```

```
MyList.addActionListener(this);
```

并在实现了监听者接口的类中分别定义响应选择事件的方法和响应动作事件的方法。

```
public void itemStateChanged(ItemEvent e); // 响应单击的选择事件
```

```
public void actionPerformed(ActionEvent e); // 响应双击的动作事件
```

这样，当列表上发生了单击或双击动作时，系统就自动调用上述两个方法来处理相应的选择或动作事件。

通常在 `itemStateChanged(ItemEvent e)` 方法里，会调用 `e.getItemSelectable()` 方法获得产生这个选择事件的列表(List)对象的引用，再利用列表对象的方法 `getSelectedIndex()` 或 `getSelectedItem()` 就可以方便地得知用户选择了列表的哪个选项。与 `Checkbox` 的使用方法类似，`e.getItemSelectable()` 的返回值需要先强制类型转化成 `List` 对象，然后才能调用 `List` 类的方法。

而在 `actionPerformed(ActionEvent e)` 方法里，调用 `e.getSource()` 可以得到产生此动作事件的 `List` 对象引用，同样要使用强制类型转换。

```
((List)e.getSource());
```

调用 `e.getActionCommand()` 可以获得事件的选项的字符串标签，在列表单选的情况下，相当于执行：

```
((List)e.getSource()).getSelectedItem();
```

的运行结果。

注意：列表的双击事件并不能覆盖单击事件。当用户双击一个列表选项时，首先产生一个单击的选项事件，然后再产生一个双击的动作事件。如果定义并注册了两种事件的监听者，`itemStateChanged()` 方法和 `actionPerformed()` 方法将分别被先后调用。

10.4.4 调整事件与滚动条

1. 调整事件 (AdjustmentEvent)

`AdjustmentEvent` 类只包含一个事件——`ADJUSTMENT_VALUE_CHANGED` 事件。与 `ItemEvent` 事件引发的离散状态变化不同，`ADJUSTMENT_VALUE_CHANGED` 是 GUI 组件状态发生连续变化的事件，引发这类事件的具体动作如下。

(1) 操纵滚动条(`Scrollbar`)改变其滑块位置。

(2) 操纵用户自定义的 `Scrollbar` 对象的子类组件，改变其滑块位置。

`AdjustmentEvent` 类的主要方法如下。

(1) `public Adjustable getAdjustable();`

这个方法返回引发状态变化事件的事件源，能够引发状态变化事件的事件源都是实现了 `Adjustable` 接口的类。

`Adjustable` 接口是 `java.awt` 包中定义的一个接口，一个方法的返回值类型被标志为一个接口，代表这个方法将返回一个实现了这个接口的类的对象。

(2) `public int getAdjustmentType();`

这个方法返回状态变化事件的状态变化类型,其返回值在 `AdjustmentEvent` 类的几个静态常量所列举的集合之内。

- ☞ `AdjustmentEvent.BLOCK_DECREMENT`: 代表点击滚动条下方引发块状下移的动作。
- ☞ `AdjustmentEvent.BLOCK_INCREMENT`: 代表点击滚动条上方引发块状上移的动作。
- ☞ `AdjustmentEvent.TRACK`: 代表拖动滚动条滑块的动作。
- ☞ `AdjustmentEvent.UNIT_DECREMENT`: 代表点击滚动条下三角按钮引发最小单位下移的动作。
- ☞ `AdjustmentEvent.UNIT_INCREMENT`: 代表点击滚动条上三角按钮引发最小单位上移的动作。
- ☞ 通过调用 `getAdjustmentType()`方法并比较其返回值,就可以得知用户发出的哪种操作引发了哪种连续的状态变动。

(3) `public int getValue();`

调用 `getValue()`方法可以返回状态变化后的滑块对应的当前数值。滑块是可以连续调整的,调整将引发 `AdjustmentEvent` 事件,`getValue()`方法可以返回调整后滑块对应的最新数值。

2. 滚动条(Scrollbar)

(1) 创建。

滚动条是一种比较特殊的 GUI 组件,它能够接受并体现连续的变化,称为“调整”。创建 `Scrollbar` 类的对象将创建一个含有滚动槽、增加箭头、减少箭头和滑块(也称为气泡)的滚动条。

构造函数的第一个参数说明新滚动条的方向,使用常量 `Scrollbar.HORIZONTAL` 将创建横向滚动条,使用常量 `Scrollbar.VERTICAL` 将创建纵向滚动条。

构造函数的第二个参数用来说明滑块最初的显示位置,它应该是一个整型量。

构造函数的第三个参数说明滑块的大小,对滑块滚动同时引起文本区域滚动的情况。

滑块大小与整个滚动槽长度的比例应该与窗口中可视的文本区域与整个文本区域的比例相当,对于滑块滚动不引起文本区域滚动的情况,可把滑块大小设为 1。

构造函数的第四个参数说明滚动槽代表的最小数据。

构造函数的第五个参数说明滚动槽代表的最大数据。

(2) 常用方法。

对于新创建的滚动条,设置它的单位增量和块增量还需要调用如下的方法。

- ☞ `mySlider.setUnitIncrement(1);`
- ☞ `mySlider.setBlockIncrement(50);`

`setUnitIncrement(int)`方法指定滚动条的单位增量,即用户单击滚动条两端的三角按钮时代表的数据改变;`setBlockIncrement(int)`方法指定滚动条的块增量,即用户单击滚动槽时代表的数据改变。与上面两个方法相对应,滚动条类还定义了 `getUnitIncrement()`方法和 `getBlockIncrement()`方法来分别获取滚动条的单位增量和块增量。

`getValue()`方法返回当前滑块位置代表的整数值，当用户利用滚动条改变滑块在滚动槽中的位置时，`getValue()`方法的返回值将相应随之改变。

(2) 事件响应。

滚动条可以引发 `AdjustmentEvent` 类代表的调整事件，当用户通过各种方式改变滑块位置从而改变其代表的数值时，都会引发调整事件。

程序要响应滚动条引发的调整事件，必须首先把这个滚动条注册给实现了 `AdjustmentListener` 接口的调整事件监听者 `mySlider`。 `addAdjustmentListener(this)`。

调整事件监听者中用于响应调整事件的方法是：

```
public void adjustmentValueChanged(AdjustmentEvent e);
```

这个方法通常需要调用 `e.getAdjustable()`来获得引发当前调整事件的事件源，如滚动条。另一个有用的方法是 `AdjustmentEvent` 类的方法 `getValue()`，它与滚动条的 `getValue()`方法功能相同，`e.getValue()`方法可以返回调整事件后的数值。调用 `e.getAdjustmentType()`方法可以知道当前调整事件的类型，即用户使用何种方式改变了滚动条滑块的位置。具体方法是把这个方法的返回值与 `AdjustmentEvent` 类的几个静态常量相比较。

- ✎ `AdjustmentEvent.BLOCK_INCREMENT`：块增加。
- ✎ `AdjustmentEvent.BLOCK_DECREMENT`：块减少。
- ✎ `AdjustmentEvent.UNIT_INCREMENT`：单位增加。
- ✎ `AdjustmentEvent.UNIT_DECREMENT`：单位减少。
- ✎ `AdjustmentEvent.TRACK`：用鼠标拖动滑块移动。

10.4.5 鼠标、键盘事件与画布

1. 鼠标事件(MouseEvent)

`MouseEvent` 类和 `KeyEvent` 类都是 `InputEvent` 类的子类，`InputEvent` 类不包含任何具体的事件，但是调用 `InputEvent` 类的 `getModifiers()`方法，并把返回值与 `InputEvent` 类的几个静态整型常量 `ALT_MASK`，`CTRL_MASK`，`SHIFT_MASK`，`META_MASK`，`BUTTON1_MASK`，`BUTTON2_MASK`，`BUTTON3_MASK` 相比较，就可以得知用户在引发 `KeyEvent` 事件时是否同时按下了功能键，或者用户在单击鼠标时单击的是哪个鼠标键。

`MouseEvent` 类包含如下的若干个鼠标事件，分别用 `MouseEvent` 类的同名静态整型常量标志。

- ✎ `MOUSE_CLICKED`：代表鼠标点击事件。
- ✎ `MOUSE_DRAGGED`：代表鼠标拖动事件。
- ✎ `MOUSE_ENTERED`：代表鼠标进入事件。
- ✎ `MOUSE_EXITED`：代表鼠标离开事件。
- ✎ `MOUSE_MOVED`：代表鼠标移动事件。
- ✎ `MOUSE_PRESSED`：代表鼠标按钮按下事件。
- (7) `MOUSE_RELEASED`：代表鼠标按钮松开事件。

调用 `MouseEvent` 对象的 `getID()`方法并把返回值与上述各常量比较，就可以知道用户

引发的是哪个具体的鼠标事件。例如，假设 `mouseEvt` 是 `MouseEvent` 类的对象，下面的语句将判断它代表的事件是否是 `MOUSECLICKED`：

```
if(mouseEvt.getID()==MouseEvent.MOUSE_CLICKED)
```

不过一般不需要这样处理，因为监听 `MouseEvent` 事件的监听者 `MouseListener` 和 `MouseMotionListener` 中有七个具体方法，分别针对上述的七个具体鼠标事件，系统会分辨鼠标事件的类型并自动调用相关的方法，所以编程者只需把处理相关事件的代码放到相关的方法里即可。

`MouseEvent` 类的主要方法如下。

- ☞ `public int getX();` // 返回发生鼠标事件的 X 坐标
- ☞ `public int getY();` // 返回发生鼠标事件的 Y 坐标
- ☞ `public Point getPoint();` // 返回 `Point` 对象，包含表示鼠标事件发生的坐标点
- ☞ `public int getClickCount();` // 返回鼠标点击事件的点击次数

前面所说 `MouseListener` 和 `MouseMotionListener` 的几个具体的事件处理方法，都以 `MouseEvent` 类的对象为形式参数。通过调用 `MouseEvent` 类的上述方法，这些事件处理方法可以得到引发它们的鼠标事件的具体信息。

2. 键盘事件(KeyEvent)

`KeyEvent` 类包含如下三个具体的键盘事件，分别对应 `KeyEvent` 类的几个同名的静态整型常量。

- ☞ `KEY_PRESSED`：代表键盘按键被按下的事件。
- ☞ `KEY_RELEASED`：代表键盘按键被放开的事件。
- ☞ `KEY_TYPED`：代表按键被敲击的事件。
- ☞ `KeyEvent` 类的主要方法如下。

(1) `public char getKeyChar();`

返回引发键盘事件的按键对应的 Unicode 字符，如果这个按键没有 Unicode 字符与之相对应，则返回 `KeyEvent` 类的一个静态常量 `KeyEvent.CHAR_UNDEFINED`。

(2) `public String getKeyText();`

返回引发键盘事件的按键的文本内容，典型的返回值有“A”，“Home”，“F3”，等等。

与 `KeyEvent` 事件相对应的监听者接口是 `KeyListener`，这个接口中定义了如下的三个抽象方法，分别与 `KeyEvent` 中的三个具体事件类型相对应。

- ☞ `public void keyPressed(KeyEvent e);`
- ☞ `public void keyReleased(KeyEvent e);`
- ☞ `public void keyTyped(KeyEvent e);`

可见，事件类中的事件类型名与对应的监听者接口中的抽象方法名很相似，也体现了二者之间的响应关系。凡是实现了 `KeyListener` 接口的类，都必须具体实现上述的三个抽象方法，把用户程序对这三种具体事件的响应代码放在实现后的方法体中，这些代码里通常需要用到实参 `KeyEvent` 对象 `e` 的若干信息，这可以通过调用 `e` 的方法，如 `getSource()`，`getKeyChar()`等来实现。

3 . 画布(Canvas)

画布是一个用来画图的矩形背景组件，在画布里可以像在 Applet 里那样绘制各种图形，也可以响应鼠标和键盘事件。

(1) 创建。

Canvas 的构造函数没有参数，所以使用简单的语句就可以创建一个画布对象。

```
Canvas myCanvas=new Canvas();
```

在创建了 Canvas 对象之后，还应该调用 setSize()方法确定这个画布对象的大小，否则用户在运行界面中将看不到这个画布。

(2) 常用方法。

确定 Canvas 大小的常用方法只有一个：public void paint(Graphics g)，用户程序重载这个方法就可以实现在 Canvas 上面绘制有关图形。

(3) 事件响应。

Canvas 对象与 Applet 相似，可以引发键盘和鼠标事件。

10.4.6 Frame 与窗口事件

除了 Applet 和 Panel 这组无边框容器，Container 还有一组有边框的容器的子类，包括 Window, Frame, Dialog 和 FileDialog, 其中 Window 是所有有边框容器的父类,但是 Window 本身并无边框，算是有边框容器的一个例外。这一组里的其他容器都是有边框可独立存在的容器。

1 . Frame

在前面的例子中已经使用过 Frame 这种容器，它是 Java 中最重要、最常用的容器之一，是 Java Application 程序的图形用户界面容器。

Frame 可以作为一个 Application 的最外层容器，也可以被其他容器创建并弹出成为独立的容器。但是无论哪种情况，Frame 都作为最顶层容器存在，不能被其他容器所包含。

Frame 有自己的外边框和自己的标题，创建 Frame 时可以指定其窗口标题。

```
Frame(String title);
```

也可以使用专门的方法 getTitle()和 setTitle(String)来获取或指定 Frame 的标题。新创建的 Frame 是不可见的，需要使用 setVisible(boolean)方法，并使用实际参数 true 使之可见。

每个 Frame 在其右上角都有三个控制图标，分别代表将窗口最小化、最大化和关闭的操作，其中最小化和最大化操作 Frame 可自动完成，而关闭窗口的操作不能通过点击关闭图标实现，需要程序专门书写有关的代码。常用的关闭窗口的方法有三个：一个是设置一个按钮，当用户点击按钮时关闭窗口；第二个方法是对 WINDOWS_CLOSING 事件做出响应，关闭窗口；第三个方法是使用菜单命令。前一种方法需要专门的按钮，而后一种方法实现 WindowListener 接口所需的代码较多，无论使用何种方法，都需要用到关闭 Frame 的 dispose()方法。

向 Frame 窗口中添加和移出组件使用的方法与其他容器相同，也是 add()和 remove()，Frame 可以引发 WindowEvent 类代表的所有七种窗口事件。

2. 窗口事件(WindowEvent)

WindowEvent 类包含如下几个具体窗口事件。

- (1) WINDOW_ACTIVATED: 代表窗口被激活(在屏幕的最前方待命)。
- (2) WINDOW_DEACTIVATED: 代表窗口失活(其他窗口被激活后原活动窗口失活)。
- (3) WINDOW_OPENED: 代表窗口被打开。
- (4) WINDOW_CLOSED: 代表窗口已被关闭(指已关闭后)。
- (5) WINDOW_CLOSING: 代表窗口正在被关闭(指关闭前, 如点击窗口的关闭按钮)。
- (6) WINDOW_ICONIFIED: 代表使窗口最小化成图标。
- (7) WINDOW_DEICONIFIED: 代表使窗口从图标恢复。

WindowEvent 类的主要方法有 `public Window getWindow()`, 此方法返回引发当前 WindowEvent 事件的具体窗口, 与 `getSource()`方法返回的是相同的事件引用。但是 `getSource()`的返回类型为 `Object`, 而 `getWindow()`方法的返回值是具体的 `Window` 对象。

10.4.7 Panel 与容器事件

1. Container 类

Container 类是一个抽象类, 里面包含了所有容器组件都必须具有的方法和功能。

(1) `add()`: Container 类中有多个经过重载的 `add()`方法, 其作用都是把 `Component` 组件, 可能是一个基本组件, 也可能是另一个容器组件, 加入到当前容器中, 每个被加入容器的组件根据加入的先后顺序获取一个序号。

(2) `getComponent(int index)`与 `getComponent(int x, int y)`: 这两个方法分别获得指定序号或指定(x, y)坐标点处的组件。

(3) `remove(Component)`与 `remove(int index)`: 将指定的组件或指定序号的组件从容器中移出。

(4) `removeAll()`: 将容器中所有的组件移出。

(5) `setLayout()`: 设置容器的布局管理器。

Container 可以引发 `ContainerEvent` 类代表的容器事件。当容器中加入或移出一个组件时, 容器将分别引发 `COMPONENT_ADDED` 和 `COMPONENT_REMOVED` 两种容器事件。希望响应容器事件的程序应该实现容器事件的监听者接口 `ContainerListener`, 并在监听者内部具体实现该接口中用来处理容器事件的两个方法。

☞ `public void componentAdded(ContainerEvent e);` // 响应向容器中加入组件事件的方法

☞ `public void componentRemoved(ContainerEvent e);` // 响应从容器中移出组件的方法

在这两个方法内部, 可以调用实际参数 `e` 的方法 `e.getContainer()`获得引发事件的容器对象的引用, 这个方法的返回类型为 `Container`; 也可以调用 `e.getChild()`方法获得事件发生时被加入或移出容器的组件, 这个方法的返回类型为 `Component`。

2 . 容器事件(ContainerEvent)

ContainerEvent 类包含两个具体的与容器有关的事件。

(1) COMPONENT_ADDED: 把组件加入当前容器对象。

(2) COMPONENT_REMOVED: 把组件移出当前容器对象。

ContainerEvent 类的主要方法有两种。

☞ public Container getContainer(); // 返回引发容器事件的容器对象

☞ public Component getChild(); // 返回引发容器事件时被加入或移出的组件对象

3 . Panel

Panel 属于无边框容器。无边框容器包括 Panel 和 Applet, 其中 Panel 是 Container 的子类, Applet 是 Panel 的子类。

Panel 是最简单的容器, 它没有边框或其他的可见的边界, 它不能被移动、放大、缩小或关闭。一个程序不能使用 Panel 作为它的最外层的图形界面的容器, 所以 Panel 总是作为一个容器组件被加入到其他的容器, 如 Frame, Applet 等中去。Panel 也可以进一步包含另一个 Panel, 使用 Panel 的程序中总是存在着容器的嵌套。使用 Panel 的目的通常是为了层次化管理图形界面的各个组件, 同时使组件在容器中的布局操作更为方便。程序不能显式地指定 Panel 的大小, Panel 的大小是由其中包含的所有组件, 以及包容它的那个容器的布局策略和该容器中的其他组件决定的。

容器的嵌套是 Java 程序 GUI 界面设计和实现中经常需要使用到的手段, 实现这一类 GUI 界面时, 应该首先明确各容器之间的包含嵌套关系。

例如, 对于一个 Java Applet 程序, 最外层的容器是一个 Applet, 其中包含了两个组件: 第一个组件是一个 Panel 对象 p1; 第二个组件是另一个 Panel 对象 p2。p1 中包含了一个标签 prompt 1 和第三个 Panel 对象 p3, p3 中包含一个标签 prompt 3 和一个按钮 btn。这里所有的容器都使用默认的 FlowLayout 布局策略。

当用户单击按钮 btn 时, 程序从 p1 移去组件 p3 并引发 CONTAINER_REMOVED 事件, 程序响应这个事件后在状态条中显示移去组件容器的信息。

Applet 是一种特殊的 Panel, 它是 Java Applet 程序的最外层容器, 但是 Java Applet 并不是完整独立的程序, 它事实上是 WWW 浏览器中的一个控件, 是作为浏览器的一部分依赖于浏览器而存在的, 所以 Java Applet 可以依赖浏览器的窗口来完成放大、缩小、关闭等功能。至于 Java Applet 程序本身, 它只要负责它所拥有的 Applet 容器中的那部分无边框区域就足够了。Applet 容器的默认布局策略与其父类 Panel 一致, 都是 FlowLayout, 但是 Applet 容器中还额外定义了一些用来与浏览器交互的方法, 如 init(), start(), stop()等。

10.4.8 组件事件、焦点事件与对话框

1 . 组件事件(ComponentEvent)

这个类是所有低级事件的根类, 一共包含四个具体事件, 可以用 ComponentEvent 类的几个静态常量来表示。

- (1) `ComponentEvent.COMPONENT_HIDDEN`: 代表隐藏组件的事件。
- (2) `ComponentEvent.COMPONENT_SHOWN`: 代表显示组件的事件。
- (3) `ComponentEvent.COMPONENT_MOVED`: 代表移动组件的事件。
- (4) `ComponentEvent.COMPONENT_RESIZED`: 代表改变组件大小的事件。

把调用 `getID()` 方法的返回值与上述常量相比较, 就可以知道 `ComponentEvent` 对象所代表的具体事件。

2. 焦点事件(FocusEvent)

`FocusEvent` 类包含两个具体事件, 分别对应这个类的两个同名静态整型常量。

- (1) `FOCUS_GAINED`: 代表获得了注意的焦点。
- (2) `FOCUS_LOST`: 代表失去了注意的焦点。

一个 GUI 的对象必须首先获得注意的焦点, 才能被进一步操作。

区域必须首先获得注意的焦点, 才能接受用户键入的文字。一个窗口只有先获得了注意的焦点, 其中的菜单才能被选中。获得注意的焦点将使对象被调到整个屏幕的最前面并处于待命的状态, 是默认操作的目标对象, 而失去注意焦点的对象则被调到屏幕的后面并可能被其他的对象遮挡。

3. 对话框(Dialog)

与 `Frame` 一样, `Dialog` 是有边框、有标题的独立存在的容器, 并且不能被其他容器所包容; 但是 `Dialog` 不能作为程序的最外层容器, 也不能包含菜单条。与 `Window` 一样, `Dialog` 必须隶属于一个 `Frame` 并由这个 `Frame` 负责弹出。 `Dialog` 通常起到与用户交互的对话框的作用。

`Dialog` 的构造函数有四种重载方式, 其中最复杂的为:

```
Dialog(Frame parent, String title, boolean isModal);
```

第一个参数指明新创建的 `Dialog` 对话框隶属于哪个 `Frame` 窗口, 第二个参数指明新建 `Dialog` 对话框的标题, 第三个参数指明该对话框是否有模式的。所谓“有模式”的对话框, 是那种一旦打开后用户必须对其做出响应的对话框, 例如, 对话框询问用户是否确认删除操作, 此时程序处于暂停状态, 除非用户回答了对对话框的问题, 否则是不能使用程序的其他部分的, 所以带有一定的强制性质; 而无模式对话框则没有这种限制, 用户完全可以不理睬这个打开的对话框而去操作程序的其他部分, 默认情况下对话框都是无模式的。新建的对话框使用默认的 `BorderLayout`, 它是不可见的, 可以使用 `show()` 方法显示它。

对于已经创建的对话框, 还可以用 `setModal(boolean isModal)` 方法来改变其模式属性, 或者使用 `boolean isModal()` 方法来判断它是否是一个有模式对话框。另外, `Dialog` 还有获得和修改其对话框标题的方法 `getTitle()` 和 `setTitle(String newTitle)`; 还有加入和移出组件的方法 `add(Component)` 和 `remove(Component)`。

`Dialog` 还有一个子类 `FileDialog`, 用来表示一种特殊的用来搜索目录和文件, 并打开或保存特定文件的对话框, 在 9.2.2 节中曾经用到过。

10.5 绘图、图形和动画

本节主要介绍如何利用 Java 类库中的类及其方法来绘制用户自定义的图形界面成分。编程人员可以利用这些方法自由地绘制图形和文字，也可以将已经存在的图形、动画等加载到当前程序中来。

10.5.1 绘制图形

绘制图形和文字将要用类 `Graphics`。`Graphics` 是 `java.awt` 包中一个类，其中包括了很多绘制图形和文字的方法。

1. 获得 `Graphics` 对象

对于一个图形用户界面中的组件，可以用 `getGraphics()` 方法来得到一个 `Graphics` 对象，它相当于组件的绘图环境，利用它可以进行各种绘图操作。

对于 `Applet` 运行时，执行它的浏览器会自动为它创建一个 `Graphics` 类的实例，利用这个实例，这个实例会传递给 `paint()` 方法。

在 `Applet` 及 `Application` 程序中，常用 `Canvas` 来进行绘图。与 `Applet` 一样，`paint()` 方法也会带一个 `Graphics` 参数，通过覆盖 `paint()` 方法，就可以绘制各种图形。

例 10-14 `SimpleMouseWidthVector.java` 用鼠标点击画图。

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class SimpleMouseWidthVector extends Applet
{
    private java.util.Vector points = new java.util.Vector();
    public void init(){
        addMouseListener( new MouseAdapter() {
            public void mousePressed( MouseEvent e ){
                points.add( new Point(e.getX(), e.getY() ) );
                repaint();
            }
        } );
    }
    public void update(Graphics g){
        paint(g);
    }
    public void paint(Graphics g){
```

```
        for( int i =0; i<points.size(); i++){
            Point p = (Point)points.elementAt(i);
            g.drawString("x", p.x, p.y);
        }
    }

    public static void main(String args[]) {
        Frame f = new Frame("CelayTree");
        SimpleMouseWidthVector p = new SimpleMouseWidthVector();
        p.init();
        p.start();
        f.add("Center", p);
        f.setSize(400, 300);
        f.addWindowListener( new WindowAdapter(){
            public void windowClosing(WindowEvent e)
            { System.exit(0); }
        });
        f.show();
    }
}
```

运行结果如图 10-16 所示。

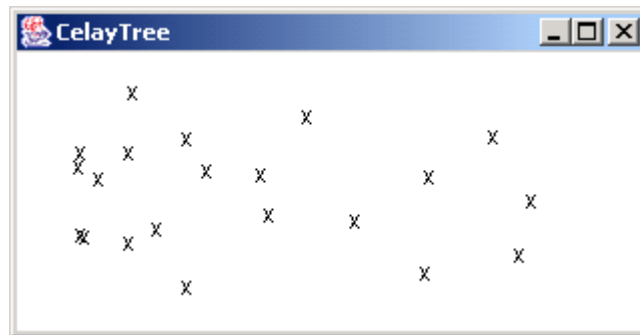


图 10-16 用鼠标点击画图

2 . Graphics 的绘图方法

绘制图形的方法很多，表 10-3 列出了常用的 Graphics 方法，更详细的解释可以查阅 JDK 文档。

表 10-3 常用的 Graphics 方法

画三维矩形	draw3DRect(int x, int y, int width, int height, boolean raised)
画弧	drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)

画文字	drawBytes(byte[] data, int offset, int length, int x, int y)
画文字	drawChars(char[] data, int offset, int length, int x, int y)
画直线	drawLine(int x1, int y1, int x2, int y2)
画椭圆	drawOval(int x, int y, int width, int height)
画多边形	drawPolygon(int[] xPoints, int[] yPoints, int nPoints)
画多边形	drawPolygon(Polygon p)
画折线	drawPolyline(int[] xPoints, int[] yPoints, int nPoints)
画矩形	drawRect(int x, int y, int width, int height)
画圆角矩形	drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)
画文字	drawString(AttributedCharacterIterator iterator, int x, int y)
画文字	drawString(String str, int x, int y)
画填充三维矩形	fill3DRect(int x, int y, int width, int height, boolean raised)
画填充弧	fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)
画填充椭圆	fillOval(int x, int y, int width, int height)
画填充多边形	fillPolygon(int[] xPoints, int[] yPoints, int nPoints)
画填充多边形	FillPolygon(Polygon p)
画填充矩形	fillRect(int x, int y, int width, int height)
画填充圆角矩形	fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)

3. 几个辅助类

除了 Graphics 类, Java 中还定义了一些用来表示几何图形的类, 对绘制用户自定义成分也很有帮助。例如, 利用 Point 表示一个像素点; 利用 Dimension 类表示宽和高; 利用 Rectangle 类表示一个矩形; 利用 Polygon 类表示一个多边形; 利用 Color 类表示颜色等。后面的例子中将利用这些系统定义的类绘图。

例 10-15 Draw_r_cos2th.java 画图, 如图 10-17 所示。

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class Draw_r_cos2th extends Applet
{
    public void paint(Graphics g){
        double w = getSize().width/2;
        double h = getSize().height/2;
        g.setColor( Color.blue );
        for( double th =0; th<10; th+=0.003){
            double r = Math.cos(16*th)*h;
```

```
        double x = r * Math.cos( th ) + w;  
        double y = r * Math.sin( th ) + h;  
        g.drawOval( (int)x-1, (int)y-1, 3, 3);  
    }  
}  
  
public static void main(String args[]) {  
    Frame f = new Frame("Draw");  
    Draw_r_cos2th p = new Draw_r_cos2th();  
    p.init();  
    p.start();  
    f.add(p);  
    f.setSize(400, 300);  
    f.addWindowListener( new WindowAdapter(){  
        public void windowClosing(WindowEvent e)  
        { System.exit(0);}  
    });  
    f.show();  
}
```

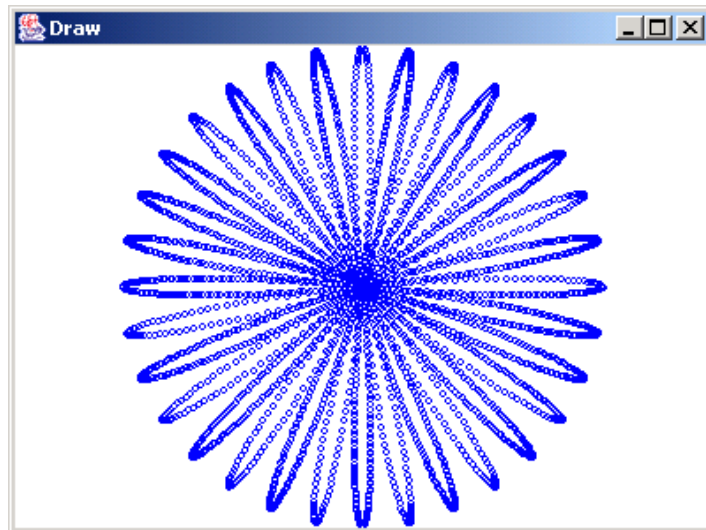


图 10-17 画图

10.5.2 显示文字

从前面的例子中可以知道，Graphics 类的方法 drawString()可以在屏幕的指定位置显示

一个字符串。Java 中还有一个类 `Font`，使用它可以获得更加丰富多彩和逼真精确的字体显示效果。

一个 `Font` 类的对象表示了一种字体显示效果，包括字体类型、字型和字号。下面的语句用于创建一个 `Font` 类的对象：

```
Font MyFont=new Font("TimesRoman", Font. BOLD, 12);
```

`MyFont` 对应的是 12 磅 `TimesRoman` 类型的黑体字，其中指定字型时需要用到 `Font` 类的三个常量：`Font.PLAIN`，`Font.BOLD`，`Font.ITALIC`。

如果希望使用该 `Font` 对象，则可以利用 `Graphics` 类的 `setFont()` 方法：

```
g.setFont(MyFont);
```

如果希望指定控制组件，如按钮或文本框中的字体效果，则可以使用控制组件的方法 `setFont()`。如，设 `btn` 是一个按钮对象，则语句如下：

```
btn.setFont(MyFont);
```

将把这个按钮上显示的标签的字体改为 12 磅的 `TimesRoman` 黑体字。

另外，与 `setFont()` 方法相对的 `getFont()` 方法将返回当前 `Graphics` 或组件对象使用的字体。

例 10-16 DrawFonts.java 使用字体。

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class DrawFonts extends Applet
{
    public void paint(Graphics g){
        double w = getSize().width/2;
        double h = getSize().height/2;
        g.setColor( Color.blue );
        GraphicsEnvironment ge = GraphicsEnvironment.
            getLocalGraphicsEnvironment();
        Font[] fonts = ge.getAllFonts();
        for( int i=0;i< fonts.length; i++ ){
            String name = fonts[i].getName();
            g.setFont( new Font( name, Font.PLAIN, 12 ) );
            g.drawString( name, 10, 15*i );
        }
    }

    public static void main(String args[]) {
        Frame f = new Frame("Draw");
```

```
DrawFonts p = new DrawFonts();  
p.init();  
p.start();  
f.add(p);  
f.setSize(400, 300);  
f.addWindowListener( new WindowAdapter(){  
    public void windowClosing(WindowEvent e)  
    { System.exit(0);}  
});  
f.show();  
}  
}
```

运行结果如图 10-8 所示。

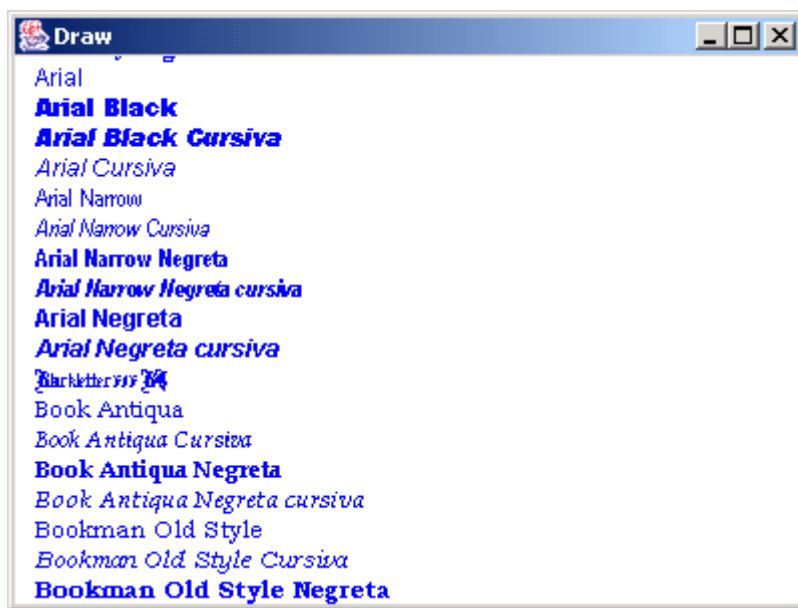


图 10-18 使用字体

10.5.3 控制颜色

Applet 中显示的字符串或图形的颜色可以用 Color 类的对象来控制,每个 Color 对象代表一种颜色,用户可以直接使用 Color 类中定义好的颜色常量,也可以通过调配红、绿、蓝三色的比例创建自己的 Color 对象。

Color 类中定义有如下的三种构造函数:

☞ public Color(int Red, int Green, int Blue);


```
    public Color(float Red, float Green, float Blue);
```

```
    public Color(int RGB)。
```

不论使用哪个构造函数创建 `Color` 对象，都需要指定新建颜色中 R(红)、G(绿)、B(蓝) 三色的比例。在第一个构造函数中通过三个整型参数指定 R, G, B, 每个参数的取值范围在 0~255 之间；第二个构造函数通过三个浮点参数指定 R, G, B, 每个参数的取值范围在 0.0~1.0 之间；第三个构造函数通过一个整型参数指明其 RGB 三色比例, 这个参数的 0~7 位(取值范围为 0~255)代表红色的比例, 8~15 位代表绿色的比例, 16~23 位代表蓝色的比例。

调用 `Graphics` 对象的 `setColor()` 方法可以把当前默认的颜色修改成新建的颜色, 使此后调用该 `Graphics` 对象完成的绘制工作, 如绘制图形、字符串等, 都使用这个新建颜色:

```
g.setColor(blueColor);
```

除了创建自己的颜色, 也可以直接使用 `Color` 类中定义好的颜色常量, 如:

```
g.setColor(Color.cyan);
```

`color` 类中共定义了 13 种静态颜色常量, 包括 `black`, `orange`, `pink`, `grey` 等, 使用时只需以 `Color` 为前缀, 非常方便。

对于 GUI 的控制组件, 它们有四个与颜色有关的方法分别用来设置和获取组件的背景色和前景色:

```
    public void setBackground(Color c);
```

```
    public Color getBackground();
```

```
    public void setForeground();
```

```
    public Color getForeground();
```

10.5.4 显示图像

由于图像的数据量要远远大于图形, 所以一般不在程序中自行绘制图像, 而是把已经存在于本机硬盘或网络某地的二进制图像文件直接调入内存。图像文件有多种格式, 如 `bmp` 文件、`gif` 文件、`tiff` 文件等, 其中 `gif` 是 Internet 上常用的图像文件格式。

Java 中可以利用 `Graphics` 类的 `drawImage()` 方法显示图像。

10.5.5 实现动画效果

动画曾是 Java Applet 最吸引人的特性之一。用 Java 实现动画的原理与放映动画片类似, 取若干相关的图像或图片, 顺序、连续地在屏幕上先显示, 后擦除, 循环往复就可以获得动画的效果。

例 10-17 DrawImageAnimator.java 实现动画效果。

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.io.*;
```

```
public class DrawImageAnimator extends Frame
{
    public DrawImageAnimator( String s ){
        super(s);
        final String dir = "D:\\tds\\MyPictures\\时装世界\\时装世界";
        String [] files = new File(dir).list();
        int num = files.length<=10? files.length : 10;
        images = new Image[num];
        for( int i=0; i<num; i++ )
            images[i] = Toolkit.getDefaultToolkit().createImage(
                dir +"\\\"+ files[i]);
        setSize( 400, 700 );
        show();
        addWindowListener( new WindowAdapter(){
            public void windowClosing(WindowEvent e ){
                bStop = true;
                System.exit(0);
            }
        });
        thread = new MyThread();
        thread.start();
    }

    public void paint(Graphics g){

        g.drawImage( images[ curImage ], 0, 0, this );
    }

    private Image[] images;
    private int curImage;
    private boolean bStop;
    private MyThread thread;
    class MyThread extends Thread
    {
        public void run(){
            while( ! bStop ){
                repaint();
                try{ sleep(1000); } catch(InterruptedException e){}
            }
        }
    }
}
```

```
        curImage++;
        if( curImage==images.length ) curImage=0;
    }
}

public static void main(String args[]) {
    Object f = new DrawImageAnimator("DrawImageAnimator");
}
}
```

运行结果如图 10-19 所示。



图 10-19 实现动画效果

10.6 Applet

Applet(小程序)是一种很重要的 Java 程序,是工作在 Internet 的浏览器上的 Java 程序。编写 Applet 小程序必须要用到 java.applet 包中的 Applet 类。java.applet.Applet 是 java.awt.Panel 的子类,如图 10-20 所示。Applet 的默认布局是 FlowLayout。

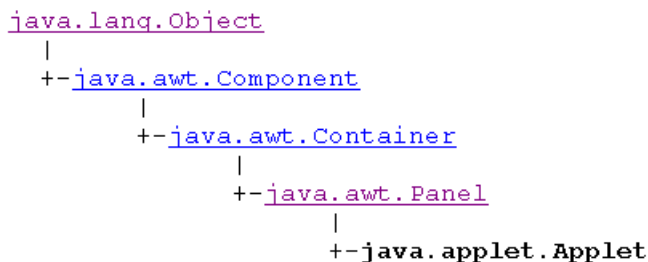


图 10-20 Applet 的继承关系

10.6.1 Applet 的基本工作原理

Applet 是一种特殊的 Java 程序。作为解释型语言, Java 的字节码程序需要一个专门的解释器来执行它。对于 Java Application 来说, 这个解释器是独立的软件, 如 JDK 的 `java.exe`, VJ++ 的 `jview.exe` 等; 而对于 Java Applet 来说, 这个解释器就是 Internet 网的浏览器软件, 或者更确切地, 就是兼容 Java 的 Internet 浏览器。

Applet 的基本工作原理是这样的: 编译好的字节码文件(. class 文件)保存在特定的 WWW 服务器上, 同一个或另一个 WWW 服务器上保存着嵌入了该字节码文件名的 HTML 文件。当某一个浏览器向服务器请求下载嵌入了 Applet 的 HTML 文件时, 该文件从 WWW 服务器上下载到客户端, 由 WWW 浏览器解释 HTML 中的各种标记, 按照其约定将文件中的信息以一定的格式显示在用户屏幕上。当浏览器遇到 HTML 文件中的特殊标记 `<applet>`, 表明它嵌有一个 Applet 时, 浏览器会根据这个 Applet 的名字和位置自动把字节码从 WWW 服务器上下载到本地, 并利用浏览器本身拥有的 Java 解释器直接执行该字节码。

从某种意义上来说, Applet 有些类似于组件或控件。与独立的 Application 不同, Applet 程序所实现的功能是不完全的, 需要与浏览器中已经预先实现好的功能结合在一起才能构成一个完整的程序。例如, Applet 不需要建立自己的主流程框架 `main()`, 因为浏览器会自动为它建立和维护主流程; 不一定要有自己专门的图形界面, 因为它可以直接借用浏览器已有的图形界面。Applet 所需要做的, 是接收浏览器发送给它的消息或事件, 如鼠标移动、击键等, 并做出及时的响应。另外, 为了协调与浏览器的合作过程, Applet 中有一些固定的只能由浏览器在特定时刻和场合调用的方法。

10.6.2 Applet 类

在前面的章节中曾说过, Applet 程序在结构上必须创建这样的一个用户类, 其父类是系统的 Applet 类。正是通过这个 Applet 类的子类, 才能完成 Applet 与浏览器的配合。

下面的语句是典型的 Applet 程序中的一部分。

```
import java.applet.Applet;
public class MyApplet extends Applet;
```

1. Applet 类

Applet 类是 Java 类库中的一个重要系统类，存在于 `java.applet` 包中。从类继承结构上来说，Applet 类应该属于构建用户图形界面的 `java.awt` 包，但是由于 Applet 类特殊，系统专门为它建立了一个包。

Applet 类是 Java 的另一个系统类 `java.awt.Panel` 的子类，Panel 是 Container 的一种。它有如下的作用。

- (1) 包容和排列其他的界面元素，如按钮、对话框或其他容器。
- (2) 响应它所包容范围之内的事件，或把事件向更高层次传递。

Applet 在拥有上述作用的基础上，还具有一些与浏览器和 Applet 生命周期有关的专门方法。

2 . Applet 类的主要方法

用户自己定义的 Applet 子类是 Java Applet 程序的标志。在实际运行中，浏览器在下载字节码的同时，会自动创建一个用户 Applet 子类的实例，并在适当事件发生时自动调用该实例的几个主要方法。

(1) init()方法。

init()方法用来完成主类实例的初始化工作。Applet 的字节码文件从 WWW 服务器端下载后，浏览器将创建一个 Applet 类的实例并调用它从 Applet 类那里继承来的 init()方法。用户程序可以重载父类的 init()方法，定义一些必要的初始化操作，如创建和初始化程序运行所需要的对象实例，把图形或字体加载入内存，设置各种参数，加载图形和声音并播放等。

(2) start()方法。

start()方法用来启动浏览器运行 Applet 的主线程。浏览器在调用 init()方法初始化 Applet 类的实例之后，接着将自动调用 start()方法启动运行该实例上的流程、用户程序可以重载 Applet 类的 start()方法，加入当前实例被激活时欲实现的相关功能，如启动一个动画，完成参数传递等。

除了在 init()初始化之后被调用，start()方法在 Applet 被重新启动时也会被系统自动调用。一般有两种情况造成 Applet 重新启动：一是用户使用了浏览器的 Reload 操作；二是用户将浏览器转向了其他的 HTML 页面后又返回。总之，当包含 Applet 的 HTML 页面被重新加载时，其中的 Applet 实例就会被重新启动并调用 start()方法，但是 init()方法只被调用一次。

(3) paint()方法。

paint()方法的主要作用是在 Applet 的界面中显示文字、图形和其他界面元素。它也是浏览器可自动调用的 Applet 类的方法。导致浏览器调用 paint()方法的事件主要有如下三种。

- ☞ Applet 被启动之后，自动调用 paint()来重新描绘自己的界面。
- ☞ Applet 所在的浏览器窗口改变时，例如窗口放大、缩小、移动或被系统的其他部分遮挡、覆盖后又重新显示在屏幕的最前方等。这些情况都要求 Applet 重画它的界面，此时浏览器就自动调用 paint()方法来完成此项工作。

☞ Applet 的其他相关方法被调用时，系统也会相应地调用 `paint()` 方法。例如，当 `repaint()` 方法被调用时，系统将首先调用 `update()` 方法将 Applet 实例所占用的屏幕空间清空，然后调用 `paint()` 方法进行重画。

与前面方法不同的是，`paint()` 方法有一个固定的参数——Graphics 类的对象 `g`。

Graphics 类是用来完成一些较低级的图形用户界面操作的类，其中包括了画圆、点、线、多边形，以及显示简单文本等方法。当一个 Applet 类实例被初始化并启动时，浏览器将自动生成一个 Graphics 类的实例 `g`，并把 `g` 作为参数传递给 Applet 类实例的 `paint()` 方法、`paint()` 方法调用实例 `g` 的相关方法，就可以绘制出 Applet 的界面。程序只要重载系统定义的 `paint()` 方法，就可以使 Applet 界面显示预定画面。

(4) `stop()` 方法。

`stop()` 方法类似于 `start()` 方法的逆操作。当用户浏览其他 WWW 页，或者切换到其他系统应用时，浏览器将暂停执行 Applet 的主线程。在暂停 Applet 之前，浏览器将首先自动调用 Applet 类的 `stop()` 方法。用户程序可以重载 Applet 类的 `stop()` 方法，完成一些必要的操作，如中止 Applet 的动画操作等。

(5) `destroy()` 方法。

当用户退出浏览器时，浏览器中运行的 Applet 实例也相应被消灭，即被内存删除。在消灭 Applet 之前，浏览器会自动调用 Applet 实例的 `destroy()` 方法来完成一些释放资源、关闭连接之类的操作。例如，终止所有当前 Applet 实例所建立并启动的子线程等。至于 Applet 实例本身，由于它是由浏览器创建的，最后也由浏览器来删除，不需要在 `destroy()` 方法中特别定义。

实际上，上述 Applet 由浏览器自动调用的主要方法 `init()`、`start()`、`stop()` 和 `destroy()` 分别对应了 Applet 从初始化、启动、暂停到消亡的生命周期的各个阶段。

10.6.3 HTML 文件参数传递

Applet 是需要嵌入在 HTML 文件中并依赖浏览器运行的程序。它的编程阶段，从源代码编写到编译生成字节码，都与 Java Application 相差不大，但是要想调试和运行 Applet，必须与 HTML 文档相配合。

前面已经简单介绍过，HTML 是超文本标记语言，它通过各种各样的标记来编排超文本信息。在 HTML 文档中嵌入 Applet 同样需要使用一组约定好的特殊标记。如 `<APPLET>` 和 `</APPLET>` 是嵌入 Applet 的标记，其中至少需包括三个参数：`code`、`height` 和 `width`。在 `<APPLET>` 标记中还可以使用其他一些可选的参数。

(1) `codebase`：当 Applet 字节码文件的保存位置与它所嵌入的 HTML 文档不同时，应使用参数 `codebase` 来指明字节码文件的位置，这个位置应使用 URL 的格式。

(2) `alt`：如果解释 HTML 页面的是一个不包含 Java 解释器的浏览器，那么它将不能执行字节码，而是把 `alt` 参数指明的信息显示给用户。

(3) `align`：表示 Applet 界面区域在浏览器窗口的对齐情况。

HTML 文件可以向它所嵌入的 Applet 传递参数，从而使这个 Applet 的运行更加灵活。这个任务是通过 HTML 文件的另一个专门标记 `<PARAM>` 来完成的。

10.6.4 Applet 的局限

Applet 可以在一定程度上与它的运行环境（浏览器）相交互，例如可以调用 Applet 方法的 `getAppletContext()` 来访问浏览器，然后用 `showStatus()` 在浏览器窗口底部的状态栏上显示一条信息，即：

```
getAppletContext().showStatus(evt + "")
```

与普通 Application 不同，Applet 出于安全缘故，并不能随意访问所有资源，Applet 的功能受到许多限制，例如：

(1) 一个 Applet 不能接触到本地的磁盘，这意味着不能在本地磁盘上写和读；

(2) Applet 只能从它所在的服务器下载数据，而不能从其他服务器下载。

但可以通过用户的信任，来解除经过签名的 Applet 的这些限制。

例 10-18 Applet 时钟。本程序是一个综合了线程、图形用户界面等多方面的应用。

```
import java.util.*;
import java.awt.*;
import java.applet.*;
import java.text.*;
import java.awt.event.*;

public class Clock2 extends Applet implements Runnable {
    Thread timer;           // 线程
    int lastxs, lastys, lastxm,
        lastym, lastxh, lastyh; // 线的位置
    SimpleDateFormat formatter; // 日期格式
    String lastdate;           // 显示的时期
    Font clockFaceFont;        // 字体
    Date currentDate;          // 当前时间
    Color handColor;           // 指针颜色
    Color numberColor;         // 数字颜色

    public void init() {
        int x,y;
        lastxs = lastys = lastxm = lastym = lastxh = lastyh = 0;
        formatter = new SimpleDateFormat ("EEE MMM dd hh:mm:ss yyyy",
            Locale.getDefault());
        currentDate = new Date();
        lastdate = formatter.format(currentDate);
        clockFaceFont = new Font("Serif", Font.PLAIN, 14);
        handColor = Color.blue;
    }
}
```

```
        numberColor = Color.darkGray;

        try {
            setBackground(new Color(Integer.parseInt(getParameter
                ("bgcolor"),16)));
        } catch (Exception E) { }
        try {
            handColor
Color(Integer.parseInt( getParameter("fgcolor1"),16));
        } catch (Exception E) { }
        try {
            numberColor
Color(Integer.parseInt(getParameter ("fgcolor2"),16));
        } catch (Exception E) { }
        resize(300,300);           // Set clock window size
    }

    public void circle(int x0, int y0, int r, Graphics g) {
        g.drawOval( x0-r, y0-r, r*2, r*2 );
    }

    // Paint 是程序的主要部分
    public void paint(Graphics g) {
        int xh, yh, xm, ym, xs, ys, s = 0, m = 10, h = 10, xcenter,
        ycenter;
        String today;

        currentDate = new Date();
        SimpleDateFormat formatter = new SimpleDateFormat
        ("s",Locale.getDefault());
        try {
            s = Integer.parseInt(formatter.format(currentDate));
        } catch (NumberFormatException n) {
            s = 0;
        }
        formatter.applyPattern("m");
        try {
            m = Integer.parseInt(formatter.format(currentDate));
        } catch (NumberFormatException n) {
```



```

        m = 10;
    }
    formatter.applyPattern("h");
    try {
        h = Integer.parseInt(formatter.format(currentDate));
    } catch (NumberFormatException n) {
        h = 10;
    }
    formatter.applyPattern("EEE MMM dd HH:mm:ss yyyy");
    today = formatter.format(currentDate);
    xcenter=80;
    ycenter=55;

    // 计算指针的坐标
    // a= s* pi/2 - pi/2 (to switch 0,0 from 3:00 to 12:00)
    // x = r(cos a) + xcenter, y = r(sin a) + ycenter

    xs = (int)(Math.cos(s * 3.14f/30 - 3.14f/2) * 45 + xcenter);
    ys = (int)(Math.sin(s * 3.14f/30 - 3.14f/2) * 45 + ycenter);
    xm = (int)(Math.cos(m * 3.14f/30 - 3.14f/2) * 40 + xcenter);
    ym = (int)(Math.sin(m * 3.14f/30 - 3.14f/2) * 40 + ycenter);
    xh = (int)(Math.cos((h*30 + m/2) * 3.14f/180 - 3.14f/2) * 30
        + xcenter);
    yh = (int)(Math.sin((h*30 + m/2) * 3.14f/180 - 3.14f/2) * 30
        + ycenter);

    // 画圆及数字

    g.setFont(clockFaceFont);
    g.setColor(handColor);
    circle(xcenter,ycenter,50,g);
    g.setColor(numberColor);
    g.drawString("9",xcenter-45,ycenter+3);
    g.drawString("3",xcenter+40,ycenter+3);
    g.drawString("12",xcenter-5,ycenter-37);
    g.drawString("6",xcenter-3,ycenter+45);

    // 除去旧线，画新线

```

```
g.setColor(getBackground());
if (xs != lastxs || ys != lastys) {
    g.drawLine(xcenter, ycenter, lastxs, lastys);
    g.drawString(lastdate, 5, 125);
}
if (xm != lastxm || ym != lastym) {
    g.drawLine(xcenter, ycenter-1, lastxm, lastym);
    g.drawLine(xcenter-1, ycenter, lastxm, lastym); }
if (xh != lastxh || yh != lastyh) {
    g.drawLine(xcenter, ycenter-1, lastxh, lastyh);
    g.drawLine(xcenter-1, ycenter, lastxh, lastyh); }
g.setColor(numberColor);
g.drawString("", 5, 125);
g.drawString(today, 5, 125);
g.drawLine(xcenter, ycenter, xs, ys);
g.setColor(handColor);
g.drawLine(xcenter, ycenter-1, xm, ym);
g.drawLine(xcenter-1, ycenter, xm, ym);
g.drawLine(xcenter, ycenter-1, xh, yh);
g.drawLine(xcenter-1, ycenter, xh, yh);
lastxs=xs; lastys=ys;
lastxm=xm; lastym=ym;
lastxh=xh; lastyh=yh;
lastdate = today;
currentDate=null;
}

public void start() {
    timer = new Thread(this);
    timer.start();
}

public void stop() {
    timer = null;
}

public void run() {
    Thread me = Thread.currentThread();
    while (timer == me) {
```

```
        try {
            Thread.currentThread().sleep(100);
        } catch (InterruptedException e) {
        }
        repaint();
    }
}

public void update(Graphics g) {
    paint(g);
}

public String getAppletInfo() {
    return "Title: A Clock \n";
}

public String[][] getParameterInfo() {
    String[][] info = {
        {"bgcolor", "hexadecimal RGB number", "The background
        color. Default is the
        color of your browser."},
        {"fgcolor1", "hexadecimal RGB number", "The color of the
        hands and dial.
        Default is blue."},
        {"fgcolor2", "hexadecimal RGB number", "The color of the
        seconds hand and
        numbers. Default is dark gray."}
    };
    return info;
}

public static void main(String [] args) // 加入 main,使之能当
Application 应用
{
    Frame f = new Frame();
    f.setSize(450,300 );
    Clock2 p = new Clock2();
    f.add ( p );
    f.setVisible( true );
}
```

```

        p.init();
        p.start();
        f.addWindowListener( new WindowAdapter(){
            public void windowClosing(WindowEvent
                e){ System.exit(0); }
        });
    }

}

```

运行结果如图 10-21 所示。



图 10-21 Applet 时钟

下面是嵌入了 Applet 的 HTML 网页。

```

<HTML>
<HEAD>
    <TITLE>A Clock </TITLE>
</HEAD>
<BODY>
    <h1>A Clock </h1>
    <hr>
    <applet codebase="." code="Clock2.class" width=170 height=150>
        alt="钟表"
        <param name=bgcolor value="000000">
        <param name=fgcolor1 value="ff0000">
        <param name=fgcolor2 value="ff00ff">
    </applet>
    <p>
        <a href="Clock2.java">源程序</a>.

```

```
</BODY>
</HTML>
```

10.7 SwingGUI 组件

10.7.1 Swing 的特点

Swing 是第二代 GUI 开发工具集。javax.swing 包被列入 Java 的基础类库(JFC)，Swing 建立在 AWT，Java2D，Accessibility 等的基础上，如图 10-22 所示。

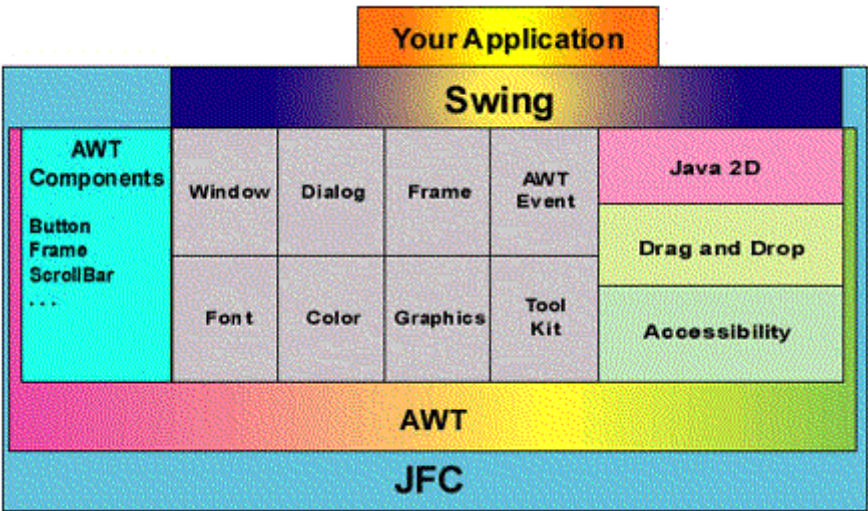


图 10-22 Swing 及其在 JFC 中所处的位置

与 AWT 相比，Swing 具有更好的可移植性，Swing 提供了更完整的组件，增加了许多功能。此外，Swing 引入了许多新的特性和能力。

- ✎ Swing 组件是 Bean，因此它们可以更方便地用到各种集成开发环境。
- ✎ Swing 提供了一个完全的 UI 组件集合。
- ✎ 所有的组件都是很小巧的（没有“重量级”组件被使用），Swing 提供了更好的跨平台性。
- ✎ Swing 同样支持“可插入外观和效果”物，UI 的外观可以在不同的平台和不同的操作系统上被动态地改变以符合用户的期望。它甚至可以创造我们自己的外观和效果。
- ✎ 双缓存和为平整屏幕重新画线的自动重画批次。
- ✎ 自动支持键盘操作（而 AWT 中需要编写更多的代码以支持键盘操作）。
- ✎ 更好地支持滚动，可以简单地将组件加到一个 JScrollPane 中即可。
- ✎ 拖放支持。

- ☞ 大部分组件自动地支持文本、图形、工具提示(Tooltip)。这样，组件使用更方便，组件外观效果也更好。
- ☞ 新的布局管理：Springs & Struts 及 BoxLayout。
- ☞ 更多特殊的组件，例如 JcolorChooser, JFileChooser, JPasswordField, JHTMLPane（完成简单的 HTML 格式化和显示），以及 JTextPane（一个支持格式化，字处理和图像的文字编辑器）。它们都非常易用。
- ☞ 分裂控制：一个间隔物式的分裂条，允许动态地处理其他组件的位置。
- ☞ JLayeredPane 和 JInternalFrame 一起被用来在当前帧中创建子帧，以产生多文件接口（MDI）应用程序。
- ☞ 自定义光标。
- ☞ JToolBar API 提供的可拖动的浮动工具条。

10.7.2 几种 Swing 组件介绍

Swing 组件在构造、事件处理等方面，与 AWT 相似，下面仅就一些 AWT 组件在 Swing 中的等价类进行简单介绍。所有的 Swing 组件位于 javax.swing 包中，组件的名称的首字母都是 J。JComponent 类是所有 SwingGUI 组件的父类，基本上，对应每个 java.awt 组件都存在一个 javax.swing 的“J 组件”。

1 . JApplet

javax.Swing.JApplet 是 java.applet.Applet 的子类，与 JApplet 小程序配合使用的 HTML 文件和与 Applet 小程序配合使用的 HTML 文件没有什么差别。

JApplet 与 Applet 的差别在于 Applet 的默认布局策略是 FlowLayout，而 JApplet 的默认布局策略是 BorderLayout。另外，向 JApplet 中加入 swing 组件时不能直接用 add() 方法，而必须先使用 JApplet 的方法 getContentPane() 获得一个 Container 对象，再调用这个 Container 对象的 add() 方法将 JComponent 及其子类对象加入到 JApplet 中。

2 . JButton

相对于 Button 类，JButton 类新增了很多非常实用的功能。例如，在 Swing 按钮上显示图标，在不同状态使用不同的 Swing 按钮图标，为 Swing 按钮加入提示信息等。

（1）创建图标按钮。

JButton 对象除了可以像 Button 对象一样拥有文件标签之外，还可以拥有一个图标。这个图标可以是用户自己绘制的图形，也可以是已经存在的.gif 图像。

（2）改变按钮图标。

JButton 按钮不但可以拥有一个图标，而且可以拥有一个以上的图标，并根据 Swing 按钮所处状态的不同而自动变换不同的 Swing 按钮图标。

（3）为按钮加入提示。

在实际使用的许多按钮中，具有这样一种功能：当鼠标在按钮上停留很短的几秒钟时，屏幕上将会出现一个简短的关于这个按钮的作用的提示信息。使用 Swing 按钮可以很方便

地实现这个加入提示的功能。

3 . JPasswordField

JPasswordField 类是 JPasswordField 类。用户在 JPasswordField 对象中输入的字符会被其他的字符替代而遮住，JPasswordField 组件主要用来输入口令。

4 . JTabbedPane

在前面曾经学习过 CardLayout 这种布局策略，使用 CardLayout 有一个不便之处，即用户不能了解被第一张卡片遮住的后面卡片的内容。使用 JTabbedPane 容器可以解决这个问题。

例 10-19 使用 JButton。在例中，使用了 ToolTipText , Border, Icon 等，并注意在容器 JFrame 及 JApplet 中加入组件要使用 getContentPane().add()方法。

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class JButtonDemo extends JApplet {
    JButton b1 = new JButton("JButton 1");
    JButton b2 = new JButton("JButton 2");
    JTextField t = new JTextField(20);
    public void init() {
        ActionListener al = new ActionListener() {
            public void actionPerformed(ActionEvent e){
                String name =
                    ((JButton)e.getSource()).getText();
                t.setText(name + " Pressed");
            }
        };
        b1.setToolTipText("Press Button will show msg");
        b1.setBorder( new BevelBorder(BevelBorder.RAISED) );
        b1.setIcon( new ImageIcon( "cupHJbutton.gif" ) );

        b1.addActionListener(al);
        b2.addActionListener(al);

        getContentPane().setLayout( new FlowLayout() );
        getContentPane().add(b1);
    }
}
```

```
        getContentPane().add(b2);
        getContentPane().add(t);
    }
    public static void main(String args[]) {
        JApplet applet = new JButtonDemo();
        JFrame frame = new JFrame("Test Swing");
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
        frame.getContentPane().add(
            applet, BorderLayout.CENTER);
        frame.setSize(400,300);
        applet.init();
        applet.start();
        frame.setVisible(true);
    }
}
```

运行结果如图 10-23 所示。



图 10-23 使用 JButton

10.8 基于 GUI 的应用程序

10.8.1 使用可视化设计工具

随着图形用户界面的普及和界面元素标准化程度的提高，许多辅助设计和实现图形用

户界面的方法和工具也相应出现,例如可视化编程方法允许设计人员直接绘出图形界面,然后交给专门的工具自动编码生成这个图形界面,免除了开发者的许多编程负担,目前许多应用软件开发工具都具有可视化编程的功能。

使用可视化设计工具,可以大大地加快界面设计。本书第 2 章中已经介绍了一些开发工具,读者可以进一步查阅那些工具的使用手册。

10.8.2 菜单的定义与使用

菜单是非常重要的 GUI 组件,每个菜单组件包括一个菜单条,称为 **MenuBar**。每个菜单条又包含若干个菜单项,称为 **Menu**。每个菜单项再包含若干个菜单子项,称为 **MenuItem**。每个菜单子项的作用与按钮相似,也是在用户点击时引发一个动作命令,所以整个菜单就是一组经层次化组织、管理的命令集合,使用它,用户可以方便地向程序发布指令。

Java 中的菜单分为两大类:一类是菜单条式菜单,通常称的菜单就是指这一类菜单;另一类是弹出式菜单。下面首先讨论菜单条式菜单的实现与使用。

1. 菜单的设计与实现

菜单的设计与实现步骤如下。

(1) 创建菜单条 **MenuBar**。例如,下面的语句创建了一个空的菜单条

```
m_MenuBar=new MenuBar();
```

(2) 创建不同的菜单项 **Menu** 加入到空菜单条中。下面的语句创建了一个菜单项并把它加入到菜单条中,菜单项的标题为“编辑”。

```
menuEdit=new Menu("编辑");
```

```
m_MenuBar.add(menuEdit);
```

(3) 为每个菜单项创建其所包含的更小的菜单子项 **MenuItem**,并把菜单子项加入到菜单项中去。

```
mi_Edit_Copy=new MenuItem("复制");
```

```
menuEdit.add(mi_Edit_Copy);
```

(4) 将整个建成的菜单条加入到某个容器中去。

```
This.setMenuBar(m_MenuBar);
```

这里的 **this** 代表程序的容器 **Frame**,需要注意的是并非每个容器都可以配菜单条式菜单,只有实现了 **MenuContainer** 接口的容器才能加入菜单。

(5) 上面的这些工作主要是为了组建菜单的结构,如有哪些菜单项,哪些菜单子项,等等。但是,如何定义各个菜单子项所对应的命令和操作呢?首先需要将各菜单子项注册给实现了动作事件的监听接口 **ActionListener** 的监听者。

(6) 为监听者定义 **actionPerformed(ActionEvent e)** 方法,在这个方法中调用 **e.getSource()** 或 **e.getActionCommand()** 来判断用户点击的菜单子项,并完成这个子项定义的操作。

2. 使用分隔线

有时希望在菜单子项之间增加一条横向分隔线,以便把菜单子项分成几组。加入分隔

线的方法是使用 `Menu` 的方法 `addSeparator()`，使用时要注意该语句的位置。菜单子项是按照加入的先后顺序排列在菜单项中的，希望把分隔线加在哪里，就要把分隔线语句放在哪里。

```
menuFile.addSeparator(); // 加一条横向分隔线
```

3. 使用菜单子项快捷键

除了用鼠标选择菜单子项，还可以为每个菜单子项定义一个键盘快捷键，这样用键盘一样可以选择菜单子项。快捷键是一个字母，定义好后按住 `Ctrl` 键和这个字母就可以选中对应的菜单子项。为菜单子项定义快捷键有两种方法：一种是在创建菜单子项的同时定义快捷键。

```
MenuItem mi_File_Open=new MenuItem("打开", new MenuShortcut('o'));
```

另一种是为已经存在的菜单子项定义快捷键。

```
mi_File_Exit setShortcut(new MenuShortcut('x')); // 单独设置菜单子项的快捷键设置后菜单子项“打开”对应快捷键 Ctrl+o，菜单子项“退出”对应 Ctrl+x。
```

4. 使用二级菜单

如果希望菜单子项还能够进一步再引出更多的菜单项，可以使用二级菜单。二级菜单的使用方法很简单，创建一个包含若干菜单子项(`MenuItem`)的菜单项(`Menu`)，把这个菜单项像菜单子项一样加入到一级菜单项中即可。

```
m_Edit_Paste=new Menu("粘贴"); // 创建二级菜单项
mi_past_All=new MenuItem("全部粘贴");
mi_past_Part=new MenuItem("部分粘贴");
m_Edit_Paste.add(mi_Paste_Part); // 为二级菜单项加入菜单子项
m_Edit_Paste.add(mi_Paste_All);
menuEdit.add(m_Edit_Paste); // 把二级菜单项加入菜单项
```

5. 使用检测盒菜单子项

Java 中还定义了一种特殊的菜单子项，称为检测盒菜单子项 `CheckboxMenuItem`。这种菜单子项与检测盒一样，有“选中”和“未选中”两种状态，每次选择这类菜单子项都使它在这两种状态之间切换，处于“选中”状态的检测盒菜单子项的前面有一个小对号，处于“未选中”状态时没有这个小对号。

创建检测盒菜单子项并把它加入菜单项的方法如下：

```
mi_Edit_Cut=new CheckboxMenuItem("剪切"); // 创建选择菜单子项
menuEdit.add(mi_Edit_Cut);
```

选择检测盒菜单子项引发的事件不是动作事件 `ActionEvent`，而是选择事件 `ItemEvent`，所以需要把检测盒菜单子项注册给 `ItemListener`，并具体实现 `ItemListener` 的 `itemStateChanged(ItemEvent e)` 事件，与响应检测盒的事件较为相似。

```
mi_Edit_Cut.addItemListener(this);
```

6. 使用弹出式菜单

弹出式菜单附着在某一个组件或容器上，一般它是不可见的，只有当用户用鼠标右键单击附着有弹出式菜单的组件时，这个菜单才“弹出”来显示。

弹出式菜单与菜单条式菜单一样，也包含若干个菜单子项，创建弹出式菜单并加入菜单子项的操作如下：

```
PopupMenu popM=new PopupMenu();           // 创建弹出窗口
MenuItem pi_New=new MenuItem("新建");       // 为弹出窗口创建菜单子项
Pi_New. addActionListener(this);           // 使菜单子项响应动作事件
popM. add(pi_New);                          // 为弹出菜单加入菜单子项
```

然后需要把弹出式菜单附着在某个组件或容器上。

```
ta.add(popM); //将弹出窗口加在文本域上
```

用户单击鼠标右键时弹出式菜单不会自动显示出来，还需要一定的程序处理，首先把附着有弹出菜单的组件或容器注册给 `MouseListener`。

`ta.addMouseListener(new HandleMouse(this));` //文本域响应鼠标事件，弹出菜单然后重载 `MouseListener` 的 `mouseReleased(MouseEvent e)` 方法。在这个方法里调用弹出式菜单的方法 `show()` 把它自身显示在用户鼠标点击的位置。

```
public void mouseReleased(MouseEvent e)      // 鼠标按键松开事件弹出菜单
{
    if(e. isPopupTrigger())                  // 检查鼠标事件是否由弹出菜单引发
        m_Parent. popM. show((Component)e. getSource(), e.getX(), e.getY());
}
```

这里方法 `e.getSource()` 返回的是附着有弹出式菜单的组件或容器，弹出式菜单应该显示在这个组件或容器中鼠标点击的位置(由 `e.getX()` 方法和 `e.getY()` 方法确定鼠标点击的坐标位置)。

例 10-20 TestMenuItem.java 使用菜单。

```
import java.awt.*;
import java.awt.event.*;

public class TestMenuItem{
    public static void main(String[] args) {
        Frame f = new Frame("Menu");
        MenuBar mb = new MenuBar();
        f.setMenuBar(mb);

        Menu m1 = new Menu("File");
        Menu m2 = new Menu("Edit");
        Menu m3 = new Menu("Help");
```

```
mb.add(m1);
mb.add(m2);
mb.setHelpMenu(m3);

MenuItem m11 = new MenuItem("New");
MenuItem m12 = new MenuItem("Save");
MenuItem m13 = new MenuItem("Load");
MenuItem m14 = new MenuItem("Quit");

m1.add(m11);
m1.add(m12);
m1.add(m13);
m1.addSeparator();
m1.add(m14);

m14.addActionListener( new ActionListener(){
    public void actionPerformed((ActionEvent e){
        System.exit(0);
    }
});

f.setSize(250,200);
f.setVisible(true);
}
}
```

运行结果如图 10-24 所示。

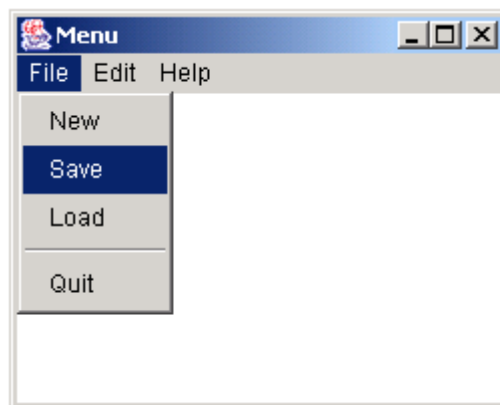


图 10-24 使用菜单

10.8.3 菜单、工具条及对话框的应用

下面以一个简单的文本编辑器来具体地介绍菜单、工具条、对话框，如图 10-25 所示。为了读者便于阅读，在程序的重要地方都加了简单的注释。

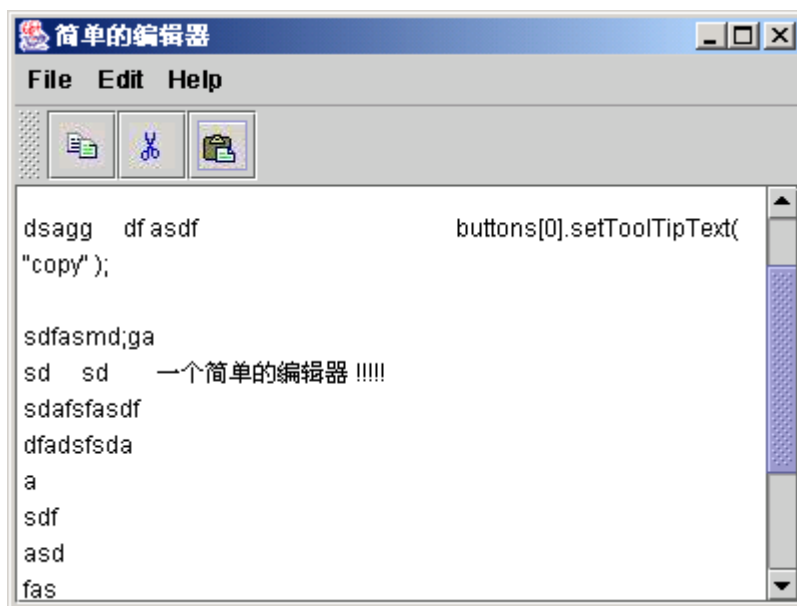


图 10-25 一个简单的文本编辑器

例 10-21 TestTextEditor.java 一个简单的文本编辑器。

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;

class TextEditorFrame extends JFrame
{

    File file = null;
    Color color = Color.black;

    TextEditorFrame(){
        initTextPane();
        initMenu();
        initAboutDialog();
    }
}
```

```
        initToolBar();
    }

    void initTextPane(){                                // 将文本框放入有滚动对象, 并
                                                         加入到 Frame 中
        getContentPane().add( new JScrollPane(text) );
    }

    JTextPane text = new JTextPane(); //文本框
    JFileChooser filechooser = new JFileChooser();
                                                         // 文件选择对话框
    JColorChooser colorchooser = new JColorChooser();
                                                         // 颜色选择对话框
    JDialog about = new JDialog(this); // 关于对话框
    JMenuBar menubar = new JMenuBar(); // 菜单

    JMenu [] menus = new JMenu[] {
        new JMenu("File"),
        new JMenu("Edit"),
        new JMenu("Help")
    };

    JMenuItem menuitems [][] = new JMenuItem[][]{{
        new JMenuItem("New"),
        new JMenuItem("Open..."),
        new JMenuItem("Save..."),
        new JMenuItem("Exit")},{
        new JMenuItem("Copy"),
        new JMenuItem("Cut"),
        new JMenuItem("Paste"),
        new JMenuItem("Color...")},{
        new JMenuItem("About")}
    };

    void initMenu(){                                    // 初始化菜单
        for( int i=0; i<menus.length; i++ ){
            menubar.add( menus[i] );
            for( int j=0; j<menuitems[i].length; j++ ){
                menus[i].add( menuitems[i][j] );
                menuitems[i][j].addActionListener( action );
            }
        }
    }
}
```

```
    }  
    }  
    this.setJMenuBar( menubar );  
}  
  
ActionListener action = new ActionListener(){ // 菜单事件处理  
    public void actionPerformed((ActionEvent e) {  
        JMenuItem mi = (JMenuItem)e.getSource();  
        String id = mi.getText();  
        if( id.equals("New" ) ){  
            text.setText("");  
            file = null;  
        }else if( id.equals("Open...") ){  
            if( file != null ) filechooser.  
                setSelectedFile( file );  
            int returnVal = filechooser.showOpenDialog(  
                TextEditorFrame.this);  
            if(returnVal == JFileChooser.APPROVE_OPTION) {  
                file = filechooser.getSelectedFile();  
                openFile();  
            }  
        }else if( id.equals("Save...") ){  
            if( file != null ) filechooser.  
                setSelectedFile( file );  
            int returnVal = filechooser.showSaveDialog(  
                TextEditorFrame.this);  
            if(returnVal == JFileChooser.APPROVE_OPTION) {  
                file = filechooser.getSelectedFile();  
                saveFile();  
            }  
        }else if( id.equals("Exit") ){  
            System.exit(0);  
        }else if( id.equals("Cut") ){  
            text.cut();  
        }else if( id.equals("Copy") ){  
            text.copy();  
        }else if( id.equals("Paste") ){  
            text.paste();  
        }else if( id.equals("Color...") ){
```

```
        color = JColorChooser.showDialog(
            TextEditorFrame.this, "", color );
        text.setForeground(color);
    }else if( id.equals("About")){
        about.setSize(100,50);
        about.show();
    }
}
};

void saveFile(){                                // 保存文件, 将字符写入文件
    try{
        FileWriter fw = new FileWriter( file );
        fw.write( text.getText() );
        fw.close();
    }catch(Exception e ){ e.printStackTrace(); }
}

void openFile(){                                // 读入文件, 并将字符置入文本框中
    try{
        FileReader fr = new FileReader( file );
        int len = (int) file.length();
        char [] buffer = new char[len];
        fr.read( buffer, 0, len );
        fr.close();
        text.setText( new String( buffer ) );
    }catch(Exception e ){ e.printStackTrace(); }
}

void initAboutDialog(){                          // 初始化对话框
    about.getContentPane().add( new JLabel("简单编辑器 V1.0") );
    about.setModal( true );
    about.setSize(100,50 );
}

JToolBar toolbar = new JToolBar(); // 工具条
JButton [] buttons = new JButton[] {
    new JButton( "", new ImageIcon("copy.jpg") ),
    new JButton( "", new ImageIcon("cut.jpg") ),
    new JButton( "", new ImageIcon("paste.jpg") )
```



```
};

void initToolBar(){ // 加入工具条
    for( int i=0; i<buttons.length; i++)
        toolbar.add( buttons[i] );
    buttons[0].setToolTipText( "copy" );
    buttons[0].addActionListener( new ActionListener(){
        public void actionPerformed((ActionEvent e ){
            text.copy();
        }
    });
    buttons[1].setToolTipText( "cut" );
    buttons[1].addActionListener( new ActionListener(){
        public void actionPerformed((ActionEvent e ){
            text.cut();
        }
    });
    buttons[2].setToolTipText( "paste" );
    buttons[2].addActionListener( new ActionListener(){
        public void actionPerformed((ActionEvent e ){
            text.paste();
        }
    });
    this.getContentPane().add( toolbar, BorderLayout.NORTH );
}

}

public class TextEditorApp // 应用程序
{
    public static void main( String [] args){
        TextEditorFrame f = new TextEditorFrame();
        f.setTitle( "简单的编辑器");
        f.setSize( 400, 300 );
        f.show();
    }
}
```

习题

1. 试列举出图形用户界面中你使用过的组件。
2. Java 中常用的布局管理各有什么特点?
3. 简述 Java 的事件处理机制。
4. 什么是事件源? 什么是监听者?
5. 列举 java.awt.event 包中定义的事件类, 并写出它们的继承关系。
6. 列举 GUI 的各种标准组件和它们之间的层次继承关系。
7. Component 类有何特殊之处? 其中定义了哪些常用方法?
8. 将各种常用组件的创建语句、常用方法、可能引发的事件、需要注册的监听者和监听者需要重载的方法综合在一张表格中画出。
9. 编写 Applet 包括一个标签、一个文本框和一个按钮, 当用户单击按钮时, 程序把文本框中的内容复制到标签中。
10. 编写 Applet 程序, 画出一条螺旋线。
11. 编写 Applet 程序, 用 paint() 方法显示一行字符串, Applet 包含两个按钮“放大”和“缩小”, 当用户单击“放大”时显示的字符串字体放大一号, 单击“缩小”时显示的字符串字体缩小一号。
12. 编写 Applet 程序, 包含三个标签, 其背景分别为红、黄、蓝三色。
13. 使用 Checkbox 标志按钮的背景色, 使用 CheckboxGroup 标志三种字体风格, 使用 Choice 选择字号, 使用 List 选择字体名称, 由用户确定按钮的背景色和前景字符的显示效果。
14. 编写一个 Applet 包含一个滚动条, 在 Applet 中绘制一个圆, 用滚动条滑块显示的数字表示该圆的直径, 当用户拖动滑块时, 圆的大小随之改变。
15. 编写一个 Applet 响应鼠标事件, 用户可以通过拖动鼠标在 Applet 中画出矩形, 并在状态条显示鼠标当前的位置。使用一个 Vector 对象保存用户所画过的每个矩形并显示、响应键盘事件, 当用户击 q 键时清除屏幕上所有的矩形。
16. 编写 Applet 程序实现一个计算器, 包括十个数字(0~9)按钮和四个运算符(加、减、乘、除)按钮, 以及等号和清空两个辅助按钮, 还有一个显示输入输出的文本框。试分别用 BorderLayout 和 GridLayout 实现。
17. Panel 与 Applet 有何关系? Panel 在 Java 程序里通常起到什么作用?
18. 为什么说 Frame 是非常重要的容器? 为什么使用 Frame 的程序通常要实现 WindowListener? 关闭 Frame 有哪些方法?
19. 练习使用列表框及组合框。
20. Swing 组件与 AWT 件有何区别。
21. 绘出以下函数的曲线:
$$y = 5\sin(x) + \cos(3x)$$
$$y = \sin(x) + \sin(6x)/10$$
22. 绘出以下函数的曲线:

$$r=\cos(2\ \theta)$$

$$r=\cos(3\ \theta)$$

23. 根据本章的所学习的内容用 `JavaApplication` 编写一个模拟的文本编辑器。给文本编辑器增设字体字号的功能。

第 11 章 网络、多媒体和数据库编程

Java 语言在网络、多媒体、数据库等方面的应用十分广泛，本章中介绍 Java 在这些方面的编程方法，一方面可以帮助读者进一步准确地掌握 Java 语言，同时也为在实践中更好地应用 Java 来解决实际问题提供一个导引。本章最后还简要介绍了 Java 语言在 J2EE 及 J2ME 中的应用。

11.1 Java 网络编程

11.1.1 使用 URL

Java 的网络编程经常要使用 `java.net` 包中的一些类，其中 `java.net.URL` 类是对网络资源地址的表示。URL 即 Uniform Resource Location(统一资源地址)，基本格式是：

协议名：主机名/目录及文件名

如 `http://www.pku.edu.cn/index.html`

`java.net.URL` 类是一个使用十分方便的类，它对网络资源的访问进行了封装，下面举一个例子，通过 URL 来直接获取网上的文件内容。

读取网络上文件内容的步骤如下。

(1) 创建一个 URL 类型的对象。

```
URL url = "ftp://ftp.microsoft.com/pub/readme.txt ";
```

(2) 利用 URL 类的 `openStream()`，获得对应的 `InputStream` 类的对象。

```
InputStream filecon = url.openStream();
```

(3) 通过 `InputStream` 来读取内容。

例 11-1 是读取网上文件内容的例子，为简洁起见，该例中只将文件的内容逐行读出，并在文本区显示出来。

例 11-1 URLGetFile.java 通过 URL 读取网络上文件内容。

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class URLGetFile extends Applet{
    URL url;

    TextArea showarea = new TextArea("下载的数据: ");

    public void init(){
        String strurl = "http://www.pku.edu.cn";
```

```
        try{
            url = new URL(strurl);
        }catch ( MalformedURLException e){
            System.out.println("URL 格式有错" );
        }
        add(showarea);
    }

    public void start(){
        InputStream filecon = null;
        BufferedReader filedata = null;
        String line;
        try{
            filecon = url.openStream();
            filedata = new BufferedReader(new
                InputStreamReader(filecon));
            while ((line = filedata.readLine()) != null){
                showarea.append(line+"\n");
            }
        }catch (IOException e){
            System.out.println("Error in I/O:" + e.getMessage());
        }
    }

    public static void main( String[] args){
        Frame f = new Frame("URL Test");
        Applet ap = new URLGetFile();
        ap.init();
        f.add(ap);
        f.addWindowListener( new WindowAdapter(){
            public void windowClosing(WindowEvent e){System.exit(0);}
        });
        f.setSize( 400,300 );
        f.show();
        ap.start();
    }
}
```

运行结果如图 11-1 所示。

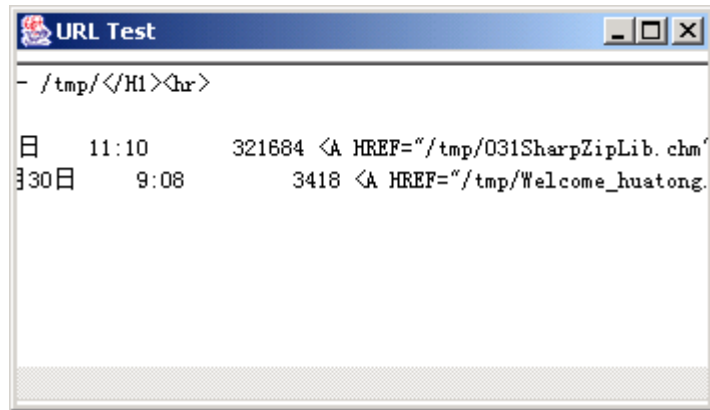


图 11-1 通过 URL 读取网络上文件内容

11.1.2 用 Java 实现底层网络通信

用 Java 实现计算机网络的底层通信，就是用 Java 程序实现网络通信协议所规定的功能的操作。这是 Java 网络编程技术的一部分。网络通信协议的种类有很多，这里只讨论其中基于套接字的 Java 编程。

套接字(Socket)是基于 TCP/IP 协议的编程接口，通信双方通过 Socket 来进行通信。在 Java 中，相关的类有 InetAddrss, Socket, ServerSocket 类等。

1 . InetAddress 类

InetAddress 类主要用来区分计算机网络中不同节点，即不同的计算机并对其寻址。每个 InetAddress 对象中包括了 IP 地址、主机名等信息。使用 InetAddress 类可以用在程序中使用主机名代替 IP 地址，从而使程序更加灵活，可读性更好。

2 . 流式 Socket 的通信机制

流式 Socket 所完成的通信是基于连接的通信，即在通信开始之前先由通信双方确认身份并建立一条专用的虚拟连接通道，然后它们通过这条通道传送数据信息进行通信，当通信结束时再将原先所建立的连接拆除。Server 端首先建立一个 ServerSocket 对象，调用 listen() 方法在某端口提供一个监听 Client 请求的监听服务，当 Client 端向该 Server 发出连接请求时，ServerSocket 调用 accept() 方法接受这个请求，并建立一个 Socket 与客户端的 Socket 进行通信。通信的基本方式是通过 Socket 得到流对象，在流对象上进行输入和输出。

3 . Socket 类

客户端要与服务端相连，则客户端需要建立 Socket 对象。Socket 的建立方法如下：

```
Socket s = new Socket("机器名或 Ip 地址", 端口号);
```

其中，端口号要与服务器上提供服务的端口号一致。

Socket 有两种重要的方法：getInputStream() 及 getOutputStream()，利用它可以得到相关的输入流及输出流，向流中读数据及写数据，就达到了与服务器通信的目的。

Socket 通信完毕，要使用 `close()` 方法进行关闭。

4 . ServerSocket 类

服务器程序不同于客户机，它需要初始化一个端口进行监听，遇到连接呼叫，才与相应的客户机建立连接。`java.net` 包的 `ServerSocket` 类包含了编写服务方所需要的功能。下面给出 `ServerSocket` 类的部分方法：

- ✎ `public ServerSocket(int port);` // 构造方法
- ✎ `public Socket accept() throws IOException;` // 接收客户的请求
- ✎ `public InetAddress getInetAddress();` // 得到地址
- ✎ `public int getLocalPort();` // 得到本地的端口
- ✎ `public void close() throws IOException;` // 关闭服务
- ✎ `public synchronized void setSoTimeout (int timeout) throws SocketException;`
- ✎ `public synchronized int getSoTimeout() throws IOException;` // 超时的时间

`ServerSocket` 构造器是服务器程序运行的基础，它将参数 `port` 指定的端口初始化作为该服务器的端口，监听客户机连接请求。`Port` 的范围是 `0~65 536`，但 `0~1 023` 是标准 Internet 协议保留端口，一般自定义的端口号在 `8 000~16 000` 之间。

仅初始化了 `ServerSocket` 还是远远不够的，它没有同客户机交互的套接字，因此需要调用该类的 `accept` 方法接受客户呼叫。`accept()` 方法直到有连接请求才返回通信套接字的实例。通过这个实例的输入、输出流，服务器可以接收用户指令，并将相应结果回应客户机。

5 . 简单的服务方程序

例 11-2 TestServer.java 简单的服务方程序。

```
import java.net.*;
import java.io.*;

public class TestServer {
    public static void main(String args[]) {
        ServerSocket s = null;
        try {
            s = new ServerSocket(8888);
        } catch (IOException e) {}
        while (true) {
            try {
                Socket s1 = s.accept();
                OutputStream os = s1.getOutputStream();
                DataOutputStream dos = new DataOutputStream(os);
                dos.writeUTF("Hello,bye-bye!");
                dos.close();
            }
        }
    }
}
```

```
        s1.close();
    } catch (IOException e) {}
}
}
```

本服务程序，在 8888 号端口上进行监听，一旦有服务相联，则接受请求，并向客户发送一串信息。

6. 简单的客户方程序

例 11-3 TestClient.java 简单的客户方程序。

```
import java.net.*;
import java.io.*;

public class TestClient {
    public static void main(String args[]) {
        try {
            Socket s1 = new Socket("127.0.0.1", 8888);
            InputStream is = s1.getInputStream();
            DataInputStream dis = new DataInputStream(is);
            System.out.println(dis.readUTF());
            dis.close();
            s1.close();
        } catch (ConnectException connExc) {
            System.err.println("服务器连接失败！");
        } catch (IOException e) {
        }
    }
}
```

本客户程序与本地机器 (127.0.0.1) 的 8888 号端口相连，从服务器上读取信息并显示。

11.1.3 实现多线程服务器程序

上面的例子中，针对一个客户进行服务后，才能进行对其他客户进行服务，为了同时对多个客户进行服务，需要利用多线程，每个线程针对一个客户进行服务。

在下面的例子中，客户方与多个服务方进行交谈 (chat)。对于每个客户服务，都使用一个线程(这里类名为 Connection)，线程的任务是接收客户方的字符并显示出来。

1. 服务端程序

例 11-4 ChatServer.java 服务端程序。

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.net.*;

public class ChatServer extends JFrame implements Runnable{
    JPanel contentPane;
    JTextField txtInput = new JTextField();
    JButton btnSend = new JButton();
    List lstMsg = new List();

    /**Construct the frame*/
    public ChatServer() {
        enableEvents(AWTEvent.WINDOW_EVENT_MASK);
        try {
            jbInit();
        }
        catch(Exception e) {
            e.printStackTrace();
        }
        ServerListen();
    }

    /**Component initialization*/
    private void jbInit() throws Exception {
        txtInput.setText("please input here");
        txtInput.setBounds(new Rectangle(42, 234, 196, 34));

        //setIconImage(Toolkit.getDefaultToolkit().createImage(ChatSe
        rver.class.getResource("[Your Icon]")));
        contentPane = (JPanel) this.getContentPane();
        contentPane.setLayout(null);
        this.setSize(new Dimension(400, 300));
        this.setTitle("Chat Server");
        btnSend.setText("Send");
    }
}
```

```
        btnSend.setBounds(new Rectangle(268, 235, 98, 35));
        btnSend.addActionListener(new java.awt.event.ActionListener(){
            public void actionPerformed(ActionEvent e) {
                btnSend_actionPerformed(e);
            }
        });
        lstMsg.setBounds(new Rectangle(42, 32, 319, 192));
        contentPane.add(txtInput, null);
        contentPane.add(btnSend, null);
        contentPane.add(lstMsg, null);
    }

    /**Overridden so we can exit when window is closed*/
    protected void processWindowEvent(WindowEvent e) {
        super.processWindowEvent(e);
        if (e.getID() == WindowEvent.WINDOW_CLOSING) {
            System.exit(0);
        }
    }

    public static void main(String[] args){
        ChatServer server = new ChatServer();
        server.show();
    }

    public void processMsg( String str ){
        this.lstMsg.add(str);
    }

    void btnSend_actionPerformed(ActionEvent e) {
        for( int i=0; i<clients.size(); i++){
            Connection c = (Connection) clients.get(i);
            try{
                c.sendMsg( this.txtInput.getText() );
            }catch(Exception ee){}
        }
    }

    public final static int DEFAULT_PORT = 6543;
    protected ServerSocket listen_socket;
```

```
Thread thread;
java.util.Vector clients;

// Create a ServerSocket to listen for connections on; start
the thread. public void ServerListen() {
    try {
        listen_socket = new ServerSocket(DEFAULT_PORT);
    } catch (IOException e) {
        e.printStackTrace();
    }
    processMsg("Server: listening on port " + DEFAULT_PORT);
    clients = new java.util.Vector();
    thread = new Thread(this);
    thread.start();
}

// The body of the server thread. Loop forever, listening for and
// accepting connections from clients. For each connection,
// create a Connection object to handle communication through the
// new Socket.
public void run() {
    try {
        while(true) {
            Socket client_socket = listen_socket.accept();
            Connection c = new Connection(client_socket, this);
            clients.add( c );
            processMsg( "One Client Comes in");
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

// This class is the thread that handles all communication with a client
class Connection extends Thread {
    protected Socket client;
    protected BufferedReader in;
    protected PrintWriter out;
```

```
ChatServer server;

// Initialize the streams and start the thread
public Connection(Socket client_socket, ChatServer server_frame) {
    client = client_socket;
    server = server_frame;
    try {
        in = new BufferedReader(new InputStreamReader
            (client.getInputStream()));
        out = new java.io.PrintWriter (client.getOutputStream());
    }
    catch (IOException e) {
        try { client.close(); } catch (IOException e2) { ; }
        e.printStackTrace();
        return;
    }
    this.start();
}

// Provide the service.
// Read a line
public void run() {
    String line;
    StringBuffer revline;
    int len;
    try {
        for(;;) {
            // read in a line
            line = receiveMsg();
            server.processMsg( line );
            if (line == null) break;
        }
    }
    catch (IOException e) { ; }
    finally { try {client.close();} catch (IOException e2) {;} }
}

public void sendMsg(String msg) throws IOException{
    out.println( msg );
}
```

```
        out.flush();
    }

    public String receiveMsg() throws IOException{
        String msg = new String();
        try {

            msg = in.readLine();
        } catch(IOException e) {
            e.printStackTrace();
        }
        return msg;
    }
}
```

2. 客户端程序

例 11-5 ChatClient.java 客户端程序。

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import javax.swing.*;
import java.net.*;
import java.io.*;

public class ChatClient extends JFrame implements Runnable{
    boolean isStandalone = false;
    public ChatClient() {
        enableEvents(AWTEvent.WINDOW_EVENT_MASK);
        try {
            jbInit();
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
    public void init() {
        try {
            jbInit();
        }
    }
}
```

```
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}

/**Main method*/
public static void main(String[] args) {
    ChatClient c = new ChatClient();
    c.show();
}

JPanel contentPane;
JTextField txtInput = new JTextField();
JButton btnSend = new JButton();
JButton btnStart = new JButton();
List lstMsg = new List();

Socket sock;
Thread thread;
BufferedReader in;
PrintWriter out;
public final static int DEFAULT_PORT = 6543;
boolean bConnected;

public void startConnect() {
    bConnected = false;
    try {
        //sock = new Socket(this.getCodeBase().getHost(),
        DEFAULT_PORT);
        sock = new Socket( "127.0.0.1",DEFAULT_PORT);
        bConnected = true;
        processMsg("Connection ok");
        in = new BufferedReader(new InputStreamReader(sock.
        getInputStream()));
        out = new java.io.PrintWriter(sock.getOutputStream());
    } catch(IOException e) {
        e.printStackTrace();
        processMsg("Connection failed");
    }
}
```

```
    }
    if(thread == null) {
        thread = new Thread(this);
        thread.start();
    }
}

public void run(){
    while(true){
        try{
            String msg = receiveMsg();
            Thread.sleep(100L); //????
            if( msg != null ){
                processMsg( msg );
            }
        } catch( IOException e ){
            e.printStackTrace();
        } catch( InterruptedException ei){}
    }
}

public void sendMsg(String msg) throws IOException{
    out.println( msg );
    out.flush();
}

public String receiveMsg() throws IOException{
    String msg = new String();
    try {
        msg = in.readLine();
    } catch(IOException e) {
        e.printStackTrace();
    }
    return msg;
}

public void processMsg( String str ){
    this.lstMsg.add(str);
}
```

```
private void jbInit() throws Exception {
    txtInput.setText("please input here");
    txtInput.setBounds(new Rectangle(42, 234, 196, 34));
    contentPane = (JPanel) this.getContentPane();
    contentPane.setLayout(null);
    this.setSize(new Dimension(500, 300));
    this.setTitle("Chat Client");
    btnSend.setText("Send");
    btnSend.setBounds(new Rectangle(258, 235, 70, 35));
    btnSend.addActionListener(new java.awt.event.ActionListener()
    {
        public void actionPerformed(ActionEvent e) {
            btnSend_actionPerformed(e);
        }
    });
    btnStart.setText("Connect To Server");
    btnStart.setBounds(new Rectangle(338, 235, 150, 35));
    btnStart.addActionListener(new java.awt.event.ActionListener()
    {
        public void actionPerformed(ActionEvent e) {
            btnStart_actionPerformed(e);
        }
    });
    lstMsg.setBounds(new Rectangle(42, 32, 319, 192));
    contentPane.add(txtInput, null);
    contentPane.add(btnSend, null);
    contentPane.add(btnStart, null);
    contentPane.add(lstMsg, null);
}

void btnSend_actionPerformed(ActionEvent e) {
    if( txtInput.getText().length() != 0 ){
        try{
            sendMsg( txtInput.getText() );
        }catch(IOException e2){ processMsg(e2.toString());}
    }
}
```



```

void btnStart_actionPerformed(ActionEvent e) {
    this.startConnect() ;
}

//static initializer for setting look & feel
static {
    try {
        //UIManager.setLookAndFeel(UIManager. GetSystemLookAndFeel-
        ClassName());
        //UIManager.setLookAndFeel(UIManager.getCrossPlatformLookAnd-
        FeelClassName());
    }
    catch(Exception e) {
    }
}
}

```

程序运行结果如图 11-2 所示。

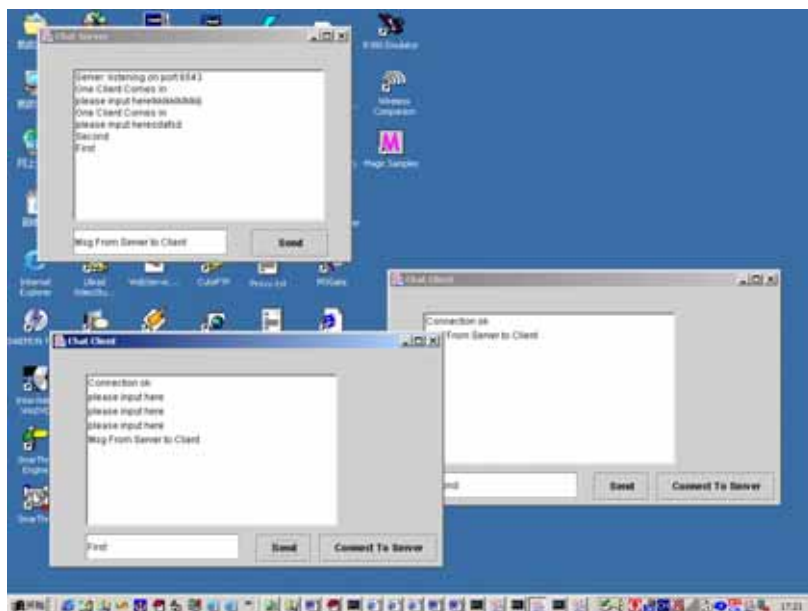


图 11-2 多线程的 chat 程序

11.1.4 Java 的 Email 编程

1. 使用 Socket 编程来获取 Email

使用 Socket 编程来获取 Email, 就是使用 Socket 向 Email 服务器的 POP 端口 (25 号端

口) 通信, 向服务器发送相关信息, 从服务器得到相关信息。这些信息的格式要遵循邮件的协议 (POP 协议)。

例 11-6 MailGet.java 使用 Socket 编程来获取 Email。

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.net.*;
import java.io.*;

public class MailGet extends Applet
{
    public void init()
    {
        //{{INIT_CONTROLS
        setLayout(null);
        setSize(540,393);
        label1 = new java.awt.Label("Server");
        label1.setBounds(60,48,48,12);
        add(label1);
        label2 = new java.awt.Label("User");
        label2.setBounds(60,72,48,12);
        add(label2);
        label3 = new java.awt.Label("Passwd");
        label3.setBounds(48,96,48,12);
        add(label3);
        txtServer = new java.awt.TextField();
        txtServer.setBounds(108,48,324,23);
        add(txtServer);
        txtUser = new java.awt.TextField();
        txtUser.setBounds(108,72,324,22);
        add(txtUser);
        txtPass = new java.awt.TextField();
        txtPass.setEchoChar('*');
        txtPass.setBounds(108,96,324,24);
        add(txtPass);
        cmdGet = new java.awt.Button();
        cmdGet.setActionCommand("button");
        cmdGet.setLabel("Get");
```

```
cmdGet.setBounds(444,48,68,60);
cmdGet.setBackground(new Color(12632256));
add(cmdGet);
txtReply = new java.awt.TextArea();
txtReply.setBounds(60,144,415,213);
add(txtReply);
//}}

Action lAction = new Action();
cmdGet.addActionListener(lAction);
}

//{{DECLARE_CONTROLS
java.awt.Label label1;
java.awt.Label label2;
java.awt.Label label3;
java.awt.TextField txtServer;
java.awt.TextField txtUser;
java.awt.TextField txtPass;
java.awt.Button cmdGet;
java.awt.TextArea txtReply;
//}}

class Action implements java.awt.event.ActionListener
{
    public void actionPerformed(java.awt.event.ActionEvent event){

        Object object = event.getSource();
        if (object == cmdGet) {
            GetMail(event);
        }
    }
}

public void GetMail(java.awt.event.ActionEvent evt){
    String sHostName;
    int nPort = 110;
    String sReply;
    sHostName = txtServer.getText();
```

```
try {
    Socket sktConn = new Socket(sHostName,nPort);
    PrintStream ps = new PrintStream(sktConn.getOutputStream());
    sReply = getReply(sktConn);
    if (sReply.indexOf("+ERR") == -1 ){
        txtReply.append(sReply+"\n");
        ps.println("USER liulili");           // 用户名
        txtReply.append(getReply(sktConn)+"\n");
        ps.println("PASS " + txtPass.getText()); // 口令
        txtReply.append(getReply(sktConn)+"\n"); // 得到邮件内容
    }
    ps.println("QUIT ");                     // 退出
    txtReply.append(getReply(sktConn)+"\n");

} catch (IOException e) {
    System.out.println(e.getMessage());
}

}

String getReply(Socket sktConn){
    try {
        BufferedReader outgoing = new BufferedReader(
            new InputStreamReader(sktConn.getInputStream()));
        return outgoing.readLine();
    } catch (IOException e) {
        return e.getMessage();
    }
}

public static void main(String[] args){
    Frame f = new Frame("Draw");
    Applet p = new MailGet();
    p.init();
    p.start();
    f.add(p);
    f.setSize(400, 300);
    f.addWindowListener( new WindowAdapter(){
        public void windowClosing(WindowEvent e){ System.exit(0);}
    });
}
```

```
f.show();  
}  
  
}
```

运行结果如图 11-3 所示。

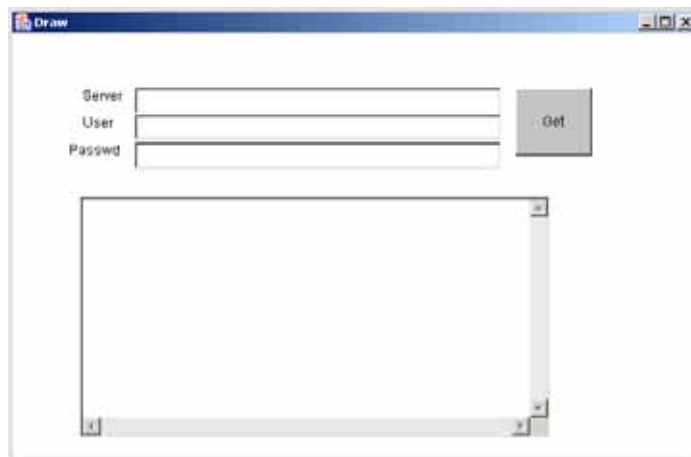


图 11-3 使用 Socket 编程来获取 Email

2. 使用 javax.mail 包来发送邮件

在新版本的 JDK 中，提供了对 mail 的支持，相关的类位于 javax.mail 包中。下面举一个例中，使用了 Session 类（表示会话）、MimeMessage 类（表示邮件消息）、Transport 类（表示邮件传送）。

例 11-7 MailSend.java 使用 javax.mail 包来发送邮件。

```
/*  
 * @(#)MailSend.java  
 */  
  
import java.util.*;  
import java.io.*;  
import javax.mail.*;  
import javax.mail.internet.*;  
import javax.activation.*;  
  
public class MailSend extends Object {  
  
    public static void main(String args[]){
```

```
String smtpServer = null;
String toEmail = null;
String fromEmail = null;
String body = null;

// 分析命令行参数
for ( int x=0; x < args.length-1; x++ ){
    if ( args[x].equalsIgnoreCase("-S") )
        smtpServer = args[x+1];
    else if ( args[x].equalsIgnoreCase("-T") )
        toEmail = args[x+1];
    if ( args[x].equalsIgnoreCase("-F") )
        fromEmail = args[x+1];
    if ( args[x].equalsIgnoreCase("-B") )
        body = args[x+1];
}

if ( smtpServer == null || toEmail == null || fromEmail ==
null || body == null ){
    System.out.println( "Usage: javamail_send -S <server> -T
<toemail> -F <from>
-B <body>" );
    System.exit(1);
}

// 得到一个 session
try{
    // 设置默认的参数
    Properties props = new Properties();
    props.put("mail.transport.protocol", "smtp" );
    props.put("mail.smtp.host", smtpServer );
    props.put("mail.smtp.port", "25" );

    // 创建 session 和一个新的邮件消息
    Session mailSession = Session.getInstance( props );
    Message msg = new MimeMessage( mailSession );

    // 设置 the FROM, TO, DATE and SUBJECT fields
    msg.setFrom( new InternetAddress( fromEmail ) );
```

```
msg.setRecipients( Message.RecipientType.TO,
    InternetAddress.parse(toEmail) );
msg.setSentDate( new Date() );
msg.setSubject( "Test Mail" );

// 创建邮件的体
msg.setText( body );

Transport.send( msg );

System.out.println( "The email below was sent
successfully" );
msg.writeTo( System.out );

    }catch(Exception E){
        System.out.println( E );
    }
}
}
```

11.2 多媒体编程

11.2.1 在 Applet 中获取声音与图像

1. 从网络上获取图像

Java Applet 可以直接从网络上节点获取图像并显示出来，来看一个简单的图像显示的例子。

例 11-8 imag0.java 在 Applet 中显示网络上的图片。

```
import java.applet.*;
import java.awt.*;
public class imag0 extends Applet{
    Image image;
    public void init() {
        image=getImage(getDocumentBase(),"test.gif");
    }
    public void paint(Graphics g) {
```

```
        g.drawImage(image, 0, 0, this);  
    }  
}
```

这是一个最简单的获取并显示图像的例子，在该例中，先用

`getImage(getDocumentBase(), 图像文件名)`

从 HTML 文档所在位置调用图像 `test.gif`，并由此生成一个 `Image` 类型的对象 `image`，然后用 `drawImage(image, 0, 0, this)` 在屏幕上将图像显示出来。

2. 从网络上获取声音

数字音频格式有很多种，其质量与采样频率和采样精度两个参数有关。频率的表示单位为赫兹(Hz)，它表示每秒采样次数。采样频率越高，音质就越好。采样精度为每次采样所存储的数据数量，它决定每个数字信号所能够表示的离散振幅的数量。存储每个样本的数据越多，音质就越好。但是高品质的声音需要占用大量的内存和磁盘空间。考虑到网络带宽，在 Internet 连接上传输就需要花费很长的时间。对于 Applet 来说，声音文件不宜太大。

Java 能够支持四种声音格式：AU，AIFF，WAVE，MIDI。

第一种声音格式 AU 为以前的 Java 1.1 版本支持的惟一的聲音格式。采样频率为 8 000 Hz，采样精度为 8 位。AIFF 和 WAVE 与 AU 格式一样，都用来表示数字化的声音。其中，WAVE 格式提供了更宽范围的音质。MIDI 格式专用于音乐，并且以音符与乐器而不是数字化的声音来描述声音的。

Java 中，简单的方法是用 Applet 类的 AudioClip 接口来装载声音。该接口封装了有关声音片断的常用方法，具有对播放声音片断的最小支持。

AudioClip 接口的方法有三个：

- ☞ `void play();` // 开始播放声音
- ☞ `void loop();` // 循环播放声音
- ☞ `void stop();` // 停止播放声音

Applet 类的 `getAudioClip()` 及 `getCodeBase()` 方法来获取声音片断及 URL 地址。可以利用此方法在 Web 页中播放指定的声音片断。

Java 从网络上获取声音文件并播放声音的编程有两种方法，一是利用 Applet 的 `play(URL)` 及 `play(URL, String)` 直接播放网络上的声音文件，另一类是通过 `getAudioClip(URL)` 或 `getAudioClip (URL, String)` 先从网络上获取声音文件，并生成 AudioClip 类型的对象，然后对该对象进行操作。如图 11-4 所示。

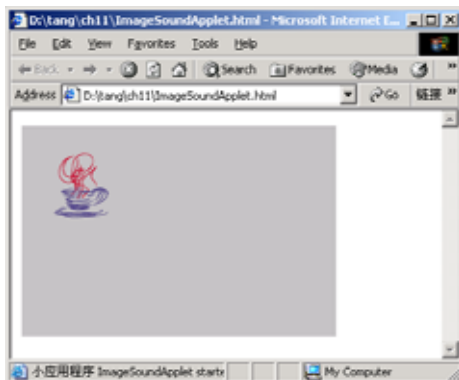


图 11-4 在 Applet 中获取声音与图像

例 11-9 ImageSoundApplet.java 在 Applet 中获取声音与图像。

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.net.*;

public class ImageSoundApplet extends Applet{
    Image img;
    AudioClip snd;
    public void init(){
        URL url = getDocumentBase();
        img = getImage( url, "cupHJbutton.gif" );
        snd = getAudioClip(url, "spacemusic.au");
    }

    public void paint(Graphics g){
        g.drawImage( img, 25, 25, this );
    }

    public void start(){
        snd.loop();
    }

    public void stop() {
        snd.stop();
    }
}
```

11.2.2 Java 图像编程

Java 的类库中有很多便于图像编程的类，这些类存在于 `java.awt`, `javax.swing`, `java.awt.image`, `com.sun.image` 等包中，这里举一个应用的例子。

在例 11-10 中，创建一个 `BufferedImage` 对象，将所要的“画”放到这个缓冲区里，再打开一个文件，将图像流编码后输入这个文件，这样就可以产生 `jpg` 图像文件，如图 11-5 所示。

例 11-10 `JpegCreate.java` 生成图像文件。

```
import java.io.*;
import java.util.*;
import com.sun.image.codec.jpeg.*;
import java.awt.image.*;
import java.awt.*;

public class JpegCreate {
    BufferedImage image;

    // 创建 jpg 文件到指定路径下
    public void createJpg(String path) {
        try {
            FileOutputStream fos = new FileOutputStream(path);
            BufferedOutputStream bos = new BufferedOutputStream(fos);
            JPEGImageEncoder encoder = JPEGCodec.createJPEGEncoder(bos);
            encoder.encode(image);
            bos.close();
        } catch (FileNotFoundException fnfe) {
            System.out.println(fnfe);
        } catch (IOException ioe) {
            System.out.println(ioe);
        }
    }

    public static void main(String[] args) {
        int width=400, height=200;
        int xLength=300, yLength=150;
        int count=5;

        Vector data=new Vector();
```

```
data.addElement(new Integer(100));
data.addElement(new Integer(120));
data.addElement(new Integer(150));
data.addElement(new Integer(40));
data.addElement(new Integer(5));

JpegCreate jg = new JpegCreate();
jg.image = new BufferedImage(width, height, BufferedImage.
TYPE_INT_RGB);
Graphics g = jg.image.getGraphics();

// 画坐标
g.setColor(Color.white);
g.fillRect(0, 0, width, height);
g.setColor(Color.blue);
g.drawLine(10,height-10,10,height-10-yLength);
g.drawLine(10,height-10,10+xLength,height-10);

// 连线
int yTo;
int yFrom = ((Integer)(data.elementAt(0))).intValue();
for (int i=1; i<count; i++) {
    yTo=((Integer)(data.elementAt(i))).intValue();
    g.drawLine(10+i*xLength/count,height-10,10+i*xLength/count,
height-15);
    g.drawLine(10+(i-1)*xLength/count,yFrom,10+i*xLength/count,
yTo);
    yFrom=yTo;
}

jg.createJpg("d:\\aaa.jpg");

}
}
```

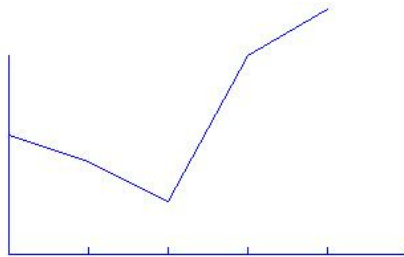


图 11-5 生成图像文件

11.2.3 Java 声音编程

Java 中提供了相当强的声音处理能力, 这些功能是由 javax.sound 包及其子包中的类形成的。由于篇幅的限制, 这里仅简单介绍 javax.sound.midi 包中的几个类。

MidiSystem 类代表 Midi 系统; Sequencer 接口代表设备, Sequence 类代表 Midi 音乐。下面的例子提供了播放声音的一段程序。

例 11-11 MidiFilePlay.java 播放 MIDI 声音。

```
import javax.sound.midi.*;
import java.io.*;
class MidiFilePlay
{
    public static void main( String [] args ){
        Sequencer sequencer;
        try {
            sequencer = MidiSystem.getSequencer();
            sequencer.open();

            File myMidiFile = new File("passport.mid");

            Sequence mySeq = MidiSystem.getSequence(myMidiFile);
            sequencer.setSequence(mySeq);
            sequencer.start();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

11.3 Java 数据库编程

数据库的应用十分广泛，Java 对数据库编程也提供了很好的支持，这里介绍一下数据库编程的基础，在实际编程时，用 Jbuilder, VisualCafe 等 IDE 工具，可以更快速地进行 Java 数据库编程。

Java 中访问数据库会用到 java.sql 包。Java 中访问数据库的基本方式称为 JDBC (Java Database Connectivity)。

11.3.1 Java 访问数据库的基本步骤

1. 装载驱动程序

Java 程序为了与 DBMS (数据库管理系统) 建立连接，首先需要装载驱动程序。

装载驱动程序只需要非常简单的一行代码。例如，要使用 JDBC-ODBC 桥驱动程序，可以用下列代码装载它：

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

不需要创建一个驱动程序类的实例并且用 DriverManager 登记它，因为调用 Class.forName 将自动将加载驱动程序类。

注意：在 Windows 中建立 ODBC 数据源，可以通过设置控制面板中的 ODBC 数据源来实现。

2. 与数据库建立连接

与数据库建立连接的一般方法是：

```
Connection con = DriverManager.getConnection(url, "myLogin", "myPassword");
```

如果正在使用 JDBC-ODBC 桥，JDBC URL 将以 jdbc:odbc 开始：余下 URL 通常是数据源名字或数据库系统。假设正在使用 ODBC 存取一个叫“JDBCDemo”的 ODBC 数据源，则 JDBC URL 是 jdbc:odbc:Fred。“myLogin”及“myPassword”为登录 DBMS 的用户名及口令。例如：

```
String url = "jdbc:odbc: JDBCDemo";  
Connection con = DriverManager.getConnection(url, " myLogin ",  
" myPassword ");
```

如果使用的是第三方开发的 JDBC 驱动程序，JDBC URL 的格式可查看相关的文档。如果装载的驱动程序识别了 JDBC URL，通过 DriverManager.getConnection()方法可以建立连接。

3. 创建 JDBC Statements 对象

Statement 对象用于把 SQL 语句发送到 DBMS。用 Connection 对象来创建 Statement 对象的实例。如：

```
Statement stmt = con.createStatement();
```

4. 执行 SQL 语句，并获得结果

使用上面例子中的 SQL 语句作为 Statements 对象的 executeUpdate() 或 executeQuery 方法的参数，则执行 SQL 语句。对 SELECT 语句来说，可以使用 executeQuery。要创建或修改表的语句，使用的方法是 executeUpdate。

可以写一个 SELECT 语句来取得这些值。下面的 SQL 语句中星号 (*) 表示选择所有的列。因为没有用 WHERE 子句来限制所选的行，因此下面的 SQL 语句选择的是整个表。

```
SELECT * FROM 表名
```

通过 executeQuery() 方法得到的是一个结果集 (ResultSet)，通过 ResultSet 的 next() 方法可以定位到不同的记录，而取得字段可以用 getInt() 及 getString() 等方法。

例 11-12 是一个演示使用 JDBC 的完整例子。

例 11-12 JDBCdemo.java 演示使用 JDBC。

```
/*
 * @(#)JDBCdemo.java    2000/06/18
 */
import java.sql.*;

/**
 * 演示 JDBC 操作数据库的各项功能，包括表的创建
 * (CREATE) 和删除 (DROP)，记录的插入 (INSERT)，
 * 选择 (SELECT) 和更改 (UPDATE) 等操作。
 * @version 1.00 2000/06/18
 * @author TangDashi
 * @since JDK1.2
 */
public class JDBCdemo
{
    public static void main(String args[])
    {
        try {
            Statement stmt;
            PreparedStatement pstmt;
            ResultSet rs;

            // 加载 jdbc-odbc 桥驱动程序
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

```
// 定义 JDBC URL
String url    = "jdbc:odbc:JDBCDemo";

// 得到与数据库的连接
Connection con = DriverManager.getConnection (url);

// 显示 URL 和连接信息
System.out.println("URL: " + url);
System.out.println("Connection: " + con);

// 得到一个 Statement 对象
stmt = con.createStatement();

// 如果表 DemoTable 已经存在, 则删除; 否则, 抛掷一个异常
System.out.println("DROP TABLE DemoTable, if it
exists.");
try{
    stmt.executeUpdate("DROP TABLE DemoTable");
}catch(Exception e){
    System.out.print(e);
    System.out.println("No existing table to delete");
}

// 在数据库中创建一个表 DemoTable
stmt.executeUpdate("CREATE TABLE DemoTable ( "
    + "test_id int,test_val char(15) not null)");
System.out.println("table DemoTable created!");

// 在表中插入一些值
stmt.executeUpdate("INSERT INTO DemoTable ( "
    + "test_id, test_val) VALUES(1,'One')");
stmt.executeUpdate("INSERT INTO DemoTable ( "
    + "test_id, test_val) VALUES(2,'Two')");
stmt.executeUpdate("INSERT INTO DemoTable ( "
    + "test_id, test_val) VALUES(3,'Three')");
stmt.executeUpdate("INSERT INTO DemoTable ( "
    + "test_id, test_val) VALUES(4,'Four')");
stmt.executeUpdate("INSERT INTO DemoTable ( "
    + "test_id, test_val) VALUES(5,'Five')");
```

```
// 得到另一个 Statement 对象
stmt = con.createStatement();

// 查询数据库中的表 DemoTable, 得到以 test_id 排序后的所有记录,
// 并存储在 ResultSet 对象 rs 中
rs = stmt.executeQuery("SELECT * from DemoTable ORDER
BY test_id");

// 显示表 DemoTable 中的所有记录
System.out.println("Display all results:");
while(rs.next())
{
    int theInt= rs.getInt("test_id");
    String str = rs.getString("test_val");
    System.out.println("\ttest_id= " + theInt + "\tstr
= " + str);
}

// 创建已准备好的语句, 更新 “DemoTable” 表中
// 某条记录的 test_val 字段
// 已准备好的语句接受两个参数
pstmt = con.prepareStatement(
    "UPDATE DemoTable SET test_val = ? WHERE test_id
= ?");

// 更改表 DemoTable 中的第 2 条记录的 test_val 字段的值
// 填充 UPDATE 语句中的 “? ”, 并执行 UPDATE 语句
pstmt.setString(1, "Hello!");
pstmt.setInt(2, 2);
pstmt.executeUpdate();
System.out.println("Update row number 2: OK.");

//显示表 DemoTable 中更新后的第 2 条记录
stmt = con.createStatement();
rs = stmt.executeQuery("SELECT * from DemoTable ORDER
BY test_id");
System.out.println("Display row 2:");
if (rs.next() && rs.next())
```



```
{  
    int theInt= rs.getInt("test_id");  
    String str = rs.getString("test_val");  
    System.out.println("\ttest_id= " + theInt + "\tstr  
    = " + str);  
}  
  
    con.close();    // 关闭与数据库的连接  
}catch( Exception e ) {  
    e.printStackTrace();  
}  
}  
}
```

11.3.2 使用 JTable 显示数据表

JTable 组件是 Swing 组件中比较复杂的组件，隶属于 javax.swing 包，它能以二维表的形式显示数据。类 JTable 在显示数据时具有以下特点。

- (1) 可定制性：可以定制数据的显示方式和编辑状态；
- (2) 异构性：可以显示不同类型的数据对象，甚至包括颜色、图标等复杂对象；
- (3) 简便性：可以以默认方式轻松地建立起一个二维表。

其可定制性可满足不同用户和场合的要求，异构性也正好符合数据库访问结果集中属性类型不一的特点。类 JTable 提供了极为丰富的二维表格操作方法，如设置编辑状态、显示方式、选择行列等。

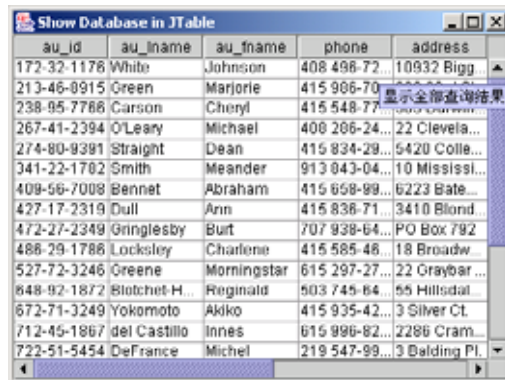
使用类 JTable 显示数据之前，应根据情况生成定制模型、单元绘制器或单元编辑器。类 AbstractListModel 用来定制用户自己的数据模型，这个类在后面要介绍。TableCellRenderer 接口用来定制单元绘制器，TableCellEditor 接口用来定制单元编辑器，这两个接口主要用于颜色对象的处理上，在示例中没有用到，不做过多说明。

类 AbstractTableModel 是一个抽象类，没有完全实现，不能实例化，使用时必须在程序中实现方法。它隶属于 javax.swing.table 包。

类 AbstractTableModel 提供了 TableModel 接口中绝大多数方法的默认实现。要想生成一个具体的 TableModel 作为 AbstractTableModel 的子类，至少必须实现下面三个方法：

- ☞ public int getRowCount();
- ☞ public int getColumnCount();
- ☞ public Object getValueAt(int row, int column);

下面给出一个例子，从数据库中查出数据并在 JTable 中显示，如图 11-6 所示。



au_id	au_lname	au_fname	phone	address
172-32-1176	White	Johnson	408 496-72...	10932 Bigg...
213-46-8915	Green	Marjorie	415 986-70...	
238-95-7766	Carson	Cheryl	415 548-77...	
267-41-2394	O'Leary	Michael	408 286-24...	22 Clevela...
274-80-9381	Straight	Dean	415 834-29...	5420 Colle...
341-22-1782	Smith	Meander	913 843-04...	10 Mississ...
409-66-7008	Bennet	Abraham	415 658-99...	6223 Bate...
427-17-2319	Dull	Ann	415 836-71...	3410 Blond...
472-27-2349	Gringlesby	Burt	707 938-64...	PO Box 782
486-29-1786	Locksley	Charlene	415 585-46...	18 Broadw...
527-72-3246	Greene	Morningstar	615 297-27...	22 Graybar ...
648-92-1872	Blotchet H...	Reginald	503 745-64...	55 Hillsdal...
672-71-3249	Yokomoto	Akiko	415 935-42...	3 Silver Ct...
712-45-1867	del Castillo	Innes	615 996-82...	2286 Cram...
722-51-5454	DeFrance	Michel	219 547-99...	3 Balding Pl...

图 11-6 使用 JTable 显示数据表

例 11-13 JDBCJTable.java 使用 JTable 显示数据表。

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.sql.*;
import java.util.*;
import javax.swing.table.*;

class JDBCJTable extends Frame
{
    AbstractTableModel tm; // 声明一个类 AbstractTableModel 对象
    JTable table;          // 声明一个类 JTable 对象
    JScrollPane scrollpane; // 声明一个滚动面板对象
    String titles[];        // 二维表列名
    Vector records;         // 声明一个向量对象

    public void init(){
        records = new Vector(); // 实例化向量
        tm = new AbstractTableModel(){
            public int getColumnCount(){
                return titles.length; // 取得表格列数
            }
            public int getRowCount(){
                return records.size(); // 取得表格行数
            }
            public Object getValueAt(int row,int column){

```

```

        if(!records.isEmpty()) // 取得单元格中的属性值
            return ((Vector)records.elementAt(row)).
                elementAt(column);
        else
            return null;
    }
    public String getColumnName(int column){
        return titles[column]; // 设置表格列名
    }
    public void setValueAt(Object value,int row,int
        column){
        // 数据模型不可编辑, 该方法设置为空
    }
    public Class getColumnClass(int c){
        return getValueAt(0,c).getClass(); // 取得列所属对象类
    }
    public boolean isCellEditable(int row,int column){
        return false; // 设置单元格不可编辑, 为默认实现
    }
};
}

public void start() throws SQLException,
ClassNotFoundException{
    // String driver="com.sybase.jdbc.SybDriver";
    // SybDriver sybdriver=(SybDriver)
    // Class.forName(driver).newInstance();
    // DriverManager.registerDriver(sybdriver);
    // String user="sa";
    // String password="";
    // String url="jdbc:sybase:Tds:202. 117.203.114:5000/
    // WORKER";
    // SybConnection connection=(SybConnection)
    // DriverManager.getConnection(url,user,password);

    // 加载 jdbc-odbc 桥驱动程序
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

    // 定义 JDBC URL

```

```
String url    = "jdbc:odbc:pubs";

// 得到与数据库的连接
Connection connection = DriverManager.getConnection (url);

// 显示查询结果
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery("select * from authors");
ResultSetMetaData meta = rs. getMetaData();// 得到元数据

int cols = meta.getColumnCount();
System.out.println( cols );
titles = new String[ cols ];
for( int i=0; i<cols; i++){
    titles[i] = meta.getColumnName(i+1);    // 得到列名
}

records.removeAllElements(); // 初始化向量对象
while(rs.next()){
    Vector rec_vector=new Vector();
    // 从结果集中取数据放入向量 rec_vector 中
    for ( int i=0; i<titles.length; i++ ){
        rec_vector.addElement(rs.getObject(i+1).
            toString());
    }
    records.addElement(rec_vector);
}

// 定制表格:
table=new JTable(tm);                // 生成自己的数据模型
table.setToolTipText("显示全部查询结果"); // 设置帮助提示
table.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
                                     // 设置表格调整尺寸模式
table.setCellSelectionEnabled(false); // 设置单元格选择方式
table.setShowVerticalLines(true);
table.setShowHorizontalLines(true);
scrollpane=new JScrollPane(table);    // 给表格加上滚动条
add( scrollpane );
```

```
        tm.fireTableStructureChanged();           // 更新表格
    }

    public static void main( String [] args ){
        JDBCJTable f = new JDBCJTable();
        f.init();
        try{
            f.start();
        }catch( Exception e){ e.printStackTrace(); }
        f.setSize( 400,300);
        f.setTitle( "Show Database in JTable" );
        f.show();
        f.addWindowListener( new WindowAdapter(){
            public void windowClosing( WindowEvent e)
            {System.exit(0);}
        });
    }
}
```

11.4 J2EE 及 J2ME 简介

目前, Java 2 平台有 3 个版本, 它们是适用于小型设备和智能卡的 Java 2 平台 Micro 版 (Java 2 Platform Micro Edition, J2ME)、适用于桌面系统的 Java 2 平台标准版 (Java 2 Platform Standard Edition, J2SE)、适用于创建服务器应用程序和服务的 Java 2 平台企业版 (Java 2 Platform Enterprise Edition, J2EE)。这三种版本从 Java 语言的角度上来看是一致的, 但在功能的裁减、系统的构架、应用的环境等方面又各有特色。在本书的前面章节中主要是基于 J2SE 的应用, 这里简要介绍 J2EE 及 J2ME 的特点。

11.4.1 J2EE 简介

1. J2EE 的概念

J2EE 是一种利用 Java 2 平台来简化企业解决方案的开发、部署和管理相关的复杂问题的体系结构。J2EE 技术的基础就是核心 Java 平台或 Java 2 平台的标准版, J2EE 不仅巩固了标准版中的许多优点, 例如“编写一次、随处运行”的特性、方便存取数据库的 JDBC API、CORBA 技术, 以及能够在 Internet 应用中保护数据的安全模式, 等等。同时, 还提供了对 EJB (Enterprise JavaBeans), Java Servlets API, JSP (Java Server Pages) 及 XML 技术的全面支持。其最终目的就是成为一个能够使企业开发者大幅缩短投放市场时间的体系结构。

J2EE 体系结构提供中间层集成框架用来满足无需太多费用而又需要高可用性、高可靠

性及可扩展性的应用的需求。通过提供统一的开发平台，J2EE 降低了开发多层应用的费用和复杂性，同时提供对现有应用程序集成强有力支持，完全支持 Enterprise JavaBeans，有良好的向导支持打包和部署应用，添加目录支持，增强了安全机制，提高了性能。

2 . J2EE 的多层模型

J2EE 使用多层的分布式应用模型，应用逻辑按功能划分为组件，各个应用组件根据他们所在的层分布在不同的机器上。事实上，Sun 设计 J2EE 的初衷正是为了解决两层模式 (Client/Server) 的弊端。在传统模式中，客户端担当了过多的角色而显得臃肿，在这种模式中，第一次部署的时候比较容易，但难于升级或改进，可伸展性也不理想，而且经常基于某种专有的协议——通常是某种数据库协议。它使得重用业务逻辑和界面逻辑非常困难。现在 J2EE 的多层企业级应用模型将两层化模型中的不同层面切分成许多层。一个多层化应用能够为不同的每种服务提供一个独立的层，以下是 J2EE 典型的多层结构。

(1) J2EE 应用程序组件：J2EE 应用程序是由组件构成的，J2EE 组件是具有独立功能的软件单元，它们通过相关的类和文件组装成 J2EE 应用程序，并与其他组件交互。

(2) 客户层组件：J2EE 应用程序可以是基于 Web 方式的，也可以是基于传统方式的。Web 层组件 J2EE Web 层组件可以是 JSP 页面或 Servlets。

(3) 业务层组件：业务层代码的逻辑用来满足银行、零售、金融等特殊商务领域的需要，由运行在业务层上的 enterprise bean 进行处理。

有三种 enterprise bean：会话 (session) bean，实体 (entity) bean，和消息驱动 (message-driven) bean。会话 bean 表示与客户端程序的临时交互。当客户端程序执行完后，会话 bean 和相关数据就会消失。相反，实体 bean 表示数据库的表中一行永久的记录。当客户端程序中止或服务器关闭时，就会有潜在的服务保证实体 bean 的数据得以保存。消息驱动 bean 结合了会话 bean 和 JMS 的消息监听器的特性，允许一个业务层组件异步接收 JMS 消息。

(4) 企业信息系统层：企业信息系统层处理企业信息系统软件，包括企业基础建设系统，例如企业资源计划 (ERP)，大型机事务处理，数据库系统和其他的遗留信息系统。例如，J2EE 应用组件可能为了数据库连接需要访问企业信息系统。

3 . J2EE 的核心 API 与组件

J2EE 平台由一整套服务 (Services)、应用程序接口 (APIs) 和协议构成，它对开发基于 Web 的多层应用提供了功能支持，如图 11-7 所示。下面是 J2EE 中常用的技术规范。

- ✧ JDBC(Java Database Connectivity)：数据库访问。
- ✧ JNDI(Java Name and Directory Interface)：名字和目录服务。
- ✧ EJB(Enterprise JavaBean)：开发和实施分布式商务逻辑。
- ✧ RMI(Remote Method Invoke)：远程对象调用。
- ✧ Java IDL/CORBA：Java 和 CORBA 的集成。
- ✧ JSP(Java Server Pages)：基于 Java 的活动网页。
- ✧ Java Servlet：运行服务端的应用程序。
- ✧ XML(Extensible Markup Language)：全面支持 XML。

- ✎ JMS(Java Message Service): 面向消息的中间件。
- ✎ JTA(Java Transaction Architecture): 事务监控。
- ✎ JTS(Java Transaction Service): 事务服务。
- ✎ JavaMail: 存取邮件服务器。
- ✎ JTA(JavaBeans Activation Framework): 处理 MIME 编码的邮件附件。

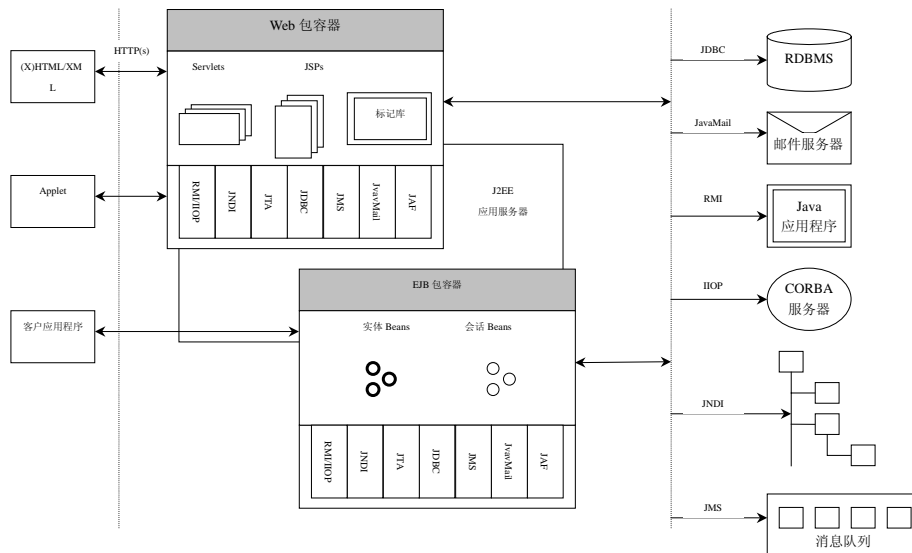


图 11-7 J2EE 的系统框架

11.4.2 J2ME 简介

J2ME (Java 2 Micro Edition) 是针对嵌入式设备及消费类电器的 Java 版本。J2ME 可广泛应用于多种设备。如，应用于手机（移动电话），可以使手机具有电话簿和电话铃声编辑功能、记事本功能、字典、图书、游戏、遥控家电和定时提醒等新的应用，并能访问电子邮件、即时消息、股票和电子地图等信息。由于 Java 是跨平台的语言，可以将第三方开发的软件方便地集成到各种设备中。

J2ME 的体系框架可以分为 3 层，从下到上分别是 VM，Configuration（配置）和 Profile（框架）。VM 负责建立 Java 虚拟机，解释 Java 代码。Configuration 负责建立核心类库，功能比较少（比如没有用户接口），主要面向水平市场。Profile 负责建立高级类库，功能丰富，面向垂直市场。

与 J2SE 相比，J2ME 针对手机等设备内存小、速度慢和 I/O 差的特点，J2ME 对 VM、Configuration 和 Profile 等 3 层结构做了特殊的实现，如在 VM 层，Motorola 在手机上移植了 KVM，KVM 只需要几百 KB 的内存就可以运行；在 Configuration 层，J2ME 规定了连接限制设备配置（Connected Limited Device Configuration，CLDC）。它适用于有双向网络连接但是硬件资源有限的设备；在 Profile 层，J2ME 规定了移动信息设备框架（Mobile Information Device Profile，MIDP），适用于手机或双向寻呼机。

J2ME 在不同的 Configuration 实现有不同的类库,如 CLDC 中的包有:java.lang, java.util, java.io, javax.microedition 等,而其他类库则没有定义。

J2ME 相关的软件,也可以从<http://java.sun.com>下载。下面举的一个例子,是用 J2ME 的 Wireless Toolkit 开发的程序。该程序是 MIDlet 的子类,与 Applet 相似,有按钮、文本框、事件处理等组件,但运行环境是在手机上,运行结果如图 11-8 所示。

例 11-14 HelloMIDlet.java 一个简单的 Java 手机程序。

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class HelloMIDlet extends MIDlet implements
CommandListener {

    private Command exitCommand;    // 命令按钮
    private Display display;        // 输出画面,即显示设备

    public HelloMIDlet() {
        display = Display.getDisplay(this);
        exitCommand = new Command("离开", Command.BACK, 1);
    }

    // 开始应用程序,创建文本框及命令按钮,并加入事件监听器
    public void startApp() {
        TextBox t = new TextBox("Hello MIDP 应用程序",
                                "Welcome to MIDP Programming", 256, 0);
        t.addCommand(exitCommand);
        t.setCommandListener(this);
        display.setCurrent(t);
    }

    // 暂停应用
    public void pauseApp() { }

    // 结束应用
    public void destroyApp(boolean unconditional) { }

    // 事件监听程序
    public void commandAction(Command c, Displayable s) {
        if (c == exitCommand) {
            destroyApp(false);
        }
    }
}
```



```
        notifyDestroyed();  
    }  
}  
}
```



图 11-8 一个简单的 Java 手机程序的运行结果

习题

1. 创建一个服务器，用它请求用户输入密码，然后打开一个文件，并将文件通过网络连接传送出去。创建一个同该服务器连接的客户，为其分配适当的密码，然后捕获和保存文件。在自己的机器上用 localhost（通过调用 `InetAddress.getByName(null)` 生成本地 IP 地址 127.0.0.1）测试这两个程序。
2. 修改前一练习的程序，用多线程机制对多个客户进行控制。
3. 通过阅读 JDK 的文档，了解 Java2D, Java3D, JavaSound 等 API 的内容。
4. 通过 JDK 中的 Demo 程序，了解 Java2D 中强大的图像处理能力。
5. J2SE, J2EE, J2ME 有什么差别？
6. 了解 J2EE 中的关键技术。
7. 下载 J2EE、J2ME 软件，并安装和阅读相关的文档。

参 考 文 献

- 1 印旻．Java 语言与面向对象程序设计教程．北京：清华大学出版社，2000
- 2 杨绍方．深入掌握 J2EE 编程技术．北京：科学出版社，2002
- 3 Bruce Eckle．Tinking in Java．PHP Hall, 1999
- 4 Sun Microsystem Inc. Java Progamming Language SL-275 Student Guide With Instructor Notes．Sun Microsystem Inc，2001

