新编计算机类本科规划教材

Java 程序设计实用教程

朱战立　沈　伟　编著

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内 容 简 介

Java 语言是目前最流行，也是最有前途的面向对象程序设计语言。本书将 Java 语言和面向对象程序设计方法相结合，以大量实例详细介绍 Java 的编程思想和编程方法。全书共分 13 章，主要内容包括：Java 语言基础、类、对象、继承、多态、接口、包、Java API 基础、图形用户界面、Java 小程序、异常处理、输入/输出流、多线程、数据库应用、网络通信和 JSP 简介。每章后都设计了大量的基本概念题和程序设计题。全书贯彻实用型教材的编写原则，概念叙述深入浅出，知识点结合实例讨论。

本书既可作为高等院校 Java 语言程序设计课程的教材，也可作为从事软件设计的工程技术人员的技术参考书。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Java 程序设计实用教程/朱战立，沈伟编著. —北京：电子工业出版社，2005.1
新编计算机类本科规划教材
ISBN 7-121-00715-0

Ⅰ. J… Ⅱ. ①朱…②沈… 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（2004）第 133361 号

责任编辑：李岩
印　　刷：北京牛山世兴印刷厂
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　邮编　100036
经　　销：各地新华书店
开　　本：787×1 092　1/16　印张：18.25　字数：467 千字
印　　次：2005 年 1 月第 1 次印刷
印　　数：5 000 册　　定价：24.00 元
骼

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系。联系电话：（010）68279077。质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

前　言

随着计算机网络的飞速发展，网站建设和基于网络的应用程序开发的需求越来越大。用户已经使用多年的许多计算机应用软件也在重新改写为基于网络的版本。面对网络化应用程序的巨大需求，选用哪种程序设计语言来进行开发呢？Java 语言是目前最佳的一种选择。

Java 语言是一种纯面向对象程序设计语言。Java 语言的支持网络应用编程、可跨平台使用、安全性好、支持线程功能等特点，使它成为非常适合网络应用程序开发的一种程序设计语言。

目前，高等学校计算机学科的所有本科专业基本上都开设了 Java 语言程序设计课程，其他许多学科或专业，如通信、仪器等，也都开设了 Java 语言程序设计课程；但根据作者的教学经验，实用类型的 Java 程序设计教材还比较缺乏。

本书的编写方针是：理论叙述深入浅出，设计举例丰富多样，设计要点结合实际例子给出。作者在编写过程中，概念介绍时力求文字准确简洁，设计举例时详细讨论设计要点。书中稍微复杂一些的设计举例都包括题目和要求，设计分析，程序代码，程序设计说明，程序运行结果及运行结果说明。另外，本书还介绍了 JDK 开发环境和 Tomcat 服务器的安装和设置方法，以及 Access 数据库和应用程序建立连接的方法。作者相信，这样的一本教材一定能受到教师和学生的欢迎。

本书内容丰富全面，全书共分 13 章，主要内容包括 Java 语言基础、类、对象、继承、多态、接口、包、Java API 基础、图形用户界面、Java 小程序、异常处理、输入/输出流、多线程、数据库应用、网络通信和 JSP 简介。这些内容可以满足大部分学校的教学要求。作者在每章后设计了大量的基本概念题和程序设计题。这些习题可以帮助学生巩固概念，并真正具备动手设计能力。

本书的所有例子都上机调试通过。

根据作者的教学体会，使用本教材授课约需 40～54 学时。课时不多时，后边几章的内容可根据情况少讲或不讲。

本书第 1～第 5 由朱战立编写，第 6～第 13 由沈伟编写。朱战立对全书进行了认真和反复的修改。本书的最终出版得到了许多人的帮助，刘天时和韩家新老师就本教材的编写大纲等问题和作者进行过多次深入探讨，马刚老师和研究生杨锦锋、李红雷、杨萌、秦金祥等协助作者做了许多工作。在本书完成之际，一并向他们表示诚挚的感谢。

尽管作者在写作过程中投入了大量的时间和精力，但由于水平有限，错误和不足之处仍在所难免，敬请读者批评指正。

编著者

2004 年 9 月 1 日

目　　录

第 1 章　概述 ········ 1

1.1　Java 语言简介 ········ 1

1.1.1　Java 语言的起源 ········ 1

1.1.2　Java 语言的版本 ········ 2

1.1.3　Java 语言的特点 ········ 2

1.2　Java 语言的运行机制 ········ 3

1.3　Java 开发运行环境 JDK ········ 5

1.3.1　JDK 安装 ········ 5

1.3.2　JDK 下 Java 程序的编辑、编译与运行 ········ 7

1.4　两种 Java 程序 ········ 9

1.4.1　Java Application ········ 9

1.4.2　Java Applet ········ 12

习题 1 ········ 13

第 2 章　Java 语言基础 ········ 14

2.1　标识符 ········ 14

2.2　变量和常量 ········ 15

2.2.1　变量 ········ 15

2.2.2　常量 ········ 16

2.3　基本数据类型 ········ 16

2.4　赋值语句 ········ 18

2.5　运算符和表达式 ········ 19

2.5.1　运算符及其分类 ········ 19

2.5.2　运算符的优先级 ········ 24

2.5.3　表达式 ········ 24

2.6　流程控制语句 ········ 25

2.6.1　分支语句 ········ 26

2.6.2　循环语句 ········ 30

2.6.3　break 语句和 continue 语句 ········ 35

2.6.4　return 语句 ········ 36

2.7　程序注释 ········ 37

2.8　数组 ········ 37

2.8.1　一维数组 ········ 37

2.8.2　二维数组 ········ 42

2.8.3　不规则的二维数组 ········ 44

2.9　字符串 ········ 46

习题 2 47
第 3 章　类和对象 51
3.1 面向对象程序设计 51
3.1.1 面向对象程序设计的基本概念 51
3.1.2 类 52
3.1.3 对象 52
3.2 类 52
3.2.1 类声明 52
3.2.2 类主体设计 53
3.2.3 构造方法 55
3.3 对象 56
3.3.1 main 方法 56
3.3.2 对象的创建和初始化 57
3.3.3 对象的使用 59
3.3.4 垃圾对象的回收 59
3.3.5 实例成员变量与类成员变量 60
3.3.6 实例方法与类方法 61
3.3.7 方法的重写 62
3.4 包 63
3.4.1 包的建立方法 64
3.4.2 包的使用方法 65
3.4.3 包的访问权限 66
3.4.4 系统定义的包 69
3.5 内部类 70
3.6 类的封装性 71
3.7 设计举例 72
习题 3 75
第 4 章　类与继承 77
4.1 面向对象的基本概念：继承 77
4.2 继承 78
4.2.1 子类和父类 78
4.2.2 创建子类 79
4.2.3 方法的三种继承形式 83
4.2.4 方法的多态性 86
4.3 抽象类和最终类 88
4.3.1 抽象类 88
4.3.2 最终类 90
4.4 接口 90
4.4.1 定义接口 90
4.4.2 实现接口 91

4.4.3 系统定义的接口 93
习题 4 94
第 5 章 Java API 基础 96
5.1 Java API 综述 96
5.2 语言包（java.lang）简介 97
5.2.1 Object 类 97
5.2.2 System 类 98
5.2.3 Class 类 100
5.2.4 Runtime 类 100
5.2.5 Float 类 101
5.2.6 String 类 101
5.2.7 Math 类 103
5.3 实用包（java.util）简介 104
5.3.1 Arrays 类 104
5.3.2 Vector 类 105
5.3.3 Data 类和 Calendar 类 105
5.3.4 Enumeration 接口 107
5.4 综合应用举例 107
习题 5 110
第 6 章 图形用户界面 111
6.1 图形用户界面和 AWT 111
6.1.1 图形用户界面的基本概念 111
6.1.2 AWT 简介 111
6.2 Component（组件）类及其子类 112
6.2.1 Component（组件）类 113
6.2.2 Container（容器）类及其子类 114
6.2.3 Component 类的其他子类 114
6.3 MenuComponent（菜单组件）类及其子类 121
6.4 AWT 中的绘图方法和常用类 123
6.4.1 Component（组件）中的绘图方法 123
6.4.2 Color（颜色）类 124
6.4.3 Font（字体）类 125
6.4.4 Graphics（图形工具）类 125
6.5 布局管理器 127
6.5.1 FlowLayout 类 128
6.5.2 BorderLayout 类 129
6.5.3 GridLayout 类 130
6.6 Java 的事件处理 131
6.6.1 事件处理的基本过程 131
6.6.2 Java 的事件处理 132

6.7 设计举例 …… 139
习题 6 …… 143
第 7 章 Java 小程序 …… 145
7.1 Java 小程序概述 …… 145
7.2 Java 小程序的特点、设计方法和运行环境 …… 146
7.2.1 Java 小程序的特点 …… 146
7.2.2 Java 小程序的基本设计方法和运行环境 …… 146
7.3 Java 子程序的生命周期 …… 148
7.3.1 Applet 类的继承关系 …… 148
7.3.2 Java 子程序的生命周期 …… 148
7.4 HTML 与 Applet …… 150
7.4.1 与 Applet 相关的 HTML 属性简介 …… 150
7.4.2 HTML 文件和 Applet 的数据传递 …… 151
7.5 两种典型的 Applet 程序设计 …… 152
7.5.1 在 Applet 中加入图像 …… 153
7.5.2 Applet 中的人机交互 …… 154
7.6 Applet 的安全限制和 JAR 文件 …… 156
7.6.1 Applet 的安全限制 …… 156
7.6.2 JAR 文件 …… 156
习题 7 …… 157
第 8 章 异常处理 …… 158
8.1 异常和异常处理的两种方法 …… 158
8.1.1 异常的基本类型 …… 158
8.1.2 if-else 形式的异常处理方法 …… 158
8.1.3 Java 的异常处理方法 …… 160
8.2 Java 的异常类 …… 161
8.3 Java 的异常处理方法 …… 163
8.4 异常的抛出和处理 …… 167
8.4.1 在同一个方法中抛出异常和处理异常 …… 168
8.4.2 抛出异常和处理异常的方法不是同一个方法 …… 169
8.5 自定义的异常类 …… 170
习题 8 …… 171
第 9 章 输入/输出流 …… 173
9.1 数据流的概念 …… 173
9.1.1 输入流和输出流 …… 173
9.1.2 字节流和字符流 …… 174
9.1.3 Java 的标准数据流 …… 174
9.2 基本输入/输出类 …… 175
9.2.1 InputStream（字节输入流）类 …… 175
9.2.2 OutputStream（字节输出流）类 …… 178

9.2.3 Reader（字符输入流）类 ······ 181
9.2.4 Writer（字符输出流）类 ······ 183
9.2.5 FileReader 和 FileWriter（字符文件输入/输出流）类 ······ 184
9.3 对象流 ······ 186
9.4 文件的操作 ······ 189
9.4.1 File（文件）类及其应用 ······ 189
9.4.2 RandomAccessFile（随机存取文件）类 ······ 193
习题 9 ······ 195
第 10 章 多线程 ······ 196
10.1 线程的基本概念 ······ 196
10.1.1 进程和线程 ······ 196
10.1.2 线程的生命周期和状态 ······ 197
10.2 Thread（线程）类和 Runnable（可运行）接口 ······ 197
10.2.1 Thread（线程）类 ······ 197
10.2.2 Runnable（可运行）接口 ······ 200
10.3 线程的状态和状态控制 ······ 202
10.3.1 线程的生命周期和状态 ······ 202
10.3.2 线程分组 ······ 203
10.3.3 线程的优先级 ······ 205
10.4 线程间的互斥 ······ 207
10.4.1 共享资源问题 ······ 207
10.4.2 互斥线程的设计方法 ······ 210
10.5 线程间的同步 ······ 212
10.5.1 共享资源的同步问题 ······ 212
10.5.2 同步线程的设计方法 ······ 215
10.6 综合应用举例 ······ 218
习题 10 ······ 221
第 11 章 数据库应用 ······ 223
11.1 JDBC 和 SQL 简介 ······ 223
11.1.1 SQL 简介 ······ 223
11.1.2 JDBC 简介 ······ 227
11.2 建立应用程序和数据库连接的环境配置 ······ 228
11.3 数据库应用编程 ······ 230
11.3.1 建立连接 ······ 230
11.3.2 操作数据库 ······ 233
11.3.3 处理操作结果 ······ 237
习题 11 ······ 241
第 12 章 网络通信 ······ 243
12.1 网络通信的基本概念 ······ 243
12.1.1 通信主体的定位 ······ 243

12.1.2 TCP 协议和 UDP 协议 245
12.1.3 网络程序设计的基本方式 245
12.2 使用 URL 类访问网络资源 246
12.2.1 资源定位器 URL 和 URL 类 246
12.2.2 URLConnection 类和 InetAddress 类 249
12.3 连接通信 252
12.3.1 Socket 和连接 252
12.3.2 Socket 类和 ServerSocket 类 252
12.3.3 Client/Server 结构的通信实例 254
12.4 数据报通信 259
习题 12 261
第 13 章 JSP 简介 262
13.1 网络服务和动态网站的基本概念 262
13.1.1 计算机网络和网络服务 262
13.1.2 网站和 JSP 263
13.2 JSP 的原理和特点 264
13.2.1 Servlet 和 JSP 264
13.2.2 JSP 的运行机制 264
13.2.3 JSP 的主要特点 265
13.3 运行环境的建立 265
13.3.1 安装支持 JSP 的服务器 265
13.3.2 JSP 运行环境的测试 266
13.4 JSP 的基本语法和内置对象 268
13.4.1 JSP 的基本语法 268
13.4.2 JSP 的指令语句 270
13.4.3 JSP 的内置对象 271
13.5 应用举例 274
习题 13 278
参考文献 279

第 1 章　概　　述

教学要点

本章内容主要包括 Java 语言的基本概念（包括 Java 语言的起源、版本、特点等），Java 语言的运行机制，Java 语言的开发运行环境 JDK（包括 JDK 的安装和 JDK 的使用方法），两种 Java 程序（Java Application 和 Java Applet）的基本形式。

要求了解 Java 语言的基本概念，掌握 Java 语言的运行机制和开发运行环境 JDK，了解两种 Java 程序的基本形式。

本章主要是让读者对 Java 语言和 Java 的开发运行环境有一个基本的了解；内容主要包括 Java 语言的发展过程、版本变化、特点以及基本的运行机制，Java 语言的 JDK 开发环境，以及用 JDK 和 JBuilder 开发 Java Application 和 Java Applet 两种类型程序的编辑、编译和运行过程。

1.1　Java 语言简介

1.1.1　Java 语言的起源

1991 年，Sun 公司为了向家用电子消费市场进军，成立了一个代号为 Green 的项目组。其目标是开发一个分布式系统，让人们可以利用网络远程控制家用电器。由于家用电器制造商众多、且制造标准各异，所以项目组希望新系统应当具有独立于软件平台的特征，并且新系统要安全易用。开始时，项目组采用当时广泛使用的 C++语言进行系统开发。但是由于 C++语言太复杂，安全性也难于满足要求，所以不得不放弃，转而研究设计出了一套新的程序设计语言用于系统的开发。这个新的程序设计语言就是 Java 语言的前身，被命名为 Oak（橡树）。

Oak 以 C++语言为蓝本，吸收了 C++中符合面向对象程序设计要求的部分，同时加入了一些满足网络设计要求的部分。可惜的是，由于一些商业上的原因，Sun 公司在以 Oak 为程序设计语言投标一个交互式电视项目时，却未能中标，这使得 Oak 语言的进一步发展一度遇到很大的问题。

20 世纪 90 年代中期，WWW 的影响在 Internet 上越来越大，WWW 浏览器开始在市场上出现。很多有远见的人意识到，计算机网络和网络应用的浪潮即将到来。1994 年，Green 项目组成员在认真分析计算机网络应用的特点后，认为 Oak 满足网络应用要求的平台独立性、系统可靠性和安全性等要求。为了展示 Oak 在网络应用方面的优良性能，项目组用 Oak 设计了一个称为 WebRunner（后来称为 HotJava）的 WWW 浏览器。1995 年 5 月 23 日，Sun 公司正式发布了 Java 和 HotJava 两项产品。

Java 语言一经推出，就受到了业界的关注。Netscape 是第一个认可 Java 语言的公司。1995 年 8 月，Netscape 公司将 Java 解释器集成到它的主打产品 Navigator 浏览器中。接着，Microsoft 公司在 Internet Explorer 浏览器中认可了 Java 语言，从而使 Java 终于站稳了脚跟，开始了自己的发展历程。

1.1.2 Java 语言的版本

到目前为止，Sun 公司先后发布了五个主要的 Java 语言版本。其发布时间和主要特点如下。

① Java 1.0：1995 年推出的主要用于网页设计的版本，目前所有的浏览器都支持该版本。

② Java 1.1：1997 年推出的一个版本，在用户界面和事件处理方面有改进，并且加入了 JavaBean 组件技术。

③ 带有 JDK 1.2 的 Java 2：1988 年发布的具有重大改进的版本，在图形化用户界面、数据库互连以及其他许多方面做了改进。这个版本也称作带有 JDK 1.2 的 Java 1.2 版本。由于该版本具有重大改变，所以从 Java 1.2 版本后的 Java 语言也称作 Java 2 语言。

④ 带有 JDK 1.3 的 Java 2：2000 年发布的版本，在多媒体应用、编译速度等方面做了改进。这个版本也称作带有 JDK 1.3 的 Java 1.3 版本。

⑤ 目前的最新版本是 JDK 1.4 版本。

JDK 是 Java Development Kit（软件开发工具集）的缩写。JDK 为用户提供了进行 Java 开发的、基于 DOS 环境的开发工具。Sun 公司新版本的帮助文档中也把 JDK 称作 Java SDK（Java Software Development Kit）。

Java 开发工具集可以在 Sun 公司网站（http://java.sun.com）上免费获取。这也是 Java 语言迅速发展的一个重要因素。当 Java 语言有较大改进时，Sun 公司总是同时推出新的 Java 语言版本和支持新版本的 Java 开发工具集版本。所以，Sun 公司的 JDK 新版本总是第一个支持 Java 新版本功能的开发工具，这通常比其他的 Java 开发工具早六个月到一年。

除了 Sun 公司的 JDK 外，Sun 公司和其他软件公司还推出了许多支持 Java 开发的工具。常见的 Java 开发工具还有：

① Sun 公司的 Forte for Java——这是一个基于 Windows 环境集成开发环境；

② Borloand 公司的 JBuilder——这也是一个基于 Windows 环境的、集成开发环境，使用非常方便。

从 2000 年 12 月开始，Sun 公司又推出了 J2ME, J2SE, J2EE 三个版本。

① J2ME——Java 2 Micro Edition。嵌入式 Java 消费电子平台。J2ME 使嵌入式 Java 技术成为无处不在的计算模式。为新的企业、商务、娱乐和通信等应用领域提供解决方案。

② J2SE——Java 2 Standard Edition。用于工作站、PC 机的 Java 标准平台。

③ J2EE——Java 2 Enerprise Edition。可扩展的企业级应用 Java 2 平台。J2EE 是分布式企业软件组件架构的规范，具有灵活性、简化的集成性、便捷性，以及 J2EE 服务器之间的互操作性等特点。

1.1.3 Java 语言的特点

1996 年，Java 语言的主要设计者、Green 项目组的 James Gosling 和 Henry McGilton 在 Java 语言白皮书中，说明了他们最初的 Java 语言设计目标和实现这些目标的主要方法。

① 简单、面向对象和与现有高级语言形式类似。Java 语言是一种简单的、不用太多学

习训练就能掌握的语言，是一种面向对象的程序设计语言，是一种和目前流行的高级语言（如C++语言）形式类似的语言。

② 鲁棒性和安全性。使用 Java 语言能生成高可靠性的软件，这主要靠 Java 语言的编译时检查和运行时检查两层检查机制实现。Java 语言主要用于分布式的网络环境，因此安全性是非常重要的。Java 语言限制应用程序从外部侵入系统传播病毒或破坏用户的文件系统。

③ 结构中立（architecture-neutral）。在网络环境下，用 Java 语言编写的应用程序必须适应各种各样的硬件结构和各种各样的操作系统平台，并能与其他程序设计语言接口协同工作。Java 语言实现这样的设计目标的主要方法是，先编译产生字节码（字节码是一种不依赖于任何硬件和操作系统的中间代码），然后用解释的方法产生最终在具体计算机上运行的机器码。

④ 高性能。通常，解释型程序设计语言运行速度较慢。Java 语言是一种半解释型语言，但Java 语言程序的解释执行能全速进行，不用检查运行时的环境（因该环境已在编译时保证）。另外，Java 语言的自动垃圾回收线程定义成优先级很低的后台线程，该线程在系统空闲时启动执行，这也保证了 Java 语言的运行速度。

⑤ 解释型、多线程和动态性。只要安装了 Java 解释器，Java 字节码就能直接在任何计算机上解释执行。这种解释执行方式是简单的、递增式的，对系统要求很低。这能极大地方便用户程序的开发，并使开发过程很快。Java 的多线程机制可提高复杂应用程序的运行速度，并且这种多线程机制有系统提供的同步功能做安全保证，这些在网络应用环境下是十分必要的。Java 的对象绑定是动态的，类也是在被要求的时候才被从指定的任意位置导入的。这种动态性可保证不断地把最新的应用系统改进，方便地融合进应用系统中。这必将促进基于Internet 的电子商务发展。

从上述 Java 语言设计者最初的设计目标，以及目前各种使用 Java 语言的应用系统使用情况来看，Java 语言具有如下显著特点。

① Java 是一种面向对象高级程序设计语言。面向对象技术是目前软件设计的主流和普遍使用的技术。和 C++等语言相比，Java 语言是一种纯面向对象的高级语言。

② 独立于各种操作平台。Java 采用先按编译方式翻译到字节码，然后再把字节码按解释方式翻译到具体机器的机器语言。这种半编译、半解释的高级语言翻译方式，使采用 Java语言编写的应用系统具有独立于各种硬件环境和各种操作系统平台的特点，真正做到程序代码一次编写、多次使用。

③ 安全性好：安全性是网络应用系统必须考虑的重要问题，Java 语言从最初设计就充分考虑了安全性问题，杜绝了病毒传播和用户重要信息泄露的各种可预见的渠道。

④ 多线程：现在的操作系统都支持多任务和分时，允许在同一时间运行多个程序。同一时间运行多个程序主要有多进程和多线程两种方式，而多线程是一种效率更高的方式。Java语言不仅支持多线程，而且为程序设计人员提供了实现线程同步机制的类库包。同步机制能保证多线程正确运行。

1.2 Java 语言的运行机制

目前大多数的高级程序设计语言，其应用程序编译产生的机器指令都与具体机器的硬件一定程度上相关。这种情况的后果是：当一个应用软件完成最后的编码设计后，在不同的硬件环境下编译生成的机器码是不一样的。因此，编译生成的可执行文件，只能在与编译时相

同的机器硬件环境下运行。否则，要么运行不成，要么运行时会出错。

Java 语言解决上述问题的方法是采用半编译、半解释的方式把高级语言的程序代码翻译成机器语言代码。为此，设计了一种称为字节码的代码。**字节码**（Bytecode）是一种和任何具体机器环境无关的中间代码。在 Java 程序编写完成后，首先，通过 Java 编译器把 Java 语言翻译成字节码，然后，通过 Java 解释器把字节码翻译成机器码。由于此种翻译方式既有编译又有解释，所以称作半编译、半解释方式。这样的过程如图 1.1 所示。

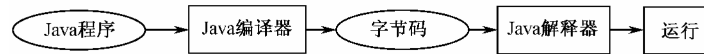


图 1.1　Java 语言程序的运行流程

Java 语言程序文件以.java 为后缀。Java 程序编写完后，用开发环境下的编译器编译生成字节码，字节码文件以.class 为后缀。这样的应用系统交付给用户后，只要用户计算机上安装有 Java 虚拟机，就可以把字节码翻译成相应计算机上的机器码。**Java 虚拟机**（Java Virtual Machine，简称 JVM）上有一个 Java 解释器，Java 解释器按解释方式把字节码翻译成具体硬件环境和操作系统平台下的机器码。这也就是说，Java 程序是相同的，只是不同运行平台上的 Java 虚拟机不同。图 1.2 说明了 Java 虚拟机的工作方式。

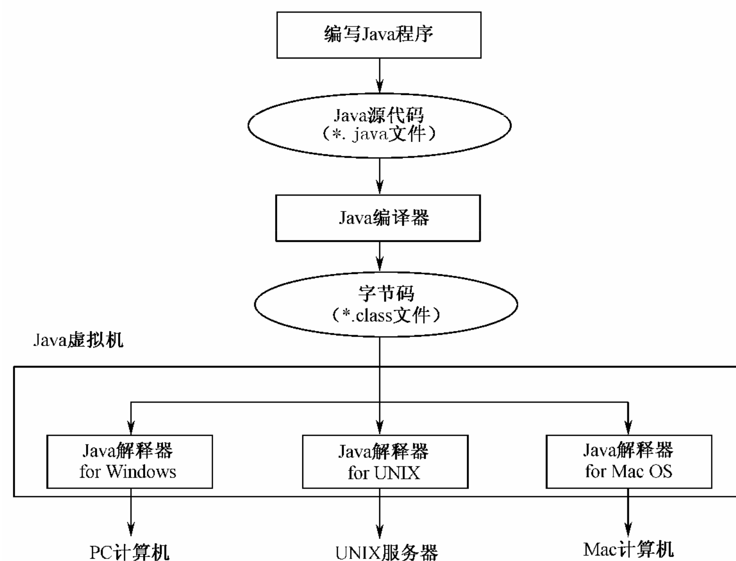


图 1.2　Java 虚拟机的工作方式

Java 语言这种一次编写、任意运行（write once, run anywhere）的方式有效地解决了目前大多数高级程序设计语言需要针对不同的机器编译产生不同的机器码的问题，即硬件环境和操作系统平台的异构问题。特别地，用 Java 语言开发的软件主要用在网络环境下。网络环境下应用软件的开发模式多是 B/S（Browse/Server）模式，即应用软件放在服务器端，客户端通过浏览器来进行应用。B/S 模式的应用系统特别需要异构环境的支持。

嵌入 HTML 中的 Java 程序称作 Applet 程序（小程序）。开发人员编写好 Applet 程序后，用开发环境下的编译器编译生成字节码文件，并把这样的字节码文件和 HTML 文件放在服务器上。当某个客户通过浏览器浏览相应服务器上的相应网页时，就把相应的 HTML 文件和 Applet 下载到了客户的计算机上，浏览器上都安装有 Java 虚拟机。此时，Java 虚拟机中

的 Java 解释器将按解释方式执行该 Applet，并把执行结果显示在网页上。

读者也许有疑问：既然只要在浏览器上安装了解释器就能解决应用软件运行环境的异构问题，那么为什么还需要先做一次编译，生成字节码呢？这是因为，如果把 Java 源程序在浏览器上按解释方式运行，那么客户计算机的运行速度将非常慢。

为了提高 B/S 模式客户计算机的响应速度，一方面，网络的频宽和流速在不断提高；另一方面，Java 虚拟机的性能也在不断提高。目前，用硬件直接实现的 Java 虚拟机正在加紧开发研究中。当 Java 虚拟机采用硬件直接实现时，其运行速度将大大快于目前的运行速度。

1.3 Java 开发运行环境 JDK

为开发和运行 Java 程序，计算机上必须安装相应的开发环境和开发工具。本节介绍 DOS 环境下的 Java 开发运行环境 JDK。Windows 环境下的 Java 开发运行环境 Jbuilder 在 1.4 节结合实例给出。

1.3.1 JDK 安装

1．程序清单

为了建立基于 JDK 的 Java 程序开发运行环境，需准备以下一些程序和文档。

① j2sdk 1_3_1-win.exe（JDK 开发包）（不同的版本后面的序号不同，最新的是 j2 sdk 1_4-win.exe）。

② j2sdk1_3_1-doc.zip（相应的帮助文档）。帮助文档只是为了学习用，也可没有。

③ Uedit32i.exe 或记事本（文本编辑器）。

前两个文件可以在 Sun 公司网站（http://java.sun.com/products）下载。Sun 公司在不断推出新的 JDK 版本，并把新版本在网站上及时发布。如果读者在该网站发现了新的 JDK 版本，也可下载新的 JDK 版本。

由于 JDK 没有提供一个 Windows 下的可视化的集成开发环境，所以必须自己准备编辑器。UltraEdit、记事本、EditPro 等都可以用来编写 Java 源程序。

2．安装过程

（1）安装 JDK 开发包

运行 JDK 开发包 j2sdk1_3_1-win.exe，安装过程中，可以设置安装路径并选择组件，也可按系统默认的安装路径和默认的安装组件安装。作者设置的安装路径为 C:\，因此，安装完成后，所有的 JDK 文件和子文件夹都在 C:\jdk1.3.1 文件夹中。

安装成功后，C:\jdk1.3.1 中的文件夹及用途如表 1.1 所示。

表 1.1　JDK 的文件夹结构

文 件 夹 名	用　　途
bin	包含编译器、解释器等可执行文件
demo	包含源代码的程序示例
include	头文件，用于本地机的 C 语言
include-old	头文件，用于兼容支持旧接口
jre	Java 运行时环境的根路径
lib	可执行程序使用的包文件

为了能正确使用 JDK，需要手工配置一些环境变量，Windows 98 和 Windows NT/2000 的配置略有不同，下面分别介绍。

（2）Windows 98 的环境配置

在 C:\autoexec.bat 文件中，添加以下内容：

```
set path=%path%;c:\jdk1.3.1\bin
set classpath=.;c:\jdk1.3.1\lib
```

path 语句设置了 Java 编译运行程序的路径。其中，%path%表示保持系统原来的设置不变，C:\jdk1.3.1 为 Java 程序安装路径。set classpath 语句设置了 Java 包的路径，其中最前面的“.”表示在当前工作路径下可执行 Java 程序（若设置中无此圆点运行时会出错），c:\jdk1.3.1\lib 为 Java 系统类包的路径。当然，如果安装的 Java 版本不同，或安装的路径不同，上述内容要做相应的改变。

上述修改完成后存盘，并重新启动计算机，这样就完成了环境设置。

注意：上述 autoexec.bat 文件中不允许有空格，否则系统不能正常运行。

（3）Windows NT/2000 的环境配置

在桌面“我的电脑”图标上单击鼠标右键，选择“属性”选项，出现“系统特性”对话框，如图 1.3（a）所示，在“高级”选项卡中单击“环境变量”按钮。在“系统变量”栏框中，找到 Path 选项。其操作窗口如图 1.3（b）所示。

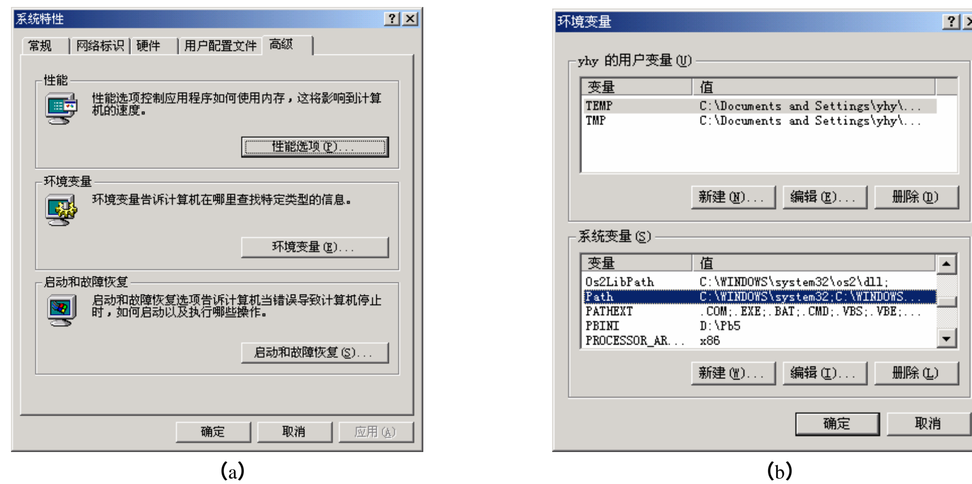


图 1.3　“系统特性”与“环境变量”对话框

单击“编辑”按钮，将“;c:\jdk1.3.1\bin”添加到变量值文本框中，单击“确定”按钮结束编辑变量。其操作窗口如图 1.4 所示。

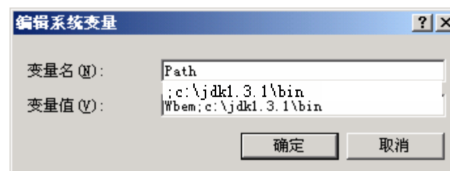


图 1.4　“编辑系统变量”对话框

同样方法再设置 classpath 环境变量，变量值要添加的字符串为“.;c:\jdk1.3.1\lib”。

设置完成后，在 DOS 窗口下，输入 java 和 javac 后，如果出现其用法的参数提示信息，则安装正确；否则安装不正确，需重新安装。

（4）安装 JDK 帮助文档

JDK 帮助文档可帮助读者学习 JDK 的使用方法和完整地掌握 JDK 的各种资料。要安装 JDK 帮助文档，用户的下载软件中必须包括 j2sdk1_3_1-doc.zip 文件。这是一个压缩文件，其安装方法是直接运行该压缩文件，此时将在 C:\jdk1.3.1\路径下再创建一个 jdk1.3 文件夹。用浏览器或 Windows 的资源管理器浏览 index.html 文件，可看到帮助文档主页如图 1.5 所示。该窗口左上窗显示 Java 的包，左下窗显示所选包相应的类，右窗显示相应类中成员变量和方法的说明文档。

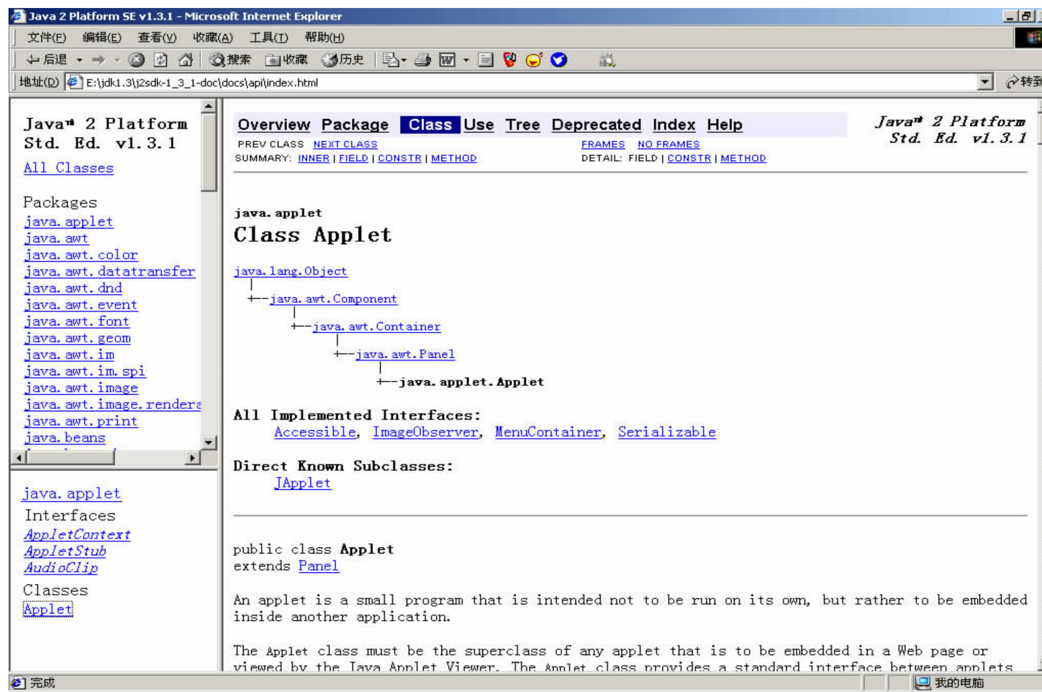


图 1.5　JDK 帮助文档主页

（5）建立一个工作文件夹

工作文件夹用于放置读者自己编写的 Java 程序。如果读者使用 JDK 工具开发 Java 程序，考虑到 JDK 工具是 DOS 环境下的命令操作方式，建议读者创建的工作文件夹名使用英文字母，不要使用汉字。

1.3.2　JDK 下 Java 程序的编辑、编译与运行

为了清楚地解释 JDK 下 Java 程序的编辑、编译与运行过程，本节用一个最简单的例子说明该过程。

【例 1.1】　设计显示 Hello!字符串的 Java 程序，并用 JDK 运行该程序。

1．编辑

用编辑器，如 UltraEdit 或记事本等编写 Java 源程序。记事本编辑窗口及键入的源程序如图 1.6 所示。

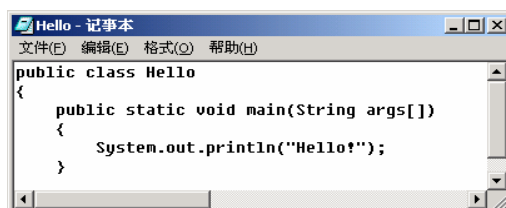


图 1.6　编辑窗口和程序

图 1.6 所示的 Java 程序中，class 是类声明的关键字，Hello 是类名。在 main 方法中，用标准输出语句 System.out.println 在屏幕上显示一个字符串“Hello!”。

把上述 Java 源程序编辑完成后保存到磁盘。保存时要注意：①Java 源程序文件名必须和类名完全一样；②Java 文件名的命名也是大小写敏感的；③Java 源程序文件名后缀必须为.java。所以，上述 Java 源程序的文件名为 Hello.java。

2．Java 程序的编译

在 JDK 下，Java 程序的编译与运行在 MS-DOS 窗口中进行。方法是：打开 MS-DOS 窗口，进入保存 Java 源程序文件的文件夹（如 d:\myjava 文件夹），输入编译命令：

```
d:\myjava>javac Hello.java
```

编译命令 javac 将对当前文件夹中的 Hello.java 文件进行编译。如果编译正确，将会产生相应的字节码文件（也称作类文件）Hello.class；如果编译时发现错误，系统将终止编译并给出出错信息。如果系统未找到 javac 命令，则说明 path 环境变量设置不正确。

3．Java 程序的运行

输入运行命令：

```
d:\myjava>java Hello
```

运行命令 java 可运行字节码文件 Hello.class，此时.class 后缀可省略。运行结果在屏幕上显示一行字符串“Hello!”，编译和运行结果如图 1.7 所示。

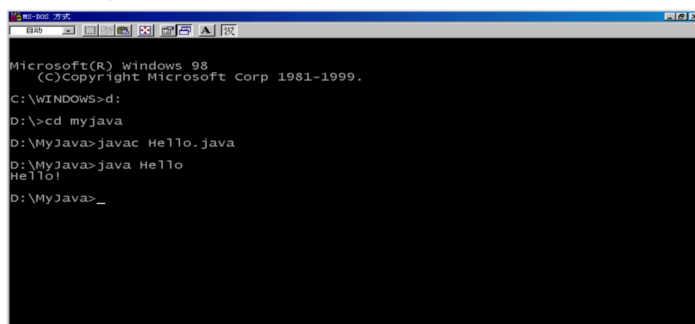


图 1.7　编译、运行和结果

这样，第一个 Java 程序就编辑、编译和运行成功了。

1.4　两种 Java 程序

Java 程序主要有两种：Java Application（应用程序）和 Java Applet（小程序）。本节初步介绍这两种类型的 Java 程序，使读者对 Java 程序的初步形式有一个初步的了解。

1.4.1　Java Application

Java Application 一般称作 Java 应用程序，它是可独立执行的应用程序。

例 1.1 已经给出一个 Java Application 程序的例子，这里我们再次用此例说明 Java Application 程序的特点和编辑、编译、运行过程。为了丰富教材内容，并方便使用不同开发环境的读者使用本教材，这里用 JBuilder 集成开发环境来完成此例。JBuilder 集成开发环境的安装十分简单，一旦安装成功则自动完成环境变量配置。

【例 1.2】　设计显示“Hello!”字符串的 Java Application，并用 JBuilder 运行该程序。

用 JBuilder 集成开发环境开发和运行 Java Application 程序的过程如下。

① 创建工程文件。JBuilder 在运行 Java 程序时需要的系统资源很多，JBuilder 把一个可独立运行程序所需要的所有资源称作一个工程。这些资源都包含在工程文件中，因此，首先要创建工程文件。

JBuilder 的窗口结构如图 1.8 所示。选择“File”菜单下的“New Project”菜单，会弹出一个如图 1.9 所示的窗口。

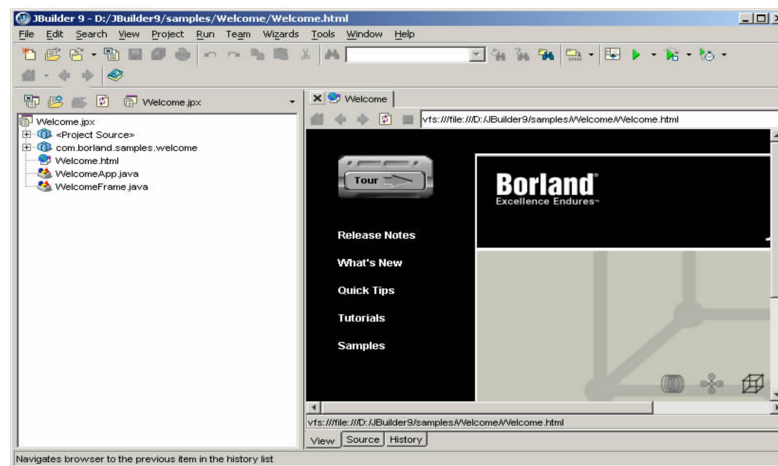


图 1.8　JBuilder 的窗口结构

图 1.9 是新建工程文件的窗口，在“Name”栏框键入工程文件名（这里是 MyProject），然后单击“Finish”按钮。

② 创建类文件。在 JBuilder 的窗口中，选择“File”菜单下的“New Class”菜单，会弹出一个如图 1.10 所示的窗口。首先在“Package”栏框中键入刚才建立的工程文件名 MyProject，然后在“Class Name”栏框中键入文件名，这里文件名是 Hello。然后单击“OK”按钮。

注意：(a) 文件名不需要后缀.java；(b) 文件名必须和类名一样。

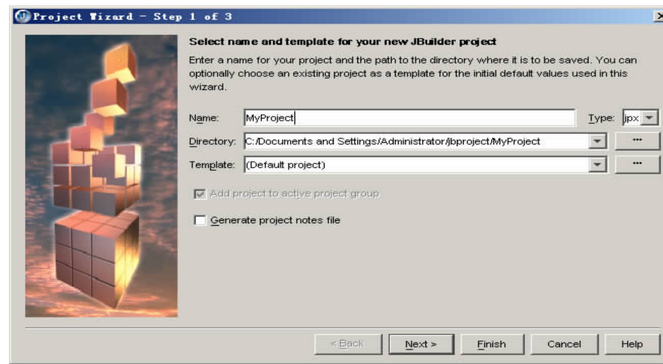


图 1.9　创建工程文件

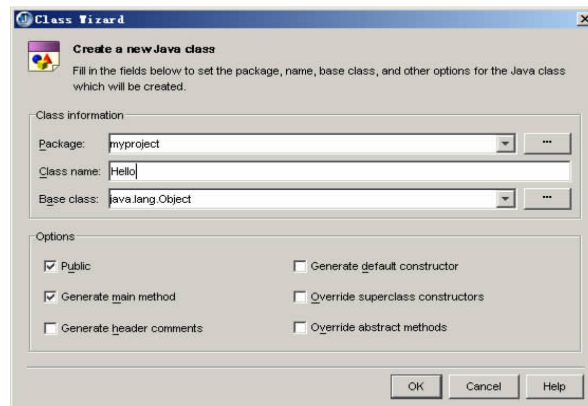


图 1.10　创建类文件

③ 编辑 Java 程序。此时可在 JBuilder 窗口的右边窗口中编辑 Java Application 程序。本例子的 Java 程序如图 1.11 所示。

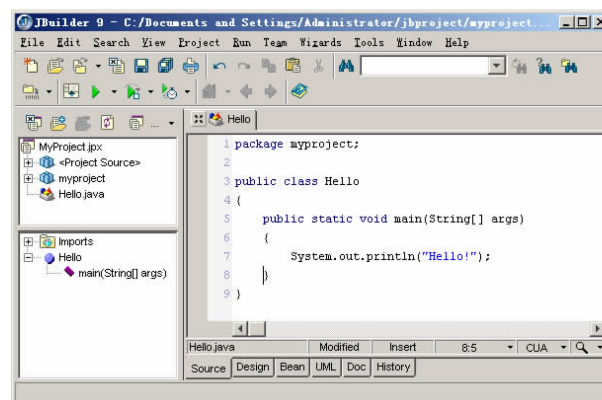


图 1.11　编辑 Java 程序

④ 编译 Java 程序。在 JBuilder 窗口中，选择“Project”菜单下的“Make Project”→“MyProject.jpx”菜单，若程序没有错误则编译完成；若程序有错误则会给出出错信息，可参照出错信息重新编辑和编译 Java 程序，直到编译正确为止。

⑤ 运行 Java 程序。在 JBuilder 窗口中，选择“Run”菜单下的“Run Project”菜单，会

弹出如图 1.12 所示的窗口。

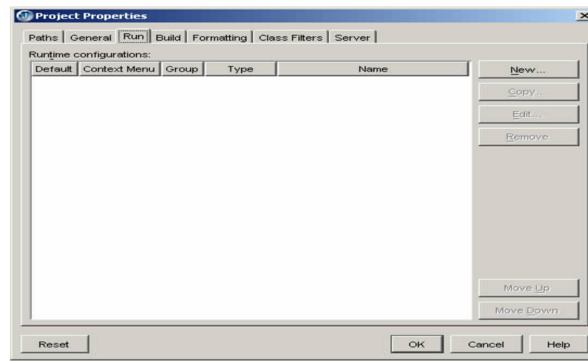


图 1.12　运行窗口

在图 1.12 所示的窗口中，先选择“Run”选项卡，然后单击“New”按钮，会弹出如图 1.13 所示的窗口。在“Search for”栏框中键入主类名 Hello，单击“OK”按钮，即完成了运行。主类名是包含 main()方法的类名。

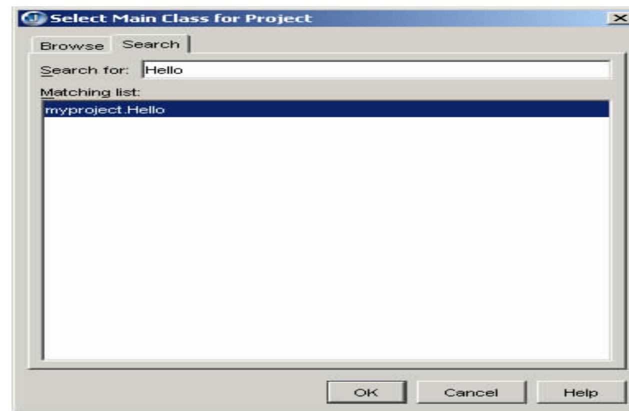


图 1.13　选择主类窗口

图 1.14 是运行结果窗口。此时窗口最下面的子窗口中显示出程序的运行结果：Hello!。

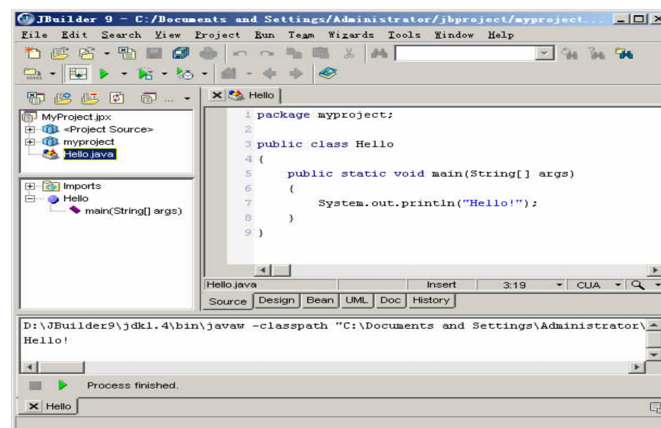


图 1.14　运行结果窗口

在 JBuilder 中，所有 Java Application 程序的开发和运行过程都和上述过程类似。

1.4.2 Java Applet

Java Applet 一般称作 Java 小程序，它是嵌入在网页中、用浏览器加载后才能解释执行的程序。

同样，我们用一个最简单的例子讨论 Java Applet 的编辑、编译、网页文件的编辑和浏览器浏览网页的过程。

【例 1.3】 设计显示“World!”字符串的 Java Applet，并用浏览器浏览网页。

用 JDK 开发环境的编辑、编译、网页文件编辑和浏览器浏览过程如下。

① 编辑 Java Applet 源代码。在记事本窗口中编辑 Applet 源程序。编辑窗口和 Applet 程序代码如图 1.15 所示。设保存该文件的文件路径和文件名为：d:\myjava> HelloApplet.java。

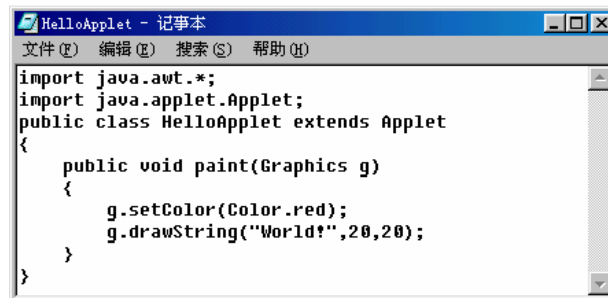


图 1.15 编辑窗口和 Applet 程序代码

注意：文件名必须和类名完全一样，文件名后缀必须为.java。所以，上述文件名为 HelloApplet.java。

关于 Applet 程序的编写方法以及各语句的功能。将在第 7 章详细讨论。

② 编译 Java Applet。输入编译命令：

```
d:\myjava>javac HelloApplet.java
```

编译命令 javac 将对 d:\myjava 文件夹中的 HelloApplet.java 文件进行编译，如果编译正确，将会产生相应的字节码文件 HelloApplet.class；如果给出编译出错信息，则需要重复编辑和编译过程，直到程序没有错误、编译成功为止。

③ 编辑嵌入 Java Applet 的网页。在记事本编辑窗口中键入 HTML 文件，编辑窗口和 HTML 网页代码如图 1.16 所示。此例的网页文件名为 HelloApplet.html。设该文件保存的文件路径和文件名为：d:\myjava> HelloApplet.html。

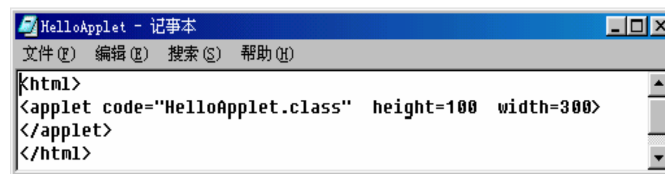


图 1.16 编辑窗口和 HTML 网页代码

注意：网页中嵌入了 HelloApplet。由于 Java 采用的是半编译、半解释的方式，所以这里实际嵌入的是编译后的后缀为.class 的 HelloApplet.class 文件。

④ 用浏览器浏览网页。在浏览器的地址框中选择或键入网页文件名（包括该网页的文件夹名），此例是 d:\MyJava\HelloApplet.html，则浏览器在浏览该网页的同时，也加载并解释执行了嵌入在该网页中的字节码文件 HelloApplet.class。浏览器地址框操作和网页显示结果如图 1.17 所示。下部窗口即为所设计的网页。

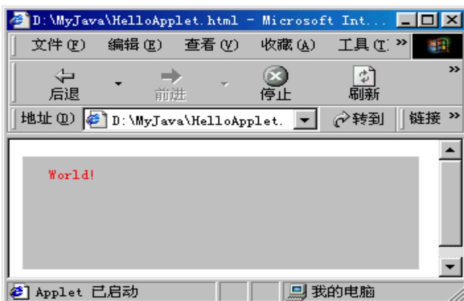


图 1.17　浏览器地址框操作和网页显示结果

Windows 的资源管理器也可作为浏览器使用，在资源管理器的地址框键入上述网页地址，其运行结果和图 1.17 所示的运行结果相同。

习题 1

基本概念题

1.1　概述 Java 语言的起源。

1.2　概述 Java 语言的版本发展过程。目前的最新版本是什么？新版本有什么特点？

1.3　概述 Java 语言的特点。

1.4　简述 Java 语言的运行机制。

1.5　在自己的计算机上安装 JDK 开发运行环境。

1.6　在自己的计算机上用 JDK 开发运行环境完成例 1.1 的编辑、编译、运行过程。

1.7　在自己的计算机上用 JDK 开发运行环境完成例 1.3 的编辑、编译和浏览器浏览网页过程。

第 2 章　Java 语言基础

教学要点

本章内容主要包括 Java 程序设计语言的基本要素（包括标识符、变量、常量、基本数据类型、赋值语句、运算符、表达式、程序注释、字符串等），分支语句、循环语句、数组（包括一维数组、二维数组和不规则的二维数组）。

要求熟练掌握 Java 程序设计语言的基本要素，熟练掌握分支语句、循环语句和数组的程序设计方法。

本章讨论 Java 语言的基础性内容，包括 Java 语言的数据类型、运算符、表达式、流程控制、数组、字符串等。Java 语言是一种面向对象程序设计语言。面向对象程序设计的核心是类和类内成员函数的设计，其中类内成员函数的设计依然需要用传统的面向过程的方法进行。本节讨论的 Java 语言基础，是类内成员函数编码实现的基础。

2.1　标识符

Java 语言的字符使用 **Unicode** 编码标准。Unicode 字符集中的每个字符为 16 位编码。这样，Java 语言的字符不仅可以表示常用的 ASCII 码字符，即数字 0～9、英文字母 A～Z、a～z，以及＋、–、*、／等常用符号，还可以表示如汉字、拉丁语、希腊语等其他语言文字。

标识符是由字母、数字、下划线（_）、美元符（$）组成的，必须以字母、下划线或美元符开头，字符个数有限的字符序列。如 i, count, myStack, GetValue, _length 等都是合法的标识符。Java 语言在定义标识符时，其字母符号区分大小写（或称是大小写敏感的），所以 count, Count 和 COUNT 表示不同的标识符。标识符可以用来命名变量名、常量名、类名等。

Java 语言中有固定含义的标识符称作关键字。用户不允许用关键字定义标识符。Java 语言中的关键字共有 47 个，如表 2.1 所示。

表 2.1　Java 的关键字

abstract	boolean	break	byte	case	catch
char	class	continue	default	do	double
else	extends	false	final	finally	float
for	if	implements	import	instanceof	int
interface	long	native	new	null	package
private	protected	public	return	short	static
super	switch	synchronized	this	throw	throws
transient	true	try	void	while	

这里给出 Java 关键字，既是告诉读者，这些系统已定义的标识符不可以再作为标识符定义，也可以帮助读者在编写程序时参考这些关键字的拼写。

2.2 变量和常量

程序通常要处理数据，处理数据首先要保存数据。程序中要保存的数据都需要系统分配内存空间。变量和常量都需要系统分配内存空间。

2.2.1 变量

变量是一个保存数据的内存区域的名字。变量在使用时必须先定义（或称声明），然后才能用赋值语句（或其他形式）来为其赋值。

变量定义是指示编译器为特定数据类型的数值保存在内存中分配适当的内存空间。这样，在随后的程序中就可以用赋值语句（或其他形式，如在变量定义时直接给出初始值）为该变量赋值。变量的命名必须符合 2.1 节所说的标识符的命名规定。

变量的声明格式为：

<数据类型名> <变量名表>；

随着程序规模的扩大，经常是多人合作完成一个程序的编写。为了防止变量使用的混乱，变量的使用范围是受限的，每个变量只在自己的使用范围内有效。变量的使用范围称作变量的作用域。变量的使用范围是定义该变量的程序块。

【例 2.1】 关于变量作用域的示例。

```
public class Example2_1
{
    public static void main(String args[])
    {
        int i = 10, j;
        j  = 20;
        {
            int k = 100;
            System.out.println("k = " + k );
        }
//      System.out.println("i = " + i + "   j = " + j + "     k = " + k);
        System.out.println("i = " + i + "      j = " + j );
    }
}
```

程序运行结果：

```
k = 100
i = 10      j = 20
```

程序说明：

① 在 main 函数中，定义了 int 类型的变量 i 和 j，变量 i 在定义时给了初始化值 10，变

量 j 随后用赋值语句赋了数值 20。在此程序块中，输出语句输出了变量 i 和 j 的当前数值。在下一层程序块（即下一层的花括号所括的范围）内，又定义了 int 类型的变量 k，由于此变量 k 的作用范围只限于下一层程序块内，所以输出语句中如果包括了变量 k（即如果使用了注释掉的语句），则编译器将指示错误。

在程序设计中，例子中所示的程序块形式并不多见。常见的程序块形式是 if 语句、while 语句、for 语句等后面跟随的一对花括号。依此类推，可以知道，在一个函数中定义的变量只在该函数内有效。

② 语句 System.out.println("k = " + k)中涉及字符串的输出概念，该语句的功能是输出 k = 100 并换行。

2.2.2 常量

在程序设计中，有时需要定义一个变量，并要求该变量在整个程序运行期间保持不变，这种在整个程序运行期间保持不变的变量称作常量。常量也是一种标识符，所以定义常量时也要符合标识符的规定。

定义常量的方法是，在定义变量的语句最前面加上关键字 final。

和变量定义相同，常量定义也是指示编译器为特定数据类型的数值保存在内存中分配适当的内存空间。常量和变量惟一的不同之处是，常量只允许在定义时给出其数值，并不允许在随后的程序中改变其数值。在习惯上，常量名通常为全大写字母。例如，

```
final int MAXSIZE = 100;
```

就定义了常量 MAXSIZE 等于 100。

2.3 基本数据类型

数据类型规定一类数据的数据位长度（或称字符个数），取值范围，以及对该类数据所能进行的操作。

Java 语言中共定义了 8 种基本数据类型，其中 4 种为整型数，2 种为浮点型数，1 种为字符型数，1 种为布尔型数。数据类型不同，所定义的变量占用的内存空间，取值范围，以及对该类数据所能进行的操作也不同。

Java 语言定义的 8 种基本数据类型及相应的关键字如下。

① 整型：byte, short, int, long；

② 浮点型：float, double；

③ 逻辑型：boolean；

④ 字符型：char。

1．整型

整型数包括零和有限范围内的正整数和负整数。由于 4 种不同整型数的数据位不同，所以相应的正整数和负整数的范围也不同。

Java 语言的整型数的类型、数据位和取值范围如表 2.2 所示。

表 2.2 Java 的整型数

类　型	数 据 位	数 值 范 围
字节型 byte	8 bits	−128～127，即-2^7～2^7-1
短整型 short	16 bits	−32 768～32 767，即-2^{15}～$2^{15}-1$
整型 int	32 bits	−2 147 483 648～2 147 483 647，即-2^{31}～$2^{31}-1$
长整型 long	64 bits	−9 223 372 036 854 775 808～9 223 372 036 854 775 807 即-2^{63}～$2^{63}-1$

Java 的整型数可以表示为十进制、八进制或十六进制。

① 十进制：用若干个 0～9 之间的数字表示，并规定首位不能为 0，如 123，−100。

② 八进制：以 0 打头，后跟若干个 0～7 之间的数字。如 0123。

③ 十六进制：以 0x 或 0X 打头，后跟若干个 0～9 之间的数字，以及 a～f 之间的小写字母或 A～F 之间的大写字母，a～f 或 A～F 分别表示数值 10 ～15。如 0X123E。

字面值是在程序中用到的显式数据值，如 123 就是一个字面值。Java 语言规定，所有的整型字面值都隐含为 int 型。若要将一个整数字面值明确地表示为 long 型时，需在后面加英文字母 L 或 l，如 21474836470L 或 12345l。

2．浮点型

浮点型数由整数部分和小数部分组成。浮点型数用来表示实数，有两种表示方式。

① 标准记数法：由整数部分、小数点和小数部分构成，如 123.123 和 123.0。

② 科学记数法：由整数、小数点、小数和指数部分构成，指数部分由一个 e（或 E）后跟带正负号的整数表示。如 123.123 用科学记数法可表示为 1.23123E+2。需要注意的是，科学记数法中，一个浮点型数的整数部分不一定是小数点前的整数。

浮点型数可以表示精度很高的或带有小数部分的数值。当一个变量要保存此类数值时，该变量必须定义为浮点型变量。

Java 语言的浮点型数有 float 和 double 两种。浮点型数的类型、数据位和取值范围如表 2.3 所示。

表 2.3 Java 的浮点型数

类　型	数 据 位	取 值 范 围
单精度浮点 float	32 bits	3.4e−038 ～3.4e+038
双精度浮点 double	64 bits	1.7e−308 ～ 1.7e+308

Java 语言规定，所有浮点型数的字面值都隐含为 double 型。若要将一个浮点型数的字面值明确地表示为 float 型时，需在后面加英文字母 F 或 f，如 123.123F 或 1.23123E+2f。

在定义变量时，对于整型变量或浮点型变量，要认真分析变量可能的取值范围，并选择合适的数据类型来定义变量，以免造成内存空间浪费，或由于超出数值范围或数值精度不够而造成出错。例如，一个变量的取值范围为 1～1000，则该变量应定义成 short 类型。又如，一个变量的可能取值无法准确估计，则应按该变量可能的最大数值、并保留相当的宽裕程度来选择该变量的数据类型。

3．逻辑型

逻辑型用来定义保存逻辑值的变量。逻辑型也称为布尔型。逻辑值只有真（true）和假（false）两个值。所有逻辑运算（如 $a<b$）的运算结果都是逻辑值，如当 $a = 4, b = 6$ 时，逻辑运算 $a<b$ 的运算结果值就是 true。

逻辑型数值主要用于流程控制语句中的条件表达式的取值，如 if, while, for 等语句的条件表达式的取值都是逻辑型数值。

4．字符型

Java 语言中，一个 Unicode 标准下的编码称作一个字符。Unicode 标准用 16 位编码表示一个字符。字符型用来表示字符型变量和字符型字面值。

Java 语言中，字符型字面值用一对单引号括起来，如'a'、'A'、'#'等都是字符型字面值。由于一些控制字符不能在屏幕上直接显示，以及字符串中特殊符号的表示等问题，需要有特殊方法表示这些符号。不能直接显示的控制字符字面值或字符串中特殊符号表示，可以使用转义字符的表示方法。表 2.4 给出了常见的转义字符及其含义。

表 2.4　转 义 字 符

转 义 字 符	含　　义
\n	换行，将光标移至下一行的开始
\t	水平制表，将光标移至下个制表符位置
\r	回车，将光标移至当前行的开始
\\	反斜杠，输出一个反斜杠
\'	单引号，输出一个单引号
\"	双引号，输出一个双引号

如系统标准输出语句 System.out.print()表示输出字符串后不换行，下面的输出语句：

```
System.out.print("Hello!\n");
```

表示输出字符串“Hello!”后换行。

2.4　赋值语句

赋值语句的语法形式是：

<变量> = <表达式>；

赋值语句中，等号（称作赋值号）是赋值语句的标识，<表达式>可以是一个常量，或另一个已赋过值的变量，或是由运算符组成的一个表达式。关于表达式将在 2.5.3 节讨论。这里，可以简单地认为赋值语句的功能，是把一个常量或另一个已赋过值的变量中的数值赋给另一个变量。当一个变量被赋值语句重新赋值时，该变量原先的取值就被冲掉了，即该变量原先的取值就没有了。

Java 语言是一种强类型语言，所谓强类型语言是指对数据类型的匹配要求十分严格。如果一个表达式的数据类型出现不一致等问题，则编译器给出类型不一致的出错信息。关于赋

值语句的类型匹配有以下两种情况。

（1）类型相同

类型相同是指赋值号左端的数据类型和赋值号右端的数据类型完全一致。此种情况可以正确赋值。例如：

```
int smallValue =100；            //类型相同
long bigValue = 100L；           //类型相同
```

（2）类型兼容

类型兼容是指赋值号左端的数据类型比赋值号右端的数据类型长。此时系统会自动将赋值号右端的数据类型转化成和赋值号左端一样的数据类型。这里所说的数据类型长，是指数据类型的位数长，如 long 类型就比 int 类型的位数长。例如：

```
long bigval = 100；              //100 是 int 类型，将自动转化为 100L
double x = 12.345F；             //12.345F 是 float 类型，将自动转化为 12.345
```

若赋值号右端的数据类型比赋值号左端的数据类型长，则类型不兼容。此时系统在编译时会产生“可能存在精度损失”的编译错误。例如：

```
int smallValuel = 100L；         // 100L 是 long 类型，不兼容
float x = 12.345；               //12.345 是 double 类型，不兼容
```

当出现类型不兼容错误时，有两种解决方法：

① 重新定义赋值号左端变量的数据类型，使之变成满足要求的较长的数据类型；

② 用强制类型转换方法把赋值号右端的数据类型转换成和赋值号左端相同的数据类型。注意，强制类型转换可能丢失数据或损失数据的精度。强制类型转换的格式为：

（<目标类型>）（<表达式>）

例如：

```
int i;
long k = 100L;
i = (int) k;                     //把变量 k 中的数值强制转换成 int 类型后赋给变量 i
```

2.5 运算符和表达式

在 Java 语言之前，C 语言和 C++语言是最为广泛使用的高级程序设计语言。为使 Java 语言尽快被程序设计人员接受，Java 语言设计者采用了把基本关键字和基本语句设计成和 C/C++语言相同形式的方法。因此，对于已经掌握 C/C++语言的人来说，Java 语言在基本关键字、运算符、表达式、赋值语句、流程控制语句等方面，是和 C/C++语言基本相同的。

2.5.1 运算符及其分类

Java 的运算符可分为四类：算术运算符、关系运算符、逻辑运算符和位运算符。

1．算术运算符

Java 的算术运算符分为一元运算符和二元运算符。一元运算符只有一个操作数；二元运

算符有两个操作数，运算符位于两个操作数之间。算术运算符的操作数必须是数值类型。

（1）一元运算符

一元运算符有四个，分别为：正（+）、负（–）、加 1（++）和减 1（– –）。

加 1、减 1 运算符只允许用于数值类型的变量，不允许用于表达式中。加 1、减 1 运算符既可放在变量之前（如＋＋*i*），也可放在变量之后（如 *i*＋＋），两者的差别是：如果放在变量之前，则变量值先加 1 或减 1，然后进行其他相应的操作（主要是赋值操作）；如果放在变量之后，则先进行其他相应的操作，然后再将变量值加 1 或减 1。例如，

```
int i=6，j，k，m，n；
j = +i；          //取原值，即 j=6
k = –i；          //取负值，即 k=–6
m = i++；         //先 m=i,再 i=i+1，即 m=6，i=7
m = ++i；         //先 i=i+1,再 m=i，即 i=7，m=7
n = j––；         //先 n=j,再 j=j–1，即 n=6，j=5
n = ––j；         //先 j=j–1,再 n=j，即 j=5，n=5
```

在书写时还要注意的是，一元运算符与其前后的操作数之间不允许有空格，否则编译时会出错。

（2）二元运算符

二元运算符有五个，分别为：加（+）、减（–）、乘（*）、除（／）、取余（%）。其中，+、–、*、／完成加、减、乘、除四则运算，%是求两个操作数相除后的余数。设有 a%b，a%b 的计算公式为：

```
a % b = a – (a / b) * b
```

取余运算符既可用于两个操作数都是整数的情况，也可用于两个操作数都是浮点数（或一个操作数是浮点数）的情况。当两个操作数都是浮点数时，如 7.6 % 2.9，计算结果为：7.6 – 2 * 2.9 = 1.8。

当两个操作数都是 int 类型数时，a%b 的计算公式为：

```
a % b = a – (int) (a / b) * b
```

当两个操作数都是 long 类型（或其他整数类型）数时，a%b 的计算公式可以类推。

当参加二元运算的两个操作数的数据类型不同时，所得结果的数据类型与精度较高（或位数更长）的那种数据类型一致。例如，

```
7 / 3             //整除，运算结果为 2
7.0 / 3           //除法，运算结果为 2.33333,即结果与精度较高的类型一致
7 % 3             //取余，运算结果为 1
7.0 % 3           //取余，运算结果为 1.0
–7 % 3            //取余，运算结果为–1，即运算结果的符号与左操作数相同
7 % –3            //取余，运算结果为 1，即运算结果的符号与左操作数相同
```

2．关系运算符

关系运算符用于比较两个数值之间的大小，其运算结果为一个逻辑类型的数值。关系运

算符有六个，分别是：等于（ == ）、不等于（ != ）、大于（ > ）、大于等于（ >= ）、小于（ < ）、小于等于（ <= ）。例如：

```
9 <= 8            //运算结果为 false
9.9 >= 8.8        //运算结果为 true
'A' < 'a'         //运算结果为 true，因字符'A'的 Unicode 编码值小于字符'a'的
```

要说明的是，对于大于等于（或小于等于）关系运算符来说，只有大于和等于两种关系运算都不成立时其结果值才为 false，只要有一种（大于或等于）关系运算成立其结果值即为 true。例如，对于 9 <= 8，9 既不小于 8 也不等于 8，所以 9 <= 8 的运算结果为 false。对于 9 >= 9，因 9 等于 9，所以 9 >= 9 的运算结果为 true。

3．逻辑运算符

逻辑运算符要求操作数的数据类型为逻辑型，其运算结果也是逻辑型值。逻辑运算符有：逻辑与（&&）、逻辑或（||）、逻辑非（!）、逻辑异或（^）、逻辑与（&）、逻辑或（|）。

真值表是表示逻辑运算功能的一种直观方法，其具体方法是把逻辑运算的所有可能值用表格形式全部罗列出来。Java 语言逻辑运算符的真值表如表 2.5 所示。

表 2.5　逻辑运算符的真值表

A	B	A &&B	A || B	! A	A ^ B	A & B	A | B
false	false	false	false	true	false	false	false
true	false	false	true	false	true	false	true
false	true	false	true	true	true	false	true
true	true	true	true	false	false	true	true

表 2.5 中，前两列是参与逻辑运算的两个逻辑变量，共有四种可能，所以表 2.5 共有四行。后六列分别是六个逻辑运算符在逻辑变量 A 和逻辑变量 B 取不同数值时的运算结果值。

要说明的是，两种逻辑与（&&和&）的运算规则基本相同，两种逻辑或（||和|）的运算规则也基本相同。其区别是：&和 | 运算是把逻辑表达式全部计算完，而&&和 || 运算具有短路计算功能。所谓**短路计算**，是指系统从左至右进行逻辑表达式的计算，一旦出现计算结果已经确定的情况，则计算过程即被终止。对于&&运算来说，只要运算符左端的值为 false，则无论运算符右端的值为 true 或为 false，其最终结果都为 false。所以，系统一旦判断出&&运算符左端的值为 false，则系统将终止其后的计算过程；对于 || 运算来说，只要运算符左端的值为 true，则无论运算符右端的值为 true 或为 false，其最终结果都为 true。所以，系统一旦判断出 || 运算符左端的值为 true，则系统将终止其后的计算过程。例如，有如下逻辑表达式：

```
(i>=1) && (i<=100)
```

此时，若 i 等于 0，则系统判断出 i>=1 的计算结果为 false 后，马上得出该逻辑表达式的最终计算结果为 false，因此，系统不继续判断 i<=100 的值。短路计算功能可以提高程序的运行速度。

作者建议：在程序设计时使用&&和 || 运算符，不使用&和 | 运算符。

用逻辑与（&&）、逻辑或（||）和逻辑非（!）可以组合出各种可能的逻辑表达式。逻辑表达式主要用在 if, while 等语句的条件组合上。例如，

```
int i = 1;
while(i>=1) && (i<=100) i++;                    //循环过程
```

上述程序段的循环过程将 i++语句循环执行 100 次。

4．位运算符

位运算是以二进制位为单位进行的运算，其操作数和运算结果都是整型值。位运算符共有七个，分别是：位与（&）、位或（|）、位非（～）、位异或（^）、右移（>>）、左移（<<）、0 填充的右移（>>>）。位运算的位与（&）、位或（|）、位非（～）、位异或（^）与逻辑运算的相应操作的真值表完全相同，其差别只是位运算相应的操作数和运算结果都是二进制整数，而逻辑运算相应的操作数和运算结果都是逻辑值。表 2.6 是位运算示例。

表 2.6　位运算示例

运　算　符	名　　称	示　　例	说　　明
&	位与	x & y	把 x 和 y 按位求与
|	位或	x | y	把 x 和 y 按位求或
～	位非	～x	把 x 按位求非
^	位异或	x^y	把 x 和 y 按位求异或
>>	右移	x>>y	把 x 的各位右移 y 位
<<	左移	x<<y	把 x 的各位左移 y 位
>>>	右移	x>>>y	把 x 的各位右移 y 位，左边填 0

现举例说明如下。

① 有如下程序段：

```
int x = 64;                //x 等于二进制数的 01000000
int y = 70;                //y 等于二进制数的 01000110
int z = x&y                //z 等于二进制数的 01000000
```

即运算结果 z 等于二进制数 01000000。位或、位非、位异或的运算方法类同。

② 右移是将一个二进制数按指定移动的位数向右移位，移掉的被丢弃，左边移进的部分或者补 0（当该数为正时），或者补 1（当该数为负时）。这是因为，整数在机器内部采用补码表示法，正数的符号位为 0，负数的符号位为 1。例如，对于如下程序段：

```
int x = 70;                //x 等于二进制数的 01000110
int y = 2;
int z = x>>y               //z 等于二进制数的 00010001
```

即运算结果 z 等于二进制数 00010001，即 z 等于十进制数 17。

对于如下程序段：

```
int x = −70;               //x 等于二进制数的 11000110
int y = 2;
int z = x>>y               //z 等于二进制数的 11101110
```

即运算结果为 z 等于二进制数 11101110，即 z 等于十进制数−18。要想透彻理解右移和左

移操作，需要掌握整数机器数的补码表示法。

③ 0 填充的右移（>>>）：不论被移动数是正数还是负数，左边移进的部分一律补 0。

5．其他运算符

（1）赋值运算符与其他运算符的简捷使用方式

赋值运算符可以与二元算术运算符、逻辑运算符和位运算符组合成简捷运算符，从而可以简化一些常用表达式的书写。表 2.7 是这些简捷运算符及其功能说明。

表 2.7　赋值运算符与其他运算符的简捷使用方式

运 算 符	用 法	等 价 于	说 明
+=	s+=i	s=s+i	s, i 是数值型
−=	s−=i	s=s−i	s, i 是数值型
*=	s*=i	s=s*i	s, i 是数值型
/=	s/=i	s=s/I	s, i 是数值型
%=	s%=i	s=s%i	s, i 是数值型
&=	a&=b	a=a&b	a, b 是逻辑型或整型
|=	a | =b	a=a | b	a, b 是逻辑型或整型
^=	a^=b	a=a^b	a, b 是逻辑型或整型
<<=	s<<=i	s=s<<i	s, i 是整型
>>=	s>>=i	s=s>>i	s, i 是整型
>>>=	s>>>=i	s=s>>>i	s, i 是整型

（2）方括号[ ]和圆括号（）运算符

方括号[ ]是数组运算符，方括号[ ]中的数值是数组的下标，整个表达式就代表数组中该下标所在位置的元素值。

圆括号（）运算符用于改变表达式中运算符的优先级。

（3）字符串加（+）运算符

当操作数是字符串时，加（+）运算符用来合并两个字符串；当加（+）运算符的一边是字符串，另一边是数值时，机器自动将数值转换为字符串，这种情况在输出语句中很常见。如对于如下程序段：

```
int max = 100;
System.out.println（"max = "+max）;
```

计算机屏幕的输出结果为：max = 100，即此时是把变量 max 中的整数值 100 转换成字符串 100 输出的。

（4）条件运算符（？ :）

条件运算符（？ :）的语法形式为：

〈表达式 1〉？〈表达式 2〉:〈表达式 3〉

条件运算符的运算方法是：先计算〈表达式 1〉的值，当〈表达式 1〉的值为 true 时，则将〈表达式 2〉的值作为整个表达式的值；当〈表达式 1〉的值为 false 时，则将〈表达式 3〉的值作为整个表达式的值。如：

```
int a=1，b=2，max；
max = a>b？a：b；                    //max 等于 2
```

（5）强制类型转换符

强制类型转换符能将一个表达式的类型强制转换为某一指定数据类型，其语法形式为：

(〈类型〉)〈表达式〉

（6）对象运算符 instanceof

对象运算符 instanceof 用来测试一个指定对象是否是指定类（或它的子类）的实例，若是则返回 true，否则返回 false。

（7）点运算符（·）

点运算符（·）的功能有两个：一是引用类中成员，二是指示包的层次等级。

2.5.2　运算符的优先级

表 2.8 按优先级从高到低的次序列出 Java 语言中的所有运算符，表中结合性一列中的"左⌦右"表示其运算次序为从左向右，"右⌦左"表示其运算次序为从右向左。

表 2.8　运算符的优先级

优　先　级	运　算　符	结　合　性
1	.　[ ]　( )　;　,	
2	++　－－　+=　～　!　+（一元）－（一元）	右⌦左
3	*　/　%	左⌦右
4	+（二元）　－（二元）	左⌦右
5	<<　>>　>>>	左⌦右
6	<　>　<=　>=　instanceof	左⌦右
7	= =　!=	左⌦右
8	&	左⌦右
9	^	左⌦右
10	|	左⌦右
11	&&	左⌦右
12	||	左⌦右
13	?：	右⌦左
14	=　*=　/=　%=　+=　－=　<<=　>>=　>>>=　&=　^=　|=	右⌦左

2.5.3　表达式

用运算符和圆括号把运算对象连接起来的、符合 Java 语言语法规则的式子称作**表达式**。运算符是算术运算符的表达式称作**算术表达式**，运算符是关系运算符的表达式称作**关系表达式**，运算符是逻辑运算符的表达式称作**逻辑表达式**，运算符是位运算符的表达式称作**位表达式**。算术表达式中的运算对象必须是数值量，关系表达式中的运算对象必须是数值量，逻辑表达式中的运算对象必须是逻辑量，位表达式中的运算对象必须是整数数值量。

运算符都有优先级，表达式按照运算符的优先级逐个进行计算，最后求得整个表达式的值。运算符中圆括号的优先级最高，圆括号可以多层嵌套。在同一级括号内（或一个括号也

没有时)，表达式按照运算符优先级高的先运算；同一优先级的运算符，按照运算符的结合性（即从左向右还是从右向左）的次序进行运算。大多数运算符的结合性是从左向右的，少数运算符的结合性（如赋值、条件运算等）是从右向左的。

由于运算对象是有数据类型的，所以最终得到的计算结果也是有数据类型的，表达式结果的数据类型不一定和运算对象相同，它还取决于表达式的运算符。如：

```
int i=5;
(i<0)&(i>100)            //结果为 boolean 型
(i–2)*8+5                //结果为 int 型
“Abcde”+“12345”          //结果为 String 型
```

Java 表达式既可以单独组成语句，也可出现在循环语句、条件语句的条件部分，还可以出现在函数的实际参数调用等场合。

实际上，赋值语句也是一个表达式，其运算方法是先计算出赋值运算符右边的表达式的值（因赋值运算符的优先级最低)，然后把该值赋给赋值运算符左边的变量（因赋值运算符的结合性是先右后左的)。

最后，给出一个例子，用来说明上面讨论过的 Java 语言基本要素。

【例 2.2】　求 $4x^2+5x+1=0$ 方程的根。已知 $b^2-4ac>0$。

```
public class Exam2_2
{
    public static void main(String args[])
    {
        double a = 4, b = 5, c = 1, disc, x1, x2, p, q;
        disc = b * b – 4 * a * c;
        p = – b / (2 * a);
        q = Math.sqrt(disc) / (2 * a);
        x1 = p + q;
        x2 = p – q;
        System.out.println("x1 = " + x1 + "        x2 = " + x2);
    }
}
```

其中，表达式 Math.sqrt( )为调用 Math 类的 sqrt( )方法，该表达式完成某个数的开平方。此例还说明，表达式还可以是某个类的方法调用（当然，数据类型要符合要求)。

程序运行结果为：

```
x1 = –0.25        x2 = –1.0
```

2.6　流程控制语句

流程控制语句用来控制程序的执行流程。流程控制语句有条件选择语句，循环语句和转移语句三种。

任何程序的结构都是由三种基本结构组成的。这三种基本结构是顺序结构、分支结构

和循环结构。其中，例 2.2 的程序结构就是最典型的顺序结构。在 Java 程序中，一个以分号（;）结束的符号串称为一条语句。在顺序结构的程序或程序段中，程序是按语句次序顺序执行的。

在分支结构或循环结构的程序或程序段中，程序不是按语句次序顺序执行的，而是按分支结构语句或循环结构语句所规定的执行流程来执行的。为了满足实际应用问题的设计需要，分支结构和循环结构有各种各样的变形。另外，实际的程序设计中，经常需要转移语句与分支语句或循环语句结合使用。

Java 语言的流程控制语句有如下三种。

① 分支语句：if，switch；

② 循环语句：for ，while，do-while；

③ 转移语句：break，continue。

另外，函数或类的成员函数在运行结束后，需要返回原调用处，并可能需要带回函数或成员函数的返回值，这种流程控制用 return 语句来实现。

2.6.1　分支语句

分支结构是指程序按照当前的分支条件控制程序流程的执行。Java 语言有两种分支结构的语句：if 语句和 switch 语句。if 语句实现二路分支，switch 语句实现多路分支。

1．if 语句

（1）基本的 if 语句

if 语句的语法形式为：

```
if  (〈逻辑表达式〉)
〈语句组 1〉;
[else  〈语句组 2〉; ]
```

其中，逻辑表达式的计算值是一个只可能是 true 或 false 的逻辑值。if 语句的含义是：当〈逻辑表达式〉的值为 true 时，执行 if 后面的〈语句组 1〉；当〈逻辑表达式〉的值为 false 时，执行 else 后面的〈语句组 2〉。当执行的语句组中的语句多于一条时，语句组必须用一对花括号“{}”括起来。在 Java 语言的语法中，一对方括号“[ ]”括起来的部分，表示该部分（除数组定义外）是任选的。因此，else 语句是任选的，当没有 else 语句时，若〈逻辑表达式〉的值为 false，就执行 if，语句后边的语句。

【例 2.3】　找出 a, b 两个数中的较大者并输出。

方法一：用 if-else 结构实现。

```
public class Exam2_3
{
    public static void main(String args[])
    {
        int a = 5, b = 6, max;
        if( a > b )
        {
```

```
                max = a;
                System.out.println("max = " + max);
            }
            else
            {
                max = b;
                System.out.println("max = " + max);
            }
        }
    }
```

方法二：用没有 else 的 if 结构实现。

```
    public class Exam2_3_2
    {
        public static void main(String args[])
        {
            int a = 5, b = 6, max;
            max = a;
            if( b > a )
                max = b;
            System.out.println("max = " + max);
        }
    }
```

例 2.3 中，if 语句的逻辑表达式中只有一个条件。当问题复杂时，if 语句的逻辑表达式中的条件可能多于一个，这称为复合条件。复合条件需要用逻辑运算符——逻辑与（&&）、逻辑或（||）、逻辑非（!）、逻辑异或（^）等逻辑运算符来组合条件。例如下面的 if 语句条件为，判断 *n* 是否为一个 1～100 区间的数：

```
    if (n >= 1 && n <= 100)
```

如果 *n* 既大于等于 1 又小于等于 100，则条件为 true，否则条件为 false。

（2）if 语句嵌套

如果 if 语句中又包括另一个 if 语句，则称为 if 语句嵌套。在两个嵌套的 if 语句中，如果有一个 if 语句省略了 else 语句，会产生二义性。如：

```
    if (n >= 1)
    if (n <= 100)     ……
    else  ……
```

此时，else 是和哪个 if 语句匹配容易产生混淆。为此 Java 语言规定：else 总是与最近的一个 if 语句匹配。所以此处 else 应理解为是与第二个 if 语句匹配（注意：书写的缩进格式与匹配无关）。如果要求 else 是与第一个 if 语句匹配，则程序应写为：

```
    if (n >= 1)
    {
```

```
        if (n <= 100)           ……
    }
    else    ……
```

一个 if 语句可以有两个分支，利用 if 语句嵌套可以组合出多于两个的分支。

【例 2.4】 找出三个整数中的最大数并输出。

方法一：用 if 语句嵌套方法。

```
public class Exam2_4
{
    public static void main(String args[ ])
    {
        int a = 3, b = 1, c = 2, max;

        if(a < b)
            if (b < c)
                max = c;
            else
                max = b;
        else
            if (a < c)
                max = c;
            else
                max = a;
       System.out.println("max = " + max);
    }
}
```

程序运行结果：

```
max = 3
```

方法二：用条件的逻辑组合方法。

```
public class Exam2_4_2
{
    public static void main(String args[])
    {
        int a = 3, b = 1, c = 2, max;
        if(a < b && b < c)
            max = c;
        else if(a < b && c < b)
            max = b;
        else if (a > b && a < c)
            max = c;
        else                                          //即(a > b && a > c)
```

```
            max = a;
        System.out.println("max = " + max);
    }
}
```

说明：条件两两组合，所以条件的逻辑组合共有四种情况。此问题可不考虑两个数相等的情况。

2．switch 语句

if 语句可以解决两个分支的程序流程控制问题。当程序的流程多于两个分支时，可以用 if 语句嵌套方法或条件组合方法控制程序的流程。但在有些应用问题中，用 if 语句嵌套方法或条件组合方法控制程序流程的方法不够简洁。为此，Java 语言还设计了专门的多个分支的流程控制语句——switch 语句。

switch 语句的语法形式为：

```
switch (〈表达式〉)
{
  case〈常量 1〉:〈语句组 1〉;
              [break；]
  case〈常量 2〉:〈语句组 2〉;
              [break；]
  ……；
  [default：〈语句组〉]
}
```

其中，switch, case, default 是关键字，default 语句是可选的。

switch 语句的语义是：将〈表达式〉的值按照从上至下的顺序与 case 语句中给出的常量值进行比较，当表达式的值与某个 case 语句中的常量值相等时，就执行 case 后的相应语句序列；若没有一个常量值与表达式的值相等，则执行 default 语句。如果没有 default 语句，并且表达式与所有 case 后的常量都不相等时，则不做任何操作。

switch 表达式和 case 常量值的类型可以是 byte, short, int, long 和 char，但不能为 boolean，并且要求两者的数据类型必须一致。

3．switch 语句中的 break 语句

switch 语句本身并不能保证执行完一组 case 后的语句或语句组后，跳过随后的 case 判断。通常情况下，此时需要用 break 语句来跳过随后的 case 语句。

break 语句的语法形式是：

```
break；
```

switch 语句中的 break 语句的语义是：跳过 break 语句所在位置后所有的 case 语句，即结束 switch 语句的执行。

【例 2.5】 把数值表示的星期转换成相应的英文表示并显示。

```
public class Exam2_5
```

```
{
    public static void main(String args[])
    {
        int week = 5;
        System.out.print("week = " + week + "      ");
        switch (week)
        {
            case 0: System.out.println("Sunday");      break;
            case 1: System.out.println("Monday");      break;
            case 2: System.out.println("Tuesday");    break;
            case 3: System.out.println("Wednesday");break;
            case 4: System.out.println("Thursday"); break;
            case 5: System.out.println("Friday");      break;
            case 6: System.out.println("Saturday"); break;
            default: System.out.println("Data Error!");
        }
    }
}
```

程序运行结果：

```
week = 5      Friday
```

2.6.2 循环语句

循环结构是指程序按照当前的循环条件控制程序流程的执行。Java 语言有三种循环结构的语句：for 语句、while 语句和 do-while 语句。这三种循环语句虽然控制循环的方式不同，但实现循环的功能相同。换句话说，对于任何一个循环问题，这三种循环语句都可以实现。但是，不同的循环问题，使用这三种循环语句的简便程度不同。因此，一个好的程序设计者，应该学会针对不同的循环问题，选择最简便的循环语句。

1．for 语句

for 语句的语法形式为：

```
for ([〈表达式 1〉]；[〈表达式 2〉]；[〈表达式 3〉])
    〈循环体〉；
```

for 语句的语义是：首先计算〈表达式 1〉的值，然后计算〈表达式 2〉的值，如果〈表达式 2〉值为 true 时，执行一次〈循环体〉，再计算〈表达式 3〉的值，并进行下一次循环过程；当〈表达式 2〉的值为 false 时，循环过程结束。

在 for 语句中，〈表达式 1〉是给循环变量赋初值，因此，〈表达式 1〉通常是一个赋值表达式；〈表达式 2〉给出循环结束条件，因此，〈表达式 2〉必须是一个关系表达式或逻辑表达式；〈表达式 3〉改变循环变量，因此，〈表达式 3〉必须是一个赋值表达式（或类似 i++形式的改变变量 i 的数值的语句）；〈循环体〉可以是一条语句，也可以是多条语句，当为多条语

句时，必须用一对花括号“{}”括起来。

还要说明的是，无论〈表达式 1〉，〈表达式 2〉，还是〈表达式 3〉，都可以任选。

【例 2.6】　求 1 到 10 的累加和。

for 语句最适合于循环次数已知的循环结构。此问题的循环次数已知，因此，此问题适合于用 for 语句实现。

```
public class Exam2_6
{
    public static void main(String args[])
    {
        int i, n = 10, sum = 0;

        for(i = 1; i <= n; i++)
            sum = sum + i;
        System.out.println("Sum = " + sum);
    }
}
```

程序运行结果为：

```
Sum = 55
```

上述 for 语句循环过程也可以设计成递减形式的，此时的 for 语句为：

```
for(i = n; i >= 1; i—)
    sum = sum + i;
```

2．while 语句

while 语句的语法形式为：

```
while (〈逻辑表达式〉)
        〈循环体〉;
```

while 语句的语义是：如果〈逻辑表达式〉的计算结果为 true，则执行循环体；如果〈逻辑表达式〉的计算结果为 false，则结束 while 语句的执行。同样，当〈循环体〉中的语句多于一条时，需要用一对花括号“{}”括起来。

例如，用 while 语句实现求 1～10 累加和的程序段如下：

```
int i = 1，n = 10，sum = 0；
while (i <= n)
{
    sum = sum + i；
    i++；
}
```

和 for 语句相比，while 语句循环中的循环变量赋初值（i = 1）、循环过程结束判断（i <= n）和循环变量修改（i++）三个部分都有，只是放在了不同的地方。

在 while 语句的循环结构中，初学者最容易犯的一个错误是，忘记在循环体中修改循环变量的值，此时循环过程将永不结束，俗称为“死循环”。如上面的程序段中如果没有 i++语句，则循环过程将永不结束。

死循环的另一种情况是，循环结束条件永远无法满足。例如：

```
int i = 1，n = 10，sum = 0；
while (i >= 0)
{
    sum = sum + i；
    i++；
}
```

上述程序段的循环体中虽然有 i++语句改变循环变量 *i* 的值,但 *i* 值的变化永远都满足 *i* >= 0，即循环结束条件永远不会满足，因此造成死循环。

while 语句适合于循环次数不确定的情况。如上所示，虽然 while 语句也可以用来构造循环次数已知的循环过程，但是很显然，此种情况用 for 语句构造循环过程更简单、更明了一些。

【例 2.7】 求两个不为 0 的正整数的最大公约数。

对于求两个不为 0 的正整数的最大公约数问题，欧几里德提出了辗转相除算法，其算法思想是：

① 令 *m* 为两个整数中的较大者，*n* 为两个整数中的较小者；

② 用 *m* 除以 *n*，令 *r* 为 *m* 除以 *n* 的余数；

③ 若 *r* 不等于 0，则令 *m* 等于 *n*，*n* 等于 *r*，返回步骤②继续；若 *r* 等于 0，则 *n* 中的数值就是两个整数的最大公约数。

从上述算法思想可知，该算法是一个循环过程，并且该循环过程的循环次数事先无法知道，因此，该问题适合于用 while 语句构造循环过程。

```
public class Exam2_7
{
    public static void main(String args[])
    {
        int m = 48, n = 32, r = 0, temp;

        if (m <= 0 || n <= 0)
        {
            System.out.println("数据错误！");
            return;
        }

        if(m < n)                                    //保证 m >= n
        {
            temp = m;
            m = n;
```

```
            n = temp;
        }

        r = m % n;                          //循环初始化
        while(r != 0)                       //循环条件判断
        {
            m = n;
            n = r;
            r = m % n;                      //循环变量修改
        }

        System.out.print("最大公约数 = " + n);
    }
}
```

程序运行结果：

```
最大公约数 = 16
```

当然，上面程序中的循环过程也可以用 for 语句构造。用 for 语句构造循环过程相应的程序段如下：

```
for(r = m % n; r != 0; r = m % n)          //for 循环
{
    m = n;
    n = r;
}
```

for 语句的循环过程中，也必须有循环初始化（r = m % n）、循环条件判断（r != 0）和循环变量修改（r = m % n），只是所处的位置不同而已。显然，在循环次数未知时，用 for 语句构造的循环过程不及用 while 语句构造的循环过程清晰。

3．do-while 语句

do-while 语句的语法形式为：

```
do
{
    〈语句组〉;
}  while (〈逻辑表达式〉) ;
```

do-while 语句的语义是：首先执行语句组（或称循环体），然后计算〈逻辑表达式〉的值，当〈逻辑表达式〉的值为 true 时，执行一次循环体；当〈逻辑表达式〉的值为 false 时，结束循环。

从语义看，do-while 语句和 while 语句的惟一差别是：do-while 语句至少执行一次循环体（因其结束条件判断在后面进行）；而对于 while 语句来说，当循环条件一开始就不满足时，循环体将一次也不执行。

例 2.7 求最大公约数程序的循环部分也可以用 do-while 语句构造。用 do-while 语句的相应程序段如下：

```
r = m % n;                          //循环初始化
if( r != 0)                         //判断初始时 r 是否为 0

    do
    {
        m = n;
        n = r;
        r = m % n;                  //循环变量修改
} while(r != 0);                    //循环结束条件判断
```

由于语句 r = m % n 的计算结果可能使 r 等于 0，而 do 语句在第一次执行循环体时，并不做任何判断，这将可能造成因 r 等于 0 使 n 等于 0，因 n 等于 0 使语句 r = m % n 的运算出现除数等于 0 的错误。因此要在 do 语句之前判断 r 是否等于 0。只有在 r 不等于 0 时才执行 do 语句；在 r 等于 0 时最大公约数已经得到。

显然，对于求最大公约数问题，用 while 语句比用 do-while 语句简单一些。但在有些情况下，用 do-while 语句构造循环过程比用 while 语句简单一些。

4．多重循环

如果循环语句的循环体内又有循环语句，则构成多重循环结构。多重循环结构中的循环语句，可以是前面讨论过的 for 语句、while 语句或 do-while 语句中的任何一种。

【例 2.8】　输出九九乘法表。

程序如下：

```
public class Exam2_8
{
    public static void main(String args[])
    {
        int i,j,n = 9;
        for(i = 1; i <= n; i++)                 //外层循环
        {
            for(j = 1; j <= i; j++)             //内层循环
                System.out.print("    " + i*j); //输出
            System.out.println();               //每行结束时换行
        }
    }
}
```

说明：这个问题中，两重循环的循环次数都已知，因此两重循环都用 for 语句构造循环过程。外层 for 语句的循环变量是 i，控制总共打印多少行，内层 for 语句的循环变量是 j，控制每行显示多少列。

程序运行结果为：

```
1
2   4
3   6   9
4   8   12   16
5   10   15   20   25
6   12   18   24   30   36
7   14   21   28   35   42   49
8   16   24   32   40   48   56   64
9   18   27   36   45   54   63   72   81
```

当应用问题复杂时，程序一般需要有多重循环结构，此时最重要的是要把程序设计的思路梳理清楚，而其中的每一重循环结构都可以按单重循环结构设计。

2.6.3　break 语句和 continue 语句

break 语句通常是和 switch 语句或循环语句配合使用的，continue 语句通常是和循环语句配合使用的。

1．break 语句

break 语句的语法形式是：

```
break；
```

在上节讨论 switch 语句时，已介绍了 switch 语句中的 break 语句使用方法。这里主要介绍循环语句中的 break 语句使用方法。在循环语句中，break 语句的功能是跳出循环体。特别需要说明的是，当 break 语句位于多重循环语句的内层时，break 语句只能跳出它当前所处的那层循环体。

【例 2.9】　循环语句中 break 语句使用方法示例。

```
public class Exam2_9
{
    public static void main(String args[])
    {
        int i, j, k;

        for(i = 1; i <= 6; i++)                         //第 1 层循环
        {
            for(j = 1; j <= 6; j++)                     //第 2 层循环
             {
                for(k = 1; k <= 6; k++)                 //第 3 层循环
                {
                    System.out.println("k = " + k + "      ");
                    if(k == 2) break;                   //跳出第 3 层循环
                }
```

```
                System.out.println("j = " + j + "      ");
                if(j == 2) break;                       //跳出第 2 层循环
            }
            System.out.println("i = " + i + "      ");
            if(i == 2) break;                           //跳出第 1 层循环
        }
    }
}
```

由于 break 语句的作用，此程序的第 3 层的输出语句（输出 *k*）只执行了 8 次，第 2 层的输出语句（输出 *j*）只执行了 4 次，第 1 层的输出语句（输出 *i*）只执行了 2 次。

2．continue 语句

continue 语句的语法形式为：

continue；

continue 语句主要用于循环语句中，continue 语句的语义是：若循环体中遇到 continue 语句，则本次循环体的后续语句被忽略，回到循环条件判断处，判断是否执行下一次循环。换句话说，continue 语句仅跳过当前层中循环体的剩余语句。

【例 2.10】　continue 语句使用方法示例。

```
public class Exam2_10
{
    public static void main(String args[])
    {
        int i, j;

        for(i = 1; i <= 6; i++)
        {
            for(j = 1; j <= 6; j++)
            {
                if(j >= 3) continue;              //continue 语句
                System.out.println("j = " + j + "      ");
            }
        }
    }
}
```

由于 continue 语句的作用，第 2 层的输出语句只执行了 12 次。

2.6.4　return 语句

return 语句的语法形式为：

return [<返回值>]；

return 语句的语义是：使函数返回原调用处，且带回<返回值>。如果函数为 void 类型，则 return 语句后没有<返回值>；如果函数为非 void 类型，则 return 语句后需要有<返回值>，并且<返回值>的类型必须和函数的类型一致。

当 return 语句不带<返回值>，并且位于函数的最后时，return 语句可以省略。

return 语句的具体使用方法可参见随后各章的程序设计举例。

2.7 程序注释

Java 语言允许在程序中添加注释，以增加程序的可读性。前面的例子中已经多次使用了程序注释。注释主要是给用户阅读的，所以系统不会对注释的内容进行编译。

概括起来，Java 语言有如下三种形式的注释。

（1）单行注释

单行注释以“//”开头，至该行行尾。格式为：

```
// 注释内容
```

（2）多行注释

多行注释以“/*”开头，以“*/”结束。格式为：

```
/* ……
单行或多行注释内容
……*/
```

（3）文件注释

文件注释用来自动生成一个 HTML 文档，从而为程序自动提供网络环境下的文档说明。文件注释以“/**”开头，以“*/”结束。格式为：

```
/**……
文件注释内容
*/
```

2.8 数组

数组是连续内存单元中一组名字和数据类型相同的数据元素的有限集合。数组可以用来保存和处理一组数据类型相同的数据元素。数组中的每个数据元素称作一个数组元素。

当把一维数组中的每个数据元素定义为一个一维数组时,就构成了 Java 语言的二维数组，依此类推，还可以有三维数组甚至更多维数组。另外，Java 语言可以构造出不规则数组。

2.8.1 一维数组

和变量的使用方法类同，一维数组也要先定义后使用，不同的是数组在定义后还要经过内存单元分配后才能使用。

Java 语言一维数组的使用分三步：定义一维数组变量、为数组分配内存单元和使用数组元素。

1．一维数组变量定义

一维数组变量定义的语法形式为：

〈数据类型〉〈数组名〉[]；

或

〈数据类型〉[]〈数组名〉；

其中，方括号[]表示定义的是数组变量，〈数据类型〉定义了数组元素的数据类型，〈数组名〉定义了数组名的标识符。也可以把数组类型看成 Java 语言在基本数据类型的基础上的扩展。例如，

```
int[]  a;
```

定义了一个数据类型为 int、数组标识符为 a 的一维数组。

在数组定义后，系统将给数组标识符分配一个内存单元，用于指示数组在内存中的实际存放位置。由于在数组变量定义时，数组元素本身在内存中的实际存放位置还没有给出，所以，此时该数组名的值为空（null）。例如，上述语句执行后数组 a 占用内存的状态如图 2.1（a）所示。

2．为数组分配内存单元

Java 语言中，new 是一个特殊的运算符。new 运算符的语法形式为：

new〈数据类型〉

new 运算符的语义是：向系统申请指定数据类型所需的内存单元空间。new 运算符返回所申请内存单元的首地址。

数组元素本身的内存空间必须用 new 运算符在程序中申请。只有用 new 运算符为数组分配了内存单元空间后，存放一个数组的所有数组元素所需的内存空间才真正被系统分配了。为数组类型变量分配内存单元的语法形式为：

〈数组名〉= new〈数据类型〉[〈长度〉]；

其中，〈数组名〉必须是已定义的数组类型变量，〈数据类型〉必须和定义数组名时的数据类型一致，方括号[ ]内的〈长度〉指出了当前数组元素的个数。例如，

```
a = new int[5];
```

就具体分配了包含 5 个 int 类型数组元素的内存单元，并把该块内存单元的首地址赋值给数组名 a。

Java 语言规定，在数组分配内存单元后，系统将自动给每个数组元素赋初值，并规定：数值类型的数组元素初值为 0，逻辑类型的数组元素初值为 false，类类型的数组元素初值为 null。图 2.1（b）是上述语句执行后的示意图。

3．使用数组元素

一旦完成了定义数组变量和为数组分配内存单元后，就可以使用数组中的任意数组元素。数组元素由数组名、一对方括号、方括号中的整数数值（一般称作下标）组成。其中下标指

出了希望操作的数组元素位置。下标由 0 开始，其最大值为用 new 运算符分配内存单元时规定的长度值减 1。各数组元素在内存中按下标的升序连续存放。上述数组 a 的 5 个元素依次是 a[0]，a[1]，a[2]，a[3]，a[4]。例如，

```
a[0] = 10;
```

语句就给数组元素 a[0]赋了数值 10。图 2.1（c）是上述语句执行后的示意图。

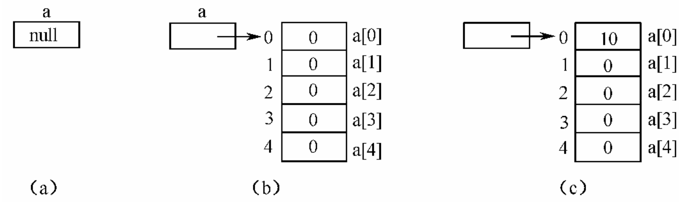


图 2.1　一维数组存储结构

4．引用类型

前面讨论的用基本数据类型定义变量和这里讨论的定义数组变量有一些不同。用基本数据类型定义的变量，其变量名表示这个变量名中存放的数值，如有下列语句段：

```
int i, x;                    //定义变量
i = 10;                      //给变量 i 赋值
x = 10 + i;                  //使用变量 i 中的数值
```

上述语句段中，变量 i 的存储结构如图 2.2（a）所示，语句 x = 10 + i 中，赋值号右边的变量 i 表示变量 i 中的数值 10。

对于数组变量，设有下列语句段：

```
int[]   a;
int x;
a = new int[5];
a[0] = 10;                   //给数组元素 a[0]赋值
x = 10 + a[0];               //使用数组元素 a[0]中的数值
x = 10 + a;                  //错误，数值 10 和数组名 a 为不兼容的类型
```

上述语句段中，数组 a 的存储结构如图 2.2（b）所示，语句 x = 10 + a[0]中，赋值号右边的数组元素 a[0]表示数组元素 a[0]中的数值，但语句 x = 10 + a 将出错，因为数组名 a 是指向内存中存放数组元素的一片连续内存单元的首地址，所以，数值 10 和数组名 a 为不兼容的类型。

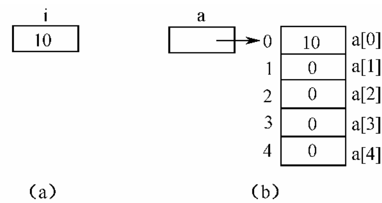


图 2.2　引用类型的内存结构

Java 语言中，数组名的类型是引用类型。所谓引用类型，是指该类型的标识符表示的是一片内存连续地址的首地址。

引用类型是非常重要的一个概念。下一节要讨论字符串，字符串名和数组名一样，也是引用类型。第 4 章要讨论对象，所有对象名都是引用类型。

5．数组的简化使用方法

数组的定义和为数组分配内存空间两步可以结合起来。例如，

```
int a[] = new int[5];
```

就在定义 int 类型数组变量 a 的同时为数组分配了 5 个 int 类型数组元素所需的内存空间，并给每个数组元素赋初值 0。

数组元素的初始化赋值也可以和上述语句结合在一起完成，此时采用简写形式。例如，

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

就在定义 int 类型数组变量 a、为数组分配了 5 个 int 类型数组元素所需的内存空间的同时，初始化给数组元素 a[0]赋初值 1，a[1]赋初值 2，……，a[4]赋初值 5。

6．数组元素允许的运算

对数组元素可以进行其数据类型所允许的任意运算。例如，

```
int u = 3, v = 4, x, y;
int a[] = {1,2,3,4,5};
x = (a[2] + a[3] – u) * v;
y = a[1] / u;
```

都是合法的表达式。

7．数组的长度

Java 语言提供了 length 成员变量返回数组元素的个数，其使用方法为：

```
〈数组名〉.length
```

例如，

```
int n；
int a[] = new int[10]；
n = a.length；
```

则有 n 等于 10。

8．数组长度的重新定义

一旦为数组规定了长度，在程序中使用数组时就不能超过所规定的长度，否则编译时会给出“数组下标越界”的语法错误。例如，若数组分配的内存空间为 5 个，则语句中一旦出现 a[5]，将产生“数组下标越界”的语法错误。

上述情况下，可以用 new 运算符重新为数组分配内存单元。例如，

```
a = new int[10];
```

上述语句后，由于重新为数组 a 分配了 10 个 int 类型的内存单元空间，所以，此时若语句中出现 a[5]，编译时将不会出现“数组下标越界”的语法错误。

【例 2.11】 求 10 个数中的最小数。

要求：用数组初始化赋值方法给出 10 个整数数值。

程序设计如下：

```
public class Exam2_11
{
    public static void main(String args[])
    {
        int i, min;
        int a[] = {30,81,37,45,26,46,44,78,80,64};          //初始化赋值

        System.out.print("数组元素为：");
        for(i = 0; i < a.length; i++)
            System.out.print("   " + a[i]);                 //输出数组元素

                                                            //寻找数组中数值最小的元素
min = a[0];
        for(i = 1; i < a.length; i++)
            if(a[i] < min)    min = a[i];

        System.out.println("\n 最小数为：" + min);
    }
}
```

程序的运行结果为：

```
数组元素为：  30   81   37   45   26   46   44   78   80   64
最小数为：26
```

如果此问题不用数组方法设计，而用简单变量方法设计，程序将非常复杂。因此，数组是复杂问题的程序设计所必需的。

【例 2.12】 把 10 个数按从小到大的次序排序。

要求：用数组初始化赋值方法给出 10 个整数数值，用直接交换排序算法排序。

直接交换排序算法思想：例 2.11 程序找到的是数组 a 中的最小数。我们可以在此基础上设计一个循环过程，把每次找到的最小数和数组中尚未排好序的数据元素交换，下次循环时，从这个数据元素的下一个位置开始，继续这样的寻找和交换过程。这样的过程共进行 a.length–1 次，则全部数组中的数据元素就按从小到大的次序排好了。

程序设计如下：

```
public class Exam2_12
{
```

```
        public static void main(String args[])
        {
            int a[] = {30,81,37,45,26,46,44,78,80,64};
            int i, j, min, temp;

            System.out.println("排序前数组元素为：");
            for(i = 0; i < a.length; i++)
                System.out.print(a[i] + "    ");

                                                          //直接交换排序
            for(i = 0; i < a.length-1; i++)               //循环 a.length-1 次
            {
                min = i;
                for(j = i+1; j < a.length; j++)
                    if(a[j] < a[min]) min = j;            //寻找最小数

                if(min != i)                              //判断是否需要交换
                {
                    temp = a[i];
                    a[i] = a[min];
                    a[min] = temp;
                }
        }

            System.out.println("\n 排序后数组元素为：");
            for(i = 0; i < a.length; i++)
                System.out.print(a[i] + "    ");
        }
    }
```

程序的运行结果为：

排序前数组元素为：

    30   81   37   45   26   46   44   78   80   64

排序后数组元素为：

    26   30   37   44   45   46   64   78   80   81

2.8.2　二维数组

Java 语言只定义了一维数组。如果一维数组的每个数组元素都是一个一维数组，则构成了 Java 语言的二维数组。和一维数组的使用方法类同，二维数组的使用也分三步：定义数组变量、为数组分配内存单元和使用数组元素。

二维数组变量定义的一个例子如下：

```
int    a[][];
```

或

```
int[]    a[];
```

上面语句定义了一个数据类型为 int[]（即一维数组类型）、标识符为 a 的一维数组，即数组 a 是二维数组。

上述语句执行后数组 a 的内存状态如图 2.3（a）所示。

为二维数组变量分配内存单元时，必须指定每一维的数组元素个数。例如，语句

```
a = new int[3][3]；
```

就具体分配了包含 3 个 int[3]类型数组元素的内存单元，并把该连续内存单元的首地址赋给数组名 a，同时为每个数组元素初始化赋值 0。图 2.3（b）是上述语句执行后的内存示意图。

使用二维数组元素的方法和使用一维数组元素的方法类同，只是这里要指出二维数组的每一维的下标。例如，语句

```
a[0][0] = 10;
```

就给数组元素 a[0][0]赋了数值 10。图 2.3（c）是上述语句执行后的示意图。

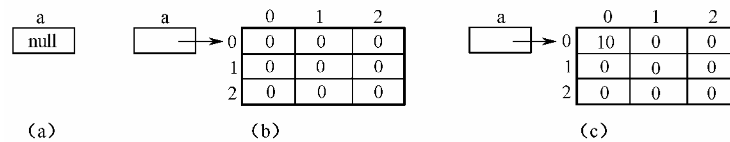


图 2.3　二维数组存储结构

同样，二维数组也可以用简化方法。例如，

```
int a[][] = new int[5][5];
```

在定义 int 类型的二维数组变量 a 的同时，为数组分配内存单元结构如图 2.2（b）所示的 5 个每一维为 int[5]类型数组元素的内存空间，并给每个数组元素初始赋值 0。

又例如，语句

```
int    a[][] = {{1,2,3},{4,5,6},{7,8,9}};
```

在定义 int 类型二维数组变量 a、并为二维数组动态分配了 9 个 int 类型数组元素内存空间的同时，初始化给数组元素 a[0][0]赋初值 1，a[0][1]赋初值 2，a[0][2]赋初值 3，a[1][0]赋初值 4，……，a[2][2]赋初值 9。

三维数组或更多维数组的使用方法和二维数组的使用方法类同。

【例 2.13】　求 $C = A \times B^T$。其中，A 是一个行向量，B^T 是一个列向量，C 是一个矩阵。例如，设 A 和 B 均为 n=3 的向量，则矩阵 C 元素的计算方法是：$c_{11}=a_1* b_1$，$c_{12}=a_1* b_2$，$c_{13}=a_1* b_3$，$c_{21}=a_2* b_1$，…，$c_{33}=a_3* b_3$。

设计思想：这是一个矩阵运算。用一维数组 a 存放行向量 A，用一维数组 b 存放列向量 B^T，用二维数组存放矩阵 C。

程序设计如下：

```
public class Exam2_13
{
    public static void main(String args[])
```

```
    {
        final int n = 3;
        int a[] = {1,2,3};
        int b[] = {4,5,6};
        int c[][] = new int[n][n];
        int i, j;

        for(i = 0; i < n; i++)
            for(j = 0; j < n; j++)
                c[i][j] = a[i] * b[j];                    //计算 cij

        System.out.println ("二维数组元素为：");
        for(i = 0; i < n; i++)
        {
            for(j = 0; j < n; j++)
                System.out.print(c[i][j] + "     ");
                System.out.println();
        }
    }
}
```

程序运行输出结果为：

二维数组元素为：

```
4    5    6
8    10    12
12    15    18
```

2.8.3 不规则的二维数组

由于 Java 语言的二维数组是由一维数组定义的，所以，可以把二维数组中的每个一维数组定义为不同的元素个数，这样就可以构成不规则的二维数组。

不规则二维数组的具体设计方法是：先定义一个二维数组变量，并指定第一维的元素个数，然后再分别为第二维数组（即第一维数组的每个数组元素）分配不同的内存单元。由于此时是分别为第二维数组分配内存单元，并且第二维数组所分配的内存单元个数可以是不相同的，因此就构成了不规则的二维数组。

例如，下面的代码先定义一个二维数组，并为数组的第一维数组元素分配空间（这就要求必须指定其具体个数），然后再分别为第二维数组元素分配不同的内存空间。

```
int twoDim [][] = new int [4][]；         //定义二维数组，并指定第一维的元素个数
twoDim[0] = new int[1]；                  //指定第二维第一个元素的个数
twoDim[1] = new int[2]；                  //指定第二维第二个元素的个数
twoDim[2] = new int[3]；                  //指定第二维第三个元素的个数
twoDim[3] = new int[4]；                  //指定第二维第四个元素的个数
```

数组 twoDim 占用的内存单元状态如图 2.4 所示。

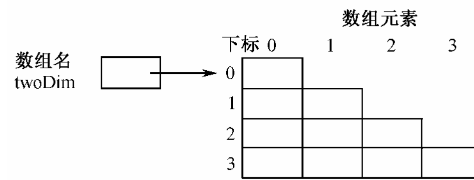


图 2.4　不规则的二维数组

【例 2.14】　计算并保存九九乘法表，要求重复的部分只保存一个。

设计思想：九九乘法表需要一个二维数组来保存，因为要求重复的部分（如 1*2 和 2*1）只保存一个，所以需要把二维数组定义成不规则的二维数组。

程序设计如下：

```
public class Exam2_14
{
    public static void main(String args[])
    {
        final int N = 9;
        int a[][] = new int [N][];              //定义二维数组，并指定第一维的元素个数
        int i,j;

        for(i = 0; i < N; i++)
            a[i] = new int [i+1];               //指定不规则的第二维的个数

        for(i = 0; i < N; i++)
            for(j = 0; j <= i; j++)
                a[i][j] = (i + 1) * (j + 1);    //保存乘法表

                                                //输出乘法表
        for(i = 0; i < N; i++)
        {
            for(j = 0; j <= i; j++)
                System.out.print(a[i][j] + "    ");
            System.out.println();
        }
    }
}
```

程序运行输出如下：

```
1
2    4
3    6    9
4    8    12    16
```

```
5    10   15   20   25
6    12   18   24   30   36
7    14   21   28   35   42   49
8    16   24   32   40   48   56   64
9    18   27   36   45   54   63   72   81
```

2.9 字符串

字符串是 n（$n \geq 0$）个字符组成的序列。为了把一个字符串和别的语言成分区分开来，Java 中的字符串用一对双引号括起来，一个字符串中的字符个数称作字符串的长度。如"abc"就是一个长度为 3、其值为 abc 的字符串。

Java 中的字符串变量用 String 来定义，但和 char, int 等基本数据类型不同的是，String 不是一个数据类型，而是一个类。String 是 Java 应用程序接口（即 Java API）中定义的一个类。由于应用程序一般要有输出，常用的系统标准输出要求输出参数是一个字符串。因此，本节简单介绍字符串的概念和使用方法。

（1）字符串常量

一对双引号括起来的任何字符序列都是一个字符串常量，如""和"sum" 都是字符串常量。字符串常量""的长度为 0，字符串常量"sum"的长度为 3。

（2）字符串变量

定义字符串变量的方法和定义基本数据类型变量的方法类同。如下面语句就定义了两个字符串变量 str1 和 str2：

```
String str1, str2;
```

在定义字符串变量时可以同时赋值。例如，下面语句就在定义字符串变量 str 的同时，给 str 赋了初值"abc"：

```
String str = "abc";
```

还可以定义 String 数组，例如，语句：

```
String[] v = new String[3];
```

就定义了一个有 3 个数组元素的 String 数组 v。

（3）字符串变量名

和数组名一样，字符串变量名也是引用类型，即字符串变量名是指向内存中一片连续内存单元的首地址。

（4）字符串的赋值

字符串变量定义后可以给该变量赋值，例如，

```
String str;
str = "abc";
```

上述字符串变量定义和变量赋值也可以写在一个语句中：

```
String str = "abc";
```

上述语句的功能是：首先，定义字符串变量 str；然后，向系统申请字符类型的长度为 3

的一片连续内存单元，并把字符'a'、'b'、'c'依次存入内存单元中；最后，把这片连续内存单元的首地址赋给字符串变量名 str，即让 str 指向存放字符串"abc"的内存单元的首地址。

给字符串变量赋值时，还可以同时赋多个，其方法等同于数组声明时赋初值。例如：

```
String[] v={"Hello world!","Hello China!","Hello XSYU!"};
```

该语句就给字符串变量 v 赋了 3 个字符串常量值。

（5）字符串的连接运算

Java 语言提供了特殊的字符串运算符"+"。运算符"+"表示把两个字符串连接起来，例如：

```
String str = "abc" + "def";
```

该语句就把字符串值"abcdef"，赋给了字符串变量 str。

（6）标准输出中的字符串

前面已经多次使用了系统的标准输出 System.out.print( )和 System.out.println( )。这两个输出语句要求的参数是字符串或字符串表达式。例如，

```
String str = "abc";
System.out.print("def");
System.out.print(str);
System.out.print(str + "def");
```

都是合法的输出语句。

前面有如下形式的输出语句：

```
int j = 10;
System.out.print("j = " + j);
```

其中，"j = "是一个字符串，j 是一个 int 类型的变量，其数值为 10，显然，表达式"j = " + j 的数据类型不一致。在这里，系统将把 int 类型的数值 10 转换为字符串类型。因此上面语句将输出：

```
j = 10
```

习题 2

一、基本概念题

2.1 什么叫标识符？

2.2 什么叫变量？什么叫常量？系统是否为变量和常量分配内存空间？

2.3 什么叫数据类型？定义变量时为什么要指出该变量的数据类型？

2.4 什么叫表达式？Java 语言共有几种表达式？

2.5 有几种形式的分支语句？解释每种分支语句的执行流程。

2.6 有几种形式的循环语句？解释每种循环语句的执行流程。

2.7 几种循环结构可以互相转换吗？

2.8 什么叫引用类型？数组变量是引用类型变量吗？字符串变量是引用类型变量吗？

2.9 Java **语言是怎样构造二维数组的？**

2.10 Java **语言是怎样构造不规则数组的？**

二、程序设计题

2.11 **下面哪些字符不合法？为什么？**

（1）HellowWorld （2）2Great （3）_come （4）–month （5）count&add

（6）23233 （7）non-stop （8）java （9）Hotjava （10）glade300

2.12 **下面哪些是** Java **中的关键字？**

Transient, constant, this, super, until, goto, repeat interrupt, synchronized, const, class, Java, throws, throw, volatile, try , catch, implements, extends, final, static, finalize。

2.13 **指出下面数值的数据类型。**

（1）396 （2）8864L （3）23.322 （4）23.299D （5）34.333F

（6）23.3e3 （7）’a’ （8）true （9）)false （10）”abcd”

2.14 **叙述数据类型转换时，不损失数据位数或精度应遵守的转换原则。**

2.15 **指出下面程序段中的错误。**

（1）in x=1;

```
    while(x<=10);
    {
        x++;
    }
```

（2）switch(n)

```
    {
        case 1:
            System.out.println("The numbe is 1");
        case 2:
            System.out.println("The numbe is 2");
            Break;
        case 3:
            System.out.println("The numbe is 3");
            Break;
    }
```

2.16 while **与** do-while **语句的区别是什么？**

2.17 **下面程序的输出结果是什么？**

```
public class A
{
    public static void main(String args[])
    {
        int[] a = {11,12,13,14};
        int[] a = {0,1,2,3};
```

```
            System.out.println(a[(a=b)[3]]);
            System.out.println(a[2]);
        }
    }
```

2.18　**下面程序的输出结果是什么？**

```
public class Mystery
{
        public static void main(String args[])
        {
            int y, x = 1, total=0;
            while (x<=10)
            {
                 y=x*x;
                System.out.println(y);
                total +=y;
                ++x;
            }
            System.out.println("Total is "+total);
        }
}
```

2.19　**下面程序的输出结果是什么？**

```
public class A
{
    public static void main(String args[])
    {
        long size=10;
        int[] array = new int[size];
        size =20;
        System.out.println(array.length);
    }
}
```

2.20　**编写程序分别输出如下所示的图形。**

```
* * * * *                  *
* * * *                  * * *
* * *                  * * * * *
* *                  * * * * * * *
* * *                  * * * * *
* * * *                  * * *
* * * * *                  *
   (1)                     (2)
```

2.21　写出三种无限循环的代码段。

2.22　设计程序计算 1+2+3+…+100 的结果。

2.23　设计一个程序，将一维数组中的元素的顺序倒置，例如，数组元素的顺序原来是：1，2，3，4，5；倒置后的顺序变为：5，4，3，2，1。

2.24　设计程序对一个整数数组中的元素进行排序（要求采用不同于例 2.14 的排序方法）。

第3章　类 和 对 象

教学要点

本章内容主要包括面向对象程序设计的基本概念，类的设计方法（包括构造方法、实例成员变量、类成员变量、实例方法、类方法），对象的设计方法（包括对象的创建，对象成员的使用，垃圾对象的回收），构造方法和成员方法的重写，包的设计方法（包括包的创建方法、使用方法和访问权限），内部类，类的封装性。

要求掌握面向对象程序设计的基本概念，理解面向对象方法具有的封装性特点，熟练掌握类的设计方法、对象的设计方法、构造方法和成员方法的重写，掌握内部类的设计方法。

Java 语言是一种面向对象的程序设计语言，类和对象是面向对象程序设计的基本概念。类是相似对象中共同属性和方法的集合体。对象是类的实例。包是 Java 组织和管理类的一种方法。类的封装性是面向对象程序设计的一个重要特点。

3.1　面向对象程序设计

3.1.1　面向对象程序设计的基本概念

Java 语言是一种面向对象的程序设计语言。**面向对象程序设计**（Object Oriented Programming，简称 OOP）是一种集问题分析方法、软件设计方法和人类的思维方法于一体的、贯穿软件系统分析、设计和实现整个过程的程序设计方法。面向对象程序设计方法的出发点和追求的基本目标，是使分析、设计和实现软件系统的方法尽可能接近人类认识事物的方法。面向对象程序设计的基本思想是：对软件系统要模拟的客观实体以接近人类思维的方式进行自然分割，然后对客观实体进行结构模拟和功能模拟，从而使设计出来的软件尽可能直接地描述客观实体，从而构造模块化的、可重用的、维护方便的软件。

在现实世界中，客观实体有两大类。

① 我们身边存在的一切有形事物和抽象概念都是客观实体。有形事物的例子如一位教师、一件衣服、一本书、一个饭店、一座楼、一个学校等；抽象概念的例子如学校校规、企业规定等。

② 我们身边发生的一切事件都是客观实体，如一场足球比赛、一次流感侵袭、一次到医院的就诊过程、一次到图书馆的借书过程，等等。

不同的客观实体具有各自不同的特征和功能。例如，饭店具有饭店的特征和功能，学校具有学校的特征和功能；又例如，就诊过程具有就诊过程的特征和功能，借书过程具有借书

过程的特征和功能。

现实世界中的一切客观实体都具有如下特征：

① 有一个名字用来惟一标识该客观实体；

② 有一组属性用来描述其特征；

③ 有一组方法用来实现其功能。

例如，作为书店客观实体，每个书店有不同的店名、负责人等属性（特征），书店的入库、出库、图书上架等是其基本的方法（功能）。

面向对象程序设计方法下的软件系统设计方法，首先是把问题中涉及的客观实体分割出来，并设计成称为类的可重复使用的模块，然后定义一个个实例化的类（称为对象），再按照问题的要求，设计出用各个对象的操作完成的软件系统。

3.1.2 类

在现实世界中，类是对一组相似客观实体的抽象描述。例如，有 A 书店、B 书店、C 书店等。而书店类则是对书店这类客观实体所应具有的共同属性和方法的抽象描述。即书店类不是具体的描述，而是抽象的描述。

在面向对象方法中，具体的客观实体称为对象，类是对具有相同属性和方法的一组相似对象的抽象，或者说，类所包含的属性和方法描述一组对象的共同的属性和方法。

面向对象方法中软件设计的主体是类。类是相同属性和方法的封装体，因此类具有封装性；子类可以在继承父类所有属性和方法的基础上，再增加自己特有的属性和方法，因此类具有继承性；在一个类层次中，定义为根类的对象可被赋值为其任何子类的对象，并根据子类对象的不同而调用不同的方法，因此类具有多态性。类的这种封装性、多态性和继承性，是面向对象程序设计的三个最重要的特点。

3.1.3 对象

类是具有相同属性和方法的一组相似对象的抽象描述，但在现实世界中，抽象描述的类并不具体存在，例如，现实世界中只存在具体的 A 书店、B 书店、C 书店，并不存在抽象的书店。我们把按照类这个模板所定义的一个个具体的对象称作类的实例，或称作对象。

首先，一个具体对象必须具有具体的属性值，如 A 书店对象就必须具有如下属性：书店名为 A 书店，负责人为张三等。其次，任何对象都具有相应类所规定的所有方法。

3.2 类

面向对象方法中，软件设计的最主要部分是设计类。类的设计可以划分为类声明和类主体设计两部分。

3.2.1 类声明

1．类声明的格式

类声明的格式如下：

```
[〈修饰符〉] class〈类名〉[extends〈父类名〉]  [implements〈接口名表〉]
{
    类主体
}
```

其中，class 是定义类的关键字，〈类名〉是所定义的类的名字，〈父类名〉是已经定义过的类名，〈接口名表〉是已经定义过的若干个接口名。当接口名多于一个时，用逗号分隔开。方括号表示该项是可选项。本章只讨论基本的类设计方法，含有父类和接口的类将在第 4 章讨论。

2．类的修饰符

类声明的〈修饰符〉分为访问控制符和类型说明符两部分，分别用来说明该类的访问权限及该类是否为抽象类或最终类。

（1）访问控制符 public 和默认

当访问控制符为 public 时，表示该类被定义为公共类。公共类表示该类能被任何类访问。由于类都放于某个包中，包中的类互相能访问，而不在一个包中的类互相不能直接访问。如果要在一个包中访问另一个包中的类，就必须用 import 语句导入所需要的类到该包中。Java 语言规定，被导入的类必须是用 public 修饰的类。

当没有访问控制符 public 时，即是默认类（或称缺省类）。默认类表示该类只能被同一个包中的类访问，而不能被其他包中的类访问。Java 语言规定，一个 Java 文件中可以有很多类，但最多只能有一个公共类，其他都必须定义为默认类。

例如，public class Teacher 就声明了一个公共类 Teacher，该类可以通过 import 语句导入其他包的类中，并能被其他所有的类访问。又如，class Student 就声明一个默认类 Student，该类只能被同一个包中的其他类访问。

（2）类型说明符 abstract 和 final

当类型说明符为 abstract 时，表示该类为抽象类。抽象类不能用来定义对象，抽象类通常设计成一些具有类似成员变量和方法的子类的父类。

当类型说明符为 final 时，表示该类为最终类。最终类不能用来再派生子类。

访问控制符和类型说明符一起使用时，访问控制符在前，类型说明符在后。例如，public abstract class Teacher 就声明了一个公共抽象类 Teacher。

3.2.2　类主体设计

在上节讨论面向对象程序设计时曾说过，类由成员变量和成员方法组成。因此，类主体设计包括类的成员变量设计和类的成员方法设计两部分。由于 Java 语言中的所有方法必定属于某个类，即方法一定是成员方法，所以成员方法可简称为方法。我们首先给出一个简单的日期类设计，然后分别讨论类成员变量和类方法的设计。

【例 3.1】　设计一个日期类。要求方法应包括设置日期、显示日期和判断是否是闰年。

类设计如下：

```
public class Date                                    //类声明
{
                                                     //以下为类成员变量声明
```

```
    private int year;                              //成员变量，表示年
    private int month;                             //成员变量，表示月
    private int day;                               //成员变量，表示日

    //以下为类方法声明和实现
    public void SetDate(int y,int m,int d)         //设置日期值
    {
        year = y;
        month = m;
        day = d;
    }

    public void Print()                            //输出日期值
    {
        System.out.println("date is "+year+'-'+month+'-'+day);
    }

    public boolean IsLeapYear()                    //判断是否为闰年
    {
        return (year%400==0) | (year%100!=0) & (year%4==0);
    }
}
```

1．声明成员变量

声明一个成员变量就是声明该成员变量的名字及其所属的数据类型，同时指定其他一些附加特性。声明成员变量的格式为：

[〈修饰符〉] [static] [final] [transient]〈变量类型〉〈变量名〉；

其中，〈修饰符〉有 private、public 和 protected 三种。当不加任何修饰符时，定义为默认修饰符。private 修饰符表示该成员变量只能被该类本身访问，任何其他类都不能访问该成员变量。不加任何访问权限限定的成员变量属于默认访问权限。默认访问权限表示该成员变量只能被该类本身和同一个包的类访问。protected 修饰符表示该成员变量除可以被该类本身和同一个包的类访问外，还可以被它的子类（包括同一个包中的子类和不同包中的子类）访问。public 修饰符表示该成员变量可以被所有类访问。

上述修饰符实现了类中成员变量在一定范围内的信息隐藏。这既符合程序设计中隐藏内部信息处理细节的原则，也有利于数据的安全性。

static 指明该成员变量是一个类成员变量，final 指明该成员变量是常量，transient 指明该成员变量是临时变量。transient 很少使用。类成员变量是一个类的所有对象共同拥有的成员变量。关于类成员变量的用途和使用方法将通过后面的例子说明。

因此，例 3.1 中的成员变量 year、month 和 day 都是 int 类型的 private 成员变量，即这三个成员变量只能被该类本身访问，任何其他类都不能访问该成员变量。

2．声明方法

声明成员方法的格式为：

```
[〈修饰符〉] [static]〈返回值类型〉〈方法名〉([〈参数列表〉])
{
    〈方法体〉
}
```

其中，〈修饰符〉和成员变量的修饰符一样，有 private、public 和 protected 三种，另外，还有默认。各个修饰符的含义也和成员变量修饰符的含义相同。static 指明该方法是一个类方法。关于类方法的用途和使用方法将通过后面的例子说明。

方法声明中必须给出方法名和方法的返回值类型；如果没有返回值，用关键字 void 标记。方法名后的一对圆括号是必须的，即使参数列表为空，也要加一对空括号。例如，

```
public void SetDate(int y,int m,int d)
```

语句声明了方法名为 SetDate 的 public 方法，其返回值为空，参数有三个，分别为 y, m 和 d，这三个参数的数据类型均为 int。

3．方法体

方法体是方法的具体实现。方法体的设计即是第 2 章讨论的变量定义、赋值语句、if 语句、for 语句等根据方法体设计要求的综合应用。例如，

```
public void SetDate(int y,int m,int d)          //设置日期值
{
    year = y;                                   //给成员变量 year 赋值 y
    month = m;                                  //给成员变量 month 赋值 m
    day = d;                                    //给成员变量 day 赋值 d
}
```

4．成员变量和变量

初学者经常会混淆成员变量和变量，一个最简单的区别方法是：定义在类中的都是成员变量，定义在方法内的都是变量。另外，还有定义在方法参数中的虚参变量，如例 3.1 SetDate() 中的 y, m 和 d。关于方法中定义变量的例子见后面的设计举例。

成员变量和变量的类型既可以是基本数据类型（如 int, long 等），也可以是已定义的类。

3.2.3 构造方法

在类的方法中，有一种特殊的方法专门用来进行对象的初始化。这个专门用来进行对象初始化的方法称为构造方法。构造方法也称作构造函数。一般来说，一个类中至少要有一个构造方法。

构造方法在语法上等同于其他方法。因此，构造方法的设计方法和前面说的其他方法的设计方法类同。但构造方法的名字必须与其类名完全相同，并且没有返回值，甚至连表示空类型（void）的返回值都没有。构造方法一般应定义为 public。

构造方法用来在对象创建时为对象的成员变量进行初始化赋值。其实现过程是：在创建对象时，将调用相应类中的构造方法为对象的成员变量进行初始化赋值。

例如，我们可以为上述日期类设计一个如下的构造方法：

```
public Date(int y, int m, int d)                    //构造方法
{
    year = y;
    month = m;
    day = d;
}
```

3.3 对象

类是对一组相似对象的抽象描述。一个软件系统是对具体问题的客观事物进行模拟或描述的，因此需要具体的描述。对象是类的实例化，对象就是软件系统中对具体问题的客观事物进行的具体模拟或具体描述。

3.3.1 main 方法

类是作为许多相似对象的抽象描述设计的。如果要使用这个类，就必须创建这个类的对象。那么，对象创建应该是在同一个类中还是应该在另一个类中呢？答案是两者都可以，但最好是在另一个类中。这样，没有对象定义的纯粹的类设计部分就可以单独保存在一个文件中，就不会影响该类的重复使用。

Java 语言规定，一个类对应一个.class 文件，一个程序文件中可以包含一个或一个以上的类，但其中只允许一个类被定义成 public 类。类中可以没有 main 方法。但是要运行的类中必须有 main 方法。程序就是从 main 方法开始执行的。

下面的例子是把对象创建在同一个类中的 main 方法中。

【例 3.2】 打印某个日期，并判断该年是否是闰年。

```
public class Date                                   //类声明
{
    private int year;                               //成员变量，表示年
    private int month;                              //成员变量，表示月
    private int day;                                //成员变量，表示日

    public Date(int y, int m, int d)                //构造方法
    {
        year = y;
        month = m;
        day = d;
    }

    //以下为其他类方法
```

```
    public void SetDate(int y, int m, int d)                    //设置日期值
    {
        year = y;
        month = m;
        day = d;
    }

    public void Print()                                         //输出日期值
    {
        System.out.println("date is "+year+'-'+month+'-'+day);
    }

    public boolean IsLeapYear()                                 //判断是否闰年
    {
        return (year%400==0) | (year%100!=0) & (year%4==0);
    }

    public static void main(String args[])                      //main()方法
    {
        Date a = new Date(2004, 8, 5) ;                         //创建对象
        a.Print();
        if(a.IsLeapYear())
            System.out.println(a.year + " 是闰年");
        else
            System.out.println(a.year + " 不是闰年");
    }
}
```

main 方法必须放在类中，且格式必须为：

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

3.3.2　对象的创建和初始化

在例 3.2 中，语句

```
Date a = new Date(2004, 8, 5) ;
```

实现了定义对象 a 和为对象分配内存空间，并初始化对象 a 的成员变量数值为：

```
year = 2004; month = 8; day = 5;
```

上述方法把定义对象和创建对象这两个步骤结合在一起，同时进行对象的初始化赋值。这是最简单，也是最经常使用的对象定义、创建和初始化方法。

对象的定义和创建过程也可以分开进行，即首先定义对象，然后为对象分配内存空间，并可同时进行初始化赋值。

1．定义对象

定义对象的语句格式为：

〈类名〉　〈对象名〉；

例如，下面语句就定义了一个 Date 类的对象 a：

```
Date a;
```

对象和数组一样，也是引用类型。即对象定义后，系统将给对象标识符分配一个内存单元，用于存放实际对象在内存中的存放位置。由于在对象定义时，对象在内存中的实际存放位置还没有给出，所以，此时该对象名的值为空（null）。上述语句后对象 a 占用内存的状态如图 3.1（a）所示。

2．为对象分配内存空间和进行初始化赋值

和为数组分配内存空间一样，为对象分配内存空间也使用 new 运算符。为对象分配内存空间的语句格式为：

〈对象名〉= new〈类名〉([〈参数列表〉])；

其中，new 运算符申请对象所需的内存空间，new 运算符返回所申请的内存空间的首地址。系统将根据〈类名〉和〈参数列表〉调用相应的构造方法，为对象进行初始化赋值（即把参数值存入相应的内存单元中）。赋值语句把 new 运算符分配的连续地址的首地址赋给对象名。正因为构造方法名和类名完全相同，所以这里的类名既用来作为 new 运算符的参数向系统申请对象所需的内存空间，又作为构造方法名为对象进行初始化赋值。例如，

```
a = new Date(2004, 8, 5) ;
```

就先向系统申请了 Date 类对象所需的内存空间（其首地址由对象名 a 指示），又用参数 2004, 8 和 5 调用了构造方法 Date(2004, 8, 5)，为对象 a 进行初始化赋值。上述语句后对象 a 的当前状态如图 3.1（b）所示。

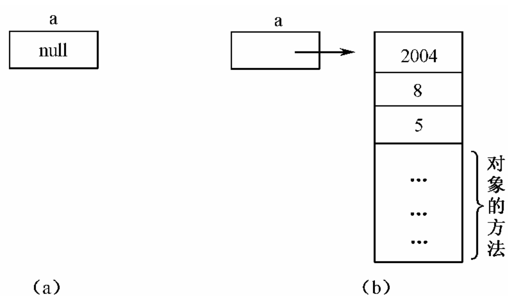


图 3.1　对象的存储结构

程序设计时最经常使用的方法，是在定义对象的同时为对象分配内存空间和进行初始化赋值。例如，

```
Date a = new Date(2004, 8, 5) ;
```

3.3.3　对象的使用

一旦定义并创建了对象(创建对象是指为对象分配内存单元),就可以在程序中使用对象。对象的使用主要有三种情况，分别是：使用对象的成员变量或方法，对象间赋值和把对象作为方法的参数。

1．使用对象的成员变量或方法

一旦定义并创建了对象，就可以使用对象的成员变量或方法。例如，例 3.2 的 main()方法中 a.year 使用了对象 a 的成员变量 year。又如，例 3.2 的 main()方法中 a.Print()使用了对象 a 的成员方法 Print()。

另外，还可以修改对象的成员变量的数值。例如，在例 3.2 的 main()方法中增加语句：

```
a.year = 2005;
```

就把对象 a 的成员变量 year 的数值修改为 2005。

2．对象间赋值

对象可以像变量一样赋值。例如，在例 3.2 的 main()方法中增加语句：

```
Date b;
b = a;
```

则对象 b 的值和对象 a 的值相同。

和变量赋值不一样的是，对象赋值并没有真正把一个对象的取值赋给另一个对象，而是让另一个对象名（如对象名 b）存储的对象的首地址和这个对象名（如对象名 a）存储的对象的首地址相同。即对象的赋值是对象的首地址的赋值。例如，上述语句就把对象 a 的首地址值赋给了对象 b，因此，对象 b 和对象 a 表示的是同一个对象。

仔细分析图 3.1 所示的对象的存储结构，就可以理解对象间赋值的实现方法。

3．把对象作为方法的参数

对象也可以像变量一样，作为方法的参数使用。但系统实现两者的方法不同，变量作为方法的实参时，系统把实参的数值复制给虚参；而对象作为方法的实参时，系统是把实参对象名（该对象必须已创建）指示的对象的首地址赋给虚参对象名。其实现方法实际上和对象间赋值的实现方法相同。

3.3.4　垃圾对象的回收

从上面的讨论可知，对象和变量在很多方面有些类同。例如，对象和变量都需要分配内存空间，但是，变量的内存空间是系统在变量定义时自动分配的；当变量超出作用范围时，系统将自动回收该变量的内存空间。

而对象的内存空间是在用户需要时，用 new 运算符创建的。对象也有作用范围，我们把超出作用范围的对象（或称不再被使用的对象）称作**垃圾对象**。那么，谁来负责这些垃圾对象的回收呢？答案是，在 Java 中，收集和释放内存由自动垃圾回收线程责任。线程的概念将在第 10 章讨论，这里可以把自动垃圾回收线程理解为一个系统自己开启、并与用户程序并行运行的一个服务程序。自动垃圾回收线程在系统空闲时自动运行，这个线程监视用户程序中

所有对象的有效作用范围；当某个对象超出其作用范围时，该线程就对这样的对象做上垃圾对象标识，并在适当的时候一次性回收这些垃圾对象。

所以，用户程序只需考虑为对象分配内存空间，不需考虑垃圾对象内存空间的回收。

3.3.5 实例成员变量与类成员变量

类有两种不同类型的成员变量：实例成员变量与类成员变量。类成员变量也称作静态成员变量。

1．实例成员变量

类定义中没用关键字 static 修饰的成员变量就是**实例成员变量**，不同对象的实例成员变量的值不相同。例 3.2 中类 Date 的成员变量定义语句：

```
private int year;
private int month;
private int day;
```

就定义了三个 private 类型的实例成员变量 year, month 和 day。若有如下对象定义：

```
Date a = new Date(2003, 1, 1) ;
Date b = new Date(2004, 5, 10) ;
```

则对象 a 和对象 b 的实例成员变量数值就不同。如 a.year 的数值为 2003，而 b.year 的数值为 2004。

2．类成员变量

用关键字 static 修饰的成员变量称为**类成员变量**，一个类的所有对象共享该类的类成员变量。类成员变量可以用来保存和类相关的信息，或用来在一个类的对象间交流信息。

【例 3.3】 用类成员变量表示当前类共有多少个对象被定义。

```
public class ObjectNumber                          //类声明
{
    private static int number = 0;                 //类成员变量

    public ObjectNumber()                          //构造方法
    {
        number++;
    }

    public void Print()                            //对象个数输出
    {
        System.out.println(number);
    }

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

```
    {
        ObjectNumber a = new ObjectNumber() ;               //创建对象
        System.out.print("当前对象个数为：");
        a.Print();                                          //对象个数输出
        ObjectNumber b = new ObjectNumber() ;               //创建对象
        System.out.print("当前对象个数为：");
        b.Print();                                          //对象个数输出
    }
}
```

输出结果为：

```
当前对象个数为：1
当前对象个数为：2
```

3.3.6 实例方法与类方法

类有两种不同类型的方法：实例方法与类方法。类方法也称作静态方法。

1．实例方法

没用关键字 static 修饰的方法就是实例方法。实例方法只能通过对象来调用。实例方法既可以访问类成员变量，也可以访问类变量。例 3.2 中类 Date 的 SetDate()方法、Print()方法和 IsLeapYear()方法都是实例方法。这些实例方法体中都访问了实例成员变量。

2．类方法

用关键字 static 修饰的方法称为类方法。类方法通过类名来调用（也可以通过对象来调用）。类方法只能访问类变量，不能访问实例变量。类方法主要用来处理和整个类相关的数据。虽然类方法既可以用类名来调用，也可以用对象来调用，但用类名调用类方法程序的可读性更好。

【例 3.4】　用类方法处理当前类共有多少个对象被定义。

程序设计如下：

```
public class ObjectNumber2                                  //类声明
{
    private static int number = 0;                          //类成员变量

    public ObjectNumber2()                                  //构造方法
    {
        number++;
    }

    public static void Print()                              //类方法
    {
        System.out.println("当前对象个数为：" + number);
    }
```

```
        public static void main(String args[])
        {
            ObjectNumber2 a = new ObjectNumber2() ;          //创建对象
            ObjectNumber2.Print();                           //对象个数输出
            a.Print();                                       //可以，但不提倡
            ObjectNumber2 b = new ObjectNumber2() ;          //创建对象
            ObjectNumber2.Print();                           //对象个数输出
            b.Print();                                       //可以，但不提倡
        }
    }
```

输出结果为：

```
当前对象个数为：1
当前对象个数为：2
```

3.3.7 方法的重写

类的各种方法（包括构造方法和其他方法）都允许重写（也称做重载）。所谓方法重写（overloading），是指一个方法名定义多个方法实现。方法重写时要求，不同的方法，其参数类型或参数个数要有所不同。若类的某个方法被重写，该类的对象在访问该方法时，以对象调用该方法的参数个数和参数类型与类的同名方法进行匹配，对象调用该方法的参数个数和参数类型与类定义中哪个方法的参数个数和参数类型完全一样，则调用类中的哪个方法。

【例 3.5】 方法重写示例。

```
    public class Date2                                  //类声明
    {
        private int year;                               //成员变量，表示年
        private int month;                              //成员变量，表示月
        private int day;                                //成员变量，表示日

        public Date2(int y, int m, int d)               //构造方法
        {
            year = y;
            month = m;
            day = d;
        }

        public Date2()                                  //构造方法
        {
            year = 2004;
            month = 8;
            day = 10;
```

```
        }

        public void Print()                              //输出日期值
        {
            System.out.println("date is "+year+'-'+month+'-'+day);
        }

        public void Print(int y)                         //输出日期值
        {
            System.out.println("date is "+y+'-'+month+'-'+day);
        }

        public static void main(String args[])
        {
            Date2 a = new Date2(2003,1,1) ;              //创建对象
            Date2 b = new Date2() ;                      //创建对象
            a.Print();
            b.Print(2000);
        }
    }
```

程序运行输出：

```
date is 2003-1-1
date is 2000-8-10
```

上述例子中，构造方法和 Print()方法都重写了两个。第一个构造方法有三个参数，第二个构造方法没有参数；第一个 Print()方法没有参数，第二个 Print()方法有一个参数。在 main()方法中，定义对象 a 时有三个参数，所以和第一个构造方法匹配，定义对象 b 时没有参数，所以和第二个构造方法匹配；对象 a 调用方法 Print()时没有参数，所以和第一个 Print()方法匹配，对象 b 调用方法 Print()时有一个参数，所以和第二个 Print()方法匹配。

必须注意的是，方法重写时必须做到：要么参数个数不同，要么参数类型不同。否则，系统无法识别与重写的哪个方法匹配。如果两个重写方法仅返回值的类型不同，则是不允许的。

3.4 包

面向对象程序设计的一个特点就是公共类资源可以重用。这样，在设计一个软件系统过程中设计的许多公共类（除包含有 main 方法的公共类外），就可以在以后的软件系统设计中重复使用。当应用软件比较大时，就有许多 Java 文件；这些 Java 文件统统放在一个文件夹中，给以后的软件资源重用带来许多不便。Java 解决此问题的方法是包。

包（package）是 Java 提供的文件（即公共类）的组织方式。一个包对应一个文件夹，一个包中可以包括许多类文件。包中还可以再有子包，称为包等级。

Java 语言可以把类文件存放在可以有包等级的不同的包中。这样，在设计软件系统时，

就可以把相关的一组文件（即相关的一组公共类）存放在一个文件夹中。当文件夹太大时，还可以设计子文件夹，按更细的分类方法存放相关文件，从而可以大大方便日后的软件资源重用。Java 语言规定：同一个包中的文件名必须惟一，不同包中的文件名可以相同。Java 语言的包等级和 Windows 的文件组织方式完全相同，只是表示方法不同。

3.4.1 包的建立方法

1．定义文件所属的包

简单的包的定义语句格式为：

```
package 〈包名〉;
```

其中，package 是关键字，〈包名〉是包的标识符。package 语句指出该语句所在文件所有的类属于哪个包。

Java 语言规定，如果一个 Java 文件中有 package 语句，那么 package 语句必须写在 Java 源程序的第一行。例如，下面的 Java 源程序 MyClass.java 中的类 MyClass 将属于包 MyPackage。

```
package MyPackage;
public class MyClass;
{
    ⋮
}
```

2．创建包文件夹

程序员自定义的包（如前面例子的 MyPackage 包）必须把其文件夹所在的路径通知系统，这就需要用户首先创建包的文件夹并设置包的路径。创建包文件夹的方法如下。

（1）创建与包同名的文件夹

例如，可以在 D 盘根目录下创建一个与包同名的文件夹 d:\MyPackage。注意，这里的包名 MyPackage 必须与 package 语句后的包名大小写完全一致。

（2）设置包的路径

用户必须通过设置环境变量 classpath，把用户自定义包所在的路径添加到环境变量 classpath 中。例如，作者设置的包 MyPackage 的路径为 d:\ ，所以要把 d:\添加到环境变量 classpath 中。因此，1.3.1 节讨论的环境配置中，第二条环境参数设置语句应改写为：

```
set classpath=.;c:\jdk1.3.1\lib;d:\;
```

环境参数设置说明：

① 分号（;）用来分隔各项，因此，上述的设置共有三项；

② c:\jdk1.3.1\lib 是作者计算机上安装的 JDK1.3.1 版本的系统包的路径；

③ 新添加的 d:\是用户自定义包文件夹的上一级路径；

④ 新添加的路径 d:\也可放在圆点（.）（表示当前工作路径）前，则操作时只需把当前路径下编译成功的.class 文件复制到自定义包文件夹中；如果路径 d:\放在圆点（.）后，则操作时需把当前路径下编译成功的.class 文件移动到自定义包文件夹中。

当多个 Java 源程序文件中都有 package 语句，且 package 语句后的包名相同时，则说明

这些类同属于一个包。

一个包还可以有子包，子包下还可以有子子包。在这种情况下，可以具体指明一个类所属包的完整路径。所以，完整的 package 语句格式为：

package 〈包名〉[.〈子包名〉[.〈子子包名〉…]]；

其中，在 package 语句中，圆点（.）起分隔作用；而在 Windows 的目录中，圆点（.）和反斜杠（\）等义，即加一个圆点（.）就表示下一级目录。

当然，要把一个类放在某个子包或子子包中，前提条件是已经创建与子包或子子包同名的且目录结构也相同的文件夹。

3．把编译生成的.class 文件移入包中

用户的源程序文件（即.java 文件）通常存放在另外的文件夹中，.java 文件编译后产生的.class 文件也存放在和.java 文件相同的文件夹中。用户在编译.java 文件成功后，需要把编译成功的.class 文件移入用户自定义的包中。要保证包中有相应的.class 文件，而当前工作目录下没有。

例如，当上述的 MyClass.java 文件编译成功后，需要设计人员自己把 MyClass.class 文件移入到 d:\ MyPackage 文件夹中，否则系统会因找不到类而给出出错信息。

3.4.2 包的使用方法

包中存放的是编译后的类文件（.class 文件）。用户可以在以后的程序中，通过 import 语句导入这些类，从而使用包中的这些类。import 语句的使用分两种情况：①导入某个包中的某个类；②导入某个包中的全部类。这两种情况分别用如下两种形式的 import 语句：

```
import   MyPackage.MyClass;            //导入包 MyPackage 中的 MyClass 类
import   MyPackage.*;                  //导入包 MyPackage 中的全部类，但不包含其子包
```

要说明的是，Java 中的包是按类似 Windows 文件的形式组织的，Windows 文件用反斜杠（\）表示一条路径下的子路径，而 Java 用圆点（.）表示一个包的子包。

【例 3.6】 设计一个日期类及其测试程序。

要求：把日期类放在包 MyPackage 中，以便于以后重复使用。

程序设计如下：

```
//Date1.java 文件
package MyPackage;                     //定义类所属的包
public class Date1                     //类声明
{
    public int year,month,day;         //成员变量，表示年、月、日
    public Date1(int y, int m, int d)  //构造方法
    {
        year = y;
        month = m;
        day = d;
    }
```

```
    public void print()                                          //输出日期值
    {
        System.out.println("日期是：" + year + '-' + month +  '-'+day);
    }
}

// UseDate1.java 文件
import MyPackage.Date1;                                          //导入包 MyPackage 中的 Date1 类
public class UseDate1
{
    public static void main(String args[])
    {
        Date1 a = new Date1(2004,5,10) ;                         //创建对象
        a.print();
    }
}
```

程序运行结果：

```
日期是：2004-5-10
```

程序设计说明：因 UseDate1.java 文件和 Date1.java 文件不在一个包中，所以，UseDate1.java 文件要用 import 语句导入文件中使用的类。

综上所述，编写、运行上述带有自定义包 Java 程序的操作步骤如下。

① 创建文件夹。如在本地计算机的 d 盘创建文件夹 MyPackage（d:\ MyPackage）。

② 在环境变量中添加自定义包的路径。如在 autoexec.bat 文件的 classpath 参数中添加 d:\（注意：若在 Windows 98 下，则设置完成后要运行一下该批处理文件）。

③ 编译包中类的.java 文件。例如，在 DOS 环境下执行命令 javac Date1.java。

④ 把编译成功的.class 文件移入包中。例如，把当前工作路径下的 Date1.class 文件移动到文件夹 d:\MyPackage 中。

⑤ 编译导入包的.java 文件。例如，在 DOS 环境下执行命令 javac UseDate.java。

⑥ 运行导入包的.class 文件。例如，在 DOS 环境下执行命令 java UseDate。

3.4.3 包的访问权限

关于包中类的访问权限已在 3.2.1 节讨论，关于包中类的成员变量和方法的访问权限已在 3.2.2 节讨论。本节分相同包中的类和不同包中的类两种情况举例说明。

1．相同包中的类和类的成员的访问权限

【例 3.7】 相同包中的访问权限举例。

程序设计如下：

```
//文件 B1.java
package MyPackage;                                               //文件中定义的两个类都在同一个包中
class C1                                                         //C1 声明为默认类
```

```
{
    int number;                         //默认成员变量 number
    protected int age;                  //protected 成员变量 age

    C1(int n, int a)                    //构造方法
    {
        number = n;
        age = a;
    }
    public void output()                // C1 类的 public 方法
    {
        System.out.println("number = " + number + "\n" + "age = " + age);
    }
}

public class B1                         //B1 声明为 public 类
{
    public void output()                //B1 类的方法 output()
    {
        C1 s1 = new C1(0,0);            //B1 类可以访问同一个包中的默认类 C1
        s1.number = 1;                  //同一包的对象可以访问默认类的默认成员变量
        s1.age = 25;                    //同一包的对象可以访问默认类的 protected 成员变量
        s1.output();                    //同一包的对象可以访问默认类的 public 方法
    }
}

//文件 D1.java
//类 D1 在当前工作路径下
import MyPackage.B1;                    //导入 MyPackage 包中的 B1 类
public class D1
{
    public static void main(String args[])
    {
        B1 t1 = new B1();
        t1.output();
    }
}
```

程序的运行结果为：

```
number = 1
age = 25
```

程序说明：D1 类中只能定义 B1 类的对象，不能定义 C1 类的对象（因 C1 类定义为默认

类）；但 B1 类中可以定义 C1 类的对象（因两个类在同一个包中）。

2．不同包中的类和类的成员的访问权限

【例 3.8】　不同包中的访问权限举例。

要求：把例 3.7 中的 C1 类和 B1 类分别放在两个不同的包中。

程序设计如下：

```
//文件 C2.java
package MyPackage.MyPackage1;
public class C2
{
    public int number;
    public int age;

    public C2(int n, int a)
    {
        number = n;
        age = a;
    }
    public void output()
    {
        System.out.println("number = " + number + "\n" + "age = " + age);
    }
}

//文件 B2.java
package MyPackage;
import MyPackage.MyPackage1.C2;
public class B2
{
    public void output()
    {
        C2 s1 = new C2(0,0);
        s1.number = 1;
        s1.age = 25;
        s1.output();
    }
}

//文件 D2.java
import MyPackage.B2;
public class D2
```

```
{
    public static void main(String args[])
    {
        B2 t1 = new B2();
        t1.output();
    }
}
```

程序的运行结果和例 3.7 的相同。

程序设计说明：

① 把例 3.7 程序稍做改变，把 C2 类放在 MyPackage.MyPackage1 包中（当然，要建立相应的文件夹），把 B2 类放在 MyPackage 包中。当然，B2.java 文件中要用 import 语句导入 C2 类。此时，由于 B2 类和 C2 类不在同一个包中，而 C2 类定义为默认类，所以编译时语句

```
C2 s1 = new C2(0,0);
```

将出错。把 C2 类定义为 public 类后，则不会出现上述错误。由于 B2 类和 C2 类不在同一个包中，所以编译时语句：

```
s1.number = 1;
s1.age = 25;
```

将出错，这是由于 C2 类的 age 和 number 成员变量定义为 protected，number 定义为默认，而修饰为 protected 和默认的成员变量不允许其他包中的 C2 类的对象调用；当把 C2 类的 age 和 number 成员变量的修饰符改为 public 时，编译成功。

② 如果把 C2 类放在 MyPackage 包中，把 B2 类放在 MyPackage.MyPackage1 包中，则编译时会出错。这是由于 JDK 规定：在一个树型结构的包中，上层包可以导入下层包，而下层包不可以导入上层包。在下层包的类中要使用上层包的类时，要在类前面加上包名。

3.4.4　系统定义的包

Java 语言提供了许多方便用户程序设计的基础类，这些系统定义的类以包的形式保存在系统包文件夹中。如果要使用这些包中的类，必须在源程序中用 import 语句导入。其导入方法和前面介绍的导入自定义包方法类同。例如，要进行图形用户界面设计，需要导入系统包 java.awt 中的所有类，所以要在源程序中使用如下导入语句：

```
import    java.awt.*；                    //导入 java.awt 包中的所有类
```

又如，在进行图形用户界面设计时，还需要进行事件处理，因此需要导入图形用户界面包 java.awt 中的所有类和事件处理包 java.awt.event 中的所有类，所以要在源程序中使用如下导入语句：

```
import java.awt.*；                       //导入 java.awt 包中的所有类
import java.awt.event.*;                  //导入 java.awt.event 包中的所有类
```

读者也许会想，从 Java 语言包的组织形式看，java.awt.event 包是 java.awt 包的子包，那么，仅有第一条导入语句就可以了，第二条导入语句似乎没有必要。需要注意：第一条导入语句只导入了 java.awt 包中的所有类，并没有导入 java.awt 包的子包，因此，也就没有导入

子包中的类。

3.5 内部类

一个类被嵌套定义于另一个类中，称为**内部类**(Inner Classes)或内隐类。包含内部类的类称为**外部类**。

内部类中还有一种更特殊的形式——匿名类，匿名类和内部类的功能类似。这里我们只讨论内部类，不讨论匿名类。

内部类与前面讨论的非内部类的设计方法基本相同，但除外部类外的其他类无法访问内部类。当一个类只在某个类中使用，并且不允许除外部类外的其他类访问时，可考虑把该类设计成内部类。

【例 3.9】 设计一个人员类，要求自动生成人员的编号。

设计思想：由于要自动生成人员的编号，因此可设计一个 static 成员变量，当每生成一个对象时，该成员变量自动加 1。由于这样的处理只是作用于外部类，所以把该处理过程用内部类的方法来实现。

程序设计如下：

```
public class PeopleCount                        //外部类 PeopleCount
{
    private String Name;
    private int ID;                             //外部类的私有成员变量
    private static int count = 0;               //外部类的 static 私有成员变量
    public class People                         //内部类 People
    {
        public People()                         //内部类的构造方法
        {
            count++;                            //访问外部类的成员变量
            ID = count;                         //访问外部类的成员变量
        }
        public void output()                    //内部类的方法
        {
            System.out.println(Name + "的 ID 为：" + ID);
        }
    }
    public PeopleCount(String sn)               //外部类的构造方法
    {
        Name = sn;
    }

    public void output()                        //外部类的方法
    {
```

```
            People p = new People();                    //建立内部类对象 p
            p.output();                                 //通过 p 调用内部类的方法
        }
        public static void main (String args[])
        {
            PeopleCount p1 = new PeopleCount("张三");
            p1.output();
            PeopleCount p2 = new PeopleCount("李四");
            p2.output();
        }
    }
```

程序的运行结果为：

```
张三的 ID 为：1
李四的 ID 为：2
```

程序设计说明：在外部类 PeopleCount 内嵌套定义了一个内部类 People，当定义外部类对象 p1 和 p2 后，调用 p1 或 p2 的方法 output 时，该方法将首先定义一个内部类对象，内部类对象的构造方法将先把外部类的 static 成员变量 count 加 1，然后把 count 的值赋给人员编号成员变量 PeopleID，然后输出 PeopleID 的值。

外部类与内部类的访问原则是：在外部类中，一般通过一个内部类的对象来访问内部类的成员变量或方法；在内部类中，可以直接访问外部类的所有成员变量和方法（包括静态成员变量和方法、实例成员变量和方法及私有成员变量和方法）。

内部类具有以下特性。

① 内部类作为外部类的成员。Java 将内部类作为外部类的一个成员，因此内部类可以访问外部类的私有成员变量或方法。

② 内部类的类名只能用在外部类和内部类自身中。当外部类引用内部类时，必须给出完整的名称，且内部类的类名不能与外部类的类名相同。

在实际的 Java 程序设计中，内部类主要用来实现接口。

3.6 类的封装性

面向对象程序设计语言的一个重要特性是其封装性。Java 语言是按类划分程序模块的，Java 语言很好地实现了类的封装性。

保证大型软件设计正确性和高效性的一个重要原则是模块化软件设计。这需要设计许多可重复使用的模块，然后在需要使用这些模块的地方调用这些模块。但是，如何划分好模块的界限，以保证模块的正确性是非常重要的问题。因为如果某人随意修改了已经被其他人使用的模块，必将使程序出错，并且这样的错误很难被发现和修改。

在面向对象程序设计中，保证模块正确性的基本方法是类的封装性。类的封装性是指类把成员变量和方法封装为一个整体，这就划分了模块的界限。

保证模块正确性的措施是由信息的隐藏性实现的。类包括成员变量和方法两部分。那些允许其他包程序访问和修改的成员变量可以定义为 public 类型。那些只允许同在一个包中的

其他类，以及该类的子类访问和修改的成员变量可以定义为 protected 类型。那些不允许其他类（内部类除外）访问和修改的成员变量可以定义为 private 类型。private 类型和 protected 类型的成员变量有效地隐藏了类的不能被随意修改的成员变量信息，从而保证了共享的类模块的正确性。类的封装性和信息的隐藏性是“双胞胎”，两者是紧密结合在一起的。

同样，允许其他包程序访问的方法可以定义为 public 类型；只允许同在一个包中的其他类，以及该类的子类访问的方法可以定义为 protected 类型。不允许其他类（内部类除外）访问的方法可以定义为 private 类型。类方法不同类型的定义，给调用者提供了权限明了的调用接口。

和别的面向对象程序设计语言（如 C++语言）相比，Java 语言增加了包的概念。这样，同一个包中类之间的信息传递就比较方便。

3.7 设计举例

本节给出一个较为复杂的程序设计举例。

【例 3.10】 设计一个包括矩阵加和矩阵减运算的矩阵类，并设计一个测试程序完成简单的测试。为简化设计代码，矩阵的元素值在构造方法中用随机函数随机给出。

程序设计如下：

```
//MyMatrix.java 文件
public class MyMatrix                                   //矩阵类 MyMatrix
{
    private int[][] table;                              //矩阵元素表
    private int height;                                 //矩阵的行
    private int width;                                  //矩阵的列

    private void init(int m,int n)                      //元素随机赋值方法
    {
        table=new int[m][n];                            //分配矩阵元素数组内存空间
        for(int i=0; i<m; i++)
            for(int j=0; j<n; j++)
            {
                table[i][j]=(int)(Math.random() * 100); //元素随机赋值
            }
    }

    public MyMatrix(int n)                              //构造方法，构造方阵
    {
        height = n;
        width = n;
        this.init(height,width);                        //调用元素随机赋值方法
    }
```

```
public MyMatrix(int m,int n)                      //构造方法，构造 m 行 n 列矩阵
{
    height=m;
    width=n;
    this.init(height,width);                      //调用元素随机赋值方法
}

public int getHeight()                            //返回矩阵的行数方法
{
    return height;
}

public int getWidth()                             //返回矩阵的列数方法
{
    return width;
}

public int[][] getTable()                         //返回矩阵方法
{
    return table;
}

public MyMatrix add(MyMatrix b)                   //矩阵加方法
{
    if(this.getHeight()!=b.getHeight()&&
            this.getWidth()!=b.getWidth())
    {
        System.out.println("the two matrix don't macth");
        return null;
    }
    MyMatrix result=new MyMatrix(b.getHeight(),b.getWidth());
    for(int i=0;i<b.getHeight();i++)
        for(int j=0;j<b.getWidth();j++)
        {
            result.table[i][j]=this.table[i][j]+b.table[i][j];
            }
        return result;
}

public MyMatrix subtract(MyMatrix b)              //矩阵减方法
{
    if(this.getHeight()!=b.getHeight()&&
```

```
            this.getWidth()!=b.getWidth())
        {
            System.out.println("the two matrix don't macth");
            return null;
        }

        MyMatrix result=new MyMatrix(b.getHeight(),b.getWidth());
        for(int i=0;i<b.getHeight();i++)
            for(int j=0;j<b.getWidth();j++)
            {
                result.table[i][j]=this.table[i][j]-b.table[i][j];
            }
        return result;
    }
}

// TestMyMatrix.java 文件
public class TestMyMatrix                                        //测试类
{
    public static void main(String[] args)
    {
        MyMatrix mm1=new MyMatrix(4,4);
        MyMatrix mm2=new MyMatrix(4,4);
        MyMatrix mm3=new MyMatrix(4,5);
        MyMatrix mm4=new MyMatrix(4,5);

        MyMatrix add_result=mm1.add(mm2);
        int[][] add_table=add_result.getTable();
        MyMatrix subtract_result=mm3.subtract(mm4);
        int[][] subtract_table=subtract_result.getTable();

        System.out.println("two matrix add result：");
        for(int i=0;i<add_result.getHeight();i++)
        {
            for(int j=0;j<add_result.getWidth();j++)
            {
                System.out.print(add_table[i][j]+"    ");
            }
            System.out.println();
        }

        System.out.println("two matrix subtract result：");
```

```
            for(int i=0;i<subtract_result.getHeight();i++)
           {
                    for(int j=0;j<subtract_result.getWidth();j++)
                    {
                        System.out.print(subtract_table[i][j]+"    ");
                    }
                    System.out.println();
            }
        }
    }
```

程序运行结果如下：

```
two matrix add result：
67   94   130   78
21   171   78   104
47   100   84   111
125   152   98   61
two matrix subtract result：
–15   88   37   –25   –21
56    –5   32   40    41
–56   31   –75   –21   –4
–17   –46   –18   2   –28
```

习题 3

一、基本概念题

3.1 **什么叫类？什么叫对象？**

3.2 **对象是怎样得到内存空间的？垃圾对象是怎样回收的？**

3.3 **什么叫引用类型？对象是引用类型吗？**

3.4 **类的修饰符共有几种？分别是什么？**

3.5 **共有几种形式的变量？这些变量各自的用途是什么？**

3.6 **成员变量和成员方法的修饰符共有几种？各自的访问权限是什么？**

3.7 **同一个包中类的成员（包括成员变量和方法）用什么修饰符修饰，就只可以互相访问？**

3.8 **类的成员（包括成员变量和方法）的** protected **访问权限是什么？**

3.9 **什么叫实例成员变量？什么叫类成员变量？各有什么用途？**

3.10 **什么叫实例方法？什么叫类方法？设计类方法时有什么要求？**

3.11 **什么叫方法的重写？构造方法可以重写吗？方法重写在面向对象程序设计中有什么意义？**

3.12 **叙述编写和运行带有自定义包** Java **程序的操作步骤。**

3.13 **什么叫内部类**?

3.14 **什么叫类的封装性？类的封装性在面向对象程序设计中有什么意义？**

二、程序设计题

3.15　**对于下面设计的类** A**，哪一个方法是构造方法？**

```
public class A
{
public void A () { }
public class A() { }
public static class A () { }
public static void class A() { }
}
```

3.16　**下面哪些** main()**方法是不正确的？**

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

3.17　**设计一个复数类。要求复数类包括实数和虚数两个成员变量，同时类中应包含复数运算的各种方法。例如，方法应包括两个复数的加、减、乘、除等。复数的格式应该是：实数**+**虚数。最后，编写一个测试程序进行测试。**

3.18　**设计一个日期类，其输出格式是：“月**/**日**/**年”或“**June 13,1993**”。利用构造方法重写技术设计适合上面输出格式的构造方法。类中的输出方法也要利用方法重写技术来满足上述的输出格式。最后，编写一个测试程序来测试所定义的日期类能否实现预定的功能。**

3.19　**设计一个分数类。要求分数类包括分子和分母两个成员变量，同时类中应包含分数运算的各种方法。例如，方法应包括两个分数的加、减、乘、除等。分数的格式应该是：分子**/**分母。最后，编写一个测试程序进行测试。**

3.20　**设计一个电视机类。成员变量包括商品编号、商品型号、生产厂家、大小、重量、开关状态等，同时设计一些方法对电视机的状态进行控制。例如，方法应包括开**/**关电视机、更换频道、提高**/**减小音量等。要求商品编号自动生成。**

注意：有些成员变量应定义成静态的（static**），控制和操纵静态成员变量的方法应是静态的（**static**）。**

3.21　**编写一个基本账户类。成员变量包含账号、储户姓名和存款余额等。方法有存款和取款等。编写一个测试程序来测试所定义的账户类能否实现预定的功能。**

3.22　**设计一个长方形类。成员变量包括长度和宽度。类中除了包含计算周长和面积的方法外，还应该能够用** set **方法来设置长方形的长度和宽度，以及能够用** get **方法来获得长方形的长度和宽度。最后，编写一个测试程序来测试所定义的长方形类能否实现预定的功能。**

要求：使用自定义包方法。

3.23　**设计一个日期类，其输出格式是“月**/**日**/**年”或“**June 13,1993**”，并编写一个测试程序来测试所定义的日期类能否实现预定的功能。**

要求：把所设计的日期类作为测试类的内部类。

3.24　**设计一个汽车类，其成员变量包括颜色、品牌、车门、车灯、行驶速度等。其方法包括：打开车门、打开车灯、加速、减速等。可以根据自己对汽车的了解来设计类中的成员变量和方法。在方法中用输出方法名称的方式来表示这个方法已被调用。最后，编写一个测试程序来测试所定义的汽车类能否实现预定的功能。**

第 4 章　类 与 继 承

教学要点

本章内容主要包括面向对象程序设计的继承和多态概念，继承的设计方法，子类方法的三种继承形式，抽象类和最终类，接口（包括定义接口和实现接口）。

要求理解面向对象方法具有的继承性特点和多态性特点，熟练掌握子类的设计方法、子类方法的三种继承形式、接口的设计方法，掌握抽象类和最终类的设计方法。

类具有继承性。子类对父类的继承关系体现了现实世界中特殊和一般的关系。类的继承性大大简化了程序设计的复杂性。和类的继承性相联系的对象的动态绑定使对象的方法具有多态性。抽象类和最终类是两种特殊的类。接口和抽象类非常类似，Java 语言只支持单继承，但接口使 Java 语言实际上实现了多继承。

4.1　面向对象的基本概念：继承

继承是面向对象程序设计的又一个重要特性。继承体现了类与类之间的一种特殊关系，即一般与特殊的关系。继承就是一个新类拥有全部被继承类的成员变量和方法。继承机制使得新类不仅有自己特有的成员变量和方法，而且有被继承类的全部成员变量和方法。通过继承，可以从已有类模块产生新的类模块，从而使两个类模块之间发生联系。通过继承产生的新的类模块不仅重用了被继承类的模块资源，而且使两个类模块之间的联系方式和人类认识客观事物的方式一致。

面向对象程序设计的继承特性使得大型应用程序的维护和设计变得更加简单。一方面，大型应用程序设计完成并交付使用后，经常面临用户的需求发生变化，程序功能需要扩充等问题。这时，程序的修改需要非常谨慎，因为某个局部的修改可能会影响其他部分，而一个正在使用中的系统要进行全面的测试，则既费时间又有很多实际的困难。另一方面，一个新的应用系统程序设计问题，在许多方面会和以前设计过的某个或某些系统的模块非常类似，怎样加快大型应用程序的开发速度，重用这些已经开发成功的程序模块，一直是软件设计中迫切需要解决的问题。

传统的软件设计解决上述两类问题的方法主要有两种：

① 对于程序功能扩充问题，通常是直接对源代码进行改动。这种方法虽然可行，但有可能对正在使用的其他模块产生影响，可通过测试的方法消除这种影响。但是，要对一个正在使用的系统进行全面测试，既非常困难，代价又很大。

② 对于模块重用问题，通常是对原模块进行复制。对复制的模块再根据需要进行改动，

以支持新的功能。这种方法虽然可行，但仍然需要设计人员做很多工作，而且需要重新测试。

面向对象程序设计的继承机制可以很好地解决上述两类问题。面向对象程序设计的继承机制提供了一种重复利用原有程序模块资源的途径。通过新类对原有类的继承，既可以扩充旧的程序模块功能以适应新的用户需求，也可以满足新的应用系统的功能要求。从而既可以大大方便原有系统的功能扩充，也可以大大加快新系统的开发速度。另外，用这种软件设计方法设计的新系统较用传统的软件方法设计的新系统，需要进行的测试工作少很多。

4.2　继承

4.2.1　子类和父类

利用面向对象程序设计的继承机制，可以首先创建一个包括其他许多类共有的成员变量和方法的一般类，然后再通过继承创建一个新类。由于继承，这些新类已经具有了一般类的成员变量和方法，此时只需再设计各个不同类特有的成员变量和方法。由继承而得到的新类称为子类，被继承的类称为父类或超类。子类直接的上层父类称作直接父类。Java 不支持多继承，即一个子类只能有一个直接父类。

例如，设父类 super 已经定义，当类 sub1 继承类 super 时，就表明类 sub1 是类 super 的子类。或者说，类 super 是类 sub1 的父类。子类 sub1 由两部分组成：继承部分和增加部分。继承部分是从父类 super 继承过来的，增加部分是子类 sub1 新增加的。这样，子类继承了父类的成员变量和方法，从而可以共享已设计完成的软件模块。不仅如此，父类 super 还可以作为多个子类的父类，如子类 sub2 也是父类 super 的子类。由于子类 sub1 和 sub2 有相同的父类，所以它们既有许多相同的性能，也有一些不同的功能。父类和子类之间的继承关系如图 4.1 所示。

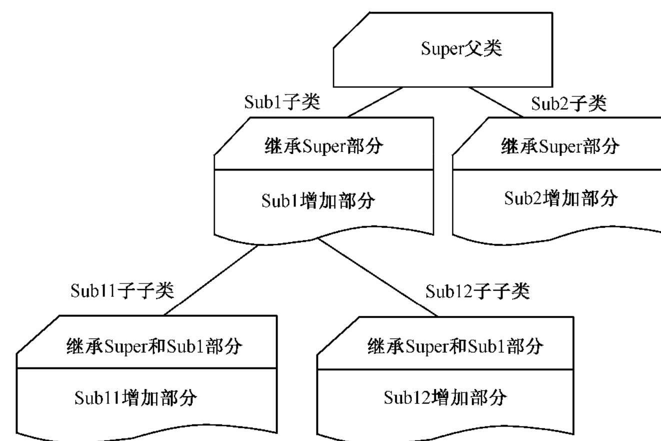


图 4.1　类的继承关系

从图 4.1 可知，具有继承关系的若干个类组成一棵类树。由于 Java 中所有的类都是从 Object 类继承（或称派生）来的，所以，Java 中所有的类构成一棵类树。

注意：在图 4.1 所示的有三层继承关系的类中，最下层的 Sub11 和 Sub12 子子类，不仅继承了直接父类 Sub1 的成员变量和方法，而且继承了间接父类 Super 的成员变量和方法。

如果没有继承机制，则一个软件系统中的各个类是各自封闭的、相互无关的。当多个类需要实现相似的功能时，势必会造成成员变量和方法的大量重复。而有了继承机制，多个类就可相互关联，新类就可以从已有的类中通过继承产生。

继承有两种基本形式：多继承和单继承。多继承是指一个子类可以继承自多个直接父类。单继承是指一个子类只可以继承自一个直接父类。Java 语言只允许单继承，不允许多继承。

4.2.2 创建子类

Java 中的类都是 Object 类的子类（当然，很多类是 Object 类的间接子类）。Object 类定义了所有对象都必须具有的基本成员变量和方法。Java 中的每个类都从 Object 类继承了成员变量和方法，因而 Java 中的所有对象都具有 Object 类的成员变量和方法。

由于 Java 中的所有类都是 Object 的直接子类或间接子类，所以 Java 中的所有类构成一棵类的层次树结构。

定义类有两种基本方法：不指明父类和显式地指明父类。Java 语言规定，若定义类时不指明父类，则其父类是 Object 类。第 3 章定义的类都没有指明父类，所以第 3 章中定义的类都是 Object 类的子类。本节介绍显式地指明父类的类定义方法。

显式地指明一个类的父类的方法是，在类定义时使用关键字 extends，并随后给出父类名。类定义语句格式为：

```
[〈修饰符〉] class 〈子类名〉 extends 〈父类名〉
```

例如，

```
class Sub1 extends Super
```

就定义类 Sub1 继承自类 Super。此时我们说类 Sub1 是类 Super 的子类，或者说类 Super 是类 Sub1 的直接父类，直接父类通常简称为父类。

1．子类继承父类的成员变量

子类继承了父类中的成员变量。具体的继承原则是：

① 能够继承父类中那些声明为 public 和 protected 的成员变量。

② 不能继承父类中那些声明为 private 和默认的成员变量。

③ 如果子类声明一个与父类成员变量同名的成员变量，则不能继承父类的同名成员变量。此时，称子类的成员变量隐藏了父类中的同名成员变量。

因此，如果父类中存在不允许其子类访问的成员变量，那么这些成员变量必须以 private 修饰符声明该成员变量；如果父类中存在只允许其子类访问、不允许其他类访问的成员变量，那么这些成员变量必须以 protected 修饰符声明该成员变量。

2．子类继承父类的方法

子类继承父类方法的原则类似于子类继承父类成员变量的原则。具体的继承原则是：

① 能够继承父类中那些声明为 public 和 protected 的方法。

② 不能继承父类中那些声明为 private 和默认的方法。

③ 如果子类方法与父类方法同名，则不能继承。此时称子类方法重写了父类中的同名方法。

④ 不能继承父类的构造方法。

注意：和子类继承父类成员变量的继承原则不同的是，子类不能继承父类的构造方法。

3．this 引用和 super 引用

（1）this 引用

Java 中，每个对象都具有对其自身引用的访问权，这称为 this 引用。访问本类的成员变量和方法的语句格式为：

```
this.〈成员变量名〉
this.〈方法名〉
```

例如，下面定义的类 X 中有成员变量 k，而在方法 D 中也用 k 作参数，这样两个不同含义的变量 k 就有可能产生混淆，此时必须用 this.k 指代对象的成员变量 k。

```
class X
{
    int k；
    void D(int k)
    {
        this.k = 2*k；                    //this.k 指成员变量 k，k 指参数 k
    }
}
```

（2）super 引用

使用关键字 super 可以引用被子类隐藏的父类的成员变量或方法，这称为 **super 引用**。super 引用的语句格式为：

```
super.〈成员变量名〉
super.〈方法名〉
```

super 引用经常用在子类的构造方法中。前面说过，子类不能继承父类的构造方法，但有时子类的构造方法和父类的构造方法相同时，或子类的构造方法只需在父类构造方法的基础上做某些补充时，子类构造方法中需要调用父类的构造方法时，此时的语句格式为：

```
super（〈参数列表〉）
```

其中，〈参数列表〉是调用父类构造方法所需的参数。

4．成员变量和方法的隐藏与覆盖

子类除了可以继承父类中的成员变量和方法外，还可以增加自已特有的成员变量和方法。当父类的某个成员变量不适合子类时，子类可以重新定义该成员变量。前面说过，此种情况下，子类隐藏了父类的成员变量（程序设计中这种情况很少，一般也不提倡）；当父类的某个方法不适合子类时，子类可以重新定义它，这称为子类对父类方法的**覆盖**（overriding）。

子类对父类方法的覆盖是面向对象程序设计中经常使用的设计方法。在软件功能扩充和软件重用中，可以通过设计新的子类，以及通过子类方法对父类方法的覆盖，可以方便、快速地实现软件功能的扩充和软件的重用。

注意：方法的重写（overloading）和方法的覆盖（overriding）是两个不同的概念，在软件设计中实现的功能也不同。

5．举例

【例 4.1】　继承举例。

要求：设计一个 Shape（形状）类，再设计 Shape 类的两个子类，一个是 Ellipse（椭圆）类，另一个是 Rectangle（矩形）类。每个类都包括若干成员变量和方法，但每个类都有一个 draw()方法（画图方法），draw()方法中用输出字符串表示画图。

程序设计如下：

```
class Shape                                         //定义父类 Shape
{
    protected int lineSize;                         //线宽

    public Shape()                                  //构造方法 1
    {
        lineSize = 1;
    }

    public Shape(int ls)                            //构造方法 2
    {
        lineSize = ls;
    }

    public void setLineSize(int ls)                 //设置线宽
    {
        lineSize = ls;
    }

    public int getLineSize()                        //获得线宽
    {
        return lineSize;
    }

    public void draw()                              //画图
    {
        System.out.println("Draw a Shape");
    }
}

class Ellipse extends Shape                         //定义子类 Ellipse
{
```

```
    private int centerX;                            //圆心 X 坐标
    private int centerY;                            //圆心 Y 坐标
    private int width;                              //椭圆宽度
    private int height;                             //椭圆高度

    public Ellipse(int x, int y, int w, int h)      //构造方法
    {
        super();                                    //调用父类的构造方法 1
        centerX = x;
        centerY = y;
        width = w;
        height = h;
    }

    public void draw()                              //覆盖父类的 draw()方法
    {
        System.out.println("draw a Ellipse");
    }
}

class Rectangle extends Shape                       //定义子类 Rectangle
{
    private int left;                               //矩形左上角 X 坐标
    private int top;                                //矩形左上角 Y 坐标
    private int width;                              //矩形长度
    private int height;                             //矩形宽度

    public Rectangle(int l, int t, int w, int h)    //构造方法
    {
        super(2);                                   //调用父类的构造方法 2
        left = l;
        top = t;
        width = w;
        height = h;
    }

    public void draw()                              //覆盖父类的 draw()方法
    {
        System.out.println("draw a Rectangle");
    }
}
```

```
public class Inherit                                    //定义类 Inherit
{
    public static void main(String args[])
    {
        Ellipse ellipse = new Ellipse(30, 30, 50,60);
                                                        //创建子类 Ellipse 的对象
        ellipse.setLineSize(2);
                                                        //调用父类方法重新设置 lineSize 值为 2
        System.out.println("LineSize of ellipse : " +
                    ellipse.getLineSize());

        Rectangle rectangle = new Rectangle(0, 0, 20, 30);
                                                        //创建子类 rectangle 对象
        rectangle.setLineSize(3);
                                                        //调用父类方法重新设置 lineSize 属性为 3
        System.out.println("LineSize of rectangle : " +
                    rectangle.getLineSize());
        ellipse.draw();                                 //访问子类方法
        rectangle.draw();                               //访问子类方法
    }
}
```

程序运行结果如下：

```
LineSize of ellipse : 2
LineSize of rectangle : 3
draw a Ellipse
draw a Rectangle
```

程序设计说明：

① 类 Shape 中定义了所有子类共同的成员变量 lineSize（线宽），椭圆类 Ellipse 和矩形类 Rectangle 在继承父类成员变量的基础上，又各自定义了自己的成员变量。

② 父类 Shape 中定义了画图方法 draw()，子类 Ellipse 和子类 Rectangle 中由于各自形状不同，画图方法 draw()也不同，所以子类 Ellipse 和 Rectangle 中重新定义了各自的 draw()方法（即覆盖了父类的 draw()）。注意：子类覆盖父类方法时，参数个数和参数类型必须相同。

③ 当一个文件中包含有多个类时，源程序文件名应该和定义为 public 的类名相同。

4.2.3　方法的三种继承形式

上节讨论了子类对父类方法继承的一般形式，本节我们进一步总结子类对父类方法继承的三种不同形式，以及系统中子类对象访问方法的匹配原则和继承在面向对象程序设计中的作用。

1．方法的三种继承形式

子类对父类方法继承可以有三种不同形式：完全继承、完全覆盖和修改继承。

（1）完全继承

完全继承是指子类全部继承父类的方法。如果父类中定义的方法完全适合于子类，则子类就不需要重新定义该方法。子类对父类的继承允许子类对象直接访问父类的方法，这就是子类对父类方法的完全继承。如例 4.1 中，如果子类不重新定义 draw()方法，则 ellipse.draw()和 rectangle.draw()访问的都是父类中定义的 draw()方法。

（2）完全覆盖

完全覆盖是指子类重新定义父类方法的功能，从而子类中的同名方法完全覆盖了父类中的方法。

如例 4.1 中，子类重新定义了父类的 draw()方法，因此子类对象 ellipse 和 rectangle 访问的就是子类中重新定义的方法 draw()，即 ellipse.draw()和 rectangle.draw()访问的都是子类中定义的 draw()方法。

（3）部分继承

部分继承是指子类覆盖父类的方法，但子类重新定义的方法中调用父类中的同名方法，并根据问题要求做部分修改。

【例 4.2】　部分继承举例。

```
class Shape                                    //定义父类 Shape
{
    public void draw()
    {
        System.out.println("Draw a Shape");
    }
}

class Ellipse extends Shape                    //定义子类 Ellipse
{
    public void draw()                         //覆盖父类的 draw()方法
    {
        super.draw();
        System.out.println("draw a Ellipse");
    }
}

class Rectangle extends Shape                  //定义子类 Rectangle
{
    public void draw()                         //覆盖父类的 draw()方法
    {
        super.draw();                          //调用父类的 draw()方法
        System.out.println("draw a Rectangle"); //修改部分
    }
}
```

```
public class CInherit                              //定义类 Inherit
{
    public static void main(String args[])
    {
        Ellipse ellipse = new Ellipse();           //创建子类 Ellipse 的对象
        Rectangle rectangle = new Rectangle();     //创建子类 rectangle 对象

        ellipse.draw();                            //访问子类方法
        rectangle.draw();                          //访问子类方法
    }
}
```

程序运行结果如下：

```
Draw a Shape
draw a Ellipse
Draw a Shape
draw a Rectangle
```

程序设计说明：

① 子类的 draw()覆盖了父类的 draw()方法，但子类的 draw()方法首先调用了父类的 draw()方法。子类 draw()方法调用父类 draw()方法的语句是：

```
super.draw();
```

② 由于子类方法在调用父类方法的基础上，又增加了子类中需要补充修改的功能，所以子类对象 ellipse 和 rectangle 访问的 draw()方法，完成的功能是在父类方法基础上的补充或修改。

2．系统中子类对象访问方法的匹配原则

在 Java 语言（以及在所有的面向对象程序设计语言）中，对象访问方法的匹配原则是：从对象定义的类开始，逐层向上匹配寻找对象要访问的方法。

在完全继承方式中，由于子类中没有定义 draw()方法，所以系统自动到它的直接父类 Shape 中去匹配 draw()方法，系统在父类 Shape 中匹配上了 draw()方法，所以子类对象 ellipse 和 rectangle 访问的是父类定义的 draw()方法。在完全覆盖方式中，由于子类中定义了 draw()方法，所以子类对象 ellipse 和 rectangle 访问的是子类定义的 draw()方法。在修改继承方式中，由于子类中定义了 draw()方法，所以子类对象 ellipse 和 rectangle 访问的是子类定义的 draw()方法，由于子类定义的 draw()方法首先调用了父类中定义的 draw()方法，然后又增加了需要修改或补充的功能，所以子类对象 ellipse 和 rectangle 访问的 draw()方法，既包含了父类 draw()方法的功能，又包含了子类修改或补充的功能。

3．继承在面向对象程序设计中的作用

继承在面向对象程序设计中有两方面的意义：一方面，继承性可以大大简化程序设计的

代码。可以把若干个相似类所具有的共同成员变量和方法定义在父类中，这样这些子类的设计代码就可以大大减少。

另一方面，继承（特别是部分修改继承和完全覆盖继承）使得大型软件的功能修改和功能扩充较传统的软件设计方法容易了许多。当要对系统的一些原有功能进行补充性修改或添加一些新的功能时，可以重新设计原先类的一个子类，利用部分修改继承方法重新设计子类中要补充性修改或添加的功能；当要废弃系统的一些原有功能，重新设计完全不同的新的功能时，可以重新设计原先类的一个子类，利用完全覆盖继承方法重新设计子类中的功能。

继承性是面向对象方法的一个非常重要的特点。这是因为继承性使得我们可以根据问题的特征，把若干个类设计成继承关系。而类的继承关系和人类认识客观世界的过程和方法基本吻合，从而使得人们能够用和认识客观世界一致的方法来设计软件。

4.2.4　方法的多态性

1．对象的动态绑定和方法的多态性

方法的多态性是面向对象程序的另一个重要特点。方法的**多态**（polymorphism）是指若以父类定义对象，并动态绑定对象，则该对象的方法将随绑定对象不同而不同。

我们把图 3.1 所示的对象的存储结构重新给出。在只定义对象、没有分配内存空间时，如图 4.2（a）所示，对象名中并没有存放实际对象的首地址。在为已定义的对象分配了内存空间后，如图 4.2（b）所示，对象名中存储的就是对象的内存空间的首地址。对象名和实际对象的这种联系称作**对象的绑定**（binding）。

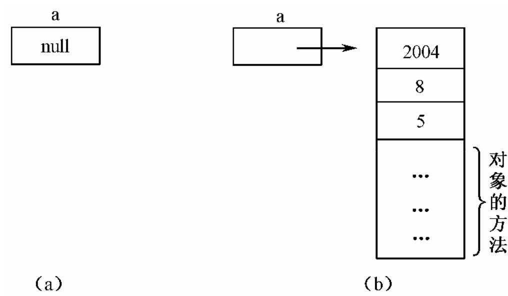


图 4.2　对象的存储结构

Java 语言还支持对象的动态绑定。所谓**对象的动态绑定**，是指定义为类树上层的对象名，可以绑定为所定义层类以及下层类的对象。这样，当对象动态绑定为哪一层子类对象时，其方法就调用那一层子类的方法。因此，对象的动态绑定和类的继承相结合就使对象的方法具有多态性。

【例 4.3】　方法的多态性示例。

```
class Shape                                    //定义父类 Shape
{
    public void draw()                         //父类的 draw()方法
    {
        System.out.println("Draw a Shape");
    }
```

```
}

class Circle extends Shape                    //定义子类 Circle
{
    public void draw()                        //覆盖父类的 draw()方法
    {
        System.out.println("draw a Circle");
    }
}

class Ellipse extends Circle                  //定义子类 Ellipse
{
    public void draw()                        //覆盖父类的 draw()方法
    {
        System.out.println("draw a Ellipse");
    }
}

public class FInherit                         //定义类 FInherit
{
    public static void main(String args[])
    {
        Shape s= new Shape();                 //动态绑定为类 Shape 对象
        Shape c = new Circle();               //动态绑定为类 Circle 对象
        Shape e = new Ellipse();              //动态绑定为类 Ellipse 对象

        s.draw();                             //访问父类方法
        c.draw();                             //访问一级子类方法
        e.draw();                             //访问二级子类方法
    }
}
```

程序运行结果如下：

```
Draw a Shape
draw a Circle
draw a Ellipse
```

程序说明：

① 类 Shape 是父类，类 Circle 是类 Shape 的直接子类，类 Ellipse 是类 Circle 的直接子类。这三个类中都定义了 draw()方法。子类中的 draw()方法覆盖了父类中的同名方法。

② FInherit 类的 main()方法中，定义了三个对象 s、c 和 e。三个对象都定义为 Shape 类，但对象 s 动态绑定为 Shape 类的对象，对象 c 动态绑定为 Circle 类的对象，对象 e 动态绑定为 Ellipse 类的对象。这样，语句 s.draw()调用的就是 Shape 类的方法 draw()，语句 c.draw()

调用的就是 Circle 类的方法 draw()，语句 e.draw()调用的就是 Ellipse 类的方法 draw()。

2．方法多态性的用途

方法的多态性在程序设计中非常有用。例如，第 5 章我们将介绍 Java API 语言包的 Vector 类，Vector 类中定义的一个方法如下：

```
copyInto(Object[] anArray)
```

该方法的功能是把当前对象的一个成分复制给对象数组 anArray。其参数 anArray 定义为 Object 类的数组，由于 Object 类是所有类的根（即最上层的类），所以，该方法可用于任何类的对象。例如，程序中可以像下面这样使用 Vector 类的 copyInto()方法：

```
Vector v = new Vector();                    //定义并创建 Vector 类的对象 v
String[] s = new String[v.size()];          //定义并创建 String 类的对象 s
v.copyInto(s);                              //把对象 v 的当前成分复制给对象 s
```

上面语句段的最后一句将把对象 v 的当前成分复制给对象 s。如果没有方法的多态性，若要定义 Vector 类的 copyInto()方法适合所有类的对象时，就要把该方法用不同类的参数重载很多个；而方法的多态性支持 Vector 类的 copyInto()方法用 Object 类参数(Object [] anArray)定义一次，就可以适合于所有类的对象了。

4.3 抽象类和最终类

在类的定义中，除了可说明该类的父类外，还可以说明该类是否是最终类或抽象类。

4.3.1 抽象类

类中允许定义抽象方法。所谓抽象方法是指只有方法的定义，没有方法的实现体的方法。Java 语言用关键字 abstract 来声明抽象方法。例如，

```
abstract    void    draw()
```

则声明类中的 draw()方法为抽象方法。但是，需要说明的是：

① 构造方法不能被声明为抽象的。

② abstract 和 static 不能同时存在，即不能有 abstract static 方法。

包含抽象方法的类称为抽象类。换句话说，任何包含抽象方法的类必须被声明为抽象类。因为抽象类中包含没有实现的方法，所以抽象类是不能直接用来定义对象的。Java 语言用关键字 abstract 来声明抽象类，例如，

```
abstract class Shape
```

声明类 Shape 为抽象类。

在程序设计中，抽象类主要用于定义为若干个功能类同的类的父类。

【例 4.4】　抽象类举例。

问题描述：设计椭圆类 Ellipse 和矩形类 Rectangle，要求这两个类都包含一个画图方法 draw()。

设计分析：椭圆类 Ellipse **和矩形类** Rectangle **有许多成员变量和方法相同。因此，可以先设计一个它们的共同的父类（也称基类）**Shape**，并把画图方法** draw()**定义在父类中。但是，由于父类** Shape **只是抽象的形状，画图方法** draw()**无法实现，所以父类中的画图方法** draw() **只能定义为抽象方法，而包含抽象方法的** Shape **类也只能定义为抽象类。**

```
abstract class Shape                          //定义抽象类 Shape
{
    public abstract void draw();              //定义抽象方法
}

class Ellipse extends Shape                   //定义子类 Ellipse
{
    public void draw()                        //实现 draw()方法
    {
        System.out.println("draw a Ellipse");
    }
}

class Rectangle extends Shape                 //定义子类 Rectangle
{
    public void draw()                        //实现 draw()方法
    {
        System.out.println("draw a Rectangle");
    }
}

public class AInherit                         //定义类 AInherit
{
    public static void main(String args[])
    {
        Ellipse ellipse = new Ellipse();          //创建子类 Ellipse 的对象
        Rectangle rectangle = new Rectangle();    //创建子类 rectangle 对象

        ellipse.draw();                           //访问子类 ellipse 的方法
        rectangle.draw();                         //访问子类 rectangle 的方法
    }
}
```

上述例子说明：

（1）在一个软件中，抽象类一定是某个类或某些类的父类。

（2）若干个抽象类的子类要实现一些同名的方法。

在后面讨论的 Java API **中，系统的许多类都是用上面形式的结构定义和实现的。**

4.3.2　最终类

最终类是指不能被继承的类，即不能再用最终类派生子类。在 Java 语言中，如果不希望某个类被继承，可以声明这个类为最终类。最终类用关键字 final 来说明。例如，

```
public final class C
```

就定义类 C 为最终类。

如果没必要创建最终类，而又想保护类中的一些方法不被覆盖，可以用关键字 final 来指明那些不能被子类覆盖的方法，这些方法称为最终方法。例如：

```
public class A
[
    public final void M();
}
```

就在类 A 中定义了一个最终方法 M()，任何类 A 的子类都不能重新定义方法 M()。

在程序设计中，最终类可以保证一些关键类的所有方法，不会在以后的程序维护中，由于不经意地定义子类而被修改；最终方法可以保证一些类的关键方法，不会在以后的程序维护中，由于不经意地定义子类和覆盖子类的方法而被修改。

需要注意的是：一个类不能既是最终类又是抽象类，即关键字 abstract 和 final 不能合用。在类声明中，如果需要同时出现关键字 public 和 abstract（或 final），习惯上，public 放在 abstract（或 final）的前面。

4.4　接口

面向对象程序设计语言的一个重要特性是继承。继承是指子类可以继承父类的成员变量和方法。如果子类只允许有一个直接父类，这样的继承称作单继承。如果子类允许有一个以上的直接父类，这样的继承称作多继承。单继承具有结构简单，层次清楚，易于管理，安全可靠的特点。多继承具有功能强大的特点。

Java 语言只支持单继承机制，不支持多继承。一般情况下，单继承就可以解决大部分子类对父类的继承问题。但是，当问题复杂时，若只使用单继承，可能会给设计带来许多麻烦。Java 语言解决这个问题的方法是使用接口。

接口和抽象类非常相似，都是只定义了类中的方法，没有给出方法的实现。

Java 语言不仅规定一个子类只能直接继承自一个父类，同时允许一个子类可以实现（也可以说继承自）多个接口。由于接口和抽象类的功能类同，因此，Java 语言的多继承机制是借助于接口来实现的。

4.4.1　定义接口

接口的定义格式为：

```
〈修饰符〉interface〈接口名〉
{
```

```
        成员变量1 = 〈初值1〉;
        成员变量2 = 〈初值2〉;
        ⋮
        方法1;
        方法2;
        ⋮
    }
```

其中，〈修饰符〉可以是 public，也可以缺省。当缺省时，接口只能被与它处在同一包中的方法访问；当声明为 public 时，接口能被任何类的方法访问。〈接口名〉是接口的名字，可以是任何有效的标识符。例如，

```
public interface PrintMessage
{
    public int count = 10;
    public void printAllMessage();
    public void printLastMessage();
    public void printFirstMessage();
}
```

就定义了一个接口 PrintMessage。接口中的方法（printAllMessage()等）只有方法定义，没有方法实现。所以接口实际上是一种特殊的抽象类。

需要说明的是：

① 若接口定义为默认型访问权限，则接口中的成员变量全部隐含为 final static 型。这意味着它们不能被实现接口方法的类改变，并且为默认访问权限。

② 接口中定义的所有成员变量都必须设置初值。

③ 若接口定义为 public 型访问控制，则接口中的方法和成员变量全部隐含为 public 型。

④ 当接口保存于文件时，其文件命名方法和保存类的文件命名方法类同。即保存接口的文件名必须与接口名相同。一个文件可以包含若干个接口，但最多只能有一个接口定义为 public，其他的接口必须为默认。

4.4.2 实现接口

一旦定义了一个接口，一个或更多的类就能实现这个接口。为了实现接口，类必须实现定义在接口中的所有方法。每个实现接口的类可以自由地决定接口方法的实现细节。

定义类时实现接口用关键字 implements。一个类只能继承一个父类，但可以实现若干个接口。因此，类定义的完整格式是：

[〈修饰符〉] class〈类名〉[extends〈父类名〉] [implements〈接口名1〉,〈接口名2〉, ……]

其中，关键字 implements 后跟随的若干个接口名表示该类要实现的接口；如果要实现多个接口，则用逗号分隔开接口名。

【例 4.5】 编写一个实现 4.4.1 节定义的接口 PrintMessage（为简化设计代码，去掉其中的成员变量定义）的类，并编写一个测试程序进行测试。

程序设计如下：

```
//接口文件 PrintMessage.java
public interface PrintMessage
{
    public int count = 10;
    public void printAllMessage();
    public void printLastMessage();
    public void printFirstMessage();
}

//实现接口的文件 MyInter.java
public class MyInter implements PrintMessage            //实现接口的类 MyInter
{
    private String[] v;                                 //类中的成员变量 v
    private int i;                                      //类中的成员变量 i

    public MyInter()                                    // MyInter 类的构造方法
    {
        v = new String[3];
        i = 0;
        this.putMessage("Hello world!");                //使用 MyInter 类的方法
        this.putMessage("Hello China!");
        this.putMessage("Hello XSYU!");
    }

    public void putMessage(String str)                  //类中的方法
    {
        v[i++] = str;
    }

    public void printAllMessage()                       //实现接口中的方法
    {
        for(int k = 0; k < v.length; k++)
        {
        System.out.println(v[k]);
        }
    }
    public void printLastMessage()                      //实现接口中的方法
    {
        System.out.println(v[v.length - 1]);
    }

      public void printFirstMessage()                   //实现接口中的方法
```

```
    {
        System.out.println(v[0]);
     }

    public static void main(String[] args)
    {
        MyInter mi=new MyInter();                     //定义 MyInter 类的对象
        System.out.println("print all messages");
        mi.printAllMessage();                         //使用实现了的接口方法
        System.out.println("print the first messages");
        mi.printFirstMessage();                       //使用实现了的接口方法
        System.out.println("print the last messages");
        mi.printLastMessage();                        //使用实现了的接口方法
    }
}
```

程序的运行结果如下：

```
print all messages
Hello world!
Hello China!
Hello XSYU!
print the first messages
Hello world!
print the last messages
Hello XSYU!
```

程序说明：在定义类 MyInter 时，后边跟有 implements PrintMessage，表示该类中要实现接口 PrintMessage。此时类 MyInter 中必须实现接口 PrintMessage 中定义的三个方法。由于类 MyInter 隐含继承了类 Object，现在又实现了接口 PrintMessage，所以类 MyInter 是一个多继承。可见，接口支持了 Java 的多继承。

4.4.3　系统定义的接口

第 5 章开始要讨论 Java API，Java API 中定义了许多接口，一旦安装了 JDK 运行环境，就可以像使用用户自己定义的接口一样使用系统定义的接口。例如，Enumeration 是系统定义的一个接口。Enumeration 接口的定义如下：

```
public interface Enumeration
{
    Object nextElement();                         //返回后续元素
    boolean hasMoreElements();                    //是否还有后续元素
}
```

许多系统定义的类都实现了 Enumeration 接口。如有必要，用户自己定义的类也可以实现 Enumeration 接口。

习题 4

一、基本概念题

4.1 在如图 4.1 所示的类树中，Sub11 能继承父类 Sub1 的哪些部分？Sub11 能否继承 Super 的成员变量和方法？

4.2 子类对父类方法的继承有几种形式？各有什么用途？

4.3 方法的重写和方法的覆盖有什么不同？

4.4 什么叫对象的动态绑定？对象的动态绑定和继承相结合怎样实现方法的多态性？

4.5 方法的多态性在程序设计中有什么用途？

4.6 super 引用的语法格式是什么？怎样在子类的构造方法中调用父类的构造方法？

4.7 this 引用的语法格式是什么？怎样在类的方法中调用和参数同名的成员变量？

4.8 什么叫抽象类？抽象类有什么设计要求？

4.9 什么叫最终类？最终类有什么设计要求？

4.10 什么叫接口？接口和抽象类有什么相同之处和不同之处？

4.11 Java 语言支持多继承吗？Java 语言怎样解决多继承问题？

二、程序设计题

4.12 下面的类中，哪些方法是覆盖？哪些方法是重写？

```
Class Car
{
    public Car() {}
    public CarM(int c) {}
}
Class SportsCar extends Car
{
    public SportsCar() {}
    public SportsCar(int s) {}
    public CarM(int c) {}
}
```

4.13 设计一个点类，它仅包含两个属性：横坐标和纵坐标。通过继承点类再设计一个圆类，它除了有一个圆心，还有半径，还应该能够计算圆的周长和面积等。编写一个测试程序来测试所设计的类能否实现预定的功能。

4.14 设计一个动物类，它包含一些动物的属性，例如名称、大小、重量等，动物可以跑或走。然后设计一个鸟类，除了动物的基本属性外，它还有自己的羽毛、翅膀等，鸟除了跑或走外，它还可以飞翔。为了继承动物类的特性，鸟类应该继承动物类。编写一个测试程序来测试所设计的鸟类能否实现预定的功能。

4.15 先设计一个长方形类，再通过继承长方形类设计一个正方形类，正方形类中通过重写父类的方法得到一些新的功能。

4.16 先设计一个基本账户类，再通过继承基本账户类设计一个储蓄账户类，储蓄账户类中增加一个静

态成员变量（年利率），并增加如下方法：

（1）计算月利息——存款金额*年利率/12

（2）更改利率（静态方法）——重新设置年利率

最后，编写一个测试程序来测试所设计的储蓄账户类能否实现预定的功能。

4.17　先设计一个基本账户类，再通过继承基本账户类设计一个储蓄账户类，储蓄账户类中增加密码、地址、最小余额和利率等成员变量，并增加一些银行账户经常用到的方法，要求：

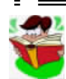
（1）类中的方法具有输入、输出储户上述信息的功能。

（2）将账号设计成不可更改，修改密码时要提供原密码。

4.18　在第 3 章习题 3.20 的电视机类的基础上，设计一台新型的纯平和超薄彩色电视机类，增加描述这些属性的成员变量和控制这些成员变量的方法。编写一个测试程序来测试所定义的新型电视机类能否实现预定的功能。

4.19　在第 3 章习题 3.24 的汽车类的基础上，设计一个跑车类，增加一些体现跑车特性的成员变量，例如，紧急制动装置、自动巡航状态、温控装置等，可以根据自己对跑车的了解来增加它的成员变量和方法。在方法中用输出方法名称的方式来表示这个方法被调用。最后，编写一个测试程序来测试所设计的跑车类能否实现预定的功能。

第 5 章　Java API 基础



教学要点

本章内容主要包括 Java API 的概念，语言包常用类（包括 Object 类、System 类、Class 类、Runtime 类、Float 类、String 类、Math 类）的功能和使用方法，实用包常用类和接口（包括 Arrays 类、Vector 类、Data 类、Enumeration 接口）的功能和使用方法。

要求理解 Java API，掌握语言包和实用包中常用类的功能和使用方法。

Java 语言的强大功能主要体现在 Java 语言完备丰富、功能强大的 Java API 上。本章介绍了 Java API 的基本结构，以及语言包和实用包中的常用类。后面讨论的每一章都还会涉及到 Java API 的不同包中的类。

5.1　Java API 综述

Java 语言的内核非常小，仅包含第 2 章讨论的 Java 语言的基本数据类型和语句。Java 语言的强大功能主要体现在 Java 语言完备丰富、功能强大的 Java API 上。

Java API（Java Application Programming Interface，Java 应用程序接口），是 Java 语言提供的组织成包结构的许多类和接口的集合。Java API 为用户编写应用程序提供了极大的便利。Java API 包含在 JDK 中，因此用户只要按照 1.3 节介绍的方法安装了 JDK 运行环境就可以使用了。

Java API 按照内容分别组织和存放在不同的包中，Java API 中的包及其主要功能按字母顺序说明如下：

java.accessibility	接口组件和助手技术的类和接口
java.applet	Applet 所需的类和接口
java.awt	图形用户界面所需的类和接口
java.beans	Java bean 所需的类和接口
java.io	系统输入/输出所需的类和接口
java.lang	Java 语言编程的基础类和接口
java.math	支持任意精度整数和任意精度小数的类和接口
java.naming	访问命名服务的类和接口
java.net	网络应用的类和接口
java.rmi	远程调用（RMI）的类和接口
java.security	用于安全框架的类和接口
java.sql	访问和处理数据源中数据的类和接口

java.text	支持按与语言无关方式处理文本、数据、数字和消息的类和接口
java.util	集合框架、事件模型、日期和时间机制、国际化等的类和接口
javax.rmi	支持 RMI-IIOP 的类和接口
javax.serverlet	支持 serverlet 编程的类和接口
javax.sound	支持音频设备数字接口（MIDI）的类和接口
javax.swing	扩充和增强基本图形用户界面功能的类和接口
javax.transaction	包含有几个关于事务上下文异常的类
org.omg.CORBA	支持 OMG CORBA API 到 Java 语言映射的类和接口

上述大部分的包都又按内容组织成子包形式，关于各包的子包这里就不再赘述。

Java 语言在不断发展，这表现在 JDK 运行环境的版本在不断提高。因此，读者学习本书时可能会发现，Java API 中又包含了新的包，或某些包中又增加了新的子包。

打开 JDK 的帮助文档，可以看到 Java API 的详细说明文档。

Java API 包含的内容很多，本章主要讨论其中两个最基本的包中的主要类：

① 语言包（java.lang）。主要讨论的类有：Object 类、Class 类、Runtime 类、Float 类、String 类和 Math 类。

② 实用包（java.util）。主要讨论的类和接口有：Arrays 类、Vector 类、Data 类和 Enumeration 接口。

其他本书涉及的 Java API 内容，将在本书的后续各章中介绍。

5.2 语言包（java.lang）简介

java.lang 包中包含了 Java 程序设计语言最基础的类。本节讨论的 java.lang 包中的类有 Object 类、System 类、Class 类、Runtime 类、String 类、Math 类和 Float 类。

java.lang 包是用 Java 语言编程时使用最频繁的包。为了简化编程，系统固定地默认导入了 java.lang 包，所以使用 java.lang 包中的类时可以不用 import 语句导入。

5.2.1 Object 类

Object 类是 Java 中所有类的根，所有其他的类都是由 Object 类派生出来的，因此，根据继承的特点，在 Object 类中定义的成员变量和方法，在其他类中都可以使用。

Object 类常用方法有：

⌘ equals(Object obj)	比较两个对象是否相等
⌘ getClass()	获取对象的运行时类
⌘ toString()	把对象转换为字符串

Object 类中还有一组关于线程同步的方法：wait()方法和 notify()方法，将在 10.7.2 节讨论同步线程的设计方法时介绍。

equals(Object obj)方法与 Java 运算符“= =”的含义相同，但用法不同。当两个值比较、对象或变量与值比较、两个变量比较时，使用运算符“= =”；当两个对象比较时，使用 equals()方法。该方法调用返回 true 时，表示两个对象相等（或称相同）；返回 false 时，表示两个对象不相等。

【例 5.1】 equals()方法应用举例。

```
public class EqualsTest
{
    public static void main(String args[])
    {
        char ch='A';
        if ((ch=='A') || (ch=='a'))                      //变量与值比较
                System.out.println("true");
        String str1="abc",str2=null;
        if (str1!=null)                                  //对象与值比较
                System.out.println(str1.equals(str2));   //两个对象比较
    }
}
```

程序运行显示结果如下：

```
true
false
```

5.2.2 System 类

System 类提供了许多获取或重新设置系统资源的静态方法。

System 类的常用方法有：

- ⌘ static Properties getProperty()　　获取系统属性
- ⌘ static Properties getProperty(String key)　　获取由 key 指定的系统属性
- ⌘ static void setProperty(Properties props)　　设置由 props 指定的系统属性
- ⌘ static void load(String fileName)
  加载本地文件系统中由文件名 fileName 指定的动态库
- ⌘ static void exit(int status)
  中断当前运行的 Java 虚拟机，status 为状态码，非 0 的状态码表示不正常中断

其中，获取系统属性方法的返回值类型为 Properties，Properties 是 java 语言包中定义的一个类。该类定义了系统属性集合，每个属性用字符串表示，其常用的几个属性值以及含义如下：

- ⌘ java.version　　java 运行时环境版本
- ⌘ java.vm.version　　java 虚拟机实现的版本
- ⌘ java.class.path　　java 类的路径
- ⌘ os.version　　操作系统的版本
- ⌘ user.name　　用户名
- ⌘ user.dir　　用户路径
- ⌘ user.home　　用户 home 路径

【例 5.2】　用 System 类获得当前系统属性示例。

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

```
    {
        String str;
        //java 运行时环境版本
        str = System.getProperty("java.version");
        System.out.println("java.version: " + str);

        //java 虚拟机实现的版本
        str = System.getProperty("java.vm.version");
        System.out.println("java.vm.version: " + str);

        //java 类的路径
        str = System.getProperty("java.class.path");
        System.out.println("java.class.path: " + str);

        //操作系统的版本
        str = System.getProperty("os.version");
        System.out.println("os.version: " + str);

        //用户名
        str = System.getProperty("user.name");
        System.out.println("user.name: " + str);

        //用户路径
        str = System.getProperty("user.dir");
        System.out.println("user.dir: " + str);

        //用户 HOME 路径
        str = System.getProperty("user.home");
        System.out.println("user.home: " + str);
    }
  }
```

程序的运行结果为：

```
java.version: 1.4.1_02
java.vm.version: 1.4.1_02-b06
java.class.path: D:\JBuilder9\jdk1.4\lib;D:\;.;
os.version: 5.0
user.name: administrator
user.dir: E:\Java\chapt5
user.home: C:\Documents and Settings\Administrator.ZXB
```

另外，System 类中定义了三个和输入/输出流有关的静态成员变量 in, out 和 err。System 类中关于 in, out 和 err 的介绍见 9.1.3 节。

5.2.3 Class 类

Class 类的实例代表一个正在运行的 Java 应用程序的类或接口。Java 的基本数据类型（boolean，byte，char，shart，int，long，float，double）以及数组和关键字 void 都是由 Class 对象来表达。

Class 类没有公共的构造方法，Class 对象由 Java 虚拟机自动构造。

Class 类的常用方法有：

⌘ String static getName()
　返回对象的类名

⌘ class static forName(String ClassName)
　使用 ClassName 指定的、与类或接口相联系的 class 对象

⌘ class static forName(String name，boolean initialize，ClassLoader loader)
　使用 loader 指定的类装载器

Class 类的 forName()方法可用于安装驱动程序。例如，第 11 章讨论的安装 JDBC-ODBC 驱动程序，可以使用下面语句：

```
Class.forName("sun.jdbc.odbc.JdbcOdcDriver");
```

要安装 Oracle Jdbc 驱动程序，可以使用下面语句：

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

5.2.4 Runtime 类

每一个 Java 应用程序都有一个 Runtime 类的实例，从而允许应用程序与其运行的环境进行交互。可利用 Runtime 类直接访问运行时环境资源。

Runtime 类常用方法有：

⌘ static Runtime getRuntime()
　返回与当前应用程序相联系的运行时环境

⌘ void exit(int status)
　中断当前运行的 Java 虚拟机，status 为状态码，非 0 的状态码表示不正常中断

⌘ native void traceInstructions(Boolean on)
　设置对指令的追踪，如果参数 on 为 true，则 Java 虚拟机对于其上执行的每一条指令都发出调试信息

⌘ native void traceMethodCalls(Boolean on)
　设置对调用方法的追踪，如果参数 on 为 true，则支持指令的追踪

⌘ totalMemory()
　返回系统的内存总数

⌘ freeMemory()
　返回系统当前内存的剩余空间总数

⌘ gc ()

运行垃圾回收程序

【例 5.3】 用 Runtime 类获得当前系统运行状况示例。

```
public class RuntimeTest
{
    public static void main(String args[])
    {
        Runtime rtime = Runtime.getRuntime();
        long totalMemory = rtime.totalMemory()/1024;
        long freeMemory = rtime.freeMemory()/1024;
        System.out.println("totalMemory: " + totalMemory + "k");
        System.out.println("freeMemory: " + freeMemory+ "k");
    }
}
```

程序运行结果为：

```
totalMemory: 1984k
freeMemory: 1734k
```

5.2.5 Float 类

为了方便学习过 C 语言的人员学习 Java 语言，Java 语言也像 C 语言一样，有数据类型的概念。但数据类型和类不同。Java 是一种纯面向对象程序设计语言，为了达到这个目标，语言包中提供了 8 个称为数据类型包装类的类，专门完成把 Java 语言的 8 个基本数据类型包装为相应的类。

Java 语言中的 8 种基本数据类型是：byte, short, int, long, float, double, char, boolean，对应的 8 个数据类型包装类是：Byte, Short, Integer, Long, Float, Double, Character, Boolean。这里仅介绍 Float 类，其余的相似。

Float 类的几个构造方法如下：

- ⌘ Float (double value)　　以 double 类型的参数 value 创建一个对象
- ⌘ Float (float value)　　以 float 类型的参数 value 创建一个对象
- ⌘ Float (String s)　　以 String 类型的参数 s 创建一个对象

Float 类的几个方法如下：

- ⌘ compareTo(Float anotherF)
  比较两个对象的大小，大于返回 1，相等返回 0，小于返回–1
- ⌘ floatValue()　　返回对象的 float 数据类型的数值
- ⌘ doubleValue()　　返回对象的 double 数据类型的数值
- ⌘ intValue()　　返回对象的 int 数据类型的数值，小数部分丢掉

5.2.6 String 类

2.9 节讨论了字符串的概念。我们说过，定义字符串变量所使用的 String 不是一个基本数据类型，而是 Java API 中提供的类。

String 类的构造方法和常用方法有：

⌘ String()
构造方法，初始化创建一个字符为空的 String 对象

⌘ String(byte[] bytes)
构造方法，初始化创建一个 String 对象，其值由指定 bytes 转换而来

⌘ compareTo(String anotherString)
比较字符串，返回两者之间的差值

⌘ length()
获取字符串的长度

⌘ substring(int beginIndex)
返回当前字符串的由 beginIndex 开始到结尾的子串

⌘ substring(int beginIndex，int endIndex)
返回当前字符串的由 beginIndex 开始到 endIndex 结尾的子串

⌘ toLowerCase()
返回小写字符串，即返回的对象中所有的字符均为小写字符

⌘ toUpperCase()
返回大写字符串，即返回的对象中所有的字符均为大写字符

【例 5.4】 String 类应用举例。

要求：在程序中进行字符串的连接（+）、取子串等操作。

```
public class StringTest
{
    public static void main(String args[])
    {
        System.out.println("abc");
        String cde = "cde";
        System.out.println("abc" + cde);
        String c = "abc";
        c.substring(2,3);
        System.out.println("c = " + c);
        String d = cde.substring(1, 2);
        System.out.println("d = " + d);
    }
}
```

程序运行显示结果如下：

```
abc
abccde
c = c
d = d
```

5.2.7 Math 类

Math 类包含了一组基本的数学运算的方法和常数。如求绝对值的 abs()方法，计算三角函数的 sin()方法和 cos()方法，求两个数中的较小值和较大值的 min()方法和 max()方法，求随机数的 random()方法等。Math 类中的所有方法都定义为静态的。另外，Math 类还把 E 和 PI 定义为类的静态成员变量，其中，E 和 PI 代表数学上的相应常数。

Math 类是最终类（final），所以不能从 Math 类中派生其他的新类。

Math 类的几个常用方法如下：

⌘ double abs(double a) 求 a 的绝对值

⌘ float abs(float a) 求 a 的绝对值

⌘ int abs(int a) 求 a 的绝对值

⌘ long abs(long a) 求 a 的绝对值

⌘ double random() 返回一个(0.0,1.0)区间的随机数

abs()方法有 4 个，包括了 4 种基本的数值类型（double，float，int，long），其他方法（如 sin()方法 等）类同，为节省篇幅，这里不再列出。

random()方法是非常有用的方法。把 random()的返回值乘上一个整数，可以得到任意区间的、double 类型的随机数；把 random()的返回值乘上一个整数再转换成 int 类型，可以得到任意区间的、int 类型的随机数。

【例 5.5】 问题和例 2.13 相同，即求 10 个数中的最小数。

要求：用 Math 类的 random()方法产生随机数来给数组赋值。

程序设计如下：

```
public class Exam5_6
{
    public static void main(String args[])
    {
        final int SIZE = 10;                          //常量
        int i, min;
        int a[] = new int[SIZE];

        for(i = 0; i 〈a.length; i++)
            a[i] = (int)(Math.random() * 100);        //产生随机数并赋值

        System.out.print("数组元素为：");              //输出一维数组
        for(i = 0; i 〈 a.length; i++)
            System.out.print("    " + a[i]);

        min = a[0];
        for(i = 1; i 〈a.length; i++〉
            if(a[i] 〈min)    min = a[i];             //寻找最小数

        System.out.println("\n 最小数为：" + min);
```

```
        }
    }
```

程序运行结果如下：

数组元素为： 64 99 44 94 28 82 39 19 6 41
最小数为：6

5.3 实用包（java.util）简介

java.util 包主要包含集合框架、事件模型、日期和时间机制、国际化等的类和接口。本节介绍的 java.util 包的类和接口有 Arrays 类、Vector 类、Data 类和 Enumeration 接口。

5.3.1 Arrays 类

Arrays 类中包含有数组操作的一些常用方法（如排序和查找），这可以大大简化程序设计人员的设计过程。

Arrays 类的常用方法有：

⌘ static void sort(byte[] a)
把 byte 类型的数组 a 中的元素按升序排序

⌘ static void sort(byte[] a，int fromIndex，int toIndex)
把 byte 类型的数组 a 中的、从 fromIndex 到 toIndex 的元素按升序排序

⌘ static void sort(int[] a)
把 int 类型的数组 a 中的元素按升序排序

⌘ static void sort(int[] a，int fromIndex，int toIndex)
把 int 类型的数组 a 中的、从 fromIndex 到 toIndex 的元素按升序排序

⌘ static void sort(double[] a)
把 double 类型的数组 a 中的元素按升序排序

⌘ static void sort(double[] a，int fromIndex，int toIndex)
把 double 类型的数组 a 中的、从 fromIndex 到 toIndex 的元素按升序排序

⌘ static int binarySearch(byte[] a，byte key)
在 byte 类型的数组 a 中，使用折半查找算法查找指定元素 key 是否存在。若存在，返回该元素的数组下标；若不存在，返回–1

⌘ static int binarySearch(int[] a，int key)
在 int 类型的数组 a 中，使用折半查找算法查找指定元素 key 是否存在。若存在，返回该元素的数组下标；若不存在，返回–1

⌘ static int binarySearch(double[] a，double key)
在 double 类型的数组 a 中，使用折半查找算法查找指定元素 key 是否存在。若存在，返回该元素的数组下标；若不存在，返回–1

说明：

① 除了上面列出的 sort()方法和 binarySearch ()方法外，还有 char, float, long, Object 等类型的 sort()方法和 binarySearch ()方法，这里不再列出。

② 在使用 binarySearch()方法时，要求数组 a 中的元素已经有序排列，否则返回值未定义。

Arrays 类的应用例子见 5.4 节。

5.3.2 Vector 类

Vector 类称作向量类，它实现了动态的数组，用于元素数量变化的对象数组。像数组一样，Vector 类也用从 0 开始的下标表示元素的位置；但和数组不同的是，当 Vector 对象创建后，数组的元素个数会随着 Vector 对象元素个数的增大和缩小变化。

Vector 类的成员变量有：

⌘ elementData　　存放对象元素的数组

⌘ elementCount　　当前对象元素的个数

⌘ capacityIncreament　　元素个数变大时自动增大数组空间的个数

Vector 类的构造方法有：

⌘ Vector ()

创建元素为空的对象，且 elementCount=10，capacityIncreament=0

⌘ Vector (int initialC)

创建元素为空的对象，且 elementCount= initialC，capacityIncreament=0

⌘ Vector (int initialC，int capacityI)

创建元素为空的对象，且 elementCount= initialC，capacityIncreament= capacityI

Vector 类的常用方法有：

⌘ void add(int index，Object elem)

在 Vector 对象的 index 下标处插入元素 elem，index 下标以后的元素依次后移

⌘ boolean add(Object elem)

在 Vector 对象的尾部添加元素 elem。添加成功，返回 true；失败返回 false

⌘ boolean addAll(Collection c)

在 Vector 对象的尾部、依次添加集合对象 c 中的所有元素。添加成功，返回 true；失败返回 false

⌘ void addElement(Object obj)

在 Vector 对象的尾部增加对象 obj，且对象的元素个数加 1

⌘ boolean removeElement(Object obj)

如果对象非空，则删除 Vector 对象 obj 第一次出现的元素

⌘ copyInto(Object[] anArray)

把 Vector 对象指针所指位置的成分复制给数组 anArray，要求数组 anArray 要足够大，否则会抛出异常

⌘ Enumeration elements()　　返回对象的序列化元素

⌘ int size()　　返回对象的元素个数

和 Arrays 类相比，Vector 类最主要特点的是长度可随对象成分个数的增加或减少任意变化。

5.3.3 Data 类和 Calendar 类

Data 类提供了获取当前精确到毫秒时间的方法，并提供了许多方法截取当前时间的年、

月、日等数值。

Data 类的构造方法有：

⌘ Data ()
  创建一个可精确到毫秒的当前时间的对象

⌘ Data (long date)
  创建一个可精确到毫秒的参数 date 指定时间的对象，date 表示从 GMT（格林威治）时间 1970-1-1 00:00:00 开始至某时刻的毫秒数。

Calendar 类定义了许多如 YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, DAY_OF_WEEK 等成员变量，Calendar 类的 get()方法可以获取这些成员变量的数值。

【例 5.6】 设计程序获取本地计算机上的当前时间并显示。

程序设计如下：

```
import java.util.*;
public class MyDate
{
    public static void main(String args[])
    {
        Date date = new Date();                                //创建对象,其值为当前时间
        System.out.println(date);                              //输出显示当前时间

                                                               //分项输出显示当前时间
        Calendar now = Calendar.getInstance();                 //获得当前时间
        int year =now.get(Calendar.YEAR);                      //年份
        int month=now.get(Calendar.MONTH)+1;                   //月份
        int day = now.get(Calendar.DATE);                      //日期
        System.out.print(year+"年"+month+"月"+day+"日  ");

        int hour = now.get(Calendar.HOUR);                     //时
        int minute = now.get(Calendar.MINUTE);                 //分
        int second = now.get(Calendar.SECOND);                 //秒
        System.out.print(hour+"时"+minute+"分"+second+"秒  ");

        int week = now.get(Calendar.DAY_OF_WEEK);              //星期 1~7
        String str = "日一二三四五六";
        int i=2*(week-1);                                      //对应中文的下标
        System.out.println("星期"+str.substring(i,i+2));
    }
}
```

程序运行结果如下：

```
Tue Sep 28 10:32:29 CST 2004
2004 年 9 月 28 日  10 时 32 分 29 秒  星期二
```

5.3.4 Enumeration 接口

Enumeration 接口主要用于集合类对象的序列化。一个集合类对象中包含有许多元素，所谓对象是序列化的，就是说可以依次取出该对象中的元素。

Enumeration 接口定义如下：

```
public interface Enumeration
{
    Object nextElement();                          //返回后续元素
    boolean hasMoreElements();                     //是否还有后续元素
}
```

nextElement()方法返回后续元素；hasMoreElements()方法判断是否还有后续元素。

任何一个类，只要实现了 Enumeration 接口，其对象就是序列化的。所谓对象是序列化的，就是说若连续调用 nextElement()方法，每次将返回该集合对象当前元素的后续元素。

例如，Vector 类是一个集合类，Vector 类就实现了 Enumeration 接口。因此，若要输出一个 Vector 类的对象 v 中的所有元素，就可以编写如下代码：

```
Enumeration e = v.elements();
while (e.hasMoreElements())
{
    System.out.println(e.nextElement());
}
```

5.4 综合应用举例

为了让读者更好地理解 Java API 的使用方法,本节给出两个简单的 Java API 的应用例子。

【例 5.7】 问题和例 2.14 相同，即把 10 个数按从小到大的次序排序。

要求：用 Arrays 类的 sort()方法实现排序。

程序设计如下：

```
import java.util.*;
public class Exam5_7
{
    public static void main(String args[])
    {
        final int SIZE = 10;                       //常量
        int i, min;
        int a[] = new int[SIZE];

        for(i = 0; i <a.length; i++)
            a[i] = (int)(Math.random() * 100);     //产生随机数并赋值

        System.out.println("排序前数组元素为：");
```

```
        for(i = 0; i <a.length; i++)
            System.out.print(a[i] + "    ");

        Arrays.sort(a);                                        //排序

        System.out.println("\n 排序后数组元素为：");
        for(i = 0; i <a.length; i++)
            System.out.print(a[i] + "    ");
    }
}
```

程序运行结果如下：

```
排序前数组元素为：
40   96   31   20   85   58   40   89   18   18
排序后数组元素为：
18   18   20   31   40   40   58   85   89   96
```

【例 5.8】 使用 Vector 类和 Integer 类求解约瑟夫环问题。

约瑟夫环（Josephus）问题：古代某法官要判决 N 个犯人死刑，他有一条荒唐的法律，让犯人站成一个圆圈，从第 S 个人开始数起，每数到第 D 个犯人，就拉出来处决，然后再数 D 个，数到的人再处决，……，直到剩下的最后一个可赦免。

当 N=5，S=0，D=2 时，约瑟夫环问题执行过程示意图如图 5.1 所示。

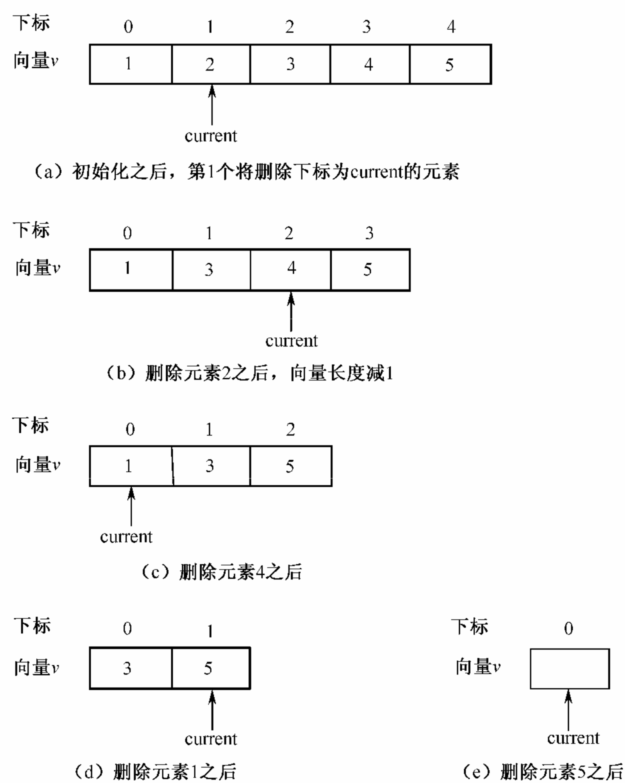


图 5.1 约瑟夫环问题执行过程示意图

设计：

① 初始化，创建一个 Vector 类的向量对象 *v*，调用 *v* 的 add()方法依次将 *N* 个人的序号添加到 *v* 中。由于 *v* 的元素类型为 Object，所以 add()的参数必须使用 new Integer(i+1)将 int 值转化为 Integer 对象。

② 设 current 为当前欲删除元素的下标，如果从第 1 个人开始数起，则 current 的初值应该设置为第 1 个人的前一个位置，即 current=*S*－1。

③ 循环，当 *v* 中元素多于一个时（v.size()>0），调用 v.remove()方法删除下标为 current 的元素。current 的值一直在增加，但既不能超出数组下标范围，又要实现环形结构，所以 current 的变化规律是 current = (current + 1) % v.size()，其中 v.size()是 *v* 的实际元素个数，每删除一个元素，v.size()的值少 1。

④ 为了更好地了解向量中的元素情况，每次循环都输出了向量中的全部元素值，调用 (Integer)v.get(i))方法将获得的下标为 *i* 的 Object 类元素转化为 Integer 类对象，再调用 Integer 类的 intValue()方法输出相应的 int 值。

⑤ 当循环停止时，向量 *v* 中只剩下一个元素，这就是我们要找的最后一个人。

程序如下：

```
import java.util.*;
public class Josephus                                   //使用 Vector 类求解约瑟夫环问题
{
    public static void main(String args[])
    {
        int n=5,s=0,d=2;
        Vector v = new Vector(n);                       //创建向量对象 v
        for(int i = 0; i <n; i++)
            v.add(new Integer(i+1));                    //向量对象 v 中添加 Integer 对象

        int current = s – 1;                            //设置当前元素下标 current 的初值
        while (v.size()>1)                              //向量对象 v 中多于一个元素时循环
        {
            System.out.print("Vector:    ");
            for(int i = 0; i <v.size(); i++)            //输出向量对象 v 中全部元素值
               System.out.print(((Integer)v.get(i)).intValue() + "    ");
            int j=0;
            while (j<d)                                 //计数
            {
                current = (current + 1) % v.size();
                j++;
            }
            System.out.println("\tcurrent = " + current + "        ");
            v.remove(current);                          //删除向量 v 中下标为 current 的元素
      }
      System.out.println("赦免者是" +
```

```
                        ((Integer)v.get(0)).intValue());
        }
    }
```

程序运行结果如下：

```
Vector:  1  2  3  4  5        current = 1
Vector:  1  3  4  5           current = 3
Vector:  1  3  4              current = 2
Vector:  1  3                 current = 0
```

赦免者是 3

习题 5

一、基本概念题

5.1　**什么叫** Java API**？**Java API **在** Java **编程中有什么作用？怎样导入** Java API **中的包？**

5.2　java.lang **包的主要用途是什么？**

5.3　java.util **包的主要用途是什么？**

5.4　Object **类的主要用途是什么？**

5.5　System **类的主要用途是什么？**

5.6　Class **类的主要用途是什么？**

5.7　Runtime **类的主要用途是什么？**

5.8　Integer **类的主要用途是什么？**

5.9　String **类的主要用途是什么？**

5.10　Math **类的主要用途是什么？**

5.11　Arrays **类的主要用途是什么？**

5.12　Vector **类的主要用途是什么？**

5.13　Data **类的主要用途是什么？**

5.14　Enumeration **接口的主要用途是什么？**

5.15　**什么叫集合类对象的序列化？**

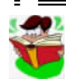
二、程序设计题

5.16　**用** System **类编程获得你的计算机的当前系统属性。**

5.17　**用** Runtime **类编程获得你的计算机的当前系统运行状况。**

5.18　**编程获得你的计算机的当前时间并显示。**

第 6 章　图形用户界面



教学要点

本章内容主要包括图形用户界面的基本概念，AWT 中几个基本类（包括组件类、容器类、菜单组件类）及其子类的继承关系，常用组件的使用方法，常用布局管理器的使用方法，绘图方法和常用类，Java 的事件处理（包括事件处理的基本过程和 Java 的事件处理设计方法）。

要求理解图形用户界面的基本概念，掌握组件类、容器类、菜单组件类及其子类的继承关系、绘图方法和常用类，熟练掌握常用组件（如框架、按钮、文本输入行等）的使用方法，熟练掌握 Java 事件处理的设计方法。

图形用户界面是程序和外界进行交流的图形接口。通过图形用户界面，可以为用户提供一个与程序交流的可视化平台，用户可以通过这个平台向程序发布指令，或者得到程序运行的结果。本章着重介绍 Java 常用的图形组件、布局管理器和事件处理方法。本章给出了较多的例子，演示了各种组件和事件处理的编程方法。灵活组合这些组件，读者可设计出各种形式的图形用户界面。

6.1　图形用户界面和 AWT

6.1.1　图形用户界面的基本概念

图形用户界面是计算机程序和用户交流的图形接口。通过图形用户界面，可以给用户提供一个和程序交流的可视化窗口。其实，经常操作计算机的人每天都在和图形用户界面打交道。在 Windows 窗口下管理和操作计算机，就是在通过 Windows 图形界面与计算机对话。常用的 IE 浏览器也为我们提供了一个图形化的用户界面，通过这个图形界面，可以浏览各种网页。例如，在地址栏中键入一个网址，就是告诉浏览器，我们想要浏览某个网站的一个页面，浏览器通过界面接收到这个命令后，运行相应的程序，就把我们需要的页面在浏览器中显示出来。

图形用户界面是由各种不同的组件按照一定的布局模式排列组成的。图形用户界面中的组件一般包括菜单、按钮、标签、文本编辑行等。同时，与这些组件配合工作的还有一些相关的事件。例如，当我们单击某一按钮时，就会触发相应的事件。系统就会执行用户为这些事件编写的事件处理程序。这样，通过前台图形界面和后台相关程序的配合，就可以方便地和计算机完成信息交互。

6.1.2　AWT 简介

Java 语言中，图形用户界面所用到的类和接口都是由 AWT（Abstract Window Toolkit）

提供的。AWT 是 Java API 的一部分，它为开发图形用户界面提供了实现各种组件、布局管理器和事件处理器的类和接口。用户在导入了 AWT 中的包或类后，通过创建组件、布局管理器和事件处理器的对象，就可以设计出所需要的各种形式的图形用户界面。

1．java.awt 包的子包

AWT 是由 Java API 中的 java.awt 包中的类和接口以及它的子包中的类和接口组成的。java.awt 包的子包主要有：

- ⌘ java.awt　　图形用户界面核心包，提供图形用户界面所需的类和接口
- ⌘ java.awt.color　　提供颜色和空间的类和接口
- ⌘ java.awt.dnd　　提供完成各种鼠标拖放操作的类和接口
- ⌘ java.awt.font　　定义与字体有关的类和接口
- ⌘ java.awt.event　　提供事件和监听器的类和接口
- ⌘ java.awt.geom.　　定义 2D 几何图形的类和接口
- ⌘ java.awt.im　　提供输入方法框架的类和接口
- ⌘ java.awt.image　　提供产生和修改图像的类和接口
- ⌘ java.awt.print　　提供打印机功能的类和接口
- ⌘ java.awt.swing　　新版本 JDK 提供的 AWT 的扩展包，提供功能更加强大的图形用户界面所需的类和接口

java.swing 包的主要功能是：可以根据环境来调整图形用户界面的外观和感觉，使得一个程序可以适应于各种不同的环境。另外，java.swing 包提供了一些比 AWT 中相应组件功能更强的组件。由于篇幅所限，本章不讨论 java.swing 包。java.swing 包中组件的使用方法和本章介绍的相应组件的使用方法基本类同。

2．java.awt 包中的类

java.awt 包中包含许多创建图形用户界面所需的类和接口，java.awt 包中的类以及它们的继承关系如图 6.1 所示。

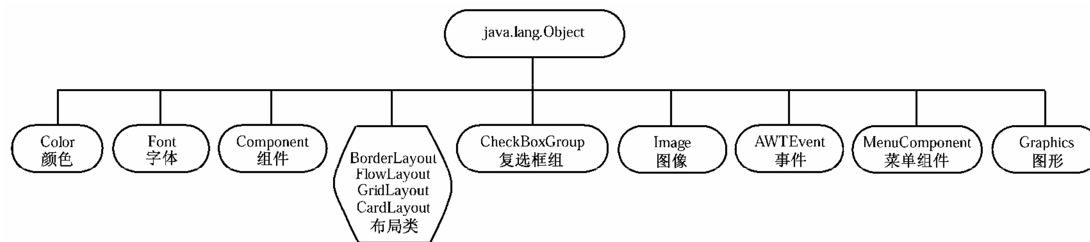


图 6.1　java.awt 中的类

6.2　Component（组件）类及其子类

Component 类是一个抽象类，通过继承它，可以得到各种功能的组件，AWT 中的图形组件大多都是 Component 类的子类，它们的继承关系如图 6.2 所示。

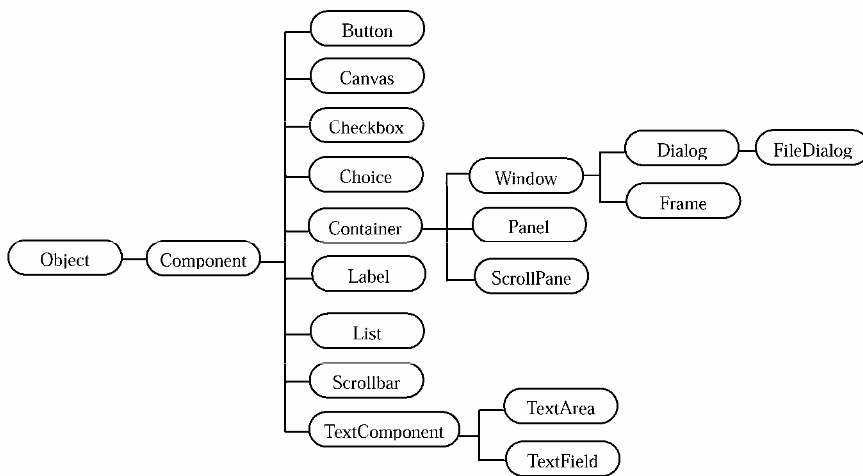


图 6.2　AWT 常用图形组件类继承关系

图形组件能够在屏幕上显示一定的图形。程序通过这些组件和用户进行信息交互。例如，按钮、标签和选择框都是典型的图形组件，用户可以通过创建这些组件的对象来方便、快速地设计出可视化的图形用户界面。

6.2.1　Component（组件）类

Component 类是 AWT 中所有图形组件类的父类，它包含许多图形组件类共有的成员变量和方法。

（1）常用的成员变量

⌘ LEFT_ALIGNMENT　　左边对齐方式
⌘ RIGHT_ALIGNMENT　　右边对齐方式

（2）构造方法

⌘ Component()　　构造方法

（3）常用方法

⌘ void add(component comp)　　在容器的末尾添加组件 comp

注意：该方法的参数 comp 的类为 component，而 component 类是所有组件类的父类，即该方法被所有 component 类的子类继承。因此，该方法可用于在任何组件类对象中添加任何组件类对象。

⌘ void setBackground(Color color)　　设置背景颜色
⌘ void setBounds(int x, int y, int w, int h)　　设置所占矩形的位置和尺寸
⌘ void setFont(Font)　　设置字体
⌘ void setForeground(Color c)　　设置前景颜色
⌘ void setVisible(boolean b)　　设置是否可见
⌘ void setLocation(int x,int y);　　移动组件到左上角位置为(x,y)
⌘ void setSize(int w,int h);　　重设组件的大小，其 w 为宽，h 为高
⌘ void setVisible(boolean b);　　设置组件是否可见。true 可见，false 不可见

注意：上述方法定义在 component 类中，而 component 类是所有组件类的父类，因此，

这些方法被所有 component 类的子类继承。

component 类还定义了许多绘图方法，component 类的绘图方法将在 6.4.1 节介绍。

6.2.2 Container（容器）类及其子类

1．Container（容器）类

Container类主要用来放置其他组件，所以称作"容器"。由于Container类继承自Component类，所以它既拥有 Component 类的所有功能，同时，它还具有"容器"的一些特征。

（1）构造方法

⌘ Container()　　构造方法

（2）常用方法

⌘ void remove (Component comp)　　删除容器的组件 comp

⌘ void setLayout(LayoutManager mgr)　　设置容器的布局管理器为 mgr

从图 6.2 可以看出，容器类的派生子类有：框架（Frame）、面板（Panel）、窗口（Window）、对话框（Dialog）等。

2．Panel（面板）类

Panel 类是 Container 的子类，它是一个放置其他图形组件（包括其他 Panel）的容器。

常用的构造方法：

⌘ Panel()　　缺省构造方法

⌘ Panel(LayoutManager layout)　　指定布局管理器的构造方法

Panel 类的常用方法都从 Component 类和 Container 类中继承而来。

3．Frame（框架）类

Frame 类是 Window 的子类（Window 是 Container 的子类），它的主要用途是用来放置其他组件。Frame 类提供了设置标题、改变窗口大小等方法。

Frame 和 Panel 都是容器类，不同的是，Frame 类对象是一个可在 Windows 中独立运行的窗口，而 Panel 类对象不能单独运行，只能放在容器中（可以放在 Frame 中，也可以嵌入另一个 Panel 中）。

（1）构造方法

⌘ Frame()　　缺省构造方法

⌘ Frame(String title)　　带标题的框架构造方法

（2）常用方法

⌘ void setMenuBar(MenuBar menubar)　　把菜单栏对象 menubar 加入框架

⌘ void setTitle(String title)　　设置框架标题

⌘ void setResizable(boolean b)　　设置框架大小是否可变。true 可变，false 不可变

6.2.3 Component 类的其他子类

Component 类的子类除 Container（容器）类外，还有其他许多子类，如 Label（标签）类、Button（按钮）类、TextField（文本输入行）类、TextArea（多行文本输入区）类、Checkbox

（复选框）类、List（列表）类、Choice（下拉列表）类等。用这些子类创建的对象都可以放置在容器类（以及容器类的子类）对象中。其实，任何一个容器类对象本身也可以放置在任何一个容器类对象中。

1．Label（标签）类

Label 类是一个用来显示文本（Text）的类。文本就是一个字符串。在程序中指定要显示的字符串，但该字符串在运行时不能被用户修改。

（1）常用成员变量

⌘ static int LEFT　　表示标签是左对齐方式
⌘ static int RIGHT　　表示标签是右对齐方式

（2）构造方法

⌘ Label()　　缺省构造方法
⌘ Label(String text)　　构造带字符串的标签
⌘ Label(String text, int alignment)　　构造指定对齐方式的字符串标签

其中，text 是字符串, alignment 是对齐方式，取值为 LEFT 或 RIGHT。

（3）常用方法

⌘ void setAlignment()　　设置对齐方式
⌘ String getText()　　返回字符串
⌘ void setText(String text)　　设置标签的显示文本为 text

2．Button（按钮）类

Button 类可以用来创建带标签的按钮。

（1）构造方法

⌘ Button()　　缺省构造方法
⌘ Button(String name)　　构造带标签 name 的按钮

（2）Button 类的常用方法

⌘ String getLabel()　　返回按钮的标签
⌘ void setLabel(String name)　　在按钮上设置标签为 name

3．TextField（文本输入行）类

TextField 类用来创建允许用户编辑的单行文本组件。用户可以通过这类组件输入和编辑字符串信息。TextField 与 Label 的本质差别是，程序运行时，TextField 可以获得焦点，而 Label 不能。所以，TextField 可编辑，而 Label 不可编辑。TextField 一般用作程序的输入。另外，TextArea、Checkbox、List、Choice 等也用作程序的输入。

（1）构造方法

⌘ TextField()　　缺省构造方法
⌘ TextField(String text)　　构造带字符串 text 的单行文本组件
⌘ TextField(int columns)　　构造列数为 columns 的单行文本组件

（2）常用的方法

⌘ void setText(String text)　　设置文本输入行的字符串

⌘ void setColumns(int length)　　设置文本输入行的列数

⌘ void addActionListener(ActionListener l)　　添加监听器

前面几章的程序举例中，数据基本都是初始化赋值给定的，基本没有用系统基本输入语句输入数据。这是因为 Java 语言的系统基本输入语句功能非常有限。Java 程序基本都是通过图形界面的文本输入行、多行文本输入区等来输入数据的。

4．TextArea（多行文本输入区）类

TextArea 类用来创建允许用户编辑的多行文本输入组件，组件中的字符串可以设定为可编辑的或只读的。另外，TextArea 类还提供水平或垂直的滚动条。

（1）常用成员变量

⌘ SCROLLBARS_HORIZONTAL_ONLY　　水平滚动条

⌘ SCROOLBARS_VERTICAL_ONLY　　垂直滚动条

⌘ SCROLLBARS_BOTH　　水平和垂直滚动条

⌘ SCROLLBARS_NONE　　无滚动条

（2）构造方法

⌘ TextArea ()　　缺省构造方法

⌘ TextArea (String text)　　带字符串 text 的构造方法

⌘ TextArea (int rows, int columns)　　设定行数 rows 和列数 columns 的构造方法

⌘ TextArea (String text, int rows, int columns)
带字符串 text 并设定行数 rows 和列数 columns 的构造方法

（3）常用方法

⌘ void append(String text)　　在多行文本输入区中加入文本 text

⌘ void setColumns(int columns)　　设置多行文本输入区的列数为 columns

⌘ void setRows(int rows)　　设置多行文本输入区的行数为 rows

【例 6.1】 AWT 组件在图形用户界面中的应用举例。

要求：通过继承 Frame 类定义一个新的框架类，然后用它创建一个框架对象，通过调用其方法设置框架对象的标题、大小、背景和位置，并且设置窗口的大小为可改变和可以被显示。采用同样的方法在程序中创建标签、按钮、文本输入行和多行文本输入区，使用各自的方法设置其属性。但不要求对用户的操作进行响应。

程序设计如下：

```
import java.awt.*;                          //导入 java.awt 包的所有类
public class  TestFrame extends Frame       //类 TestFrame 继承自 Frame
{
    public static void main(String args[])
    {
        TestFrame f = new TestFrame();          //创建框架对象
        f.setTitle("我的第一个框架标题");         //设置框架标题
        f.setSize(260,160);                     //设置框架大小
        f.setLayout(new FlowLayout());          //布局管理器设为 FlowLayout
        f.setLocation(0,0);                     //设置框架位置
```

```
        f.setResizable(false);                          //设置框架大小为不可变
        Label lb1=new Label();                          //创建文本标签对象
        lb1.setAlignment(Label.LEFT);                   //设定标签的对齐方式
        lb1.setText("第一个标签");                       //设定标签的文本
        f.add(lb1);                                     //在框架中加入标签
        Button b1 = new Button("第一个按钮");            //创建按钮对象
        f.add(b1);                                      //在框架中加入按钮
        TextField t = new TextField();                  //创建空的文本输入行对象
        t.setText("第一个字符串");                       //设置文本输入行中的字符串
        f.add(t);                                       //在框架中加入文本输入行
        TextArea t1 = new TextArea("第一个多行输入区文本", 3,20);
                                    //创建行数 3、列数 20 的带字符串的多行文本输入区
        t1.setEditable(false);                          //将多行输入区中文本设置为不可编辑
        f.add(t1);                                      //在框架中加入多行文本输入区
        f.setVisible(true);                             //设置框架为可见
    }
}
```

设计说明：

① 程序中创建了一个框架对象 f，然后调用它的方法，对框架对象的标题、大小、位置、窗口尺寸是否可变等属性进行了设置。

② f.setLayout(new FlowLayout())为框架设置 FlowLayout 布局管理器。FlowLayout 布局管理器的功能是：在此框架中加入的组件将自上而下、从左到右安排放置。布局管理器将在 6.5 节介绍。

③ 程序中还创建了标签、按钮、文本输入行和多行文本输入区四个组件，然后调用各自的方法设置属性，最后将它们放置在框架中。

④ 组件最重要的功能是引发事件，本例仅展示了组件的图形，而没有涉及组件的事件处理。所以，程序运行后框架上的任何组件都不响应用户的操作。关于组件的事件处理将在 6.6 节详细讨论。

程序运行的结果如图 6.3 所示。关闭图 6.3 所示的窗口要用 Windows 的任务管理器来结束 java.exe 进程。

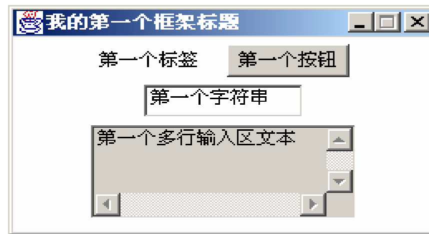


图 6.3　AWT 组件的创建和属性设置

5．Checkbox（复选框）类

Checkbox 类用来创建复选框组件。复选框组件的状态可以设置为开或关，用鼠标单击该

组件，组件状态将由开到关，或由关到开。

（1）构造方法

⌘ Checkbox()　缺省构造方法

⌘ Checkbox(String label)　带标签 label 的构造方法

⌘ Checkbox(String label, boolean state)　带标签 label 和状态 state 的构造方法

⌘ Checkbox(String label, boolean state, CheckboxGroup group)

带标签 label、状态 state 和复选框组 group 的构造方法

（2）常用方法

⌘ void setLabel(String label)　设置复选框的标签为 label

⌘ void setState(boolean state)　设置复选框的状态为 state

⌘ boolean getState()　返回复选框的状态

如果将几个复选框 Checkbox 对象放在逻辑上的一个组（group）中，则构成一组单选按钮。在这种情况下，每一时刻只允许其中一个处于选中状态。当单击其中某个时，它变成选中状态，其他则自动变成未选中状态。将几个复选框 Checkbox 对象组成一组需用到 CheckboxGroup 类。使用方法是：首先，创建一个 CheckboxGroup 对象，然后，创建需要组成一组的 Checkbox 对象。创建需要组成一组的 Checkbox 对象时，需要用到的构造方法和设置方法如下：

⌘ Checkbox(String label, CheckboxGroup group, boolean state)　构造方法

其中，label 是按钮标签名称，group 是 CheckboxGroup 对象，state 为复选框的开关状态。

⌘ void setCheckboxGroup(CheckboxGroup group)　设置复选框组为 group

【例 6.2】　复选框和单选按钮组应用举例。

要求：创建三个复选框和一个由四个复选框组成的单选按钮组，并选中该单选按钮组中的一个按钮。

程序设计如下：

```
import java.awt.*;
public class  TestCheckbox extends Frame
{
    public static void main(String args[])
    {
        TestCheckbox f = new TestCheckbox();
        f.setTitle("测试复选框组件");
        f.setSize(300,200);
        f.setLayout(null);                                          //不使用任何布局管理器
        Checkbox checkbox1 = new Checkbox("计算机",true);
                                                                    //创建选中的复选框
        Checkbox checkbox2 = new Checkbox("管理",true);
        Checkbox checkbox3 = new Checkbox("金融",false);
                                                                    //创建未选中的复选框
        checkbox1.setBounds(10,30,60,20);                           //设置位置和大小
        checkbox2.setBounds(10,50,60,20);
```

```
            checkbox3.setBounds(10,70,60,20);
            f.add(checkbox1);                                //添加复选框 checkbox1
            f.add(checkbox2);
            f.add(checkbox3);
            CheckboxGroup zct = new CheckboxGroup();         //创建单选按钮组 zct
            Checkbox c1 = new Checkbox("数学",zct, false);
                                                             //创建单选按钮组的按钮 c1
            Checkbox c2 = new Checkbox("英语",zct,true);
            Checkbox c3 = new Checkbox("美术",zct,false);
            Checkbox c4 = new Checkbox("体育",false);         //该按钮未加入组中
            c4.setCheckboxGroup(zct);                        //设置 c4 属于 zct 组
            c1.setBounds(150,30,60,20);                      //设置按钮 c1 的位置和大小
                c2.setBounds(150,50,60,20);
                c3.setBounds(150,70,60,20);
                c4.setBounds(150,90,80,20);
                f.add(c1);                                   //把按钮 c1 加入框架 f
                f.add(c2);
                f.add(c3);
                f.add(c4);
                f.setVisible(true);                          //设置框架 f 为可见
        }
    }
```

程序设计说明：程序中创建了三个复选框对象，并将第一个和第二个的状态设定为开。另外创建了一个单选按钮组，该单选按钮组中包含四个按钮。其中，“英语”按钮被选中。框架中组件的位置由 setBounds 方法来确定（不使用任何布局管理器）。

程序运行结果如图 6.4 所示。

图 6.4 复选框程序运行结果

6．List（列表）类

List 组件给用户提供了一个滚动的文本项清单，用户可以选中其中一项或多项文本。

（1）构造方法

⌘ List()　　缺省构造方法

⌘ List(int rows)　　设定行数为 rows 的构造方法

⌘ List(int rows, boolean multiple)

设定行数为 rows 且是否允许选择多项为 multiple 的构造方法

（2）常用方法

⌘ void add(String item)　　在列表中添加项目 item

⌘ void add(String item, int index)　　在列表的 index 位置处添加项目 item

7．Choice（下拉列表）类

Choice 是一个下拉式的数据选项，当前的选项会显示在选中列表框中。

（1）构造方法

⌘ Choice()　　构造方法

（2）常用方法

⌘ void add(String item)　　在下拉列表中添加项目 item

【例 6.3】 下拉列表和列表应用举例。

要求：在框架中创建下拉列表和列表。

程序设计如下：

```
import java.awt.*;
public class  TestChoiceList extends Frame
{
    public static void main(String args[])
    {
        TestChoiceList f = new TestChoiceList();
        f.setTitle("测试下拉列表和列表");
        f.setSize(300,200);
        f.setLayout(new FlowLayout());        //布局管理器设为 FlowLayout
        List list1 = new List();              //创建列表对象 List1
        list1.setMultipleMode(true);          //设定 list1  允许多选
        list1.add("语文");                    //在 list1 中添加项目"语文"
        list1.add("数学");
        list1.add("美术");
        list1.add("体育");
        list1.add("武术");
        list1.select(2);                      //预设 list1 的选择项目为 2（美术）。编号从 0 开始
        list1.select(4);                      //预设选择项目为 4
        f.add(list1);                         //把列表 List1 加入框架 f
        Choice choice1 = new Choice();        //创建下拉列表对象 choice1
        choice1.add("大学");                  //在 choice1 中添加项目"大学"
        choice1.add("中学");
        choice1.add("小学");
        choice1.add("幼儿园");
        choice1.select(2);                    //预设 choice1 的选择项目为 2。编号从 0 开始
        f.add(choice1);
        f.setVisible(true);                   //设框架 f 为可见
    }
}
```

程序设计说明：程序中创建了一个列表和一个下拉列表组件。列表和下拉列表的区别是：列表只能显示当前选择的项目，而下拉列表可以显示多个项目。

程序运行结果如图 6.5 所示。

图 6.5　下拉列表和列表程序运行结果

6.3　MenuComponent（菜单组件）类及其子类

在 AWT 中除了 Component 类可以作为图形组件的基本类以外，还有一个比较特殊的类，叫做 MenuComponent 类，它是直接继承自 Object 的菜单组件的基本类。MenuComponent 类及其子类是专门用来创建菜单类组件的，其继承关系如图 6.6 所示。

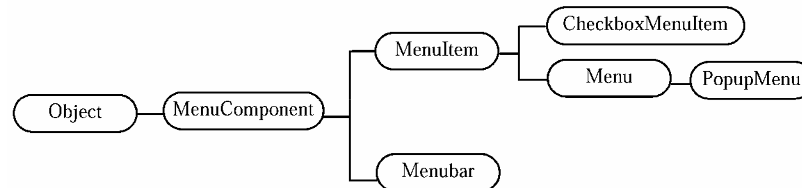


图 6.6　菜单组件类的继承关系

1．MenuComponent（菜单组件）类

MenuComponent 类常用的方法：

⌘　MenuContainer getParent()　　返回菜单组件所在的容器

⌘　String getName()　　返回菜单组件名称

⌘　Font getFont()　　返回菜单组件的字体

⌘　void setFont(Font f)　　设置菜单组件的字体为 f

一个用户程序菜单通常都是由 MenuComponent 类的子类对象构成的，如 MenuBar 类的对象、Menu 类的对象和 MenuItem 类的对象都是 MenuComponent 类的子类对象。

2．MenuBar（菜单栏）类

MenuBar 是放置菜单的容器。可以通过 Frame 类的 setMenuBar()方法把 MenuBar 对象加入一个框架中。

（1）构造方法

⌘　MenuBar()　　缺省构造方法

（2）常用方法

⌘　void add(Menu menu)　　在菜单栏中添加菜单 menu

⌘　void SetHelpMenu(Menu m)　　设置菜单 m 为帮助菜单

3．Menu（菜单）类

Menu 是菜单栏上放置的菜单。每一个菜单由一些菜单项组成。可以通过 MenuBar 类的

add()方法，把 Menu 对象放置在 MenuBar 对象上（即设置菜单栏上的主菜单）。

（1）构造方法

⌘ Menu()　　缺省构造方法

⌘ Menu(String title)　　带标题 title 的构造方法

（2）常用方法

⌘ MenuItem add(MenuItem menuitem)　　在菜单中添加菜单项 menuitem

⌘ void addSeparator()　　在菜单中添加分隔符

⌘ void insertSeparator(int index)　　在菜单的位置 index 处添加分隔符

4．MenuItem（菜单项）类

所有菜单中的菜单项都是 MenuItem 类或者它的子类的对象。可以通过 Menu 类的 add()方法，把 MenuItem 对象添加到 Menu 对象上（即设置菜单栏上某个主菜单的子菜单）。

（1）构造方法

⌘ MenuItem()　　缺省构造方法

⌘ MenuItem(String title)　　带标题 title 的构造方法

⌘ MenuItem(String title, MenuShortcut s)　　带标题 title 和快捷键 s 的构造方法

（2）常用方法

⌘ void setShortcut(MenuShortcut s)　　设置当前菜单项的快捷键为 s

⌘ void setLabel(String label)　　设置当前菜单项的标签为 label

⌘ String getLabel()　　返回当前菜单项的标签

【例 6.4】　菜单设计举例。

要求：利用菜单栏、菜单和菜单项等类创建一个窗口中的菜单系统。

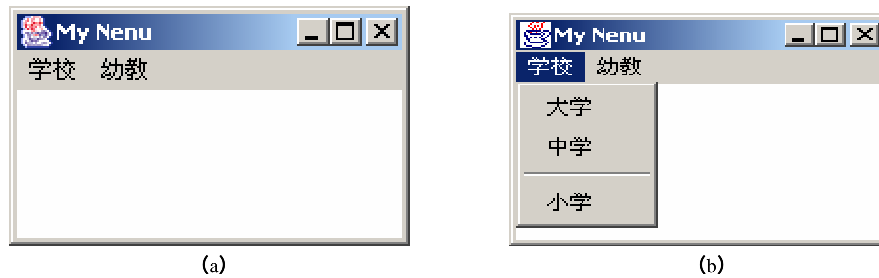
程序设计如下：

```
import java.awt.*;
public class TestMenu extends Frame
{
    public TestMenu(){}
    public static void main(String args[])
    {
        TestMenu frame1 = new TestMenu();
        frame1.setTitle("My Nenu");
        frame1.setSize(200,120);
        MenuBar menubar1 = new MenuBar();                //创建菜单栏对象 menubar1
        Menu menu1 = new Menu("学校");                   //创建菜单对象 menu1
        Menu menu2 = new Menu("幼教");
        MenuItem menuitem1 = new MenuItem("大学");       //创建菜单项对象
        MenuItem menuitem2 = new MenuItem("中学");
        MenuItem menuitem3 = new MenuItem("小学");
        MenuItem menuitem4 = new MenuItem("幼儿园");
        MenuItem menuitem5 = new MenuItem("托儿所");
```

```
            menubar1.add(menu1);                    //将菜单 menu1 加入菜单栏 menubar1
            menubar1.add(menu2);
            menu1.add(menuitem1);                   //将菜单项 menuitem1 加入菜单 menu1
            menu1.add(menuitem2);
            menu1.addSeparator();                   //在 menu1 中的当前位置加分隔符
            menu1.add(menuitem3);
            menu2.add(menuitem4);                   //将菜单项 menuitem4 加入菜单 menu2
            menu2.add(menuitem5);
            frame1.setMenuBar(menubar1);            //把菜单栏 menubar1 加入框架 frame1
            frame1.setVisible(true);
        }
    }
```

程序设计说明：程序中首先创建了框架、菜单栏、菜单和菜单项对象，然后将菜单对象加入菜单栏对象，将菜单项对象加入菜单对象，最后将菜单栏对象加入框架对象。这样就组成了一个图形用户界面的菜单系统。

程序运行后的界面如图 6.7（a）所示，如果用户选择了“学校”菜单，则界面如图 6.7（b）所示。



(a)　　　　(b)

图 6.7　菜单界面

6.4　AWT 中的绘图方法和常用类

我们在设计图形用户界面时，除了用到 AWT 中的一些图形组件外，还经常需要在组件上绘制各种图形和字符串，有时还需要设置组件的颜色和组件上字符串的字体，对于这些需求，AWT 中提供了专门的类。

本节首先介绍 Component（组件）类中的几个重要的绘图方法，然后介绍 Graphics（绘制图形工具）类、Font（字体）类和 Color（颜色）类。

6.4.1　Component（组件）中的绘图方法

6.2 节介绍了 Comonent 的许多常用方法，这里，再介绍几个 Comonent 的图形绘制方法：

⌘　void paint(Graphics g)　　　　绘制图形

说明：当需要绘制图形时，系统会自动执行该方法。该方法运行时，会在显示器上绘制指定的图形。一般在下面几种情况下系统会调用这个方法：(a) 当一个组件第一次显示时；(b) 当容器重新调整大小时；(c) 当容器从非作用窗口变为作用窗口时。

⌘　void repaint()　　　　　　　　　　　重新绘制图形

该方法有以下几种变形：

（a）void repaint(long time)　　　　　　等待时间 time 后重新绘制图形

（b）void repaint(int x, int y, int w, int h)

在区域（x, y）处重新绘制宽为 w 高为 h 的图形

（c）void repaint(long time, int x, int y, int width, int height)

等待时间 time 后在区域（x,y）处重新绘制宽为 w 高为 h 的图形

说明：上面三种形式的重新绘制图形方法都可以用来重新绘制图形。repaint()方法一般不是马上执行，当程序运行在较慢的平台上或计算机较忙时，应该考虑使用第 1 种或第 3 种 repaint 方法，即指定在多长时间内必须执行 repaint 方法，否则就放弃。

⌘　void update()　　　　　　　　　　　更新图形

说明：AWT 调用该方法来响应对 repaint()方法的调用，在调用 update()或 paint()之前，图形的外观将不会发生改变。update()方法更新图形的步骤为：首先，通过填充组件的背景色来清除该图形；然后，设置图形的颜色为该图形的前景色；最后，调用 paint 方法完整地重新绘制该图形。

AWT 的图形绘制过程如图 6.8 所示。

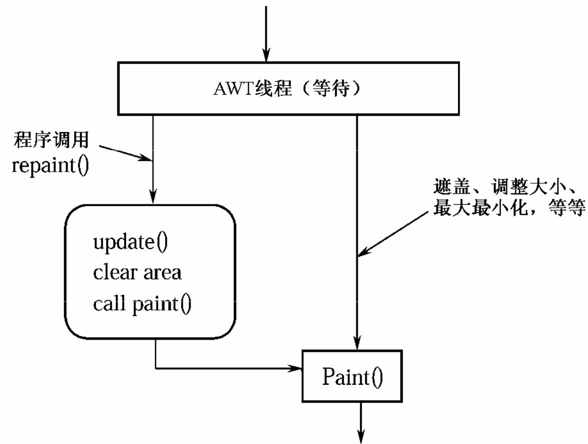


图 6.8　AWT 的图形绘制过程

6.4.2　Color（颜色）类

Color 类用来设置文本或图形的颜色。创建了 Color 对象后，就可以将其用于后面的文本或图形中。

在 Java 中，颜色有两种定义方法：

（1）Java 中的标准色

Java 中的 Color 类中有很多用于表示颜色的常量，例如：

⌘　static final Color black　　　　　　黑色

⌘　static final Color blue　　　　　　 蓝色

⌘　static final Color red　　　　　　　红色

这样，红色可以用 Color.red 表示，蓝色可以用 Color.blue 表示。其他常用的颜色都可以

用这种方式表示，例如，青色为 Color.cyan、灰色为 Color.gray。

（2）用 RGB 值表示

RGB 值表示法通过设置颜色中的红、绿、蓝三种颜色的色值来定义一种颜色，每种颜色的色值范围在 0～255 之间。

Color 类中设计有构造方法，最常用的是 Color (int red, int green, int blue)。例如，

```
Color myColor = new Color (255,0,0);
```

就定义了大红色。

这种方法的好处在于它可以灵活地定义出许多非标准色。定义好颜色对象后，就可以使用这些颜色对象来设置文本或图形的颜色。

6.4.3 Font（字体）类

Font 类用来定义文字的字体、风格和大小等特性。我们使用的文字都可以用这三个属性来描述：

① 字体：字的字体名称，如 serif, monopaced, Courier, Dialog 或 Times。

② 风格：字的风格，如粗体、斜体。

③ 大小：字的大小，以像素为单位。

（1）构造方法

⌘ Font(String name, int style, int size)

其中，name 指字体名称，style 是字体的风格，size 指字体的大小。

（2）常用方法

⌘ String getName()　　　　获取字体名称

⌘ int getStyle()　　　　获取字体风格

例如，语句

```
Font myFont = new Font("Times", Font.PLAIN,20);
```

就定义了一种 Times 字体、Font.PLAIN 风格和 20 像素单位大小的字体对象 myFont。

6.4.4 Graphics（图形工具）类

Graphics 类用来绘制各种图形和字符串。

常用的绘图方法有：

⌘ abstract void drawLine(int x1, int y1, int x2, int y2)
绘制直线，起点坐标(x1, y1)，终点坐标（x2, y2）

⌘ abstract void drawRect(int x, int y, int w, int h)
绘制矩形，矩形左上角坐标(x, y)，宽度 w，高度 h

⌘ abstract void drawOval(int x, int y, int w, int h)
绘制椭圆，包含椭圆的矩形左上角坐标(x, y)，宽度 w，高度 h
说明：当宽度 w 等于高度 h 时即为绘制圆。

⌘ abstract void fillOval(int x, int y, int w, int h)
填充椭圆，包含椭圆的矩形左上角坐标(x, y)，宽度 w，高度 h
说明：填充椭圆方法 fillOval( )可以绘制实心椭圆。

⌘ abstract void drawString(String str, int x, int y)

绘制字符串，字符起点坐标为(x, y)，str 为要绘制字符串

⌘ abstract boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)

绘制图像，起点坐标为(x, y)，图像宽为 width，高为 height，图像浏览的对象为 observer。图像绘制成功，返回 true；否则，返回 false。

虽然 Graphics 类提供的方法是抽象（abstract）方法，但用户编程时可直接使用，这是因为系统内部已经实现了这些抽象方法。

关于 drawImage ()方法的使用方法见第 7 章的例 7.4。

Graphics 类中还有很多绘制和填充其他各种图形的方法，读者可参阅 Java API 文档或相关参考文献。

【例 6.5】 用 Graphics 对象绘制各种图形

```
import java.awt.*;                              //导入 java.awt 包中的所有类
public class    TestGraphics extends Frame
{
    public void paint(Graphics g)
    {
        int x,y,i=0;
        Font font = new Font("Serif",Font.ITALIC|Font.BOLD,40);
        //创建字体对象并初始化赋值
        g.setFont(font);                        //设置字体
        g.drawOval(60,50,80,80);
                        //在包含圆的正方形左上角坐标(60,50)处绘制直径为 80 像素单位的圆
        g.setColor(Color.red);                  //设置颜色为红色
        g.fillOval(150,50,80,80);               //在(150,50)处绘制直径为 80 的实心圆
        g.setColor(Color.pink);                 //设置颜色为粉红色
        g.drawOval(240,50,80,80);               //在(240,50)处绘制直径为 80 的圆
        g.setColor(Color.green);                //设置颜色为绿色
        g.drawString("I love Java",80,260);
                                                //在坐标(80,260)处绘制字符串
        g.setColor(Color.blue);                 //设置颜色为蓝色
        for(i=0;i〈20;i++)                      //绘制 20 个实心圆
        {
            x = (int)(Math.random() * 300)+30;  //随机生成坐标 x
            y = (int)(Math.random() * 200)+130; //随机生成坐标 y
            g.fillOval(x,y,10,10);              //在(x,y)处绘制直径为 10 的实心圆
        }
        g.setColor(Color.orange);               //设置颜色为橘色
        y = 100;
        for(i=0;i <40;i++)                      //绘制 40 个矩形
        {
            y+= 5;
```

```
            g.drawRect(30,30,320,y);                //依次绘制 40 个间隔为 5 的矩形
        }
    }
    public static void main(String args[])
    {
        TestGraphics f = new TestGraphics();        //创建框架
        f.setTitle("我的第一个 Graphics");           //设置框架标题
        f.setSize(400,345);                         //设置框架大小
        f.setLocation(0,0);                         //设置框架位置
        f.setVisible(true);                         //设置框架可见
    }
}
```

程序设计说明：

（1）程序中运用 Graphic 类的对象 g 绘制了直线、字符串、矩形和圆等。当没有设置颜色时，组件会使用黑色（默认色），当设置一种新的颜色后，这种颜色将会用到以后所有绘制的图形中去，直到重新设置一种新的颜色为止。

（2）main()方法中不需要调用 paint()方法，但程序设计者需要通过重新定义 paint()方法来告诉系统需要绘制的图形。

程序运行结果如图 6.9 所示。

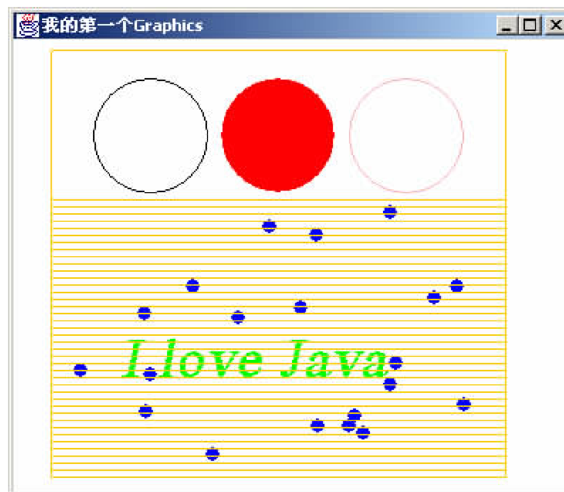


图 6.5　用 Graphics 对象绘制各种图形

6.5　布局管理器

容器中可以放置许多不同的组件。这些组件在容器中的摆放方式称作布局。Java 中不使用坐标这种绝对定位的方法，而使用布局管理器进行相对定位。这种方式的优点是显示界面能够自动适应不同分辨率的屏幕。布局管理器是用 AWT 中的布局管理器类创建的对象。常用的布局管理器类有：FowLayout 类、BorderLayout 类和 GridLayout 类。

6.5.1 FlowLayout 类

FlowLayout 布局管理器从左到右排列组件，一行放满后，再从第二行开始。FlowLayout 布局管理器是容器类组件 Pane 的默认布局管理器。

（1）静态成员变量

⌘ CENTER　　各行组件居中排放

⌘ LEFT　　各行组件从最左边开始排放

⌘ RIGHT　　各行组件从最右边开始排放

说明：上述三个成员变量称为对齐方式的成员变量。

（2）构造方法

⌘ FlowLayout()　　缺省构造方法

⌘ FlowLayout(int align)　　指定对齐方式的构造方法

⌘ FlowLayout(int align, int hgap, int vgap)

其中，align 是指定的对齐方式，hgap 是水平间距，vgap 是垂直间距。

（3）常用方法

⌘ int getAlignment()　　获取对齐方式

⌘ int setAlignment(int align)　　设置对齐方式

⌘ int setHgap()　　设置水平间距

⌘ int setVgap()　　设置垂直间距

构造方法举例：

FlowLayout(FlowLayout.RIGHT,10,20);

第一个参数表示组件的对齐方式，即组件在一行中是右对齐；第二个参数是组件之间的横向间隔；第三个参数是组件之间的纵向间隔。后两个参数的单位都是像素。

【例 6.6】　FlowLayout()布局管理器应用示例。

要求：设计若干个组件放置在 Frame 中，Frame 采用 FlowLayout()布局管理器管理其中组件的摆放。

程序设计如下：

```
import java.awt.*;
public class TestFlowLayout extends Frame
{
    public static void main(String args[])
    {
        TestFlowLayout f = new TestFlowLayout();            //创建框架对象 f
        f.setTitle("My FlowLayout Manager");
        f.setSize(200,120);
        FlowLayout fl= new FlowLayout();                    //创建布局管理器对象 fl
        fl.setHgap(10);                                     //设置水平间距
        f.setLayout(fl) ;                                   //设置框架 f 使用布局管理器 fl
        f.add(new Button("OK"));                            //在框架 f 中加入"OK"按钮
        f.add(new Button("Cancel"));                        //在框架 f 中加入"Cancel"按钮
```

```
            f.add(new Button("Password"));          //在框架 f 中加入"Password"按钮
            f.add(new Button("Reset"));             //在框架 f 中加入"Reset"按钮
            f.setVisible(true);
        }
    }
```

程序运行的结果如图 6.10 所示。

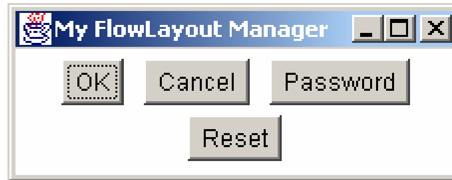


图 6.10　FlowLayout 管理器程序运行结果

程序运行结果说明：若使用 FlowLayout 布局管理器，当容器大小发生变化时，容器上组件的排放位置会发生变化，其变化规律是：组件的大小不变，但是相对位置会发生变化。例如，图 6.10 中三个按钮都处于同一行，但是如果把该窗口变窄，窄到刚好能够放下一个按钮，则第二个按钮会被放置在第二行，第三个按钮会被放置在第三行，第四个也同样放置。

6.5.2　BorderLayout 类

BorderLayout 布局管理器按照东、西、南、北、中五个区域放置容器中的组件。BorderLayout 布局管理器是容器类组件 Window, Frame 和 Dialog 等的默认布局管理器。

（1）静态成员变量

- ⌘ CENTER　　将组件放置在容器中央
- ⌘ EAST　　将组件放置在容器右边
- ⌘ NORTH　　将组件放置在容器上边
- ⌘ SOUTH　　将组件放置在容器下边
- ⌘ WEST　　将组件放置在容器左边

（2）构造方法

- ⌘ BorderLayout()　　缺省构造方法
- ⌘ BorderLayout(int hgap, int vgap)　　水平间距 hgap 和垂直间距 vgap 的构造方法

（3）常用方法

- ⌘ void setHgap(int hgap)　　设置水平间距
- ⌘ void setVgap(int vgap)　　设置垂直间距

【例 6.7】　BorderLayout()布局管理器应用示例。

要求：设计若干个组件放置在 Frame 中，Frame 采用 BorderLayout()布局管理器。

```
import java.awt.*;
public class    TestBorderLayout extends Frame
{
```

```
    public static void main(String args[])
    {
        TestBorderLayout    frame1= new TestBorderLayout ();
        frame1.setTitle("My BorderLayout");
        frame1.setSize(200,200);
        BorderLayout border = new BorderLayout(5,10);
                            //创建水平间距为 5，垂直间距为 10 的 BorderLayout 对象 border
        frame1.setLayout(border);                          //框架将使用 border 布局管理器
        frame1.add(new Button("South"),border.SOUTH);      //放置在南区
        frame1.add(new Button("West"),border.WEST);        //放置在西区
        frame1.add(new Button("North"),border.NORTH);      //放置在北区
        frame1.add(new Button("East"),border.EAST);        //放置在东区
        frame1.add(new Button("Center"),border.CENTER);    //放置在中区
        frame1.setVisible(true);
    }
}
```

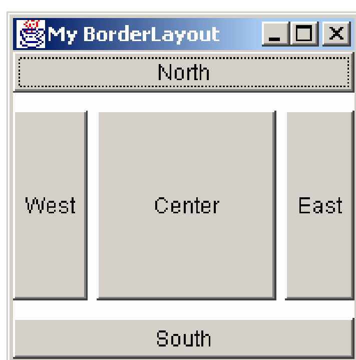


图 6.11　BorderLayout 管理器程序运行结果

程序运行的结果如图 6.11 所示。

程序运行结果说明：若使用 BorderLayout 布局管理器，当容器大小发生变化时，容器上组件的变化规律是：组件的相对位置不变，大小发生变化。如果容器变高了，则 North, South 区域不变，West, Center, East 区域变高；如果容器变宽了，West, East 区域不变，North, Center, South 区域变宽。不一定所有的区域都有组件，如果四周的区域（West, East, North, South 区域）没有组件，则由 Center 区域去补充，但是如果 Center 区域没有组件，则保持空白。

6.5.3　GridLayout 类

GridLayout 布局管理器可以将容器分为若干个大小相等的矩形。

（1）构造方法

⌘　GrideLayout()　　缺省的构造方法

⌘　GrideLayout(int rows, int cols)　　指定行数、列数的构造方法

⌘　GrideLayout(int rows, int cols, int vgap)　　指定行数、列数和垂直间距的构造方法

其中，row 是行数，cols 是列数，vgap 是垂直间距。

（2）常用方法

⌘　int setColumns(int cols)　　设置列数

⌘　int setHgap(int hgap)　　设置水平间隔

⌘　int setVgap(int vgap)　　设置垂直间隔

GrideLayout 布局管理器设计的图形界面风格如图 6.12 所示。

以上分别介绍了三种风格的布局管理器，在实际设计图形用户界面时，对于比较复杂的布局，有时还需要使用容器 Panel 组件。例如，一个 Frame 中可以添加若干个 Panel，这些 Panel 将 Frame 分隔成几部分。每个 Panel 上又可以添加若干个组件。Frame 的布局指的是若干个 Panel 的相对位置方式。每个 Panel 的布局指的是其上若干个组件的相对位置方式。每个 Panel 都可以设置不同的布局方式。

图 6.12　GridLayout 管理器运行结果

6.6　Java 的事件处理

在图形用户界面设计方法的学习中，我们已经认识了各种图形组件，也了解了如何将这些组件组合在一起，组成用户和计算机交流的图形界面。但是，我们设计图形用户界面的目的是为了方便程序和用户进行交流，也就是通过这个界面来控制程序的执行。例如，打开一个记事本，这就是一个文本编辑器软件的图形用户界面，我们单击记事本窗口右上角“×”按钮，窗口就会关闭；单击菜单中的“存盘”菜单项，计算机就会将当前文件存入磁盘。那么，这些图形用户界面是怎样响应用户的操作，并完成这些功能的呢？

其实，这些功能都是由后台的事件处理程序完成的。本节将详细介绍 Java 语言的事件处理程序设计方法。

6.6.1　事件处理的基本过程

下面首先介绍在计算机处理事件的过程中，事件处理机制的一些概念，然后介绍事件处理机制的工作原理。

1．事件源和事件

事件源：事件源是产生事件的图形组件。例如，按钮、窗口等，当对它们进行某种操作时，它们会产生各种不同的事件。如单击按钮时，系统会产生一个 Action 事件。

事件：用户对事件源进行的操作叫作事件。一旦一个事件发生，系统就会自动产生描述该事件的一些信息。描述事件的信息称作事件的属性信息。一般来说，事件的属性信息包含有事件的名称、来源、产生时间等。

2．事件处理的基本过程

事件处理过程包括事件的接收和事件的处理两部分。事件的接收由系统负责。当用户对某一事件源进行操作时，就会产生相应事件。系统一旦监听到事件发生，就把该事件交给相应的事件处理程序进行处理，从而完成用户希望的操作。例如，若用户用鼠标单击了“确定”按钮，该按钮就是事件源，该事件源上就会产生一个鼠标单击事件。系统一旦监听到事件发生，就把该事件交给“确定”按钮上的鼠标单击事件处理程序进行处理，从而完成用户希望的操作。事件处理的基本过程如图 6.13 所示。

图 6.13　事件处理的基本过程

事件处理过程的一个重要问题是：系统如何把事件交给事件处理程序？事件发生后，系统把事件传递给事件处理程序的方式称为**事件模型**。有两种基本的事件模型：一种称为层次事件模型，另一种称为委托事件模型。

层次事件模型的基本传递方法是：当系统监听到一个事件后，首先传递给直接相关的组件，该组件可以对事件进行处理，也可以不处理；如果组件没有对事件进行处理，则向上传递给组件所在的容器。同样，该容器可以对事件进行处理，也可以不处理；如果容器没有对事件进行处理，则继续向上传递给容器所在的容器。依此类推，直至顶层容器。层次事件模型虽然原理简单，容易理解，但程序设计却比较麻烦。Java 语言早期的 JDK1.0 版本采用的就是层次事件模型。

委托事件模型的基本传递方法是：事件的传递由事件监听器进行管理。任何事件处理程序需要首先向事件监听器注册。这样，当系统监听到一个事件后，就把事件的传递工作委托给事件监听器来完成。事件监听器通过分析事件的属性信息，把事件交给已注册的相应事件处理器来处理。Java 语言 JDK1.1 以后的版本采用的都是委托事件模型。

6.6.2　Java 的事件处理

1．Java 的事件处理程序设计

下面首先给出一个包含事件处理的简单例子，然后讨论 Java 语言的事件处理程序设计方法。

【例 6.8】　设计一个简单的聊天窗口。

要求：聊天窗口中能响应用户输入的字符串，并能响应关闭窗口操作。

程序设计如下：

```
import java.awt.*;
import java.awt.event.*;
public class ChatFrame extends WindowAdapter implements ActionListener
                            //定义事件处理类，由它对按钮和框架上的窗口产生的事件进行处理
{
    Frame    f;                                     //框架对象
    Button b1;
    TextArea ta;
    TextField tf;
    public ChatFrame()                              //构造窗口界面
    {
        f = new Frame("聊天程序界面");              //创建带标题的框架
        f.setSize(400,300);                         //设置框架大小

        ta = new TextArea();                        //多行文本框
```

```
        f.add(ta);

        Panel p = new Panel();
        f.add(p,"South");
        tf = new TextField(20);                     //创建文本输入行对象
        b1 = new Button("Send");                    //创建按钮对象
        p.add(tf);
        p.add(b1);                                  //在框架中加入按钮
        b1.addActionListener(this);                 //注册，按钮的单击事件由对象自己处理

        f.setVisible(true);                         //设置框架为可见
        f.addWindowListener(this);                  //注册监听框架上的窗口事件
    }
    public void actionPerformed(ActionEvent e)      //处理按钮单击事件
    {
        ta.append(tf.getText()+"\n");
                          //把文本输入行上用户当前输入的字符串添加到多行文本框对象 ta 上
    }

    public void windowClosing(WindowEvent e)        // 处理窗口关闭事件
    {
        System.exit(0);                             //程序停止运行,关闭框架窗口
    }

    public static void main(String args[])
    {
        new ChatFrame();
    }
}
```

程序运行后的界面如图 6.14 所示。

图 6.14　通过按钮改变框架颜色程序的运行结果

例 6.8 的程序中黑体字部分是处理事件需要的部分。分析例 6.8 的程序可以总结出，用户程序的事件处理主要要做三件事：

① 实现一个事件监听器接口，或继承一个事件适配器类。例 6.8 中在定义类 ChatFrame 时既继承了 WindowAdapter 类（窗口事件适配器类），也实现了接口 ActionListener（Action 事件接口）。程序中的语句为：

```
public class ChatFrame extends WindowAdapter implements ActionListener
```

② 注册。例 6.8 中按钮 b1 向 Action 事件监听器（ActionListener）注册，注册语句为：

```
b1.addActionListener(this);     //注册，按钮的单击事件由对象自己处理
```

该语句表示一旦按钮 b1 上发生了 Action 事件，则由本类上（this 表示本类）实现的 Action 事件处理方法处理，即由本类上实现的 actionPerformed()方法处理。

例 6.8 中框架上的事件向 Window 事件监听器（WindowListener）注册，注册语句为：

```
f.addWindowListener(this);                    //注册监听框架上的窗口事件
```

该语句表示一旦框架 f 上发生了窗口操作事件，则由本类上实现的 Window 事件处理方法处理，本例是关闭窗口事件由本类上实现的 windowClosing()方法处理。

③ 实现事件接口的方法，或定义事件适配器类中要重新处理的相应方法。

本例要求响应窗口关闭事件，因此要重新定义窗口事件适配器类的 windowClosing()方法。程序设计如下：

```
public void windowClosing(WindowEvent e)      // 处理窗口关闭事件
{
    System.exit(0);                           //程序停止运行,关闭框架窗口
}
```

ActionListener 接口中只定义了一个处理按钮单击事件的方法 actionPerformed()。实现方法设计为：

```
public void actionPerformed(ActionEvent e)      //处理按钮单击事件
{
    ta.append(tf.getText()+"\n");
                          //把文本输入行上用户当前输入的字符串添加到多行文本框对象 ta 上
}
```

2．事件类和事件

Java 语言中的组件就是事件源，它们可以产生一个或多个事件。 Java 语言定义了所有事件的事件类。java.awt.AWTEvent 类是所有事件类的父类，所有事件类都是由它派生出来的。事件类之间的继承关系如图 6.15 所示。

每个事件类对象可以产生若干事件，各事件类中常见的事件有：

⌘ ComponentEvent（组件事件）　例如，组件尺寸的变化、移动。
⌘ ContainerEvent（容器事件）　例如，容器内组件的增加、删除。
⌘ WindowEvent（窗口事件）　例如，关闭和打开窗口。
⌘ FocusEvent（焦点事件）　例如，焦点的获得和释放

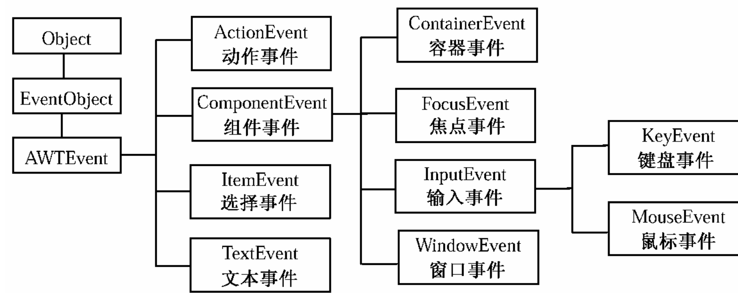


图 6.15 事件类的继承关系

⌘ KeyEvent（键盘事件） 例如，按键的按下、松开。
⌘ MouseEvent（鼠标事件） 例如，单击鼠标、拖动鼠标。
⌘ ActionEvent（动作事件） 例如，按钮按下，TextField 中按 Enter 键。
⌘ AdjustmentEvent（调节事件） 例如，移动滚动条上的滑块。
⌘ ItemEvent（项目事件） 例如，选中项目或放弃选中项目。
⌘ TextEvent（文本事件） 例如，文本内容改变。

3．事件类和事件监听器接口

一个事件类对应一个事件监听器接口，事件监听器接口的命名方法是：去掉事件类后边的“Event”，换成“Listener”。例如，动作事件类取名为 ActionEvent，则相应的动作事件监听器接口名字为 ActionListener。

不同的监听器接口中定义了不同的处理方法。例如，Action Listener 接口中只定义了一个处理方法 ctionPerformed(ActionEvent)。

常用的几种事件类、事件监听器接口和接口中包括的处理方法如表 6.1 所示。

表 6.1 事件类、事件监听器接口和方法

事 件 类	事件监听器接口	接口中定义的方法
ActionEvent	ActionListener	actionPerformed(ActionEvent)
ItemEvent	ItemListener	itemStateChanged(ItemEvent)
MouseEvent	MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
	MouseListener	mousePressed(MouseEvent) mouseReleased(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mouseClicked(MouseEvent)
KeyEvent	KeyListener	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
FocusEvent	FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)

续表

事 件 类	事件监听器接口	接口中定义的方法
WindowEvent	WindowListener	windowClosing(WindowEvent) windowOpened(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent)
TextEvent	TextListener	textValueChanged(TextEvent)

4．Component 类及其子类中的注册方法

前面说过，一个事件处理器创建了以后，要让系统把相应的事件交给该事件处理器来处理，需要在程序中用语句来注册。注册语句的一般格式为：

〈事件源〉.addListener(〈事件处理器〉);

其中，〈事件源〉是用户可能进行某种操作的组件，〈事件处理器〉是实现某种事件监听器接口的类。addListener()是 Component 类中提供的方法。Component 类中提供的 addListener()方法主要有：

⌘ void addKeyListener(KeyListener l)

用键盘事件处理器 l 向键盘事件监听器注册，即要求键盘事件监听器把组件上的键盘事件交给键盘事件处理器 l 处理。这里的 KeyListener 表示实现了 KeyListener 接口的类

⌘ void addMouseListener(MouseListener l)

用鼠标事件处理器 l 向鼠标事件监听器注册

由于 Component 类是其他组件类的父类，所以所有组件类都继承了这些方法。

还有一些注册方法是某些组件类特有的，这些注册方法由这些组件类提供。例如，动作事件的按钮等组件特有的，因此，动作事件的注册方法由这些组件类给出。按钮组件中给出的动作事件注册方法如下：

⌘ void addActionListener(ActionListener l)

用动作事件处理器 l 向动作事件监听器注册

部分常用组件和可以产生的事件的关系如表 6.2 所示。

表 6.2　组件和组件上可以产生的事件

组件	产生的事件类型								
	action	adjustment	focus	item	key	mouse	mousemotion	text	window
Button	×		×		×	×	×		
Choice			×	×	×	×	×		
Container			×		×	×	×		
Dialog			×		×	×	×		×
Frame			×		×	×	×		×
Label			×		×	×	×		

续表

组件	产生的事件类型								
	action	adjustment	focus	item	key	mouse	mousemotion	text	window
List	×		×	×	×	×	×		
Panel			×		×	×	×		
Scrollbar		×	×		×	×	×		
TextArea			×		×	×	×	×	
TextField	×		×		×	×	×	×	
Window			×		×	×	×		×

表中，“×”表示可以产生相应的事件。

5．键盘事件处理

例如，要在程序中处理一个键盘事件，与键盘事件 KeyEvent 相对应的键盘事件监听器接口是 KeyListener，该接口有三个方法：

⌘ void keyPressed(KeyEvent ev)

⌘ void keyReleased(KeyEvent ev)

⌘ void keyTyped(KeyEvent ev)

键盘事件监听器接口中的三个方法对应键盘的三个相应操作：当键盘刚按下去时，调用 keyPressed()方法；当键盘抬起来时，调用 keyReleased()方法；当键盘单击一次时，调用 keyTyped()方法。只要实现了键盘事件监听器接口中的这三个方法，就可以对键盘的三个不同操作（事件）做出相应的处理。

程序可以有几种实现事件监听器接口中方法的方式：

① 在类定义时加 implements 语句，然后在类中实现接口中的方法。例 6.8 程序采用的就是这种方法。

② 用内部类定义和实现事件监听器接口。如要定义和实现一个键盘事件监听器接口类，内部类的语句将为如下形式：

```
Class KeyEventHandler implement KeyListener
{
    void keyPressed(KeyEvent ev)
    {
        处理当键盘刚按下去时产生的事件
    }
    void keyReleased(KeyEvent ev)
    {
        处理当键盘刚松开时产生的事件
    }
    void keyTyped(KeyEvent ev)
    {
        处理键盘单击一次时产生的事件
    }
}
```

设计好了 KeyEventHandle 类以后，就可以把某个组件上产生的键盘事件注册给该类的对象。如把按钮组件上产生的键盘事件注册给 KeyEventHandle 类对象的语句为：

```
Button button = new Button ();                    //创建按钮对象 button
button.addKeyListener(new KeyEventHandler());
//注册，把按钮对象 button 上的键盘事件交给 KeyEventHandler 对象处理
```

6．窗口事件处理

要在程序中处理窗口事件，就必须实现窗口事件处理接口中的所有方法，如果暂时不处理某个窗口操作（事件），可以用一个空语句实现相应的方法。窗口事件对应的窗口事件监听器接口是 WindowListener，其内部类实现方法如下：

```
WindowEventHandler implement WindowListener
{
    void windowClosing(WindowEvent e)                  //关闭窗口事件处理方法
    {
        System.exit();                                 //退出
    }
    void windowOpened(WindowEvent e) { }
    void windowIconified(WindowEvent e) { }
    void windowDeiconified(WindowEvent e) { }
    void windowClosed(WindowEvent e) { }
    void windowActivated(WindowEvent e) { }
    void windowDeactivated(WindowEvent e) { }
}
```

上述窗口事件处理类中，我们仅对一种操作（关闭窗口）进行了处理，对于其他的操作暂时不做处理，所以其他方法都用空语句实现。

7．事件适配器

如果要实现某个事件监听器接口，就必须实现该事件监听器接口中所有的方法，即使有些方法为空，也必须实现。显然，这种实现方法很麻烦，因为有些事件监听器接口中定义的方法很多。设计事件处理类时，即使某些用户操作（事件）不准备处理，也要在事件处理类中把相应的方法实现成空语句。事件适配器可以用来简化这类问题的设计。

Java 语言为一些事件的 Listener（监听器）接口提供了事件适配器（Adapter）类。例如，对于前面讨论过的 WindowListener 接口，相应的适配器类是 WindowAdapter 类。WindowAdapter 类的定义如下：

```
public abstract class WindowAdapter implements WindowListener
{
    public void windowOpened(WindowEvent e) { }
    public void windowClosing(WindowEvent e) { }
    public void windowClosed(WindowEvent e) { }
    public void windowIconified(WindowEvent e) { }
```

```
        public void windowDeiconified(WindowEvent e) {}
        public void windowActivated(WindowEvent e) {}
        public void windowDeactivated(WindowEvent e) {}
        public void windowStateChanged(WindowEvent e) {}
        public void windowGainedFocus(WindowEvent e) {}
        public void windowLostFocus(WindowEvent e) {}
    }
```

可以看出，WindowAdapter 类实现了 WindowListener 接口的所有方法（内容为空）。有了这个适配器，需要实现 WindowListener 接口时，只需要继承 WindowAdapter 类，并根据需要覆盖相应的方法就可以了。这样就避免了实现接口中所有方法的麻烦。例如，如果只需处理关闭窗口的事件，就不必像前面那样，实现 WindowListener 接口中的所有方法，而只需要设计新的方法覆盖 windowClosing()方法就可以了。例 6.8 程序的关闭窗口事件就是用这种方法设计的。

java.awt.event 包中定义的事件适配器类包括以下几个：

- ComponentAdapter　　组件适配器
- ContainerAdapter　　容器适配器
- FocusAdapter　　焦点适配器
- KeyAdapter　　键盘适配器
- MouseAdapter　　鼠标适配器
- MouseMotionAdapter　　鼠标运动适配器
- WindowAdapter　　窗口适配器

6.7 设计举例

本节给出两个包括了本章主要内容的设计举例。

【例 6.9】　设计一个图形用户界面的电话簿。

要求：电话簿保存的内容包括姓名和电话，能够添加新的记录，也能够删除某个不再需要的记录。

程序设计如下：

```
import java.awt.*;
import java.awt.event.*;
public class PhoneBook extends WindowAdapter
        implements ActionListener,ItemListener
{
    Frame f;                                //框架窗口
    TextField tf1,tf2;                      //文本行
    List l;                                 //列表框
    Button b1,b2;                           //按钮

    public PhoneBook()                      //构造窗口界面
```

```
{
    f = new Frame("电话簿");                         //创建框架,默认布局是 BorderLayout
    f.setSize(640,480);                              //设置框架大小

    Panel p = new Panel();                           //创建面板,默认布局是 FlowLayout
    p.add(new Label("姓名"));                        //添加标签
    tf1 = new TextField(10);                         //文本行宽度为 10 个字符
    p.add(tf1);
    p.add(new Label("电话号码"));
    tf2 = new TextField(20);
    p.add(tf2);
    b1 = new Button("添加");
    b2 = new Button("删除");
        b1.addActionListener(this);                  //注册监听按钮的单击事件
        b2.addActionListener(this);
    p.add(b1);
    p.add(b2);
    f.add(p,"North");                                //加在框架的北边

    l = new List();
    l.add("姓名      电话号码");                      //在列表框中加入选项
    l.addItemListener(this);                         //注册监听列表框的单击选中事件
    f.add(l);                                        //在框架中添加列表框

    f.setVisible(true);
    f.addWindowListener(this);                       //注册监听框架的关闭事件
}

public void actionPerformed(ActionEvent e)           //按钮的单击事件
{
        if(e.getSource()==b1)                        //单击"添加"按钮时
    {
            l.add(tf1.getText()+"      "+tf2.getText());
    }
        if(e.getSource()==b2)                        //单击"删除"按钮时
    {
            l.remove(l.getSelectedIndex());          //删除列表框中当前选中项
    }
}

public void itemStateChanged(ItemEvent e)            //列表框的单击事件
{
```

```
            String str = l.getSelectedItem();                    //获得当前选中的数据项
            int i = str.indexOf(' ');                            //获得串中第一个空格的位置
            tf1.setText(str.substring(0,i));                     //设置文本行显示内容
            str = str.substring(i);                              //取从 i 位置开始的子串
            str = str.trim();                                    //去掉串中空格
            tf2.setText(str);
        }

        public void windowClosing(WindowEvent e)                 //关闭框架窗口
        {
                System.exit(0);                                  //关闭窗口
            }

        public static void main(String args[])
        {
            new PhoneBook();
        }
    }
```

程序运行结果如图 6.16 所示。

图 6.16　电话本程序运行结果

【例 6.10】　设计一个图形用户界面的颜色合成程序。

要求：颜色由红、绿、蓝三种基色合成。每种基色的选择范围值为 0～255。用户一旦改变了某种基色的选择值，界面中的配色板就立即显示出新的配色。

程序设计如下：

```
import java.awt.*;
import java.awt.event.*;
public class RGBColor extends WindowAdapter implements TextListener
{
```

```
    Frame f;                                              //框架窗口
    TextField tf1,tf2,tf3;                                //文本行
    Panel p2;
    public RGBColor()                                     //构造窗口界面
    {
        f = new Frame("合成颜色");                        //创建框架
        f.setSize(500,200);                               //设置框架大小

        Panel p1 = new Panel();
        p2 = new Panel();
        f.add(p1,"North");                                //框架上添加面板
        f.add(p2);
        p1.add(new Label("Red"));                         //添加标签
        tf1 = new TextField("255",10);                    //创建文本行时设置初值及宽度
        p1.add(tf1);

        p1.add(new Label("Green"));
        tf2 = new TextField("0",10);
        p1.add(tf2);

        p1.add(new Label("Blue"));
        tf3 = new TextField("0",10);
        p1.add(tf3);

        tf1.addTextListener(this);                        //注册监听文本行的修改事件
        tf2.addTextListener(this);
        tf3.addTextListener(this);

        p2.setBackground(new Color(255,0,0));



        f.setVisible(true);
        f.addWindowListener(this);                        //注册监听框架的关闭事件
    }

    public void textValueChanged(TextEvent e)             //修改文本行时
    {
        int r = (new Integer(tf1.getText())).intValue();
//从文本行获得字符串
        int g = (new Integer(tf2.getText())).intValue();  //转换成整型
        int b = (new Integer(tf3.getText())).intValue();
        if(r)=0 && r <=255 && g > =0 && g <=255 && b > =0 && b <=255)
```

```
            p2.setBackground(new Color(r,g,b));                    //设置面板的背景色
        }

        public void windowClosing(WindowEvent e)                   //关闭框架窗口
        {
            System.exit(0);
        }

        public static void main(String args[])
        {
            new RGBColor();
        }
    }
```

程序运行结果如图 6.17 所示。

图 6.17　颜色合成程序运行结果

习题 6

一、基本概念题

6.1　什么是图形用户界面？它的主要用途是什么？试举例说明。

6.2　什么是 AWT？AWT 有哪些图形组件？

6.3　java.awt 包中包含哪些子包？java.awt 包中包含哪些类？

6.4　Component 类和 MenuComponent 类的关系如何？它们各自的子类有哪些？

6.5　Checkbox 组件是一个复选框，如何用它实现单选的功能？

6.6　Java 有哪些常用的布局管理器？它们各有什么特点？

6.7　Frame 和 Panel 的默认布局管理器是什么？

6.8　Frame 和 Panel 属于哪一类组件，它们的用途有何不同？

6.9　什么叫事件源？什么叫事件？

6.10　简述 Java 语言中的事件处理机制。

6.11　给出鼠标事件监听器接口的完整定义。

6.12　事件适配器类是怎样定义的？它的用途是什么？

6.13　写出事件处理程序必须包括的三个部分。

二、程序设计题

6.14　设计一个如图 6.18 所示的图形用户界面。不要求进行事件处理。

图 6.18　要求的设计界面

6.15　参考 Windows 的记事本界面，设计一个自己的记事本图形用户界面（不必提供操作响应功能）。

6.16　参考 Windows 的计算器界面，设计一个自己的简易计算器图形用户界面（不必提供操作响应功能）。

6.17　利用 AWT 的图形绘制工具和组件绘制一个 9×9 的围棋棋盘。

6.18　设计一个如图 6.18 所示的控制背景色的图形用户界面，当用户选择 List 中的某一种颜色时，界面的背景颜色变为用户选中的颜色。

6.19　设计如图 6.19 所示的图形用户界面。不要求进行事件处理。

提示：最好采用两种布局管理器。

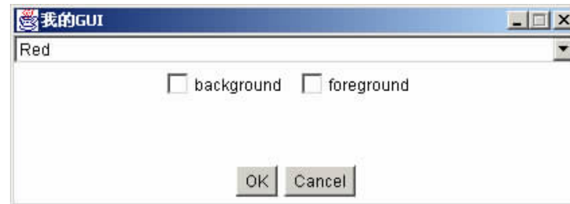


图 6.19　要求的设计界面

6.20　设计一个用户注册的图形用户界面。用户注册信息包括用户名和密码。当用户输入合法的注册信息并单击“确认”按钮时，界面显示注册成功；当用户输入非法的注册信息并单击“确认”按钮时，界面提示注册失败，并指示用户重新注册。

说明：合法的注册信息是指用户按照程序中定义的合法的用户名和密码输入的信息。例如，用户名中字符不能为数字，密码必须为至少包含 4 个字符的字符串。

第 7 章 Java 小程序

教学要点

本章内容主要包括 Java 小程序的基本形式，Java 小程序的设计方法和运行环境，Java 小程序的生命周期，与 Java 小程序相关的 HTML 语言， Java 小程序的安全限制和 JAR 文件。

要求掌握 Java 小程序的基本形式、Java 小程序的生命周期，熟练掌握 Java 小程序的设计方法和运行环境、与 Java 小程序相关的 HTML 语言。

Applet 是 Java 语言中独具特色的重要部分，它体现了 Java 和网络的密切关系。本章根据 Applet 在网络编程中的实际应用，着重介绍了 Applet 的运行机制和 Applet 程序的设计方法，以及图像显示、声音播放和人机交互的 Applet 程序设计方法，最后还简单介绍了 Applet 安全机制和 JAR 文件。

7.1 Java 小程序概述

在 Java 语言中，前几章介绍的能够独立运行的程序称作 **Java 应用程序**。Java 语言还有另外一种程序——Java 小程序。**Java 小程序**（也称 Java Applet）是运行于各种网页文件中，用于增强网页的人机交互、动画显示、声音播放等功能的程序。

Java Applet 能够跨平台地运行于网络中各种不同类型的计算机上。客户端的计算机，只要安装了支持 Java 的浏览器，并且和网络中的服务器建立了连接，就可以从服务器下载嵌入了 Applet 的网页文件，并在本地机的浏览器上运行含有 Applet 的网页。Applet 的工作过程如图 7.1 所示。

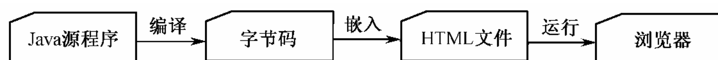


图 7.1 Applet 的工作过程

在安装好的 JDK 中有许多 Java Applet 演示程序。下面，演示一个排序 Applet 的运行过程。JDK 中的演示程序都存放在 JDK 安装目录下的 Demo 子目录中，其中 Applet 演示程序放在 Demo 的子目录 Applet 下。在这个子目录下面有一个 HTML 文件，排序 Applet 就嵌入在这个网页文件中运行。在 DOS 环境下运行这个排序 Applet 的步骤如下：

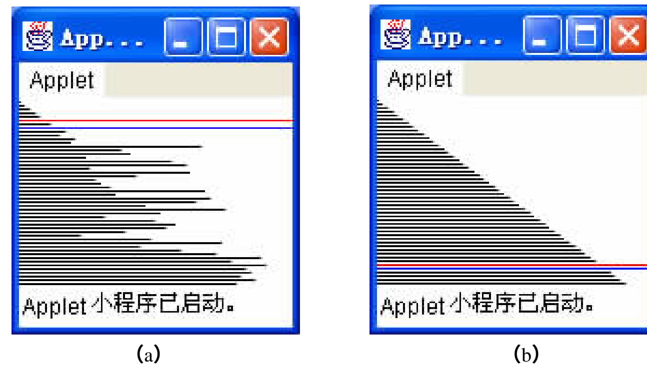
（1）进入 Sort Demo 子目录；

（2）键入 appletviewer example1.html 命令。

说明：appletviewer 命令允许用户脱离浏览器来运行嵌有 Java 小程序的网页文件。这可以大大方便 Java 小程序的调试。本章给出的运行结果图都是用此命令运行的。

在 Windows 环境下运行这个排序 Applet 的方法是：用鼠标双击 SortDemo 子目录中的 example1 页面文件。当然，浏览器必须支持 Applet，否则就只能在 DOS 环境下运行。

程序运行后，初始未排序的显示结果如图 7.2（a）所示。单击窗口界面，排序程序开始执行，排序后的显示结果如图 7.2（b）所示。



(a) (b)

图 7.2　排序 Applet 运行的显示结果

说明：本章的显示结果都是用 appletviewer 命令运行的，如果读者用浏览器运行，其显示结果略有不同。

7.2　Java 小程序的特点、设计方法和运行环境

7.2.1　Java 小程序的特点

Java 小程序是专门为网络而设计的，和前几章讨论的 Java 应用程序相比，Java 小程序在结构和执行过程两个方面具有如下特点：

在结构方面，Java 小程序和 Java 应用程序的不同之处主要表现在：

（1）Java 小程序不能单独运行，它必须依附于一个网页并嵌入其中才能运行，要浏览网页还需要有浏览器；而应用程序是可以独立运行的程序，只要有支持 Java 的虚拟机，它就可以独立运行而不需要其他文件的支持。

（2）Java 小程序必须继承自 Applet 类。Applet 类实现了建立 Java 小程序和浏览器之间联系关系的接口。当然，Applet 类也是 Object 类的子类。

（3）Java 小程序中没有应用程序中的 main()方法，因为 Java 小程序不需要也不能独立运行。

在执行过程方面，Java 小程序和 Java 应用程序的不同主要表现在：Java 应用程序一般是在本地机上运行，而 Java 小程序一般存放在服务器上，它是根据本地机的请求被下载到本地机，然后才在本地机上运行。

7.2.2　Java 小程序的基本设计方法和运行环境

运行 Java 小程序需要有两个文件：一个是由 Java Applet 生成的字节码文件（即“.class”文件），和 Java 应用程序相同，这里的.class 文件也是编译.java 文件后生成的文件。另一个是用于运行 Java 小程序的 HTML 文件。HTML 文件是用来嵌入和运行 Java 小程序的容器。下

面通过一个简单的例子来认识 Java 小程序的设计方法和运行环境。

【例 7.1】 设计一个简单的显示字符串的 Java 小程序，并在网页中运行该 Java 小程序。

设计和运行一个 Java 小程序，一般需要有如下操作步骤：

（1）设计 Java 小程序

Java 小程序设计如下：

```
import java.applet.*;                        //导入 applet 包中的类
import java.awt.*;                           //导入 awt 包中的类
public class WelcomeApplet extends Applet
{
    public void paint(Graphics g)            //绘制图形的方法
    {
        g.drawString( "Welcome to Java Programming!",25, 25 );
                                             //用 Graphics 对象在指定的位置(25,25)绘制字符串
    }
}
```

程序中定义了一个 WelcomeApplet 类，因为它是一个 Java 小程序，所以必须继承 Applet 类。程序中重写了 Component 类的 paint()方法，Graphics 类的对象 g 调用重写的方法，由坐标（25, 25）开始输出了一行字符串“ Welcome to Java Programming! ”。

另外，该 Java 小程序用到了 Applet 类和 Graphics 类，它们分别存放在 java.applet 包和 java.awt 包中。因此，程序需要导入这两个包。

（2）编译 Java 小程序

编译上述 WelcomeApplet.java 文件，生成的字节码文件为 WelcomeApplet.class。

（3）设计 HTML 文件

HTML 文件是用来放置和运行 Java 小程序字节码文件的网页文件。HTML 文件设计如下：

```
〈html〉
〈applet code = "WelcomeApplet.class" width = "300" height = "45"〉
〈/applet〉
〈/html〉
```

HTML 文件的文件名后缀是.html（或.htm）。上述 HTML 文件名为 WelcomeApplet.html。其中的语句含义为：

⌘ code = “ WelcomeApplet.class ”语句告诉浏览器，需要运行的 Java 小程序的字节码文件是“ WelcomeApplet.class ”文件。

⌘ width = "300" height = "45"语句定义了运行 Java 小程序时，Applet 窗口显示区域的高度和宽度分别是 45 和 300 像素。

（4）运行 Java 小程序

在保存上述文件的工作目录下键入命令：

```
appletviewer WelcomeApplet.html
```

或用鼠标双击 WelcomeApplet.html。但两种方法的显示界面略有不同。

（5）运行结果浏览器显示的嵌有 Java Applet 程序的网页显示结果（如图 7.3 所示）

图 7.3　Java Applet 的运行结果

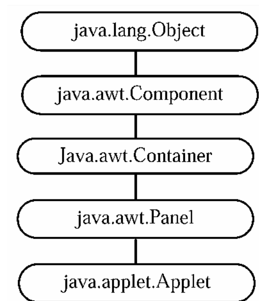


图 7.4　Applet 类的继承关系

7.3　Java 子程序的生命周期

7.3.1　Applet 类的继承关系

Java API 的 Applet 类定义了 Java 小程序和浏览器的接口，Java 小程序只要继承了 Applet 类就可以在浏览器中运行。Applet 类的继承关系如图 7.4 所示。

由于 Applet 类继承了第 6 章讨论的 Component 类、Container 类和 Panel 类，所以可以在 Java 小程序中使用图形组件和处理事件。

7.3.2　Java 子程序的生命周期

Java 子程序的生命周期是指 Java Applet 从创建、启动、停止到消亡的过程。Java Applet 的生命周期如图 7.5 所示。

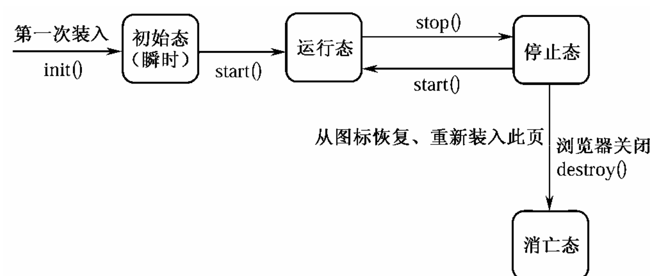


图 7.5　Applet 的生命周期

Applet 类中提供了几个和这个过程相关的方法。这些方法将在 Java Applet 的不同生命阶段被调用。方法如下：

⌘　void init()

Applet 创建阶段调用。它是 Applet 生命的起点，一般在此方法中设置需要的环境变量

⌘　void start()

Applet 启动阶段调用。该方法在 init()方法后被调用，并且每次访问该页面时都会运行此方法

⌘　void stop()

Applet 停止阶段调用。当前的页面被另一页面所替代时，即从作用窗口变成非作用窗口时将调用此方法，在调用 destroy()方法前也会调用此方法

⌘ void destroy()

Applet 退出阶段调用。当浏览器关闭时调用，一般用这个方法来释放和清除分配给 Applet 的系统资源。例如，关闭和数据库的连接等

【例 7.2】 Applet 生命周期中，init(), start(), stop()和 destroy()方法的执行时间示例。

要求：用输出相应字符串的方法来示例 Applet 中 init(), start(), stop()和 destroy()方法的执行时间。

程序设计如下：

```
import java.awt.*;
import java.applet.*;
public class AppletLifeCycle extends Applet
{
    String status = "";                          //成员变量

    public void init()                           //创建时调用此方法
    {
        status+= "Call init() ->";
    }

    public void start()                          //启动时调用此方法
    {
        status+="Call start() ->";
    }

        public void stop()                       //停止时调用此方法
    {
        status+="Call stop() ->";
    }

    public void destroy()                        //退出时调用此方法
    {
        status+="Call destory() ->";
    }

    public void paint(Graphics g)                //被 repaint()调用的方法
    {
        g.drawString(status,20,40);              //绘制字符串
    }
}
```

HTML 文件设计如下：

```
〈html〉
〈applet code = "AppletLifeCycle.class" width = "300" height = "45"〉
〈/applet〉
〈/html〉
```

首次运行 Applet 时，显示结果如图 7.6（a）所示。这说明：首次运行 Applet 时，init()方法和 start()方法被调用。当停止 Applet 后再重新运行，可以发现 stop()方法和 destroy()方法在停止时被调用过。重新运行 Applet 时，显示结果如图 7.6（b）所示。

注意：为了看到图 7.6（b），要把窗口最小化后再最大化，即要刷新窗口。

（a） （b）

图 7.6 运行 Applet 的界面

7.4 HTML 与 Applet

Java Applet 程序必须嵌入在 HTML 文件中才能执行，即 HTML 文件是嵌入和运行 Java Applet 的容器。HTML（Hyper Text Markup Language，超文本标记语言）是一种用来设计网页（或称超文本）的标准语言。

7.4.1 与 Applet 相关的 HTML 属性简介

Applet 是运行在浏览器中的，所以可在 HTML 中定义如何调用 Applet，所有对 Applet 的使用方式都定义在 HTML 文件中的〈APPLET〉和〈/APPLET〉之间。HTML 文件中常用的几个和 Applet 相关的属性如下：

⌘ code = appletFile

该属性是必选属性，该属性提供了 HTML 文件中嵌入的 Applet 的字节码文件名

⌘ width = pixels height = pixels

这两个属性是必选属性，它们提供了 Applet 显示区域的初始宽度和高度（单位为像素）

⌘ alt = message

该属性是可选属性，用来指定当浏览器能识别 Applet 标记但不能运行 Java Applet 时显示的内容。如果一个浏览器不支持 Applet，当运行嵌入了 Applet 的页面文件时，message 部分的信息会被显示出来

⌘ name = className

该属性是可选属性，用来为 Applet 指定一个符号名称，这个名称可以在相同网页的不同 Applet 之间传递参数

⌘ codebase = codebaseURL

该属性是可选属性，用来指定 Java 字节码的路径或 URL。如果未指定该属性，则将使用与 HTML 文档相同的路径。该属性的主要用途是告诉 Java 浏览器，到哪里去寻找不在当前路径下的字节码文件

⌘ align = alignment

该属性是可选属性，该属性指定 Applet 的对齐方式

⌘　param

该属性是可选属性，利用这个属性，可以将 HTML 文件中的参数传递给 Applet

关于 HTML 的其他属性和如何用 HTML 设计网页本章不做讨论，有兴趣的读者可参阅相关文献。

7.4.2　HTML 文件和 Applet 的数据传递

可以在 HTML 文件中用 param 属性来定义一些和 Applet 程序相关的参数和数值，在 Java 小程序中，可以用 getParameter(String name)方法来取得参数 name 的数值。

【例 7.3】　HTML 文件和 Applet 的数据传递示例。

要求：把 HTML 文件中设置的两组数据（"Yaoming", 30）和（"Shark", 40）传递给 Applet，并在 Applet 界面中显示。

HTML 文件设计如下：

```
〈html〉
〈title〉 Applet Parameter Setting 〈/title〉
〈body〉
〈applet
code= AppletParameters.class
name = AppletParameters
width = 280
height = 100
alt = Browser does not support Java Applet
〉
〈param name ="Yaoming" value = "30"〉
〈param name ="Shark" value = "40"〉
〈/applet〉
〈/body〉
〈/html〉
```

HTML 文件设计说明：

① 在 HTML 文件中指定要运行的 Applet 为 AppletParameter.class，图形界面的显示宽度为 280 像素，高度为 100 像素，如果不支持 Applet，运行时会显示“Browser does not support Java Applet”。

② 用属性 param 设置 name 为“Yaoming”的数值为 30，name 为“Shark”的数值为 40。

Java Applet 程序设计如下：

```
import java.awt.*;
import java.applet.*;
public class AppletParameters extends Applet
{
    String score1, score2;
    String message1, message2;
    int s1, s2;
```

```
    public void init()                                   //启动浏览器时调用
    {
        score1 = getParameter("Yaoming");
                                  //取得 Applet 所嵌入的 HTML 文件的参数 Yaoming 的数值
        score2 = getParameter("Shark");
                                  //取得 Applet 所嵌入的 HTML 文件的参数 Shark 的数值
        s1= Integer.parseInt(score1);                    //将字符串 score1 转换成整数
        s2= Integer.parseInt(score2);                    //将字符串 score2 转换成整数
        message1 = "Yaoming scores is:"+score1;
                                                         //构造字符串 message1
        message2 = "Shark scores is:"+score2;            //构造字符串 message2
    }

    public void paint(Graphics g)
    {
        g.drawString(message1,20,40);                    //在(20,40)位置绘制 message1
        g.drawString(message2,20,55);                    //在(20,55)位置绘制 message2
        g.fillRect(150,35,s1,10);
                                  //在(150,35)位置绘制宽为 s1、高为 10 的实心矩形
        g.fillRect(150,50,s2,10);                        //绘制一个实心的矩形
                                  //在(150,50)位置绘制宽为 s2、高为 10 的实心矩形
    }
}
```

程序运行的结果如下图 7.7 所示。

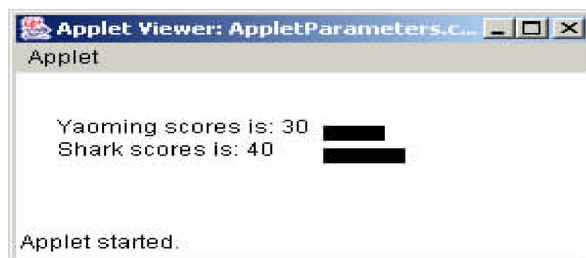


图 7.7　数据传递的运行结果

7.5　两种典型的 Applet 程序设计

前面已经说过，Applet 程序可以为网页加入图像、图形和声音等，还可以给网页增加人机交互的功能。这一节用简单例子讨论在 Applet 中加入图像和进行人机交互的设计方法。在 Applet 中加入图形的设计方法和在 Applet 中加入图像的设计方法类同，Applet 类提供了播放声音的方法 play()，使用该方法就可以在 Applet 中播放声音了。

7.5.1 在 Applet 中加入图像

图像是由不同颜色的像素排列组成的图形。图像文件通常有两种格式，即 GIF 格式和 JPEG 格式。JPEG 格式是一种高效的图片存储格式，而 GIF 格式主要用于存储图标。

Applet 类中常用的获取图像的方法有：

- ⌘ URL getCodeBase()　　返回当前工作目录的 URL 地址
- ⌘ Image getImage(URL url)　　取得 URL 地址 url 的图像
- ⌘ Image getImage(URL url, String name)　　取得 URL 地址 url 的文件名 name 的图像

使用上述方法，可以在 Applet 程序中获取存放在某个网站的某个目录下的图像文件。例如：

```
Image ig=
    getImage(new URL(“http://www.sina.com/test/images/tiger.gif”));
Image ig=
    getImage(new URL(“http://www.sina.com/test”), “images/tiger.gif”);
```

上述两条语句都可以得到新浪网网址 www.sina.com 上 test/images/目录下的图像文件 tiger.gif。

【例 7.4】 在 Applet 中加入和显示当前工作目录下的“3.jpg”图像文件。

Applet 程序设计如下：

```
import java.awt.*;
import java.applet.Applet;
public class ImageApplet extends Applet
{
    Image image;

    public void init()
    {
        image = getImage(getCodeBase(),"3.jpeg");
                        //取得当前工作目录下的图像文件"3.jpeg"的图像并赋值给 image
    }

    public void paint(Graphics g)
    {
        setBackground(Color.lightGray);                   //设置背景颜色
        g.drawString("The following is a image",20,20);
                                                          //在(20,20)位置绘制字符串
        g.drawImage(image,20,30,150,100,this);
                        //在当前组件起点坐标为(20,30)，宽为 150，高为 100 的矩形区域
                                                          //显示图像 image
    }
}
```

HTML 文件“ ImageApplet.html ”设计如下：

```
<html>
〈applet code = "ImageApplet.class" width = "300" height = "150"〉
〈/applet〉
〈/html〉
```

程序运行结果如图 7.8 所示。

图 7.8　Applet 中显示的图像

7.5.2　Applet 中的人机交互

可以通过 Applet 给页面文件增加人机交互的功能。这里的人机交互指的是计算机用户和网页中的 Applet 之间的交互。这是因为 Applet 类继承了第 6 章讨论的组件类，而组件支持事件处理。这里的事件源就是 Applet 中的图形组件，当用户对组件有某种操作产生了事件后，可以用和第 6 章讨论的事件处理方法相同的设计方法处理事件，从而实现用户和网页中的 Applet 之间的交互。

【例 7.5】　嵌有 Applet 的页面和鼠标动作交互的例子。

要求：用鼠标单击页面中的某个位置，在这个位置就出现一个黑色的实心圆。

Applet 程序设计如下：

```
import java.awt.*;                                  //导入组件包中的类
import java.awt.event.*;                            //导入事件包中的类
import java.applet.*;                               //导入 applet 包中的类
public class MouseClickApplet extends Applet implements MouseListener
{
    Point p[];                                      //保存所有鼠标单击过的点的坐标
    int cps;                                        //数组 p 的实际元素个数

    public void init()
    {
        p = new Point[100];
        cps = 0;
            addMouseListener(this);                 //注册事件处理方法
    }
            public void mouseClicked(MouseEvent e)  //单击鼠标时触发
```

```
        {
                p[cps++] =new Point(e.getX(),e.getY());
                                          //取得当前鼠标位置的(x,y)坐标
                repaint();                //调用 paint()方法
        }
        public void paint(Graphics g)
        {
            for (int i = 0; i < cps; i ++)
            {
                g.fillOval(p[i].x, p[i].y, 15,15);
                                  //在坐标位置(p[i].x,p[i].y)绘制宽 15、高 15 的实心圆
            }
        }
        public void update(Graphics g)
        {
            paint(g);                     //重写该方法是为了消除抖动现象
        }

                                          //实现接口 MouseListener 中的所有方法
        public void mousePressed(MouseEvent mep)   {    }
        public void mouseReleased(MouseEvent mer)  {    }
        public void mouseEntered(MouseEvent mee)   {    }
        public void mouseExited(MouseEvent mex)    {    }
        public void mouseDragged(MouseEvent med)   {    }
    }
```

HTML 文件“MouseClickApplet.html”设计如下：

```
〈html〉
〈APPLET CODE=MouseClickApplet WIDTH=300 HEIGHT=150〉
〈/APPLET〉
〈/html〉
```

程序设计说明：程序中用数组 p 来保存所有鼠标单击过的点的坐标，然后调用 paint()方法重绘图形。paint()方法在每一个单击过的点的位置绘制一个实心圆。

浏览器上看到的网页界面如图 7.9 所示。

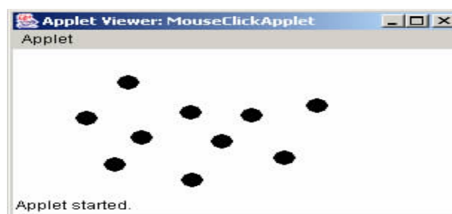


图 7.9　页面和鼠标动作交互运行结果

7.6 Applet 的安全限制和 JAR 文件

7.6.1 Applet 的安全限制

由于 Applet 是从网络上下载到本地机上运行的，如果不对其进行安全方面的限制，它将对互联网上的各种软、硬件资源产生极大的危害。所以，Java 对 Applet 的运行进行了诸多的限制。

Java 对 Applet 最主要的限制如下：

① 不允许 Applet 创建、修改或删除本地机上的文件。

② 不允许 Applet 检查本地机上的文件是否存在。

③ 不允许 Applet 检查目录的内容。

④ 不允许 Applet 读写本地机上的文件。

⑤ 不允许 Applet 检查文件的属性。例如：不允许检查文件的大小、类型和最后更改时间等。

⑥ Applet 不能充当网络服务器，监听或接收来自远程系统的连接请求。

⑦ Applet 不能执行任何本地机上的程序。

⑧ 不允许 Applet 装载动态库或定义本地方法调用。

⑨ 不允许 Applet 在本地机上创建目录。

⑩ 不允许 Applet 关闭本地机上的 Java 虚拟机。

⑪ 不允许 Applet 在本地机上创建对象。

⑫ 不允许 Applet 操纵不在自己线程组中的任何其他线程。

除此之外，在 Java 环境中，对于一些标准的系统属性，Java 应用程序可以通过一些方法来读写这些属性；但对于 Java Applet 而言，许多重要的系统属性都是只允许读、不允许修改，这些属性包括 java.class, java.path, java.home 和 java.version 等。

7.6.2 JAR 文件

前面说过，Applet 程序是嵌入在 HTML 文件中，通过浏览器下载到本地机上运行的。

本地机上下载的内容包括 Applet 字节码、运行它的 HTML 文件及 Applet 要用到的图像文件、声音文件等。有些情况下，在用浏览器浏览网页时，需要建立多次连接才能完成下载（下载完当前需要的文件后，连接会自动关闭，以后再需要某个文件时再重新建立连接），效率很低。另外，有时需要下载的文件很大，花费的时间将会很长。

文件打包技术是解决上述两类问题的一种方法。文件打包就是把若干个文件压缩、打包在一个文件中。Java 中打包后的文件名后缀为.jar，所以打包文件也称为 **JAR** 文件。

用 JDK 制作 JAR 文件的方法是，在 DOS 提示符下键入如下命令：

```
jar cf〈存档文件名〉〈文件 1〉〈文件 2〉…〈文件 n〉
```

其中，选项 c 表示创建一个新的文件，选项 f 表示指定文件的名字；〈存档文件名〉是用户打包的 JAR 文件的文件名；〈文件 1〉，〈文件 2〉，… ，〈文件 *n*〉是要打包的 *n* 个文件的文件名，文件名之间用空格分隔。

例如，对于例 7.4 的例子，就可以把 Applet 字节码文件（ImageApplet.class）和图像文件（3.jpeg）打包成一个 JAR 文件（test.jar）。方法是，在当前工作目录提示符下键入：

```
jar cf test.jar ImageApplet.class 3.jpeg
```

创建了JAR文件后，如何把它嵌入在HTML文件中呢？方法是在HTML文件中用archive属性指定要下载的JAR文件名。例如，对于例7.4的例子，HTML文件应修改为：

```
<html>
<body>
<applet code = "ImageApplet.class"   archive = "test.jar"
        width = "300" height = "150">
</applet>
</body>
</html>
```

JAR命令的选项非常多，cf是最常用的。如果想得到该命令的详细帮助信息，只需在DOS提示符下键入JAR后回车，系统将列出JAR命令的所有选项和含义。

习题 7

一、基本概念题

7.1 什么是Java应用程序？什么是Java小程序（或称作Java Applet）？它们的区别是什么？

7.2 Java Applet的特点是什么？

7.3 编写和运行Java Applet的步骤是什么？

7.4 Java Applet在它的生命周期中都调用哪些方法？这些方法的作用是什么？

7.5 Applet和AWT图形用户界面有什么异同？它们之间的关系如何？

7.6 什么叫HTML？写出在HTML文件中嵌入Applet时用到的属性。

7.7 Applet有哪些安全限制？

7.8 JAR文件是一种什么文件？JAR文件的作用是什么？写出一个在HTML文件中嵌入JAR文件的例子。

二、程序设计题

7.9 编写一个Applet程序和相应的页面文件，绘制奥运五环标志图形。

7.10 编写一个Applet程序和相应的页面文件，通过页面文件传递参数，在Applet中绘制一个长方形（长方形的长度和宽度都由页面文件传递）。

7.11 编写一个Applet程序，试用CheckBox组件来控制文本输入行中的字体。用户界面如图7.10所示。

图7.10 用户界面

7.12 把习题6.14要求的图6.18所示的图形界面用Applet实现。

7.13 设计一个人机交互的Applet，在Applet上画出一个人的头像（用简单的图形表示人的头像），头像下方放置两个按钮，单击“增长”按钮时鼻子增长，单击“缩短”按钮时鼻子缩短。

7.14 把习题6.19要求的图6.19所示的图形界面用Applet实现。

第 8 章　异 常 处 理

教学要点

本章内容主要包括异常的概念（包括异常的基本类型和 Java 的异常处理方法），Java 的异常类，Java 异常的处理方法，两种 Java 的异常抛出和处理方式，自定义异常类。

要求掌握异常的概念、Java 的异常类、自定义异常类，熟练掌握 Java 异常的处理方法、两种 Java 的异常抛出和处理方式。

任何一个软件或程序都可能在运行的过程中出现故障，问题的关键是故障出现以后如何处理，谁来处理，怎样处理。以及处理后系统能否恢复正常的运行。本章在介绍 Java 处理这类问题基本方法的基础上，讨论包含异常处理的 Java 程序的设计方法。

8.1　异常和异常处理的两种方法

异常是指程序在执行过程中出现的意外事件。异常通常会使程序的正常流程被打断。例如，程序在向磁盘中读写文件时磁盘出现问题、算术运算中被除数为 0、数组下标越界、输入/输出时文件不存在、输入数据格式不正确、程序本身错误，等等。

一般情况下，程序中需要对异常情况进行处理。通过对异常情况的处理，可以使程序的执行流程继续下去，并进行一些新的异常处理；否则，程序的正常执行流程会被中断。

8.1.1　异常的基本类型

异常处理的第一步是确定异常的类型，常见的异常情况一般分为以下几类：

① 用户输入错误。指用户键盘输入错误，输入格式不对或输入内容不符合程序要求等。例如，用户界面要求输入一个整数，而用户输入了一个字符串；或者要求用户输入一个 URL 地址，但用户输入的地址语法错误等。

② 设备故障。计算机硬件设备有时也会出故障，如打印机没连接好，要求的网页没有找到等。

③ 物理限制。物理设备本身的限制，如硬盘已存满，内存已用完等。

④ 代码错误。程序员编写的代码出现错误，如从一个空的堆栈中弹出元素，数组下标为负数等。

8.1.2　if-else 形式的异常处理方法

用 if-else 语句（或 switch-case 语句）可以发现异常并做出相应处理。例如，在程序中要将一个文件全部读入内存，通常的步骤是：

① 打开文件；
② 判断文件的大小；
③ 分配内存；
④ 将文件读入内存；
⑤ 关闭文件。

在上述的过程中可能出现很多问题，例如：

① 文件打不开；
② 不能决定文件的大小；
③ 内存不能分配；
④ 读入失败；
⑤ 文件不能关闭。

针对这些情况，需要在程序中做出相应的处理，若用 if-else 语句，则程序伪码如下：

```
Class ReadFile
{
    初始化 错误代码 = 0;
    打开文件;
    if (文件可以打开)
    {
        计算文件的长度;
        if (文件的长度可以决定)
        {
            分配内存
            if (有足够的内存)
            {
                将文件读入内存
                if (读入失败)
                {
                    错误代码 = "读入文件时失败"
                }
            }
            else
            {
                错误代码 = "内存不足"
            }
        }
        else
        {
            错误代码 = "不能决定文件的大小"
        }
        关闭文件;
        if (文件没有关闭 && 错误代码 == 0)
```

```
        {
            错误代码 = “文件关闭失败”
        }
    }
    else
    {
        错误代码 = “文件打不开”
    }
    return 错误代码
}
```

用 if-else 语句方式可以发现异常并做出相应的处理。但是，如果程序员过多地分析程序中异常情况的发生情况，在程序中过多地使用 if-else 语句，会使程序员正常的程序设计思路受到影响。另外，程序中过多地使用 if-else 语句，也会影响程序的可读性。而且，更为重要的是，有些异常情况是不可预见的。如计算机的连接中断，打印机纸张用完等。所以，if-else 形式不是处理异常最好的方法。

8.1.3 Java 的异常处理方法

在 Java 语言中，用 try 模块和 catch 模块把程序的正常流程代码和异常处理代码分离。对于上述例子，按照 Java 的异常处理方式，伪码表示的程序代码如下：

```
Class ReadFile
{
    try
    {
        打开文件
        计算文件的长度
        分配内存
        将文件读入内存
        关闭文件
    }
    catch (文件打开失败异常)
    {
        处理文件打开失败异常
    }
        catch (文件的长度不能确定异常)
    {
        处理文件的长度不能确定异常
    }
        catch (分配内存失败异常)
    {
        处理分配内存失败异常
    }
```

```
        catch (文件读入内存失败异常)
    {
        处理文件读入内存失败异常
    }
        catch (关闭文件失败异常)
    {
        处理关闭文件失败异常
    }
}
```

从上述伪码可以看出，程序中可能出现的异常情况都放进了 try 模块中，而对于各种异常情况，设计了相应的 catch 模块，这些模块可以用来捕捉这些异常情况，并进行相应的处理。

在 Java 程序中，如果设计人员对可能出现的异常没有设计相应的 try 模块和 catch 模块，或出现设计人员无法预见的异常情况，系统会将出现的异常交由 Java 虚拟机（JVM）处理。此时，Java 虚拟机会自动捕捉这些异常情况，并将异常情况在屏幕上显示出来。

8.2 Java 的异常类

Java 语言对大多数常见的异常都定义了异常类。这些异常类可以分为两大类：Error 类和 Exception 类。Error（错误）类和 Exception（异常）类的区别在于：错误指的是系统异常或运行环境出现的异常，这些异常一般是很严重的异常，即使捕捉到通常也无法处理，如 Java 虚拟机异常；而 Exception 类的异常指的是一般的异常，如输入/输出（I/O）异常、数据库访问（SQL）异常等，对这些异常应用程序可以进行处理。Java 的所有异常类都继承自 Throwable 类。异常类是 Java 语言包（java.lang）中的类。

Java 异常类的继承关系如图 8.1 所示。

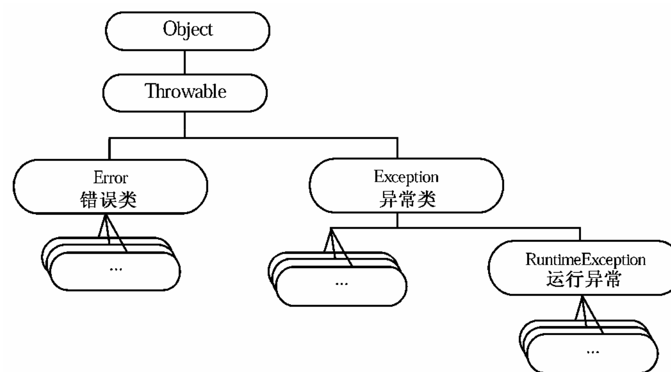


图 8.1 Exception 类和 Error 类的继承关系图

1．Error 类及其子类

当系统动态连接失败，或者出现虚拟机的其他故障时，Java 虚拟机会抛出 Error 错误。程序一般不用捕捉 Error 错误，由系统进行这类错误的捕捉和处理。Error 类及其子类定义了

系统中可能出现的大多数 Error 错误。

这里介绍 Error 类的两个子类及其子类：LinkageError（结合错误）类及其子类和 VirtualMachineErrror（虚拟机错误）类及其子类。

（1）LinkageError 类及其子类

LinkageError（结合错误）类及其子类定义的是一个类依存于另一个类，但在编译前者时后者出现了与前者不兼容情况的各种错误。

LinkageError 类的子类有：

⌘ ClassFormatError	类格式错误
⌘ ClassCircularityError	类循环错误
⌘ NoClassDefFoundError	类定义无法找到错误
⌘ VerifyError	校验错误
⌘ AbstractMethodError	抽象方法错误
⌘ NoSuchFieldError	没有成员变量错误
⌘ InstantiationError	实例错误

（2）VirtualMachineErrror 类及其子类

VirtualMachineErrror（虚拟机错误）类及其子类定义的是系统在虚拟机损坏或需要运行程序的资源耗尽时出现的各种错误。

VirtualMachineErrror 类的子类有：

⌘ InternalError	内部错误
⌘ OutOfMemoryError	内存溢出错误
⌘ StackOverflowError	堆栈溢出错误
⌘ UnkownError	未知错误

2．Exception 类及其子类

Exception 类及其子类定义了程序中大多数可以处理的异常。

这里介绍 Exception 类的两个子类及其子类：RuntimeException（运行时异常）类及其子类和 CheckedException（检查异常）类及其子类。

（1）RuntimeException 类及其子类

RuntimeException（运行时异常）类及其子类定义的是 Java 程序执行过程中可能出现的各种异常。RuntimeException 类的子类有：

⌘ ArithmeticException	算术运算异常
⌘ ArrayStoreException	数组存储异常
⌘ ArrayIndexOutOfBoundsException	数组下标越界异常
⌘ CaseCastException	类型转换异常
⌘ IllegalArgumentException	非法参数异常
⌘ IllegalThreadStateException	非法线程状态异常
⌘ NumberFormatException	数字格式异常
⌘ IlegalMonitorStateException	非法监视状态异常
⌘ IndexOutofBoundsException	下标超出范围异常
⌘ NegativeArraySizeEXception	负数组个数异常

⌘ NullPointerException　　空指针异常
⌘ SecurityException　　安全异常
⌘ EmptyStackException　　空栈异常
⌘ NoSuchElementException　　没有元素异常

（2）CheckedException 及其子类

CheckedException （检查异常）类及其子类定义的是 Java 程序编译时编译器发现的各种异常。

CheckedException 类的子类有：

⌘ ClassNotFoundException　　类找不到异常
⌘ CloneNotSupportedException　　复制不支持异常
⌘ IllegalAccessException　　非法访问异常
⌘ InstantiationException　　实例异常
⌘ InterruptedException　　中断异常
⌘ IOException　　输入/输出异常
⌘ FileNotFoundException　　文件找不到异常
⌘ InterruptedIOException　　中断输入/输出异常

3．Throwable 类的方法

Error 类和 Exception 类的方法基本都是从 Throwable 类中继承来的。

Throwable 类的构造方法主要有：

⌘ Throwable()　　构造一个对象，其错误信息串为 null
⌘ Throwable(String message)　　构造一个对象，其错误信息串为 message

Throwable 类的方法主要有：

⌘ String getMessage()　　返回对象的错误信息
⌘ void printStackTrace()　　输出对象的跟踪信息到标准错误输出流
⌘ void printStackTrace(PrintStream s)　　输出对象的跟踪信息到输出流 s
⌘ String toString()　　返回对象的简短描述信息

4．异常类的对象

应用程序中一旦出现异常，系统会产生一个异常类的对象，下面讨论的 catch 语句后面的参数将接收到这样一个对象。

为了语言简洁，通常把异常类的对象也简称异常。这就像第 6 章中，也把组件类的对象简称组件。

8.3　Java 的异常处理方法

在 Java 程序中，用 try-catch（或 try-catch-finally）语句来抛出、捕捉和处理异常。

try-catch-finally 语句的语法格式是：

```
try
{
```

```
    可能抛出异常的语句模块；
}

catch （异常类型 1 ）
{
    处理异常类型 1 语句；
}
……
catch （异常类型 n ）
{
    处理异常类型 n 语句；
}
finally
{
    无论是否抛出异常都要执行的语句；
}
```

try-catch-finally 语句的功能为：

（1）try 模块

所有可能抛出异常的语句都放入 try 模块中。try 模块中的语句是程序正常流程要执行的语句，但是在执行过程中有可能出现异常。如果像前面程序设计例子那样没有使用 try 模块，则除了系统定义的异常外，语句执行过程中出现的其他异常不会被抛出。

（2）catch 模块

主要负责对抛出的异常做相应的处理。所有抛出的异常都必须是 Throwable 类或其子类的对象。try 模块中的语句可能抛出很多不同类型的异常，所以需要针对不同类型的异常分别设计 catch 模块。一般程序中会有若干个 catch 模块，每一模块处理一种类型的异常。

对于这些 catch 模块，其排列的先后顺序有如下规定：出现在前面的 catch 模块中的异常类一定要是后面的 catch 模块中的异常类的子类，即前面的 catch 模块中的异常类对象一定比后面的 catch 模块中的异常类对象特殊。这是因为，catch 模块是按顺序执行的。如果 try 模块抛出异常，系统会按 catch 模块的排列顺序依次查找，如果前面 catch 模块的异常类型匹配不上，就按顺序和后面的 catch 模块匹配。如果在所有的 catch 模块中都匹配不上，就交给系统的虚拟机，由虚拟机来处理这个异常。

程序中即使没有 try 模块，若出现了系统定义的异常，系统也会自动抛出，并由系统负责捕捉和处理；但是程序中如果有 catch 模块，则一定要有 try 模块。另外，如果用户希望按照自己的意图处理程序中可能出现的系统定义的异常，也要在程序中使用 try-catch 语句，否则出现的异常将由系统捕捉和处理。

（3）finally 模块

finally 模块中的语句是必须执行的语句。无论 try 模块中是否抛出异常，finally 模块中的语句都要被执行。这个模块是可选的。如果设计了 finally 模块，通常在这个模块中做一些资源回收的工作。

【例 8.1】 try-catch-finally 的执行过程示例。

要求：设计一个用命令行参数输入数据的程序，若该参数的数值输入不正确会出现异常。

程序设计如下：

```
public class TryCatchSequence
{
    public static void main(String args[])
    {
        int a,b,c;
        try
        {
            a=100;
            b=Integer.parseInt(args[0]);             //把命令行输入的字符串转换成整数
            c=a/b;
            System.out.println("c="+c);
            System.out.println("No exception.");
        }
        catch(ArrayIndexOutOfBoundsException e)       //数组下标越界异常
        {
            System.out.println("Exception caught!");
            e.printStackTrace();
        }
        catch(ArithmeticException e)                  //处理算术运算异常
        {
            System.out.println("Exception caught!");
            e.printStackTrace();
        }
        finally
        {
            System.out.println("Always executed");
        }
    }
}
```

下面分几种情况分别运行上述程序：

（1）没有异常抛出

在当前工作目录下键入如下命令（注意：该程序正常运行需要从命令行给出参数，且该参数值不能为 0）：

```
java TryCatchSequence 10
```

则程序的运行结果如下：

```
c=10
No exception.
Always executed
```

程序的运行结果说明：系统执行完了 try 模块的所有程序，由于程序运行时没有异常抛出，所以系统跳过 catch 模块，最后执行了 finally 模块。

（2）有异常抛出

上述程序可能有如下两种类型的异常发生。

① 抛出数组下标越界异常。在当前工作目录下键入如下命令（注意：该程序运行时无法从命令行得到参数值）：

```
java TryCatchSequence
```

程序运行结果如下：

```
Exception caught!
java.lang.ArrayIndexOutOfBoundsException: 0
    at TryCatchSequence.main(TryCatchSequence.java:9)
Always executed
```

程序的运行结果说明：由于命令行没有给出参数，在执行语句：

```
b=Integer.parseInt(args[0]);
```

时，系统抛出了数组下标越界异常。程序中处理数组下标越界异常的 catch 模块匹配上了该异常。程序执行完该 catch 模块语句后，跳过后面的其他 catch 模块，又执行了 finally 模块中的语句。

② 抛出算术运算异常。在当前工作目录下键入如下命令（注意：该程序运行时从命令行得到的参数值为 0）：

```
java TryCatchSequence 0
```

程序运行结果如下：

```
Exception caught!
java.lang.ArithmeticException: / by zero
    at TryCatchSequence.main(TryCatchSequence.java:10)
Always executed
```

程序的运行结果说明：由于命令行给出的参数为 0，程序中的被除数为 0，因此运行时系统抛出了算术运算异常。程序中处理算术运算异常的 catch 模块匹配上了该异常。程序执行完该 catch 模块语句后，跳过后面的其他 catch 模块，接着执行 finally 模块中的语句。

【例 8.2】 程序中设计的 catch 模块捕捉不到异常的示例。

要求：程序中设计了若干 catch 模块，但忽略了其中某种可能发生的异常。

程序设计如下：

```
public class TryCatchSequence1
{
    public static void main(String args[])
    {
        int a,b,c;
        try
```

```
        {
            a=100;
            b=Integer.parseInt(args[0]);
            c=a/b;
            System.out.println("c="+c);
            System.out.println("No exception.");
        }
        catch(ArithmeticException e)
        {
            System.out.println("Exception caught!");
            e.printStackTrace();
        }
        finally
        {
            System.out.println("Always executed");
        }
    }
}
```

程序设计说明：上述程序和例 8.1 的程序基本相同，只是去掉了前面程序的 catch(ArrayIndexOutOfBoundsException e)模块。

程序运行结果如下：

```
Always executed
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at TryCatchSequence2.main(TryCatchSequence2.java:9)
```

程序的运行结果说明：程序中出现的 ArrayIndexOutOfBoundsException 异常不能和程序中的任何 catch 模块匹配，因此交给系统处理，系统内置的异常处理模块输出了相应的错误信息。

8.4 异常的抛出和处理

系统的异常类定义了很多异常，如果程序运行时出现了系统定义的异常，系统会自动抛出。此时，若应用程序中有 try-catch 语句，则这些异常由系统捕捉并交给应用程序处理；若应用程序中没有 try-catch 语句，则这些异常由系统捕捉和处理。

对于系统定义的有些应用程序可以处理的异常，一般情况下并不希望由系统来捕捉和处理，也不希望这种异常造成用户不希望的后果，因为这两种情况都有可能造成运行的应用程序产生不良后果。这种情况下，设计应用程序的一般方法是：在 try 模块中，应用程序自己判断是否有异常出现，如果有异常出现，则创建异常对象并用 throw 语句抛出该异常对象；在 catch 模块中，可以设计用户自己希望的异常处理方法。

throw 语句的语法格式为：

throw〈异常类对象〉;

这种情况下，异常的抛出和处理有两种方式：一种是在同一个方法中抛出异常和处理异常，另一种是在一个方法中抛出异常，在调用该方法的方法中处理异常。

8.4.1 在同一个方法中抛出异常和处理异常

应用程序中，通常采用在同一个方法中抛出异常和处理异常的设计方法。

【例 8.3】 在同一个方法中抛出异常和处理异常示例。

要求：设计一个堆栈类，堆栈类中要求有一个入栈方法，该方法向堆栈中加入一个元素。当调用入栈方法且堆栈已满时，会出现堆栈已满异常。处理异常的模块捕捉到该异常后，显示异常信息并退出系统。

程序设计如下：

```
class SeqStack                                      //堆栈类
{
    int data[];                                     //成员变量，存放元素
    int MAX;                                        //成员变量，数组个数
    int top;                                        //成员变量，当前元素个数

    SeqStack(int m)                                 //构造方法
    {
        MAX = m;                                    //为 MAX 赋值 m
        data = new int[MAX];                        //创建数组对象 data
        top = 0;                                    //初始值为 0
    }

    public void push(int item)                      //入栈方法
    {
        try                                         //try 模块
        {
            if(top == MAX)                          //判断是否有异常产生
                throw new Exception("stackFull");   //抛出异常对象
            data[top] = item;                       //在堆栈中加入元素 item
            top++;                                  //元素个数加 1
        }
        catch(Exception e)                          //catch 模块
        {
            System.out.println("exception:"+e.getMessage());
            System.exit(0);
        }
    }

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

```
            SeqStack stack = new SeqStack(10);              //创建对象 stack
            for(int i = 0; i <11; i++)
                stack.push(i);                              //调用 push()方法
        }
    }
```

程序运行结果如下：

```
exception:stackFull
```

程序设计说明：

① main()中第一条语句创建的对象 stack 的数组个数 MAX 为 10，但循环语句最后一次执行时，要向对象 stack 的数组 data 中加入第 11 个元素，所以此时会产生异常。

② 由于 push()方法中的异常情况（top == MAX）不是系统定义的异常，所以，系统不会抛出该异常，需要在该方法中用 throw 语句自己抛出异常。

③ 抛出异常语句 throw new Exception("stackFull")首先调用 Exception 类的构造方法创建一个异常信息为“ stackFull ”的异常对象，然后抛出该异常。

④ push()方法中的 catch 模块捕捉到异常后进行所要求的异常处理。

8.4.2 抛出异常和处理异常的方法不是同一个方法

如果抛出异常的方法和处理异常的方法不是同一个方法时，则要求在抛出异常的方法定义后加如下语句：

```
throws Exception
```

然后，把 catch 模块放在调用该方法的方法中。

throws Exception 语句的功能，是在调用方法和可能产生异常的被调用方法之间建立起系统处理异常所需的联系。

【例 8.4】 抛出异常和处理异常的方法不是同一个方法示例。

要求：问题同例 8.3，但要求抛出异常和处理异常的方法不是同一个方法。

程序设计如下：

```
class SeqStack2
{
    int data[];
    int MAX;
    int top;

    SeqStack2(int m)
    {
        MAX = m;
        data = new int[MAX];
        top = 0;
    }
```

```
    public void push(int item) throws Exception            //加 throws 语句
    {
        if(top == MAX)                                      //判断是否有异常产生
            throw new Exception("stackFull");               //抛出异常
        data[top] = item;
        top++;
    }

    public static void main(String args[])
    {
        SeqStack2 stack = new SeqStack2(10);
        try                                                 //try 模块
        {
            for(int i = 0; i <11; i++)
                stack.push(i);
        }
        catch(Exception e)                                  //catch 模块
        {
            System.out.println("exception:"+e.getMessage());
            System.exit(0);
        }
    }
}
```

程序运行结果和例 8.3 相同。

需要说明的是，此例子只是为了说明此种设计方法，像例 8.3 这类问题应用程序的异常抛出和异常处理通常采用前一种设计方法。

但是，Java API 中的许多可能出现异常的方法都采用此种设计方法。例如，第 9 章讨论的输入/输出流类中的许多方法在调用时都可能出现异常，而调用这些方法的方法是在用户编写的应用程序中，Java API 不可能设计出适合所有应用情况的异常处理方法，因此，Java API 的这些方法中没有设计 try 模块和 catch 模块，只是在方法定义后添加了 throws 语句。如下方法定义就是一个例子：

```
public void close()    throws IOException              //表示 close()方法可能抛出IOException异常
```

如果应用程序调用该方法，一般情况下，要在调用方法中设计 try 模块和 catch 模块，来处理IOException异常。IOException类是 Exception 类的一个子类。

8.5 自定义的异常类

应用程序中除了可能出现系统定义的异常外，有时还可能出现系统没有考虑的异常。此时需要在应用程序中自定义异常类。一般情况下，自定义的异常类都是一些应用程序可以处理的异常。所以，自定义的异常类一般是 Exception 类的子类。

【例 8.5】 自定义的异常类示例。

```
class UserDefinedException extends Exception
{
    String  msg;
    public UserDefinedException()
    {
        this.msg = "";
    }
    public UserDefinedException(String s)
    {
        this.msg = s;
    }
}
```

例 8.5 中，用户自己定义的异常类 UserDefinedException 继承了类 Exception，类 UserDefinedException 中定义了两种情况的异常，一种情况参数为空，另一种情况参数为字符串 s。

习题 8

一、基本概念题

8.1 什么是异常？举出程序中常见异常的例子。

8.2 Java 的异常处理方法有什么优点？

8.3 如果在 Java 程序中不对出现的异常进行处理，程序运行时若出现异常会发生什么情况？

8.4 Java 的异常类中，Throwable 类、Error 类和 Exception 类之间的关系如何？什么叫错误？什么叫异常？

8.5 在 Java 中，throw 和 throws 有何不同？它们各自用在什么地方？

8.6 简述 try-catch-finally 语句的功能。

8.7 简述 try-catch-finally 语句的执行顺序。

8.8 catch 模块的排列顺序对异常处理有什么影响？

8.9 在什么情况下，try 模块和 catch 模块设计在一个方法内？在什么情况下，try 模块和 catch 模块不设计在一个方法内？

8.10 举例说明如何定义一个新的异常类？如何使用这个异常类？

二、程序设计题

8.11 设计一个 Java 程序，程序中要进行数组操作和除法运行，所设计的程序要对可能出现的异常进行处理。

8.12 设计一个 Java 程序，该程序应说明异常处理的 catch 模块排列顺序的重要性。

8.13 重写下面方法，该方法自己不处理异常，而是只抛出异常，让调用方法自己处理异常。

```
int division (int c)
{
    try
    {
        int a = 100/c
        System.out.print("100/c="+a);
    }
    catch (ArithmetiExeption e)
    {
        e.printStackTrace();
    }
}
```

8.14 **设计一个堆栈类，堆栈类中要求有入栈方法和出栈方法。入栈方法是向堆栈中加入一个元素，出栈方法是从堆栈中取出一个元素。当调用入栈方法且堆栈已满时会出现堆栈已满异常，当调用出栈方法且堆栈已空时会出现堆栈已空异常。要求分别用** 8.4 **节讨论的两种方法捕捉和处理异常，异常处理为显示异常信息后退出系统。最后，设计一个测试主方法进行测试。**

8.15 **定义一个邮件地址异常类，当用户输入个邮件地址不合法时，抛出异常。**

8.16 **定义一个数学运算的方法，此方法在特定的情况下可能抛出异常。设计一个测试程序，在程序中调用这个数学运算方法。**

第 9 章　输入/输出流

教学要点

本章内容主要包括数据流的概念（包括输入流、输出流、字节流、字符流），InputStream 类、OutputStream 类、Reader 类、Writer 类，以及这些类的常用子类的功能和设计方法，对象流的功能和基本使用方法，文件的操作方法。

要求掌握数据流的概念、对象流的功能和基本使用方法，熟练掌握 InputStream 类、OutputStream 类、Reader 类、Writer 类，以及这些类的常用子类的功能和设计方法、文件的操作方法。

在网络环境下，现在的计算机程序不仅要和用户交互数据，而且要和网络上的某个服务器或主机交互信息。Java API 的输入/输出流就是为此目的设计的。本章在介绍 Java 字节输入/输出流和字符输入/输出流的基础上，讨论了字节文件输入/输出的基本设计方法和字符文件输入/输出的基本设计方法,以及随机存取文件的输入/输出设计方法和对文件的操作方法。另外，字节输入/输出流和字符输入/输出流也是进行复杂输入/输出（如对象流、数据流、管道流等）的基础，本章还介绍了对象流的基本设计方法。

9.1　数据流的概念

9.1.1　输入流和输出流

在 Java 中，把所有输入和输出都当作流来处理。流是按一定顺序排列的数据的集合。例如，字符文件、声音文件或图像文件等都可以看作是一个个的数据流。输入数据时，一个程序打开数据源上的一个流（文件或内存等），然后按顺序输入这个流中的数据，这样的流称为**输入流**，其过程如图 9.1（a）所示。输出数据时，一个程序可以打开一个目的地的流，然后按顺序从程序向这个目的地输出数据，这样的流称为**输出流**，其过程如图 9.1（b）所示。输入和输出的方向是以程序为基准的。向程序输入数据的流定义为输入流，从程序输出数据的流定义为输出流。通常，也把从输入流中向程序中输入数据称为**读数据**（read），反之，从程序中将数据输出到输出流中称为**写数据**(write)。

无论是读数据还是写数据，也不管数据流是何种类型，读数据或写数据的算法都是基本相同的，其具体步骤一般为：

```
打开一个流
while（数据存在时）
    读数据或写数据
关闭流
```

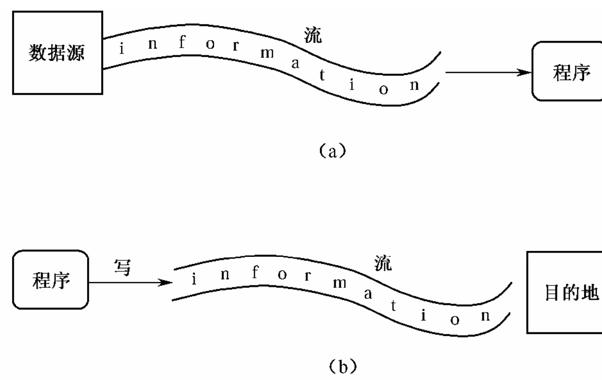


图 9.1　输入流和输出流

9.1.2　字节流和字符流

Java 的输入/输出流中，根据它们的数据类型，主要可分为两类：字节流和字符流。

1．字节流

字节流是按字节读/写二进制数据。在 Java 中，字节流有两个基本的类：InputStream 类和 OutputStream 类。InputStream 类和 OutputStream 类分别为字节型输入/输出流提供了读/写字节码的基本方法。

字节流主要用于读/写二进制数据，例如，图像或声音。

2．字符流

字符流的输入/输出数据是字符码，即 Unicode 字符（当遇到不同编码的字符时，Java 的字符流会自动将其转换成 Unicode 字符）。字符流有两个基本类：Reader 类和 Writer 类。Reader 类和 Writer 类分别为字符型输入/输出流提供了读/写字符的基本方法。

9.1.3　Java 的标准数据流

Java 的标准数据流指程序和 DOS 交互时的输入/输出方式。java.lang 包的 System 类定义了三个成员变量，分别是：

- ⌘ static final InputStream in　　标准输入 in
- ⌘ static final PrintStream out　　标准输出 out
- ⌘ static final PrintStream err　　标准错误输出 err

因为这三个成员变量都是静态（static）成员变量，即类成员变量，所以可以直接使用，不需要创建对象。in 定义为下面要讨论的 InputStream 类的 static 成员变量，所以可以直接调用 InputStream 类提供的输入方法；out 和 err 定义为下面要讨论的 PrintStream 类的 static 成员变量，所以可以直接调用 PrintStream 类提供的输出方法。

out 用于屏幕输出，常用的调用方法有：

- ⌘ System.out.print(String str);　　向屏幕输出字符串 str
- ⌘ System.out.println(String str);　　向屏幕输出字符串 str 后换行

这两个语句已在前面各章的程序中多次使用。

in 用于屏幕输入，常用的调用方法有：

⌘ System.in.read() 返回从键盘输入的字符

⌘ System.in.read(byte[] b) 从键盘输入多个字符到数组 b，并返回字符个数

例如，要想读入一个长度小于 512 的字符串到字符数组 b，可以使用下面语句：

```
byte b = byte b[512];
int c = System.in.read(b);
```

由于 Java 程序大多使用图形用户界面方式输入数据，系统也没有提供高效且使用方便的标准输入方法，因此，应用程序中一般很少使用屏幕输入方法来输入数据。

err 和 out 类似，主要是系统输出错误信息时使用。

9.2 基本输入/输出类

在 Java 中，输入/输出类和接口都在 java.io 包中。各种功能的输入/输出类和接口非常之多，作为初学者，应该首先掌握一些基本输入/输出类，因为很多类都是由这些基本类继承来的。

Java 有四个最基本的输入/输出类：InputStream 、OutputStream、Reader 和 Writer。前面已经简单介绍过，InputStream 和 OutputStream 用于字节流的输入/输出，Reader 和 Writer 用于字符流的输入/输出。下面分别对它们中的每一个类及主要子类做一介绍。

9.2.1 InputStream（字节输入流）类

InputStream 类称作字节输入流类，是一个抽象类。InputStream 类为其他输入字节流子类提供了一些基本方法和标准接口。由于 InputStream 类是一个抽象类，所以它本身不能直接用来创建对象。

1．InputStream 类的常用方法

InputStream 类提供的方法主要有：

⌘ void close() 关闭输入流

⌘ void mark(int readlimit) 标记输入流的当前位置

⌘ boolean markSupport()
测试是否支持 mark()方法和 reset()方法，true 为支持，false 为不支持

⌘ int read(type[] b) 从输入流读若干个字节到数组 b，返回字节个数

⌘ int read(byte[], int off, int len)
从输入流读若干个字节到数组 b，起始位置为 off，长度为 len，返回字节个数

⌘ void reset() 复位，即将输入流的当前位置置为初始值

⌘ long skip(long n) 跳过 n 个字节，返回跳过的字节个数

⌘ abstract int read() 抽象方法，从输入流读下一个字节

其中，abstract int read()方法为抽象方法。因为抽象类的功能和接口的功能非常类同，所以我们说，InputStream 类不仅为其他输入字节流子类提供了一些基本方法，还为它们提供了标准接口。InputStream 类的子类重写了不同功能的 read()方法。

2．异常的抛出

InputStream 类中的许多方法在调用时有可能出现异常，例如，调用 close()方法关闭文件时，文件因种种原因有可能关闭不了，因此定义的这些方法都有 throws IOException 语句。所以，close()方法的完整定义为：

```
void close() throws IOException
```

应用程序在调用这些方法时，如果不希望由系统进行异常处理，而希望由应用程序进行异常处理，则需要按照 8.4.2 节讨论的方法进行异常的抛出和异常的处理。

3．InputStream 类的常用子类

InputStream 是抽象类，它本身不能直接用来创建对象，但这个抽象类有很多子类，这些不同子类通过各自定义的抽象父类的抽象方法，实现了各种不同功能的输入字节流。可以通过实例化 InputStream 类的子类来创建所需的对象。比较常用的 InputStream 类的子类及继承关系如图 9.2 所示。

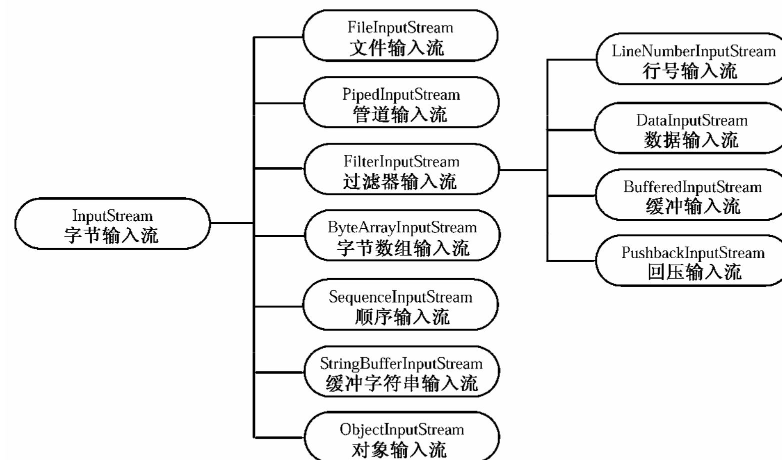


图 9.2　InputStream 类的子类及继承关系

下面介绍两个 InputStream 类的两个常用子类：BufferedInputStream 类和 FileInputStream 类。

（1）BufferedInputStream 类

当一个 BufferedInputStream 类对象被创建时，就产生了一个内部缓冲数组，因此，可以根据需要从连接的输入数据流中一次性读多个字节的数据到内部缓冲数组中，这样可以提高读输入数据流的效率。

BufferedInputStream 的常用构造方法有：

⌘ BufferedInputStream(InputStream in)
　用 InputStream 类的对象为参数构造缓冲字节输入流

⌘ BufferedInputStream(InputStream in, int size)
　用 InputStream 类的对象为参数构造缓冲数组大小为 size 的缓冲字节输入流

（2）FileInputStream 类

FileInputStream 类主要用于文件的输入，创建的对象可以顺序地从本地机上的文件中读数据。

FileInputStream 类常用的构造方法有：

⌘ FileInputStream(String name) throws FileNotFoundException
  用本地机上的文件 name 构造文件字节输入流

⌘ FileInputStream(File fileName) throws FileNotFoundException
  用本地机上的文件 fileName 构造文件字节输入流

BufferedInputStream 类和 FileInputStream 类都是 InputStream 类的子类，它们都重写了 InputStream 类中的 read()方法，它们也都继承了 InputStream 类中定义的方法。

【例 9.1】 读取键盘的输入数据和文件中的内容，并都显示在屏幕上。

要求：

① 让用户从键盘输入一行字符串，并在屏幕上输出显示该字符串。

② 从本地机的文件中读入字符串，然后在屏幕上输出显示文件字符串。

```
import java.io.*;                              //导入 java.io 中的所有类
public class TestInputStream
{
    public static void main(String args[])
    {
        int count;
        byte input[] = new byte[256];          //用于字符输入缓冲的数组
        String s;
        BufferedInputStream bufin =
                new BufferedInputStream(System.in);
                                               //用 System.in 为参数创建缓冲字节输入流对象 bufin
        try                                    //抛出异常
        {
            System.out.print("输入字节:");
            bufin.mark(256);
            count = bufin.read(input);
                                               //将键盘输入的字节放入字节输入缓冲数组 input 中
            System.out.println("您输入的字节个数: " + count);
            s = new String(input,0,count);
                                               //创建字符串对象，初始字符串长度为 count-1
            System.out.println("文件中的字符为: " );
            FileInputStream filein = new FileInputStream("a.txt");
                                           //用文件名"a.txt"为参数创建文件字节输入流对象 filein
            while (filein.read(input)!=-1)
                                   //从文件中逐个读入字节到输入字节缓冲数组 input 中直到文件结束
            {
                System.out.println(new String(input));
                                   //将 input 中的字节在屏幕上显示，系统将自动转换为字符形式显示
```

```
            }
            filein.close();                     //关闭文件字节输入流
            bufin.close();                      //关闭缓冲字节输入流
        }
        catch(IOException e)                    //处理异常
        {
            System.out.println("I/O 错误");
        }
    }
}
```

程序设计说明：

① 语句 BufferedInputStream bufin = new BufferedInputStream（System.in）表示用键盘输入（System.in）为参数创建缓冲字节输入流对象 bufin。要注意的是，从键盘输入的字符串结尾都会自动加上回车和换行（"\r\n"）两个字节。

② 语句 new FileInputStream（"a.txt"）表示用本地机当前目录上的文件"a.txt"创建文件字节输入流对象。

③ 语句 filein.read（input）表示从对象 filein 中逐个读入字节到输入字节缓冲数组 input 中。

④ 程序中对捕捉到的所有异常都只是输出了出错信息，并没有分类处理。

假设当前工作目录下文件"a.txt"中的内容为：

```
I am learning java
```

则程序执行的结果如下：

```
输入字节:How are you
您输入的字节个数: 13
文件中的字符为:
I am learning java
```

9.2.2 OutputStream（字节输出流）类

OutputStream 类称作字节输出流类，也是一个抽象类。OutputStream 类为其他的输出字节流子类提供了一些基本方法和标准接口。由于 OutputStream 类是一个抽象类，所以它本身不能直接用来创建对象。

1．OutputStream 类的常用方法

OutputStream 类提供的常用方法主要有：

- ⌘ void close()　　关闭输出流
- ⌘ void flush()　　清空输出流，使所有缓冲字节写完
- ⌘ void write(byte[] b)　　将数组 b 中的字节写到输出流
- ⌘ void write(byte[] b, int off, int len)
  将指定数组 b 中字节写到输出流，其起始位置为 off，长度为 len
- ⌘ abstract void write(int b)　　抽象方法，将字节 b 写到输出流

同样，OutputStream 类定义的许多方法都有 throws IOException 语句。

OutputStream 类的子类重写了不同功能的 write(int b)方法。

2．OutputStream 类的常用子类

OutputStream 是抽象类，它本身不能直接用来创建对象，但这个抽象类有很多子类，这些子类实现了各种具体功能的输出字节流，可以通过实例化它的子类来创建所需的对象。

比较常用的 OutputStream 类的子类及继承关系如图 9.3 所示。

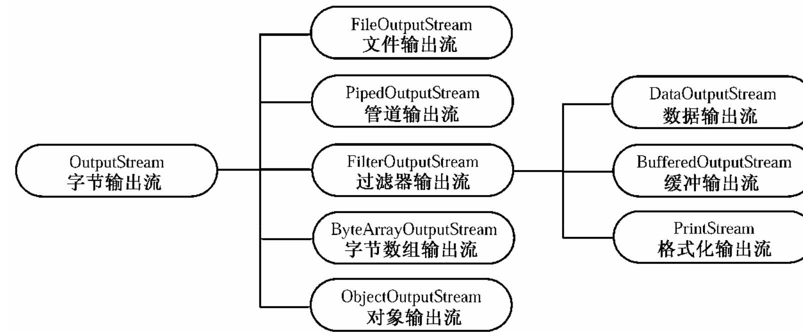


图 9.3　OutputStream 类的子类及继承关系

下面介绍两个常用的 OutputStream 类的子类：BufferedOutputStream 类和 FileOutputStream 类。

（1）BufferedOutputStream 类

当一个 BufferedOutputStream 类对象被创建时，就产生了一个内部缓冲数组，因此，可以从连接的输出数据流中一次性向内部缓冲数组中写多个字节的数据。这样就提高了写输出流的效率。

根据需要，内部缓冲数组中的字节可以输出也可以跳过。

BufferedOutputStream 类的常用构造方法有：

⌘ BufferedOutputStream(OutputStream out)
  以 OutputStream 类的对象 out 为参数构造缓冲字节输出流

⌘ BufferedOutputStream(OutputStream out, int size)
  以 OutputStream 类的对象 out 为参数构造缓冲数组大小为 size 的缓冲字节输出流

（2）FileOutputStream 类

FileOutputStream 类主要用于文件的输出，它的对象可以顺序地向本地机上的文件中写数据。

FileOutputStream 类的常用构造方法有：

⌘ FileOutputStream(String name) throws FileNotFoundException
  用本地机上的文件 name 为参数构造文件字节输出流

⌘ FileOutputStream(File fileName) throws FileNotFoundException
  用本地机上的文件 fileName 为参数构造文件字节输出流

⌘ FileOutputStream(String name, boolean app) throws FileNotFoundException
  用本地机上的文件 name 构造写文件方式为 app 的文件字节输出流。其中，app 为 true 表示添加到文件尾部，app 为 false 表示覆盖原来文件

BufferedOutputStream 类和 FileOutputStream 类都是 OutputStream 类的子类，它们都重写了 OutputStream 类中的 write(int b)方法，它们也都继承了 OutputStream 类中定义的方法。

【例 9.2】 将键盘输入的一行字符串在屏幕上显示，并将其添加到本地机文件 a.txt 的尾部。

程序设计如下：

```
import java.io.*;
public class TestOutputStream
{
    public static void main(String args[])
    {
        int count;
        byte input[] = new byte[256];          //用于字节输入缓冲的数组
        String s;
        BufferedInputStream bufin
                = new BufferedInputStream(System.in);
                                               //用键盘输入创建缓冲字节输入流对象 bufin
        BufferedOutputStream bufout =
        new BufferedOutputStream(System.out);
                                               //用屏幕输出创建缓冲字节输出流对象 bufout
        try
        {
            System.out.print(" 输入字节:");
            count = bufin.read(input);
                                               //将键盘输入的字节放入字节输入缓冲数组 input 中
            System.out.println("您输入的字节个数: " + count);
            System.out.print("您输入的字符串为: " );
            bufout.write(input,0,count);
                                               //将 input 中的所有字节写到字节输出缓冲数组中
            bufout.flush();                    //将字节输出缓冲数组内容输出
            FileOutputStream fileout
                = new FileOutputStream(“a.txt”,true);
                                           //创建文件字节输出流对象 fileout，参数为文件名“a.txt”
            fileout.write(input,0,count);
                                               //将 input 中的所有字节写到对象 fileout 中
            bufin.close();                     //关闭缓冲字节输入流
            bufout.close();                    //关闭缓冲字节输出流
            fileout.close();                   //关闭文件字节输出流
        }
        catch(IOException e)
        {
            System.out.println("I/O 错误");
```

```
            }
        }
    }
```

程序设计说明：

① 语句 BufferedOutputStream bufout = new BufferedOutputStream (System.out) 用屏幕输出（System.out）创建了缓冲字节输出流对象 bufout。

② 语句 FileOutputStream fileout = new FileOutputStream (“a.txt”,true) 表示用本地机当前目录上的 a.txt 文件创建了文件字节输出流对象 fileout。其中参数 true 表示将输出到文件中的内容添加到文件的尾部。

③ 缓冲字节输入流对象 bufin 调用的是父类 InputStream 的 close()方法；缓冲字节输出流对象 bufout 和文件字节输出流对象 fileout 调用的 close()方法是它们的共同父类 OutputStream 中提供的方法。

程序运行的结果如下：

```
输入字节: 我正在学习输入/输出
您输入的字节个数: 20
您输入的字符串为: 我正在学习输入/输出
```

若打开当前工作目录下的文件 a.txt，可以看到输入到文件的字节被添加到了文件 a.txt 的尾部。

9.2.3 Reader（字符输入流）类

Reader 类称作字符输入流类，是一个抽象类。Reader 类为通用的输入字符流提供了一些基本方法和标准接口。

Reader 类定义了两个抽象方法：

⌘ abstract void close()　　关闭输入字符流

⌘ abstract int raed(char[] cbuf, int off, int len)
　从输入字符流读起始位为 off、长度为 len 的若干个字符进数组 cbuf 中

Reader 类的子类重写了不同功能的这两个抽象方法。由于 Reader 类是一个抽象类，所以它本身不能直接用来创建对象。

Reader 类和 InputStream 类的很多方法很相似，它们的主要区别是：InputStream 类操作的是字节，Reader 类操作的是字符。

1．Reader 类的常用子类

常用的 Reader 类的子类及继承关系如图 9.4 所示。

Reader 类中提供的方法和 9.2.1 节介绍的 InputStream 类中提供的方法非常相似，这里不再介绍。

BufferedReader 类和 InputStreamReader 类是 Reader 类的子类中常用的两个子类。

（1）BufferedReader 类

BufferedReader 称作缓冲字符输入流，它在字符输入流的基础上，增加了字符缓冲的功能。当一个 BufferedReader 类对象创建时，就产生了一个内部缓冲数组，这样就可以根据需要从连接的输入字符流中一次性读多个字符。因此，BufferedReader 类可以提高读输入字符流的效率。

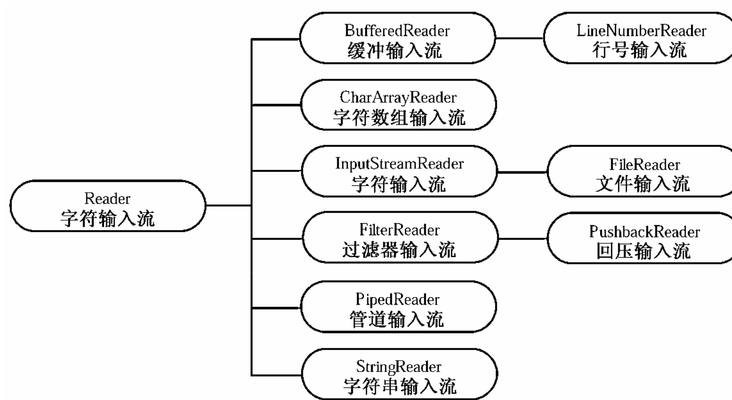


图 9.4 Reader 类的子类及继承关系

BufferedReader 类的构造方法有：

⌘ BufferedReader(Reader in)

构造带字符缓冲数组的字符输入流，in 是字符输入流对象。

⌘ BufferedReader(Reader in, int size)

构造指定字符缓冲数组大小的字符输入流，in 是字符输入流对象，size 是缓冲数组的大小。

BufferedReader 类的常用方法主要继承自 Reader 类。

（2）InputStreamReader 类

从 BufferedReader 类的构造方法可以看出，其参数只能是 Reader 类的对象，这样 BufferedReader 类的对象就只能从 Reader 对象读数据。而代表标准输入的 System.in 是一个 InputStream 类的对象，所以需要将 InputStream 类的对象转换成 Reader 类的对象，这就需要一个转换器。InputStreamReader 类是将字节输入流转换成字符输入流的转换器。

InputStreamReader 类的构造方法有：

⌘ InputStreamReader(InputStream in)

用字节输入流对象构造字符输入流，in 是字节输入流对象

⌘ InputStreamReader(InputStream in, String enc)

throws unsupportedEncodingException

用字节输入流对象构造字符输入流，in 是字节输入流对象。enc 是所支持的字符编码名。假如 enc 为不支持的字符编码名，则抛出异常

BufferedReader 类的常用方法也主要继承自 Reader 类。

2．应用举例

【例 9.3】 从键盘输入一行字符串存储在一个字符数组中。

要求：用 InputStreamReader 对象把键盘输入的字节输入流类型转换为字符输入流，并用缓冲字符输入流来读数据。

程序设计如下：

```
import java.io.*;
public class TestReader
{
```

```
    public static void main(String args[])
    {
        int count;
        char input[]=new char[256];         //字符输入缓冲数组。注意类型为 char
        String s;
        InputStreamReader stdin = new InputStreamReader(System.in);
                                            //用键盘输入流对象为参数创建字符输入流对象 stdin
        BufferedReader bufin = new BufferedReader(stdin);
                                            //用字符输入流对象 stdin 为参数创建缓冲字符输入流对象 bufin
        try
        {
            System.out.print("输入字符串: ");
            count = bufin.read(input);
                                            //把从键盘输入的字符放在字符输入缓冲数组 input 中
            System.out.println("您输入的字节个数: " + count);
            s = new String(input,0,count);
            System.out.print("您输入的字符串为: " + s);
            bufin.close();
        }
        catch(IOException E)
        {
            System.out.println("I/O 错误");
        }
    }
}
```

程序运行的结果如下：

```
输入字符串：我正在学习计算机课程
您输入的字节个数:22
您输入的字符串为：我正在学习计算机课程
```

9.2.4　Writer（字符输出流）类

1．Writer 类及其子类

Writer 类称作字符输出流类，也是一个抽象类。Writer 类为通用的输出字符流提供了一些基本方法和标准接口。

Writer 类定义了两个抽象方法：

⌘　abstract void close()　　首先调用 flush()方法，然后关闭输出字符流

⌘　abstract int write(char[] cbuf, int off, int len)

　将指定数组 cbuf 中字符写到输出流，其起始位置为 off，长度为 len

Writer 类的子类通过重写这两个抽象方法，实现了不同功能的写输出流。由于 Writer 类是一个抽象类，所以它本身不能直接用来创建对象。

Writer 类和 OutputStream 类的很多方法很相似，它们的主要区别是：OutputStream 类操作的是字节，Writer 类操作的是字符。

常用的 Writer 类的子类及继承关系如图 9.5 所示。

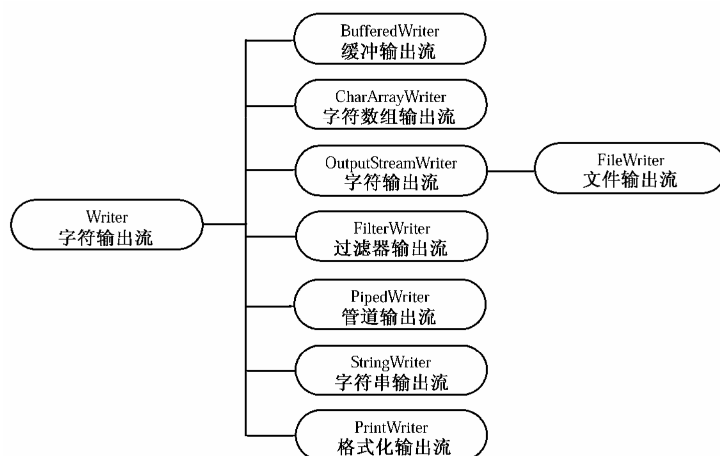


图 9.5 Writer 类的子类及继承关系

Writer 类中提供的方法和 9.2.2 节介绍的 OutputStream 类中提供的方法非常相似，这里不再介绍。

这里简单介绍一个 Writer 类的常用子类 PrintWriter。

2．PrintWriter（屏幕字符输出流）类

PrintWriter 类称作屏幕字符输出流类，它主要用来把字符输出流做屏幕输出。

常用的构造方法有：

⌘ PrintWriter(OutputStream out) 用 OutputStream 对象 out 构造

⌘ PrintWriter(OutputStream out, boolean autoFlush)
用 OutputStream 对象 out 构造，autoFlush 为 true 表示自动清空输入字符流

常用的方法有：

⌘ void print(String s) 打印字符串 s

⌘ void println(String s) 打印字符串 s 并换行

⌘ void print(Object obj) 打印对象 obj

⌘ void println(Object obj) 打印对象 obj 并换行

另外，PrintWriter 类还提供了打印系统定义的所有基本数据类型（如 int，double 等）的 print()方法和 println()方法，这里不再一一列出。

在第 12 章网络通信的 Client/Server 结构的简易聊天室中，我们将用到 PrintWriter 类。简易聊天室中，两人要把各自从自己键盘上输入的数据传送给对方，并显示在对方的屏幕上。

9.2.5 FileReader 和 FileWriter（字符文件输入/输出流）类

在 Java 中，用于文件输入/输出的类可以分为两大类：①用于字节文件输入/输出的 FileInputStream 类和 FileOutputStream 类，这已分别在 9.2.1 节和 9.2.2 节作了介绍；②用于字符文件的输入/输出的 FileReader 类和 FileWriter 类。

FileReader 类是 Reader 类的子类的子类，FileWriter 类是 Writer 类的子类的子类。与 9.2.1 节和 9.2.2 节的编写结构不同，9.2.3 节和 9.2.4 节中没有介绍这两个类，本节补充介绍这两个类。

1．FileReader 类

FileReader 类称作字符文件输入流类。FileReader 类常用的构造方法是：

⌘ FileReader(String fileName)
构造字符文件输入类对象。参数 fileName 是需要读的文件名

2．FileWriter 类

FileWriter 类称作字符文件输出流类。FileWriter 类常用的构造方法是：

⌘ FileWriter(String fileName)
构造字符文件输出类对象。参数 fileName 是需要写的文件名

【例 9.4】 FileReader 类和 FileWriter 类使用示例。

要求：首先将一个字符串写到本地机的一个文件中，然后再从该文件中读出数据并显示在屏幕上。

设计方法：首先用 FileWriter 对象将字符串写到本地机的一个文件中，然后用 FileReader 对象从该文件中读入数据并显示到屏幕上。

程序设计如下：

```
import java.io.*;
public class TestFileRandW
{
    public static void main(String args[]) throws IOException
    {
        String str = "I am learning Java";              //要输出的字符串
        String str2;
        BufferedWriter bw =
            new BufferedWriter(new FileWriter("a.out"));
                                    //用文件名为 a.out 的参数创建字符文件输出流对象，
                                    //并用所创建的对象为参数创建缓冲字符输出流对象 bw
        bw.write(str,0,str.length());//把字符串 str 中的全部字符写到 bw 中
        bw.flush();                 //输出 bw 中的全部字符
        BufferedReader br =
            new BufferedReader(new FileReader("a.out"));
                                    //用文件名为 a.out 的参数创建字符文件输入流对象，
                                    //并用所创建的对象为参数创建缓冲字符输入流对象 br
        str2 = br.readLine();       //从 br 中读一行字符到 str2
        System.out.println(str2);   //把 str2 中的字符串显示在屏幕上
        bw. close();                //关闭 bw
        br. close();                //关闭 br
    }
}
```

程序设计说明：

① 语句 BufferedWriter bw = new BufferedWriter(new FileWriter("a.out")) 创建了两个对象：首先，用文件名为 a.out 的参数创建了一个字符文件输出流对象；然后，又用该对象作为参数创建了一个缓冲字符输出流对象 bw。用这种方式创建对象是输入/输出流中常见的。

② 可以调用缓冲字符输出流对象 bw 的方法向文件 a.out 中写字符串。语句 bw.write(str,0,str.length()) 将字符串 str 中的全部字符写到对象 bw 中，也即写到了文件 a.out 中。

③ 语句 BufferedReader br = new BufferedReader(new FileReader("a.out"))创建了两个对象：首先，用文件名为 a.out 的参数创建了一个字符文件输入流对象；然后，又用该对象作为参数创建了一个缓冲字符输入流对象 br。

④ 可以调用缓冲字符输入流对象 br 的方法从文件 a.out 中读数据。语句 str2=br.readLine() 从 br 中读一行字符到字符串变量 str2 中。

⑤ bw.close()调用的是 Writer 类中的 close()方法，br. close()调用的是 Reader 类中的 close()方法。

程序的运行结果如下：

```
I am learning Java
```

打开程序所在目录下的文件 a.out，可以看到，文件 a.out 中的内容和屏幕显示的内容相同。

9.3 对象流

在字节输入流 InputStream 类和在字节输出流 OutputStream 类的子类中，除了 9.2.1 节和 9.2.2 节介绍的 FileInputStream 类和 FileOutputStream 类外，还有对象流、数据流和管道流，这些可为一些特殊的数据交互提供方便。这里只讨论对象流。

对象都有一定的生命周期。一个对象包含了它生命周期当前的状态。有时候，需要将一个对象生命周期的某一阶段的状态保存下来；当需要的时候，再将保存的对象状态恢复。对象流可以实现这样的功能。对象流有对象输入流 ObjectInputStream 类和对象输出流 ObjectOutputStream 类。

对象流中的对象通常不止一个，为了保证保存和恢复的对象不会出错（对错位置），必须保证这些对象都是序列化的对象。序列化的对象是指能够按顺序操作对象流中的对象。一个类如果实现了 Serializable 接口，它的对象就是序列化的对象。

1．Serializable 接口

Serializable 接口的定义如下：

```
public interface Serializable(){}
```

这个接口是空的，其中没有定义任何方法，因此实现这个接口非常简单，仅需在类定义时，包含 implements Serializable 就可以了。

2．对象输入流 ObjectInputStream 类

ObjectInputStream 类是 InputStream 类的子类，它主要用于对象的恢复。

ObjectInputStream 类的构造方法有：
⌘ ObjectInputStream()　　空的构造方法
⌘ ObjectInputStream(InputStream in)　　用 InputStream 对象 in 构造对象
ObjectInputStream 类的常用方法：
⌘ final Object readObject()　　恢复对象

3．对象输出流 ObjectOutputStream 类

ObjectOutputStream 类是 OutputStream 类的子类，它主要用于对象的保存。
ObjectOutputStream 类的构造方法有：
⌘ ObjectOutputStream()　　空的构造方法
⌘ ObjectOutputStream(OutputStream out)　用 OutputStream 对象 out 构造对象
ObjectOutputStream 类的常用方法有：
⌘ final Object writeObject(Object obj)　　保存对象 obj

进行对象流的保存（或称写）时需要注意，只能保存对象的非静态成员变量，不能保存任何静态的成员变量，而且保存的只是变量的当前值，对于变量的任何修饰符都不能保存。

【例 9.5】　将一个对象保存在一个文件中， 然后恢复该对象。

程序设计如下：

```
import java.io.*;
public class Student implements Serializable
                                                //定义实现接口 Serializable 的学生类
{
    int idnum;                                  //成员变量
    String name;                                //成员变量
    int age;                                    //成员变量
    String department;                          //成员变量

    public Student(int i dnum,String name,int age,String dep)
                                                //构造方法
    {
        this.idnum = idnum;
        this.name = name;
        this.age = age;
        this.department = dep;
    }

    public void show()                          //显示学生信息
    {
        System.out.println(""+idnum+" "+name+" "+age+" "+department);
    }

    public static void main(String [] args)          //main()方法
```

```
        {
            Student stu = new Student(2003001,"Smith",19, "Computer");
                                                        //创建对象 stu
            try
            {
                System.out.print("保存的学生对象为:");
                stu.show();                             //显示学生信息
                FileOutputStream fo = new FileOutputStream("a.txt");
                                                        //用参数"a.txt"创建文件输出流对象 fo
                ObjectOutputStream so = new ObjectOutputStream(fo);
                                                        //用参数 fo 创建对象输出流对象 so
                so.writeObject(stu);                    //保存对象 stu 到文件 a.txt 中
                so.close();
        }
        catch(IOException e)
        {
                System.out.println(e);
        }
        try
        {
                FileInputStream fi = new FileInputStream("a.txt");
                                                        //用参数"a.txt"创建文件输入流对象 fi
                ObjectInputStream si = new ObjectInputStream(fi);
                                                        //用参数 fi 创建对象输入流对象 si
                Student newstu = (Student)si.readObject();
                                    //定义对象 newstu，并让该对象指向从文件"a.txt"中恢复的对象
                System.out.print("恢复的学生对象为:");
                newstu.show();                          显示学生信息
                si.close();
            }
            catch(Exception e)
            {
                System.out.println(e);
            }
        }
    }
```

程序设计说明：

① 程序首先定义了实现接口 Serializable 的学生类 Student，然后创建了 Student 对象 stu 并赋值，最后分别创建了对象输入流对象和对象输出流对象，分别进行了对象的保存和恢复。

② 语句：

```
FileOutputStream fo = new FileOutputStream("a.txt");
```

```
ObjectOutputStream so = new ObjectOutputStream(fo);
```

第一条语句用参数“a.txt”创建了文件输出流对象 fo，第二条语句用参数 fo 创建了对象输出流对象 so，这样，语句：

```
so.writeObject(stu);
```

就可以实现把对象 stu 保存到文件 a.txt 中。

③ 语句：

```
FileInputStream fi = new FileInputStream("a.txt");
ObjectInputStream si = new ObjectInputStream(fi);
```

第一条语句用参数“a.txt”创建了文件输入流对象 fi，第二条语句用参数 fi 创建了对象输入流对象 si，这样，语句：

```
Student newstu = (Student)si.readObject();
```

在定义了对象 newstu 后，就让该对象名指向从文件“a.txt”中恢复的对象

程序运行结果如下：

```
保存的学生对象为: 2003001 Smith 19 Computer
恢复的学生对象为: 2003001 Smith 19 Computer
```

9.4 文件的操作

在程序的运行过程中，经常需要从文件中读取数据或将运行的结果存入文件中，在前面的例子中，这类操作我们都只给出了文件名，这样就只能在当前工作目录下操作文件。实际上，复杂的文件操作还需要知道文件的路径、文件的长度等许多信息，本节讨论的 File 类可以实现文件的这些复杂操作。

9.4.1 File（文件）类及其应用

在 Java 中，File 类是专门描述文件的各种属性（如文件名、大小、可读性等），并提供方法操纵文件的路径等的类。

1．File 类的构造方法

⌘ File(String name) 用文件名 name 创建文件

⌘ File(File path, String name) 用路径名 path 和文件名 name 创建文件

⌘ File(String path, String name) 用路径名 path 和文件名 name 创建文件

注意：第二个构造方法的 path 指的是文件对象 path 的路径，而第三个构造方法的 path 指的是绝对路径。

例如，可以用下面的语句创建文件对象：

```
File f1 = new File (“file1.java”);
                                    //用文件名“file1.java”创建文件对象 f1
File f2 = new File (“c:\dir2”, ”file2.java”);
                                    //用路径“c:\dir2”和文件名”file2.java”创建文件对象 f2
```

```
File f3 = new File (f2, "java3.java");
                                        //用文件对象 f2 和文件名"java3.java"创建文件对象 f3
```

上述语句后，文件"file1.java"的路径就是当前工作路径，文件"file2.java"的路径就是"c:\dir2"，文件"java3.java"的路径也是"c:\dir2"。

2．File 类的常用方法

⌘ boolean canRead()　　是否可读
⌘ boolean canWrite()　　是否可写
⌘ boolean isFile()　　是否是文件
⌘ boolean isDirectory()　　是否是目录
⌘ boolean exists()　　是否存在
⌘ long length()　　返回文件的长度
⌘ long lastModified()　　返回文件的最后修改时间
⌘ String getName()　　返回文件名
⌘ String getPath()　　返回文件的路径
⌘ String getParent()　　返回文件的上一级目录
⌘ boolean mkdir()　　创建文件的目录，返回 true 表示成功，false 失败
⌘ boolean delete()　　删除文件，返回 true 表示成功，false 失败
⌘ boolean renameTo(File name)　　重命名文件 name，返回 true 表示成功，false 失败
⌘ String getAbsolutePath()　　返回文件的绝对路径
⌘ boolean isHidden()　　是否是隐藏文件

【例 9.6】　用命令行参数输入一个文件名，程序检查该文件是否存在。如果该文件存在，则在屏幕上显示该文件的属性。

说明：命令行参数就是和运行该程序的命令一起输入的参数。命令行参数将保存在 main() 方法的 args[]参数中。args 参数是一个数组，表示允许多个命令行参数。

程序设计如下：

```
import java.io.*;
import java.util.*;
public class TestFile
{
    public static void main(String args[])
    {
        String input = args[0];
        File fp = new File(input);
        System.out.println("路径: " +fp.getParent());
        System.out.println("文件名: " + fp.getName());
        System.out.println("绝对路径: " + fp.getAbsolutePath());
        System.out.println("文件大小: " + fp.length());
        System.out.println("是否为文件: " + fp.isFile());
        System.out.println("是否为目录: " + fp.isDirectory());
```

```
            System.out.println("是否为隐藏: " + fp.isHidden());
            System.out.println("是否可读: " + fp.canRead());
            System.out.println("是否可写: " + fp.canWrite());
            System.out.println("最后修改时间: "
                                    + new Date(fp.lastModified()));
        }
    }
```

编译完成后，用如下命令运行该程序（说明：文件 a.txt 是当前工作目录下存在的文件）：

```
java TestFile a.txt
```

程序运行结果如下：

```
路径: null
文件名: a.txt
绝对路径: E:\Java\ch9\a.txt
文件大小: 640
是否为文件: true
是否为目录: false
是否为隐藏: false
是否可读取: true
是否可写入: true
最后修改时间: Sun Aug 22 17:31:26 CST 2004
```

3．字符文件的应用

若用 FileReader 类和 FileWriter 类（字符文件输入流类和字符文件输出流类）来创建 File 对象，则读写的就是字符文件。

【例 9.7】 将指定的"in.txt"字符文件中的内容复制到指定的"out.txt"文件中。

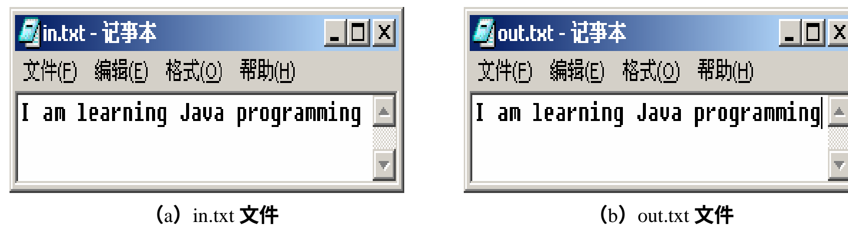
程序设计如下：

```
import java.io.*;
public class Copy
{
    public static void main(String[] args)
    {
        File inputFile = new File("in.txt");
                                        //用"in.txt"创建文件类对象 inputFile
        File outputFile = new File("out.txt");
                                        //用"out.txt"创建文件类对象 outputFile
        FileReader in = new FileReader(inputFile);
                                        //用 inputFile 作参数创建字符文件输入流对象 in
        FileWriter out = new FileWriter(outputFile);
                                        //用 outputFile 作参数创建字符文件输出流对象 out
        int c;
```

```
        while ((c = in.read()) != –1) out.write(c);
                                        //循环复制文件"in.txt"中的所有内容到文件"out.txt"中
        in.close();                     //关闭文件
        out.close();                    //关闭文件
    }
}
```

程序设计说明：这里使用字符输入/输出流类（FileReader 类和 FileWriter 类）来创建对象，所以这里读写的是字符文件。

假设当前工作目录下存在文件 in.txt，且文件 in.txt 中的内容如图 9.6（a）所示，则程序运行后，文件 out.txt 被创建，且文件中 out.txt 的内容如图 9.6（b）所示。可见，两个文件的内容是完全一样的。



（a）in.txt 文件　　（b）out.txt 文件

图 9.6　字符文件的拷贝

4．二进制文件的应用

用 FileInputStream 类和 FileOutputStream 类（字节文件输入/输出流类）来创建对象，则读写的就是字节文件。图像等类型的文件不能用字符方式读写，只能用字节方式读写。由于一个字节是 8 个二进制位（bit）组成的，所以，字节文件也称作二进制文件。

【例 9.8】　将指定的“source.GIF”文件中的内容复制到指定的“target.GIF”文件中。

程序设计如下：

```
import java.io.*;
public class TestCopyBytes {
public static void main(String[] args)
{
        File inputFile = new File("source.GIF");          //创建文件对象
        File outputFile = new File("target.GIF");         //创建文件对象
        FileInputStream in = new FileInputStream(inputFile);
                                                          //创建字符文件输入流对象
        FileOutputStream out = new FileOutputStream(outputFile);
                                                          //创建字符文件输出流对象
        int c;
        while ((c = in.read()) != –1)    out.write(c);
        in.close();
        out.close();
    }
}
```

程序设计说明：此程序和上一个程序基本相同，差别只是这里是用 FileInputStream 类和 FileOutputStream 类创建对象，所以这里读写的是二进制文件。

本程序运行时，当前工作路径下一定要存在文件“source.GIF”，否则会出现文件不存在异常。

9.4.2 RandomAccessFile（随机存取文件）类

前面讲到的文件读写都是按照顺序进行的，本节介绍 RandomAccessFile 类，RandomAccessFile 类称作随机存取文件类，它提供了随机访问文件的方法。和前面讨论的输入/输出流类相比，RandomAccessFile 类有两点不同：

① RandomAccessFile 类是直接继承自对象类 Object，同时实现了 DataInput 接口和 DataOutput 接口。

② 由于 RandomAccessFile 类实现了 DataInput 接口和 DataOutput 接口，所以，RandomAccessFile 类既可以作为输入流，又可以作为输出流。

RandomAccessFile 类之所以允许随机访问文件，是由于它定义了一个文件当前位置指针，文件的存取都是从文件当前位置指针指示的位置开始的。通过移动这个指针，就可以从文件的任何位置开始进行读/写操作。

1．构造方法

⌘ RandomAccessFile (File file, String mode) throws FileNotFoundException

⌘ RandomAccessFile (String name, String mode) throws FileNotFoundException

其中，file 是一个文件对象，mode 是访问方式，r 表示读，w 表示写，rw 表示既可以读又可以写。创建文件时若出现问题，将抛出 FileNotFoundException 异常。

例如，下面语句创建了随机存取文件 ra1，文件名为“a.txt”，属性为只读文件。

```
RandomAccessFile ra1 = new RandomAccessFile("a.txt", "r");
```

又例如，下面语句创建了随机存取文件 ra2，文件名为“b.txt”，属性为可读可写文件。

```
RandomAccessFile ra2 =new RandomAccessFile("b.txt", "rw");
```

2．常用方法

⌘ long getFilePointer()　　返回文件指针

⌘ long length()　　返回文件长度

⌘ void seek(long position)　　文件指针到达位置 position

⌘ int read()　　从文件读入一个字节

⌘ int read(byte[] b)　　从文件读入 b.length 个字节存入数组 b 中

⌘ int read(byte[], int off, int len)
　从文件当前位置指针为 off 开始读入 len 个字节存入数组 b

⌘ String readLine()　　从文件读入一行字节

⌘ void write(int b)　　向文件中写入字节 b

⌘ void write(byte[] b)　　把数组 b 中的字节全部写入文件中

⌘ void write(byte[] b, int off, int len)

把数组 b 中 len 个字节写入文件中文件当前位置指针为 off 开始的位置

⌘ void writeChars(String s)　　　　向文件中写入字符串 s

要说明的是，随机存取文件是按照文件当前位置指针当前的值进行读/写操作的，上述方法中，如果参数中有这个指针参数，表示首先把这个参数的值赋给文件当前位置指针，然后进行读/写操作 如果参数中没有这个指针参数，则表示按照文件当前位置指针进行读/写操作。

【例 9.9】　在文件的指定位置进行访问。

要求：从键盘输入一个整数 n，利用 RanomAccessFile 对象访问本地机上的文件，从文件开头跳过 n 个字节，将后面的内容在屏幕上显示出来。

程序设计如下：

```
import java.io.*;
public class RandomDemo
{
    public static void main(String args[])
    {
        int   skipNo = Integer.parseInt(args[0]) ;
        try
        {
            RandomAccessFile rf = new RandomAccessFile("b.txt","rw");
                                                          //创建随机访问文件类对象
            System.out.println("原始字符串："+ rf.readLine());
            rf.seek(skipNo);                              //跳过若干字符
            System.out.println(" 跳过字节数：  "+skipNo +": "+ rf.readLine());
            rf.close();
        }
        catch(Exception e)
        {
            System.out.println("发生 I/O 错误!!!");
        }
    }
}
```

运行该程序的命令如下：

```
java   RandomDemo 5
```

程序运行结果如下：

```
原始字符串：1234567890
跳过字节数：  5: 67890
```

程序运行结果说明：该程序运行前，要求在当前工作目录下存在文件“b.txt”，假设文件“b.txt”中保存了字节数据“1234567890”；命令行参数 5 被程序作为了文件当前位置指针定位数值（从 0 开始）。所以，程序运行时只输出了 67890。

习题 9

一、基本概念题

9.1　什么是流? 什么是输入流？什么是输出流？

9.2　按输入/输出的数据类型来分，java.io 包中有哪两种基本的输入/输出流？它们对应的类是什么？

9.3　Java 中的标准输入/输出是如何实现的？

9.4　字符输入/输出流和字节输入/输出流的方法有哪些相同点？有哪些不同点？

9.5　File（文件）类的主要用途是什么？如何在程序中创建一个文件？

9.6　RandomAccessFile（随机存取文件）类的主要用途是什么？它和 File 类有什么区别？

9.7　什么叫对象流？对象流的主要用途是什么？

二、程序设计题

9.8　编写一个程序，让用户从键盘键入一个字符串，程序将字符串重新排序，并将得到的新的字符串在屏幕上输出。

9.9　编写一个程序，让用户从键盘上键入一个文件名，程序将文件中的内容在屏幕上输出。

9.10　下面程序的运行结果是什么？为什么？

```
import java.io.*;
public class B
{
    public static void main(String args[])
    {
        try
        {
            File f = new File("b.txt");
            FileOutputStream out = new FileOutputStream(f);
        }
        catch(IOException e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

9.11　编写一个程序，让用户从键盘上键入一个文件名（要求该文件在当前工作目录下存在），程序将该文件的内容复制到当前工作目录下的另一个文件中。

9.12　编写一个程序，将一个图像文件复制到指定的文件中（要求被复制的文件在当前工作目录下存在）。

9.13　编写一个程序，将一个文件中的内容添加到另外一个文件的尾部。

9.14　编写一个程序，列出某个目录及其子目录下的所有文件的名称和它们的属性。

9.15　编写一个用命令行参数表示文件名的程序，当确信该文件不存在时，用该文件名在当前工作目录下新建一个文件。

9.16　修改例 9.5 程序，将一个对象数组保存在一个文件中，然后再从该文件中恢复该对象数组。

第 10 章　多　线　程

教学要点

本章内容主要包括线程的基本概念，Thread 类和 Runnable 接口的功能和使用方法，线程的状态和状态控制，线程的分组，共享资源的两种模式，互斥线程的设计方法，同步线程的设计方法。

要求掌握线程的基本概念、线程的状态和状态控制、线程的分组、同步线程的设计方法，熟练掌握 Thread 类和 Runnable 接口的功能和使用方法、共享资源的两种模式、互斥线程的设计方法。

多线程是面向对象程序设计语言的一个特点。多线程不仅可以使一个程序同时完成多项任务，而且为此消耗的系统资源也比进程方法少许多。本章主要介绍 Java 语言中线程的概念和基本设计方法，包括线程和多线程的基本概念、Thread 类和 Runnable 接口、线程的状态和控制、互斥线程的设计方法和同步线程的设计方法等。本章最后还给出一个前面几章概念综合应用的综合设计问题。

10.1　线程的基本概念

10.1.1　进程和线程

1．进程

进程是一个执行中的程序。系统可以同时创建多个进程，也就是说，系统可以同时完成多个任务。对每一个进程来说，它拥有自己独立的内存空间、数据等运行中需要的系统资源，它的状态和拥有的资源是完全独立的。例如，在一台计算机上用浏览器打开一个网页，此时如果再使用 Word 软件进行文字编辑工作，就是系统运行了浏览器软件和 Word 软件两个进程。这两个进程各自拥有自己的资源，独立存在，彼此之间互不影响。系统还可以为一个程序创建多个进程。例如，同时打开两个记事本文件，就是创建了两个记事本程序进程。每一个进程都有自己独立的一块内存空间和其他系统资源，即使是同类进程（如两个记事本程序进程）之间也不会共享系统资源。

2．线程

进程也存在一些问题。例如，如果一个程序要同时完成多项任务，如果把这些任务都设计成一个个进程，当系统频繁地进行进程切换（术语称为进程调度）时，就要占用大量的系统资源（主要是 CPU 时间）。线程是解决这类问题的一个途径。

对于一个进程来说，并不是所有的任务切换都要“大动干戈”地进行进程调度。有些子

任务的任务切换问题，可能只是程序中很小一部分代码被分开执行，它们使用的系统资源基本没有发生变化，这时可以不进行系统使用资源的切换。这就提出了线程技术。

线程是一个程序中实现单一功能的一个指令序列，一个程序中可以包含若干个线程。简单地说，线程是轻量级的进程。线程是一个程序的一部分。线程不能单独运行，它必须在一个程序之内运行。也就是说，一个进程之内的所有线程使用的系统资源是一样的。由于一个进程内的线程切换（术语称为线程调度）不需要进行系统占用资源的切换，所以可以大大提高系统的使用效率。

因此可以总结说，线程的意义在于：对于有些设计问题来说，可以将一个进程按不同功能划分为多个线程。这样，既能同时为用户提供多项服务，同时线程间切换花费的 CPU 时间也很少。例如，一个浏览器在下载一些图片的同时，也能供用户浏览已经下载下来的页面，这就是利用线程技术提供高效服务的一个例子。

多线程是指一个进程中同时运行的多个完成不同子任务的线程。多线程不仅可以使一个程序同时完成多项任务，而且为此消耗的系统资源也比进程方法少许多。

10.1.2 线程的生命周期和状态

线程的生命周期是指线程从创建、运行到死亡的过程。在一个线程的生命周期内，可以有五种状态，即创建、可运行、运行中、不可运行（或称阻塞）和死亡状态。通过对线程的控制和调度，一个线程可以在这五种状态之间转换。线程生命周期的状态图如图 10.1 所示。

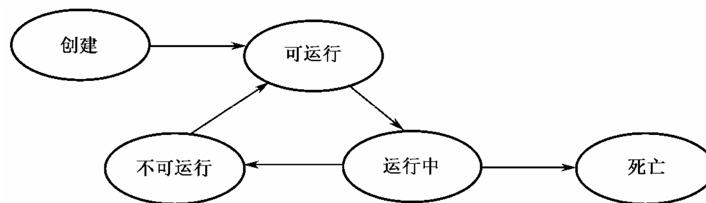


图 10.1 线程生命周期的状态图

Java 语言支持多线程程序的运行，要编写多线程的程序，需要 Java 提供的支持线程的类和接口。

10.2 Thread（线程）类和 Runnable（可运行）接口

Thread（线程）类和 Runnable（可运行）接口提供了支持线程的功能。由于 Thread 类和 Runnable 接口是在 java.lang（语言）包中，所以使用 Thread 类和 Runnable 接口时不需要使用 import 导入语句。

10.2.1 Thread（线程）类

Thread 类是 Java 中支持多线程机制的一个重要类，它提供了很多控制和调度线程的方法。这些方法包括：启动、睡眠、运行等。

Thread 类的常用成员变量、构造方法和方法如下：

（1）成员变量

⌘ MIN_PRIORITY　　线程可以拥有的最小优先级

⌘ MAX_PRIORITY　线程可以拥有的最大优先级
⌘ NORM_PRIORITY　线程默认的优先级

（2）构造方法

⌘ Thread()
⌘ Thread(Runnable target)
⌘ Thread(Runnable target，String name)
⌘ Thread(String name)
⌘ Thread(ThreadGroup group，Runnable target)
⌘ Thread(ThreadGroup group，String name)

其中，Runnable 是系统的 Runnable 接口，这里表示实现了 Runnable 接口的类；target 是执行线程体的目标对象；name 为线程名；ThreadGroup 是系统的线程组类；group 是线程所属的线程组的名称。

（3）常用方法

⌘ static int activeCount()　返回线程组当前线程的数目
⌘ static Thread currentThread()　返回当前运行线程的引用
⌘ void destroy()　撤销线程
⌘ String getName()　返回线程名称
⌘ int getPriority()　返回线程优先级
⌘ ThreadGroup getThreadGroup()　返回当前线程所属的线程组
⌘ void interrupted()　中断当前线程
⌘ boolean isAlive()　是否激活态线程，true 是，false 否
⌘ void join()　等待当前线程死亡
⌘ void run()　运行线程
⌘ void setName(String name)　设置线程名称为 name
⌘ void setPriority(int newPriority)　设置线程优先级为 newPriority
⌘ static void sleep(long millis)　指定当前运行线程休眠 millis 毫秒
⌘ void start()　启动线程
⌘ Static void yield()　让当前运行线程临时退出

上述方法中，run()方法是运行一个线程。程序设计时，把想要运行的程序代码（即线程）放入这个方法中。当创建了一个线程对象后，可以用 start()方法来启动这个线程，start()方法通过自动调用 run()方法来执行当前线程包含的代码。下面的例子说明了如何运用 Thread 类及该类提供的方法来控制线程的执行。

如果要设计一个包含线程的程序，一种常用的方法是继承 Thread 类，并要在这个类中实现 run()方法。run()方法中包含了要运行的线程代码。

【例 10.1】　简单的多线程示例。

要求：设计两个轮流执行的线程，一个取名为“Java”，另一个取名为“C++”。每个线程循环输出 5 次线程名，每输出一次线程名切换一次。

程序设计如下：

```
public class TestSimpleThread extends Thread          //继承 Thread 类的类
{
```

```
    public TestSimpleThread(String str)                     //构造方法
    {
        super(str);                                         //调用父类的构造方法给线程名赋值
    }

    public void run()                                       //线程执行的代码放在此方法中
    {
        for (int i = 0; i < 5; i++)
        {
            System.out.println(i + " " + getName());        //输出当前线程名
            try
            {
                sleep((long)(Math.random() * 1000));        //当前线程休眠
            } catch (InterruptedException e) { }            //捕捉处理中断
        }
        System.out.println(getName()+" Finish! ");
    }

    public static void main(String[] args)
    {
        new TestSimpleThread("Java").start();               //创建并启动一个线程
        new TestSimpleThread("C++").start();                //创建并启动一个线程
    }
}
```

设计说明：

① 由于 TestSimpleThread 类中没有定义成员变量，所以不需设计特殊的构造方法，只需调用 Thread 类的构造方法。

② 线程要执行的语句都放在 run()方法中。这里将做 5 次循环，每一次在屏幕上输出序号和线程名。每次输出后，sleep()方法让线程休眠一段时间，然后继续下一个循环。循环结束时，打印线程名和 finish!。

③ sleep()方法会发出一个系统定义的中断异常，把该语句放在 try 模块中，随后的 catch 模块语句用来处理该中断异常，实现当前线程休眠一段时间。线程休眠也就是线程进入了不可运行状态（或称阻塞状态），但当休眠时间一到，该线程又进入了可运行状态，排队等待线程调度程序调度进入运行状态。

④ main()方法中分别创建并用 start()方法启动了两个名字分别为 Java 和 C++的线程。start()方法将自动调用 run()方法。实际上，这个程序中一共有三个线程，除了创建的两个线程外，main()方法本身也是一个线程，称为主线程。

程序运行的结果如下：

```
0 Java
0 C++
1 C++
```

```
1 Java
2 C++
2 Java
3 C++
3 Java
4 Java
Java Finish!
4 C++
C++ Finish!
```

程序运行结果分析：

① 两个线程是交替运行的，感觉就像是两个线程在同时运行。实际上，一台计算机通常只有一个 CPU，在某个时刻只能有一个线程在使用 CPU。为了给多个线程中的每一个线程执行的时间和机会，通常，我们使用让线程睡眠（调用 sleep()方法）的办法来暂停当前线程的执行，这样，当前线程就由运行状态转入阻塞状态，另一个线程就可以由阻塞状态转入运行状态。

② 线程的执行顺序依赖系统的线程调度程序，所以两个线程的执行顺序并不一定是交替执行的。

10.2.2 Runnable（可运行）接口

Java 中的 Runnable 接口为设计多线程程序提供了另外一种方法。任何实现 Runnable 接口的类都支持多线程。实际上 Thread 类就实现了 Runnable 接口。

在 Runnable 接口中只有一个方法：run()。只要在程序中实现该方法，并把想要运行的线程代码放入这个方法中就可以了。

【例 10.2】 通过实现 Runnable 接口进行多线程程序设计示例。

要求：和例 10.1 的要求基本相同，只是要求用实现 Runnable 接口进行多线程程序设计。

程序设计如下：

```
public class TestSimpleRunnable implements Runnable
                                                //实现接口定义的类 TestSimpleRunnable
{
    private String str;                         //成员变量，表示线程名
    private Thread myThread;                    //成员变量，表示线程

    public TestSimpleRunnable(String str)       //构造方法
    {
        this.str = str;                         //给线程名赋值
    }

    public void myStart()
    {
        myThread = new Thread(this, str);       //创建线程对象
        myThread.start();                       //调用 start()方法
```

```
    }

    public void run()                                   //实现接口 Runnable 的 run()方法
    {
        for (int i = 0; i < 5; i++)
        {
            System.out.println(i + " " + myThread.getName());
                                                        //输出当前线程名
            try
            {
                myThread.sleep((long)(Math.random() * 1000));
                                                        //当前线程休眠
            } catch (InterruptedException e) { }        //捕捉处理中断
        }
        System.out.println(myThread.getName()+" Finish! ");
    }

    public static void main (String[] args)
    {
        new TestSimpleRunnable("Java").myStart();       //创建并启动一个线程
        new TestSimpleRunnable("C++").myStart();        //创建并启动一个线程
    }
}
```

程序运行结果和例10.1的运行结果类同。

程序设计说明：

① 实现接口定义的类（如TestSimpleRunnable）要使用Thread类提供的方法，设计方法是定义Thread类的对象。这里首先定义了成员变量myThread，myThread是Thread类对象，然后在myStart()方法中创建线程并把创建的线程赋值给myThread。对应语句为：

```
myThread = new Thread(this, str);
```

其中，this 是执行线程体的目标对象，str 是线程名。

② 实现的run()方法中，通过myThread对象来调用getName()方法和sleep()方法。如：

```
myThread.getName()
```

对比例10.1和例10.2可以发现，要设计线程程序，用继承Thread类方法比用实现Runnable接口方法更简单，因为用继承Thread类方法设计时，因为已经继承了Thread类，所以定义的对象可以直接调用Thread类的方法；而用实现Runnable接口方法设计时，要使用Thread类提供的方法必须定义Thread类的对象，即通过Thread类的对象来调用Thread类的方法。

那么读者一定会问，既然有了Thread类，Runnable接口有什么作用呢？Runnable接口主要用于多继承。例如，要设计一个具有线程功能的Java Applet程序，由于Java Applet程序必须继承Applet类，因此就不能再继承Thread类，这时就只能通过继承Applet类并实现Runnable接口来完成设计。

10.3 线程的状态和状态控制

上一节介绍了Thread类和Runnable接口，以及如何创建和使用线程对象，本节将进一步介绍线程的生命周期和状态控制。

10.3.1 线程的生命周期和状态

线程的生命周期是指一个线程从产生到消亡的发展过程。在这个过程中，每一个时刻都是线程生命周期的一个阶段。我们把线程生命周期的不同阶段称作线程的不同状态。图 10.2 描述了线程的状态变化及引起状态变化的原因。

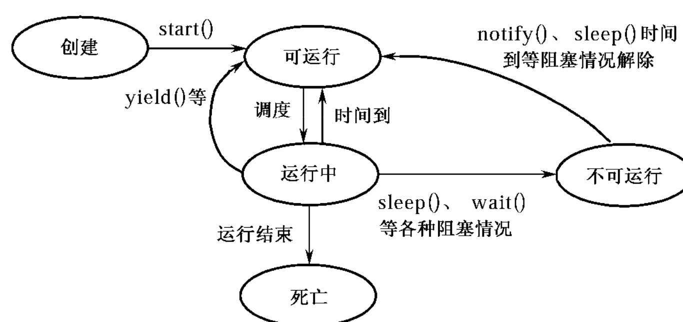


图 10.2 线程状态变化图

可以用 Thread 类中提供的方法和 Object 类提供的 wait()和 notify()方法在程序中改变线程的状态。

1．创建状态（new Thread）

通过继承 Thread 类或实现 Runnable 接口，可以定义一个具有线程功能的类，再使用 new 运算符就可以创建一个线程对象。例如，执行下面语句后，线程就处于创建状态：

```
Thread myThread = new MyThreadClass( );
```

这时，线程 myThread 处于创建状态，它仅仅是一个空的线程对象，还没有被分配可运行的系统资源（主要是没有分配 CPU 时间）。

2．可运行状态（Runnable）

运行一个线程必须具备两个条件：①可以使用的 CPU；②运行线程的代码和数据。运行线程的代码和数据由线程对象在 run 方法中提供。CPU 则由线程调度程序分配。在调用了新创建线程对象的 start()方法后，线程就处于可运行状态。

可运行状态的线程可能会同时有很多，这些可运行状态的线程在一个优先级队列中排队，等待操作系统的线程调度程序调度。

可运行状态的线程来自下列三种情况之一：

① 一个线程创建完毕；

② 处于运行状态线程的时间片到（系统给每个运行状态线程规定有时间）等；

③ 处于阻塞状态的线程的阻塞条件解除。例如，调用 sleep()方法延迟的时间到，notify()方法唤醒了 wait()方法阻塞的线程。

3．运行状态（Running）

线程调度程序按照一定的线程调度原则，从等待运行的处于可运行状态的线程队列中选择一个，使它获得 CPU 的使用权。这时，该线程就从可运行状态变为了运行状态。

处于运行状态的线程将首先执行 run()方法。

4．不可运行状态（not Runnable）

不可运行状态也称为阻塞状态（Blocked）。因为某种原因系统不能执行线程的状态称为不可运行状态。这时就是处理器空闲也不能执行该线程。

进入不可运行状态的原因通常有：

① 线程调用了 sleep()方法；
② 线程调用了 Object 类的 wait()方法；
③ 输入/输出流中发生了线程阻塞。

5．死亡状态（Dead）

有两种情况使线程处于死亡状态：

① 自然消亡，即 run()方法执行完；
② 应用程序停止运行。

10.3.2 线程分组

每一个线程都是一个线程组的成员。线程分组提供了一个统一管理多个线程而不需单独管理的机制。例如，当多个线程分为一组时，可以用一个方法启动或挂起线程组中的所有线程。Java 语言的 java.lang 包中的 ThreadGroup 子包提供了实现线程分组的类。

当创建一个线程时，系统就把这个线程放在一个线程组中。一个线程放在一个线程组中后，它就是这个线程组中的永久成员，不能再把这个线程加入其他线程组中。在应用程序中，当建立一个线程时，如果不指定线程组，系统会把这个线程放入缺省线程组中去。缺省线程组的组名是 main。图 10.3 所示是一个线程组和它的成员线程。

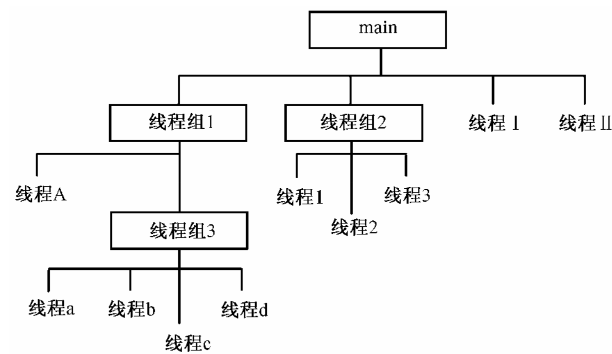


图 10.3 线程组和它的成员线程

ThreadGroup 类的构造方法和常用方法如下：

（1）构造方法

⌘ ThreadGroup(String name)

⌘ ThreadGroup(ThreadGroup parent, String name)

其中，name 是线程组名，parent 是父线程组名，默认值是当前线程组

（2）常用方法

⌘ String getName()　　返回当前线程组名

⌘ int activeCount()　　获得本组活动线程的数目

⌘ int enumerate(Thread[] list)　　将本组活动线程拷贝到数组 list 中

【例 10.3】　应用线程组来控制线程的运行示例。

要求：首先创建并启动两个线程，然后创建线程组对象，最后输出线程组对象中的所有线程名。

程序设计如下：

```
public class EnumerateTest extends Thread
{
    public void listCurrentThreads()                    //列出线程组的所有线程名
    {
        ThreadGroup currentGroup =
            Thread.currentThread().getThreadGroup();
                                                        //获得当前线程所在的线程组对象
        int numThreads = currentGroup.activeCount();    //获得活动线程数目
        System.out.println("numThreads = " + numThreads);
        Thread[] listOfThreads = new Thread[numThreads-1];
                                                        //创建线程对象数组
        currentGroup.enumerate(listOfThreads);
                                                        //将本组活动线程复制到数组listOfThreads中
        for (int i = 0; i < numThreads – 1; i++)
            System.out.println("Thread #" + i + " = " +
                        listOfThreads[i].getName());    //输出每个线程对象的名称
    }

    public static void main(String[] args)
    {
        EnumerateTest a =new EnumerateTest();           //创建线程对象 a
        EnumerateTest b =new EnumerateTest();           //创建线程对象 b
        a.start();                                      //启动线程
        b.start();                                      //启动线程
        a.listCurrentThreads();                         //列出线程组的所有线程名
    }
}
```

程序运行结果如下：

```
numThreads = 4
Thread #0 = main
Thread #1 = Thread-1
Thread #2 = Thread-2
```

程序运行结果说明：由于程序中没有为线程名定义成员变量（因此没有定义构造方法），所以系统自动调用 Thread 类的构造方法给每个线程对象一个默认名，这里两个线程对象的默认名分别是 Thread-0 和 Thread-1。

10.3.3 线程的优先级

线程有优先级。线程调度程序在进行线程调度时要考虑线程的优先级。另外，线程的执行顺序还与操作系统的线程调度方式有关。

如果操作系统线程调度方式是抢先式调度，则在当前线程执行过程中，一个更高优先级的线程一旦进入可运行状态，则这个线程将立即被安排运行。如果操作系统线程调度方式是独占方式，则线程一旦获得执行权，就一直执行下去，直到执行完毕或由于某种原因主动放弃 CPU 后，其他线程才能按优先级顺序被调度执行。

线程的优先级用数字来表示，范围从 1 到 10，即 MIN_PRIORITY 的值最小为 1，MAX_PRIORITY 的值最大为 10。一个线程的默认优先级是 5，即没有给线程优先级赋值时，线程优先级 NORM_PRIORITY 值为 5。

（1）Thread 类中关于优先级的静态成员变量

⌘ static final int NORM_PRIORITY　　线程默认优先级
⌘ static final int MIN_PRIORITY　　最低优先级
⌘ static final int MAX_PRIORITY　　最高优先级

（2）Thread 类中对优先级进行操作的方法

⌘ int getPriority()　　返回线程的优先级
⌘ void setPriority(int newPriority)　　设置线程的优先级为 newPriority

【例 10.4】　通过线程的优先级来控制线程的运行示例。

要求：创建并启动 5 个线程，并为它们设立不同的优先级。每个线程的执行代码都是循环两次，打印出当前线程的优先级，每次打印后，让当前线程休眠一段时间，这样线程调度程序就可以把处于可运行状态的其他线程按照优先级排队次序调度进入运行状态。

程序设计如下：

```
public class TestThreadPrio extends Thread
{
    private String name;

    TestThreadPrio (String name)
    {
        this. name = name;
    }

    public void run()
    {
```

```
        for ( int i=0; i<2; i++ )
        {
            System.out.println(name +" "+getPriority() );              //输出
            try
            {
                Thread.sleep((int)(Math.random()*100));                //休眠
            }
            catch(InterruptedException e){}
        }
    }

    public static void main( String args [] )
    {
        Thread t1 = new TestThreadPrio("Thread1");                     //创建线程 t1
        t1.setPriority(Thread.MIN_PRIORITY);                           //设置优先级为 1
        Thread t2 = new TestThreadPrio("Thread2");                     //创建线程 t2
        t2.setPriority(3);                                             //设置优先级为 3
        Thread t3 = new TestThreadPrio("Thread3");                     //创建线程 t3
        t3.setPriority(5);                                             //设置优先级为 5
        Thread t4 = new TestThreadPrio("Thread4");                     //创建线程 t4
        t4.setPriority(7);                                             //设置优先级为 7
        Thread t5 = new TestThreadPrio("Thread5");                     //创建线程 t5
        t5.setPriority(9);                                             //设置优先级为 9
        t1.start();                                                    //启动线程 t1
        t2.start();                                                    //启动线程 t2
        t3.start( );                                                   //启动线程 t3
        t4.start();                                                    //启动线程 t4
        t5.start( );                                                   //启动线程 t5
    }
}
```

程序运行的结果如下：

```
Thread5 9
Thread5 9
Thread4 7
Thread4 7
Thread3 5
Thread3 5
Thread2 3
Thread2 3
Thread1 1
Thread1 1
```

程序运行的结果说明：线程的执行顺序既和线程的优先级有关，也和操作系统的线程调度方式有关。另外，线程的运行具有不确定性。所以，读者在自己的计算机上运行上述程序时结果可能不一样。实际上，就是作者自己运行上述程序时，每次的运行结果也略有不同。

10.4 线程间的互斥

到目前为止，我们仅介绍了独立的、非同步的线程，也就是说，每一个线程在运行时与其他线程的状态和行为无关。但是，在实际软件系统中，许多线程之间有时会共享一些数据，并且它们之间的状态、行为是相互影响的。线程间共享的数据，以及线程状态、行为的相互影响有两种：一种是线程间的互斥，一种是线程间的同步。本节讨论线程间的互斥，下节讨论线程间的同步。

10.4.1 共享资源问题

1．共享资源

共享资源是指在程序中并行运行的若干个线程操作相同的数据资源，这样的数据资源叫做共享资源。

在程序中，并行运行的若干个线程操作共享资源时可能出错。例如，在一个民航的售票系统中，一个售票处工作人员（甲）欲给一个客户分配一个座位，于是调出飞机的座位分配图，但当甲察看座位分配图时，另外一个售票处的工作人员（乙）也在为其他客户分配座位，乙也调出了这张飞机座位分配图。乙为客户分配了座位 6A，同时在图中标记此座位（6A）已被分配。此时，甲还在察看那张未被更新的座位分配图，甲并不知道座位分配图已经发生了变化（座位 6A 已被分配），甲也为客户分配了座位 6A。这时问题就出现了，一个座位被两个人同时拥有，这在一个完善的民航售票系统中是绝对不允许的。那么，问题出在哪里呢？显然是飞机的座位分配图，这是一个共享的数据。在一个工作人员使用它时，应该禁止其他工作人员使用。试想，如果乙在甲使用完后，即在甲完成售票工作并更新了座位分配图后，再使用更新后的座位分配图，这时还会出现上述问题吗？当然不会。因为在新的座位分配图中，座位 6A 已经被标明“已分配”。

这类问题实际是一种生产者—消费者问题。生产者一次或多次提供货物，若干个消费者同时消费。我们定义这种生产者—消费者问题是**生产者—消费者模式Ⅰ**。

下面再通过一个程序例子说明，并行运行的若干个线程操作共享资源时可能存在问题。

【例 10.5】 两个线程同时操作一个共享资源（Queue 对象）。

要求：程序中首先向队列中插入 4 个字符，然后创建并启动两个删除队列中字符的线程。要求删除队列中字符的线程具体设计成：处于运行状态的线程，在任务还未处理完的情况下，调用 sleep()方法转入不可运行状态，从而让另一个线程有机会从可运行状态转入运行状态（说明：队列是一个先进先出的线性表，即先插入队列中的数据将先被删除）。

设计分析：按照任务要求，程序应包括三个部分，分别是：队列类、删除字符的线程类和测试类。

程序设计如下：

（1）Queue（队列）类

```
public class Queue
{
    private int count;                              //成员变量，个数计数器
    private int front;                              //成员变量，队头下标
    private int rear;                               //成员变量，队尾下标
    private char[ ] dat = new char[10];             //成员变量，保存数据的数组

    public Queue()                                  //构造方法
    {
        count=0;
        front = 0;
        rear = 0;
    }

    public void push(char c)                        //插入方法，在队尾插入字符 c
    {
        if (count >= 10)
        {
            System.out.println("队列已满");
            return;
        }
        dat[rear] = c;                              //插入字符 c
        System.out.println("插入的字符: " + c);     //输出
        rear++;                                     //队尾指针加 1
        count ++;                                   //个数加 1
    }

    public char pop()                               //删除方法，删除队头字符并返回
    {
        if (count <= 0)
        {
            System.out.println("                    队列已空");
            return ' ';
        }
        char temp = dat[front];
        System.out.println("                        删除的字符: " + temp);
        try
        {
            Thread.sleep((int)(Math.random()*100));     //休眠
        }catch(InterruptedException e){ }               //处理异常
```

```
            front++;                                    //队头指针加 1
            count−−;                                    //个数减 1
            return temp;
        }
    }
```

Queue 类的 push()方法和 pop()方法分别完成插入一个字符和删除一个字符。为了说明共享资源可能存在的问题，我们在 pop()方法中使用了休眠语句。这样，在这段休眠时间里（这意味着操作者的这个操作还未执行完），其他操作者可能会访问共享资源 Queue 中的数据。

（2）删除 3 个字符的实现接口 Runnable 的 Consumer 类

```
public class Consumer implements Runnable
{
    private Queue qu;                                   //成员变量

    public Consumer(Queue s)                            //构造方法
    {
        qu = s;
    }

    public void run()
    {
        for(int i=0; i<3; i++)
            qu.pop();
    }
}
```

这个线程将对一个 Queue 对象进行连续删除 3 个字符的操作。

（3）两个删除线程同时访问共享队列的测试类

```
public class TestQueue
{
    public static void main(String args[])
    {
        Queue qu = new Queue();                         //创建队列 qu
        for(char c ='a'; c <='d'; c ++)
            qu.push(c);
        Runnable sink = new Consumer(qu);
//用 qu 创建 Consumer 接口 sink
        Thread t1 = new Thread(sink);                   //用 sink 创建线程 t1
        Thread t2 = new Thread(sink);                   //用 sink 创建线程 t2
        t1.start();                                     //启动线程 t1
        t2.start();                                     //启动线程 t2
    }
}
```

程序运行的结果如下：

```
插入的字符: a
插入的字符: b
插入的字符: c
插入的字符: d
                删除的字符: a
                删除的字符: a
                删除的字符: b
                删除的字符: c
                删除的字符: d
                队列已空
```

程序运行结果分析：显然，字符 a 被删除了两次。造成错误的原因是，第一个删除线程（如 t1）在删除了字符 a 后，没有执行语句 front++（修改队头指针）和语句 count --（个数减 1）就进入休眠（这意味着这个操作还未执行完）。这样，在第一个删除线程休眠的这段时间里，第二个删除线程（如 t2）被启动，此时，第二个删除线程删除的自然也是字符 a。

注意：因这三个类都是 public 类，所以要分别存放在三个文件中。

2．共享资源的互斥

基于上述分析，应当认识到，当并行运行的几个线程操作共享资源，且问题的模型属于生产者-消费者模式 I 时，一定要保证操作的互斥，即一个共享资源每次只能由一个线程操作。当这个线程还没有操作完时，不允许其他线程操作共享资源。只有这样，才能保证并行运行的多个线程对共享资源操作的正确性。

10.4.2　互斥线程的设计方法

为了避免多个并行运行的线程对共享资源操作时可能出现的问题，Java 语言引入了互斥锁。互斥锁是基于共享资源的互斥性设计的，用来标记那些多个并行运行的线程共享的资源。被互斥锁标记的共享资源，在一个时间段内，只能有一个线程使用；只有当加互斥锁的线程使用完了该共享资源，另一个线程才可以使用。这样就可以保证线程对共享资源操作的正确性。

Java 语言中，关键字 synchronized 就是用来给共享资源加互斥锁的。当某个共享资源被用 synchronized 修饰时，就表明该共享资源在某段时间内只能由加锁的线程访问。

为共享资源加互斥锁有两种方法：

（1）锁定一个对象和一段代码

声明格式为：

```
synchronized (〈对象名〉)
{
    〈语句组〉
}
```

对象表示要锁定的共享资源，一对花括号内的语句组表示锁定对象期间执行的语句，或

者说，表示对象的锁定范围。此格式可以用来在一个线程的一部分代码上加互斥锁。

当一个线程执行这段代码时，就锁定了指定的对象。此时，如果其他线程也要对加了互斥锁的对象进行操作，就无法进行；其他线程必须等候，直到该对象的锁被释放为止。加锁的线程执行完花括号内的语句后，将释放对该对象加的锁。这就形成了多个线程对同一个对象的“互斥”使用方式，因此该对象也称为互斥对象。

（2）锁定一个方法

声明格式为：

```
synchronized 〈方法声明〉
{
    〈方法体〉
}
```

这里虽然没有指出锁定的对象，但是一个方法必然属于一个类，因此，此格式锁定的是该方法所属类的对象，锁定的范围是整个方法，即在一个线程执行整个方法期间对该方法所属类的对象加互斥锁。

【例 10.6】 修改例 10.5，避免对共享资源（队列）操作出错。

例 10.5 产生错误的原因是：并行运行有两个线程，由于线程执行代码的时间较长（人为加入了休眠语句），一次操作尚未运行完，就可能转入运行另一个同样操作的线程，从而造成两个删除操作线程删除了同一个数据。

修改的方法是：对 pop()方法加互斥锁 synchronized，在一个线程尚未运行完（即未解锁前）时，不允许另一个线程操作共享资源——队列。

程序修改如下：

```
public synchronized char pop()                    //加互斥锁的删除方法
{
    if (count <= 0)
    {
        System.out.println("队列已空");
        return ' ';
    }
    char temp = dat[front];
    System.out.println("删除的字符: " + temp);
    try
    {
        Thread.sleep((int)(Math.random()*100));   //休眠
    }catch(InterruptedException e){}              //处理异常
    front++;                                      //队头指针加 1
    count --;                                     //个数减 1
    return temp;
}
```

运行程序结果如下。

```
插入的字符: a
插入的字符: b
插入的字符: c
插入的字符: d
                    删除的字符: a
                    删除的字符: b
                    删除的字符: c
                    删除的字符: d
                    队列已空
                    队列已空
```

运行结果分析：

① 由于程序中对 pop()方法加了互斥锁（即标识符 synchronized），任何一个线程调用该方法时，必须等待该方法的所有语句都执行完后，其他线程才能操作 pop()方法所属类的共享资源——队列。这样就保证了并行运行的两个线程对共享资源队列操作的正确性。

② 两个删除字符线程共执行了 6 次删除操作，所以最后两次输出"队列已空"。这样的输出是队列当前状态的正确反映，并没有出错。

10.5 线程间的同步

上节讨论了线程间的互斥。还有一种线程间对共享资源的相互影响，是一个线程的运行要依赖于另一个线程对共享资源的处理结果，这称为线程间的同步。本节讨论线程间的同步。

10.5.1 共享资源的同步问题

用计算机模拟生产—销售问题。规定：①由于仓库有限，必须把仓库中的商品销售完后才能再生产；②由于担心不能及时供货被罚款，只有当仓库中有货时才能销售。在这类问题中，两个操作（生产和销售）中任何一个操作执行的前提，是另一个操作已经完成。另一个操作已经完成的标志是共享资源（仓库）满足本操作的执行条件。

这样一类问题也称作生产者—消费者问题。我们定义这种生产者—消费者问题是**生产者—消费者模式 II**。消费者操作执行的前提条件是生产者操作已经生产了；生产者操作执行的前提条件是消费者已经消费了。

在存在共享资源同步问题的程序设计中，如果不考虑共享资源的同步问题，必然引起错误。

【例 10.7】 计算机模拟生产—销售问题。

要求：为了简化生产—销售问题，规定仓库中一次可放若干数量的货物，但必须满足：仓库中一旦有货就一次销售完；仓库中一旦无货就一次完成本次生产。

设计分析：需要设计四个类，分别是仓库类、生产类、销售类和测试类。

程序设计如下：

（1）仓库类

```
public class Storage
{
```

```
    private int goods;                          //货物的数量

    public Storage(int g)
    {
        goods = g;
    }

    public void put(int g)                      //向仓库放货
    {
        goods = g;
    }

    public int get()                            //从仓库取货
    {
        int temp = goods;
        goods = 0;
        return temp;
    }
}
```

（2）生产类

```
public class Producer extends Thread            //生产者 Producer 类
{
    private Storage tb;                         //生产者存放货物的仓库对象

    public Producer(Storage c)
    {
        tb = c;
    }

    public void run()
    {
        for (int i = 11; i < 16; i++)           //开始生产后，不断向仓库放货
        {
            tb.put(i);
            try
            {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
            System.out.println("put goods = " + i);
        }
    }
}
```

为了方便测试，run()方法执行一次循环语句就休眠一段时间（表示本操作尚未执行完）。

（3）销售类

```
public class Consumer extends Thread                    //消费者 Consumer 类
{
    private Storage tb;                                 //消费者取货的仓库对象

    public Consumer(Storage c)
    {
        tb = c;
    }

    public void run()
    {
        int g;
        for (int i = 11; i < 16; i++)                   //开始消费后，不断从仓库取货
        {
            g = tb.get();
            try
            {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
            System.out.println("            get goods = " + g);
        }
    }
}
```

为了方便测试，run()方法执行一次循环语句就都休眠一段时间（表示本操作尚未执行完）。

（4）测试类

```
public class TestStorage
{
    public static void main(String[] args)
    {
        Storage com = new Storage(0);
                                    //创建仓库对象 com，初值为 0，表示空仓库
        Producer p = new Producer(com);
                                    //创建生产者对象 p，生产的货物放在仓库 com 中
        Consumer c = new Consumer(com);
                                    //创建消费者对象 c，从仓库 com 中取出货物消费
        p.start();                  //生产者对象 p 开始生产
        c.start();                  //消费者对象 c 开始消费
    }
}
```

程序运行结果如下：

```
                    get goods = 11
put goods = 11
                    get goods = 0
put goods = 12
                    get goods = 12
                    get goods = 13
put goods = 13
put goods = 14
                    get goods = 0
put goods = 15
```

程序运行结果分析：显然，程序运行结果不符合要求，因为既存在连续向仓库放两次货物的行为，也存在连续从仓库取两次货物的行为，甚至存在仓库中还没有货物就取到货物的行为。

出现错误的原因是没有协调好生产和消费的步调。或者说，出现错误的原因是没有对两个线程进行同步。

10.5.2 同步线程的设计方法

线程同步的设计方法是，除了加互斥锁外，还要在并行运行的线程上加信号量。信号量是一个标志，表示一种操作（如生产或销售）是否已执行完，另一种操作（如销售或生产）是否可以执行了。

Object 类提供了一组关于线程同步的方法——wait()方法和 notify()方法。现介绍如下：

- ⌘ void wait()　　引起当前线程等待,直到另一个线程调用 notify()方法或 notifyAll()方法
- ⌘ void wait(long time)　　引起当前线程等待,直到另一个线程调用 notify()方法或 notifyAll()方法，或者等待的时间 time 到
- ⌘ void notify()　　唤醒一个正在等待这个对象的线程
- ⌘ void notifyAll()　　唤醒所有正在等待这个对象的线程

wait()方法的所谓等待，是把当前线程从运行状态转为阻塞状态；notify()方法的所谓唤醒是把等待线程从阻塞状态转为可运行状态，从而有机会让线程调度程序调度进入运行状态。

还有一点要说明的是：wait()方法所在的代码段一定要加互斥锁 synchronized。这是因为：wait()方法在把当前线程从运行状态转为阻塞状态后，还要释放互斥锁锁定的共享资源（否则其他同步线程无法运行），这样的操作不允许中间被打断，这显然是共享资源的互斥问题。因此，wait()方法所在的代码段一定要加互斥锁 synchronized。

【例 10.8】　修改例 10.7 程序，使之满足生产和销售的同步要求。

程序设计如下：

```
class Storage
{
    private int goods;
```

```
    private boolean available;                    //信号量

    public Storage(int g)
    {
        goods = g;
        available = false;                        //设定初始信号量值为 false
    }

    public synchronized void put(int g)
    {
        while(available == true)                  //等待信号量满足要求
        {
            try
            {
                wait();
            } catch(InterruptedException e) {}
        }
        available = true;                         //修改信号量值

        goods = g;
        System.out.print("put goods = " + goods);
        notify();
    }

    public synchronized int get()
    {
        while(available == false)                 //等待信号量满足要求
        {
            try
            {
                wait();
            } catch(InterruptedException e) {}
        }
        available = false;                        //修改信号量值

        int temp = goods;
        System.out.println("                      get goods = " + goods);
    goods = 0;
        notify();
        return temp;
    }
}

class Producer extends Thread
```

```
{
    private Storage tb;
    public Producer(Storage c)
    {
        tb = c;
    }

    public void run()
    {
        for (int i = 11; i<16; i++)
            tb.put(i);

    }
}

class Consumer extends Thread
{
    private Storage tb;
    public Consumer(Storage c)
     {
         tb = c;
     }

     public void run()
     {
         int g;
         for (int i = 11;i < 16; i++)
             g = tb.get();
     }
}

public class TestStorage2
{
     public static void main(String[] args)
     {
         Storage com = new Storage(0);
         Producer p = new Producer(com);
         Consumer c = new Consumer(com);
         p.start();
         c.start();
     }
}
```

程序运行结果如下：

```
put goods = 11          get goods = 11
put goods = 12          get goods = 12
put goods = 13          get goods = 13
put goods = 14          get goods = 14
put goods = 15          get goods = 15
```

程序运行结果显示，完全达到了题目要求的满足生产和销售同步的设计要求。

程序设计说明：

① 和例 10.7 的程序相比，修改了四处：(a) 在 Storage 类中增加了 Boolean 类型的信号量成员变量 available；(b) 在 Storage 类的 put()方法和 get()方法定义中增加了互斥锁标识符 synchronized；(c) 把在线程中调用 Thread 类的 sleep()方法，改为在 Storage 类的 put()方法和 get()方法中调用 Object 类的 wait()方法；(d) 在 Storage 类的 put()方法和 get()方法中增加调用 Object 类的 notify()方法。

② 信号量 available 的作用是控制线程的同步操作。

③ wait()方法和 notify()方法是配对的一组方法。程序中当判断信号量不满足要求时调用 wait()方法，则使当前线程从运行状态转入阻塞状态，并释放当前运行线程互斥锁锁定的共享资源——对象 com；当判断信号量满足要求时，执行完方法（put()方法或 get()方法）要求的语句后，调用 notify()方法，唤醒一个正在等待这个共享资源——对象 com 的线程。

④ Thread 类的 sleep()方法和 Object 类的 wait()方法虽然让用户感觉都使程序执行过程迟缓了一些，但两者有根本的不同：Thread 类的 sleep()方法只是延缓一段时间再执行后续代码。具体执行过程是：sleep()方法使处于运行状态的线程进入阻塞状态，但休眠时间一到，就从阻塞状态自动转入可运行状态；而 Object 类的 wait()方法让用户感到的延缓，相同之处也是使当前线程从运行状态转入了阻塞状态，但不同点是，wait()方法的等待时间是不固定的，它什么时候被唤醒依赖于其他线程的操作。另外，调用 sleep()方法休眠期间，如果该段代码（或方法）加了互斥锁，则互斥锁锁定的共享资源不释放，而 wait()方法将释放互斥锁锁定的共享资源，否则其他同步线程无法运行。

10.6 综合应用举例

下面的例 10.9 程序设计问题，包含了第 5 章的 Data 类、第 6 章的 Component 类的绘图方法、第 7 章的 Java 小程序、第 8 章的异常处理及本章的多线程，所以是一个综合应用问题。

【例 10.9】 设计一个 Java 小程序，要求在图形界面中显示当前时间，并要求每隔一秒钟执行图形界面的刷新功能。

设计分析：显然，程序包含两个子任务，一个是在窗口中显示系统的当前时间，另一个是每隔一秒钟取一次系统当前时间，并把显示窗口刷新。由于，每隔一秒钟取一次系统当前时间这个子任务是时间一到就要执行，所以要设计成独立的线程。由于所设计的是一个 Java 小程序，必须继承 Applet 类，因此多线程功能只能通过实现 Runnable 接口来达到。

Java Applet 设计如下：

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

```
import java.util.*;
import java.text.DateFormat;
import java.applet.*;
public class Clock extends Applet implements Runnable
                                              //支持多线程的 Applet
{
    private Thread clockThread;               //线程对象

    public void init()                        //重写 Applet 类的 init()方法
    {
        setBackground(Color.white);           //设置背景色为白色
        clockThread = null;                   //线程对象初值为 null
    }

    public void start()                       //覆盖 Applet 类的 start()方法
    {
        if (clockThread == null)
        {
            clockThread = new Thread(this, "Clock");   //创建时钟线程
            clockThread.start();                        //启动线程
        }
    }

    public void run()                         //实现 Runnable 接口 run()方法
    {
        while (clockThread !=null)
        {
            repaint();                        //重绘图形
            try
            {
                Thread.sleep(1000);           //休眠一秒
            }
            catch (InterruptedException e){ }
        }
    }

    public void paint(Graphics g)             //覆盖 Component 类的 paint()方法
    {
        Date date = new Date();               //获取当前时间
        DateFormat dateFormatter = DateFormat.getTimeInstance();
                                              //创建时间格式对象
        g.setFont(new Font("Serif",Font.BOLD,34));     //设置输出字体
```

```
            g.drawString(dateFormatter.format(date), 5, 30);
                                                        //绘制时间字符串
        }

        public void stop()                              //覆盖 Applet 类的 stop()方法
        {
            clockThread = null;                         //停止线程对象运行
        }
    }
```

HTML 文档设计如下：

```
<HTML>
<APPLET
    code    = "Clock"
    width   = "500"
    height  = "300"
    >
</APPLET>
</HTML>
```

程序运行结果如图10.4所示。如果是在计算机上实际运行该程序，图形界面将每隔一秒钟将刷新一次。

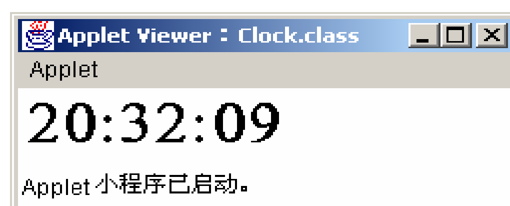


图 10.4　时间 Applet

程序设计说明：

① Java不支持多继承，即不能同时继承Applet类和Thread类，但Java支持接口（且允许一个类中同时实现若干个接口）。因此可以在继承Applet类的同时实现Runnable接口。本例采用继承Applet类的同时实现Runnable接口的方法实现了多继承。

② Clock类继承了Applet类，Clock类中的init()方法、start()方法和stop()方法是对Applet类中相应方法的覆盖，即程序运行时执行的是Clock类定义的这些方法，而不是执行Applet类定义的相应方法。

③ 由于Applet类继承了Component（组件）类，所以它拥有Component类的绘图方法，Clock类中的paint()方法是对Component类相应方法的覆盖，完成具体Clock类对象要求的图形绘制。关于Component类的绘图方法参见6.4.1节，关于Applet类对Component类的继承参见7.3.1节。

④ Clock类中的run ()方法是对Runnable接口中run ()方法的实现。

⑤ 本例Java Applet的运行过程是：当在浏览器中浏览HTML文档时，嵌入在HTML文档

中的Clock类即被下载到本地计算机并运行。在Java Applet创建阶段，系统调用了init()方法，将背景色设置为白色；随后系统调用start()方法，创建并启动时钟线程；时钟线程启动即意味着系统调用run ()方法。run ()方法中，当判断出当前线程是时钟线程时，首先调用repaint()方法重绘图形，repaint()方法将自动调用paint()方法获取当前时间并以字符串形式显示在窗口中起始坐标为（500, 300）的位置上，然后让时钟线程休眠一秒钟，这样Java Applet主线程就被线程调度程序调度为运行状态，Java Applet主线程（Java Applet程序）将运行paint()方法将图形窗口的当前显示时间刷新一次。

习题 10

一、基本概念题

10.1　什么是进程？什么是线程？它们有什么区别？试举例说明。

10.2　什么是多线程？ 多线程程序有什么特点？Java 是如何支持多线程的?

10.3　什么是线程的生命周期？线程在它的生命周期中都有哪些状态？

10.4　什么是线程的优先级？它的主要用途是什么？

10.5　如何改变线程的状态？run()方法的作用和 start()方法的作用有什么不同？

10.6　下面说法是否正确？为什么？

① 要在程序中实现线程必须导入 java.io.Thread 类。

② 线程启动后，该线程实际运行的代码放置在 run()方法中。

③ 线程之间可以共享数据。

④ 启动线程时调用的是 run()方法，而不是 start()方法。

10.7　哪些方法可以终止线程的运行？

10.8　定义一个线程类有几种基本方法？它们各有什么特点？

10.9　多线程对共享资源的访问会带来什么问题？如何解决这些问题？

10.10　并行运行的若干个线程操作共享资源时可能存在什么问题？如何解决？

10.11　什么是共享资源的同步问题？如何解决？

二、程序设计题

10.12　编译下面代码，解释出现的情况。

```
public class A implements Runnable
{
    public static void main(String[] arg)
    {
        A   r = new A();
        Thread t= new Thread(r);
        t.start();
    }
    public void start()
    {
        for (int i=0;i<10;i++)
```

```
            System.out.print(i);
        }
    }
```

10.13 **编译下面代码，解释出现的情况。**

```
public class A implements Runnable
{
    public static void main(String[] arg)
    {
        A r = new A();
        Thread t= new Thread(r);
        t.start();
    }
    public void run()
    {
        while(true)
        {
            Thread.currentThread().sleep(1000);
            System.out.println("in the while");
        }
    }
}
```

10.14 **设计一个程序，该程序中包括 5 个线程，设法控制线程运行先后顺序。通过运行结果得出结论。**

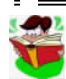
10.15 **设计一个程序，每隔一秒钟显示一次本地机系统时间。**

10.16 **设计一个程序，该程序包含两个线程，一个线程做冒泡排序，一个线程做插入排序。**

10.17 **设计一个生产者—消费者模式 I 的程序。对一个对象（堆栈）进行操作，生产者是一个压入线程，它不断向堆栈中压入数据；消费者是一个弹出线程，它不断从堆栈中弹出数据。设计的程序应避免共享资源带来的问题，使两个线程能够正确地对堆栈对象进行操作。**

10.18 **设计一个生产者—消费者模式 II 的程序。对一个对象（枪膛）进行操作，生产者线程是一个压入线程，它不断向枪膛中压入子弹；消费者是一个射出线程，它不断从枪膛中射出子弹。设计的程序应考虑线程的同步问题，使两个线程能够正确地对枪膛对象进行操作。**

第 11 章　数据库应用



教学要点

本章内容主要包括 JDBC 和 SQL 的基本概念，建立应用程序和数据库连接的环境配置，数据库应用程序设计方法（包括建立应用程序和数据库的连接、操作数据库、处理操作结果）。

要求掌握 JDBC 和 SQL 的基本概念，熟练掌握建立应用程序和数据库连接的环境配置方法、数据库应用程序的设计方法。

Java 在实际应用中和数据库有着密切的联系，可以在 Java 程序中对各种各样的数据库进行操作。JDBC 是 Java 数据库连接技术的简称。本章主要介绍 Java 语言对关系型数据库的操作方法和 JDBC 中用于数据库操作的类和接口。并重点讨论如何装载数据库的驱动程序，如何建立与数据库的连接，如何访问数据库中的数据等基本的数据库操作方法。

如果学生没有数据库的基本概念，建议教师授课时补充相关内容或跳过本章。

11.1　JDBC 和 SQL 简介

学习本章的内容，要求读者已经对数据库的基本知识有了一定的了解。如果读者认为自己还缺乏数据库的基本知识，可参阅相关参考文献。

JDBC（Java DataBase Connectivity 的缩写）是 Java 语言数据库连接技术的简称。**JDBC API**（Application Program Interface 的缩写） 是 Java 程序和数据库建立连接和访问数据库中数据的应用程序接口。在使用 JDBC 的 Java 应用程序中，对数据库的操作主要有下面四个步骤：

① 和目标数据库建立连接。

② 向数据库发送 SQL 语句，实现对数据库的访问和操作。

③ 从数据库返回结果到应用程序。

④ 关闭数据库连接。

由于 SQL 语言在数据库应用中起着非常重要的作用，因此，在介绍 JDBC 之前，首先对 SQL 语言做一个简单的介绍。

11.1.1　SQL 简介

SQL 语言是一种定义、操纵、查询和控制关系型数据库的结构化查询语言。目前，常用的关系型数据库有 SQL Server, Oracle, Sybase, MySQL, Access 等。这些不同的关系型数据库，都支持 SQL 语言。

SQL 语言的定义、操纵、查询和控制四方面的功能，是由表 11.1 列出的九个命令动词完成的。

表 11.1　SQL 语言的动词

SQL 功能	动　词	说　明
数据定义	CREATE, DROP, ALTER	创建表、删除表、修改表
数据操纵	INSERT, UPDATE, DELETE	插入、更新、删除
数据查询	SELECT	查询
数据控制	GRANT, REVOKE	授予权限、收回权限

1．SQL 语言的使用方式

SQL 语言以同一种语法提供两种使用方式：自含式和嵌入式。

① 自含式：是指独立地用于联机交互的使用方式。用户可以从键盘上直接键入 SQL 命令对数据库进行操作。

② 嵌入式：是指把 SQL 语句嵌入到高级语言的使用方式。在 Java 语言中使用 SQL 语言，就是 SQL 语言的嵌入式使用方式。

在这两种不同的使用方式下，SQL 语言的语法结构基本一样，仅在某些地方略有不同。

2．数据定义

数据定义主要用于定义表。常用的命令和命令格式如下：

（1）CREATE 命令

CREATE 命令用来创建一个新的表，定义表的 CREATE TABLE 语句格式如下：

```
CREATE TABLE 表名
     (列名    数据类型    [列级完整性约束条件]
      {,列名    数据类型    [列级完整性约束条件]}
      [,表级完整性约束条件]);
```

其中，〈表名〉是所要创建的表的名字，它由多个列组成，每列需要声明列名和所属数据类型。[]和{}中的都是可选项，[]可重复 0 或 1 次，{}可重复 0 到多次。如果完整性约束条件涉及到该表的多个列，则必须定义在表级上，否则通常定义在列级上。

例如，若要求创建一个存储用户信息的表，表中的字段包括 name, password, birthday, sex, phone, email。SQL 语句如下：

```
CREATE TABLE userInfo                                  //用户信息表
{
    name        varchar(18) NOT NULL UNIQUE,      //姓名，要求非空
    password    varchar(18) NOT NULL UNIQUE,           //口令，要求非空
    birthday    date,                                  //生日
    sex         varchar(2),                            //性别
    phone       integer,                               //电话
    email       varchar(20)                            //电子邮件
```

```
    PRIMARY KEY (name)
};
```

执行上述命令后，创建的表的名称是 userInfo，主键是 name，表中包含的六个字段属于 varchar, date, integer 数据类型之一。另外，还定义了是否允许某字段为空值（有 NOT NULL UNIQUE 的字段不允许为空值）。

（2）DROP 命令

DROP 命令用于删除一个表，命令格式为：

```
DROP <表名称>;
```

例如，若要删除上述表 userInfo，可以使用如下命令：

```
DROP TABLE userInfo;                    //删除表
```

（3）ALTER 命令

ALTER 命令用于修改表中的字段，ALTER 命令格式如下：

```
ALTER TABLE  表名
    [ADD  新列名   数据类型  [完整性约束]]
    [MODIFY   列名   数据类型];
```

其中，ADD 子句用于增加新列及其完整性约束条件，MODIFY 子句用于修改原有的列定义，包括修改列名和数据类型。例如，

```
ALTER TABLE userInfo
    ADD    postcoding    varchar(6);          //添加邮政编码列

ALTER TABLE userInfo
    MODIFY name varchar(20);                  //修改 name 列的数据类型
```

3．数据操纵

表结构定义好之后，表中没有任何记录，可以通过 SQL 的数据操纵语句在表中插入、删除和修改记录。常用的命令和命令格式如下：

（1）INSERT 命令

INSERT 命令用于在指定表中添加记录，命令格式为：

```
INSERT INTO <表名称>(<字段名 1>，…，<字段名 n> )
        values (<字段值 1>，…，<字段值 n>);
```

例如，若要在表 userInfo 中添加一个记录，其 name 字段值为“ Wang Hai ”，password 字段值为“ asas ”，email 字段值为“ a@b.com ”，phone 字段值为“ 94994949 ”，“ birthday ”字段值为“ 12-2-78 ”，sex 字段值为“ 男 ”，可以使用如下命令：

```
INSERT    INTO userInfo
    (name, password, email ,phone, birthday,sex)
    values ('Wang Hai', 'asas', 'a@b.com', 94994949, 12-2-78, '男');
```

（2）DELETE 命令

DELETE 命令用来删除指定表中符合条件的记录，命令格式为：

```
DELETE FROM <表名称>
WHERE <条件语句>
```

其中，条件语句表示出要删除记录的条件。

例如，若要删除表 userInfo 中 name 字段的值为“ Wang Hai ”的记录，可以使用如下命令：

```
DELETE FROM userInfo
WHERE name ='Wang Hai';
```

（3）UPDATE 命令

UPDATE 命令用来修改指定表中符合条件的记录，命令格式为：

```
UPDATE <表名称>
    SET        <字段名 1> = <字段值 1>，…，<字段名 n> = <字段值 n>
    WHERE     <条件语句>;
```

其中，条件语句用来选择符合条件的记录。

例如，要把表 userInfo 中 name 字段的字段值为“ Wang Hai ”的记录，其 email 字段的值修改为“ fa@bdd.com ”，其 phone 字段的值修改为“ 8888 ”，则可使用如下语句：

```
UPDATE userInfo
SET email ='fa@bdd.com', phone = 8888
WHERE name ='Wang Hai';
```

4．数据查询

数据查询是数据库操作的核心功能。SQL 语言提供了 SELECT 语句进行数据库的查询，该语句具有灵活的使用方式和丰富的功能。SELECT 语句最基本的功能是选出指定表中符合条件的记录。

命令格式为：

```
SELECT [<字段名表>]
FROM <表名称>
WHERE<条件语句>;
```

其中，条件语句表示出所选择记录的条件，字段名表给出要显示的字段，如果没有给出字段名表，则表示要显示全部字段。

例如，要选择出表 userInfo 中 sex 字段值为“ 男 ”的所有记录，并显示出这些记录的 name 字段值和 email 字段值，可以使用下面命令：

```
SELECT    name, email
FROM userInfo
WHERE sex ='男';
```

最后要说明的是，SQL 语言中的语句标识符（如查询语句标识符 SELECT）既可以大写，也可以小写。

11.1.2 JDBC 简介

JDBC 由一组用 Java 语言编写的类和接口组成，它为软件开发人员提供了一个标准的应用程序接口，使他们能够通过 Java API 来编写数据库应用程序。

有了 JDBC，向各种关系数据库发送 SQL 语句就是一件很容易的事。应用程序可通过调用 JDBC API 中提供的类和接口的方法，向当前连接的数据库发送 SQL 语句，并返回 SQL 语句的执行结果。

另外，JDBC API 还是与平台无关的。我们不必为访问 Sybase 数据库专门写一个程序，为访问 Oracle 数据库又专门写另一个程序等。只须写一遍程序，就可让该程序在任何数据库平台上运行。

1．JDBC 的用途

简单地说，JDBC 可做三件事：

① 与数据库建立连接；

② 发送 SQL 语句；

③ 处理 SQL 语句的操作结果。

2．JDBC 的工作原理

JDBC API 通过数据库管理器和为各种具体数据库定制的驱动程序提供了和这些数据库的连接。JDBC 数据库管理器通过安装数据库专用的驱动程序来和各种具体数据库（称作数据源）建立连接。

一旦应用程序和数据库建立了连接，应用程序就可以操作这些数据库了。应用程序对数据库的操作，是通过数据库管理器将标准的 JDBC 指令，转换成适合不同数据库通信的网络协议指令或其他指令。这种指令的转换机制，是基于 JDBC API 开发的程序可以独立于具体的数据库。如果数据库更换了，只需在应用程序中把原来的数据库驱动程序更换为新的数据库驱动程序就可以了。

3．JDBC 驱动程序

JDBC 可实现应用程序和数据库的连接。所谓连接是指 Java 应用程序和数据库之间的通信连接。连接由 JDBC 驱动程序建立。目前常用的 JDBC 驱动程序可以分为四种类型：

（1）JDBC-ODBC 桥

这种驱动程序通过连接另一种数据库的 ODBC 来使用数据库，这称作 JDBC-ODBC 桥。ODBC（开放式数据库连接）是使用最广的、用于访问关系型数据库的连接接口，它能在几乎所有的平台上连接几乎所有的数据库。通过 JDBC-ODBC 桥，应用程序能够使用 ODBC 的驱动程序和数据库建立连接，并访问数据库。这种类型驱动程序的优点是适合所有的数据库，因为几乎所有的数据库开发商都提供了相应的 ODBC 驱动程序。其缺点是运行速度较慢，因为连接的层次太多。

（2）本地 API

和第一种类型相比，这种类型的驱动程序没有 ODBC 层。它的速度高于第一种类型的速度，但要求在本地机器上安装连接目标数据库的类库。这种方式使用不太方便，因此不太常用。

（3）网络协议

这种驱动程序在 JDBC 与数据库驱动程序之间加有一个称作网络协议的中间件，中间件把应用程序 JDBC 调用映射到相应的数据库驱动程序上。这种类型的驱动程序最灵活，因为在这种类型下，中间件可以和许多不同的数据库驱动程序建立连接，因此可以和许多不同的数据库建立连接。

（4）数据库协议

这种 JDBC 驱动程序通过实现一定的数据库协议直接和数据库建立连接。这种驱动程序的效率最高，因为它直接和数据库连接。其缺点是当目标数据库类型更换时，必须更换相应的驱动程序。

目前大多数的数据库厂商都提供有上述四种类型的驱动程序，设计人员可根据自己的机器环境、软件资源和问题要求选择不同的数据库驱动程序。

前两种类型的 JDBC 驱动程序使用不多，后两种类型的 JDBC 驱动程序使用较多，后两种类型的 JDBC 驱动程序也是未来 JDBC 驱动程序发展的主流技术。

本章后面的例子使用的数据库是 Microsoft 公司的 Access 数据库，该数据库只提供有 ODBC 驱动程序。因此，要建立应用程序和数据库的连接只能采用第一种类型的 JDBC 驱动程序，即 JDBC-ODBC 桥驱动程序。

11.2 建立应用程序和数据库连接的环境配置

要在应用程序中使用数据库，即要建立应用程序和数据库的连接，前提条件是各种资源要准备好，即进行环境配置工作。这些工作主要包括：

① 安装 JDBC API。在下载和安装 JDK 时，已经安装好了 JDBC API，因为 JDBC API 是由 Java API 中的一些类和接口组成的，它们在 java.sql 包中。

② 安装数据库。如果用户的计算机上还没有安装所需要的数据库，则必须按照厂商提供的说明书安装一个数据库。常用的数据库有 SQL Sever, Oracle 等。作为学习目的，Microsoft Access 数据库就可以了。

③ 创建数据库。

④ 安装数据库驱动程序。驱动程序的安装要根据用户使用的数据库，以及想要使用的 JDBC 驱动程序类型来进行。对于某些数据库专用的驱动程序，仅需要将这个驱动程序复制到计算机上，不需要做特别的配置。对于 JDBC-ODBC 桥驱动程序，需要查阅厂商提供的安装和配置说明书，根据说明书来进行安装和配置。

下面以 Microsoft Access 为例说明上述过程：

① 在安装 JDK 时已经安装好 JDBC API。

② 在安装 Windows 时选择同时安装 Microsoft Access。

③ 建立数据库目录和数据库。首先利用资源管理器建立数据库目录，如 c:\javaBook，然后运行 Microsoft Access，建立一个放在目录 c:\javaBook 下的数据库（本例数据库取名为“mydb”）。

④ 在 ODBC 中添加数据源。在 Windows 中控制面板的管理工具文件夹中，双击“数据源（ODBC）”，会出现一个弹出窗口，在该窗口中选择“系统 DNS”（DNS 就是 Data Source Name（数据源）的缩写），然后单击“添加”按钮，就会再弹出一个如图 11.1 所示的窗口。

此时就可以选择所要添加的数据源。本例选择的是 Microsoft Access 数据源。

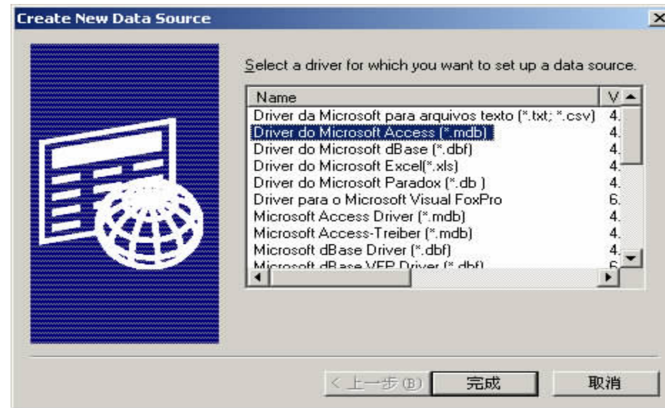


图 11.1　创建新的 DSN

选择完毕单击“完成”按钮，出现如图 11.2 所示的窗口。在该窗口中填写数据源的名字，这个名字可以任意取，本例取名为 mysource。然后单击“选择（S）”按钮，会再弹出一个如图 11.3 所示的窗口，用来选择目录和数据库名。本例选出第③步时建立的目录 c:\javaBook 下的数据库名 mydb.mdb。

图 11.2　选择数据源

图 11.3　选择目录和数据库

到此为止，建立应用程序和数据库连接的准备工作已经完成。下来就可以在应用程序中使用 JDBC 提供的类和接口建立应用程序和数据库的连接。一旦应用程序和数据库的连接建

立了，就可以在应用程序中操作数据库或数据库中的数据。

11.3 数据库应用编程

编写数据库应用程序一般包括如下几个步骤：

① 指定数据库驱动程序，并向驱动程序管理器注册驱动程序。这通常称作安装驱动程序。

② 建立与数据库的连接。

③ 对数据库中的表和表中的数据进行操作。

④ 返回操作结果。

⑤ 关闭与数据库的连接。

安装数据库的驱动程序很简单，只需给出驱动程序的名字，将它传递给 Class 类的 forName()方法就可以了。Class 类是负责系统管理的一个类，其中的 forName()方法是一个 static 方法，因此不用定义对象，直接用类名后跟方法名就可以了。例如，安装 JDBC-ODBC 驱动程序，可以使用如下语句：

```
Class.forName("sun.jdbc.odbc.JdbcOdcDriver" );
```

又例如，安装 Oracle JDBC 驱动程序，可以使用如下语句：

```
Class.forName("oracle.jdbc.driver.OracleDriver" );
```

一旦完成驱动程序的安装，就可以进行后面的工作。下面讨论怎样用 JDBC 中提供的类和接口来实现上述步骤②～步骤⑤的工作。JDBC API 中常用的类和接口都放在 java.sql 包中。其中，步骤②和步骤⑤的工作由 DriverManager 类来实现，步骤③的工作由 Connection 接口和 Statement 接口来实现，Statement 接口也能完成步骤④的部分工作，但步骤④的复杂工作需要用 ResultSet 接口来实现。DatabaseMetaData 接口用来获取数据库的元信息，ResultSetMetaData 接口用来获取表的元信息。所谓元信息是指关于某种信息的结构的信息。

要说明的是，所有下面讨论的 JDBC 中的接口都已实现。用户编程时可直接定义各接口的对象，并通过对象使用接口的方法。

11.3.1 建立连接

DriverManager 类和 Driver 接口主要用于实现编写数据库应用程序的步骤②和步骤⑤，即建立和数据库的连接和关闭连接，并可以获得数据库驱动程序的各种信息。

1．DriverManager 类

DriverManager 类负责管理 JDBC 驱动程序并建立与数据库的连接。用 Class.forName()语句完成驱动程序的加载和注册后，就可以用 DriverManager 类来建立 Java 程序和数据库的连接，并管理 JDBC 驱动程序。

DriverManager 类的主要方法如下（注意：这些方法都是 static 方法）：

⌘ Connection getConnection(String url)
建立和数据库的连接。其中 url 是要连接数据库的 URL

⌘ Connection getConnection(String url, String user, String password)
建立和数据库的连接。其中，url 是要连接数据库的 URL， user 是用户名，password

是用户密码

⌘ void doregisterDriver(Driver driver)　　注册一个驱动程序 driver

⌘ Driver getDriver(String url)　　返回指定 url 的驱动程序

⌘ void setLoginTimeout(int seconds)　　设置登录时等待的最长时间 seconds

⌘ int getLoginTimeout()　　返回登录时等待的最长时间

在这些方法中，最常用的是和数据库建立连接的方法 getConnection()。如果程序中要和一个数据库建立连接，可以使用如下语句：

```
DriverManager.getConnection(url, user, pwd);
```

其中，url 的格式为：jdbc:<subprotocol>:<subname>。

url 的格式中，subprotocol 是用来说明驱动程序类型的，subname 是数据源的名称。

例如，url = jdbc:odbc:mysource 语句中，jdbc:odbc 表明使用的是 jdbc:odbc 桥驱动程序类型，mysource 是要连接的数据源的名称。

下面通过一个例子说明如何在 Java 程序中建立和数据库的连接。

【例 11.1】　编写一个安装数据库驱动程序并连接数据库 mysource 的 Java 程序。

```
import java.sql.*;                                          //导入数据库编程所需的类和接口
public class DbConnection
{
    public static void main(String agrs[])
    {
        String driver = "sun.jdbc.odbc.JdbcOdbcDriver";     //指定驱动程序
        String connStr= "jdbc:odbc:mysource";               //指定 URL
        Connection con = null;                              //定义 Connection 对象
        try
        {
            Class cl = Class.forName(driver);               //安装数据库驱动程序
            System.out.println("加载成功的驱动程序名是: " + cl.toString());
            con = DriverManager.getConnection(connStr);
                                                            //建立和数据库的连接
            if(!con.isClosed())                             //判断数据库是否连接好
               System.out.println("数据库连接成功!");
            con.close();                                    //关闭和数据库的连接
        }
        catch(Exception e)
        {
            System.out.println("驱动程序加载失败: " + driver);
            e.printStackTrace();
        }
        catch(SQLException e)
        {
            System.out.println("数据库连接失败!");
```

```
            e.printStackTrace();
        }
    }
}
```

程序设计说明：

① 程序开始必须导入 java.sql 包。JDBC API 中所有的类和接口都在这个包中。

② 由于我们使用的是 JDBC-ODBC 桥类型的驱动程序，所以用语句：

```
Class cl =Class.forName("sun.jdbc.odbc.JdbcOdcDriver")
```

来加载 JDBC-ODBC 桥数据库驱动程序。

程序中上述语句分成了如下两句：

```
String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
Class cl = Class.forName(driver);
```

③ 本例使用的是前面创建的 Access 数据库，其数据库名为 mysource，因此用语句：

```
con = DriverManager.getConnection("jdbc:odbc:mysource");
```

来建立和数据库的连接。本例没有给出用户名和密码。

程序中上述语句分成了如下两句：

```
String connStr= "jdbc:odbc:mysource";
con = DriverManager.getConnection(connStr);
```

④ 语句 Connection con = null，定义了一个 Connection 对象引用 con，其初始值为空。语句 con = DriverManager.getConnection(connStr)后，con 就指向了所建立数据库的 Connection 对象。语句 con.close()调用 Connection 的方法 close()关闭了与数据库的连接。Connection 是一个接口，其定义和使用方法下面将详细介绍。

2．Driver 接口

Driver 接口是每个驱动程序类必须实现的。Java 的 SQL 框架允许有许多数据库驱动程序。每个驱动程序要提供一个实现 Driver 接口的类。

一旦用 Class 类的 forName()方法完成了驱动程序的加载和注册，就会产生一个 Driver 对象，通过这个对象可以获得数据库驱动程序的各种信息。

Driver 接口常用的方法有：

- ⌘ Connection connection(String url, Properties info)
  建立和数据库的连接。其中，url 是要连接数据库的 URL， info 可用来传递用户名、用户密码等属性
- ⌘ boolean acceptsURL(String url)　　检查驱动程序能否打开指定 url 的连接
- ⌘ boolean JdbcCompliant()　　检查驱动程序是否支持 JDBC
- ⌘ Properties getPropertyInfo(String url, Properties info)
  返回建立的数据库 url 的属性 info 的信息
- ⌘ int getMajorVersion()　　返回驱动程序的主版本号
- ⌘ int get MinorVersion()　　返回驱动程序的次版本号

11.3.2 操作数据库

Connection 接口和 Statement 接口负责管理数据库，向数据库发送 SQL 语句，以及返回执行结果。DatabaseMetalData 接口可以获取数据库的元信息。

1．Connection 接口

Connection 接口负责维护 Java 程序和数据库之间的连接,执行 SQL 语句,返回执行结果。

Connection 接口提供表的描述信息、对 SQL 语法的支持、对存储过程的支持，等等。对表的描述信息可由 getMetaData()方法获得。

Connection 接口由数据库制造商提供的驱动程序实现。接口一旦实现，就可以像类一样用来定义对象，所以，用户编程时可直接定义 Connection 对象。

Connection 接口的常用方法如下：

- ⌘ Statement createStatement()　　创建给数据库发送 SQL 语句的 Statement 对象
- ⌘ DatabaseMetalData getMetaData()　　获取所连接数据库的元信息
- ⌘ boolean setAutoCommit(boolean AutoCommit)
  设置 Connection 对象为 Atuocommit 状态，true 为自动提交，false 为非自动提交
- ⌘ void commit()　　提交数据库操作
- ⌘ void rollback()　　回滚数据库操作
- ⌘ void close()　　关闭数据库
- ⌘ boolean isClosed()　　测试数据库是否关闭

2．Statement 接口

Statement 接口的主要功能是将 SQL 命令传递给数据库，并返回数据库执行 SQL 命令的结果。

其常用方法如下：

- ⌘ ResultSet executeQuery()　　执行对数据库的查询操作，并返回查询结果
- ⌘ boolean execute(String sql)　　执行可返回多个结果的 SQL 语句 sql
- ⌘ boolean executeUpdate(String sql)
  执行 SQL 的 INSERT, UPDATE 或 DELETE 操作
- ⌘ addBatch(String sql)　　给当前的 Statement 对象添加一个批处理 SQL 语句 sql
- ⌘ int[] executeBatch()
  执行批处理命令，假如命令执行成功，返回数组为更新记录的个数
- ⌘ setQueryTimeout(int seconds)
  设置驱动程序最多等待 Statement 对象执行的时间为 seconds 秒

实际上，Statement 接口提供的方法包括了所有 SQL 语句对数据库的操作，作为教材，这里只给出了主要的几个。

【例 11.2】　建立和数据库的连接并对表进行操作示例。

具体要求：在数据库 mysource 中创建表 userInfo，并在表中插入四条记录。

程序设计如下：

```
import java.sql.*;
public class DbInsert
{
    public static void main(String agrs[])
    {
        String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
        String connStr= "jdbc:odbc:mysource";
        Connection con = null;
        Statement stm = null;
        try
        {
            Class.forName(driver);
            con = DriverManager.getConnection(connStr);
          if(!con.isClosed())
                System.out.println("数据库连接成功!");
            stm = con.createStatement();                    //创建 Statement 对象
            String sql ="insert into userInfo values
              ('王二', 'seeyou', 'smile@163.com', '2637', '5-6-83')";
            stm.executeUpdate(sql);
            stm.addBatch("insert into userInfo values
              ('李明', 'kevin', 'li@sina.com', '2536', '10-12-82')");
            stm.addBatch("insert into userInfo values
              ('王武', 'morning', 'wang@yahoo.com', '2939', '2-3-83')");
              stm.addBatch("insert into userInfo values
              ('张尹', 'well', 'zhay@hotmail.com', '2938', '10-12-83')");
              int [] updates = stm.executeBatch();
                                                    //执行批处理的 SQL 命令并返回更新个数
              System.out.println("表的更新个数为 "
                            +updates.length);
              System.out.println("插入数据成功!");
              con.close();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

程序说明：

① 语句

stm = con.createStatement();

创建了一个 Statement 对象并赋值给 Statement 引用 stm。

② 语句

```
String sql ="insert into userInfo values
    ('王二', 'seeyou', 'smile@163.com', '2637', '5-6-83')";
stm.executeUpdate(sql);
```

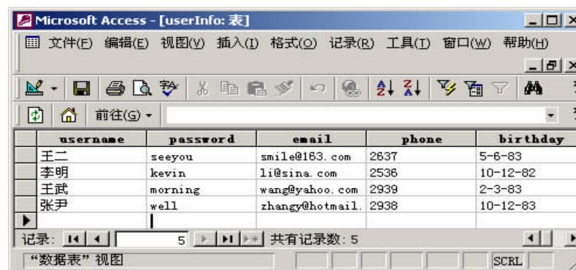
在数据库 mysource 中创建一个表 userInfo，并向表中插入记录('王二', 'seeyou', 'smile@163.com', '2637', '5-6-83')。

③ 语句

```
int [] updates = stm.executeBatch();
```

执行前面三个批处理语句 stm.addBatch()，添加三条记录，并返回更新记录个数。

用 Access 打开表 userInfo 可看到程序运行的结果。程序运行后表 userInfo 中的当前记录如图 11.4 所示。



username	password	email	phone	birthday
王二	seeyou	smile@163.com	2637	5-6-83
李明	kevin	li@sina.com	2536	10-12-82
王武	morning	wang@yahoo.com	2939	2-3-83
张尹	well	zhangy@hotmail.	2938	10-12-83

图 11.4　表 userInfo 中的当前记录

3．DatabaseMetaData 接口

调用 DatabaseMetaData 接口的方法可获得所连接数据库的大量元信息。

常用的方法有：

- ⌘ String getDatabaseProductName()　　返回数据库名称
- ⌘ String getDatabaseProductVersion()　　返回数据库版本号
- ⌘ String getDriveName()　　返回驱动程序名称
- ⌘ String getURL()　　返回连接的数据库
- ⌘ String getUserName()　　返回数据库的登录用户名
- ⌘ int getMaxRowSize()　　返回记录每行的最大长度
- ⌘ int getMaxStatementLength()　　返回数据库允许的 SQL 语句的最大长度

【例 11.3】　输出数据库 mysource 的版本号等信息。

程序设计如下：

```
import java.sql.*;
public class DbMeta
{
    public static void main(String agrs[])
    {
```

```
            String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
            String connStr= "jdbc:odbc:mysource";
            Connection con = null;
            Statement stm = null;
            try
            {
                Class.forName(driver);
                System.out.println("驱动程序加载成功: " + driver);
                con = DriverManager.getConnection(connStr);
                DatabaseMetaData dbmd = con.getMetaData();
                                        //获取所连接数据库的元信息并赋值给对象 dbmd
                System.out.println("JDBC URL: " + dbmd.getURL());
                System.out.println("JDBC 驱动程序: " + dbmd.getDriverName());
                System.out.println("JDBC 驱动程序的版本代号: "
                        + dbmd.getDriverVersion());
                System.out.println("用户名: " + dbmd.getUserName());
                System.out.println("数据库名称: "
                        + dbmd.getDatabaseProductName());
                System.out.println("数据库版本号: "
                        + dbmd.getDatabaseProductVersion());
                con.close();
            }
            catch(Exception e)
            {
                System.out.println("数据库连接失败!");
                e.printStackTrace();
            }
        }
    }
```

程序说明：在建立了和数据源的连接后，通过 Connection 对象 con 获得一个 DatabaseMetaData 对象 dbmd，此对象包含了程序连接的数据库的各种信息，通过调用这个对象的方法可获取数据库的属性信息。

程序运行结果如下：

```
驱动程序加载成功: sun.jdbc.odbc.JdbcOdbcDriver
JDBC URL: jdbc:odbc:mydatasource
JDBC 驱动程序: JDBC-ODBC Bridge (odbcjt32.dll)
JDBC 驱动程序的版本代号: 2.0001 (04.00.5303)
用户名: admin
数据库名称: ACCESS
数据库版本号: 04.00.0000
```

11.3.3 处理操作结果

ResultSet 接口用来处理执行 SQL 语句产生的查询结果表，ResultSetMetaData 接口用来获取表中字段的元信息。

1．ResultSet 接口

ResultSet 对象通常是由执行 SQL 语句产生的表。ResultSet 对象维护一个指向表中当前行的指针。指针的初始位置在第一行记录的前边，调用 next()方法可以把指针移动到下一行。当不存在下一行时 next()方法返回 false。因此，next()方法可以作为程序中 while 循环语句的条件，这样就可以操作 ResultSet 对象中的所有记录。

一般的 ResultSet 对象是不可以更改的，而且指针只能向后移动。这样，指针只能从第一行向后移动到最后一行，而且只能这样移动一次。但是，通过调用 JDBC 2.0 中的方法可以创建一个多次前后移动指针的 ResultSet 对象。以下语句可以创建这样的一个 ResultSet 对象：

```
    Statement stm = con.createStatement
        (ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_UPDATABLE);
    ResultSet rs = stm.executeQuery("SELECT a, b FROM TABLE2");
```

其中，TYPE_SCROLL_INSENSITIVE 和 CONCUR_UPDATABLE 是 ResultSet 接口定义的 static 类型的成员变量。TYPE_SCROLL_INSENSITIVE 表示记录可滚动、但对其他语句引起的修改不敏感的 ResultSet 对象类型。CONCUR_UPDATABLE 表示可能被更新的 ResultSet 对象的并发模式。"SELECT a, b FROM TABLE2"是查询语句字符串。

ResultSet 接口中对指针操作的方法都是针对当前 ResultSet 对象的。

ResultSet 接口的常用方法有：

- ⌘ boolean absolute(int row)　　将指针移动到 row 行
- ⌘ void beforeFirst()　　将指针移动到第一行前面
- ⌘ void afterLast()　　将指针移动到最后一行之后
- ⌘ boolean first()　　将指针移动到第一行
- ⌘ boolean last()　　将指针移动到最后一行
- ⌘ boolean next()　　将指针移动到下一行
- ⌘ boolean previous()　　将指针移动到前一行
- ⌘ void deleteRow()　　删除当前行记录
- ⌘ void moveToInsertRow()　　将指针移动到插入行
- ⌘ void moveToCurrentRow()　　将指针移动到当前行
- ⌘ void insertRow()　　把插入行的记录插入到 ResultSet 对象和数据库中
- ⌘ void updateRow()　　用 ResultSet 对象当前行的记录更新数据库
- ⌘ ResultSetMetaData getMetData()　　获取当前 ResultSet 对象字段的个数、类型和属性
- ⌘ String getString(int columnIndex)
  获取 ResultSet 对象当前行的列 columnIndex 的数值，返回值为 String 类型

【例 11.4】　选择数据库 mysource 中表 userInfo 的符合条件的记录。

程序设计如下：

```
import java.sql.*;
import java.io.*;
public class DbSearch
{
    public static void main(String agrs[])
    {
        String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
        String connStr = " jdbc:odbc:mysource ";
        Connection con = null;
        Statement stm = null;
        InputStreamReader sin = new InputStreamReader(System.in);
        BufferedReader bin = new BufferedReader(sin);
        try
        {
            Class.forName(driver);
            con = DriverManager.getConnection(connStr);
            stm = con.createStatement
                (ResultSet.TYPE_SCROLL_SENSITIVE,
                ResultSet.CONCUR_READ_ONLY);
            ResultSet    rst = stm.executeQuery
                ("SELECT * from userInfo");
            System.out.println("表中所有记录:");
            while(rst.next())                //移动指针到下一条记录并判是否为 true
            {
                System.out.println(rst.getString(1)
                    + " " + rst.getString(2) + " "+ rst.getString(3)
                    +" " + rst.getString(4)+" " + rst.getString(5));
            }
            System.out.println("请输入要查询的客户名称:");
            String un= bin.readLine().trim();
            ResultSet    rst1 =
                    stm.executeQuery("SELECT * from userInfo");
            while(rst1.next())               //移动指针到下一条记录并判是否为 true
            {
                if (rst1.getString(1).trim().equals(un))
                {
                    System.out.println("你要查询的客户信息：");
                    System.out.println(rst1.getString(2) + " "
                    + rst1.getString(3)+" " + rst1.getString(4)+" "
                    + rst1.getString(5));
```

```
                        break;
                    }
                }
                con.close();
            }
            catch(Exception e)
            {
                System.out.println("数据库连接失败!");
                e.printStackTrace();
            }
        }
    }
```

程序设计说明：此程序非常简单，关键是要通过 rst=smt.executeQuery ()语句，获得一个 ResultSet 对象 rst。rst 对象中包含一个执行查询语句获得的结果集，并且指针可移动。程序中，第一个 while 循环通过移动指针，将 userInfo 表中的所有记录的字段信息打印出来；第二个 while 循环通过移动指针，将 userInfo 表中查询到的记录的字段信息打印出来。

程序运行结果如下：

```
表中所有记录:
王二  seeyou smile@163.com 2637 5-6-83
李明  kevin li@sina.com 2536 10-12-82
王武  morning wang@yahoo.com 2939 2-3-83
张尹  well zhangy@hotmail.com 2938 10-12-83
请输入要查询的客户名称:
王武
你要查询的客户信息：
Password: morning
Email: wang@yahoo.com
Phone: 2939
Birthday: 2-3-83
```

2．ResultSetMetaData 接口

ResultSetMetaData 对象可用于获取 ResultSet 对象字段的类型和属性等信息。

常用的方法有：

- ⌘ int getColumnCount()　　返回字段的个数
- ⌘ int getColumnDisplaySize(Column)　　返回字段的长度
- ⌘ String getColumnName(int Column)　　返回字段的名称
- ⌘ String getColumnTypeName(int Column)　　返回字段的类型名称
- ⌘ String getTableName(int Column)　　返回字段的所属表的名称
- ⌘ boolean isCaseSenstive(int Column)　　检查字段是否区分大小写
- ⌘ boolean isReadOnly(int Column)　　检查字段是否为只读

【例 11.5】 输出数据库 mysource 中表 userInfo 的每一个字段的属性信息。

程序设计如下：

```
import java.sql.*;
public class DbMetaData
{
    public static void main(String agrs[])
    {
        String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
        String connStr = "jdbc:odbc:mysource";
        Connection con = null;
        try
        {
            Class.forName(driver);
            con = DriverManager.getConnection(connStr);
            Statement   smt = con.createStatement();
            ResultSet   rst = smt.executeQuery("select *
                              from userInfo");
            ResultSetMetaData rsmd = rst.getMetaData();
            System.out.println("表名称: " +
                              rsmd.getTableName(1) + "\n");
            System.out.println("表字段个数: " +
                              rsmd.getColumnCount() + "\n");
            System.out.println("表字段:");
            for(int i=1;i<=rsmd.getColumnCount();i++)
            {
                System.out.println("****************************");
                System.out.println("字段" + i + "的名称: " +
                              rsmd.getColumnName(i));
                System.out.println("字段类型: " +
                              rsmd.getColumnTypeName(i));
                System.out.println("字段长度: " +
                              rsmd.getColumnDisplaySize(i));
                System.out.print("是否为只读: ");
                if(rsmd.isReadOnly(i))
                    System.out.println("是");
                else
                    System.out.println("否");
            }
            smt.close();
            con.close();
        }
        catch(Exception SE)
```

```
            {
                System.out.println("打开数据库连接失败!");
                    SE.printStackTrace();
            }
        }
    }
```

程序运行结果如下：

```
表名称: userInfo
表字段个数: 5
表字段:
*****************************
字段 1 的名称: username
字段类型: VARCHAR
字段长度: 50
是否为只读: 否
*****************************
字段 2 的名称: password
字段类型: VARCHAR
字段长度: 50
是否为只读: 否
*****************************
字段 3 的名称: email
字段类型: VARCHAR
字段长度: 50
是否为只读: 否
*****************************
字段 4 的名称: phone
字段类型: VARCHAR
字段长度: 50
是否为只读: 否
*****************************
字段 5 的名称: birthday
字段类型: VARCHAR
字段长度: 50
是否为只读: 否
```

习题 11

一、基本概念题

11.1　**什么是 SQL？SQL 的主要用途是什么？SQL 和应用程序的关系是什么？**

11.2 JDBC 的驱动程序有几种类型？它们各有什么特点？

11.3 在使用 JDBC 的应用程序中，对数据库的操作主要有哪几个步骤？

11.4 在应用程序中使用数据库之前需要进行环境配置工作，这些工作主要包括哪些内容？

11.5 Class.forName()语句在数据库编程中的作用是什么？

11.6 ResultSetMetaData 接口和 DatabaseMetaData 接口各自的功能是什么？

11.7 已经设计实现的接口是否可用来定义对象？JDBC 中的接口是否可用来定义对象？

二、程序设计题

11.8 在计算机上安装合适的数据库软件，创建一个数据库，并完成相应的环境配置。

11.9 编写一个数据库的创建表程序。程序将根据用户的要求在数据库中创建新的表。

11.10 编写一个录入程序。将用户输入的数据提交并插入到数据库的表中。

11.11 编写一个查询程序。可以根据用户的要求查询表中的记录信息。

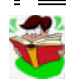
11.12 编写一个 JDBC 应用程序，在屏幕上输出数据库的属性信息和指定表的属性信息。

11.13 重新设计例 11.3，用图形界面显示输出结果。

11.14 重新设计例 11.4，用图形界面显示输出结果。

11.15 重新设计例 11.5，用图形界面显示输出结果。

第 12 章　网 络 通 信



教学要点

本章内容主要包括网络通信的基本概念（包括 IP 地址、端口、TCP 协议、UDP 协议），使用 URL 类访问网络资源的程序设计方法，基于 Socket 的连接通信程序设计方法，数据报通信。

要求掌握网络通信的基本概念、数据报通信，熟练掌握使用 URL 类访问网络资源的程序设计方法、基于 Socket 的连接通信程序设计方法。

Java 是伴随网络发展成长起来的语言。Java 与网络有密切的关系。在 Java 中有许多与网络通信有关的类和接口，利用它们可以方便地实现网上资源的定位和获取，以及实现计算机之间的通信和文件的传输等。本章介绍网络通信的基本概念和 Java 中常用的与网络通信有关的类和接口，并用实例说明这些类和接口是如何实现网络通信的。另外，本章还介绍了 Client/Server 结构的基本概念，编写的例子也是基于 Client/Server 结构的。

12.1　网络通信的基本概念

计算机网络是指通过各种通信设备连接起来的、支持特定网络通信协议的、许许多多的计算机或计算机系统集合。

网络通信是指网络中的计算机通过网络互相传递信息。

通信协议是网络通信的基础。**通信协议**是网络中计算机之间进行通信时共同遵守的规则。不同的通信协议用不同的方法解决不同类型的通信问题。常用的通信协议有 HTTP, FTP, TCP/IP 等。其中，HTTP 是互联网上常用的通信协议，FTP 是一种用于网络中文件传输的通信协议，TCP/IP 协议是目前广泛使用的一种网络协议。在 TCP/IP 协议中，IP 协议主要负责网络主机的定位、数据传输的路由。由 IP 地址可以惟一地确定 Internet 上的一台主机。TCP 协议则提供面向应用的可靠的或非可靠的数据传输机制。

目前较为流行的网络通信模型是客户机/服务器（Client/Server，缩写为 C/S）结构。客户机端程序在需要服务时向服务器提出服务申请；服务器端程序则等待客户提出服务请求，并予以响应。服务器端程序始终运行，监听网络端口，一旦有客户请求，就会启动一个服务进程来响应该客户机端的请求，同时继续监听网络端口，准备为其他客户请求提供服务。

网络上的计算机要互相通信，必须首先做到：

① 能够准确定位网络上的任意一个通信主体；

② 有一套通信规则保证网络上可靠高效的数据传输。

12.1.1　通信主体的定位

通信主体有两级：IP 地址和端口号。IP 地址可定位网络上的任意一台主机。端口号可定

位主机的进程。

1．IP 地址

IP 地址是计算机网络中任意一台计算机地址的惟一标识。知道了网络中某一台计算机的 IP 地址，就可以定位这台计算机。通过这种地址标识，网络中的计算机可以互相定位和通信。目前，IP 地址由四个 8 位的二进制数组成，中间以小数点分隔。如 127.129.121.3，又如 165.103.32.30。

主机名是计算机网络中一台计算机的标识名，也可以看作是 IP 地址的助记名。如 www.sina.com.cn、www.china.com 等。在 Internet 上，IP 地址和主机名是一一对应的。由于机器名更接近自然语言，容易记忆，所以使用起来更方便。但是对机器而言，只有 IP 地址才是有效的标识名。通过主机名到 IP 地址的解析，可以由主机名得到对应的 IP 地址。在访问网上资源时，一般只需记住服务器的主机名就可以了。因为网络中的域名解析服务器可以根据主机名查出对应的 IP 地址。有了服务器的 IP 地址，就可以访问这个网站了。例如，要用浏览器访问新浪网，可在地址栏中输入 http://www.sina.com.cn，域名解析服务器就会自动找到它对应的 IP 地址，然后定位网上的这台服务器并向它发出服务请求。服务器接到该服务请求后，运行相应的程序，将一个 HTML 文件返回发出请求的浏览器，浏览器显示结果如图 12.1 所示。

图 12.1　浏览器访问新浪网

2．端口号

一台主机上允许有多个进程，这些进程都可以和网络上的其他计算机进行通信。更准确地说，网络通信的主体不是主机，而是主机中运行的进程。这时候光有主机名或 IP 地址显然是不够的。因为一个主机名或 IP 地址对应的主机可以拥有多个进程。端口就是为了在一台主机上标识多个进程采取的一种手段。主机名（或 IP 地址）和端口的组合能惟一确定网络通信的主体——进程。**端口**（port）是网络通信时同一主机上的不同进程的标识。**端口号**（port number）是端口的数字编号，例如 80。一个服务器可以通过不同端口提供许多不同的服务。图 12.2 表示服务器的一个端口和客户机连接。

图 12.2　服务器的一个端口和客户机连接

12.l.2 TCP 协议和 UDP 协议

TCP 协议和 UDP 协议是网络通信时使用的两种主要协议。

1．TCP 协议

TCP（Transfer Control Protocol 的简称）协议是一种面向连接的、可以提供可靠传输的协议。使用 TCP 协议传输数据，接收端得到的是一个和发送端发出的完全一样的数据流（包括顺序）。发送方和接收方之间的两个端口必须建立连接，以便在 TCP 协议的基础上进行通信。在程序中，端口之间建立连接一般使用的是 Socket（套接字）方法。当服务器的 Socket 等待服务请求（即等待建立连接）时，客户机的 Socket 可以要求进行连接，一旦这两个 Socket 连接起来，它们就可以进行双向数据传输，即双方都可以发送或接收数据。这种通信方式和电信局的电话系统很相似。两者的操作步骤都是：首先建立一个连接，然后开始传输数据，传输的数据顺序和接收的数据顺序是完全一样的，最后，断开连接。TCP 协议为实现可靠的数据传输提供了一个点对点的通道。

2．UDP 协议

UDP（User Datagram Protocol 的简称）是一种无连接的协议，它传输的是一种独立的数据报（Datagram）。每个数据报都是一个独立的信息，包括完整的源地址或目的地址。数据报在网络上以任何可能的路径传往目的地，因此，数据报能否到达目的地、到达目的地的时间、数据的正确性和各个数据报到达的顺序都是不能完全保证的。这种通信方式和邮局的信件传送方式很相似。发信人只要将若干信件放入信箱就可以了，至于信件传递的路径、时间、到达的顺序、是否一定能到达等，邮局会尽可能做好，但不做完全保证。

3．两种协议的比较

① 使用 UDP 协议时，每个数据报中都给出了完整的地址信息，因此无需建立发送方和接收方的连接。使用 TCP 协议时，由于它是一个面向连接的协议，在 Socket 之间进行数据传输之前必然要建立连接。

② 使用 UDP 协议传输数据是有大小限制的，每个被传输的数据报必须限定在 64KB 之内。而 TCP 协议没有这方面的限制，一旦连接建立起来，双方的 Socket 就可以按统一的格式传输大量的数据。

③ UDP 协议是一个不可靠的协议，发送方所发送的数据报并不一定以相同的次序到达接收方，还有可能会丢失。而 TCP 是一个可靠的协议，它确保接收方完全正确地获取发送方所发送的全部数据。

12.1.3 网络程序设计的基本方式

Java 用于网络通信的包是 java.net，它包含了多个用于各种标准网络协议通信的类和接口。Java 网络程序设计的基本方式主要有三种：

1．使用 URL 访问网络资源

URL 表示了 Internet 上某个资源的地址。URL 支持 http，file，ftp 等多种协议。Java 通过

URL 标识，可以直接使用 http, file, ftp 等多种协议，获取远端计算机上的资源信息，方便快捷地开发 Internet 应用程序。

2．连接通信

连接通信主要使用 Socket。Socket 是 TCP/IP 协议中的传输层接口。连接通信是使用 TCP/IP 协议、建立在稳定连接基础上的、以流传输数据的通信方式。它是目前实现 Client/Server 模式应用程序的主要方式。常见的聊天程序等就是连接通信的应用。

3．数据报通信

数据报是一种在网络上传播的、独立的、自包含地址信息的格式化信息。数据报通信使用 UDP 协议。数据报通信不需要建立连接，通信时所传输的数据报能否到达目的地、到达的时间、到达的次序都不能准确知道。数据报通信主要用于传输一些数据量大的、非关键性的数据。

下面我们分别讨论这三种网络通信方式。

12.2 使用 URL 类访问网络资源

URL 用来表示 Internet 上资源的位置。通过 URL 标识，可以直接利用 http, file, ftp 等多种网络协议来获取远端计算机上的资源或信息，从而方便快捷地开发出 Internet 应用程序。

12.2.1 资源定位器 URL 和 URL 类

URL 类是 Java 语言提供的支持 URL 编程的基础类。要掌握 URL 类，首先需要了解 URL 的基本概念。

1．资源定位器 URL

URL（Uniform Resource Locator）是统一资源定位器的简称，用来表示 Internet 上资源的地址。知道了网络上任意一个资源的 URL，使用浏览器就可以访问它。浏览器通过解析给定的 URL，可以在网络上查找到相应的文件或其他类型资源。

URL 的语法格式是：

<传输协议名>:// <主机名>:<端口号>/<文件名>#<引用>

其中，<传输协议名>指明获取资源所使用的传输协议，如 http, ftp, gopher, file 等；<主机名>是网络中的计算机名或 IP 地址；<端口号>是计算机中代表一个服务的进程的编号；<文件名>是服务器上的文件的包括路径的名称；<引用>是文件中的标记。

在上述格式中，<端口号>、<文件名>和<引用>是可选的。<传输协议名>和<主机名>是必需的。当没有给出<传输协议名>时，浏览器默认的传输协议是 http。

下面都是合法的 URL：

http://www.sun.com	<传输协议名>：// <主机名>
http://172.17.99.3	<传输协议名>：// <IP 地址>
http://localhost:80	<传输协议名>：// <主机名>：<端口号>

http://home.netscape.com/home/welcome.html　　<传输协议名>：// <主机名>/<文件名>

http://www.china.com/index.html#a　　<传输协议名>：// <主机名>/<文件名>#<引用>

2．URL 类

为了表示 URL，java.net 包中定义了 URL 类。URL 类表示一个统一资源定位器，它是指向互联网上某一资源的指针，这个资源可以是某个主机的一个文件或路径，也可以是文件上的一个锚（或称引用）。

（1）构造方法

⌘ URL (String spec)

⌘ URL(URL context, String spec)

⌘ URL(String protocol, String host, String file)

⌘ URL(String protocol, String host, int port, String file)

其中，protocol 是协议名，host 是主机名，port 是端口号，file 是文件名，context 是 URL 对象，spec 是引用名。

例如，下面的每个语句分别创建并初始化了一个 URL 对象：

```
URL u1=new URL("http://www. 263.net/");
URL u2=new URL(u1, "index.html");
URL u3=new URL("http", "www.gamelan.com", "/pages/Gamelan.net. html");
URL u4=new URL("http", "www.gamelan.com", 80,
"Pages/Gamelan.network.html");
```

如果构造方法的参数不合法，就会抛出 MalformedURLException 异常，因此需要运用下面的语句格式来捕捉和处理异常。

```
try
{
    URL myURL = new URL(……)                    //创建 URL 对象
}
catch (MalformedURLException e)                //捕捉异常
{
    异常处理
}
```

如果程序中不捕捉异常，也可以在 main()方法中用类似下面的语句直接将异常抛出：

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

（2）常用方法

⌘ String getProtocol()　　返回当前 URL 的协议名

⌘ String getHost()　　返回当前 URL 的主机名

⌘ int getPort()　　返回当前 URL 的端口号

⌘ String getFile()　　返回当前 URL 的文件名

⌘ String getQuery()　　返回当前 URL 的查询

⌘ String getPath()　　返回当前 URL 的路径

⌘ String getAuthority()　　返回当前 URL 的权限

⌘ String getUserInfo()　　返回当前 URL 的用户信息

⌘ String getRef()　　返回当前 URL 的锚（或称引用）

⌘ InputStream openStream()

打开当前 URL 的连接，返回从这个连接读取的输入流

⌘ URLConnection openConnection()

返回一个由 URL 指示的、表示与远程对象连接的 URLConnection 对象

【例 12.1】 创建两个参数分别为"http://java.sun.com:80/docs/books/"和"http://java.sun.com:80/docs/books/tutorial.intro.html#DOWNLOADING"的 URL 对象，并输出第二个对象的信息。

程序设计如下：

```
import java.net.*;
public class TestURL
{
    public static void main (String [] args) throws Exception
    {
        URL u1=new URL("http://java.sun.com:80/docs/books/");
        URL u2=new URL(u1,"tutorial.intro.html#DOWNLOADING");
        System.out.println("protocol="+ u2.getProtocol());          //获取 URL 的协议名
        System.out.println("host ="+ u2.getHost());                 //获取 URL 的主机名
        System.out.println("filename="+ u2.getFile());              //获取 URL 的文件名
        System.out.println("port="+ u2.getPort());                  //获取 URL 的端口号
        System.out.println("ref="+u2.getRef());                     //获取 URL 的锚
        System.out.println("query="+u2.getQuery());                 //获取 URL 的查询
        System.out.println("path="+u2.getPath());                   //获取 URL 的路径
        System.out.println("UserInfo="+u2.getUserInfo());           //获得 URL 的用户信息
        System.out.println("Authority="+u2.getAuthority());         //获取 URL 的权限
    }
}
```

程序设计说明：程序中把第二个 URL 对象的参数"http://java.sun.com:80/docs/books/tutorial.intro.html#DOWNLOADING"分解成了两部分："http://java.sun.com:80/docs/books/"和"tutorial.intro.html#DOWNLOADING"，其中前一个参数和 u1 对象的相同，所以用 u1 表示。参数#DOWNLOADING 是 HTML 文件"tutorial.intro.html"中的一个锚。

程序运行结果如下：

```
protocol=http
host =java.sun.com
filename=/docs/books/tutorial.intro.html
port=80
ref=DOWNLOADING
query=null
path=/docs/books/tutorial.intro.html
```

```
UserInfo=null
Authority=java.sun.com:80
```

【例 12.2】 创建一个参数为“http://www.sina.com.cn/index.html”的 URL 对象，然后读取这个对象的文件。

程序设计如下：

```
import java.net.*;
import java.io.*;
public class TestURLReader
{
    public static void main(String[] args) throws Exception
    {
        URL u1 = new URL("http://www.sina.com.cn/index.html");    //创建 URL 对象
        BufferedReader in = new BufferedReader
        (new InputStreamReader(u1.openStream()));                 //创建 BufferedReader 对象
        String inputLine;
        while ((inputLine = in.readLine()) != null)               //从输入流循环读取数据
        System.out.println(inputLine);                            //把读取的数据输出到屏幕上
        in.close();                                               //关闭字符输入流
    }
}
```

程序说明：openSteam()方法打开当前 URL 的连接，InputStream 对象 in 可以读取指定资源的文件。

注意：本地计算机必须和互联网连接好以后才能运行上述网络通信程序。

程序的运行结果省略，要求读者在自己的计算机上运行这个程序，并观察屏幕输出结果。

12.2.2 URLConnection 类和 InetAddress 类

URLConnection 类支持 URL 连接的输入/输出流方式的通信，并可以获得 URL 对象资源的相关信息。InetAddress 类表示了一个 IP 地址，应用程序可调用该类的方法来创建一个新的 InetAddress 对象。

1．URLConnection 类

URLConnection 类是所有表示应用程序和 URL 连接通信的类的父类。该类的对象可以用来输出/输入（或称读/写）URL 对象所表示的 Internet 上的数据。应用程序和 URL 要建立一个连接通常需要几个步骤，但首先要做的，是创建一个由 URL 指示的、表示与远程对象连接的 URLConnection 对象。

URLConnection 类的成员变量：

- ⌘ connected　　如果该值为假，表示和指定的 URL 对象的连接还没建立好
- ⌘ url　　连接要打开的 Internet 上的 URL 形式表示的远程对象
- ⌘ useCaches　　如果该值为真，表示协议允许使用高速缓冲存储器

URLConnection 类的构造方法：

⌘ URLConnection(URL url)　　创建参数为 url 的 URLConnection 对象

URLConnection 类的常用方法：

⌘ void setUseCaches(boolean usecaches)　　设置成员变量 useCachesd 的逻辑值
⌘ object getContent()　　获取当前 URL 连接的信息内容
⌘ String getContentType()　　返回连接类型的头域值
⌘ int getContentLength()　　返回连接长度的头域值
⌘ long getDate()　　返回日期头域值
⌘ long getLastModified()　　返回最后修改时间头域值
⌘ InputStream getInputSTream()　　返回从所打开连接读数据的输入流
⌘ OutputStream getOutputSTream()　　返回向所打开连接写数据的输出流

【例 12.3】 创建参数为“http://www.sina.com.cn/index.html”的 URLConnection 对象，在本地机屏幕上输出该对象的数据。

程序设计如下：

```
import java.io.*;
import java.net.*;
public class TestURLConn
{
    public static void main(String[] args) throws Exception
    {
        int c;
        URL u1 = new URL("http://www.sina.com.cn/index.html");          //创建 URL 对象
        URLConnection uc = u1.openConnection();                          //创建 URLConnection 对象
        System.out.println(" 文件类型："+uc.getContentType());
        System.out.println(" 文件长度："+uc.getContentLength());
        System.out.println(" 文件内容如下：");
        System.out.println("******************************");
        InputStream in = uc.getInputStream();        //定义输入流引用并使其指向当前连接的输入流
        while ((c=in.read())!=-1)                                        //循环直到文件结束
        System.out.print((char)c);                                       //显示输入流读取的字符
        in.close();                                                      //关闭输入流
    }
}
```

程序的运行结果省略，要求读者在自己的计算机上运行这个程序，并观察屏幕输出结果。

2．InetAddress 类

InetAddress 类表示了一个 IP 地址，应用程序可调用该类的 getLocalHost()、getByName()或者 getAllByName()方法来创建一个新的 InetAddress 对象。

InetAddress 类的常用方法有：

⌘ static String getLocalHost()　　返回本地主机名
⌘ static String getByName (String host)　　根据主机名 host 确定主机的 IP 地址
⌘ static String[] getAllByName ()　　根据主机名 host 确定主机的所有 IP 地址
⌘ String getHostName()　　返回当前 IP 地址的主机名
⌘ String getHostAddress()　　返回 IP 地址字符串
⌘ byte[] getAddress()　　返回当前对象的 IP 地址的字节数组
⌘ String toString()　　转换 IP 地址为字符串

InetAddress 类在网路编程中非常有用，使用 InetAddress 类的对象，可以很容易地获得网上资源的各种信息。

【例 12.4】　分别创建本地机的 InetAddress 对象和“java.sun.com”的 InetAddress 对象，并输出它们的 IP 地址。

程序设计如下：

```
import java.net.*;
public class TestInet
{
    public static void main(String[] args) throws Exception
    {
        if (args.length!=1)                          //如果命令行没有给出主机名则退出
        {
            System.exit(1);
        }
        InetAddress adr=InetAddress.getByName(args[0]);
                                                  //创建 InetAddress 类对象，其参数来自命令行参数
        System.out.println(adr);
        InetAddress[] ips =InetAddress.getAllByName("java.sun.com");
                                                  //通过指定字符串获得 InetAddress 对象数组
        for (int i=0;i<ips.length; i++)
        System.out.println(ips[i]);                  //打印每一个 InetAddress 对象代表的 IP 地址
    }
}
```

程序说明：创建本地机 InetAddress 对象的参数由命令行参数给出，另一个 InetAddress 对象的参数（"java.sun.com"）直接给出。

运行程序的命令行如下：

```
C:\javaBook\ch12>java TestInet localhost
```

程序运行结果如下：

```
localhost/127.0.0.1
java.sun.com/209.249.116.142
java.sun.com/209.249.116.143
java.sun.com/128.11.159.83
java.sun.com/209.249.116.141
```

12.3 连接通信

在客户机/服务器应用模式中，要求服务器可以提供各种服务，如处理数据库查询请求、为客户提供股票行情信息，等等。这些服务要求客户机和服务器之间的通信必须是可靠的，并要求互相之间传送的数据内容和数据顺序是不会改变的。TCP 协议为互联网上客户机/服务器应用程序提供了一个点对点的通道。运用 TCP 协议通信时，客户机和服务器之间首先需要建立一个连接，然后客户机端和服务器端程序各自将一个 Socket 对象和这个连接绑定，这样，两端的程序就可以通过和连接绑定的 Socket 对象来读写数据了。

12.3.1 Socket 和连接

在 Java 网络编程中，Socket 是一个用于端点连接和数据交换的对象。网络的每一个端点，都可以通过和连接绑定的 Socket 对象来交换数据。在网络中，Socket 可以由 IP 地址和端口号惟一确定。

在 Client/Server 模式下，按照 Socket 在网络中所起的作用不同，可以将它们分为两类：客户机端 Socket 和服务器端 Socket。

两类 Socket 的工作过程如下：

① 服务器端。服务器端的 Socket 始终在监听是否有连接请求，如果发现客户机端 Socket 向服务器发出连接请求，且服务器接受服务请求，则服务器端 Socket 向客户机端 Socket 发回“接受”的消息。这样，两个 Socket 对象之间的连接就建立了。

② 在客户机端。建立一个和服务器的连接。这需要客户机端知道服务器的主机名和提供服务的端口号。有了这些信息，当客户机端发出的建立连接的请求被服务器端接受时，客户机上就会创建一个 Socket 对象。利用这个 Socket 对象，客户机就可以和服务器进行通信了。

上述服务器端和客户机端的工作过程可以归纳如下：在 Client/Server 模式的网络应用中，客户机端 Socket 主要用于向服务器发出连接请求和交换数据。服务器端的 Socket 始终在监听是否有连接请求，如果发现客户机端 Socket 向服务器发出连接请求，且服务器接受服务请求，则服务器端 Socket 向客户机端 Socket 发回“接受”的消息。这时，一个连接就建立起来了。随后，服务器端 Socket 和客户机端 Socket 就可以通过这个连接进行数据的传送。这样的工作过程包含以下四个基本步骤：

① 创建 Socket 对象；

② 打开连接到 Socket 对象的输入/输出流；

③ 按照一定的协议对 Socket 对象进行读/写操作；

④ 关闭 Socket 对象（即关闭 Socket 对象绑定的连接）。

12.3.2 Socket 类和 ServerSocket 类

Java 语言的 java.net 包中提供了两个类——Socket 类和 ServerSocket 类，分别用来表示双向连接的客户机端 Socket 和服务器端 Socket。在程序中，只需要在创建了 Socket 对象和 ServerSocket 对象后调用相应的方法，就可以实现网络通信了。

1．Socket 类

Socket 类实现了客户机端的 Socket。Socket 对象可以用来向服务器发出连接请求，并交

换数据。

（1）Socket 类的构造方法

⌘ Socket(InetAddress address, int port);

⌘ Socket(String host, int prot);

⌘ Socket(String host, int port, InetAddress localAddr, int localPort)

⌘ Socket(InetAddress address, int port, InetAddress localAddr, int localPort)

其中，address, host 和 port 分别是主机的 IP 地址、主机名和端口号；localAddr 和 localPort 分别表示本地机的 IP 地址和端口号；count 表示服务器端所能支持的最大连接数。

例如，以下语句：

```
Socket client = new Socket("127.0.0.1", 1029);
```

创建了一个 Socket 对象并赋了初值，要连接的远程主机的 IP 地址是 127.0.0.1，端口号是 1029。

需要说明的是：

① 每一个端口提供一种特定的服务，只有给出正确的端口，才能获得相应的服务。为此，系统特意为一些服务保留了一些端口。例如，http 服务的端口号为 80，telnet 服务的端口号为 21，ftp 服务的端口号为 23，0～1023 都是系统预留端口，所以在应用程序中设置自己的端口号时，最好选择一个大于 1023 的数字以防止发生端口冲突。

② 在创建 Socket 时可能会出现异常。可以在程序中捕捉异常并处理异常。

（2）Socket 类的常用方法

⌘ InetAddress getLocalAddress()	获取 Socket 对象绑定的本地机地址
⌘ int getLocalPort()	返回 Socket 对象绑定的本地机端口号
⌘ InetAddress getAddress()	获取 Socket 对象绑定的远程地址
⌘ int getPort()	返回 Socket 对象绑定的远程端口号
⌘ OutputStream getOutputSTream()	返回 Socket 对象的输出流
⌘ InputStream getInputSTream()	返回 Socket 对象的输入流
⌘ void setSoTimeout(int timeout)	设置操作允许的毫秒时间 timeout

2．ServerSocket 类

ServerSocket 类实现了服务器端 Socket。ServerSocket 对象监听网络中来自客户机的服务请求，根据请求建立连接，并根据服务请求运行相应的服务程序。

（1）构造方法

```
ServerSocket(int port)
ServerSocket(int port, int count)
```

其中，port 是服务器的端口号；count 是服务器所能支持的最大连接数。

例如，下面语句：

```
ServerSocket server=new ServerSocket(4700);
```

创建了一个 ServerSocket 对象 server 并赋了初值，其端口号是 4700。

（2）常用方法

⌘ InetAddress getInetAddress()　　返回 Socket 的本地地址

⌘ int getLocalPort()　　返回 Socket 正监听的端口号
⌘ Socket accept()　　监听当前 Socket 的连接并接收请求
⌘ void close()　　关闭连接
⌘ void setSoTimeout(int timeout)　　设置连接允许的毫秒时间 timeout

要说明的是：在创建客户端 Socket 和服务器端 Socket，以及在调用 Socket 类和 ServerSocket 类的某些方法时，可能出现异常。一般情况下，应用程序应处理这些异常。

12.3.3 Client/Server 结构的通信实例

【例 12.5】　设计一个基于 Socket 结构的简易聊天程序。

要求：

① 用户界面采用图形用户界面；
② 通信的双方都为本机（即当前主机既是服务器又是客户机）；
③ 服务器同时只和一个客户机建立连接。

程序设计如下：

（1）服务端程序

```
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
public class Server1 extends WindowAdapter implements ActionListener
{
    ServerSocket server;                    //提供服务的 ServerSocket 对象
    Socket client;                          //与客户端通信的 Socket 对象
    boolean flag;                           //标记是否继续提供 Socket 服务

    BufferedReader cin;                     //Socket 对象的字符输入流
    PrintWriter cout;                       //Socket 对象的字符输出流

    Frame    f;                             //聊天程序界面框架
    Button b1;
    TextArea ta;
    TextField tf;

    public Server1()
    {
        try
        {
            server = new ServerSocket(1234);        //创建 ServerSocket 对象
            this.ChatFrame("服务端    端口号:" + server.getLocalPort());        //显示聊天程序界面
            flag = true;
            while(flag)                     //提供 Socket 服务
```

```
        {
            client = server.accept();                             //接收一个客户端的连接申请
            System.out.println("已建立连接!");

             InputStream is = client.getInputStream();            //获取 Socket 对象的输入流
             cin = new BufferedReader(new InputStreamReader(is));         //创建字符输入流
             OutputStream os = client.getOutputStream();          //获取 Socket 对象的输出流
             cout = new PrintWriter(os, true);                    //创建字符输出流

             String aline;
             while((aline=cin.readLine())!=null)                  //等待接受输入流数据
             {
                 ta.append(aline+"\n");            //将从输入流中读入的字符串添加到多行文本框
                 if(aline.equals("bye"))                          //获得结束标记时停止服务
                 {
                     flag = false;
                     break;
                 }
             }
             is.close();                                          //关闭流
             cin.close();
             os.close();
             cout.close();
             client.close();                                      //关闭与客户端的 Socket 连接
        }
    }
    catch(Exception e)
    {
        System.out.println(e);
    }
}

public void ChatFrame(String str)                                 //构造聊天程序窗口界面
{
    f = new Frame("聊天程序"+str);
    f.setSize(400,300);

    ta = new TextArea();                                          //创建多行文本输入区对象
    f.add(ta);
    ta.setEditable(false);                                        // ta 置为不可编辑状态

    Panel p = new Panel();                                        //创建面板对象
```

```
        f.add(p,"South");                               // p 放在 f 的下部
        tf = new TextField(20);                         //创建文本输入行对象
        b1 = new Button("Send");                        //创建按钮对象
        p.add(tf);                                      //把 tf 添加在 p 上
        p.add(b1);                                      //把 b1 添加在 p 上
        b1.addActionListener(this);                     //注册 b1 的 Action 事件
        tf.addActionListener(this);                     //注册 tf 的 Action 事件

        f.setVisible(true);
        f.addWindowListener(this);                      //注册 f 的 Window 事件
    }

    public void actionPerformed(ActionEvent e)          //处理按钮单击事件
    {
        ta.append(tf.getText()+"\n");
        cout.println(tf.getText());                     //向 Socket 对象的字符输出流发送字符串
        tf.setText("");
    }

    public void windowClosing(WindowEvent e)            //单击窗口关闭按钮时
    {
        cout.println("bye");                            //向客户端发送结束标记
        System.exit(0);                                 //程序运行结束
    }

    public static void main(String args[])
    {
        new Server1();
    }
}
```

（2）客户端程序

```
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
public class Client1 extends WindowAdapter implements ActionListener
{
    Socket client;                                      //客户端 Socket 对象
    boolean flag;                                       //标记 Socket 对象是否已连接

    BufferedReader cin;                                 //Socket 对象的字符输入流
```

```
PrintWriter cout;                                        //Socket 对象的字符输出流

Frame    f;                                              //聊天程序界面框架
Button b1;
TextArea ta;
TextField tf;

public Client1()
{
    try
    {                                                    //连接服务器端，这里使用本机
        client = new Socket("localhost",1234);
        System.out.println("已建立连接!");
        this.ChatFrame("客户端    端口号:1234");             //聊天程序界面
        flag = true;
        while(flag)
        {
            InputStream is = client.getInputStream();        //获取 Socket 对象的输入流
            cin = new BufferedReader(new InputStreamReader(is));        //创建字符输入流
            OutputStream os = client.getOutputStream();      //获取 Socket 对象的输出流
            cout = new PrintWriter(os,true);                 //创建字符输出流

            String aline;
            while((aline=cin.readLine())!=null)              //等待接受输入流数据
            {
                ta.append(aline+"\n");            //将从输入流中读入的字符串添加到多行文本框
                if(aline.equals("bye"))
                {
                    flag = false;
                    break;
                }
            }
            is.close();                                      //关闭流
            os.close();
            System.exit(0);                                  //程序运行结束
        }
    }
    catch(Exception e)
    {
        System.out.println(e);
    }
}
```

```
    public void ChatFrame(String str)              //构造和服务器端相同的聊天程序窗口界面
    {
        f = new Frame("聊天程序"+str);
        f.setSize(400,300);

        ta = new TextArea();
        f.add(ta);
        ta.setEditable(false);

        Panel p = new Panel();
        f.add(p,"South");
        tf = new TextField(20);
        b1 = new Button("Send");
        p.add(tf);
        p.add(b1);
        b1.addActionListener(this);
        tf.addActionListener(this);

        f.setVisible(true);
        f.addWindowListener(this);
    }

    public void actionPerformed(ActionEvent e)      //处理按钮单击事件
    {
        ta.append(tf.getText()+"\n");
        cout.println(tf.getText());                 //向 Socket 对象的字符输出流发送字符串
        tf.setText("");
    }

    public void windowClosing(WindowEvent e)        //单击窗口关闭按钮时
    {
        cout.println("bye");                        //向客户端发送结束标记
        System.exit(0);                             //程序运行结束
    }

    public static void main(String arg[])
    {
        new Client1();
    }
}
```

程序说明：

① 程序的运行步骤是，首先打开一个 DOS 窗口，运行服务器端的程序，将弹出一个如图 12.3（a）所示的窗口（但还没有聊天的内容）；然后运行客户机端的程序，将弹出一个如图 12.3（b）所示的窗口（也还没有聊天的内容）。此时双方建立起了连接，就可以聊天了。

② 两个窗口相同。都是窗口的下部是一个文本编辑行，用来输入要聊天的内容；窗口的上部是一个多行文本输入区，设置成不能编辑状态，用来显示双方聊天的内容。

③ 无论哪一方（客户机端或服务器端）都可以在自己窗口的文本编辑行中输入要聊天的内容，内容输入完后，单击“Send”按钮，把这些内容发送给对方（显示在对方窗口的多行文本输入区中），并把这些内容显示在自己窗口的多行文本输入区中。

④ 程序中的语句

```
b1.addActionListener(this);
tf.addActionListener(this);
```

表示按钮对象 b1 和文本编辑行对象 tf 的 Action 事件都注册为本类的处理方法。这样用户输入完聊天内容后，按回车换行键和单击“Send”按钮效果将一样。

服务器端和客户机端的窗口分别如图 12.3（a）和图 12.3 （b）所示。

（a）服务器端窗口　　（b）客户机端窗口

图 12.3　聊天室图形用户界面

12.4　数据报通信

TCP/IP 协议是一个点对点的、可靠连接的通信协议。但是，在实际的网络通信应用中，有时并不要求连接百分之百的可靠，也不要求数据传递必须按照一定的顺序进行，只要求用户发送的数据能够在一定的时间内到达指定目的地就可以了。这时，就可以使用数据报通信方式。

数据报（Datagram）是通过网络传递的一个独立的、自我封装的数据包，它可以从网络中的一个点传递到另外一个点。数据报通信以 UDP 协议为基础。

数据报通信主要使用在数据量大的应用环境下，如网络游戏，视频会议等。

java.net 包中提供了两个类：DatagramSocket 类和 DatagramPacket 类，用来支持数据报通信。DatagramSocket 类用来在程序之间建立传送数据报的通信连接， DatagramPacket 类用来表示数据报。

1．DatagramSocket 类

DatagramSocket 类主要用来在程序之间建立传送数据报的通信连接。

DatagramSocket 类的构造方法有：

- ⌘ DatagramSocket()
- ⌘ DatagramSocket(int prot)
- ⌘ DatagramSocket(int port, InetAddress laddr)

其中，port 指明要使用的端口号，如果未指明端口号，则为本地主机上一个可用的端口；laddr 指明一个可用的本地 IP 地址。

DatagramSocket 类的常用方法有：

- ⌘ void connect(InetAddress address, int port)　　以地址 address、端口号 port 建立连接
- ⌘ InetAddress getInetAddress()　　获取 IP 地址
- ⌘ int getPort()　　返回端口号
- ⌘ void receive (DatagramPacket p)　　接收数据报 p
- ⌘ void send(DatagramPacket p)　　发送数据报 p

2．DatagramPacket 类

DatagramPacket 类用来表示一个数据报。

DatagramPacket 的构造方法有：

- ⌘ DatagramPacket(byte buf[], int length)；
- ⌘ DatagramPacket(byte buf[], int length, InetAddress addr, int port);
- ⌘ DatagramPacket(byte[] buf, int offset, int length)；
- ⌘ DatagramPacket(byte[] buf, int offset, int length, InetAddress addr, int port)；

其中，buf 用来存放数据报数据，length 为数据报中的字节长度，addr 和 port 分别为机器的 IP 地址和端口号，offset 指明了数据报的位移量。

DatagramPacket 的常用方法有：

- ⌘ InetAddress getAddress()　　获取 IP 地址
- ⌘ int getPort()　　返回端口号
- ⌘ byte[] getData()　　返回要发送或要接收的数据
- ⌘ int getLength()　　返回要发送或要接收的数据长度
- ⌘ void setAddress(InetAddress iaddr)　　设置 IP 地址为 iaddr
- ⌘ void setPort(int iport)　　设置端口号为 iport

在接收数据前，要创建一个 DatagramSocket 对象和一个 DatagramPacket 对象（要给出接收数据的缓冲区及其长度），然后调用 DatagramSocket 类的方法 receive()等待数据报的到来。receive()方法将一直等待，直到收到一个数据报为止。如下面语句就完成了这样的任务：

```
DatagramSocket socket = new DatagramSocket(4445);      //4445 为端口号
DatagramPacket packet = new DatagramPacket(buf, 256);
Socket.receive(packet);
```

发送数据前，也要创建一个 DatagramSocket 对象和一个 DatagramPacket 对象（参数要包

括：缓冲区、目的地 IP 地址和端口号）。发送数据是通过 DatagramSocket 的方法 send()实现的。send()方法根据数据报的目的地址来寻址和传递数据报。如下语句就完成了这样的任务：

```
DatagramSocket socket = new DatagramSocket(4446);
DatagramPacket packet=new DatagramPacket(buf, length, address, port);
Socket.send(packet)；
```

用数据报方式编写客户机/服务器模式应用程序时，无论在客户端还是服务器端，首先都要创建一个 DatagramSocket 对象，用来发送数据报或接收数据报，然后使用 DatagramPacket 对象作为传输数据报的载体。

习题 12

一、基本概念题

12.1 什么是计算机网络？按照功能划分，计算机网络由哪些部分组成？它们的作用是什么？

12.2 什么是主机名和 IP 地址？它们的关系如何？

12.3 下列说法是否正确，为什么？

① 使用 IP 地址可以惟一标识网络上的一台计算机。

② TCP 和 UDP 协议都是面向连接的、可靠的协议。

③ 0～2048 之间的端口被系统保留，应用程序应使用 2048 以上的端口进行通信。

12.4 什么是端口？它有什么作用？

12.5 UR 类和 InetAddress 类有什么区别？

12.6 解释 C/S 模式的工作方式？

12.7 解释基于 TCP /IP 的通信和基于 UDP 的通信各自的工作机制。说明它们的应用场合有何不同。

12.8 使用 URL 类通信和使用 URLConnection 类通信有什么不同？

二、程序设计题

12.9 运行本章中要求读者自己运行的程序，并观察屏幕输出。

12.10 编写一个可以查询网络域名的程序。

12.11 编写一个可以下载二进制格式的文件程序。

12.12 编写一个可以周期性地更新服务器上的一个文本文件的程序。

12.13 编写一个 C/S 模式的程序，让客户机可以周期性地获得服务器上的系统事件。

12.14 参照本章的聊天室程序，编写一个功能更加完善的聊天室程序。

第 13 章　JSP 简介

教学要点

本章内容主要包括网络服务和动态网站的基本概念，JSP 的运行机制和主要特点，建立运行环境的方法（包括 JSP 服务器的安装方法），JSP 的基本语法、指令语句和内置对象。

本章是进一步学习高级 Java 程序设计的导引，要求能够理解本章的内容。

JSP（Java Server Page）是基于 Java 语言、运行于网络服务器的一种脚本语言。JSP 技术主要用于开发网站，它能够为用户提供包含动态数据的网页。本章主要介绍 JSP 技术及其在网络服务中的应用，包括网络服务和动态网站的基本概念、JSP 技术的原理和特点、JSP 运行环境的建立和 JSP 的基本语法。另外，本章还介绍 B/S 结构的基本概念，编写的例子也是 B/S 结构的。

13.1　网络服务和动态网站的基本概念

13.1.1　计算机网络和网络服务

计算机网络是指由多台计算机通过各种通信设备连接组成的、支持特定网络通信协议的计算机系统。根据计算机在网络中所起的不同的作用，可以把网络中的计算机分为服务器和客户机两大类。客户机提出服务请求，服务器接到服务请求后，提供所需要的服务，这样就构成了一个基于网络的计算机服务系统。

1．客户机、服务器和通信协议

（1）客户机

客户机是网络中发出服务请求的计算机终端或应用程序。这些服务请求通过网络通信设备按照一定的通信协议，以一定的文件格式传送到服务器上。

（2）服务器

服务器是网络中响应服务请求的计算机终端或应用程序。服务器在接收到客户机发来的服务请求后，安排运行相应的服务程序，然后服务器将程序运行的结果输出到适当的文件中，最后将文件返回给用户，这样就完成了一个网络服务的过程。

（3）通信协议

通信协议是计算机网络中计算机之间进行通信时共同遵守的规则。从网络服务的过程可以看出，在客户机和服务器两端各自运行有不同的程序，但是，由于客户机和服务器两端都执行了同样的通信协议，使得客户机和服务器上运行的程序可以按照协议规定的信息格式传递或交换信息。

在计算机网络中，一台计算机既可以作为客户机，同时又可以作为服务器。这台计算机通过运行一个客户端软件来发出服务请求，这时它扮演一个客户机的角色；如果这台计算机又安装了如 Tomcat 这样的服务器，它也能对用户发来的请求进行服务，这时它又是一个服务器。本章的一些例子，其客户机和服务器都是一台计算机。

2．Internet 和 WWW

（1）Internet

Internet（因特网）是由分布在全世界的成千上万的计算机网络、遵循一定的通信协议联系在一起而组成的国际互连网络。它是一个由建立和使用这些网络的人群、团体、公司及各种网络资源组成的集合体。

（2）WWW

WWW（World Wide Web）是 Internet 上的一种基于 HTML 协议的信息载体。WWW 用链接的方法能非常方便地从 Internet 上的一个站点访问另一个站点，从而可根据用户需要从 WWW 上获取丰富的信息。

要访问 WWW 上的信息资源需要三个部件的支持：浏览器、Web 服务器和 HTTP 协议。浏览器是运行在客户端用来显示 HTML 文件内容的客户端软件，它支持 HTTP 协议，通过该协议，浏览器可以和 Web 服务器建立联系并互相传递信息。用户只要在本地机上安装了浏览器，就可以向 Web 服务器发出各种服务请求，并且接收和显示从 Web 服务器上发来的 HTML 文件。目前常用的浏览器有 Internet Explorer, Netscape 等。

3．B/S 结构和 HTTP 协议

（1）B/S 结构

B/S 结构中，B 是指浏览器（Browser），S 是指服务器（Server）。浏览器和服务器通过 HTTP 协议，可以交换包括 HTML 在内的很多种类型的文档。这些文档包括了文本、图像、音频、视频等多种类型。

（2）HTTP 协议

HTTP 协议的全称是超文本传输协议，是专门为浏览器/服务器网络服务模式设计的一种网络协议。用户和服务器建立连接后，就可以按照 HTTP 协议约定的格式传递信息了。客户通过浏览器可以向服务器发出服务请求，也可以通过浏览器来显示从服务器返回的查询结果。

4．B/S 结构的网络服务

B/S 结构的网络服务的工作过程是这样的：服务器上的一个进程始终监听端口，当发现有用户发出服务请求时，服务器就会和客户机建立一个连接，并且在服务器上运行相应的服务程序，随后将程序运行的结果放入一个 HTML 文件中返回给用户，最后释放客户机和服务器之间的连接。

B/S 结构的网络服务的一个重要特征是无状态连接，即连接仅在一个服务过程中保持，浏览器与服务器都不保存以往的连接状态。

13.1.2　网站和 JSP

可以用浏览器浏览的 HTML 文件称为网页。网站是连接在计算机网络上的若干网页文件

的集合。网页文件一般分为两种：静态网页和动态网页。静态网页是用单纯的 HTML 语言编写的，一般只能显示一些静态的信息。它的缺点是不能和用户进行交互。由于静态网页只能表现固定不变的数据，所以它的用途非常有限。动态网页支持动态数据访问，它可以运行在服务器上，动态地获取各种数据，并将这些数据插入 HTML 文件中传递给用户。

包含动态网页的网站开发技术目前有许多种技术方案。JSP 是一种新的、基于 Java 语言的技术方案。

JSP（Java Server Page）是由 Sun 公司推出的、基于 Java 语言的、运行于网络服务器上的一种脚本语言。JSP 主要用于开发包含动态网页的网站。JSP 被认为是未来最有发展前途的网站技术之一。

13.2 JSP 的原理和特点

13.2.1 Servlet 和 JSP

1．Servlet

Servlet（服务器小程序）是在服务器中运行的 Java 代码，这就像 Applet 是在浏览器中运行的代码一样。所有基于 Java 的服务器端程序都是构建在 Servlet 之上的。编写 Servlet 程序需要的类和接口都放在 javax.servlet 包中。

Servlet 的工作过程是：①接受从客户端发来的服务请求；②运行相应的程序，获得动态数据；③将动态数据传递给服务器。

Servlet 和 JSP 有密切的关系，所有的 JSP 文件最终都要转化成 Servlet，通过 Servlet 来实现动态网页的功能。

2．JSP

JSP 文件由 HTML, Java 代码和 JSP 标记组成，其文件名后缀是“.jsp”。

在用户浏览器访问带有 JSP 的页面文件时，支持 JSP 的服务器会解释其中的程序，将执行结果插入 HTML 文件，然后把这个文件返回给客户机上的浏览器。所有的 JSP 程序都在服务器上执行，客户机上的浏览器仅仅用于显示返回的结果。

13.2.2 JSP 的运行机制

JSP 运用 Java 技术对 HTML 网页功能进行了扩展，在一个 JSP 文件中，代码一般分为两部分：HTML 代码和 JSP 代码。执行 HTML 代码时，系统就直接把静态页面文件返回给用户浏览器。

执行 JSP 代码时，系统要做许多工作。JSP 程序的执行过程如下：

① JSP 容器（对文件中的 JSP 标识进行识别和转换的程序）解析 JSP 文件并生成一个 Java 文件（.java 文件）。

② 编译 Java 文件，生成字节码文件（.class 文件），这个文件就是可执行的 Servlet 文件。

③ Servlet 容器（对文件中的 Servlet 标识进行识别和转换的软件）装载可执行的 Servlet 文件并执行。

④ 服务器将程序的运行结果放入 HTML 文件，并将此文件返回用户浏览器显示。

JSP 服务器工作过程原理如图 13.1 所示。

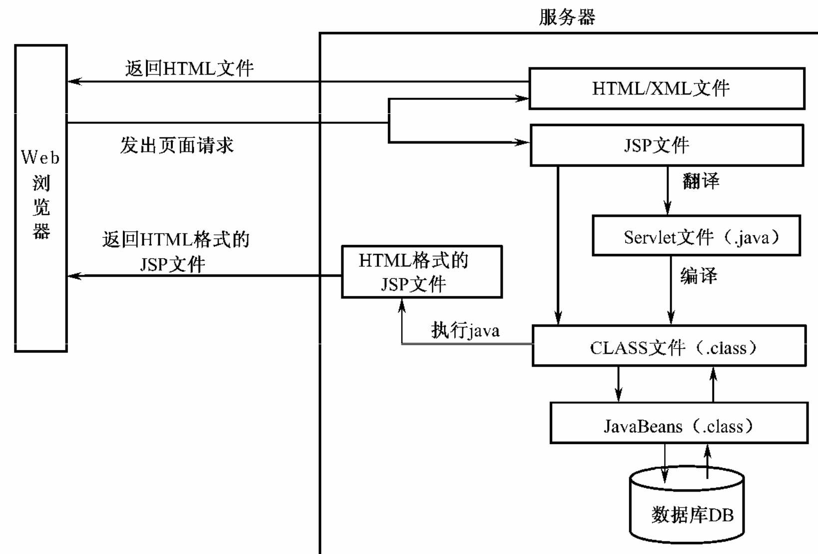


图 13.1　JSP 服务器工作过程原理图

在一台计算机上安装了支持 JSP 的服务器软件后，这台计算机就可以作为服务器使用。服务器软件（如 Tomcat）一般都既支持执行 HTML 代码，也具有上述执行 JSP 代码的所有功能。支持 HTML 的服务器称作 Web 服务器，支持 JSP 的服务器称作 JSP 服务器。所以说，大部分服务器软件（如 Tomcat）都兼备了 Web 服务器和 JSP 服务器的功能。

13.2.3　JSP 的主要特点

JSP 的主要特点是：

① 高效和安全。JSP 文件执行时被编译成字节码，因此执行效率高、安全性好。

② 方便程序的开发和维护。一个基于 JSP 技术的网络应用系统，其程序结构主要由三部分组成，即客户端程序，服务器端的 JSP 程序和服务器端的 Java 程序。服务器端的 JSP 程序主要实现业务逻辑功能。这样，当用户的需求发生变化时，只需修改 JSP 程序就可以了。实际上，目前广泛使用的一些解决方案（如 J2EE 等）是 JSP 技术的延伸。

③ 适合多种服务器环境。JSP 的代码可以在 Tomcat, IIS 等服务器上执行，支持的操作系统包括 Unix、Windows、Linux 等。开发的 JSP 程序基本上不用任何修改就可以在其他的平台上运行。

13.3　运行环境的建立

13.3.1　安装支持 JSP 的服务器

构建 JSP 环境主要是安装和配置支持 JSP 服务器的软件。目前流行的服务器软件有 JSWDK, Tomcat, Jrun, Resin, Weblogic 等。这里以 Tomcat 为例，介绍 JSP 服务器软件的安装和配置。

Tomcat 是 Sun 公司推出的支持 JSP 服务器的软件。Tomcat 支持 JSP 和 Servlet，可免费使用，非常适合个人学习使用。在安装 Tomcat 之前，要先安装好 1.3.1 节介绍的 JDK 环境。

1．安装 Tomcat 服务器

从 http://Jakarta.apache.org.（Apache 网站）下载合适的（或最新版本的）Tomcat 服务器软件，有光盘的也可直接用光盘安装。按照 Tomcat 服务器安装指南，将 Tomcat 服务器安装到适当的硬盘目录下。安装完成以后，系统将建立许多目录，主要的几个目录及作用如下：

\bin	存放系统运行文件
\conf	存放系统配置文件
\lib	存放系统的各种类和接口文件
\logs	存放日志文件
\webapps	系统提供的应用范例文件和用户文件

用户开发的 JSP 文件应存放在\webapps 目录的下一级目录 ROOT 下。

2．环境变量的配置

环境变量是用来记录系统或应用程序配置信息的一些变量。可通过 autoexec.bat 文件中的参数来配置。通过该文件给 CLASSPATH 环境变量添加 Apache Tomcat 4.0\lib 目录的路径（下面的配置语句是基于 jdk1.4 和 tomcat 都安装于 C 盘的根目录下）：

```
SET CLASSPATH=.;c:\jdk1.4\lib;C:\Apache Tomcat 4.0\lib
```

环境变量也可以通过 Windows 的“我的电脑”\“属性”中“高级”选项的“环境变量”来配置。

13.3.2 JSP 运行环境的测试

1．启动服务器

在 Windows 的“开始”的“程序”菜单中选择“Apache Tomcat 4.0”菜单的 start Tomcat 选项。Tomcat 将启动，启动成功后出现如图 13.2 所示的窗口。

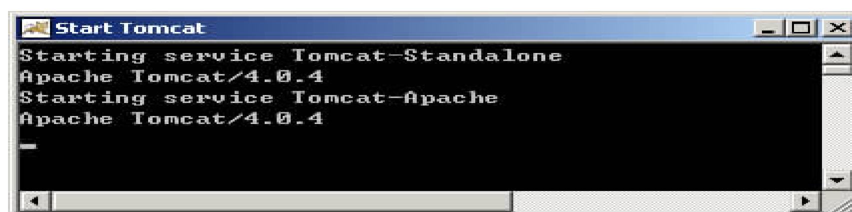


图 13.2 启动 Tomcat 服务器的结果图

2．运行 Tomcat 的首页

打开浏览器，在浏览器的地址栏中输入“http://locathost:8080/”，如果出现如图 13.3 所示的页面，则说明服务器环境已经构建好了。上面地址中，locathost 是 Tomcat 安装后系统设置的主机名，8080 是 Tomcat 安装后系统设置的默认端口号。

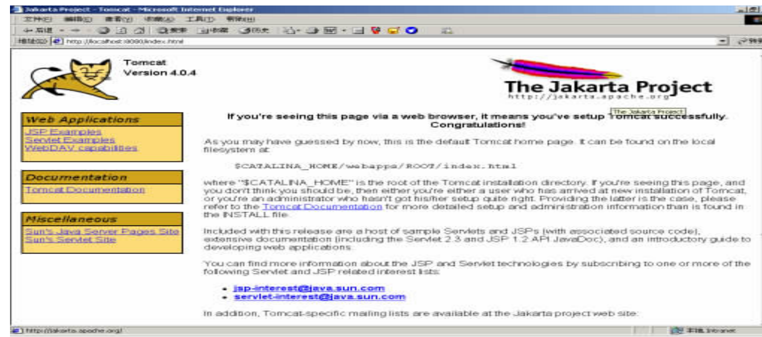


图 13.3　Tomcat 服务器首页

3．运行自己的 JSP 文件

构建好了服务器环境，就可以编写和运行自己的 JSP 文件了。

【例 13.1】　在浏览器窗口显示“ Hello, World! ”。

程序代码如下：

```
〈%@ page language="Java" %〉
〈%@ page info="a hello world example" %〉
〈html〉
〈head〉
〈title〉 Hello 〈/title〉
〈/head〉
〈body〉
〈h1〉
〈%out.println(" Hello, World! ");%〉
〈/h1〉
〈/body〉
〈/html〉
```

文件名取为“ helloworld.jsp ”，将该文件存入目录 C:\ Apache Tomcat 4.0\webapps\ROOT\ 下。C:\ Apache Tomcat 4.0\webapps\ROOT 目录是上述配置下存放用户程序的目录。

运行：在浏览器的地址栏键入“ http://localhost:8080/helloworld.jsp ”。

运行结果：将出现如图 13.4 所示窗口。

图 13.4　JSP 程序示例

上述程序中：

①〈%@ page language="Java" %〉说明脚本语言的类型，这里使用 Java 语言作为脚本语言。所有能嵌入 HTML 文件中的程序语言都称为脚本语言，Java 语言只是这些语言的一种，所以必须用此语句给服务器说明。

②〈%@ page info="a hello world example" %〉用来描述文件信息。

③〈%out.println(" Hello, World! "); %〉调用了一个内部页面输出对象 out，在 HTML 文件中加入" Hello, World! "字符串。服务器在执行到该语句时，由 Tomcat 服务器首先将它翻译成 Servlet 程序；然后，服务器运行这个程序，将结果输出到 HTML 文件中；最后，将文件返回给用户浏览器。这样就在浏览器窗口看到了图 13.4 所示的结果。

13.4 JSP 的基本语法和内置对象

13.4.1 JSP 的基本语法

JSP 文件一般由注释、声明、表达式和脚本等部分组成。不同的部分用不同的标记语法表示。

JSP 文件包括 HTML 和 JSP 两种类型的标记。HTML 标记用一对尖括号〈〉括起来。JSP 标记用一对尖括号加一对百分号〈% %〉括起来。

1．注释

注释分 HTML 注释和 JSP 注释两种。

HTML 注释语法为：〈!--注释语句--〉。例如，〈!—这是一个变量--〉为 HTML 注释。

JSP 注释语法为：〈%--注释语句--%〉。例如，〈%这是一个 JSP 程序%〉为 JSP 注释。

2．声明语句

声明语句用于声明变量或方法。在 JSP 文件中，一次可以声明一个或多个变量（或方法），它们之间用逗号隔开。声明中使用的 Java 语言必须符合 Java 语言规范。

声明语句语法为：〈%! 声明；…[声明;] %〉。

例如，下面语句声明了五个变量 *i*，*x*，*y*，*z*，*s*。其中，*i* 赋初值 9，*s* 赋初值" Welcome to JSP world "。

```
〈%! int i =9 %〉
〈%! int x, y, z; %〉
〈%! String s = "Welcome to JSP world"; %〉
```

注意：

① 变量必须先声明，然后才能在后面语句中使用。

② 声明变量语句必须以分号结尾。

③ 所声明变量的使用范围是本页面。

【例 13.2】 在 JSP 文件中定义一个变量，然后使用该变量。文件名为" variable.jsp "。

JSP 程序（文件名为" variable.jsp "）设计如下：

```
<%@ page language="Java" %>
<%@ page info="a JSP example" %>
<html>
<head>
<title> 测试声明字符串 </title>
</head>
<body>
<%! String str=new String("声明字符串");%>
<%out.println(str+" 第一次看到");%>
<br>
<%out.println(str+" 第二次看到");%>
</body>
```

运行该 JSP 程序的命令在浏览器的地址栏输入，运行结果窗口如图 13.5 所示。

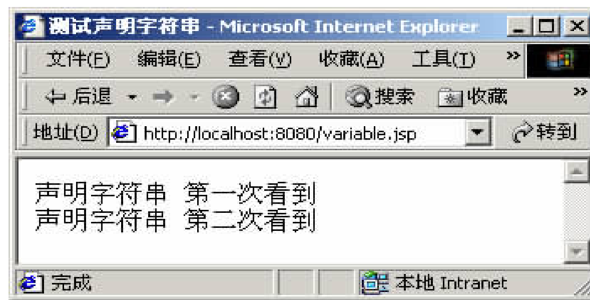


图 13.5　声明语句示例

3．表达式

表达式用于在 JSP 文件中输出表达式的值。当表达式的值为数值类型时，需要转化成字符串类型才能输出。

表达式语法为：<%= 表达式%>

例如，下面第一个语句定义了 String 类型变量 s，且初值为"I love Java"；第二个语句输出变量 s 的值。

```
<%String s = "I love Java"; %>
<%=s%>
```

4．脚本段

JSP 是一种基于 Java 语言的脚本语言。在 JSP 文件中，可以声明多个变量和方法，可以使用任何有效的表达式，可以使用任何 JSP 内置对象或任何用<jsp:useBean>标签声明的对象，还可以使用任何有效的 Java 程序，这些都在脚本段标记。

脚本段语法为：<% 代码段 %>。

下面的 JSP 文件中就通过<% %>标记插入了一段 Java 代码。

```
<html>
<head>
<title> 测试一 JSP 文件 </title>
</head>
<body>
<%
      int i;
      for(i=1;i <=10;i++)
      {
          out.println(" <font size="+i+"> "+i+" </font> ");
      }
%>
</body>
</html>
```

该文件运行后浏览器窗口的显示结果如图 13.6 所示。

图 13.6　Java 代码示例

13.4.2　JSP 的指令语句

JSP 文件中还包括很多指令语句，下面对这些指令语句做一个简单介绍。

1．include 指令语句

include 指令是静态包含指令，即将被包含的文件的执行结果插入到原 JSP 文件中。include 指令语句的语法格式是：

```
<% include file="relativeURL"%>
```

其中，URL 是统一资源定位器（参见 12.2.1 节），这说明只要给出 URL 地址，就可以包含网络上的任何一个文件；relativeURL 指 URL 的相对地址，下面的例子就是一个 URL 相对地址的例子。

```
<%@include file="top.jsp"%>
```

这里，文件 top.jsp 是指当前 URL 下的文件。作者计算机上当前的 URL 是“ http://localhost:8080 ”，所以该文件的 URL 地址是 “ http://localhost:8080/ top.jsp ”。

被 include 指令包含的文件可以是 JSP 文件、HTML 文件或文本文件。如果被包含的文件是 JSP 文件，则 JSP 容器编译完该 JSP 文件后，将执行的结果插入到原文件中 Include 指令所在的位置。如果被包含的文件是 HTML 文件或文本文件，则 JSP 容器不对其进行编译，直接将其内容插入到原文件中 Include 指令所在的位置。

2．Page 指令

Page 指令用于定义文件的全局属性，其作用范围是整个页面。

Page 指令的语法格式是：

```
〈%@ page
[ language = "java" ]
[ extends = "package.class" ]
[import= "package.class" ]
……
%〉
```

语句含义说明如下：

① language = “java”，用来指定程序语言的类型，其值为“ java ”。例如：

```
〈%@ page language="java"%〉
```

② extends = "package.class"，用来指定需要继承的类名。例如：

```
〈%@ page extends="java.lang.thread"%〉
```

③ import=”package.class”，用来指定文件中需要导入的类或包。例如：

```
〈%@ page import="java.io.*, java.util.*"%〉
```

Page 指令中还可以定义许多其他属性，如 buffer（缓冲区大小），isThreadSafe（是否支持多线程）等，这里省略。

3．〈jsp:useBean〉指令

Java Bean 是实现特定功能的组件。〈jsp:useBean〉指令的功能是能够在 JSP 页面中使用一个 Java Bean 对象。此语句的语法较为复杂，也较难理解，这里以后面购物车例子中的语句为例来说明：

```
〈jsp:useBean id="cart" scope="session" class="myservlet.MainServlet" /〉
```

其中，id="cart"指明 Java Bean 中定义的对象标识是 cart；scope="session"指明 Java Bean 中对象 cart 的作用范围是 session。session 表示其作用范围是该网页的连接持续时间；class="myservlet.MainServlet"指明对象 cart 的类名是 MainServlet，包名是 myservlet，即类 MainServlet 在包 myservlet 中。

13.4.3 JSP 的内置对象

所谓内置对象，是 JSP 在客户端和服务器之间建立连接时创建的对象。这些对象包含了许多与特定用户请求、页面、应用程序等相关的信息。用户可以直接使用 JSP 的内置对象。

JSP 的内置对象是在 Serverlet API 中定义的，因此，本节讨论的内容在一定程度上也是

在介绍 Servlet 的方法。

常用的内置对象有 out, session, request, response, application, pageContext 等。

1．out 对象

out 对象用于向客户端输出数据。

常用的方法有：

⌘ void print (String s)　　向用户输出字符串 s

⌘ void println (String s)　　向用户输出字符串 s 并换行

out 对象的使用方法见下面的例子。

2．session 对象

session 对象是保存客户端信息的对象，用来保存从和服务器建立连接到断开连接期间的用户私有信息。当一个用户登录某一个网站时，系统就会为它生成一个独一无二的 session 对象（该对象有惟一的 Id 标识）来记录该用户的个人信息。一旦用户退出网站，这个对象就会被注销。

常用的方法有：

⌘ getId()　　获得 session 的 ID 地址

⌘ getCreationTime()　　获得 session 的创建时间

⌘ getLastAccessedTime()　　获得 session 的最后一次访问时间

【例 13.3】　利用 session 对象获得客户端信息。

JSP 文件设计为：

```
〈%@page import="java.util.*"%〉
〈%
out.print("sessioID 是：");
out.println(session.getId()+"〈br〉" );
out.print("session 对象创建的时间是：");
out.println(new Date (session.getCreationTime()).toString()+"〈br〉" );
out.print("客户端最后一次访问时间是：");
out.println(new Date (session.getLastAccessedTime()).toString()+"〈br〉" );
out.println("〈br〉");
%〉
```

浏览器的显示窗口如图 13.7 所示。

图 13.7　session 对象的使用

3．request 对象

request 对象包含客户端向服务器发出请求的内容。使用 request 对象可以获得用户请求的各种信息。

常用的方法有：

⌘ getProtocol()　　　　　　获得连接使用的协议
⌘ getServerName()　　　　　获得服务器的名称
⌘ getServerPort()　　　　　获得服务器的端口
⌘ getHeader (String name)　　获得客户端使用的浏览器参数，name 是客户名

【例 13.4】　利用 request 对象获得客户端向服务器发出请求的信息。

JSP 文件设计为：

```
〈html〉
〈body bgcolor="white"〉
〈h1〉 Request Information 〈/h1〉
〈font size="4"〉
连接使用的协议：          〈%= request.getProtocol() %〉〈br〉
服务器的名称：            〈%= request.getServerName() %〉〈br〉
服务器的端口：            〈%= request.getServerPort() %〉〈br〉
客户端使用的浏览器        〈%= request.getHeader("User-Agent") %〉〈hr〉
〈/font〉
〈/body〉
〈/html〉
```

浏览器的显示窗口如图 13.8 所示。

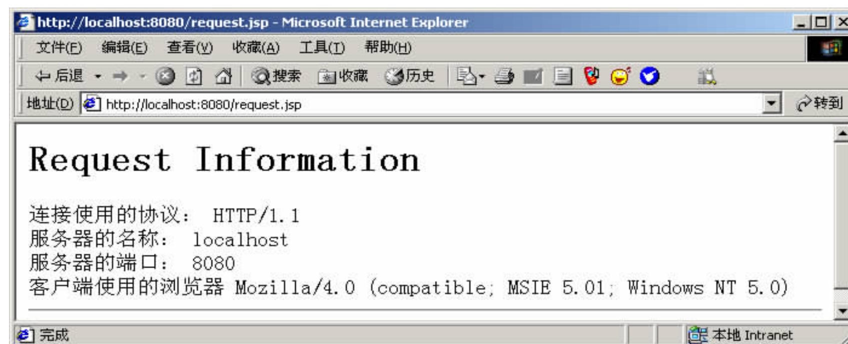


图 13.8　request 对象的使用

4．response 对象

response 对象是与 request 对象相对的一个对象，response 对象是服务器端用于向客户端做出响应的对象，其中包含了许多服务器端向客户端做出响应的信息。

5．application 对象

application 对象用来保存服务器从启动到关闭过程中的信息。服务器启动时 application

对象创建，服务器关闭时这个对象就撤销了。application 对象中存放着服务器与本次服务相关的信息。

常用的方法有：

⌘ getServerInfo()　　　　　　　　获得服务器的信息

⌘ getRealPath()　　　　　　　　　获得服务器的当前工作路径

【例 13.5】　用 application 对象获得服务器的信息。

JSP 文件设计为：

```
〈%@page import="java.util.*"%〉
〈%@page contentType="text/html;charset=gb2312"%〉
〈%
out.print("服务器的信息：");
out.println(application.getServerInfo()+"〈br〉");
out.println(application. getAttibuteNames()+"〈br〉");
out.print("服务器的当前工作路径：");
out.println(application.getRealPath(request.getServletPath())+"〈br〉");
%〉
```

运行结果如图 13.9 所示。

图 13.9　application 对象的使用

6．pageContext

pageContext 对象相当于当前页面的容器，可以通过 pageContext 对象访问当前页面的所有对象。

常用方法有：

⌘ HttpSession getSession()　　　　取得当前页面的 session 对象

⌘ ServletRequest getRequest()　　　取得当前页面的 request 对象

⌘ ServletResponse getResponse()　　取得当前页面的 response 对象

13.5　应用举例

本节以一个简单例子说明 JSP 技术的应用。

【例 13.6】　设计一个简单的网上购物应用系统。

程序设计如下：

（1）客户端页面文件（文件名为 carts.html）

这是一个客户端的页面文件，页面中的下拉列表中有许多供用户选择购买的物品。

文件清单如下：

```
〈html〉
〈body bgcolor="white"〉
〈form type=POST action=mycarts.jsp〉〈BR〉
请选择增加或删除的项目:〈br〉
增加或删除的项目:
〈SELECT NAME="item"〉
〈OPTION〉Java 程序设计
〈OPTION〉C++ 程序设计
〈OPTION〉操作系统
〈OPTION〉数据库原理
〈/SELECT〉
〈br〉〈br〉
〈INPUT TYPE=submit name="submit" value="增加"〉
〈INPUT TYPE=submit name="submit" value="删除"〉
〈/form〉
〈/body〉
〈/html〉
```

程序设计说明：

语句：〈form type=POST action=mycarts.jsp〉〈BR〉的作用是：当用户选择完所购物品，单击了"增加"或"删除"按扭后，将由服务器端的 mycarts.jsp 文件来进行处理。

（2）服务器端的 JSP 文件（mycarts.jsp）

服务器端的 JSP 文件负责处理客户端的服务请求。

文件清单如下：

```
〈html〉
〈jsp:useBean id="cart" scope="session" class="myservlet.MainServlet" /〉
〈jsp:setProperty name="cart" property="*" /〉
〈% cart.processRequest(request);%〉
〈br〉你已经选择了下列项目放入了你的购物车:
〈ol〉
〈%
    String[] items = cart.getItems();
    for (int i=0; i <items.length; i++〉  {%}
        〈li〉〈%= items[i] %〉
      〈%〉
%〉
〈/ol〉
〈/html〉
```

程序设计说明：

① 语句：〈jsp:useBean id="cart" scope="session" class="myservlet.MainServlet" /〉的作用是：告诉服务器端的 JSP 文件（即 mycarts.jsp），对象 cart 所属的类及该类定义的方法（如 getItems()方法）到哪里去寻找。这里说明的该对象所属的类在 myservlet 包的 MainServlet 类中。

上述语句中，id="cart"指明 Java Bean 中定义的对象标识是 cart；scope="session"指明 Java Bean 中对象 cart 的作用范围是 session。session 表示其作用范围是该网页的连接持续时间；class="myservlet.MainServlet"指明对象 cart 的类名是 MainServlet，包名是 myservlet，即类 MainServlet 在包 myservlet 中。

② 语句：〈jsp:setProperty name="cart" property="*" /〉的作用是：指明 Java Bean 中定义的对象名是 cart。

（3）服务器端的 Java 程序（MainServlet.java）

服务器端的 Java 程序（即 MainServlet.java 程序）用来完成服务器端的业务逻辑处理。它是用 Java 语言编写的程序，运行前需要编译成.class 文件，系统实际运行时调用的是编译后的 MainServlet.class 文件。

文件清单如下：

```
package myservlet;
import javax.servlet.http.*;
import java.util.Vector;
public class MainServlet                              //定义一个购物车
{
    Vector v = new Vector();
    String submit = null;
    String item = null;
    private void addItem(String name)                 //添加物品
{
        v.addElement(name);
    }
    private void removeItem(String name)              //删除物品
{
        v.removeElement(name);
    }
    public void setItem(String name)                  //选择物品
{
        item = name;
    }
    public void setSubmit(String s)                   //设置提交内容
{
        submit = s;
    }
    public String[] getItems()                        //获取用户选择的物品
```

```
    {
        String[] s = new String[v.size()];
        v.copyInto(s);
        return s;
      }
      public void processRequest(HttpServletRequest request)      //完成客户端的提交处理
    {
        if (submit == null)
        addItem(item);
        if (submit.equals("增加"))
            addItem(item);
        else if (submit.equals("删除"))
            removeItem(item);
        reset();
      }
    }
```

程序运行过程如下：

在客户端的浏览器的地址栏内键入"http://localhost:8080/carts.html"（这是作者计算机上存放客户端 HTML 文件的 URL 地址），浏览器窗口显示如图 13.10 所示。

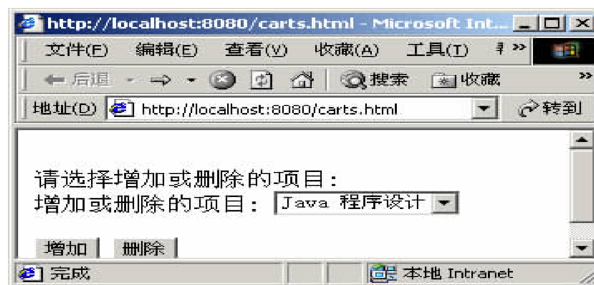


图 13.10　浏览器窗口显示

用户选择要增加或删除的物品，单击了"增加"或"删除"按钮后，购物车的显示结果如图 13.11 所示。这里选择了三项物品。

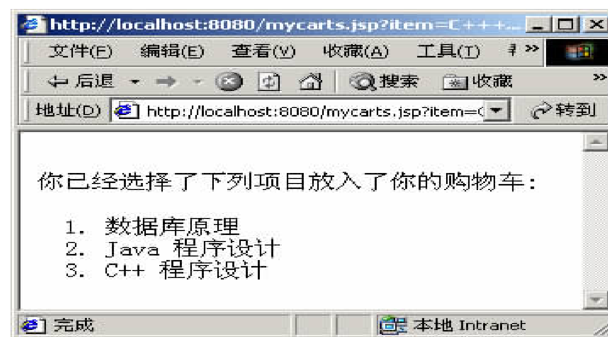


图 13.11　购物车中的货物

习题 13

一、基本概念题

13.1　什么是服务器？什么是客户机？什么是浏览器？它们在网络中的作用各是什么？

13.2　什么叫静态网页？什么叫动态网页？什么叫网站？

13.3　什么是 JSP 容器？它的作用是什么？

13.4　解释 C/S 模式和 B/S 模式的工作方式有何不同？它们各有什么特点？

13.5　什么是 JSP？什么是 Servlet？它们在网路服务中分别起什么作用？

13.6　JSP 和 HTML 文件的关系如何？

13.7　JSP 文件由哪几部分组成？

13.8　JSP 的主要特点是什么？

13.9　JSP 中有哪些常用的内置对象，试说明 request 和 response 的主要方法。

二、程序设计题

13.10　安装和配置可以运行 JSP 文件的服务器，运行服务器上的 JSP 文件例子。

13.11　编写一个可以统计网站访问者信息的 JSP 文件。

13.12　编写一个用于网站用户注册的 JSP 文件。

参 考 文 献

1 [美]Harvey M Deitel & Paul J Deitel 著．Java 大学教程（第 4 版）．北京：电子工业出版社，2002.6

2 [美]Laura Lermay & RogersCadenhead 著．束闻，王国良，林勇民译．Java2 编程 21 天自学通．北京：清华大学出版社，2002.4

3 [美]Terrence W Pratt & Marvin V Zelkowitz 著．傅育熙，黄林鹏，张冬莱等译校．程序设计语言设计与实现（第 3 版），北京：电子工业出版社，1998.11

4 [美]William Stanek 著．Java 2 认证考试指南（原书第 3 版）．北京：机械工业出版社，2002.3

5 [美]Bruce Eckel 著．Java 编程思想．北京：机械工业出版社，2002.9

6 叶核亚，陈立编著．Java 2 程序设计实用教程．北京：电子工业出版社，2003.5

7 王克宏主编．Java 技术教程（基础篇）．北京：清华大学出版社，2002.4

8 杨绍方，王颖，林锦全编著．Java 程序设计基础．北京：科学出版社，2001.7