第 1 章 Java 概述

Java 是 Sun 公司推出的一种面向对象的编程语言，从其诞生之日起，就因其易学易用、功能强大等特点得到广泛的应用，并逐渐成为程序开发的必然选择。Java 最主要的优势是强大的跨平台性，该特性使得 Java 程序可以运行在任何一个系统平台上，尤其是随着嵌入式技术和产品的发展，在各类电器、手机中都能发现 Java 程序的身影。

1.1 Java 简介

1995 年，Sun 公司推出了一种面向对象的程序设计语言——Java，这是一种通过解释方式来执行的语言，其语法规则与 C++语言类似。但是 Java 语言在语法规则上摒弃了在使用上较难掌握的指针，用 Java 语言编写的程序既可以是编译型的，也可以是解释型的，这一点与 C++语言完全不同。

1.1.1 Java 的发展

为了更好地了解 Java、学习 Java 编程，下面对其发展历史进行简要的回顾。

1990 年，Sun 公司成立了一个名为 Green Team 的研究小组，致力于开发一种能够在消费性电子产品上运行的分布式系统架构。该小组的成员 James Gosling 对 C++在执行过程中的表现非常不满，于是把自己封闭在办公室里编写了一种新的语言，并将其命名为 Oak，这个名称源于 Gosling 从他的办公室窗户中向外看到的一棵橡树。后来，一位律师发现已经有另一个产品注册了这个名称，为了避免不必要的麻烦，就将其改为 Java。

在 Java 语言开发完成之时，正赶上 Internet 的起步。1993 年，随着 Internet 网页浏览器 Mosaic 的诞生，James Gosling 认为 Internet 和 Java 语言的特性是非常吻合的，于是开始使用 Java 在 Internet 平台上编写具有交互性的网页程序，即 Java Applet。1994 年，Sun 公

司用 Java 语言编写了 HotJava。这是一个能够运行 Java 程序的 Web 浏览器，人们开始认识到 Java 语言的两项功能，即能向网络提供什么和能够创建什么类型的程序。从此，越来越多的人将目光投向了 Java，并开始学习 Java。

Netscape 是第一个认可 Java 语言的公司，于 1995 年将 Java 解释器集成到 Navigator 浏览器中。Microsoft 随后也在 Internet Explorer 中认可了 Java，这使得人们可以在 IE 浏览器中首次运行交互程序。

1995 年，Oak 被正式命名为 Java。Sun 公司在当年发布了其 JDK（Java development kits）1.0 版本，这标志着 Java 语言正式面向编程开发人员发布。由于 Java 语言去除了 C++语言中为了兼容 C 语言而保留的非面向对象的内容，所以使用 Java 编写的程序更加严谨、可靠、易懂。尤其是 Java 语言特有的“一次编写，处处运行”的跨平台优点，使其特别适合在网络应用程序开发中使用，从而成为一种极具潜力的面向对象开发工具。

2009 年，Oracle 公司正式收购 Sun 公司，并致力于完善 Java 的功能，使其更加成熟和强大。

到目前为止，Java 语言已经成功地应用于程序开发的各个方面，包括桌面程序开发、网站开发和嵌入式的开发等。现在市场上很多品牌的手机都支持 Java 游戏。除了一些应用软件，如 OpenOffice 等是用 Java 语言开发的，一些 Java 程序员经常使用的开发工具，如 Eclipse、NetBeans 和 JBuilder 等也都是用 Java 语言编写的。

1.1.2 Java 的不同版本

自 Java 问世起，Sun 公司就致力于使其无所不能。因此按照应用范围的不同，Java 可以分为 3 个不同的版本，分别是 Java SE、Java EE 和 Java ME。

1）Java SE

Java SE 是 Java 的标准版，也是 Java 的基础，包含 Java 语言基础、JDBC 数据库操作、输入/输出、网络通信和多线程等技术，主要用于桌面应用程序的开发。

2）Java EE

Java EE 是 Java 的企业版，其核心是 EJB，主要用于开发企业级分布式的网络程序，如企业 ERP 系统与电子商务网站等。

3）Java ME

Java ME 主要用于嵌入式系统的开发，如手机等移动通信电子设备。现在大多数品牌手机都支持 Java 技术。

1.1.3 Java 与 C++的关系

Java 语言的出发点是给 C++语言增加新功能，同时去掉 C++语言中难以掌握的功能，因此从这一点能够明确 Java 语言与 C++语言的直接关系，即 Java 语言继承了 C 语言的大部分语法，但其对象模型却是从 C++语言中改编得来的。

现代编程语言的先驱是 C 语言，C++语言是在 C 语言的基础上通过增加面向对象特性

而形成的，因此C++语言包括了所有 C 语言的特征、属性和优点。而 Java 语言是为满足 Web 的需要而在 C 语言和 C++语言的基础上开发出来的，所以 Java 语言大部分的特性也是从 C 语言和 C++语言中继承得到的。

由于 Java 语言与 C++语言的相似性，特别是它们对面向对象程序设计的支持，很多人认为 Java 语言就是 C++语言的 Web 版本，或者 C++语言的增强版本。但实际上这种观点是错误的，因为 Java 在实际应用以及基本原理上都与 C++语言有显著的不同。例如，Java 语言不提供 C++语言的向下或向上兼容功能，最主要的是 Java 语言并不是为了替代 C++语言而设计的。

1.1.4　Java 的实现机制

用 Java 语言编写的程序既是编译型的，又是解释型的。Java 源程序代码经过编译之后转换为一种称为 Java 字节码的中间语言，Java 虚拟机将对这些字节码进行解释和运行。编译只进行一次，而解释在每次运行程序时都会进行。编译后的字节码采用一种针对 Java 虚拟机优化过的机器码形式保存，由 Java 虚拟机将字节码解释为机器码，然后在计算机上运行。所以，Java 语言和 C++语言明显的区别不是语法规则，而是实现机制。

1)Java 虚拟机

Java 虚拟机(Java virtual machine，JVM)是用软件模拟实现的虚拟计算机。Java 虚拟机定义了指令集、寄存器集、类文件结构栈、垃圾回收集等，提供了 Java 跨平台能力的基础框架。

Java 虚拟机执行过程有以下 3 个典型的特点。

(1)多线程。Java 虚拟机支持多个线程同时运行，这些线程可独立执行 Java 代码并处理公共数据区和私有栈中的数据。

(2)动态连接。Java 虚拟机具有动态连接的特性，使得 Java 程序很适合在网络上运行。

(3)异常处理。Java 虚拟机提供了可靠的异常处理机制，这使得 Java 程序更安全、更可靠。

2)垃圾回收机制

Java 系统不仅要分配对象使用的内存资源，还要跟踪资源的使用情况，定期检测不再使用的内存资源，并自动回收后进行再分配，这就是 Java 中的垃圾回收机制。与使用 C++语言不同，程序员在使用 Java 语言开发程序时不用再考虑对象的释放问题，同时还提高了程序的安全性，避免因资源消耗而造成的系统隐患。

3)代码安全性检查

Java Applet 可以将远程代码下载到 Web 浏览器上运行，这改变了传统模式下程序的运行方式，扩展了 Web 浏览器的功能，但同时也带来了安全隐患。因此，必须建立行之有效的安全模型，进行代码安全性检测，通过对一些操作的限制来增强网络的安全性，如不能对本地文件进行访问、不能建立新的网络连接等。

Java 采用域管理方式的安全模型，无论是本地代码还是远程代码都可以通过配置策略设定可访问的资源域。这一策略可有效地解决区分本地代码和远程代码所带来的困难。

1.1.5 Java 的特性

总体说来，Java 语言具有以下特点。

1)简单

Java 语言的语法与 C++语言的非常相似，因此，熟悉 C++语言的程序设计人员会很快掌握 Java 语言。Java 语言不仅去掉了 C++语言中模糊、复杂、不常用、容易出错的特性及影响程序健壮性的地方，如指针、结构、运算符的重载、多重继承等，而且通过实现对无用单元的自动收集，大大简化了程序设计者的内存管理工作。从某种程度上说，Java 语言是从 C 语言和 C++语言转变而来的。

Java 语言还提供了丰富的类库、API 文档、第三方开发包和大量基于 Java 语言的开源项目，并开放了 JDK 源代码，程序设计人员可以通过分析这些资料来提高自己的编程水平，使编程变得更简单。

2)面向对象

面向对象(object oriented，OO)是 Java 语言的基础，也是 Java 语言的重要特性。面向对象编程是指将计算机程序概念化成一组分离的对象，这些对象彼此之间可以进行交互。一个对象中包含了信息的访问及改变这些信息的方法，这使得程序更易于改进、理解并具有重用性。

在 Java 语言中，一切都是对象，因此不能在类外面定义单独的数据和函数，所有对象都要派生于同一个基类 Object，并共享它所有的功能。也就是说，Java 语言最外部的数据类型是对象，所有元素都要通过类和对象来访问。

3)平台无关

“一次编写，处处运行”是 Java 语言的宣传口号，这一口号反映了 Java 语言的平台无关性。在 Java 语言中规定了统一的数据类型，这为 Java 程序跨平台的无缝移植提供了很大便利。Java 编译器将 Java 程序编译成二进制代码，即字节码。字节码有统一的格式，不依赖于具体的硬件环境，在任何安装有 Java 解释器的系统中都可以执行这些代码。解释器针对不同的处理器指令系统将字节码转换为不同的具体指令，保证了 Java 程序的“处处运行”。

Java 语言这种与平台无关的特性使得 Java 程序可以方便地被移植到不同系统的计算机中。同时，Java 语言的类库中也实现了针对不同平台的接口，使得这些类库也可以被移植，进而提高了 Java 程序的性能。

4)分布性

Java 语言的分布性包括操作分布和数据分布。其中，操作分布是指在多个不同的主机上布置相关的操作；数据分布是将数据分别存储在不同的主机上。这些主机是网络中的不同成员，Java 语言借助于 URL 对象来访问网络对象，其访问方式与访问本地系统的方式相同。

5)安全性

Java 语言删除了类似 C 语言中的指针和内存释放等方面的语法，从而有效地阻止了对

内存的越权访问。Java 程序代码需要经过代码校验、指针校验等很多测试步骤才能够运行，因此未经允许的 Java 程序是不可能出现损害系统平台的行为的，从而保证了系统的安全性。

6)高效性

Java 程序经过编译器编译后生成的字节码与平台无关，能够很容易地直接转换成对应于特定 CPU 的机器码或汇编码，因此，Java 程序的运行速度是很快的。

7)多线程

多线程机制能够使应用程序在同一时间内并行执行多项任务，而且相应的同步机制可以保证不同线程能够正确地共享数据。Java 内置了语言级的多线程功能，使用户程序可以并行执行。使用多线程，程序设计者可以分别用不同的线程完成特定的行为，使程序具有更好的交互能力和实时运行能力。

8)动态性

在 Java 语言中，动态调整库中的方法和变量是非常简单、直接的，客户端无须做任何改变。这是因为 Java 语言中的类是根据需要装入的，在类库中加入新的方法和实例变量是不会影响用户程序执行的；并且 Java 语言通过接口来支持多重继承，比严格的类继承更灵活，也更具有扩展性。

1.2　搭建 Java 开发环境

Java 不仅是一种程序语言，也是一种程序设计平台；既是一种开发环境，又是一种应用环境。开发 Java 程序，必须安装 JDK。JDK 是专门针对 Java 平台上发布的应用程序、Applet 及组件而提供的开发环境，其中包括了 Java 程序运行所必需的环境及在开发过程中经常用到的各种库文件。

1.2.1　安装 JDK

JDK 可以从 Java 的官方网站 http://www.oracle.com/technetwork/java/index.html 上获得，下载后就可以进行安装。本书使用的是 JDK 1.6.0_26 版本，下载文件全名为 jdk-6u26-windows-i586.exe。

具体的安装步骤如下。

(1)双击已下载的安装文件，打开 JDK 安装向导的“设置”对话框，如图 1-1 所示。

(2)单击“下一步”按钮，打开图 1-2 所示的“自定义安装”对话框。

图 1-1 “设置”对话框

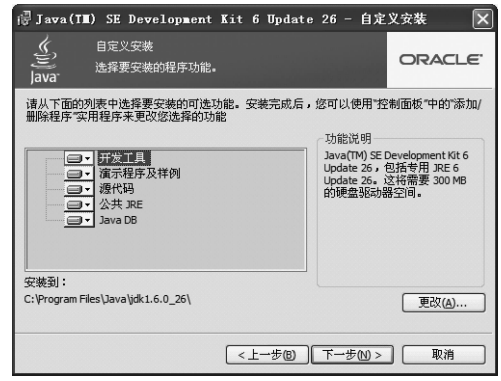

图 1-2 “自定义安装”对话框

提示:在“自定义安装”对话框中单击“更改”按钮可以更改 JDK 的安装位置。这里采用的是默认设置。

(3)在“自定义安装”对话框中,用户可以对需要安装的 JDK 内容进行自定义选择。选择完毕后,单击“下一步”按钮进行必要的文件复制和解压过程,同时打开图 1-3 所示的“目标文件夹”对话框。

(4)单击“下一步”按钮,打开图 1-4 所示的“进度”对话框,程序继续进行安装。

图 1-3 “目标文件夹”对话框

图 1-4 “进度”对话框

(5)安装完毕后,打开图 1-5 所示的“完成”对话框。

图 1-5 “完成”对话框

(6)单击该对话框中的“完成”按钮,结束 JDK 的安装。同时,系统会自动打开浏览器,弹出图 1-6 所示的注册提示信息。

图 1-6　注册提示信息

可以直接关闭浏览器忽略这一注册提示信息。至此,JDK 就被成功地安装到系统中了。

提示:在安装完 JDK 后,一般都需要重新启动系统才能生效。安装了 JDK 之后,在计算机的桌面上并没有产生快捷方式,这是与其他安装程序不同的。

1.2.2　配置 JDK

现有版本的 JDK 一般无须进行手动配置就可以直接使用,但如果安装错误或由于其他原因,运行程序时可能会出现类似于“找不到 JDK”的错误提示。为了避免这种情况的发生,还是需要手动配置 JDK 的环境变量。

配置 JDK 的操作步骤如下。

(1)在桌面上右击“我的电脑”图标,在弹出的快捷菜单中选择“属性”选项,弹出“系统属性”对话框,切换到“高级”选项卡,如图 1-7 所示。

(2)单击“环境变量”按钮,弹出“环境变量”对话框,如图 1-8 所示。

图 1-7　“系统属性”对话框

图 1-8　“环境变量”对话框

(3)选择“系统变量”选项区的列表框中的 Path 选项,单击“编辑”按钮,打开图 1-9 所示的“编辑系统变量”对话框。在“变量值”文本框中的已有内容之后输入“C:\Program Files\Java\jdk1.6.0_26\bin”,即 JDK 的安装路径,并用分号与前面的内容隔开。单击“确定”按钮结束编辑。

提示:双击选择的系统变量也可以打开对应的对话框并对其进行编辑。在进行编辑时,一定要注意用于分隔当前内容和前面内容的分号为西文符号。

(4)单击“系统变量”选项区中的“新建”按钮,弹出图 1-10 所示的“新建系统变量”对话框。在“变量名”文本框中输入 ClassPath,在“变量值”文本框中输入“.;C:\Program Files\Java\jdk1.6.0_26\lib\dt.jar;C:\Program Files\Java\jdk1.6.0_26\lib\tools.jar”,单击“确定”按钮结束编辑。

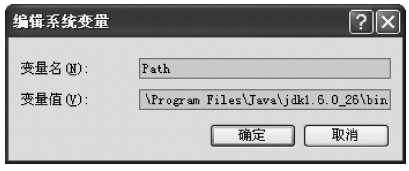

图 1-9 “编辑系统变量”对话框

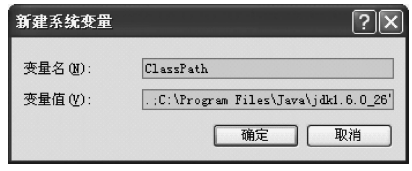

图 1-10 “新建系统变量”对话框

提示:在进行设置时,一定要输入正确的安装路径。

(5)依次关闭打开的对话框,并重新启动计算机,结束对 JDK 的配置操作。

还有一种更为简便的配置方法,操作步骤如下。

(1)在图 1-8 所示的“环境变量”对话框中单击“系统变量”选项区中的“新建”按钮,弹出“新建系统变量”对话框,在“变量名”文本框中输入 JAVA_HOME,在“变量值”文本框中输入 JDK 的安装路径,即“C:\Program Files\Java\jdk1.6.0_26”,如图 1-11 所示,单击“确定”按钮。

(2)选择“系统变量”选项区的列表框中的 Path 选项,然后单击“编辑”按钮,弹出“编辑系统变量”对话框。在“变量值”文本框中的起始位置处添加“.;%JAVA_HOME%bin;”,如图 1-12 所示。

图 1-11 新建变量 JAVA_HOME

图 1-12 编辑系统变量 Path

(3)依次关闭打开的对话框,并重新启动计算机,结束对 JDK 的配置操作。

以上两种方法均可以完成对 JDK 的配置,建议读者按操作习惯进行掌握。

1.2.3 测试 JDK

JDK 是否安装和配置正确,需要在系统的命令行中进行测试。测试方法如下。

(1)执行“开始”→“运行”命令,打开“运行”对话框,如图 1-13 所示,在“打开”文本框中

输入 cmd 后单击“确定”按钮。

图 1-13　“运行”对话框

(2)在打开的命令行窗口中的命令提示符后输入 javac 并按 Enter 键，系统会输出关于 javac 命令的帮助信息，如图 1-14 所示。

图 1-14　命令行窗口中的输出信息

(3)关闭命令行窗口，结束测试。

提示：完成以上步骤说明系统中已经成功安装和配置了 JDK，否则需要检查前面的安装和配置步骤，找出导致安装和配置失败的环节。

1.3　第一个 Java 程序

在成功安装和配置了 JDK 之后，就可以着手编写 Java 程序了。可以使用任何一种文本编辑器进行 Java 程序源代码的编写，最简单的就是 Windows 系统自带的记事本。当然，现在流行的 IDE 功能更加强大，提供完整的语法代码，自动完成 Java 程序的编译和运行，甚至有些还能够提供强大的代码辅助功能。

1.3.1　Java 程序的开发步骤

不管采用何种文本编辑器，也不管采用何种 IDE，利用 Java 来创建和开发应用程序都需要遵循以下步骤。

1)编写 Java 源程序

使用文本编辑器编写 Java 源程序，在保存时一定要注意文件的扩展名必须是.java。

2)编译 Java 源程序

使用 Java 编译器(javac. exe)将保存好的 Java 源文件编译成 Java 类文件,得到的文件的扩展名为. class。Java 类文件是由字节码构成的,所以也称为字节码文件。

3)运行 Java 程序

Java 程序可以分为 Java 应用程序(Java application)和 Java 小应用程序(Java Applet)两类。其中,Java 应用程序必须通过 Java 解释器(java. exe)解释执行,并得到 Java 类文件;Java 小应用程序则必须使用支持它的浏览器来运行。

1.3.2 Java 应用程序

简单地说,Java 应用程序是一个与浏览器无关、能独立运行的程序。Java 应用程序还可以进一步分为只支持在计算机屏幕上以字符形式输出的控制台应用程序(console applications)和可以生成并管理多个窗口的窗口化应用程序。后者使用了基于窗口的程序常用的图形化用户界面(GUI)机制,其中包括菜单、工具栏和对话框等。

对 Java 应用程序来说,在编写时必须定义一个 main()方法,程序所要完成的功能都通过语句编写在 main()方法中。该方法会在程序启动时执行。

下面就以一个简单的控制台应用程序为例介绍 Java 应用程序的开发步骤。该控制台应用程序的功能是在命令行窗口中输出一串字符。

1)编写源程序

前面已经提到,编写 Java 源程序最简单的工具就是 Windows 系统自带的记事本,以下程序演示了如何使用记事本来编写 Java 源程序。

【例 1-1】 在记事本中编写 Java 源程序。

执行“开始”→“所有程序”→“附件”→“记事本”命令,打开记事本并输入以下代码。

```
//HelloJava.java                                    //这里是注释
public class HelloJava                              // 类的名字,也是源文件的名字
{
    public static void main (String args[])         // 程序的入口处
    {
        System.out.println("Hello,I love Java!");   // 程序主体,一条输出语句
    }
}
```

注意: Java 源程序中的标点符号都是在英文状态下输入的,在输入时一定不要出错。

对于【例 1-1】中的程序,需要进行以下几点说明。

(1)一个 Java 源程序是由若干类组成的,【例 1-1】中的应用程序中只有一个类,类的名字是 HelloJava(类的定义和使用会在后面的章节中专门进行介绍)。

(2)class 是 Java 中用来定义类的关键字;public 也是关键字,用来说明 HelloJava 是一个公有的类。第一个大括号和最后一个大括号之间的内容叫做类体。

(3)public static void main(String args[])是类体中的一个方法,一个 Java 应用程序必

须且只能有一个类含有 main()方法，这个类称为 Java 应用程序的主类。

(4)在 Java 应用程序中，main()方法必须被说明为 public static void。public 关键字说明 main()方法是公有方法，可以被任何对象访问；static 关键字说明 main()方法是静态的；void 用于说明 main()方法不返回任何值。

(5)String args[]声明一个字符串类型的数组 args，它是 main()方法的参数。main()方法就是程序开始执行的入口。

(6)“//”表示程序中的注释。

注意：String 的第一个字母是大写的，程序中的大括号必须成对出现。

2)保存源程序

编写完源程序后需要将其保存在硬盘上。保存的步骤如下。

(1)执行“文件”→“保存”命令，弹出“另存为”对话框，选择文件保存的位置后，在“文件名”文本框中输入文件名 HelloJava.java，并在“保存类型”下拉列表中选择“所有文件”选项，如图 1-15 所示。

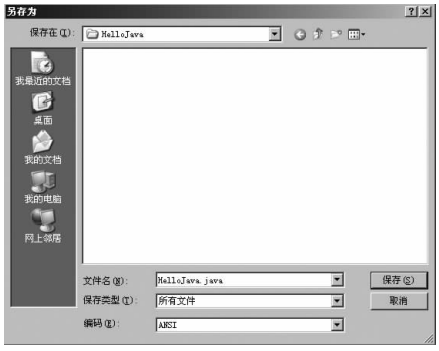

图 1-15　“另存为”对话框

注意：在保存文件时，文件名一定要和类的名字相同，包括大小写，而且一定要带扩展名.java；“保存类型”一定要选择为“所有文件”，否则保存的源文件就是文本文件，而不是后续编译时所需的源文件。

(2)单击“保存”按钮，即完成了对源程序的保存。

提示：本书所有的程序都按章节保存在“C:\JavaLearning\章节编号\文件名\”目录中。

3)编译源程序

创建了 Java 源程序后需要用 Java 编译器(javac.exe)对其进行编译。Java 编译器的作用是将编写好的 Java 源程序(.java 文件)编译成类文件(.class 文件，又称字节码文件)。Java 编译器一次可以编译多个 Java 源程序，对源程序中定义的每个类，都生成一个单独的类文件，因此，Java 源程序和生成的类文件之间并不是一一对应的。

javac 的命令格式如下。

javac 源文件表

其中，“源文件表”是指一个或多个源程序文件的名称列表。

对于源程序 HelloJava.java，其编译过程如下。

(1)打开命令行窗口，并切换到 Java 源程序的保存目录，如图 1-16 所示。

(2)在命令提示符后输入命令“javac HelloJava.java”并按 Enter 键,编译结果如图 1-17 所示。

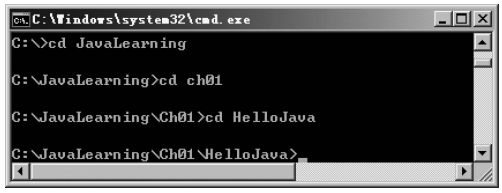

图 1-16　切换至源程序保存目录

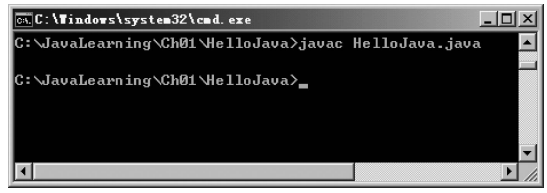

图 1-17　在命令行窗口中执行编译命令

提示:若命令行窗口中没有出现任何提示,说明源程序无语法错误,编译通过。

编译通过后,源文件所在的目录下会生成一个 HelloJava.class 文件。由于HelloJava.java 源程序中只有一个类,所以编译的结果只产生一个类文件。如果对源文件的内容进行了修改,那么必须在保存文件后对文件进行重新编译,并生成新的字节码文件。

注意:在用 javac 命令对 Java 源程序进行编译时,一定要注意 Java 源程序名是否正确,包括大小写,且不可省略文件扩展名.java。

4)执行 Java 程序

完成源程序的编译后,就可以执行 Java 程序了。前面提到过,执行 Java 应用程序时需要通过 Java 解释器(java.exe)解释执行,并得到 Java 类文件。换句话说,Java 解释器专门对由 Java 编译器编译得到的类文件进行解释执行。它的格式如下。

java 类名

提示:类名就是以.class 为扩展名的文件的名字,在这里不要求带扩展名,但必须是包含 main()方法的那个类。

对于前面编译得到的 HelloJava.class 文件,在命令行窗口的提示符后输入命令“java HelloJava”并按 Enter 键,即可得到程序运行的结果,如图 1-18 所示。

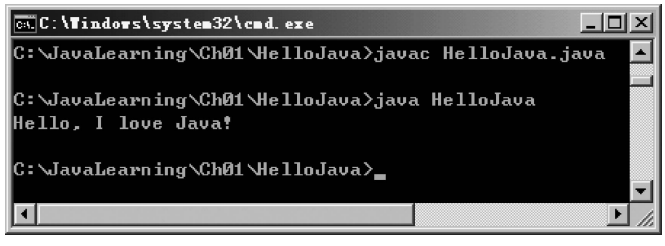

图 1-18　在命令行窗口中执行 Java 应用程序

需要说明的是,与 javac 命令一样,java 命令也要求输入正确的文件名,包括大小写等。如果在输入命令时忽略了文件名的大小写,那么会出现图 1-19 所示的错误提示。

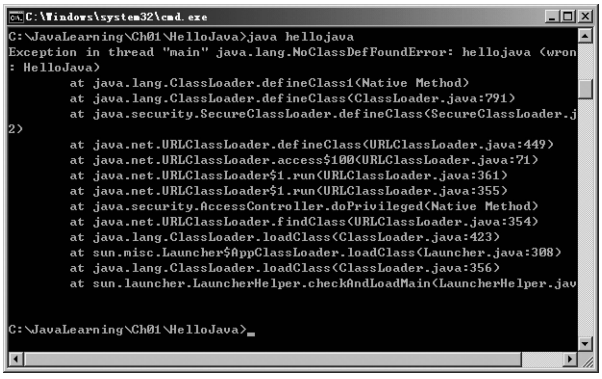

图 1-19　忽略文件名大小写时的执行错误信息

1.3.3　Java Applet

Java Applet 也称 Java 小应用程序。与 Java 应用程序不同，Java Applet 可以直接嵌入网页中，并能够产生特殊的效果，即 Java Applet 是专门为网络使用而设计的。与 Java 应用程序相比，Java Applet 借助于浏览器窗口来展示其功能，并可以响应用户界面事件。

当用户访问包含 Java Applet 的网页时，Applet 被下载到用户计算机上执行，但前提是用户所使用的浏览器支持 Java。由于 Applet 是在用户计算机上执行的，因此它的执行速度不受网络速度的限制。通过 Java Applet，可以实现图形绘制、字体和颜色控制、动画和声音插入、人机交互及网络交流等功能。

Java Applet 在结构和对运行环境的要求上与 Java 应用程序不同，下面就以一个简单的 Java Applet 为例介绍其开发步骤。

1)编写源程序

与 Java 应用程序的编写相同，Java Applet 源程序也可以使用记事本编写。以下程序演示了如何使用记事本编写 Java Applet 源程序。

【例 1-2】　在记事本中编写 Java Applet 源程序。

执行“开始”→“所有程序”→“附件”→“记事本”命令，在打开的记事本中输入以下代码。

```
// HelloJavaApplet.java                    //这里是注释
import java.applet.Applet;                 //引入 java.applet 包中的 Applet 类
import java.awt.Graphics;                  //引入 java.awt 包中的 Graphics 类
public class HelloJavaApplet extends Applet  //通过 extends 生成 Applet 的子类
{
    public void paint(Graphics g)          //Java 自动调用 paint()方法
    {
        g.drawString("Hello World!",50,50);
      //使用 Graphics 类的 drawString 方法，在 Applet 窗口中的指定位置显示文本
    }
}
```

对于【例 1-2】中的程序，需要进行以下几点说明。

(1)创建一个 Java Applet，需要创建 Applet 类的一个子类。

(2)Applet 类是 Java 预先定义好的，并存储在 java. applet 包中，提供了处理各种用户界面事件和进行屏幕绘画操作的功能。由于 Java Applet 需要使用 Applet 类中的这些方法，所以必须把 java. applet 包引入。

(3)Applet 类中定义的内容使 Java Applet 可以嵌入浏览器中工作。paint()方法就是 Applet 类中已经定义好的方法，浏览器可以自动识别并调用这个方法，将 Java Applet 想要显示的内容显示在浏览器中。

(4)Graphics 类中定义了图形用户界面中的显示方法，其 drawString()方法用于在指定的坐标位置显示字符串。

(5)在图形界面的显示上，Java 语言以左上角为坐标原点，横坐标向右延伸，纵坐标向下延伸。

(6)Java Applet 源程序的书写规范与 Java 应用程序的书写规范相同。

2)保存源程序

按照与保存 Java 源程序一样的步骤，可以对 Java Applet 源程序进行保存。本例中保存得到的文件为 HelloJavaApplet. java。

3)编译源程序

Java Applet 源程序的编译也要在命令行窗口中通过 javac 命令进行。Java Applet 源程序的编译过程与 Java 源程序的编译过程完全相同，在此不再赘述。

4)执行 Java Applet

Java Applet 的运行环境需要借助于浏览器和 HTML。浏览器是 Internet 上遵循相关协议的软件，HTML 则是浏览器的通用语言。使用浏览器可以在 Internet 上访问任何可获得的信息，这些信息以 HTML 文件格式保存在 Internet 的服务器上。用户在访问网络中的某个信息时，其实是将该处的 HTML 文件下载到本地，再通过通用的浏览器将该 HTML 文件翻译成图文并茂的网页。

早期的 HTML 文件只能包含静态的文本信息，Java Applet 的出现使得在 HTML 文件中加入动画、音频等动态内容成为可能。具体的方法就是把 Java Applet 编译生成的字节码文件嵌入 HTML 文件中，当这些文件传送到浏览器时，由浏览器中内置的 Java 解释器来执行。

要运行一个 Java Applet，首先需要为它编写一个 HTML 文件，然后在浏览器中观看这个 HTML 文件，激活浏览器中的 Java 解释器。更简单的方法是调用一些能够模拟浏览器环境并能执行 Java Applet 的命令(如 JDK 中的 appletviewer. exe)来直接运行 Java 小程序。

执行本例中的 HelloJavaApplet. java 需要按以下步骤进行。

(1)执行“开始”→“所有程序”→“附件”→“记事本”命令，在打开的记事本中输入以下代码。

```
//HelloJavaApplet.html
<html>
    <head><title>HelloJavaApplet</title></head>
```

```
    <body>
        <hr>
        <hr>
        <applet code=HelloJavaApplet width=300 height=200>
        </applet>
    </body>
</html>
```

(2)执行“文件”→“保存”命令,弹出“另存为”对话框,选择文件的保存位置,在“文件名”文本框中输入文件名 HelloJavaApplet. html,并在“保存类型”下拉列表框中选择“所有文件”选项,单击“保存”按钮。

(3)打开命令行窗口,切换到保存 HelloJavaApplet. html 文件的目录,输入命令“appletviewer HelloJavaApplet. html”,并按 Enter 键,运行结果如图 1-20 所示。

图 1-20　HelloJavaApplet 的运行结果

注意:要想使该程序正常运行,需要在执行“appletviewer HelloJavaApplet. html”命令之前,使用 javac 命令编译 HelloJavaApplet. java 程序。

至此,一个简单的 Java Applet 已经完成。需要说明的是,Java Applet 只是在最后执行的过程与 Java 应用程序有所不同,其他步骤都是相同的。

1.3.4　两类 Java 程序的比较

通过对 HelloJava. java 和 HelloJavaApplet. html 的编写、编译和执行,可知两者具有以下不同点。

(1)Java 应用程序是一个独立完整的程序。

(2)在命令行窗口中通过命令调用解释器就可以运行 Java 应用程序。

(3)Java 应用程序的主类中必须有一个定义为 public static void 的 main 方法,这是 Java 应用程序的标志,也是程序执行的入口。

(4)Java Applet 是嵌入在浏览器中运行的。

(5)运行 Java Applet 的解释器不是独立的软件工具,而是浏览器软件的一部分。

(6)Java Applet 不需要有 main 方法,但是其主类必须是 Java 类库中已经定义好的 java. applet. Applet 类的子类。

(7)Java Applet 可以直接利用浏览器或 appletviewer 提供的图形用户界面,Java 应用程序则需要另外编写代码来创建图形用户界面。

(8)Java Applet 更多体现的是状态与状态之间的切换,而不是固定的顺序执行过程,这使得它更适合于在图形界面下进行面向对象的程序设计与开发。

为了将读者的注意力更好地集中在 Java 语言的特定要点上,本书将主要以 Java 应用程序为例来讲解 Java 语言。

思考与练习

(1)开发和运行 Java 应用程序需要经过哪些主要步骤和过程?

(2)Java 应用程序由哪些部分组成? 详细说明这些组成部分的含义。

(3)与 Java 应用程序相比,Java Applet 有哪些特点?

(4)设计一个 Java 应用程序,在屏幕上用“*”输出一个菱形图案。

(5)试将 JDK 安装在 C 盘的 JavaJDK 目录中,进行配置后调试本章的示例程序。

第 2 章 Java 语言基础

所谓“工欲善其事，必先利其器”，是指想要做好一件事，需要先做好准备工作。学习编程语言也是如此，虽然前面已经介绍了如何编写一个简单的 Java 程序，但其功能仅仅是简单地在屏幕上输出一些文字和符号。如何设计出功能更加强大的应用程序呢？这就需要先从 Java 语言最基础的语法开始学起。对于任何一种由计算机语言编写的程序来说，不管其形式如何，程序中的每一条语句都是由按照一定语法规则构成的符号串组成的。

2.1 Java 的基本要素

从机器的角度看，计算机程序是一系列计算机可以执行的指令按照一定的逻辑组成的序列；从编程人员的角度看，计算机程序是一系列符合计算机语言要求的语句序列。因此，不论学习哪一种计算机程序设计语言，都需要从组成语句的基本要素开始学起。Java 的基本要素包括语句、标识符、关键字、分隔符和注释等。

2.1.1 语句

语句就是用 Java 语言书写的命令。在第 1 章中，用到了以下语句。

```
System.out.println("Hello,I love Java!");
```

Java 中的语句是程序执行的最小单位，各语句之间都用“;”隔开。一条语句可以写在连续的若干行内，一行内也可以写多条语句，如下所示。

```
int age;
String student_Name;
sum=sum+1;
return;
```

或者写成以下形式。

```
int age=23;student_Name="John";
```

虽然多条语句同时写在 Java 源程序编辑工具中的一行内并不影响 Java 编译器对语句的编译，但为了增加程序的可读性，一般不提倡将多条语句写在同一行内。相反，在语句之间加入适当的空行，或使用缩进格式，会在某种程度上增加程序的可读性，而且这些空行和使用格式不会影响 Java 编译器的工作。

为了表现一系列语句在逻辑上的连贯性，往往使用相邻的两个大括号“{”和“}”来包含这一系列语句。这一系列语句称为语句块(简称块)。

提示：语句块之间是可以嵌套的，即语句块中还可以包含子块。

2.1.2 标识符

标识符是 Java 程序的主要组成部分，每条语句中都会包含标识符。标识符是一个唯一标识变量、方法和类的名字。在 Java 中，标识符是以字母、下划线(_)或美元符($)开头，由字母、数字、下划线或美元符组成的字符串。除了这些符号，标识符中不能包含其他符号。标识符区分大小写，但没有限制长度。

下面是一些合法的标识符。

```
userName    user_Id    _system_24bit    $Hk_money
```

下面的标识符则不合法。

```
64int        &user_name      #profile
```

尽管 Java 允许用户在遵循标识符命名原则的基础上，按照自己的喜好来命名一个标识符，但对于 Java 初学者来说，标识符的命名其实是非常重要的事情。因为好的标识符命名习惯能够使编程更加规范，同时可以减少程序中错误出现的次数，也便于他人阅读和维护程序。

一般地，进行标识符命名时应要注意以下 3 点。

(1)标识符的长度适合。太长的标识符容易在程序中出现拼写错误，而太短的标识符则容易混淆。例如，一个名为_system_MaxCountOfFile 的标识符，尽管在命名中可以找到一定的规律，记忆也相对容易，但是却容易发生拼写错误，如误写为_system_MaxCountofFile。

(2)遵循见名知意原则。对于标识符的命名，尽量使用简易的英文单词组合，如类似 userName、fileSize、varLong 的标识符，而尽量避免使用简单的字符组合，如 abc、S1 这样的标识符。

提示：对于后续章节中将介绍的循环控制变量，其命名可以采用 i、j 或 k 等。

(3)尽量避免以下划线或美元符号开头。尽管 Java 的标识符命名规则中允许一个标识符以下划线或美元符号开头，但实际编程中却很少这么做。这是因为很多 C 语言程序的库名是以下划线或美元符号开头的，如果在 Java 程序中使用了这样的标识符，那么把 C 语言程序库导入 Java 程序中时，就有可能造成名字的冲突和混乱。

为了使标识符具有更好的可读性，还应当遵守一定的命名规范，具体如下。

(1)类的命名。对于类的命名，可以采用每个单词的首字母大写的书写方法，如 HelloWorld、StudentInfo、MinCount 等。

(2)函数和变量的命名。对于函数和变量的命名,可以采用第一个单词的首字母小写、其他单词首字母大写的书写方法,如 getUserID、setFileName 等。

(3)常量的命名。对于常量的命名,常采用所有字母都大写、单词之间使用下划线分开的写法,如 MAX_LENGTH、MIN_WEIGHT 和 FILE_SIZE 等。

提示:类、函数、变量与常量的概念在后续章节中会逐一介绍,这里只介绍命名规范。

尽管上面介绍的命名规范并不能提高 Java 编译器的工作效率,但显然这是一种良好的编程习惯。

2.1.3　关键字

与标识符一样,关键字也是 Java 程序的主要组成部分。所谓关键字,就是 Java 预先定义的标识符,即已经被 Java 预先赋予了特定意义的一些标识符,用户不能在程序中重新定义它们。

注意:关键字也称保留字,均为小写。Java 中不允许使用关键字作为标识符。

Java 中的关键字及其意义如表 2-1 所示。

表 2-1　Java 中的关键字及其意义

关 键 字	意　　义	关 键 字	意　　义
abstract	声明抽象类或抽象方法	interface	声明一个接口
boolean	声明一个布尔类型的变量	long	声明一个长整数类型的变量
break	中断一个循环,终止一个 switch 分支	native	声明一个本地方法
byte	声明一个字节类型的变量	new	生成一个对象,并为该对象分配内存空间
case	创建一个 switch 分支	null	空值
catch	捕获程序中的异常	package	声明一个包
char	声明一个字符类型的变量	private	声明一个私有的成员
class	声明一个类	protected	声明一个受保护的成员
continue	进入下一个循环过程	public	声明一个公有的类或成员
default	创建一个默认的 switch 分支	return	从一个方法返回(值)
do	创建一个 do-while 循环	short	声明一个短整数类型的变量
double	声明一个浮点类型的双精度变量	static	声明类的成员
else	创建一个 else 子句	super	引用一个超类成员,或调用超类的构造方法
extends	从一个类派生另一个类	switch	创建一个 switch 分支语句
false	一个布尔类型的常量	synchronized	声明一个同步方法
final	声明一个常量、最终类或方法	this	引用对象
float	声明一个浮点类型的单精度变量	throw	抛出异常

（续表）

关 键 字	意　　义	关 键 字	意　　义
for	创建一个 for 循环	throws	抛出异常
if	创建一个 if 语句	transient	声明一个临时变量
implements	实现一个接口	true	一个布尔类型的常量
import	导入一个包或类	try	封装可能抛出异常的代码块
instanceof	获得运行时对象信息	void	声明一个没有返回值的方法
int	声明一个整数类型的变量	while	创建一个 while 循环

实际上，在最新版的 Java 中，已经不再使用其中一些关键字了，但是这里仍然将其列出，希望用户在编程时避免产生将关键字作为标识符的错误。

2.1.4　分隔符

在 Java 中，有一些用于分隔不同组成要素的符号，称为分隔符。Java 中的分隔符及其用法如表 2-2 所示。

表 2-2　Java 中的分隔符及其用法

分 隔 符	解　　释	用　　法
;	分号	用于表示语句的结束，或者在 for 语句中分隔循环条件
,	逗号	用于分隔变量声明语句中连续的标识符
.	句号	用于分隔包、子包和类，或者分隔引用变量中的变量和方法
()	括号	用于在方法定义和访问时将参数列表括起来、在表达式中定义运算的先后次序，或者在控制语句中将表达式和类型转换括起来
[ ]	方括号	用于声明数组类型和引用数组的元素值
{ }	大括号	用于将若干条语句括起来作为一个语句块（代码块），或者在数组初始化时为数组赋初值

2.1.5　注释

所谓注释，就是对语句的功能进行说明、解释或标记。注释是程序中不可缺少的部分。在 Java 程序中使用注释，主要有以下两类用途。

（1）对 Java 程序进行必要的注解，增加程序的可读性。这是一种良好的编程习惯。

（2）有助于对程序进行调试和维护。最常见的做法是：将可能发生错误的代码加以注释，重新进行编译后运行，检查是否还会存在错误。如果错误出现，说明错误可能不是出现在添加了注释的这段代码中，就去除这段代码的注释，再去检查其他代码段；否则说明错误可能出现在添加了注释的这段代码中。

除了可以在单条语句末尾添加注释，在 Java 程序中的任何位置都可以添加注释。Java 编译器并不对注释中的符号和文字进行编译，因此，任何注释内容都不会影响 Java 程序的编译和运行。

提示：添加注释会增加 Java 源程序的大小，但不会增加 Java 执行文件的大小，因为 Java 编译程序会忽略语句中的注释。

Java 提供了以下 3 种添加注释的方法。

1）单行注释

Java 采用“//”作为单行注释符号。这种注释方法称为单行注释，顾名思义，就是从“//”开始到这一行的结束，都是注释的内容。例如：

```
System.out.println("  Hello,I love Java!  "); // 程序主体，一条输出语句
int age=33;  //定义了 int 型变量，并赋值为 33
```

提示：这种注释方法一般用于说明变量、语句的作用。如果连续多行进行单行注释，则相当于多行注释。

2）多行注释

Java 采用“/* …… */”作为多行注释符号。这种注释方法用于注释一行或多行，注释的内容全部被包含在“/*”和“*/”之间，例如：

```
/* ……注释内容 1
   ……注释内容 2
   …… 注释内容 3*/
```

提示：这种注释方法一般用于说明某段程序或方法的功能。在书写时，不允许再嵌套一个多行注释。

3）文档注释

Java 采用“/** …… */”作为文档注释符号。这种注释方式是 Java 独有的。当文档注释出现在任何声明（类的声明、类的成员变量或函数的声明等）之前时，都会被 JavaDoc 文档工具读取为 JavaDoc 文档内容。例如：

```
/** Class Name:  HelloJava.java
    Author:  Li
    Version:  2.0
    Date:     2012.1.16
*/
public class HelloJava
……
```

提示：文档注释一般位于变量或函数声明之前。

文档注释方法与多行注释方法在形式上很相似。如果在 Java 程序中不考虑JavaDoc 文档的读取和生成，那么文档注释的作用与多行注释是一样的。

2.2 常量与变量

Java 语言功能强大、使用灵活，与 C 语言在语法上有很多相似之处。要想熟练使用 Java 语言进行程序开发，就必须从了解 Java 语言的基础开始。本节将讲解 Java 中常量和变量的声明与应用、变量的命名规则及作用范围等。

1)常量

常量是固定不变的，一旦被定义，它的值就不能再被改变。

(1)声明常量。

声明常量的语法格式如下。

final 数据类型 常量名称[=值]

常量名称通常用大写字母来表示，如 PI、YEAR 等，但并不是硬性要求，仅仅是一个习惯而已。

(2)常量应用示例。

当常量用于一个类的成员变量时，必须给其赋值，否则会出现编译错误。

【例 2-1】 声明一个常量作为成员变量。

在记事本中输入以下代码。

```
public class FinalDemo
{
    static final int YEAR=365;
    public static void main(String[] args)
    {
    System.out.println("3/4 年等于"+3 * YEAR/4+"天");
    }
}
```

保存程序，并进行编译和运行。程序的运行结果如图 2-1 所示。

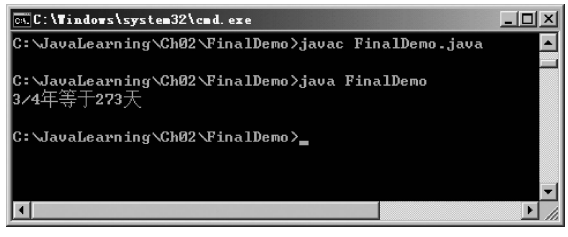

图 2-1 【例 2-1】程序运行结果

2)变量

变量是利用声明的方式将内存中的某块区域保留下来以供程序使用。可以声明为块记

载的数据类型为整型、字符型、浮点型或其他数据类型，以作变量保存之用。

变量的声明与应用同常量的声明与应用相似。下面来看一个简单的示例，在下面的程序中声明了两种 Java 经常用到的变量，分别为整型变量 num 与字符变量 ch，为它们赋值后，再把它们的值分别显示出来。

【例 2-2】 声明两个变量，一个是整型，另一个是字符型，将其赋值并输出。

在记事本中输入以下代码。

```
public class IntCharDemo
{
    public static void main(String[] args)
    {
    int num=3;
    char ch='z';
    System.out.println(num+"是整数!");
    System.out.println(ch+"是字符!");
    }
}
```

保存程序，并进行编译和运行。程序的运行结果如图 2-2 所示。

图 2-2　【例 2-2】程序运行结果

3)变量的命名规则

变量名也是一种标识符，所以它也遵循标识符的命名规则。

(1)变量名可由任意顺序的大小写字母、数字、下划线(_)和美元符号($)等组成。

(2)变量名不能以数字开头。

(3)变量名不能是 Java 中的关键字。

4)变量的作用范围

变量是有作用范围的，一旦超出它的范围，就无法再使用这个变量了。例如，张三在 A 村很知名，如果我们在 A 村打听张三，人人都知道，可如果到 B 店打听，就没人知道。也就是说，在 B 店张三是无法访问的，就算碰巧 B 店也有个叫张三的，但此张三非彼张三。

按作用范围的不同，变量分为成员变量和局部变量。

(1)成员变量。在类体中定义的变量是成员变量。成员变量的作用范围是整个类，也就是说，在该类中都可以访问这个成员变量。

【例 2-3】 探讨成员变量的作用范围。

在记事本中输入以下代码。

```
public class MemberDemo
{
    static int a=5;
    public static void main(String[] args)
    {
    System.out.println(" 成员 a 的值是:"+a);
    }
}
```

保存程序,并进行编译和运行。程序的运行结果如图 2-3 所示。

图 2-3 【例 2-3】程序运行结果

(2)局部变量。可以在 Java 程序的任何地方声明局部变量,当然也可以在循环中进行声明。

注意:在循环中声明的变量只是局部变量,只要跳出循环,这个变量就不能再使用了。

下面用实例说明局部变量的使用方法。

【例 2-4】 局部变量的使用。

在记事本中输入以下代码。

```
public class LoopDemo
{
    public static void main(String[] args)
    {
        int count=0;
        //下面是一个 for 循环,计算 1～5 的数字累加之和
        for(int i=1;i<=5;i++)
        {
            count=count+i;
            System.out.println("i="+i+" "+"count="+count);
        }
    }
}
```

保存程序,并进行编译和运行。程序的运行结果如图 2-4 所示。

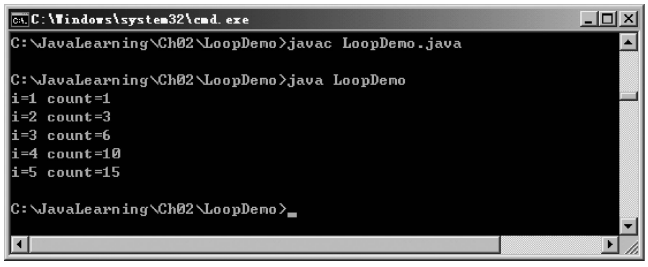

图 2-4　【例 2-4】程序运行结果

2.3　数据类型

数据类型指明了变量或表达式的状态和可操作行为。在 Java 中，主要包括整型、实型、字符型和布尔型 4 类数据类型。

2.3.1　整型数据

整型数据可分为整型常量和整型变量。

1)整型常量

Java 中的整型常量有三种形式，分别是十进制整数、八进制整数和十六进制整数。十进制整数对读者来说并不陌生，如 128、－260 等；八进制整数以 0 开头，如 0173，－0253 等；十六进制整数则以 0x 或 0X 开头，如 0x6BE 等。

2)整型变量

整型变量的类型有 byte、short、int 和 long 四种。可以在程序中通过以下语句声明一个整型变量。

```
byte x;                    //定义变量 x 为 byte 型
short y;                   //定义变量 y 为 short 型
int varCount;              //定义变量 varCount 为 int 型
long varSum;               //定义变量 varSum 为 long 型
```

也可以在一条语句中同时定义多个变量并赋初值，例如：

```
int xLength,yWidth=10;
long varLong1=75,varLong2=100;
```

对于 byte 型变量，Java 在内存中为其分配 1 字节(8 位)进行存储，因此，byte 型变量的取值范围是$-2^7 \sim 2^7-1$；对于 short 型变量，在内存中需要分配 2 字节(16 位)进行存储，因此，short 型变量的取值范围是$-2^{15} \sim 2^{15}-1$；对于 int 型变量，在内存中需要分配 4 字节(32 位)进行存储，因此 int 型变量的取值范围是$-2^{31} \sim 2^{31}-1$；对于 long 型变量，需要在内存中分配 8 字节(64 位)进行存储，因此，long 型变量的取值范围是$-2^{63} \sim 2^{63}-1$。

通常情况下，由于 byte 型变量能表示的数据范围小，容易造成溢出，所以在程序中应该避免使用，但是在分析网络协议和文件格式时，用该类型的变量却比较合适；对于 short 型变量，由于其限制了数据从高字节向低字节进行存储，所以在某些机器上也可能会出现错误；int 型和 long 型是最常用的两种整数类型，可以在程序中根据变量可能的取值进行定义。

2.3.2 实型数据

实型数据通常也称为浮点型数据，可分为实型常量和实型变量。

1)实型常量

Java 中的实型常量有两种表示形式，分别是十进制数和科学计数法。十进制数形式的实型常量由数字和小数点组成，而且必须有小数点，如 23.16、23.、.18、－0.67 等；用科学计数法表示的实型常量，如 12e5、32E－2 等，要求 e 或 E 之前必须有数，而且 e 或 E 后的指数必须为整数。

2)实型变量

实型变量有 float 和 double 两种类型。可以在程序中通过以下语句来声明一个实型变量。

```
float varLength;                    //定义变量 varLength 为 float 型
double varArea;                     //定义变量 varArea 为 double 型
```

与整型变量的定义相同，也可以在一条语句中同时定义多个实型变量并赋初值，例如：

```
float varLength,varArea=102.7;
double varFloat1=5.5,varFloat2=264.62;
```

在程序中，用 float 或 double 表示小数，而对于小数，我们最关心的是它们的精度，也就是小数的有效位数。对于某些小数，如循环小数或无限不循环小数，用二进制并不能精确地表达，只能尽量地逼近。因此，实型数据所占用的位数越多，它所表示的小数的精度就越高。

在 Java 中，float 型变量的取值范围是 3.4E－38～3.4E＋38，double 型变量的取值范围是 1.7E－308～1.7E＋308。一般来说，float 型变量所能表示的精度已经很高了，但是对于某些数值计算类型的程序，考虑到它们之间的计算所引起误差的累积和放大，因此，尽可能采用 double 型常量和变量。

如果在 Java 程序中使用一个实型常数，那么如何判断这个常数是实型数据中的哪一类呢？这需要根据这个常数的写法来判断：如果在常数后有一个字母 f 或 F，如 3.14E2f，就表明该常数是 float 型的；如果在常数后有一个字母 d 或 D，如 3.14E2D，则表明这是个 double 型常数；如果在常数后没有任何字母，则它将被默认为 double 型常数。

2.3.3 字符型数据

字符型数据可分为字符型常量和字符型变量。

1)字符型常量

Java 使用的是 Unicode 字符集，一个字符用一个 16 位的 Unicode 码表示。在 Java 中，

字符型常量一共有 65 535 个，如'A'、'J'、'文'等。

2)字符型变量

可以在程序中通过以下语句声明一个字符型变量。

```
char x;                    //定义变量 x 为字符型
char DriverName;           //定义变量 DriverName 为字符型
```

与整型变量的定义相同，也可以在一条语句中同时定义多个字符型变量并赋初值，例如：

```
char x='a',DriverName='C';
```

一些控制字符不能直接显示，可以通过转义序列来表示，如表 2-3 所示。

表 2-3　转义序列

转义序列	说　　明
\n	换行，将光标移至下一行的开始
\t	水平制表，将光标移至下一个制表符的位置
\r	回车，将光标移至当前行的开始，但不移至下一行
\\	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

注意：Java 中的字符串是一种特殊的类，在后面的章节中将对字符串和数组进行专门介绍。

2.3.4　布尔型数据

布尔型数据又称为逻辑型数据，可分为布尔型常量和布尔型变量。

1)布尔型常量

Java 中的布尔型常量只有两个，即 true 和 false。

2)布尔型变量

可以在程序中通过以下语句声明一个布尔型变量。

```
boolean T_F;               //定义变量 T_F 为布尔型
boolean Stop;              //定义变量 Stop 为布尔型
```

与整型变量的定义相同，也可以在一条语句中同时定义多个布尔型变量并赋初值，例如：

```
boolean T_F=true,Stop=false;
```

2.3.5　类型转换

在 Java 程序中，经常会遇到需要将一种类型的常量或变量转换为另一种数据类型的情

况，这称为类型转换。

类型转换的方式有两种，即隐含类型转换和强制类型转换。凡是把占用字节较少的数据转换成占用字节较多的数据，Java 都使用隐含类型转换，这一转换过程由编译系统自动完成，不需要在程序中进行特别说明。

例如，下面将 int 型数据转换成 long 型数据的语句是没有任何错误的。

```
int varInt=10;
long varLong=varInt;
```

反过来，如果使用隐含类型转换将占用字节较多的数据转换成占用字节较少的数据，就会产生编译错误。

例如，下面将 long 型数据转换成 int 型数据的语句是错误的。

```
long varLong=10;
int varInt=varLong;
```

这时，需要使用强制类型转换。强制类型转换的格式并不复杂，只在被转换的数据前加上用一对小括号括起来的数据类型名称即可，格式如下。

```
(数据类型)变量名
```

经过强制类型转换，将得到一个括号中声明的数据类型的数据。这个结果是从指定变量的值转换而来的，因此并不影响变量本身的值。

例如，将 long 型数据转换成 int 型数据的语句的正确写法如下。

```
long varLong=10;
int varInt=(int)varLong;
```

在进行强制类型转换时，必须注意将占用字节较多的数据转换成占用字节较少的数据时是否会出现数据超出类型取值范围的现象，这一现象称为数据溢出。在上面的强制类型转换中，被转换的数据是取值较小的 10，因此不存在数据溢出，但是如果被转换的数据取值很大，就可能会产生数据溢出。例如，将 long 型的 10000000000 转换成 int 型的语句如下。

```
long varLong=10000000000;
int varInt=(int)varLong;
```

执行完上面两条语句后系统会检查变量 varInt 和 varLong 的值，会发现 long 型变量 varLong 的取值保持不变，但 int 型变量 varInt 的取值却不是意料中的数据，这是因为产生了数据溢出。

除了赋值运算可能会引起类型转换外，在程序的运算过程中也可能会引起类型转换。也就是说，当程序中两个不同数据类型的数据进行运算时，系统会默认将其中占用字节较少的数据转换成占用字节较多的数据类型，然后再进行运算。

2.4 运算符与表达式

运算是对数据进行的操作，表示各种不同运算的符号称为运算符。参与运算的数据称为操作数。由操作数和运算符按一定的语法形式组成的有意义的符号序列，就是表达式。

表达式是有值的，一个变量名是最简单的表达式，它的值就是变量的值。表达式的值还可以作为其他运算的操作数。

2.4.1　赋值运算符与表达式

赋值运算就是修改变量的内容，运算的一般形式如下。

＜变量名＞＝＜表达式＞；

其含义是计算出等号右边表达式的值并将该值赋给等号左边的变量。等号“＝”是赋值运算符。例如：

```
int sum,i;        //定义整型变量 sum 与 i
sum=0,i=1;        //分别对变量 sum 和 i 赋值
sum=sum+i;        //将变量 sum 和 i 的值相加，重新赋值给变量 sum
```

提示：赋值运算中的变量名必须已经声明，表达式也必须能计算出一个确定的值，否则程序将产生编译错误。

2.4.2　算术运算符与表达式

算术运算符包括加、减、乘、除运算符和求余运算符。算术运算符是程序中最基本、最常用的运算符。表 2-2 列出了这 5 种运算符的表示、含义与用法。

表 2-2　算术运算符

运 算 符	含　义	用　法
＋	加法操作	操作数 1＋操作数 2
－	减法操作	操作数 1－操作数 2
*	乘法操作	操作数 1＊操作数 2
/	除法操作	操作数 1/操作数 2
%	求余操作	操作数 1%操作数 2

这 5 种运算符都是双目操作符，也就是说，它们都是连接两个操作数的运算符，且操作数都是整型或浮点型的常量或变量。

以下算术表达式都是合法的。

```
int varInt=10;
float varFloat=2.0;
float varResult;
varResult=varInt+varFloat;
varResult=varInt-varFloat;
varResult=varInt*varFloat;
varResult=varInt/varFloat;
varResult=varInt%varFloat;
```

2.4.3 自增、自减运算符与表达式

自增运算符++和自减运算符--都是单目运算符，也就是说，它们只对一个操作数进行运算，且该操作数必须为整型或浮点型变量，运算的结果是使操作数增 1 或减 1。

自增、自减运算符可以放在操作数之前，也可以放在操作数之后，两者的运算方式不同。如果放在操作数之前，操作数先加 1 或减 1，然后再进行表达式的运算；如果放在操作数之后，则操作数先参加表达式的运算，再自身加 1 或减 1。例如：

```
int temp=0,i=5;
```

分别执行以下 4 条语句，将得到不同的结果。

```
temp=i++;     //先计算 temp=i,再计算 i=i+1。计算结果是:temp=5,i=6
temp=++i;     //先计算 i=i+1,再计算 temp=i。计算结果是:temp=6,i=6
temp=i--;     //先计算 temp=i,再计算 i=i-1。计算结果是:temp=5,i=4
temp=--i;     //先计算 i=i-1,再计算 temp=i。计算结果是:temp=4,i=4
```

2.4.4 关系运算符与表达式

关系运算符用于比较两个操作数的关系，所有运算的结果都是布尔型的值。当运算符对应的关系成立时，运算结果是 true，否则返回 false。如果表达式 x>y 的运算结果是 true，就表明该表达式中的大小关系成立，即 x 大于 y；否则，运算结果为 false 时，表明该表达式中的大小关系不成立，即 x 小于 y。

关系运算符包括大于、小于、大于或等于、小于或等于、等于和不等于 6 种运算符。表 2-3 列出了这 6 种运算符的表示、含义与用法。

表 2-3 关系运算符

运 算 符	含　义	用　法
>	大于	操作数 1>操作数 2
<	小于	操作数 1<操作数 2
>=	大于或等于	操作数 1>=操作数 2
<=	小于或等于	操作数 1<=操作数 2
==	等于	操作数 1==操作数 2
!=	不等于	操作数 1!=操作数 2

以下关系表达式都是合法的。

```
5>7;
(x+y)<9;      //x 和 y 必须是整型或浮点型变量
```

对于关系运算符，需要注意以下两点。

(1)判断两个数是否相等的等于运算符是由两个等号组成的，如果错误地写成一个等号，将会导致程序出现逻辑错误。

(2)整型或浮点型的变量和表达式可以通过以上关系运算符形成关系表达式。但是浮点型的数据在表达上难免会有微小的误差,无法进行精确的相等比较,因此,对两个浮点型数据进行相等比较是没有意义的。

2.4.5　逻辑运算符与表达式

逻辑运算符只能处理布尔型数据,得到的结果也都是布尔值,也就是说,参加运算的操作数和运算结果都是布尔值。

常用的逻辑运算符包括逻辑与、逻辑或和逻辑非。表 2-4 列出了这 3 种运算符的表示、含义与用法。

表 2-4　逻辑运算符

运 算 符	含　　义	用　　法
&&	逻辑与	操作数 1&&操作数 2
||	逻辑或	操作数 1||操作数 2
!	逻辑非	!操作数 1

其中,逻辑与、逻辑或是双目操作符,逻辑非是单目操作符;逻辑运算符的操作数必须为布尔型常量或变量。表 2-5 列出了用逻辑运算符进行逻辑运算的结果。

表 2-5　逻辑运算结果

操作数 1	操作数 2	操作数 1&&操作数 2	操作数 1||操作数 2	!操作数 1
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

以下逻辑表达式都是合法的。

```
(2<9)&&((10-5)>3)
// 2 小于 9,结果为 false,(10-5)大于 3,结果为 true,逻辑与操作后的结果为 false
(16/4)>=(4-1)||true
// 16 除以 4 等于 4,(4-1)等于 3,4 大于或等于 3,其结果为 true,与 true 进行逻辑或
//操作后结果仍为 true
```

注意:由于关系运算符和算术运算符的级别均高于逻辑运算符,因此,在逻辑表达式中,应先计算级别高的关系表达式和算术表达式。

2.4.6　运算符的级别和结合性

Java 中的表达式是由运算符连接起来、符合 Java 规则的语句,而运算符的优先级别决定了表达式中运算执行的先后顺序。也就是说,优先级高的先运算,优先级低的后运算。运

算符的结合性决定了并列的相同运算的先后执行顺序。

例如，对于加法操作，x+y+z 等价于(x+y)+z。所以在表达式中可以根据实际需要，在表达式中使用括号“()”来显式地标明运算次序，括号中的内容先被计算，得到的值再与其他运算数进行运算。如果在一个表达式中有多个括号嵌套，则内层括号中的表达式具有更高的优先级，即要先计算内层括号中的表达式。

本书只介绍了 Java 的基本运算符，表 2-6 给出了 Java 中所有的运算符，并按照优先级顺序进行排列。排在上面的运算符具有较高的运算优先级，同一行中的运算符的优先级相同。

表 2-6　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

注意：表 2-6 中的“左→右”表示从左向右的运算顺序。

2.5　程序结构

按照程序的执行流程，其控制结构可以分为顺序结构、选择结构和循环结构 3 种。

1)顺序结构

一般情况下，程序是按照语句的书写顺序执行的，如图 2-5 所示，程序先执行<语句 1>，再执行<语句 2>。

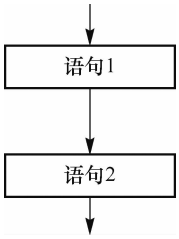

图 2-5　顺序结构

顺序结构是最简单，也是最基本的一种流程控制。该结构下的程序语句将按照它们被书写的先后顺序依次执行。因此，在任何一种语言中，都没有为顺序结构定义专门的流程控制语句，只要在编程时将需要的语句按照希望其执行的顺序来书写即可。

与顺序结构不同，选择结构和循环结构是比较复杂的程序控制结构。Java 语言分别为它们定义了专门的流程控制语句。

2)选择结构

在有些情况下，程序的执行是有条件的。当某个条件成立时，执行一段代码；条件不成立时，则执行另一段代码，或者不执行任何一条语句。这种情况称为选择结构，如图 2-6 所示。

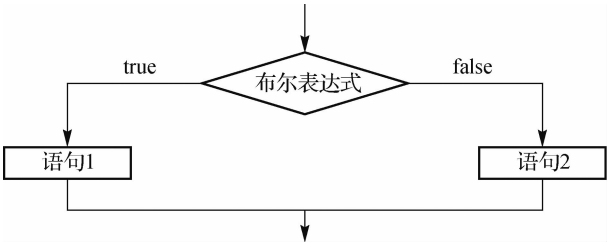

图 2-6　选择结构

在程序设计方法学中，条件判断一般用菱形框表示。程序根据＜布尔表达式＞的值选择执行＜语句 1＞或＜语句 2＞。

Java 语言中的选择结构可以用两种语句来实现，分别是 if 语句和 switch 语句。

(1)if 语句。if 语句的定义形式如下。

```
if(＜布尔表达式＞)
    ＜语句 1＞;
[else＜语句 2＞;]
```

其中，if 和 else 是关键字。if 语句的含义是：当＜布尔表达式＞的值为 true 时，程序执行 if 后面的＜语句 1＞，否则执行 else 后面的＜语句 2＞。所执行的＜语句 1＞和＜语句 2＞可以是一条单独的语句，也可以是用“{”和“}”括起来的一个语句块。else 子句是可选的。

例如，以下 if 语句比较两个整数的大小，并输出较大的数。

```
int varInt1＝1,varInt2＝2;
if(varInt1＞varInt2)
    System.out.println("Max＝"＋varInt1);
```

```
else
    System.out.println("Max="+varInt2);
```

if 语句可以嵌套使用，即在 if 语句中包含另一个 if 语句，也称为嵌套的 if 语句，例如：

```
if(grade>=60)
    if(grade<=80)
        System.out.println("成绩:良");
```

以上嵌套语句相当于以下 if 语句。

```
if(grade>=60&&grade<=80)
    System.out.println("成绩:良");
```

需要注意的是，在两个嵌套的 if 语句中，如果有一个 if 语句省略了 else 子句，则会产生二义性，例如：

```
if(grade>=60)
    if(grade<=80)
        System.out.println("成绩:良");
    else
        ……
```

在这种情况下，很难确定 else 子句与两个 if 语句中的哪一个相对应。因此，在 Java 中规定：else 子句总是与最近的一个 if 语句匹配。所以在上面的程序段中，else 子句是与第二个 if 语句相对应的。如果要和第一个 if 语句对应，可以写成以下形式。

```
if(grade>=60)
{if(grade<=80)
    System.out.println("成绩:良");
    }
else
    ……
```

为了保证良好的编码风格，可以将上面的代码写成下面的形式。

```
if(grade>=60)
{
    if(grade<=80)
        System.out.println("成绩:良");
}
else
    ……
```

【例 2-5】 将 3 个整数按从小到大的顺序输出。

在记事本中输入以下代码。

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

```
        int a=5,b=6,c=4;
        if(a<b)
            if(b<c)
                System.out.println(a+",  "+b+",  "+c);
            else if(a<c)
                System.out.println(a+",  "+c+",  "+b);
            else
                System.out.println(c+",  "+a+",  "+b);
        else if(a<c)
            System.out.println(b+",  "+a+",  "+c);
        else if(c<b)
            System.out.println(c+",  "+b+",  "+a);
        else
            System.out.println(b+",  "+c+",  "+a);
    }
}
```

保存程序,并进行编译和运行。程序的运行结果如图 2-7 所示。

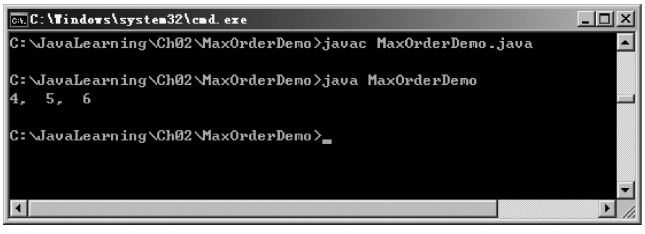

图 2-7　【例 2-5】程序运行结果

(2)switch 语句。if 语句可以很好地解决有两个分支的选择结构,但在实际应用中,还经常遇到多个分支的选择问题。解决这种选择问题的方法除了使用多个 if 语句嵌套外,还可以采用 switch 语句,根据 switch 语句中表达式的取值决定选择哪一个分支。

switch 语句的定义形式如下。

```
switch(<表达式>)
{
    case<常量 1>:<语句 1>;
        break;
    case<常量 2>:<语句 2>;
        break;
   ……
    case<常量 n>:<语句 n>;
        break;
    [default:<语句>;]
}
```

其中，switch、case 和 default 都是关键字，default 子句是可选的。

在执行 switch 语句时，需要将＜表达式＞的值按照从上到下的顺序与 case 语句中给出的常量进行比较，当＜表达式＞的值与某个 case 语句中的常量相同时，就执行该 case 语句后面的语句序列，直到执行到 break 语句或 switch 语句执行完毕；如果在比较的过程中，没有一个常量与＜表达式＞的值相同，则执行 default 后面的语句序列，如果没有default语句，则不执行任何语句。

需要注意的是，switch 语句与 if 语句不同，if 语句的条件语句是布尔表达式，而 switch 语句中的表达式则不能为布尔表达式，可以是 byte、short、int、char 型表达式中的一种，而不能是 float、long 或字符串等其他类型。case 语句中的常量类型要与表达式的类型保持一致，每个 case 语句中的常量值也必须是唯一的。

【例 2-8】 输出星期几。

在记事本中输入以下代码。

```
public class SwitchWeekDemo
{
    public static void main(String[] args)
    {
    int day=2;
        switch(day)
          {
              case 0:System.out.println("今天是周日");
                     break;
              case 1:System.out.println("今天是周一");
                     break;
              case 2:System.out.println("今天是周二");
                     break;
              case 3:System.out.println("今天是周三");
                     break;
              case 4:System.out.println("今天是周四");
                     break;
              case 5:System.out.println("今天是周五");
                     break;
              case 6:System.out.println("今天是周六");
                     break;
          }
    }
}
```

保存程序，并进行编译和运行。程序的运行结果如图 2-8 所示。

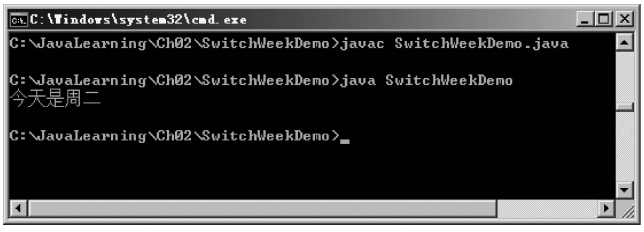

图 2-8　【例 2-8】程序运行结果

3)循环结构

循环结构的作用是反复执行同一段语句直到满足某个结束条件。在实际问题中有很多都需要用循环来解决,如数学中的累加求和、迭代求根等。Java 语言中的循环结构可以用 3 种语句来实现,分别是 while 语句、do-while 语句和 for 语句。

(1)while 语句。while 语句的定义形式如下。

```
while(<布尔表达式>)
{
    <语句>;
}
```

其中,while 是关键字。在执行 while 语句时,先计算<布尔表达式>的值,如果表达式的值为 true,就执行大括号中的语句块(循环体),然后继续计算表达式的值,直到其为 false。

执行 while 语句的循环过程如图 2-9 所示。

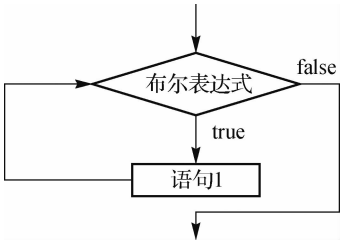

图 2-9　while 循环结构

用 while 语句对自然数 1,2,……,10 求和的程序段如下。

```
int i=1,n=10,sum=0;
while(i<=n)
{
    sum=sum+i;
    i=i+1;
}                    //循环结束后,i=11,sum=55
```

注意:在 while 语句中,如果<布尔表达式>的值永远为 true,那么循环将不会停止,这种无限次循环的现象称为死循环。

如果将上面的程序段稍作修改,就会出现死循环。

```
int i=1,n=10,sum=0;
```

```
while(i<=n)
{
    sum=sum+i;
}
```

不难看出，导致死循环产生的原因是在循环执行的过程中没有改变变量 i 的值。变量 i 的值始终为 1，总是小于变量 n 的值，因此 while 语句中<布尔表达式>的值始终为 true，这就导致了死循环的产生。在程序设计中遇到循环结构时，一定要合理地设计循环条件。

提示：死循环在程序设计中是不允许的，也是要避免的。

(2)do-while 语句。do-while 语句的定义形式如下。

```
do
{
    <语句>;
}while(<布尔表达式>)
```

其中，do 和 while 都是关键字。与 while 语句不同的是，在每次执行循环时，do-while 语句先执行大括号中的循环体，然后再计算<布尔表达式>的值，如果<布尔表达式>的值为 true，就再次执行循环体，直到<布尔表达式>的值为 false。

注意：do-while 语句至少要执行一次循环体。

执行 do-while 语句的循环过程如图 2-10 所示。

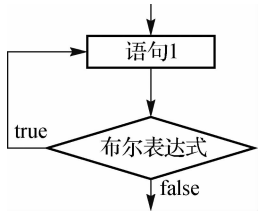

图 2-10　do-while 循环结构

用 do-while 语句对自然数 1,2,……,10 求和的程序段如下。

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

注意：在 do-while 语句中也会因为布尔表达式的值永远为 true 而导致死循环。

(3)for 语句。for 语句提供了更简便、更灵活的方法来处理循环。for 语句的定义形式如下。

```
for(<表达式 1>;<表达式 2>;<表达式 3>)
{
    <语句>;
}
```

其中，for 是关键字，3 个表达式之间以分号(;)隔开。

执行 for 语句的循环过程如图 2-11 所示。

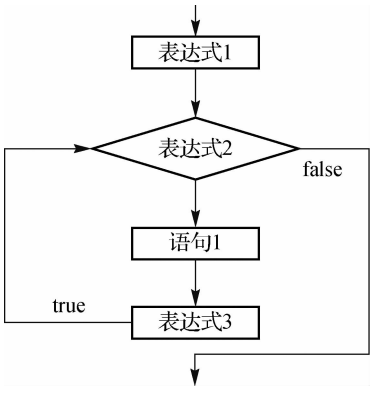

图 2-11　for 循环结构

＜表达式 1＞对循环进行初始化，在循环开始时被执行一次。Java 允许用户在＜表达式 1＞中对一个循环变量进行声明和初始化，如“int i=0”。在＜表达式 1＞中定义的变量只是针对 for 循环的局部变量，在循环结束后，变量将不复存在。如果需要的话，还可以在＜表达式 1＞中声明并初始化多个变量，每个表达式之间用逗号分隔，如“int i=0,int j=0”。

＜表达式 2＞是循环结束的标志，决定了何时结束循环。该表达式必须是一个布尔表达式或能返回布尔值的函数，在每次循环中都要被计算一次。当该表达式的计算结果是 false 时，循环就结束，否则继续执行循环。

＜表达式 3＞表示循环一次循环变量要增加多少，也称为步长或增量，用于逐步将循环的状态引向返回 false 的状态并最终结束循环。

for 语句的执行过程是：先计算＜表达式 1＞的值，再计算＜表达式 2＞的值，如果＜表达式 2＞的值为 true，则执行一次循环体，然后计算＜表达式 3＞的值，再进行下一次循环；如果＜表达式 2＞的值为 false，则结束循环，程序将执行 for 语句后的下一条语句。

【例 2-9】　用 for 语句对自然数 1，2，……，10 求和。

在记事本中输入以下代码。

```
//Sum.java
public class Sum
{
    public static void main(String args[])
    {
        int i=1,n=10,sum=0;
        for(i=1;i<=n;i++)
            sum=sum+i;
            System.out.println("Sum of 1 to 10 is:"+sum);
    }
}
```

保存程序，并进行编译和运行。程序的运行结果如图 2-12 所示。

图 2-12 【例 2-9】程序运行结果

注意：如果将 for 语句中的 3 个表达式都省略，那么 for 循环就成了一个死循环。

(4)多重循环。如果在一个循环语句的循环体中又包含了另一个循环语句，就构成了多重循环结构。多重循环可以是相同循环语句的嵌套，也可以是不同循环语句的嵌套。常见的多重循环是二重循环或三重循环。

以下程序用两个 for 循环语句嵌套的二重循环结构输出九九乘法表。其中，外层 for 循环的循环变量是 i，用于控制打印的行数；内层 for 循环的循环变量是 j，用于控制每行打印的数字个数。

【例 2-10】 输出九九乘法表。

在记事本中输入以下代码。

```
//Multiplication.java
public class Multiplication
{
    public static void main(String args[])
    {
        int i,j,n=9;
        for(i=1;i<=n;i++)
        {
            for(j=1;j<=i;j++)
                System.out.print("  "+i*j);
                System.out.println();  //转到下一行
        }
    }
}
```

保存程序，并进行编译和运行。程序的运行结果如图 2-13 所示。

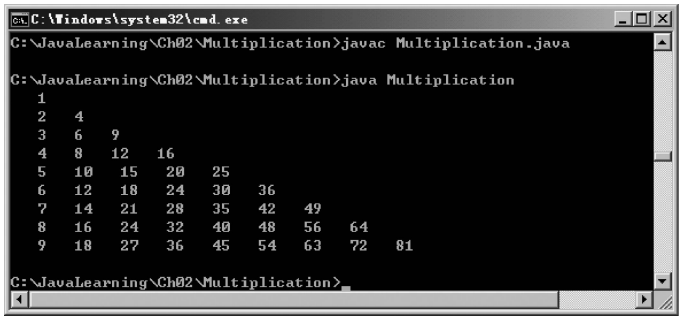

图 2-13　【例 2-10】程序运行结果

4)跳转语句

Java 语言中有 3 种跳转语句,即 break 语句、continue 语句和 return 语句。

(1)break 语句。break 语句的作用是使程序的执行从一个封闭的语句中跳出,如 switch 语句、while 语句、do-while 语句和 for 语句等,并执行该封闭语句后面的语句序列。

break 语句的定义形式如下。

```
break<标号>;
```

其中,break 是关键字;<标号>指定了一个封闭语句的标号,用于指示程序应该从何处开始继续执行。

在没有标号的情况下,嵌套循环中的 break 将跳出最近的循环,进入包含这层循环的外层循环中;单层循环的 break 跳出循环,执行循环体外的下一条语句。使用带标号的 break 语句,可以使程序执行循环外的代码。通常情况下,可以使用 break 语句在 switch 语句中终止某个 case 语句或终止一个循环。

要使用带标号的循环,需要在循环的前面添加标号,在标号和循环之间用冒号分隔,举例如下。

```
endSign:
for(int i=0;i<50;i++)
{
    sum=sum+i;
    if(sum>=200)
      break endSign;
}
```

(2)continue 语句。continue 语句的定义形式如下。

```
continue<标号>;
```

其中,continue 是关键字。continue 语句的作用是跳过循环体内的剩余语句。也就是说,continue 语句并不终止当前循环。在循环体中,程序执行到 continue 语句时就会结束本次循环,回到循环条件,判断是否进行下一次循环操作。

在 continue 语句中,标号的含义和用法与 break 语句中标号的含义和用法是一样的。如下所示。

```
Start_loop:
for(int i=1;i<50;i++)
{
    for(int k=0;k<i;k++)
        if(i%k!=0)
            continue Start_loop;
        System.out.println("i=:"+i)
}
```

(3)return 语句。return 语句的定义形式如下。

```
return 返回值;
```

return 语句使程序从方法中返回一个具体的值。如果在子方法中没有使用 return 语句,则当程序执行完子方法后自动返回主方法。

思考与练习

(1)在 Java 中对标识符的规定有哪些?

(2)Java 中的表达式有哪几类?

(3)试说明几种循环结构的区别。

(4)编写完整的 Java 应用程序,计算出 1 000 以内的水仙花数。所谓水仙花数,是指个位、十位和百位上的 3 个数的平方和等于该数。

(5)编写完整的 Java 应用程序,求任意一个整型数和实型数的和、差、积与商。

第 3 章 Java 与面向对象

在前面的章节中我们学习了如何在 Java 中使用基本的数据类型、控制语句编写简单的 Java 应用程序。但是这些知识在传统的面向过程的编程中也都存在，而 Java 的最大特点是面向对象，本章将介绍 Java 中对象的概念。

所谓对象，就是日常生活中所遇到的各类事物。面向对象是一种编程思想，为了更准确地理解面向对象的概念，须对比之前的面向过程的思想。“面向过程”强调从计算机的处理方式出发设计程序，而“面向对象”则强调从人的思维模式出发，按照人的习惯设计程序。

3.1 面向对象的基本概念

面向对象技术是当前计算机技术发展的一个突破，面向对象程序设计是 Java 语言中最重要的部分，因为任何一个 Java 程序都是由多个类组成的。

面向对象是现实世界模型的自然延伸，可以将现实世界中的任何实体都看作对象，对象之间通过消息相互作用。另外，现实世界中的任何实体都可以归属于某类事物，任何对象都是某一类事物的实例。如果说传统的过程式编程语言是以过程为中心、以算法为驱动的，那么面向对象编程语言则以对象为中心、以消息为驱动。用公式表示，过程式编程语言为：程序＝算法＋数据，面向对象编程语言为：程序＝对象＋消息。

3.2 类和对象

在结构化程序设计中，数据类型是对数据本身特征的描述，而语句是对数据运算和操作的描述，数据和对数据的操作是分开的。而在面向对象程序设计中，数据和对数据的操作被

封装在一个集合体中，即被定义为类。类是用来定义一组对象的共有状态和行为的模板。

类是一种复杂的数据类型，它是将数据和对数据的操作封装在一起的集合体，是普通数据类型的扩展。类不仅包含数据，还包含对数据进行操作的方法，正是这些方法反映了数据的行为。

类中的数据称为成员变量，对数据进行的操作称为成员方法。程序中可以生成一个对象，并用标识符表示，通过对象引用类中的成员变量和成员方法。

对象是类的实例，是某个类的变量。当一个程序运行时，系统为对象分配内存单元，而不是为类分配内存单元。类和对象是密切相关的，类脱离不了对象，对象必须依赖类。

例如，可以把台灯看成一个类，百货商场里的各种台灯就是台灯类的实例，也就是台灯类的对象。每盏台灯的颜色、价格、功率和生产厂家等特征都是台灯类中的数据，即台灯类的属性，而开灯、关灯等行为都是台灯类中的方法，这些方法决定了台灯对象的行为和表现。还可以把人看成一个类，泛指所有的人，这样的泛指是抽象的，因为不可能把所有人的特征集中表现在某个个体上，不同种族的人所表现出来的特征也是不同的。

提示：在 Java 中，所有数据类型都是用类来实现的，类是 Java 的核心内容。

3.2.1 类的声明

在类的声明中，需要定义该类的名称、对该类的访问权限和该类与其他类的关系等。类声明的格式如下。

[<修饰符>] class<类名> [extends<父类名>] [implements<接口名>]

其中，class 是定义类的关键字，<类名>、<父类名>和<接口名>必须是合法的 Java 标识符。在类的声明中，必须包括关键字 class 和自定义的类名，方括号中是可选的项。<修饰符>是修饰类的关键字，用来说明类属于哪个父类、类有哪些访问权限、类是否为抽象类或最终类等。例如，可以定义一个关于 People 的类。

```
public class People
{
    ……
}
```

以上语句定义了一个名为 People 的类，public 是类访问权限的修饰符，说明了 People 类可以被它所在的包之外的类访问和引用。后面将学习访问控制修饰符及其用法。

3.2.2 成员变量

Java 用成员变量来表示类的状态和属性。在成员变量的声明中必须给出变量的名称和该变量的类型，同时还可以指定其他特性。声明成员变量的格式如下。

[<修饰符>] [static] [final] [transient]<变量类型><变量名>;

其中，方括号中是可选项，各项的含义如下。

(1)static 指明了变量是一个类的成员变量。

(2)final 指明了变量是常量。

(3)transient 指明了变量是临时变量。

例如，可以在前面声明的 People 类中定义以下成员变量。

```
String name;
int age;
```

类的成员变量必须在类的主体内声明，而且不能包括在方法体中。成员变量的类型可以是 Java 中的任何一种数据类型，包括基本类型（整型、浮点型和字符型）和引用类型（数组类型和对象）。

3.2.3　成员方法

类的行为由类的成员方法来实现。声明成员方法的格式如下。

```
[<修饰符>]<方法返回值类型><方法名>([<参数列表>])[throws<异常类>]
{
    <方法体>
}
```

在成员方法的声明中，必须给出方法名和方法的返回值类型。如果该成员方法没有返回值，则需要用关键字 void 进行标识。对成员方法进行修饰的修饰符是可选的，成员方法的参数也是可选的，如果没有参数，方法名后的括号不能省略。

方法体是成员方法的实现部分。例如，在 People 类中对成员变量 name 和 age 的赋值操作就是在方法体中进行的。

在方法体中也可以声明变量，但声明的变量是只属于成员方法的局部变量，而不是该类的成员变量。成员变量和局部变量的区别在于：成员变量在整个类内都是有效的，而局部变量只在定义它的成员方法内有效。在下面的例子中可以看出成员变量与局部变量的区别。

```
class classA
{
    int x;
    int setX()
    {
        int a;
        x=a;   //合法
    }
    int setY()
    {
        int y;
        y=a;
        //不合法，因为变量 a 不是成员方法 setY()中的变量，而成员方法 setX()中的
        //变量 a 已经失效
    }
}
```

3.2.4 构造方法

创建对象的格式如下。

类名称 对象名称=new 类名称();

从以上语句可以看出,创建一个对象时是调用了一个与类同名的方法,该方法即构造方法。

在类中除了成员方法之外,还存在一种特殊的方法,即构造方法。构造方法是一个与类同名的方法,一般对象的创建就是通过构造方法完成的。

每次创建一个对象,都需要对该对象中所有的成员变量进行初始化。Java 允许在创建对象时就对其进行初始化,这种初始化方法是通过构造方法来实现的。

构造方法用于在创建对象时为该对象的成员变量赋初值。构造方法的名字与包含它的类名相同,在语法上与成员方法类似,但是构造方法没有返回值,也没有任何修饰符,因为类的构造方法的返回值是该类本身。

Java 中的基本类型也可以用类来实现。例如,可以通过以下方法构造 Integer 类。

```
Integer I=new Integer(10);
Integer I=new Integer("10");
```

在自定义类中,如果要创建类的实例,必须通过构造方法进行实例化。

在构造方法的声明中必须注意以下几点。

(1)构造方法名称必须与类名称一致。

(2)构造方法的声明处不能有任何返回类型。

(3)不能在构造方法中使用关键字 return 返回一个值。

构造方法的声明格式如下。

```
[<修饰符>]<方法名>([<参数列表>])
{
    <方法体>
}
```

以下代码显示了如何声明构造方法。

```
class People
{
    public People()        //声明构造方法
    {
        System.out.println("一个新的 People 对象产生。");
    }
}
```

在构造方法中,一般完成一些初始化的工作。可以为成员变量赋值,这样当实例化一个类的对象时,相应的成员变量就会被初始化。

如果在类中没有显式地定义一个构造方法,则编译器会自动创建一个不带参数的构造方法。如果类中定义了构造方法(带参数或者不带参数的),则编译器不会再创建构造方法。

例如：

```
class People
{
    public People()
    {
    }
}
```

前面说过，构造方法的主要作用是初始化类中的成员变量。既然构造方法也是方法，那么它当然可以传递参数。下面定义一个带参数的构造方法，通过参数初始化类中的成员变量。

```
class People
{
    private String name;
    private int age;
    public People(String n,int a)     //声明构造方法，初始化类中的属性
    {
        name=n;
        age=a;
    }
}
```

构造方法与普通方法一样，也支持方法的重载操作，只要参数的类型或个数不同就可以。以下代码定义了几个重载的构造方法。

```
class People
{
    private String name;
    private int age;
    public People(){}          //声明一个不带参数的构造方法
    public People(String n)   //声明有一个参数的构造方法
    {
        name=n;
    }
    public People(String n,int a)   //声明有两个参数的构造方法
    {
        name=n;
        age=a;
    }
}
```

3.2.5　对象的创建和使用

在 Java 中，创建对象包括声明对象和实例化对象两部分。

1)声明对象

对象是类的实例,属于某个已经声明的类。声明对象的格式如下。

类名 对象名;

对象是在栈内存中声明的,这与数组一样,数组名称就是保存在栈内存中的。但只开辟了栈内存的对象是无法使用的,也就是说,只声明对象是无法使用的,必须有其在堆内存中的引用才可以使用。

例如,声明一个 People 类的对象的代码如下。

```
People ple;
```

2)实例化对象

在 Java 中使用关键字 new 来实例化对象,具体的格式如下。

对象名=new *构造方法名*([*参数*]);

实例化对象实际上就是在堆中开辟内存空间,所有的内容都是对应类型的默认值。如果是引用数据类型,则其值为 null。

例如,实例化上面的 ple 对象的具体代码如下。

```
ple=new People();
```

这个对象的实例化中使用的是无参数的构造方法。

当然,对象的声明与实例化也可由一个语句来完成,即在声明对象的同时实例化对象,具体代码如下。

```
People ple=new People();
```

3)对象的使用

创建对象后,可以通过对象来引用其成员变量,并改变成员变量的值;同时,也可以通过对象来调用其成员方法。在 Java 中通过使用运算符“.”实现对成员变量的访问和对成员方法的调用。

【例 3-1】 对象的使用方法。

在记事本中输入以下代码。

```
class People
{
    String name;          //声明姓名属性
    int age;              //声明年龄属性
    public void ShowInfo()
    { // 取得信息
        System.out.println("姓名:"+name+"\n 年龄:"+age);
    }
}
public class PeopleDemo
{
    public static void main(String args[])
    {
```

```
        People ple=null;          // 声明对象
        ple=new People();         // 实例化对象
        ple.name="如花";          // 为姓名赋值
        ple.age=30;               // 为年龄赋值
        ple.ShowInfo();           // 调用方法,打印信息
    }
}
```

程序的运行结果如图 3-1 所示。

图 3-1　【例 3-1】程序运行结果

3.3　类的封装性

封装、继承和多态性是面向对象技术的核心特性。其中，封装是一种信息隐藏技术；继承是实现类中方法和数据共享的机制；而多态性使得现有系统有更好的可扩展性。本节介绍类的封装性。

3.3.1　封装的概念

如果把类设计成一个黑匣子，则使用者只能看见类中定义的公用方法，看不到方法的实现细节，也不能直接操作类中的数据，这样可以有效地防止外部对类的干扰。如果改变了类中数据的定义，只要方法名不改，就不会对使用该类的程序产生任何影响，这就是封装的含义。也就是说，类的封装可以减少程序对类中数据的依赖性。

通过封装，可以达到以下目的。

(1)隐藏类的实现细节。

(2)要求用户只能通过接口访问数据。

(3)提高代码的可维护性。

3.3.2　访问权限修饰符

在 Java 中是通过设置类的访问权限和类中成员的访问权限来实现封装的，需要用访问控制修饰符设定类和类中成员的访问权限。

在定义类、变量和方法时，所有修饰符被放在语句的最前面，并与类的定义、变量的类型和方法的返回值之间由一个空格分隔。如果有一个以上的修饰符同时修饰某个类、变量或方法，则需要将这些修饰符并列，相互之间也要用空格分隔。示例如下。

[＜修饰符 1＞][＜修饰符 2＞]……class 类名

[＜修饰符 1＞][＜修饰符 2＞]……数据类型 变量名

[＜修饰符 1＞][＜修饰符 2＞]……方法返回值类型 方法(参数列表)

虽然所有修饰符定义的位置都一样，但是它们的含义和作用却不相同。访问控制符的作用是限定类、变量或方法被程序的其他部分访问和调用。具体地说，类和类的变量、方法的访问控制符规定了程序的哪些部分可以访问和调用它们，哪些不能访问和调用。无论修饰符是如何定义的，一个类总是能够访问和调用它自己的变量和方法，但是这个类之外的其他类能不能访问和调用，则要看该类的变量和方法是如何被定义的。通过声明类的访问控制符可以使整个程序结构更加清晰、严谨，也能有效地减少可能产生的错误。

关键字 public、private、protected 和 private protected 是变量和方法的访问控制符，同时关键字 public 是类的访问控制符，也可以不使用任何访问控制符定义变量和方法。

1)公共访问控制符 public

Java 中类的访问控制符是关键字 public，该关键字表示类是公共的。一个类如果被声明为公共的，就表明该类可以被所有其他类访问和引用。也就是说，程序的其他部分可以创建这个类的对象、访问这个类内部可见的成员变量和调用它可见的方法。Java 中的类是通过包的概念来组织的，处于同一个包中的类可以不需要任何说明而方便地互相访问和引用，而对于不同包中的类，如果该类被声明为 public，就可以被其他包中的类访问和引用。

如果一个类中定义了常用的操作，并希望这些操作能够作为公用工具被其他类和程序使用，也应该把类本身和这些方法都定义成 public 类型。Java 的类库中所有的公用类和它们的公用方法就是最典型的例子，每个 Java 程序的主类也都必须是 public 类。

用 public 修饰的类的变量称为公用变量，如果公用变量属于一个公用类，那么这些公用变量就可以被所有其他类引用。但是这样在某种程度上降低了数据的安全性和封装性，因此在程序中一般应该减少公用变量的使用。

2)默认访问控制符

如果一个类没有访问控制符，则说明该类具有默认的访问控制特性。这种默认的访问控制权限规定了该类只能被同一个包中的类进行访问和引用，而不可以被其他包中的类使用。这种访问特性又称为包访问性。

同样地，类的变量和方法如果没有访问控制符限定，也说明它们具有包访问性，可以被同一个包中的其他类访问或调用。包的概念和使用方法将在后面的章节中介绍。

3)私有访问控制符 private

用 private 修饰的变量和方法只能被该类本身访问和修改，而不能被其他任何类(包括该类的子类)访问和引用。private 修饰符用于声明类的私有成员，并提供了最高的保护级别。

4)保护访问控制符 protected

用 protected 修饰的成员变量可以被三种类引用和访问：该类本身、与该类在同一个包

中的其他类和在其他包中的该类的子类。使用 protected 修饰符的主要作用是允许其他包中的子类访问父类的特定变量。

5)私有保护访问控制符 private protected

由 private protected 修饰的成员变量可以被两种类访问和引用:一种是该类本身;另一种是该类的所有子类,而不管这些子类是否与该类在同一个包中。

3.3.3　方法的重载

方法的重载是指在一个类中,方法的名称相同,但是方法的参数个数或参数类型不同。在调用具有重载关系的方法时,通过传递参数的个数及传递参数的类型来区分调用了哪个方法。

【例 3-2】 实现加法函数的重载。

在记事本中输入以下代码。

```
public class OverloadDemo
{
    public static void main(String[] args)
    {
        int one=add(10,20);              // 调用整型数的加法操作
        float two=add(10.3f,13.3f);      // 调用浮点数的加法操作
        int three=add(10,20,30);         // 调用有 3 个参数的加法操作
        System.out.println("add(int x,int y)的计算结果:"+one);
        System.out.println("add(float x,float y)的计算结果:"+two);
        System.out.println("(int x,int y,int z)的计算结果:"+three);
    }
    // 定义方法,完成两个数的相加操作,返回一个整型数据
    public static int add(int x,int y)
    {
        int temp=0;                      // 方法中的参数,是局部变量
        temp=x+y;                        // 执行加法运算
        return temp;                     // 返回计算结果
    }
    //定义方法,完成三个数的相加操作,返回一个整型数据
    public static int add(int x,int y,int z)
    {
        int temp=0;                      // 方法中的参数,是局部变量
        temp=x+y+z;                      // 执行加法运算
        return temp;                     // 返回计算结果
    }
    // 定义方法,完成两个数的相加操作,返回一个 float 型数据
```

```
    public static float add(float x,float y)
    {
        float temp=0;                    // 方法中的参数,是局部变量
        temp=x+y;                        // 执行加法运算
        return temp;                     // 返回计算结果
    }
}
```

程序的运行结果如图 3-2 所示。

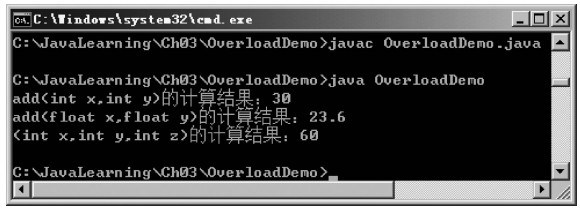

图 3-2 【例 3-2】程序运行结果

提示:方法的重载一定是在参数个数或参数类型上不同。

在下面的代码中,第二个加法函数的参数个数没有变,参数类型也没有变,只是函数体内部的返回值类型由 int 型变为 float 型,因此不是重载。

```
public class Overload2
{
    // 定义方法,完成两个数的相加操作,返回一个 int 型数据
    public static int add(int x,int y)
    {
        int temp=0;      // 方法中的参数,是局部变量
        temp=x+y;        // 执行加法运算
        return temp;     // 返回计算结果
    }
    // 定义方法,完成两个数的相加操作,返回一个 float 型数据
    public static float add(int x,int y)
    {
        float temp=0;    // 方法中的参数,是局部变量
        temp=x+y;        // 执行加法运算
        return temp;     // 返回计算结果
    }
}
```

3.3.4 this 引用

在 Java 中,每个对象都可以对自身进行访问,称为 this 引用。this 引用的规则

如下。

1)指代对象本身

指代对象本身的格式如下。

```
this
```

可以通过 this 引用来指代对象本身。例如,在对象的方法中,用以下语句可以生成对象自身。

```
Object obj=this;
```

2)访问对象自身的成员变量和方法

访问对象自身的成员变量和方法的格式如下。

```
this.<成员变量名>
this.<成员方法名>
```

如果类的成员变量和成员方法中的变量同名,则可以用 this 引用区分二者。以下程序说明了如何使用 this 引用。

【例 3-3】 使用 this 引用。

在记事本中输入以下代码。

```
class People
{  // 定义 People 类
    private String name;  // 姓名
    private int age;      // 年龄
    public People(String name,int age)
    {    // 通过构造方法赋值
        this.name=name; // 为类中的 name 属性赋值
        this.age=age;   // 为类中的 age 属性赋值
    }
    public String getInfo()
    {    // 取得信息的方法
        return "姓名:"+name+"\n 年龄:"+age;
    }
}
public class PeopleInfo
{
    public static void main(String args[])
    {
        People per1=new People("小强",33);          // 调用构造方法实例化对象
        System.out.println(per1.getInfo());             // 取得信息
    }
}
```

程序的运行结果如图 3-3 所示。

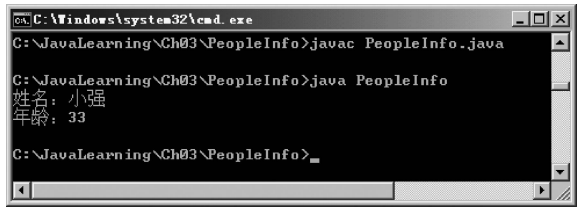

图 3-3 【例 3-3】程序运行结果

3)调用本类的构造方法

如果一个类中有多个构造方法,也可以使用关键字 this 相互调用。格式如下。

this(<参数列表>)

假设一个类中有多个构造方法,而且每个构造方法都要打印一句“新对象实例化”,可以使用以下代码来实现。

```
class People
{  // 定义 People 类
    private String name;  // 姓名
    private int age;      // 年龄
    public People()
    {  // 无参构造方法
        System.out.println("新对象实例化");
    }
    public People(String name)
    {
        this.name=name;
        this();              // 调用本类中的无参构造方法
    }
    public People(String name,int age)
    {    // 通过构造方法赋值
        this(name);          // 调用有一个参数的构造方法
        this.age=age;        // 为类中的 age 属性赋值
    }
    public String getInfo()
    {   //取得信息的方法
        this();              // 调用本类中的无参构造方法
        return "姓名:"+name+",年龄:"+age;
    }
}
public class ThisDemo02
```

```
{
    public static void main(String args[])
    {
        Person per1=new Person("张三",33);  // 调用构造方法实例化对象
        System.out.println(per1.getInfo()); // 取得信息
    }
}
```

提示:在使用关键字 this 调用其他构造方法时,有以下两点限制。

(1)使用 this()调用构造方法的语句只能放在构造方法的首行。

(2)在使用 this()调用本类中其他构造方法时,至少有一个构造方法是不用 this()调用的。

3.4　类的继承性

在编写大型应用程序时,如果定义的类已经被使用,那么如何才能增加程序的可维护性? 既要保证程序在修改后不影响原来的使用效果,又要对程序现有的功能进行扩充,一般可以采取以下两种方法。

(1)直接修改源代码。

(2)对原来类的定义进行复制。

但以上两种方法都不能很好地解决问题,因为在直接修改源代码的同时可能会影响其他类,对原来类的定义进行复制的同时也可能复制了原来存在的错误。最好的解决办法是使用面向对象的类的继承机制,通过对类的继承,扩充旧的程序适应新的需求,这能有效地节省程序的开发时间,还可以提高程序代码的可重用性。

3.4.1　继承的概念

继承是一种由已有的类创建新类的机制,利用继承,可以创建一个具有特殊属性的新类。新类继承了已有类的状态和行为,并可以根据需要增加自己的状态和行为。由继承得到的类称为子类,被继承的类称为父类。Java 语言规定,子类只能有一个父类,即不允许多重继承。

例如,当类 a1 继承类 a 时,就表明 a1 是 a 的子类,a 是 a1 的父类。子类 a1 从父类 a 继承了成员变量和方法,因此可以共享数据和方法。类 a1 的主体中包含了继承部分和增加部分,继承部分是从类 a 继承而来,增加部分是为类 a1 编写的新代码。

继承性是软件重用的一种形式,也是自动实现类中方法和数据共享的机制。如果没有继承,一个系统中的类是封闭的、相互无关的,可能用多个类来实现相似的功能,这就造成了数据和方法的大量重复。引入继承机制,多个类就可以相互关联,新的类由已有的类创建,通过保留父类的属性和行为,并为新类增加新的属性和行为,从而提高了软件的可重用性,

缩短了软件的开发时间。

3.4.2 子类的定义

Java 中的每个类都是 Object 类的子类，也就是说，每个类都有父类。在声明类的同时可以说明类的父类，如果没有显式地标出父类，则隐含地假设父类是在语言包 java.lang 中说明的 Object 类。

可以在声明类的同时使用 extends 关键字显式地指明父类，声明的格式如下。

[＜修饰符＞] class＜子类名＞ extends＜父类名＞

例如，可以用以下语句指明 a1 继承了其父类 a 的变量和方法。

```
class a1 extends a
```

子类可以继承父类中所有可以被子类访问的成员变量，但必须遵循以下规则。

(1)子类能够继承父类中被声明为 public 和 protected 的成员变量。

(2)子类能够继承在同一个包中由默认修饰符修饰的成员变量。

(3)子类不能继承父类中被声明为 private 的成员变量。

(4)如果子类声明了一个与父类成员变量同名的成员变量，则子类不能继承父类的成员变量，此时称子类的成员变量覆盖了父类的成员变量。

子类可以继承父类中由 public、protected 和默认修饰符修饰的变量，不能继承由 private 修饰变量，这体现了子类封装的信息隐蔽原则。

与继承父类中的成员变量类似，子类继承父类中所有可被子类访问的成员方法也需要遵循以下规则。

(1)子类能够继承父类中被声明为 public 和 protected 的成员方法。

(2)子类能够继承在同一个包中的由默认修饰符修饰的成员方法。

(3)子类不能继承父类中被声明为 private 的成员方法。

(4)子类不能继承父类的构造方法。

(5)如果子类的方法与父类中的方法同名，则子类不能继承父类中的方法，此时称子类方法覆盖了父类中的方法。

可以使父类中的一些方法在子类中继续使用。这样一来，如果子类有一些重复的方法，就不用重新定义了。以下代码说明了定义子类的方法。

```
class People
{ // 定义 People 类
    private String name;          // 定义 name 属性
    private int age;              // 定义 age 属性
    public void setName(String name)
    {
        this.name=name;
    }
    public void setAge(int age)
    {
```

```
        this.age=age;
    }
    public String getName()
    {
        return this.name;
    }
    public int getAge()
    {
        return this.age;
    }
}
class Student extends People
{    // 定义 Student 类
    //此处不进行操作
}
public class ExtendsDemo
{
    public static void main(String args[])
    {
        Student stu=new Student();      // 实例化子类对象
        stu.setName("小李");            // 此方法在 Student 类中没有明确定义
        stu.setAge(30);
        System.out.println("姓名:"+stu.getName()+"\n 年龄:"+stu.getAge());
    }
}
```

需要特别说明的是,子类除了可以继承父类中的成员变量和成员方法外,还可以增加自己的成员。当一个父类的成员不适合子类时,子类就应该以合适的方法来重新定义它。以下程序显示了在子类中定义成员的方法。

【例 3-4】 在子类中定义成员。

在记事本中输入以下代码。

```
class People
{ // 定义 People 类
    private String name;          // 定义 name 属性
    private int age;              // 定义 age 属性
    public void setName(String name)
    {
        this.name=name;
    }
    public void setAge(int age)
```

```
    {
        this.age=age;
    }
    public String getName()
    {
        return this.name;
    }
    public int getAge()
    {
        return this.age;
    }
}
class Student extends People
{ // 定义 Student 类
    private String school;          // 定义 school 属性
    public void setSchool(String school)
    {
        this.school=school;
    }
    public String getSchool()
    {
        return this.school;
    }
}
public class ExtendsDemo
{
    public static void main(String arsg[])
    {
        Student stu=new Student();  // 实例化子类对象
        stu.setName("小李");          // 此方法在 Student 类中没有明确定义
        stu.setAge(30);
        stu.setSchool("乐园小学");
        System.out.println("姓名:"+stu.getName()+"\n 年龄:"+stu.getAge()+
        "\n 学校:" +stu.getSchool());
    }
}
```

保存程序,并进行编译和运行。程序的运行结果如图 3-4 所示。

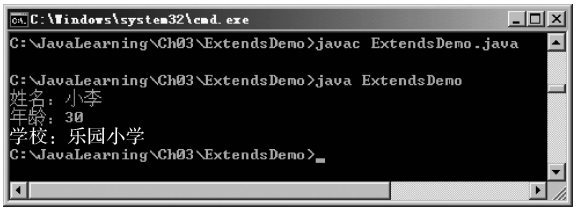

图 3-4　【例 3-4】程序运行结果

3.4.3　super 引用

对于那些被子类隐藏的父类的成员变量和成员方法，可以使用关键字 super 来引用，称为 super 引用。super 引用的规则如下。

1)访问被子类隐藏的父类的成员变量和成员方法

格式如下。

super.＜成员变量名＞

super.＜成员方法名＞

2)调用父类的构造方法

格式如下。

super(＜参数列表＞)

在子类实例化时系统会默认调用父类中的无参构造函数，如果希望调用有参构造函数，则必须在子类中明确声明。

【例 3-5】 使用 super 引用。

在记事本中输入以下代码。

```
class People
{// 定义 People 类
    private String name;  // 定义 name 属性
    private int age;      // 定义 age 属性
    public People(String name,int age)
    {
        this.setName(name);
        this.setAge(age);
    }
    public void setName(String name)
    {
        this.name=name;
    }
    public void setAge(int age)
    {
```

```
        this.age=age;
    }
    public String getName()
    {
        return this.name;
    }
    public int getAge()
    {
        return this.age;
    }
    public String getInfo()
    {
        return "姓名:"+this.getName()+"\n 年龄:"+this.getAge();
    }
}
class Student extends People
{    // 定义 Student 类
    private String school;  // 定义 school 属性
    public Student(String name,int age,String school)
    {
        super(name,age);     // 明确调用父类中有两个参数的构造方法
        this.school=school;
    }
    public void setSchool(String school)
    {
        this.school=school;
    }
    public String getSchool()
    {
        return this.school;
    }
    public String getInfo()
    {
        return super.getInfo()+"\n 学校:"+this.getSchool();
    }
}
```

```
public class SuperDemo
{
    public static void main(String arsg[])
    {
        Student stu=new Student("张三",30,"清华大学");  // 实例化子类对象
        System.out.println(stu.getInfo());
    }
}
```

保存程序,并进行编译和运行。程序的运行结果如图 3-5 所示。

图 3-5　【例 3-5】程序运行结果

3.4.4　抽象类和最终类

在 Java 中有两种特殊的类,分别是抽象类和最终类。

1)抽象类

有时需要创建一个类,这个类只定义一个为所有子类共享的一般形式,至于细节则交由每个子类去实现。这种类非常抽象,没有具体的实例,因此将其称为抽象类。

所谓抽象类就是只声明方法的存在而不实现方法。抽象类不能被实例化,即不能创建其对象。在定义抽象类时,要在关键字 class 前加上关键字 abstract,具体格式如下。

abstract class 类名

抽象类的子类必须实现父类中所有的抽象方法,或将自己也声明为抽象类。

在抽象类中创建的那些没有实际意义但必须由子类重写的方法称为抽象方法。抽象方法是通过指定 abstract 类型创建的。抽象方法没有内容,因此无须被执行,但是子类必须重写该方法。换句话说,如果某个成员方法被声明为抽象方法,那么该成员方法必须被子类的方法所覆盖。

声明抽象方法的格式如下。

abstract 成员方法名(<参数列表>)

提示:构造方法不能被声明为抽象的,修饰符 static 不能与关键字 abstract 同时存在。

任何包含了抽象方法的类必须被声明为抽象类。

2)最终类

最终类是指不能被继承的类,也就是说,最终类不可以有子类。如果不希望某个类被继承,可以在程序中将该类声明为最终类。

关键字 final 用来声明最终类,声明格式如下。

final class 类名

如果某个类没有被声明为最终类，但又想保护类中的成员方法不被覆盖，则可以在成员方法的声明前用关键字 final 修饰。用 final 修饰的成员方法称为最终方法。

被定义为最终类的类通常都是一些有固定作用、用来完成某种标准功能的类，如在 Java 中预先定义好的用来实现网络功能的 InetAddress 类、Socket 类等。当通过类名引用一个类或对象时，实际上有可能引用的是这个类，也有可能引用的是这个对象，还有可能引用的是这个类的某个子类或子类的对象。而将类定义为最终类，就可以固定它的属性和功能，与类名形成稳定的对应关系，从而保证引用这个类时能正确地实现其功能。

一般来说，一个类不能既是最终类又是抽象类，修饰符 public 或 private 放在关键字 final 或 abstract 的前面。但是关键字 abstract 和 final 可以各自与其他修饰符同时使用，修饰符之间的先后排列顺序对类的性质没有影响。

3.4.5 内部类

简单地说，内部类就是在一个类内部的类。

在类的内部也可以定义类。如果在类 Outer 的内部再定义一个类 Inner，则 Inner 类称为内部类，Outer 类称为外部类。

内部类的定义格式如下。

```
<修饰符> class<外部类名>
{
    <修饰符> class<内部类名>
    {
    }
}
```

以下代码显示了在应用程序中使用内部类的方法。

```
class Outer
{  // 定义外部类
    private String info="hello world";        // 定义外部类的私有属性
    class Inner
    {  // 定义内部类
        public void print()
            {// 定义内部类的方法
                System.out.println(info);  // 直接访问外部类的私有属性
            }
    }
        public void fun()
        {           // 定义外部类的方法
            new Inner().print();  // 通过内部类的实例化对象调用方法
        }
```

```
}
public class InnerClassDemo01
{
    public static void main(String args[])
    {
        new Outer().fun();  // 调用外部类的 fun()方法
    }
}
```

在上述程序中，Inner 类作为 Outer 类的内部类存在，并在外部类的方法 fun()中实例化内部类对象，且调用方法 print()。

一个类的内部类除了可以在外部类中访问外，也可以在其他类中调用，调用的格式如下。

外部类.内部类 内部类对象名＝外部类实例.new 内部类()

以下代码显示了在其他类中调用内部类的方法。

```
class Outer
{  // 定义外部类
    private String info="hello world";     // 定义外部类的私有属性
    class Inner
    {                                      // 定义内部类
        public void print()
        {                                  // 定义内部类的方法
            System.out.println(info);      // 直接访问外部类的私有属性
        }
    }
    public void fun()
    {                                      // 定义外部类的方法
        new Inner().print();               // 通过内部类的实例化对象调用方法
    }
}
public class InnerClassDemo02
{
    public static void main(String args[])
    {
        Outer out=new Outer();             // 外部类实例化对象
        Outer.Inner in=out.new Inner();    // 实例化内部类对象
        in.print();                        // 调用内部类的方法
    }
}
```

3.5 接　　口

接口定义了若干抽象方法和常量，形成了一个属性集合。该属性集合往往对应某种功能，其主要作用是实现类的多重继承。

Java 只支持单重继承机制，不支持多重继承。但在实际应用中，常需要多重继承来解决问题。所谓多重继承，是指一个子类可以有一个以上的父类，该子类可以继承所有父类的成员。Java 虽然不支持多重继承，但提供了接口，从而实现类的多重继承，即通过一个类实现多个接口，从而实现多重继承的功能。

3.5.1 接口的特点

接口是 Java 中的重要概念，可以将其理解为一种特殊的类，由全局常量和公共的抽象方法组成。与声明类的格式相似，接口的定义格式如下。

```
<修饰符> interface<接口名>[extends 父接口名列表]
{
    [public][static][final] 常量;
    [public][abstract] 方法;
}
```

其中，<修饰符>可以是 public，也可以省略。在省略时，接口使用默认的访问控制，此时接口只能被与它在同一个包中的成员访问；当修饰符为 public 时，接口能被任何类的成员访问。

接口中的方法只有定义没有被实现，即接口中的方法都是抽象方法，因此接口实际上是一种特殊的抽象类。定义在接口中的常量全部隐含为 final 和 static 类型，这表明它们不能被实现接口方法的类改变，这些常量必须设置初值。如果接口声明为public，则接口中的方法和常量全部为 public。

接口也有继承性。可以在定义接口时通过关键字 extends 声明该接口是某个已经存在的父接口的派生接口，子接口将继承父接口的所有属性和方法。Java 允许接口有多个父接口，它们之间用逗号分隔，形成父接口列表。

以下代码声明了一个接口。如果将接口的声明单独存储在文件中，则必须确保源程序的文件名与接口名相同。

```
interface Student_info
{
    String strDept="计算机系";
    String getDept();
    void output();
}
```

3.5.2　接口的实现

接口的声明只是给出了抽象方法，在声明了接口后，类就可以实现这个接口。类必须实现定义在接口中的方法，每个类能够自由决定方法的具体实现细节，这称为接口的实现。

接口的实现需要用到关键字 implements，格式如下。

[＜修饰符＞] class＜类名＞ [extends＜父类名＞] [implements＜接口名＞]

其中，＜修饰符＞可以是 public，也可以省略。如果一个类实现一个接口，就必须实现接口中的所有方法，而且方法必须声明为 public；如果一个类实现多个接口，需要在关键字 implements后面用逗号连接多个接口名。

在实现接口时，需要注意以下几点。

(1)在类的声明部分用关键字 implements 指明该类要实现哪些接口。

(2)如果实现接口的类不是抽象类，则在类的定义部分必须实现指定接口的所有抽象方法。也就是说，要为所有的抽象方法定义方法体，而且要求方法的返回值和参数列表完全一致。

(3)如果实现接口的类是抽象类，则该类不需要实现接口中所有的方法，但对于该类的任何非抽象子类，在它们的父类所实现的接口中，所有的抽象方法都必须有方法体。这表明在非抽象类中不能存在抽象方法。

(4)类在实现接口的抽象方法时，必须使用完全相同的方法定义(包括返回值和参数列表)。如果所实现的方法与抽象方法只是同名而参数列表不同，则只是对方法进行重载，而不是对已有抽象方法的实现。

(5)由于系统指定 public 为接口的抽象方法的修饰符，所以类在实现方法时，必须显式地使用 public 修饰符，否则编译时系统会提示缩小了接口中定义的方法的访问控制范围。

在以下程序中，声明的类 Student 实现了接口 Student_info。

【例 3-6】　接口的实现。

在记事本中输入以下代码。

```
class People
{
    protected String strName;          // 姓名
    protected int nAge;                // 年龄
    protected int nSex;                // 性别,1 表示男性,0 表示女性
    public People(String name,int age,int sex)
    {
        strName=name;
        nAge=age;
        nSex=sex;
    }
    public String getName()
    {
```

```
        return strName;
    }
    public int getAge()
    {
        return nAge;
    }
    public int getSex()
    {
        return nSex;
    }
    public void setName(String name)
    {
        strName=name;
    }
    public void setAge(int age)
    {
        nAge=age;
    }
    public void setSex(int sex)
    {
        nSex=sex;
    }
    public void print()
    {
        System.out.println("我的名字是\""+strName+"\""+",我今年"+nAge+"岁");
    }
}
interface Student_info
{
    String strDept="软件工程学院";
    String getDept();
    void output();
}
class Student extends People implements Student_info
{
    String strNum;
    public Student(String name,int age,int sex,String num)
    {
        super(name,age,sex);
```

```
        strNum=num;
    }
    public String getDept()
    {
        return strDept;
    }
    public void output()
    {
        System.out.println("我是"+strDept+",我的学号是"+strNum);
    }
}
public class InterfaceDemo
{
    public static void main(String args[])
    {
        Student s=new Student("盼盼",22,1,"0912604");
        s.print();
        s.output();
    }
}
```

保存程序,并进行编译和运行。程序的运行结果如图 3-6 所示。

图 3-6　【例 3-6】程序运行结果

Java 还允许一个类实现多个接口,从而实现多重继承。以下程序演示了如何实现多重继承,类 Student 同时实现了接口 Student_info1 和 Student_info2。

【例 3-7】　多重继承的接口实现。

在记事本中输入以下代码。

```
class People
{
    protected String strName;                    // 姓名
    protected int nSex;                          // 性别,1 表示男性,0 表示女性
    public People(String name,int sex)
    {
        strName=name;
```

```
        nSex=sex;
    }
    public String getName()
    {
        return strName;
    }
    public int getSex()
    {
        return nSex;
    }
    public void setName(String name)
    {
        strName=name;
    }
    public void setSex(int sex)
    {
        nSex=sex;
    }
    public void print()
    {
        System.out.println("我的名字叫\""+strName+"\"");
    }
}
interface Student_info1
{
    int nAge=21;
    int getAge();
    void printAge();
}
interface Student_info2
{
    String strDept="软件工程学院";
    String getDept();
    void printDept();
}
class Student extends People implements Student_info1,Student_info2
{
    String strNum;
    public Student(String name,int sex,String num)
```

```
    {
        super(name,sex);
        strNum=num;
    }
    public int getAge()
    {
        return nAge;
    }
    public void printAge()
    {
        System.out.println("我今年"+nAge+"岁");
    }
    public String getDept()
    {
        return strDept;
    }
    public void printDept()
    {
        System.out.println("我是"+strDept+",我的学号是"+strNum);
    }
}
public class MultiInterface
{
    public static void main(String args[])
    {
        Student s=new Student("盼盼",1,"0099611");
        s.print();
        s.printAge();
        s.printDept();
    }
}
```

保存程序,并进行编译和运行。程序的运行结果如图 3-7 所示。

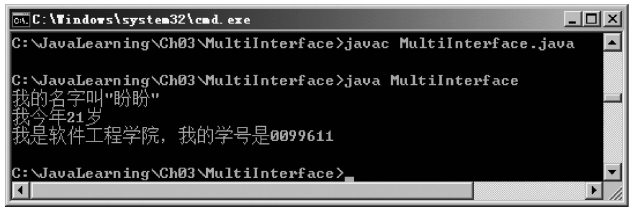

图 3-7　【例 3-7】程序运行结果

3.6 对象的多态性

对象的多态性主要分为两种类型，即向上转型和向下转型。

向上转型是指用子类对象实例化父类。对象的向上转型是自动完成的，格式如下。

父类 父类对象＝子类实例；

向下转型是指用父类对象实例化子类。向下转型时，必须明确指明要转型的子类类型，具体格式如下。

子类 子类对象＝(子类)父类实例；

以下代码显示如何通过子类进行父类对象的实例化操作。

```
class A{                          // 定义类 A
    public void fun1(){           // 定义方法 fun1()
        System.out.println("A-->public void fun1(){}");
    }
    public void fun2(){
        this.fun1();              // 调用方法 fun1()
    }
}
class B extends A{
    public void fun1(){           // 此方法被子类覆写了
        System.out.println("B-->public void fun1(){}");
    }
    public void fun3(){
        System.out.println("B--> public void fun3(){}");
    }
}
public class PolDemo01{
    public static void main(String args[]){
        B b=new B();              // 实例化子类对象
        A a=b;                    // 向上转型关系
        a.fun1();                 // 此方法被子类覆写过
        a.fun2();
    }
}
```

提示：转型之后，因为操作的是父类对象，所以无法调用在子类中定义的新方法。

以下程序显示了父类对象与子类对象之间的转型。

【例 3-8】 父类对象与子类对象之间的转型。

在记事本中输入以下代码。

```
class A
{ // 定义类 A
    public void fun1()
    { // 定义方法 fun1()
        System.out.println("A--> public void fun1(){}");
    }
    public void fun2()
    {
        this.fun1();     //调用方法 fun1()
    }
}
class B extends A
{
    public void fun1()
    { // 此方法被子类覆写了
        System.out.println("B--> public void fun1(){}");
    }
    public void fun3()
    {
        System.out.println("B--> public void fun3(){}");
    }
}
public class InheritDemo
{
    public static void main(String args[])
    {
        A a=new B();     // 向上转型关系
        B b=(B)a;        // 发生了向下转型关系
        b.fun1();
        b.fun2();
        b.fun3();
    }
}
```

保存文件，并进行编译和运行。程序的运行结果如图 3-8 所示。

图 3-8 【例 3-8】程序运行结果

3.7 包

Java 语言要求文件名与类名相同，所以当将多个类放在一起时，要保证类名不重复。当源程序中声明的类很多时，发生类名冲突的可能性就比较大，因此需要一种机制来管理这些类的名称，这就是 Java 中的包。

包是 Java 提供的一种区别于类的名字空间的机制，是类的组织方式，对应于一个文件夹。包中可以再有包，这称为包的等级。

在源程序中可以声明类所在的包，同一个包中的类名不能重复，但不同包中的类名可以相同。当源程序中没有声明类所在的包时，Java 将类封装在默认的包中，这意味着每个类必须使用唯一的名字，否则会发生名字冲突。

3.7.1 包的概念

包是在使用多个接口或类时，为了避免名称重复而采用的一种措施，在程序中通过关键字 package 定义包。

包的定义格式如下。

```
package 包名.子包名;
```

举例如下。

```
package nk.cs;                    // 定义一个包
class Demo{
    public String getInfo(){
        return "Hello World!!!";
    }
}
public class PackageDemo01{
    public static void main(String args[]){
        System.out.println(new Demo().getInfo());
    }
}
```

定义包之后，类的全名就是“包.类名称”，如 nk.cs.Demo。

包就是一个文件夹，一个 class 文件要保存在一个文件夹中。在 Java 的编译指令中提供了专门的打包编译命令，如下所示。

```
javac -d . PackageDemo01.java
```

3.7.2　使用包中的类

当一个包中的 class 文件需要使用另外一个包中的 class 文件时，就需要使用导入命令。

import 语句的格式如下。

```
import <包名 1>[.<包名 2>……].<类名>.*;
```

其中，import 是关键字，多个包名和类名之间用“.”分隔，“*”表示包中所有的类。例如：

```
import java.io.*;          //导入输入/输出包
import java.net.*;         //导入网络包
```

Java 中的 java.lang 包是系统自动默认导入的，因此无须在源程序中导入就能直接使用其中的类。

思考与练习

(1)什么是类？什么是对象？对象与类之间是什么关系？

(2)Java 语言设置了几种类成员的访问权限？分别用什么访问控制修饰符？

(3)什么是类的封装？对类进行封装有什么作用？

(4)什么是继承？什么是多态性？

(5)什么是方法的重载？什么是方法的覆盖？

(6)什么是接口？怎样实现接口？

(7)Java 中有哪些常用的包？如何在程序中使用 Java 已经定义好的类？