Java语言程序设计

前　言

Java 作为在 Internet 上最流行的编程语言，其发展非常迅速。从 1995 年诞生以来，经过短短的几年发展，如今它已不仅仅是一门语言，而已发展为一门技术，包括 Java 的芯片技术、Java 的编译技术、Java 的数据库连接技术、基于 Java 的信息家电的联网技术、企业信息服务的综合求解方案技术等等。

Java 语言作为一种优秀的面向对象的程序设计语言，具有平台无关性、安全机制、高可靠性和内嵌的网络支持等特点。由于 Java 语言的这些特点，使得 Java 语言成为当前网络应用程序编写的首选工具之一。还有人预言，不久的将来，全世界 90%的程序代码将用 Java 语言重写和改写。Java 语言和技术的大量使用也促使 Java 语言本身不断发展。Java 语言的发明者美国 Sun 公司在 2002 年 2 月推出了 Java 开发工具的最新版本 Java SDK 1.4.0 版，供使用者免费下载使用。

本书是面向高职专科学生及其他对 Java 语言和面向对象编程技术感兴趣的读者的。书中对内容的编排、取舍和例题、习题的选择，保证了一定的新颖性和深度、广度，在编写方法上注意遵循由浅入深、循序渐进、难点分散的原则。本书可作为高职类同名课程的教材，也可作为一般读者的自学用书。学习本书前应对计算机有一定的认识，最好了解 DOS、Windows 等系统的基础操作。可以将本书作为第一门开发语言来学习，获得开发程序的初步经验。对有其他高级语言编程经验的读者，学习本书也会感到得心应手，从中领略 Java 语言面向对象、易学易用的特点。

本书第 1 章概要介绍了面向对象的一些基本概念，引出两种 Java 程序的介绍，并简述了 Java 程序的结构和开发过程。第 2 章和第 3 章介绍 Java 语言编程的基础知识，包括一般程序设计语言的大部分内容，有基本数据类型、常量、变量、运算符、表达式和流程控制语句、方法（函数）等，还介绍了 Java 语言的异常处理，初步涉及面向对象的程序设计技术。第 4 章介绍 Java 语言的一维数组、二维数组的定义和使用。第 5 章介绍面向对象设计的主要内容，包括类、继承、接口和包等。第 6 章介绍 Java 语言的字符串，包括大量的字符串处理方法和 main()方法参数的使用等。第 7 章介绍 Java 语言的流处理，包括面向字节和面向字符的两种流。第 8 章介绍 Java Applet，同时也介绍了相关的 HTML 和图形界面的 AWT 绘图等内容。第 9 章介绍 Java GUI 程序设计，包括布局、观感、Swing 组件和事件处理等。第 10 章介绍 Java 的多线程程序设计，包括线程的概念、多线程的实现方法、线程的生命周期、线程的控制、线程的异步和同步等。第 11 章介绍 Java 语言的多媒体技术，包括在 Applet 和 Application 中显示图像、制作动画和播放声音等。第 12 章介绍一个简单实用的 Java 应用程序的例子：Java 支持的图像文件的演示和声音文件的播放应用程序。它应用了 GUI 的程序设计技术，包括多种 Swing 组件、事件处理、文件打开对话框、信息对话框等。第 13 章介绍本课程的实验内容与安排。在课时不足时，可暂不学标有“*”的章节。

计算机语言的学习应是课堂学习和上机实验的有机结合，特别要重视上机实验的环节。优秀的软件开发人员都有大量上机编程的经验，从实践中可学到很多书本上没有的东西。读者除在学校安排固定实验时间之外，还要利用更多的其他实验机会。只要注重实验，边学边

练，Java 语言程序设计入门是不难的。

本书的出版与所有帮助作者完成本书的领导、同事和家人的支持分不开，在此表示衷心感谢。特别要感谢浙江大学陈海燕老师审阅了全书，江汉大学乔维声老师大力的支持和帮助。虽然作者在编写本书时尽了最大努力，但书中仍难免有疏漏错误之处，欢迎广大读者、专家批评指正。

编　者

2002 年 3 月

第 1 章　Java 语言概述

1.1　Java 语言的发展和特点

1.1.1　Java 语言的发展

1990 年，美国 Sun 公司的 James Gosling、Bill Joe 等人，为在电视、控制烤箱等家用消费类电子产品上进行交互式操作而开发了一种与平台无关、可靠性强、小而灵活的编程语言，但当时并没有引起人们的注意。直到 1994 年下半年，Internet 的迅猛发展，环球信息网 WWW 的快速增长，人们才发现 Java 这种中性平台及可靠性强的语言恰恰就是全球信息网在等待的语言。Java 的开发人员基于网络对 Java 进行了一系列的改进，融合了 C 和 C++等语言的优点，形成了现在这套与众不同的面向对象的通用程序设计语言。

Java 的原名叫 Oak（橡树），但在申请注册商标时，发现 Oak 已经有人用了。在想了一系列名字后，最终，使用了提议者在喝一杯 Java 咖啡时无意提到的 Java。

Java（JDK 1.0）正式发表于 1995 年 5 月。Java 的“Write Once,Run Anywhere”口号使得 Java 一出现就引起广泛的注意，用 Java 开发的软件可以不用修改或重新编译而直接应用于任何计算机上。Java 语言的众多优点使得它逐渐成为 Internet 上受欢迎的开发与编程语言。Java 的诞生对传统的计算模型提出了新的挑战。

Java 语言有着广泛的应用前景，可以从以下几个方面来考虑其应用：

（1）所有面向对象的应用开发，包括面向对象的事件描述、处理、综合等；

（2）计算过程的可视化、可操作化的软件的开发；

（3）动态画面的设计，包括图形图像的调用；

（4）交互操作的设计（选择交互、定向交互、控制流程等）；

（5）Internet 的系统管理功能模块的设计，包括 Web 页面的动态设计、管理和交互操作设计等；

（6）Intranet（企业内部网）上的软件开发（直接面向企业内部用户的软件）；

（7）与各类数据库连接查询的 SQL 语句实现；

（8）其他应用类型的程序。

1.1.2　Java 语言的特点

Java 是一门迅速发展的网络编程语言，它是一种新的计算概念。

首先，作为一种程序设计语言，它简单，面向对象，分布式，解释执行不依赖于机器的结构，具有可移植性、鲁棒性、安全性，并且是多线程的、动态的，具有很高的性能。

其次，Java 最大限度地利用了网络。一种被称为 Java 小程序（Applet）的 Java 程序是动态、安全、跨平台的网络应用程序，可在网络上运行而不受 CPU 和环境的限制。Java Applet 嵌入 HTML 语言中，通过 Web 页发布到 Internet。网络用户访问服务器的 Applet 时，这些

Applet 从网络上进行传输，然后在支持 Java 的浏览器中运行。由于 Java 语言的安全机制，用户一旦载入 Applet，就可以放心地生成多媒体的用户界面或完成复杂的计算而不必担心病毒的入侵。虽然 Applet 可以和图像、声音、动画等一样从网络上下载，但它并不同于这些多媒体文件格式，它可以接收用户的输入，动态地进行改变，而不仅仅是动画的显示和声音的播放。

另外，Java 还提供了丰富的类库，以满足网络化、多线程、面向对象系统的需要，使程序设计者可以很方便地建立自己的系统。

（1）语言包提供的支持包括字符串处理、多线程处理、异常处理、数学函数处理等，可以用它简单地实现 Java 程序的运行平台。

（2）实用程序包提供的支持包括哈希表、堆栈、可变数组、时间和日期等。

（3）输入输出包用统一的“流”模型来实现所有格式的输入/输出，包括文件系统、网络、输入/输出设备等。

（4）低级网络包用于实现 Socket 编程。

（5）抽象图形用户接口包实现了不同平台计算机的图形用户接口组件，包括窗口、菜单、滚动条、对话框等，使得 Java 可以移植到不同平台的机器中去。

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

1.1.3 Java 程序的工作机制

学习 Java 语言，有必要了解 Java 的工作机制，这将更有助于理解 Java 语言的特点。

对于运行在 Internet 上的网络应用程序，需要有良好的可移植性。因为 Internet 是由各种各样不同类型的终端、服务器和 PC 等硬件设备组成的，而且在这些设备上运行的软件系统也是多种多样的，所以 Internet 上的网络应用程序应该具有在各种不同的软硬件平台上均可正常工作的能力。Java 的工作机制使得它具有了这样的能力。

Java 的工作机制是这样的：编程人员首先编写好源代码，然后经编译生成一种二进制的中间码，称为字节码（byte code），最后再通过运行与操作系统平台环境相对应的一种称为 Java 解释器的运行机构来执行编译生成的字节码。虽然不同的平台环境需要有各自相应的解释器，但是任何一个平台上的解释器，对于一段 Java 程序的字节码来说却是相同的，它们对 Java 字节码呈现出完全相同的面貌。也就是说，Java 的运行机制是利用解释器来隐藏网络上平台环境的差异性的。由此可见，Java 实现了二进制代码级的可移植性，在网络上实现了跨平台的特性。

Java 的解释器又称为“Java 虚拟机（JVM, Java Virtual Machine）”，是驻留于计算机内存的虚拟计算机或逻辑计算机，实际上是一段负责解释执行 Java 字节码的程序。JVM 能够从字节码流中读取指令并解释指令的含义，每条指令都含有一个特殊的操作码，JVM 能够识别并执行它。从这个意义上说，Java 可以被称为是一种“解释型”的高级语言。高级语言的解释器（或称解释程序）对程序边解释边执行，执行效率较低。因此，运行 Java 程序比可直接在操作系统下运行的 C 或 C++等“编译型”语言程序速度慢得多，这是 Java 语言的一个不足。

若 Java 解释器是一个独立的应用程序，并可以在操作系统下直接启动，那么它解释执行的程序被称为“Java Application（Java 应用程序）”；若 Java 解释器包含在一个 WWW 的客户端浏览器内部，使得这个浏览器能够解释字节码程序，则这种浏览器能够自动执行的 Java 程

序被称为“Java Applet（Java 小程序）”。这两种程序从程序结构到运行机理都不相同，Application 多在本地或服务器上运行，而 Applet 则只能通过浏览器从服务器上下载后再运行。

无论是 Java Application 还是 Java Applet，其程序源代码文件都以.java 为文件扩展名，而.class 则是编译后二进制字节码文件的文件扩展名。

1.2 面向对象程序设计

Java 语言是一种面向对象（OO, Object Oriented）的程序设计语言。无论是 Java 应用程序，还是 Java 小程序，它们都是以类为基础构建的。在认识 Java 程序前，这里先介绍一些关于面向对象程序设计的概念。

1.2.1 传统与面向对象程序设计语言

传统的面向过程的程序设计方法从解决问题的每一个步骤入手，较适合于解决比较小的简单问题。如广为流传的传统程序设计语言 BASIC、C 等采用面向过程的程序设计模型，但是由于这类语言本身几乎没有支持代码重用的语言结构，并且缺乏统一的接口，使得当程序的规模达到一定程度时，程序员很难控制其复杂性。面向对象的程序设计方法则按照现实世界的特点来管理复杂的事物，把它们抽象为对象，具有自己的状态和行为，通过对消息的反应来完成一定的任务。

面向对象程序设计（OOP）是当今计算机领域最流行的程序设计方法。这里的“对象”是什么含义呢？借用一个现实世界中“车”的例子来说明对象的含义。

在现实生活中，人们理解的“车”有各种各样的种类，如手推车、自行车、摩托车、汽车等。从各种车可归纳出它们的共性，如车有车轮、重量、颜色等，是汽车还有车速、耗油量等。这些是从“车”这类事物抽象出来的共性，也即所谓的数据、数据成员或属性。车不仅有这些静态的数据，还有很多与这些数据有关的动作和行为，如车的启动、加速、刹车和修理等，这就是所谓的代码、成员函数或方法。将上述车的数据和动作代码组合起来，就得到一个车类（class）。

在 Java 语言中定义一个关于车的类，一般的形式为：

```
class 车{ // 定义一个车类
  // 车的数据成员定义
  车轮数;
  车的颜色;
  车的重量;
  车速;
  ......
  // 车的成员方法定义
  启动();
  加速();
  刹车();
  修理();
  ......
```

}

有了抽象的车类后，一辆实际的车，如一辆自行车、一辆汽车等，就是车类的一个一个对象或实例。对象是一个实体，而不像车是一个抽象概念。现实世界中，人们使用的一辆自行车、一辆汽车是车类的一个对象。类是一类事物共性的反映，而对象是一类事物中的一个，是个性的反映。每个对象都有与其他对象不完全一样的特性，如你和我的自行车虽然都是自行车，但二者的颜色、重量等就不可能完全一致。

通过抽象来处理复杂事物的方法可从现实世界对应到计算机程序设计。传统的算法程序可以抽象成各种要处理的对象，一个对象就是数据和相关的方法的集合，其中，数据表明对象的状态，方法表明对象所具有的行为。一个对象的数据构成这个对象的核心，包在它外面的方法使这个对象和其他对象分离开来。每个对象都封装了自己的行为，将这些对象甚至程序对象当做现实世界具体的实例，它们能响应外界的刺激并进行相应的动作，这就是面向对象编程的基础。与人类理解复杂事物的方式一样，面向对象的概念构成了 Java 的核心。面向对象具有封装、继承、多态三个主要特性。

1.2.2 对象的性质

1，封装性

操纵汽车时，不用去考虑汽车内部各个零件如何运作的细节，而只需根据汽车可能的行为使用相应的方法即可。实际上，面向对象的程序设计实现了对象的封装，使用户不必关心对象的行为是如何实现这样一些细节的。从最基本的角度看，任何程序都包含两个部分：数据和代码。在传统的代码模型中，数据在内存中进行分配并由子程序或函数代码来处理；而面向对象设计的核心一环是将处理数据的代码、数据的声明和存储封装在一起。

可以把封装想像为一个将代码和数据包起来的保护膜。这个保护膜定义了对象的行为，并且保护代码和数据不被任何其他代码任意访问。即一个对象中的数据和代码相对于程序的其他部分是不可见的，它能防止那些不希望的交互和非法的访问。

Java 封装的基本单元是类，即 Java 程序的基本元素是类。用户可以创建自己的类，它是一组具有行为和结构的对象的一种抽象。对象是相关类的具体实例，是将类作为模子造出的一个翻版，因此，有时也称对象为类的实例。

封装的目的是为了减少复杂性，因此，类具有一套隐藏复杂性的机制。类中的每个数据和方法可以被定义为公共或私有。类的公共部分可以让外界的用户知道或必须知道，公共的数据和方法是类与外部的接口，程序的其他部分通过这个接口使用类的功能。而定义成私有的数据和方法则不能被类以外的其他代码访问。

借助于类的封装性便可以将数据和方法像部件一样用于各种程序，而不必了解记忆其内部细节。改变类的其他部分不会对整个程序产生预料之外的影响，只要保持类接口不变，它的内部工作方式可以随意改动。

需要改变对象的数据或状态，或需要进行对象之间的交互时，面向对象的程序设计方法提供消息机制。例如，要使汽车加速，必须发给它一个消息，告诉它进行何种动作（这里是加速）以及实现这种动作所要的参数（这里是需要达到的速度等）。这里要指明消息的接收者，接收对象应采用的方法，方法所需要的参数。同时，接收消息的对象在执行相应的方法后，可能会给发送消息的对象返回一些信息（如加速后，汽车的车速表上会出现已经达到的速度

等)。

由于任何一个对象的所有行为都可以用方法来描述，通过消息机制就可以完全实现对象之间的交互，同时，处于不同处理过程甚至不同主机的对象间都可以通过消息实现交互。

2. 继承性

人们通常都会将世界看成相互关联的可划分层次的各种对象，如车、汽车和轿车。这里的汽车是车的继承，而轿车又是汽车的继承。下一层次继承了上一层次的所有特性。一个多层次的继承关系构成了一个类树结构。

在已经定义了车类后，再定义一个汽车类，若再一次声明车轮数、颜色、重量等车类已定义的数据和方法，那么效率就太低了，其实只需定义汽车与一般意义下的车的不同即可。例如，增加定义一下汽车的车门数、座位数等，而继承车类中已定义过的数据和方法，这样可使得程序代码得到充分复用，结构更加清晰易懂，程序的可维护性、逻辑性就更强。

在面向对象的程序设计中，继承是指在已有类的基础上建立一个新类。新类自动拥有父类的所有元素：数据成员和成员方法，然后再根据需要添加新任务所需的数据成员和成员方法。合理使用继承可以减少很多的重复劳动。若类实现了一个特别的功能，那么由该类继承的新类（派生类）就可以重复使用这些功能，而不再需要重新编程。对 Java 的内置类可以创建派生类，也可以对自己创建的类建立派生类。

一个不由任何类派生来的类称为基类；一个派生类的最近的上层类叫做该类的父类；从某一类派生出来的类叫做该类的子类。通过父类和子类，实现了类的的层次，可以从最一般的类开始，逐步特殊化，定义一系列的子类。子类的层次并不是越多越好，因为，若层次太多，还不如重新创建一个新类。

一个类从派生它的基类到它自身可能要经过好几个层次。类不仅能继承其父类的所有方法和实例变量，而且还能继承从它的基类开始到它自身之间经过的所有层次上的类的方法和实例变量。通过继承也实现了代码的复用，使程序的复杂性线性地增长，而不是呈几何级数增长。

在 Java 中，继承车类派生汽车类的一般形式为：

```
class 汽车 extends 车 { ... ...}
```

继承和封装具有很好的合作性。若一个给定类封装了某些属性，那么它的任何子类将继承这些属性并可增加它们特有的属性。

3. 多态性

不同的对象对于相同的方法表现其不同的理解和响应。可以想像的到，对于自行车和汽车，它们都定义了刹车的方法，但它们的刹车方法却是完全不同的。

对象中的方法是通过参数来传送信息的。这些参数作为方法（函数）的输入，需要利用方法加以执行。

为了在大部分函数型编程语言中完成两个不同的任务，需要给两个函数定义不同的名字。而在面向对象的程序设计语言中，多态性意味者一个对象具有多个面孔。Java 通过方法重写和方法重载来实现多态。

通过方法重写，一个类中可以有多个具有相同名字的方法，通过传送给它们不同个数和

不同类型的参数来决定使用哪种方法。

通过方法重载，子类可以重新实现父类的某些方法，使其具有自己的特征。例如，对于汽车类的加速方法，其子类（如赛车）中可能增加了一些新的部件来改善提高加速性能，这时可以在赛车类中重载父类的加速方法。重载隐藏了父类的方法，使子类拥有自己具体实现，更进一步表明了与父类相比子类所具有的特殊性。

Java 不仅允许程序设计者自己创建类，还针对各种应用提供了大量的预定义类库，如屏幕显示、文件访问、数学计算等方面的类库。大量的预定义类的学习是 Java 语言学习的一个重点，也是一个难点。要掌握基本概念，通过对一些典型系统类的学习，学会举一反三，触类旁通。

有了有关类、对象的概念后，下面来看一看简单的 Java 程序。

1.3 Java 程序举例

前面已经介绍过，Java 程序分为应用程序（Application）和小程序（Applet），下面通过这两种程序的简单例子，对它们进行介绍。

1.3.1 Java 应用程序举例

【例 1.1】 输出信息为“Hello World!”的 Java 应用程序。

```
/* Hello.java */
public class Hello{        // 一个 Application
  public static void main (String args[]){
    System.out.println(" Hello World!" );
  }
}
```

程序中第 1 行用“/*”和“*/”括起来和第 2 行后面以双斜线“//”引导的内容是 Java 语言的注释信息。在程序中使用注释，可增加程序的可读性。

第 2 行开始是类的定义，保留字 class 用来定义一个新的类，其类名为 Hello，它是一个公共类（public）。Java 程序中可以定义多个类，但是最多只有一个公共类，程序文件名要求与这个公共类的类名相同。整个类定义由大括号{}括起来，其内部是类体，类体中定义类的数据成员和成员方法。在本例中定义了一个 main()方法，其中 public 表示访问权限，指明所有的类都可以使用这一方法；static 指明该方法是一个类方法，它可以通过类名直接调用；void 则指明 main()方法不返回任何值。

对于一个 Java 应用程序来说，main()方法是必须的，而且必须按照如上的格式来定义。Java 解释器在没有生成任何对象的情况下，以 main()作为入口来执行程序。每个类中可以定义多个方法，但 main()方法只能有一个，作为程序的入口。main()方法定义圆括号()中的 String args[]是传送给 main()方法的参数，参数名为 args，它是类 String 的一个对象。方法的参数用“类名 参数名”来指定，多个参数间用逗号分隔。在 main()方法的实现（方法体——大括号括起来的部分）中只有一条语句：

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

它的功能是在标准输出设备（显示器）上输出一行字符：

```
Hello World!
```

这里调用 java.lang 包中 System 类的功能，而 System.out 又是 java.io 包中 OutputStream 类的对象，方法 println()的作用是将圆括号内的字符串在屏幕输出并换行。

1.3.2 Java 小程序举例

【例 1.2】 显示信息为“Hello World!”的 Java 小程序。

```
import java.awt.Graphics;
import java.applet.Applet;
public class HelloApplet extends Applet{ // 一个 Applet
  public void paint(Graphics g){
    g.drawString("Hello World!",20,20);
  }
}
```

在这个小程序中，首先用 import 语句引入 java.awt.Graphics 类和 java.applet.Applet 类，这是本小程序需要的两个类：小程序需要继承 Applet 类来构造；图形界面的小程序输出常用具有绘图功能的 paint()方法，该方法需要 Graphics 类的参数。然后定义一个公共类 HelloApplet，用 extends 指明它是 Applet 的子类。Java 小程序都是 Applet 类或 JApplet 类的子类。在类体中，这里重写父类 Applet 的 paint()方法，其中参数 g 为 Graphics 类的对象，可认为是用于绘图的画板。在 paint()方法中，调用对象 g 的方法 drawString()，在坐标(20,20)处输出字符串“Hello World!”，其中坐标是用像素点来表示的。

这个程序中没有 main()方法，这是 Applet 与 Application（如例 1.1）的主要区别之一。

1.3.3 Java 程序结构

从上述例子中可以看出，Java 程序是由类构成的，对于一个应用程序来说，必须在一个类中定义有 main()方法，包含 main()方法的类是该应用程序的主类。而对小程序来说，它必须作为 Applet 类或 JApplet 类的一个子类，继承 Applet 或 JApplet 类的类是小程序的主类。下面说明典型的 Java 程序的书写规范。

1. 命名

若 Java 文件包含一个 public 类，它必须按该类的名称来命名。例如，在例 1.1 中，包含一个 public 类型的类 Hello，所以程序文件名必须命名为 Hello.java。在例 1.2 中，包含一个 public 类型的类 HelloApplet，所以程序文件名必须命名为 HelloApplet.java。类名与程序文件名的大小写也要一致，否则编译器会指出错误。

2，类个数

一个源文件中最多只能有一个 public 类，其他类的个数不限。

3．结构

package 语句；// 包语句，0 或 1 个，必须放在文件开始位置
import 语句；// 引入语句，0 或多个，必须放在所有类定义之前
interfaceDefinition；// 接口定义，0 或多个
public classDefinition；// public 类定义，0 或 1 个，必须与文件名同名
classDefinition；// 类定义，0 或多个

在书写源程序时，最好采用分层次的缩进方式书写，这有助于阅读理解程序，也为调试程序提供便利。书中例题的书写格式供参考。

1.4 Java 程序开发工具与开发方法

若要编写 Java 程序，就需要开发工具。现在可用于开发 Java 程序的工具很多，常用的有美国 Sun 公司的 Java SDK，Borland 公司的 Jbuilder，Microsoft 公司的 Visual J++等。第一个工具是免费的，可以到相应公司的网站或其他提供软件下载的网站去下载。本书使用的开发工具采用 Java SDK，这是因为 Java SDK 是 Sun 公司在其网站上提供不断更新的免费下载版本，比较容易得到。Java SDK 是一个命令行版本，设计图形界面程序时方便程度稍差一些。

1.4.1 Java SDK

Java SDK 的意思是 Software Development Kit，即 Java 软件开发工具包（以前的版本也称为 JDK）。截止到 2002 年 2 月，提供下载的 SDK 标准版软件最新正式版本为 1.4.0，有不同操作系统的不同版本。下面的介绍采用 Windows 95/98 系统版本。安装、运行 Java SDK 一般需要 Pentium 以上的 CPU、32 MB 以上内存、硬盘剩余空间 70 MB 以上的机器。

1，下载并安装 Java SDK 开发工具

可以从网址 http://java.sun.com 下载最新的 SDK 开发工具 j2sdk-1_4_0-win.exe（37 067 134 字节）和 API 说明文档 j2sdk-1_4_0-doc.zip（31 262 514 字节，若需要）。下载完成后运行 j2sdk-1_4_0-win.exe（自解压文件）即进行开发工具的安装，安装时可指定安装盘和目录，也可安装到默认的盘和目录。若用默认值，安装后将在 C 盘根目录下创建一个名为 jdk1.4 的目录，可运行的程序工具安装在 jdk1.4\bin 目录中，并可以在任何目录中运行，前提是在设置运行程序的系统中设置了路径。对帮助文档 j2sdk-1_4_0-doc.zip，可用 Winzip.exe 软件进行解压操作，解压到 jdk1.4 文件夹中去。

Java 命令行方式的编程环境由一系列目录文件、类库字节码文件、动态连接库 DLL 文件组成，具体的命令文件包括 javac.exe、java.exe、javah.exe、javap.exe、jdb.exe、javadoc.exe、appletviewer.exe 等。

下面对几个主要的开发工具进行简单介绍。

（1）appletviewer 小程序浏览器

appletviewer.exe 提供了一个 Java Applet 运行环境，在其中可测试 Java Applet。appletviewer 读取包含 Java 小程序的 HTML 文件并在一个窗口中运行它们。命令行格式如下：

```
appletviewer options URL
```

URL 表示由 URL 描述的 HTML 文档，要指出文件的扩展名 html 等。

（2）java 解释器

java.exe 文件是 Java 语言的解释器，用来解释执行 Java 字节码（.class）文件。命令行格式如下：

```
java [options] className <args>
```

类的字节码在称为 className.class 的文件中，这个文件是由 javac.exe 编译类文件源代码产生的，所有 Java 字节码文件都有.class 扩展名，扩展名是在编译时自动加上的。className 中必须包含一个 main()方法。一般情况下，命令行中指明字节码文件名即可，文件扩展名.class 在命令中不需写出。args 是 className 类的参数。

（3）javac 编译器

javac.exe 文件是 Java 语言的语言编译器。该编译器读取 Java 程序源代码文件，并将其编译成类文件（一组*.class 文件），类文件中包含有 Java 字节码。调用 javac.exe 的命令行中指定程序源文件时必须有文件扩展名.java。命令行格式如下：

```
javac [options] fileName.java...
```

（4）javah 头文件生成器

javah.exe 文件创建 C 头文件和存根文件，这些是把本地 C 成员函数包入 Java 所需要的。被创建的头文件给出了有关 Java 类的信息，这些信息是 C 成员方法与 Java 类交换数据所必须的。存根文件将用来创建将定义 Java 对象的结构与 Java 对象本身数据相联系的 C 文件。命令行格式如下：

```
javah [options] className...
```

javah.exe 程序有些像 Java 解释器，它只需要类名而不需要写.class 扩展名。javah 程序可接受多个类名以产生文件头和存根文件。

（5）javap 反汇编器

javap.exe 文件用于反汇编 Java 字节码文件，其输出结果由用户使用的控制符决定，若不选任何控制符，将在屏幕上显示类的公共方法和类数据。调用 javap.exe 的命令行格式如下：

```
javap [options] className...
```

（6）jdb 调试器

jdb.exe 文件用来调试 Java 语言编写的程序。有两种执行 jdb.exe 的方法：一种是用 jdb 直接解释执行要调试的类，这与 Java 解释器的执行类似；另一种方法是将 jdb 附加到一个已运行的解释器上，该解释器必须带 - debug 控制参数。调用 jdb 的命令行格式如下：

```
jdb [–help]
```

进入调试环境后，可用？或 help 命令获得调试命令的帮助信息。

（7）javadoc API 文件产生器

javadoc.exe 文件用于从 Java 的源文件生成 HTML 格式的文件。javadoc 扫描 Java 源文件中的注释及类声明，生成 HTML 格式的 API 文档供用户使用。这些 HTML 文件描述了 Java 类文件的类、变量、方法成员。所有 Java 类库 API 的 HTML 文件都可以由此程序创建。用户可以在 Java 源程序的注释中插入 HTML 标记。

javadoc 所需的 Java 文件注释为：

```
/**  注释  */
```

调用 javadoc 的命令行格式如下：

```
javadoc [options] package...
```

2，Java SDK 开发工具基本使用方法

一般情况下，自己开发的程序应创建在一个特定的目录（文件夹）中为好。可根据自己的方便和需要，在一个有空余空间的硬盘上创建一个属于自己 Java 程序的目录，下面为方便说明，设为 D 盘根目录下的 USEJAVA 目录。方法是：先进入 DOS 提示符状态，然后选择 D 盘，在根目录下创建目录，命令是：

```
D:<Enter>
MD \USEJAVA <Enter>
```

<Enter>表示按回车键。以后在开发 Java 程序的过程中，该目录作为当前目录使用。可用如下的命令选择\USEJAVA 为当前目录：

```
CD \USEJAVA <Enter>
```

为使用 Java 开发工具，可将 Java 命令程序所在目录设置到搜索路径中。命令是：

```
PATH C:\JDK1.4\BIN;%PATH%
```

这里，假设 Java SDK 安装到了 C 盘的 JDK1.4 目录中。上述操作就为开发 Java 程序设置好了环境。需要时，还要设置 classpath 类搜索路径（一般情况下不需要）。下面针对两种不同的 Java 程序，介绍开发 Java 程序的基本步骤。

（1）开发 Java 应用程序

以例 1.1 为例，首先用任意文本编辑程序（如 Notepad.exe 等）输入程序文本，并把它存入到一个名为 Hello.java 的文件中。这里，文件名应和公共类名相同（字母的大小写也要一致），因为 Java 解释器要求公共类必须放在与其同名的文件中。文件存储到自己创建的目录中，然后对它进行编译：

```
D:\USEJAVA>javac Hello.java
```

编译的结果是生成字节码（bytecode）文件 Hello.class。最后用 Java 解释器来运行该字节码文件：

```
D:\USEJAVA>java Hello
```

结果在屏幕上显示：

```
Hello World!
```

（2）开发 Java 小程序

以例 1.2 为例，首先也用任意文本编辑软件输入程序文本，要把它保存到文件 HelloApplet.java 中，然后对它进行编译：

```
D:\USEJAVA>javac HelloApplet.java
```

编译通过后得到字节码文件 HelloApplet.class。由于 Applet 中没有 main()方法作为 Java 解释器的入口，所以必须编写 HTML（Hyper Text Markup Language 超文本标记语言）文件，把该 Applet 嵌入其中，然后用 appletviewer 来运行，或在支持 Java 的浏览器上运行。它的 HTML 文件如下：

```
<html>
<head></head><body>
<applet code=HelloApplet.class  width=200 height=40>
</applet></body>
</html>
```

其中用<applet>标记来启动 HelloApplet，code 指明字节码所在的文件，width 和 height 指明 Applet 显示区域的大小，把这个 HTML 文件存入 HelloApplet.html，然后运行：

```
D:\USEJAVA>appleviewer HelloApplet.html
```

这时屏幕上弹出一个可调整大小的窗口，其中显示“Hello World!”。见图 1.1。

图 1.1　例 1.2 运行结果

1.4.2　Visual J++ 6.0 简介

Visual J++ 6.0 是美国微软公司 Visual Studio 98 程序员开发工具集中的一员。Visual J++ 是开发 Java 程序的 Windows 集成开发环境，在这个集成环境中，所有程序开发、运行必须的东西都有。例如，提供的 HTML 编辑器可编辑 HTML 文件，快速预览 Java 小程序显示结果，动态调试等。因为采用标准的 Windows 界面，令用户更容易接受和理解。Visual J++ 6.0 对系统的要求不高，内存容量 48MB，多于 100 MB 的硬盘剩余空间即可，而且编译、运行速度也较快，所以选用 Visual J++ 6.0 的开发人员也较多。

下面简单介绍使用 Visual J++ 6.0 开发的方法和步骤。

1，开发 Java Application

因为篇幅的关系，下面仅给出使用 Visual J++ 6.0 开发 Java Application 的一种方法，即创建非图形界面的控制台应用程序（Console Application）的方法。可以使用在 WFC 或 Java API 中的非图形类来开发控制台应用程序。创建控制台应用程序的步骤如下：

（1）启动 Visual J++ 6.0。

（2）单击“File”菜单中的“New Project...”命令，打开“New Project”对话框。

（3）在“New”选项卡上展开“Visual J++ Projects”文件夹，并单击“Applications”文

件夹，然后选择“Console Application”图标。

（4）在“Name”框中输入项目的名称，如 Project1。

（5）在“Location”框中输入要保存该项目的路径，或单击“Browse”按钮来定位到某个文件夹。

（6）单击“Open”或“打开”按钮，一个项目的折叠视图出现在“Project Explorer”中。

（7）在“Project Explorer”中展开该项目的节点，带有默认名字 Class1.java 的文件已经添加到该项目中。

（8）双击“Project Explorer”中的 Class1.java，在“Text”编辑区出现如下的系统自动生成代码：

```
/**
 * This class can take a variable number of parameters on the command
 * line. Program execution begins with the main() method. The class
 * constructor is not invoked unless an object of type 'Class1'
 * created in the main() method.
 */
public class Class1
{
  /**
 * The main entry point for the application.
  *
 * @param args Array of parameters passed to the application
  * via the command line.
  */
 public static void main (String[] args)
  {
    // TODO: Add initialization code here
  }
}
```

（9）可修改或添加程序代码构建自己的程序。例如：

在“// TODO: Add initialization code here”处添加如下程序语句：

```
System.out.println("Hello world!");
System.out.println("按回车键继续...");
System.in.read();
```

在 main (String[] args) 后添加“throws IOException”。

在程序的开始部分添加语句：

```
import java.io.*;
```

（10）按<F5>键，编译并运行程序，系统弹出 MS-DOS 窗口，在其中显示运行结果。

2．开发 Java Applet

这里也只介绍一种开发 Java Applet 的方法，其他方法读者可参考其他资料。

（1）启动 Visual J++ 6.0。

（2）单击“File”菜单中的“New Project...”命令，打开“New Project”对话框。

（3）在“New”选项卡上展开“Visual J++ Projects”文件夹，并单击“Web Pages”文件夹，然后在右边窗口中选择“Applet on HTML”图标。

（4）在“Name”框中输入项目的名称，如 Project1。

（5）在“Location”框中输入要保存该项目的路径，或单击“Browse”按钮来定位到某个文件夹。

（6）单击“Open”或“打开”按钮，一个项目的折叠视图出现在“Project Explorer”中。

（7）在“Project Explorer”中展开该项目的节点，带有默认名字 Applet1.java 的文件和 Page1.html 已经添加到该项目中。

（8）双击“Project Explorer”中的 Applet1.java，在“Text”编辑区出现一段较长的系统自动生成 Java Applet 代码。

（9）可修改或添加程序代码构建自己的程序。例如，将程序代码修改为如例 1.2 中的 Applet 程序。

（10）按<F5>键，系统将编译 Applet1.java，并在浏览器中运行包含编译结果 Applet1.class 的 Page1.html 文件，在浏览器的一个窗口区域中显示程序的运行结果。

习 题 一

1.1 Java 语言有哪些特点？什么叫 Java 虚拟机？

1.2 Java 程序分哪两类？各有什么特点？

1.3 Java 应用程序和小程序的结构有什么区别？

1.4 用 Java SDK 开发工具如何开发 Java 应用程序和小程序？

1.5 以第 1 章的例 1.1 为例，编写一个分行显示自己的姓名、地址和电话的 Java 应用程序。

1.6 以第 1 章的例 1.2 为例，编写一个分行显示自己的姓名、地址和电话的 Java 小程序。

1.7 若安装了 Visual J++ 6.0 系统，可在该系统中应用例 1.1 和例 1.2。

第 2 章　Java 语言基础

本章介绍数据类型、运算符、表达式以及简单输入输出等 Java 语言的基本内容，这些是用 Java 进行程序设计的基础。

Java 使用了类似 C/C++语言的语法，但作为一种新的计算机语言，Java 在某些方面更偏重于面向对象的思想或具有自己的特色。

2.1　标识符、保留字和分隔符

2.1.1　标识符

程序中使用的各种数据对象，如符号常量、变量、方法、类等，都需要一定的名称，这种名称叫做标识符（identifier）。Java 的标识符由字母、数字、下划线（_）或美元符（$）组成，但必须以字母、下划线或美元符开始。因为 Java 语言使用 Unicode 字符集，因此，组成标识符的字母、数字都是广义的。例如，对字母，不仅限于是英文的，还可以是希腊的、日文的、朝鲜文的等，甚至可以是中文的。Java 标识符是大小写敏感的，没有字符数的限制。

下面是一些合法的标识符：try，group_7，opendoor，boolean_1，θ，求和。而 try#，7group，open-door，boolean（保留字）是一些非法的标识符。

习惯上，表示类、接口名的标识符用大写字母开头，表示变量、方法名的标识符用小写字母开头，表示常量名的标识符中全部使用大写的字母。

2.1.2　保留字

保留字（reserved word）又称为关键字，是 Java 语言本身使用的标识符，它有其特定的语法含义。所有的 Java 保留字将不能被用做标识符，如 for、while、boolean 等都是 Java 语言的保留字。在表 2.1 中列出了 Java 语言中的所有保留字。

表 2.1　Java 保留字表

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

注：加*号的保留字是 Java 目前未用的。

2.1.3 分隔符

分隔符用来分隔开 Java 程序中的基本语法元素，可分为注释、空白符和普通分隔符三种。

1. 注释

在程序中加适当的注释可提高程序的可读性。注释有如下三种形式：

（1）“// 注释内容”表示以“//”开始的该行后面部分的内容为注释，用于单行的注释，放在一行的开头或语句的后部。

（2）“/* 注释内容 */”形式的注释可用于一段（多行）注释。

（3）“/** 注释内容 */”形式的注释也可用于一段（多行）注释，但这种注释可以由 javadoc 程序处理。

2. 空白符

空白符包括空格符、回车符、换行符和制表符等。在使用中，多个空白符与一个空白符的作用相同。

3. 普通分隔符

普通分隔符具有确定的语法含义，要按照语法规定去使用。有如下四种分隔符：

（1）大括号（{}），用于定义复合语句和数组的初始化以及定义类体、方法体等。

（2）分号（;），用于结束语句。

（3）逗号（,），用于分隔变量说明的各个变量和方法的各个参数等。

（4）冒号（:），用于分隔标号和语句。

2.2 数据类型

2.2.1 数据类型概述

计算机程序处理的对象是各种数据，数据类型是指数据的内在表现形式。例如，用计算机处理职工的有关信息，职工的年龄和工资都可以进行加、减等算术运算，具有一般数值的特点，在 Java 语言中称为数值型，其中年龄是整数，所以称为整型，工资一般为实数，所以称为实型。但对职工的姓名这样的数据是不能进行任何算术运算的，这种数据具有文字的特征，由一系列字符和汉字组成，在 Java 语言中称为字符串。根据现实世界数据的不同形式，数据就划分为多种不同的类型。数据类型不同，能进行的运算不同，取值范围也不同，在计算机语言中还反映为数据的存储形式不同。

通常，整型和实型合称为数值型，数值型、字符型和布尔型合称为基本数据类型（以后简称为基本类型）或简单数据类型或原始数据类型（Primitive）。基本类型是不可再分割、可直接使用的类型。复合数据类型（以后简称为复合类型）或称引用数据类型（Reference）是指由若干个相关的基本类型的数据（在面向对象的程序设计语言中还允许包含程序代码）组合在一起形成的一种复杂的数据类型。Java 语言支持的基本类型和复合类型见表 2.2。

表 2.2　Java 语言的数据类型

基本类型	数值类型	整型（byte，short，int，long）
		实型（float，double）
	字符型（char）	
	布尔型（boolean）	
复合类型	数组	
	类（class）	
	接口（interface）	

2.2.2　常量与变量

程序中所处理的数据表现为两种形式：常量和变量。

1，常量

Java 中的常量值是用文字串表示的，反映了一类在程序执行中不能变化的量。常量区分为不同的类型，如整型常量 123，实型常量 4.56，字符常量'a'，布尔常量 true 和 false 以及字符串常量“Java programming”。

2，变量

为了在 Java 中存储一个数据，必须将它容纳在一个变量之中。在变量中可根据需要存入不同的数据，这就是“变量”的含义。变量具有名称、类型、值和作用域等特性，在使用一个变量前必须先定义。变量定义是用标识符为变量命名、确定其数据类型，还可以根据需要为它赋初值（变量初始化）。定义了变量即为变量指定了存储方式。若是基本类型的变量，因为它们的存储长度是固定的，如何分配存储单元就确定了。若是复合类型的变量，定义后还需要用 new 运算符为用户数据分配存储单元，复合类型变量中存储用户数据存储单元的引用（指针）。可以用如下语法定义变量：

```
type identifier[[ = element],identifier];
```

该语句告诉编译器用指定的类型 type 和以标识符 identifier 为名字建立一个变量，这里的分号将告诉编译器这是一个说明语句的结束；方格中的逗号和标识符表示可以把几个类型相同的变量放在同一语句进行说明，变量名中间用逗号分隔。

在创建了一个变量以后，就可以给它赋值，或者用运算符对它进行允许的运算。

2.3　基本类型

2.3.1　整型数据

整型数据是最普通的数据类型，可表示日常生活中的整数。

1，整型常量

Java 的整型常量有三种形式：十进制、十六进制和八进制。

十进制整数以 10 为基数，用 0～9 这 10 个数字和正、负号组成，如 123，-456，0 等。在 Java 中，十进制整数的第一位数字不能为 0。

十六进制整数以 16 为基数，用 0～9 的 10 个数字、字母 A～F（小写也可，代表 10～15 这些整数）和正、负号组成。十六进制整数必须以 0X 或 0x 作为开头。如 0x123，-0xabc 等。

八进制整数以 8 为基数，用 0～7 的 8 个数字和正、负号组成。八进制整数必须用 0 开始，如 0567，-0123 等。

每一个整型常量默认为 int 类型，占有 32 位（即 4 个字节）的存储空间。整型常量所表示的范围为 -2 147 483 648～2 147 483 647，若要使用更大的数，可用 64 位（8 字节）的长整型数（long 类型）。若希望把一个整数强制存为一个长整型数，可以在数字后面加字母 l 或 L。

2，整型变量

整型变量按所占内存大小的不同有 byte、short、int、long 四种。Java 的整数都是有符号数。表 2.3 列出了各整型数据所占内存的位数和表示范围。

表 2.3　整型数据类型

类　　型	所占字节数	值　范　围
byte	1	-128～127
short	2	-32768～32767
int	4	-2147483648～2147483647
long	8	-9223372036854775808～9223372036854775807

下面详细介绍这四种整型数据。

（1）byte 类型

字节型 byte 是一种 8 位整数类型，取值范围很小，为 -128～127。它适用于表示网络和文件的字节流，用于分析网络协议或文件格式等。字节变量用 byte 定义，例如：

```
byte b; // 定义字节变量 b
byte c = 0x55; // 定义字节变量 c，并赋给初值十六进制数 55
```

（2）short 类型

短整型 short 是一种 16 位整数类型，值范围为 -32 768～32 767。这可能是 Java 语言中最不常用的一种类型，因为它采用了一种高位在前的数据格式，在使用低位在前数据格式的 PC 机上容易出错。短整型变量用 short 定义，例如：

```
short s; // 定义短整型变量 s
short t = 03377; // 短整型变量 t，并赋给初值八进制数 3377
```

（3）int 类型

整型 int 是一种 32 位整数类型，值范围为 -2 147 483 648～2 147 483 647。由于数值范围大，且在任何时候，带有 byte、short 和整型常量的一个整型表达式在计算前都会提升为 int 类型，所以使用较为广泛。例如：

```
int i; // 定义 int 整型变量 i
int j = 0x3344aabb; //定义 int 整型变量 j，并赋给初值十六进制数 3344aabb
```

（4）long 类型

长整型 long 是一种 64 位整数类型，可以表示比 int 类型取值范围更大的数据，它足以表示宇宙中的所有原子。例如，当使用 ms 来表示一年时，int 数据已经溢出（超出了 int 数据的值范围），所以需要 long 类型。例如：

```
long l;  // 定义长整型变量 l
long n = 0x33445566aa7788bb; //定义长整型变量 n，并赋给初值
```

2.3.2 实型（浮点型）数据

在 Java 中，实型数据只有十进制形式，通常用于精确到小数的计算中。

1. 实型常量

Java 的实型常量有标准和科学计数法两种表现形式。

（1）标准形式，由数字和小数点组成，且必须有小数点，如 0.123、4.56、789.0 等。

（2）科学计数法形式，数字中带 e 或 E，如 123e 或 4.56E3，其中 e 或 E 前必须有数字，且 e 或 E 后面的数字（表示以 10 为底的乘幂部分）必须为整数。

实数后面可带后缀 F、f 或 D、d，分别表示单精度实数和双精度实数，如 1.2f、3.4d、0.56e-7f、89e10d 等。

实型常量的默认存储空间为 64 位（8 个字节），即 double 型。若带有后缀，则按后缀决定类型。单精度 float 实数的存储空间占 32 位。

实数在机器中的存储格式以 IEEE754 格式存在。

单精度实数有效位数为二进制 23 位、十进制 7 位精度，多余位四舍五入。双精度实数有效位数为二进制 52 位、十进制 15 位精度，多余位四舍五入。

2. 实型变量

实型变量有 float 和 double 两种，见表 2.4。

表 2.4 实型数据类型

类　型	所占字节数	值　范　围
float	4	绝对值约 1.4E-45~3.4E38
double	8	绝对值约 4.9E-324~1.8E308

（1）float 类型

float 型的实型变量用来表示一个 32 位的单精度实数，它具有较快的运算速度，占用存储空间也较少，但在数值很大或很小时精度会受到影响。

（2）double 类型

double 型的实型变量用来表示一个 64 位的双精度实数。所有的几何函数如 sin、cos 等和开方函数 sqrt 都返回 double 型。double 所表示的实数精度比 float 更高，值范围比 float 更大。

实型变量可用关键字 float 或 double 来说明：

```
float f;
float g = 3.45f;
double d;
double mypi = 3.1415926535897932384;
```

注意：实数的默认类型是 double 型。若需要 float 型的常量，必须显式指明，即在一个实型常量的后面加一个字母 f 或 F。由于 Java 加强了类型检查，所以初始化一个实数时必须分清它是 float 型还是 double 型。如语句：

```
float g = 3.45;
```

在编译时将产生一个错误，应该用上述例子中的正确形式。

2.3.3 字符型数据

字符型 char 数据是由一对单引号括起来的单个字符。Java 使用 Unicode 格式的 16 位字符集，而不仅仅为 ASCII 字符集，因此 char 类型的范围从 0 到 65 535。Unicode 能够容纳所有语言字符集，包括拉丁语、希腊语、阿拉伯语、希伯来语、汉语、日语、朝鲜语等各国语言，因此使用 16 位是非常必要的。

在 Java 语言中，以反斜杠（\）开头的多个字符表示一个转义字符，转义字符一般用于表示某些非图形（非可视）字符。表 2.5 中列出了 Java 中的转义字符。例如，'\u0061'表示 ISO 拉丁码的'a'。若写出一个不存在的转义字符，则会出错。

表 2.5 转义字符表

转 义 字 符	描　　述
\ddd	1 到 3 位 8 进制数据所表示的字符（ddd）
\uxxxx	1 到 4 位 16 进制数所表示的字符（xxxx）
\'	单引号字符（\u0027）
\"	双引号字符（\u0022）
\\	反斜杠字符（\u005C）
\r	回车（\u000D）
\n	换行（\u000A）
\f	走纸换页（\u000C）
\t	横向跳格（\u0009）
\b	退格（\u0008）

字符变量用 char 说明，用来存放单个字符，它不是完整的字符串。示例如下：

```
char c1 = 'c';
char c2 = '\u0020';
char ch1 = 88;      // 正确，字符'X'的代码
char ch2 = 'ab';    // 错误，只能存储 1 个字符
char ch3 = "a";     // 错误，不能用字符串初始化
```

2.3.4 布尔型数据

布尔 boolean 类型是最简单的一种数据类型，布尔数据只有两个值：true 和 false，且都是保留字，表示“真”和“假”这两种状态。关系运算和逻辑运算返回布尔类型的值。

布尔型变量用 boolean 定义，例如：

```
boolean mouseOn = true;
boolean done = false;
```

布尔型是一个独立的类型，它不像 C 语言中的布尔型代表 0 和 1 两个整数，由于这个原因，Java 中的布尔类型不能转换成数字。

2.3.5 字符串数据

1. 字符串常量

字符串常量是用双引号括起来的零个或多个字符（包括转义字符）。例如：

```
" "                          // 空串
"我们赢了！\n"               // 一个包含转义字符的字符串
```

一个长字符串可以用“+”号分成几个短字符串。例如：

```
"中国 北京 "+
    "2008 奥运！"          // 一个字符串
```

在 Java 语言中，每个字符串常量被作为字符串类 String 的对象来处理。

2，字符串变量

在 Java 语言中，字符串变量是对象，是复合类型。有两种字符串变量类型：String 类和 StringBuffer 类。String 类是常量类，初始化后不能改变；StringBuffer 类是字符串缓冲区，可以修改。例如：

```
String str1 = new String("This is a string.");
String str2 = "This is a string.";
StringBuffer str3 = new StringBuffer();
StringBuffer str4 = new StringBuffer("This is a string.");
```

关于字符串的详细介绍见第 6 章。

2.3.6 默认初始值

在 Java 程序中，每个变量都对应一个值，不存在没有值的变量。若一个变量只声明而没有初始化，则使用它的默认初值。对于数值数据，默认数值为相应类型的零；字符的默认初值为'\u0000'；布尔初值为 false；复合类型初值为 null。

【例 2.1】 基本类型应用。

```
public class BasicType{
  public static void main(String args[]){
    byte b = 077;
```

```
        short s = 0x88;
        int i = 88888;
        long l = 8888888888881;
        char c = '8';
        float f = 0.88f;
        double d = 8.88e-88;
        boolean bool = false;
        String str = "我们赢了！";
        StringBuffer sb = new StringBuffer("中国 北京 2008 奥运！");
        System.out.println("b = "+b);
        System.out.println("s = "+s);
        System.out.println("i = "+i);
        System.out.println("l = "+l);
        System.out.println("c = "+c);
        System.out.println("f = "+f);
        System.out.println("d = "+d);
        System.out.println("boolean = "+bool);
        System.out.println("str = "+str);
        System.out.println("sb = "+sb);
    }
}
```

程序运行结果如下：

```
b = 63
s = 136
i = 88888
l = 888888888888
c = 8
f = 0.88
d = 8.88E-88
boolean = false
str = 我们赢了！
sb = 中国 北京 2008 奥运！
```

2.4 运算符

运算符的作用是与一定的运算数据组成表达式来完成相应的运算。对不同的数据类型，有着不同的运算符。对运算符，有运算对象（操作数）个数及类型、运算优先级、结合性等特性。

Java 语言对运算符的优先级、结合性和求值顺序有明确的规定，从根本上消除了运算符运算过程的二义性以及结果不统一的弊端。

运算符的优先级是指不同运算符在运算中执行的先后顺序。在 Java 语言中共有 17 种优先级，每个运算符分属确定的一个优先级别。Java 语言严格按照运算符的优先级由高到低地顺序执行各种运算。

运算符的结合性确定同级运算符的运算顺序。左结合性规定，运算数据先与左边的运算符结合，然后与右边的运算符结合。右结合性正好相反。

本节介绍一些基本的 Java 运算符，部分未介绍的运算符在以后的章节中陆续介绍。

2.4.1 算术运算符

算术运算符按操作数的多少可分为一元（或称单目）和二元（或称双目）两类，一元运算符一次对一个操作数进行操作，二元运算符一次对两个操作数进行操作。算术运算符的操作数类型是数值类型。

1，一元算术运算符

表 2.6 是一元运算符的列表。一元运算符的结合性是右结合性。

表 2.6　一元运算符列表

运　算　符	实 际 操 作	例　　子
+	正值	+ x
-	改变操作数符号	- x
++	加 1	x++, ++x
- -	减 1	x- -,- -x

+ 和 - 运算符使操作数取正、负值，用得不多，但它们有提升操作数类型的作用。如：

```
byte i=10,j；j = -i;
```

上述语句将产生编译错误，原因是 i 经过 - 运算后，已经提升为 int 类型，直接向字节变量 j 赋值是不允许的。

++ 和 - - 既可以是前置运算符也可以是后置运算符，这就是说，它们既可以放在操作数（必须是变量）前面（如++x），也可以放在后面（如 x++）。单独使用的时候，前置后置作用相同。但若在表达式中使用，前置后置的意义是不同的。前置时，变量的值先增 1 或减 1，然后用变量的新值参加表达式的计算；后置时，变量的值先参加表达式的计算，然后变量再增 1 或减 1。例如：

```
i = 5;
```

```
j = i ++;
```

执行后，j 为 5，i 为 6。

而

```
i = 5;
j = ++ i;
```

执行后，j 为 6，i 为 6。

【例 2.2】　说明++和--运算符的使用。

```
class IncrementDecrement {
  public static void main(String args[]) {
     int i,j,k;
     i = 1;
      k=(j=i++);
     System.out.println("i= "+i+" j="+j+" k= "+k);
      k=(i=2)*j--;
     System.out.println("i= "+i+" j="+j+" k= "+k);
     k=++i+ ++i+ ++i;
      System.out.println("i= "+i+" j="+j+" k="+k);
      k=i-- + i-- + i--;
     System.out.println("i= "+i+" j="+j+" k="+k);
   }
}
```

程序运行结果如下：

```
i= 2 j=1 k= 1
i= 2 j=0 k= 2
i= 5 j=0 k=12
i= 2 j=0 k=12
```

2，二元算术运算符

算术运算符的第二种类型是二元运算符，这种运算符并不改变操作数的值，而是返回一个必须赋给变量的值，表 2.7 列出了二元算术运算符。二元算术运算符具有左结合性。

表 2.7　二元算术运算符

运　算　符	实 际 操 作	例　　子
+	加运算	a + b
-	减运算	a - b
*	乘运算	a * b
/	除运算	a / b
%	取模运算	a % b

这些都是常用的运算，对二元算术运算符，有两点要注意：

（1）两个整数运算的结果是整数，5/2 结果是 2 而不是 2.5。

（2）取模运算是求两个数相除的余数，如 17%3 的结果是 2。可以对实数求余数。实数 a%b 的结果为 a-(int)(a/b)*b，余数的符号与被除数 a 相同。如 123.4%10 的结果为 3.4。

2.4.2 关系运算符

关系运算符用于确定一个数据与另一个数据之间的关系，即进行关系运算。所谓关系运算是比较运算，将两个值进行比较。关系运算的结果值为 true 或 false（布尔型）。表 2.8 列出了 Java 语言提供的 6 种关系运算符，它们都是双目运算符。

表 2.8 关系运算符

运 算 符	实 际 操 作	例 子
<	小于	a<b，4<5
>	大于	a>b，6>5
<=	小于或等于	a<=b，x<=0
>=	大于或等于	a>=b，b*b>=a*a*c
==	等于	a==b，y==x*x
!=	不等于	a!=b，x!=4

在这里要指出的是，等于运算符==不要与赋值运算符=混淆，以致于关系运算变成了赋值运算。

关系运算符常用于 if 语句、循环语句的条件中。运算符“==”和“!=”的运算优先级低于另外四个关系运算符，同一优先级中遵循自左至右的执行顺序。

2.4.3 布尔运算符

布尔运算符可以对布尔类型的数据（布尔常量、布尔变量、关系表达式和布尔表达式等）进行运算，结果也为布尔类型。表 2.9 列出了 Java 语言的布尔运算符，布尔运算规则见表 2.10。

表 2.9 布尔运算符

运 算 符	名 称	例 子
!	逻辑非	!a
&&	简洁与	a && b
||	简洁或	a || b
^	异或	a ^ b
&	非简洁与	a & b
|	非简洁或	a | b

表 2.10 布尔运算真值表

a	b	!a	a & b	a | b	a ^ b
false	false	true	false	false	false
false	true	true	false	true	true

续表

a	b	!a	a & b	a | b	a ^ b
true	false	false	false	true	true
true	true	false	true	true	false

简洁与、或和非简洁与、或的结果有时不同。非简洁与、非简洁或运算时，运算符两边的表达式都先要运算执行，然后两表达式的结果再进行与、或运算。简洁与、简洁或运算时，若只运算左边表达式即可确定与、或结果时，则右边的表达式将不会被计算。

例如:

```
int x = 2,y = 3,a = 4,b = 5;
boolean b = x ++ > y++ && a ++ > b ++;
```

则运算结果为:

```
b = false,x = 3,y = 4, a = 4, b = 5
```

而

```
boolean b = x ++ > y++ & a ++ > b ++;
```

运算结果为:

```
b = false,x = 3,y = 4, a = 5, b = 6
```

几个布尔运算符中，单目布尔运算符!的优先级最高，而&又高于|。运算符！又高于算术运算符和关系运算符，运算符&、|低于关系运算符。布尔运算符的执行顺序为自左至右。在一个布尔表达式中，使用的运算符种类可能较多，应注意运算符的运算优先级。

【例 2.3】 布尔运算符的使用。

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

程序运行结果为:

a<b = false

a/e = 8

f = 0

熟练地掌握关系运算符和布尔运算符，可以用逻辑表达式描述复杂的条件。

【例 2.4】　布尔表达式的使用。

（1）用 int 变量 a,b,c 表示三个线段的长，它们能构成一个三角形的条件是：任意两边之和大于第三边。该条件的 Java 布尔表达式为：

```
(a + b) > c && (b + c) > a && (a + c) > b
```

或

```
!((a + b) <= c || (b + c) <= a || (a + c) <= b)
```

（2）用 int 变量 y，表示年号 y 是闰年：

```
y % 400 == 0 | y % 4 == 0 & y % 100 != 0
```

（3）用 int 变量 age 存放年龄，boolean 变量 sex 存放性别（true 为男），表示 20 岁与 25 岁之间的女性：

```
20 <= age & age <= 25 & !sex
```

2.4.4　位运算符

位运算符用来对二进制位进行运算，运算操作数应是整数类型，结果也是整数类型。Java 中提供了如表 2.11 所示的位运算符。表中前四种称为位逻辑运算符，后三种称为算术移位运算符。

表 2.11　位运算符列表

运　算　符	实 际 操 作	例　　子
~	按位取反	~ a
&	与运算	a & b
|	或运算	a | b
^	异或运算	a ^ b
<<	左移	a << b
>>	算术右移	a >> b
>>>	逻辑右移	a >>> b

为了理解位运算符的功能，应掌握运算数据的二进制表示形式。Java 使用补码表示二进制数，在补码表示中，最高位为符号位。正数的符号位用 0 表示，其余各位代表数值本身。例如，+1 的 8 位补码为 00000001。负数的符号位用 1 表示，通常用将负数的绝对值的补码取反加 1 的方法来得到负数的补码。例如，-1 的 8 位补码为 11111111（-1 的绝对值的 8 位补码 00000001 按位取反加 1 为 11111110+1 = 11111111），-42 的补码为 11010110（-42 的绝对值的 8 位补码 00101010 按位取反加 1 为 11010101+1=11010110）。

若两个数据长度不同（如 short 和 int 型），对它们进行位运算时，则系统首先会将长度

短的数据的左侧用符号位填满（称为符号位扩展）。

1，按位取反运算符（~）

按位取反运算符“~”是一元运算符，对数据的各个二进制位取反，即将 0 变为 1，1 变为 0。例如：

```
int a= 0x45，b；
b= ~a；// b = 0xba
```

2．按位与运算符（&）

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

按位与可以用来把某些特定的位置 0（复位），其他位不变。这时只需将要置 0 的位同 0 与，而维持不变的位同 1 与。例如：

```
int a = 0x45，b = 0x31；
b = a & b；              // b = 1
```

3，按位或运算符（|）

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

按位或可以用来把某些特定的位置 1（置位），而不影响其他位。这时只需将要置 1 的位同 1 或，而维持不变的位同 0 或。例如：

```
int a = 0x45，b = 0x31；
b = a | b；              // b = 0x75
```

4．按位异或运算符（^）

参与运算的两个值，如果两个相应的位相同，则该位的结果为 0，否则为 1。即 0 ^ 0 = 0，0 ^ 1 = 1，1 ^ 0 = 1，1 ^ 1 = 0。

按位异或也称为按位加，可用于求反某些位。要求求反的位同 1 异或，维持不变的位同 0 异或。这种运算有如下特性：（x^y）^y = x。例如：

```
int a = 0x45，b = 0x31；
b = a ^ b；              // b = 0x74
```

5，左移运算符（<<)

用来将一个数据的所有二进制位全部左移若干位。

在不产生溢出的情况下，数据左移 1 位相当于乘以 2，而且用左移来实现乘法比乘法运算速度要快。例如：

```
int a = 7，b；
b = a << 1；        // b = 14
```

6. 算术右移运算符（>>）

用来将一个数据的所有二进制位全部右移若干位。移出的低位被舍弃，最高位用符号位补入。

右移 1 位相当于除以 2 取商，而且用右移来实现除法比除法运算速度要快。例如：

```
int a = -20，b；
b = a >> 1；        // b = -10
```

但要注意，当移位数据为负且最低位有 1 移出时，数据算术右移 1 位与数据除以 2 的结果不同。例如，-5(0xfb)右移 1 位等于-3，而(-5)/2 的结果是-2。

7. 逻辑右移运算符（>>>）

用来将一个数据的所有二进制位全部右移若干位。移出的低位被舍弃，最高位用 0 补入。例如：

```
int a = 0x88，b；
b = a >>> 2；       // b = 0x22(34)
```

2.4.5 赋值运算符

赋值运算符都是二元运算符，具有右结合性。

1. 简单赋值运算符（=）

赋值运算符“=”用来将一个数据赋给一个变量。在赋值运算符两侧的类型不一致的情况下，若左侧变量的数据类型的级别高，则右侧的数据被转换为与左侧相同的高级数据类型，然后赋给左侧变量。否则，需要使用强制类型转换运算符。

2. 复合赋值运算符

Java 语言允许使用复合赋值运算符，即在赋值符前加上其他运算符。复合赋值运算符是表达式的一种缩写。例如：a + = 5等价于 a = a + 5。复合赋值运算符有 11 种，如表 2.12 所示。

表 2.12 复合赋值运算符

运 算 符	用 法	等 价 于
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 – op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2
&=	op1 &= op2	op1 = op1 & op2

续表

运 算 符	用 法	等 价 于
|=	op1 |= op2	op1 = op1 | op2
^=	op1 ^= op2	op1 = op1 ^ op
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

复合赋值运算符使用比较简单，但要注意下述两点：

（1）复合赋值运算符的右边是一个整体，例如：

```
a *= b + c;
```

等价于

```
a = a * (b + c);
```

而不是

```
a = a * b + c;
```

（2）表 2.12 中的等价是有条件的，即 op1 仅计算 1 次。例如：

```
设有 int  a[]={1,2},b=2,i=0;          // a 是一数组
则           a[i++]  +=  b;              // 执行后 i=1
不等价于   a[i++]=a[i++]  +  b;      // 执行后 i=2
```

2.4.6 条件运算符

条件运算符是一种三元运算符，它的格式如下：

```
Operand ? Expression1:Expression2
```

在这个式子中，先计算 Operand 的真假，若为真，则计算并返回 Expression1，若为假，则计算并返回 Expression2。例如：

```
(a>b)?a:b;
```

这个表达式将返回 a 和 b 中较大的那个数值。

2.4.7 字符串运算符

运算符“+”可以实现两个或多个字符串的连接，也可实现字符串与其他类对象的连接，在连接时，其他类对象会被转换成字符串。另外，运算符“+=”把两个字符串连接的结果放进第一个字符串里。在前面的例子里，当想把几项输出内容输出在同一行里时使用的就是“+”运算符。

2.5 常用 Java 数值计算方法

在 Java 的类 Math 中包含了一些数值常量，如 E 和 PI，以及一些基本的数值计算方法，

如指数、对数、平方根和三角函数等，为用户程序进行数值计算带来了方便。

在表 2.13 中列出了一些常用的数值计算方法，未列出部分请参考 Java 的帮助文件。

表 2.13　类 Math 中的常用方法

方　法	功　能	参数类型	返回类型
abs(a)	求 a 的绝对值	(1)	(1)
sin(a)	求 a(弧度)的正弦值	double	double
cos(a)	求 a(弧度)的余弦值	double	double
tan(a)	求 a(弧度)的正切值	double	double
asin(a)	求 a 的反正弦值	double	double
acos(a)	求 a 的反余弦值	double	double
atan(a)	求 a 的反正切值	double	double
ceil(a)	求不小于 a 的最小整数	double	double
floor(a)	求不大于 a 的最大整数	double	double
pow(a,b)	求 a 的 b 次方	double	double
random()	产生 0.0≤1.0 的伪随机数	double	double
sqrt(a)	求 a 的平方根	double	double
log(a)	求 a 的自然对数	double	double
max(a,b)	求 a,b 中的大者	(1)	(1)
min(a,b)	求 a,b 中的小者	(1)	(1)

表 2.13 中参数类型和返回类型标(1)处的类型可以是 int、long、float 和 double，返回类型与参数类型两者类型相同。

【例 2.5】　类 Math 中常量和方法的使用。

```
class UseMath {
  public static void main(String args[]) {
    double a = 3.0, b = 4.0;
      double c = Math.sqrt(a * a + b * b);
      System.out.println("半径为 3.0 的圆面积是 ：" + Math.PI * a * a);
      System.out.println("直角三角形直角边为 3.0 和 4.0 时的斜边长为：" + c);
      System.out.println("-8 和-4 的较大者是：" + Math.max(-8, -4));
      System.out.println("-18 的绝对值是：" + Math.abs(-18));
      System.out.println("不小于 45.7 的最小整数是：" + Math.ceil(45.7));
      System.out.println("不大于 45.7 的最大整数是：" + Math.floor(45.7));
      System.out.println("sin25 度的值是：" + Math.sin(25.0/180*Math.PI));
    }
}
```

程序运行结果如下：

```
半径为 3.0 的圆面积是：28.274333882308138
直角三角形直角边为 3.0 和 4.0 时的斜边长为：5.0
```

-8 和 -4 的较大者是：-4

-18 的绝对值是：18

不小于 45.7 的最小整数是：46.0

不大于 45.7 的最大整数是：45.0

sin25 度的值是：0.42261826174069944

2.6 表达式

表达式是由操作数和运算符按一定的语法形式组成的符号序列。每个表达式运算后都会产生一个确定的值，称为表达式的值。表达式的值是有类型的，该类型称为表达式类型。表达式类型由运算符和参与运算的数据的类型决定。可以是简单类型，也可以是复合类型。

一个常量或一个变量是最简单的表达式，表达式的值即该常量或变量的值。用运算符连接几个表达式构成的式子仍是表达式。

可以按表达式值的类型将表达式分类。

2.6.1 类型转换

在一个表达式中可能有不同类型的数据进行混合运算，这是允许的，但在运算时，Java 将不同类型的数据转换成相同类型，再进行运算。

1，自动类型转换

整型、实型和字符型数据可以进行混合运算。在运算中，不同类型的数据先转换成相同类型，然后再进行运算。转换从低级到高级。可混合运算数据类型从低到高排列如下：

（低）→byte，short，char，int，long，float，double →(高)

不同类型数据之间的转换规则如表 2.14 所示。

表 2.14 不同类型数据之间的转换规则

类型 1	类型 2	转换后的类型
byte 或 short	int	int
byte 或 short 或 int	long	long
byte 或 short 或 int 或 long	float	float
byte 或 short 或 int 或 long 或 float	double	double
char	int	int

2，强制类型转换

高级数据要转换为低级数据时，需进行强制类型转换，Java 不像 C/C++那样允许自动类型转换。从一种类型转换到另一种类型可以使用下面的语句：

```
int a;
char b;
b=(char)a;
```

加括号的 char 告诉编译器把整型变成字符，并将它赋值给 b。

由于整型和字符型变量位长是不同的，整型是 32 位长，字符型是 16 位长，所以从整型转换到字符型可能会丢失信息。同样，当把 64 位的长整型数转换为整型时，由于长整型可能有比 32 位更多的信息，也很可能会丢失信息。即使两个量具有相同的位数，比如整型和单精度实型数据（都是 32 位），在转换小数时也会丢失信息，当进行类型转换时要注意使目标类型能够容纳原始类型的所有信息。不会丢失信息的类型转换见表 2.15。

表 2.15　不会丢失信息的类型转换

原 始 类 型	目 标 类 型
byte	short,char,int,long,float,double
short	int,long,float,double
char	int,long,float,double
int	long,float,double
long	float,double
float	double

需要说明的是，当执行一个这里并未列出的类型转换时可能并不总会丢失信息，但进行这样一个理论上并不安全的转换总是很危险的。

3. 表达式求值中的自动类型提升

在表达式的求值过程中，运算中间值的精度有时会超出操作数的取值范围。例如：

```
byte x = 30,y = 50,z = 100;
int a = x * y / z;
```

在运算 x*y 项时，结果 1500 已经超出了操作数 byte 类型的范围。为解决这类问题，Java 语言在对表达式求值时，自动提升 byte 或 short 类型的数据为 int 类型。Java 语言对表达式求值的自动类型提升规则为：

（1）所有 byte 和 short 类型提升为 int 类型。

（2）若一个操作数是 long 类型，则整个表达式提升为 long 类型。

（3）若一个操作数是 float 类型，则整个表达式提升为 float 类型。

（4）若有 double 类型，则表达式值为 double 类型。

自动类型提升对数据的运算带来了方便，但也容易引起编译错误。例如：

```
byte x = 30;
x = - x;          // 编译错误！不能向 byte 变量赋 int 值
```

【例 2.6】　不同类型数据的混合运算。

```
class Promote {
  public static void main(String args[]) {
    byte b = 42;
    char c = 'a';
    short s = 1024;
```

```
    int i = 50000;
    float f = 5.67f;
    double d = .1234;
     double result = (f * b) + (i / c) - (d * s);
    System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
     System.out.println("result = " + result);
   }
}
```

程序运行结果如下：

```
238.14 + 515 - 126.3616
result = 626.7784146484375
```

2.6.2 优先级

在一个表达式中可能有各种运算符，Java **语言规定了表达式中出现各种运算符的时候，哪种运算符先进行运算，哪种运算符后进行运算的运算符运算顺序，称为运算符的优先级。它指明了同一表达式中多个运算符被执行的次序，同一级里的操作符具有相同的优先级。在表达式中，优先级高的运算符先进行运算。例如对于表达式：**

```
a = b + c * d / (c ^ d)
```

Java **处理时将按照表** 2.16 **所列从最高优先级到最低优先级的次序进行。在上例中，因为括号优先级最高，所以先计算** c^d**，接着是** c*d**，然后除以** c^d**，最后，把上述结果与** b **的和存储到变量** a **中。**

不论任何时候，若一时无法确定某种计算的执行次序时，可以使用加括号的方法明确为编译器指定运算顺序，这也是提高程序可读性的一个重要方法。例如，对表达式：

```
a | 4 + c >> b & 7 || b > a % 3
```

运算次序的理解就不如下面的表达式清晰，因为在下面的表达式中用括号()**显式地表明了运算次序。**

```
(a | (((4 + c) >> b) & 7)) || (b > (a % 3))
```

表 2.16 **按从高到低的优先级列出了运算符，同一行中的运算符优先级相同。**

表 2.16　**运算符优先级**

优 先 次 序	**运　算　符**
1	.　[]　()
2	++　--　!　~　instanceof
3	new，(type)
4	*　/　%
5	+　-
6	<<　>>　>>>

续表

优先次序	运算符
7	< > <= >=
8	== !=
10	&
11	^
12	|
13	&&
14	||
15	?:
16	= += -= *= /= %= ^=
17	&= |= <<= >>= >>>=

2.6.3 结合性

在表达式中出现多个相同优先级的运算符时，就需要考虑结合性。结合性确定同级运算符的运算顺序。运算符有左结合性和右结合性两种。左结合性指的是从左向右使用运算符，例如，二元算术运算符具有左结合性，计算 a + b - c 时，操作数 b 的左、右运算符 +、- 是同级运算符，计算时，b 先与左边的 + 结合，计算 a + b，其和再与 c 相减；而右结合性是从右向左使用运算符，例如，赋值运算符具有右结合性，计算 a = b = c 时，操作数 b 的左、右运算符都是同级的赋值运算符，所以，先执行 b = c，再执行 a = b。

2.7 包装类

Java 不是纯面向对象的程序设计语言，这是因为它有 byte、short、int、long、float、double、char 和 boolean 这些基本类型。Java 保留它们是出于效率方面的原因，但有时候确实需要将这些基本类型作为类来处理。例如在调用方法时，若采用 int 数据作为参数，因为它是按值传送的，在被调方法中就不能改变调用方法中对应参数的值。但是，若将一个含有 int 数据的对象作为参数，那么，调用方法中对应参数中 int 值就可以被改变。

Java 语言中专门提供了所谓的包装类（wrapper class），这些类将以上基本类型包装成类。基本类型与它们对应的包装类见表 2.17。

表 2.17 基本类型与包装类

基本类型	包装类
byte	Byte
short	Short
int	Integer
long	Long
char	Character
float	Float
double	Double
boolean	Boolean
void	Void

使用包装类的方法与其他类一样，定义对象的引用，用 new 运算符创建对象，用方法来对对象进行操作。

例如:

```
Integer i = new Integer(10); // i 是 Integer 类的一个对象，值为 10
Integer j = new Integer(20); // j 是 Integer 类的一个对象，值为 20
```

【例 2.7】 输出 int 类型的最大值与最小值和 double 类型绝对值的最大值与最小值。

```
class Hello{
  public static void main(String args[]){
     System.out.println(Integer.MAX_VALUE); // int 类型的最大值
     System.out.println(Integer.MIN_VALUE); // int 类型的最小值
     System.out.println(Double.MAX_VALUE); // double 类型绝对值的最大值
     System.out.println(Double.MIN_VALUE); // double 类型绝对值的最小值
   }
}
```

程序运行结果如下:

```
2147483647
-2147483648
1.7976931348623157E308
4.9E-324
```

包装类中各类的方法虽然不完全相同，但有一些是类似的。下面的例子以 Integer 类为例，其他类请读者理解这些类中方法名称的构成规律，举一反三。

【例 2.8】 包装类 Integer 类常用方法的使用。

```
class UseWrapper {
  public static void main(String args[]) {
    int num = 2001;
    System.out.println(num + " 的二进制是: " +
        Integer.toBinaryString(num));  // 转换十进制数为二进制数
    System.out.println(num + " 的八进制是: " +
        Integer.toOctalString(num));   // 转换十进制数为八进制数
    System.out.println(num + " 的十六进制是: " +
        Integer.toHexString(num));     // 转换十进制数为十六进制数
    Integer iobj = Integer.valueOf("123");
    Integer iobj1 = new Integer(234);
    Integer iobj2 = new Integer("234");
    int i = iobj.intValue();
    System.out.println("iobj = " + iobj);
    System.out.println("i = " + i);
```

```
        System.out.println("iobj1==iobj2 ? " + iobj1.equals(iobj2));
    }
}
```

程序运行结果如下：

```
2001 的二进制是: 11111010001
2001 的八进制是: 3721
2001 的十六进制是: 7d1
iobj = 123
i = 123
iobj1==iobj2 ? true
```

2.8 输入输出初步

为了使读者尽早进行 Java 程序的上机实践，这里先介绍一些 Java 语言中有关输入和输出的基本内容。

2.8.1 输出

从前面的例子程序可以看到， System.out.println **在程序中可以将常量、变量或表达式的值输出到屏幕。** println **方法可有** 0 **个或** 1 **个参数。若参数是** 0 **个，则输出一个回车换行，光标移动到下一行行首；若有** 1 **个参数，该参数可以是** char、byte、int、boolean、float、double、String、char[]（**字符数组**）**或** Object（**对象**）**类型的，各种类型的数据转换成相应的字符串类型输出。输出给定所有内容后，输出一个回车换行。另外也常用** System.out.print **进行输出，** print **方法需要** 1 **个参数来输出，可用的参数类型与** println **相同，输出参数的值后不输出回车换行，因此，若输出内容本身不包含控制光标的内容或未满行，** System.out.print **输出后，光标将停留在输出内容后。**

在实际输出时，经常需要将提示信息与值一起输出，例如，将变量名提示与变量值一起输出，但 print **和** println **方法只支持一个输出项，这时，可参考前面各例中的方法，用“+”运算符连接多个输出项为一项进行输出。例如：**

```
int i = 10,j = 20;
System.out.println("i= " + i + " j= " + j);        // 输出为 i= 10 j= 20
System.out.println(i + j);                          // 输出为 30
System.out.println("sum= " + i + j);                // 输出为 sum= 1020
System.out.println("sum= " + (i + j));              // 输出为 sum= 30
```

2.8.2 输入

在 Java 中，未提供与 System.out.print()**对应的输入方法，仅有一个** System.in.read()**可用于从键盘输入整数值在** 0～255 **之间的** byte **类型数据。若程序运行时需要从键盘输入其他类型的数据，则需要使用** Java **提供的输入/输出功能。不论需从键盘输入何种类型的数据，**Java

从键盘接收数据都是以字符串的形式进行，再根据实际数据的需要进行类型转换。

在 Java 中实现输入要使用异常处理机制，这是 Java 特有的部分内容，如下列程序中的 throws IOException。关于异常处理的知识，可见第 3 章和以后各章的相应章节的介绍。

【例 2.9】 从键盘输入一个字符，并输出它在 Unicode 字符集中的前一字符和后一字符。

```
import java.io.*;  // 引入 java.io 包
class CharDemo {
  public static void main(String args[]) throws IOException{ // 抛弃异常
     char c,c1,c2;
     c=(char)System.in.read(); // 输入字符
      c1=(char)(c-1);
      c2=(char)(c+1);
     System.out.println(" : " + c1);
     System.out.println(" : " + c2);
   }
}
```

【例 2.10】 从键盘输入一个整数和实数，并输出它们的和。

```
import java.io.*;  // 引入 java.io 包
class InputDemo {
  public static void main(String args[])throws IOException {  // 抛弃异常
     // 用标准输入 System.in 创建一个 BufferedReader
     BufferedReader br =
        new BufferedReader(new InputStreamReader(System.in));
      System.out.print("请输入一个整数：");
     String str = br.readLine();          // 输入字符行存入字符串
     int i = Integer.parseInt(str);        // 转换字符串为整型数据
      System.out.print("请输入一个实数：");
     str = br.readLine();
     float f = Float.parseFloat(str);     // 转换字符串为实型数据
      System.out.print("它们的和是："+(i+f));
   }
}
```

程序中的语句“i = Integer.parseInt(str);”可用“i = Integer.valueOf(str).intValue();”代替，它们都可以将 str 字符串转换为 int 数据；同理，语句“f = Float.parseFloat(str);”可用“f = Float.valueOf(str).floatValue();”代替。

习 题 二

2.1 Java 定义了哪几种数据类型?

2.2 下列符号中不能作为 Java 标识符的是哪一个?

（1）条件 （2）xyz （3）45six （4）ω

2.3 下列哪些不是 Java 的关键字? 请选择出所有的答案。

（1）superclass （2）open （3）close （4）integer （5）import

2.4 在 Java 语言中，'\u0041'在内存中占用的字节数是多少?

（1）2 （2）4 （3）6 （4）8

2.5 在 Java 语言中，整型常数 123 占用的存储字节数是多少?

（1）1 （2）2 （3）4 （4）8

2.6 在 Java 语言中，实型常数 45.67 占用的存储字节数是多少?

（1）1 （2）2 （3）4 （4）8

2.7 下列数据哪些是常量? 这些常量各是什么类型的? 哪些是变量?

null false name Math.PI ff 0120 100L E 200d

2.8 下列哪些赋值是合法的? 请选择出所有的答案。

（1）long test = 012;

（2）float f = -412;

（3）int other = (int)true;

（4）double d = 0x12345678;

（5）short s = 10;

2.9 选出所有不合法的表达式。

（1）(true & true) （2）(4 & 5) （3）(int myInt = 0 > 3)

（4）float myFloat = 40.0; （5）boolean b = (boolean)99;

2.10 将下列代数式改写为等价的 Java 表达式。

（1） $\frac{1+xy}{1-xy}$ （2） $\sqrt{ab-c\frac{e}{d}}$

2.11 根据所给条件，列出逻辑表达式。

（1）满足下列条件之一为闰年：年号（year）能被 400 整除；年号能被 4 整除但不能被 100 整除。

（2）一元二次方程 $ax^2+bx+c=0$ 有实根的条件是：$a\neq 0$ 且 $b^2-4ac\geqslant 0$。

2.12 写出下列表达式的值。

（1）(3 + 4 * 5) / 2

（2）3 * 4 >= 8

（3）4 == 4 && 5 > 2 + 2

（4）!(2 * 3 != 10)

（5）8 != 6 || !(10 > 11 + 3)

2.13 代数式 sin45° 的 Java 表达式应该是什么?

2.14 设有代码:

```
Boolean b1 = new Boolean(true);
Boolean b2 = new Boolean(true);
```

下面哪些表达式是返回 true 的合法表达式？选择所有正确的答案。

（1）b1 == b2　　（2）b1.equals(b2)　　（3）b1 & b2

（4）b1 | b2　　（5）b1 && b2　　（6）b1 || b2

2.15　**分析下列程序，写出程序运行结果。**

```
class Ex2_15{
  public static void main(String args[]){
      boolean b;
      int i=0,j=0;
      b = i++>0 && ++j>0;
      System.out.println(b+" "+i+" "+j);
      i=j=0;
      b = i++>0 || ++j>0;
      System.out.println(b+" "+i+" "+j);
      i=j=1;
      b = i++>0 && ++j>0;
      System.out.println(b+" "+i+" "+j);
      i=j=1;
      b = i++>0 || ++j>0;
      System.out.println(b+" "+i+" "+j);
   }
}
```

第3章　流 程 控 制

一个计算机程序是由一系列的语句组成的，构成程序的所有语句对应着一个实际问题的一个计算机求解方法。在计算机程序设计技术中，称解决问题的详细步骤为算法。算法包括解决问题过程中需要执行的操作以及执行这些操作的顺序。一种计算机语言的程序即用该语言的语句来描述算法。

Java 语言虽然是一种面向对象的计算机语言，但在一个局部，如方法内、复合语句块内等，仍然需要面向过程的一些程序设计思想和方法。结构化程序设计方法是常用的一种面向过程程序设计的原则。采用结构程序设计的方法使得编程人员在进行程序设计时，为了提高程序质量而不能随心所欲地编程，要遵从一些原则。结构化程序设计的基本原则是：尽管现实世界的问题是复杂的、千变万化的，但对应的任意复杂的计算机算法流程只有三种基本结构，分别为顺序结构、选择结构和循环结构。每种结构是单入口、单出口的。Java 语言提供支持结构化程序设计的所有语句。

一般情况下，程序运行时按程序语句书写次序从前往后一条一条地执行语句，正如在前一章中见到的几个程序。流程控制语句可用来改变这个次序，从而控制程序的流程，实现一些复杂的算法。

学习编写计算机程序时应勤于思考，多动手实验。首先，必须透彻理解所要解决的问题，研究解决问题的方法和步骤。然后，按照结构化程序设计思想和确定的算法，自顶向下、逐步求精地进行程序设计。学习中要不断积累程序设计的经验和教训，融会贯通各种程序设计的技术和方法。

本章介绍 Java 的流程控制语句：分支语句、循环语句以及与流程有关的异常和方法使用等。

3.1　语句和块

3.1.1　语句

在 Java 语言中，有下面几类语句：空语句、标识语句、声明语句、表达式语句、分支语句、循环语句、跳转语句、同步语句、异常语句等。有两种基本的语句：表达式语句和声明语句。

1，表达式语句

表达式语句是由表达式加分号构成的语句。例如：

```
i++;
System.out.println("Hello! ");
```

在 Java 语言中语句用分号终止，但并不是所有的表达式都可以构成语句。例如，表达式

a <= b，加分号构成的语句无意义。下面几种类型的表达式可以通过添加分号构成表达式语句：

（1）赋值表达式，包含赋值运算符=或复合赋值运算符之一。

（2）++或- - 的前后缀形式。

（3）方法调用（无论它是否有返回值）。

（4）对象创建表达式，用 new 来创建一个对象的表达式。

2. 声明语句

声明语句声明一个变量，并可为其赋初值。声明语句可以出现在任意块内。定义在方法内或块内的局部变量在使用前必须赋初值，或在声明时进行初始化，或在声明后赋值。

3.1.2 块

用一对花括号“{}”将零个或多个语句括起来，就构成一个块（也称复合语句）。在块中允许包含另一个块（块嵌套）。在 Java 语言中，允许一个块出现在任何单一语句可以出现的地方。回顾前面的内容可以知道，类体和方法体都是块。

块体现了 Java 面向对象程序设计的封装概念，在一个块中声明的局部变量的作用域是该变量的声明开始点到最小的包含其声明的块结束。

【例 3.1】 块的使用。

```
class UseBlock{
  public static void main(String args[]){
    int a = 5;
    a += 8;
    {                        // 开始一个块
      int b = 6;             // 变量 b 只能在这个块中使用
      b = b + a;
      System.out.println(a + " " + b);
    }
  }
}
```

程序运行结果如下：

```
13 19
```

在这个程序中，main()方法体是一个块，变量 *a* 在第 3 行被声明，在整个程序中都有效；变量 *b* 在一个内嵌的块中声明，它只在声明它的块中有效。块嵌套时，外层声明的变量在内层仍然有效，但内层不能再声明与外层变量同名的变量。

3.2 分支语句

分支语句在程序中的作用是使程序更灵活，它允许程序根据不同的情况、不同的条件等

采取不同的动作，进行不同的操作，实现选择结构。在 Java 语言中使用的分支语句有 if - else 语句和 switch 语句。

3.2.1 if - else 语句

1. 用单个 if 语句实现单选、双选结构

用一个 if - else 语句，可实现根据一个关系或逻辑表达式的值是 true 还是 false 的两种情况下进行不同操作的程序结构，它的语法结构如下：

```
if(boolean-expression)
  statement1
[else
  statement2]
```

执行到本语句，首先要计算布尔表达式 boolean-expression，若值为真，则执行语句 statement1，否则（值为假时）执行语句 statement2（实现双选结构）。statement1 和 statement2 都可以是复合语句。

if - else 语句的 else statment2 部分可省略，省略时，若 boolean 表达式的值为假，则不执行任何语句（实现单选结构）。

【例 3.2】 用 Math 类的 random()方法产生一个字符，若该字符是一个大写英文字母，则输出“Yes!”，否则输出“No!”。

random()方法产生的随机数在 0.0 和 1.0 之间，乘以 128 后，其值在 0.0 和 128.0 之间，将它转换为 char 类型后，用 if 来判断是否在 A 和 Z 之间。程序如下：

```
class IsUpper{
  public static void main(String args[]){
     char ch;
     ch=(char)(Math.random()*128);
    if(ch >= 'A' && ch <= 'Z')
       System.out.println(ch + "是大写字母。");
     else
       System.out.println(ch + "不是大写字母。");
  }
}
```

程序的一次运行结果如下：

```
e 不是大写字母。
```

2. 用多个 if 语句实现多选结构

（1）if 语句并列

在多个 if 语句中可书写多个条件，若这些 if 并列，且这些条件包含了所有的情况，无一遗漏也不重复，就可以实现多选结构。

【例 3.3】　对任意三个存放在变量 *a*、*b*、*c* 中的 int 类型数据，按从小到大的顺序输出它们。

下面程序的设计思想是：将最小数据存入变量 *a* 中，这需要将 *a* 与 *b*、*c* 比较两次；然后，将 *b*、*c* 中的小者存入变量 *b* 中，这需要 *b* 与 *c* 比较 1 次，共需比较 3 次。用 3 个并列 if 实现。

```
class Sort3{
  public static void main(String args[]){
    int a = 10,b = 8,c = 12,t;
     if(a > b){t = a;a = b;b = t;}
    if(a > c){t = a;a = c;c = t;}
    if(b > c){t = b;b = c;c = t;}
    System.out.println(a + " " + b + " " + c);
    }
  }
```

程序运行结果如下：

```
8 10 12
```

在本例中，程序运行前，数据 *a*、*b*、*c* 中的值是无序的，程序运行后，*a*、*b*、*c* 已经按从小到大的顺序排序。

（2）if 语句嵌套

在 if - else 语句中，若 statement1 或 statement2 又是 if - else 语句时，就构成了 if 语句嵌套。采用 if 语句嵌套的程序设计方法也可实现多选操作。

【例 3.4】　检查用 random()方法产生的一个字符，判断是否为英文大写字母、小写字母、数字、空格或是其他符号，并输出相应信息。

程序要根据 5 种不同的字符情况输出不同的信息。这属于多分支的程序设计，采用 if 嵌套的方法。程序如下：

```
class Letter{
  public static void main(String args[]){
     char ch;
     ch=(char)(java.lang.Math.random()*128);
    if(ch < ' ')
        System.out.println("是不可显示字符!");
   else if(ch >= 'a' && ch <= 'z')
       System.out.println(ch + "  是小写字母!");
   else if(ch >= 'A' && ch <= 'Z')
       System.out.println(ch + "  是大写字母!");
   else if(ch >= '0' && ch <= '9')
       System.out.println(ch + "  是数字!");
     else
       System.out.println(ch + "  是其他符号!");
```

```
    }
}
```

需要注意的是，Java 编译器总是将 else 与其最近未配对的 if 配对。因为 if - else 语句允许 else 部分省略，会出现 if 的个数多于 else 个数的情况，此时要注意 if 和 else 的配对，否则可能发生逻辑错误。例如：

```
if(x > 5)
  if(y > 10)
     System.out.println("x 大于 5 并且 y 大于 10。");
 else
   System.out.println("x 不大于 5。");
```

程序员想实现 *x* 是否大于 5 输出不同字符串的本意，但 Java 编译器却理解为：当 *x* 大于 5 且 *y* 大于 10 时输出第一个字符串，当 *x* 大于 5 且 *y* 不大于 10 时输出第二个字符串。两个字符串均在 *x* 大于 5 时输出，当 *x* 不大于 5 时，什么也不输出。此时，出现逻辑错误。

为避免嵌套时 if 和 else 配对时逻辑出错，可用下面的方法之一来解决：

① 在用 if 嵌套方法进行多分支程序设计时，将 if 语句嵌套在 else 部分。例如，上述例子中的逻辑关系可改写为：

```
if(x <= 5)
   System.out.println("x 不大于 5。");
 else
  if(y > 10)
     System.out.println("x 大于 5 并且 y 大于 10。");
```

② 若需要在 if - else 语句的 statement1 部分嵌套 if 语句，则将内层的 if 语句用括号括起来，指明正确的 if 与 else 的配对组合。例如，上述例子中的逻辑关系可改写为：

```
if(x > 5){
  if(y > 10)
     System.out.println("x 大于 5 并且 y 大于 10。");
   }
 else
   System.out.println("x 不大于 5。");
```

3.2.2 switch 语句

switch 多分支语句结构实际上也是一种 if - else 结构，不过它使得在编码时很容易写出判断条件，特别是条件有很多选项而且比较简单的时候。switch 的语法结构如下：

```
 switch(expression){
   case  value1：statement1;break;
   case  value2：statement2;break;
    ...
```

```
    case valueN：statementN;break;
    default：defaultstatement;break;
  }
```

执行 switch **语句时，首先计算表达式** expression **的值，其类型是整型或字符型，并与各个** case **之后的常量值** value **类型相同。然后将该值同每种情况** case **列出的值** value **做恒等比较：若相等，则程序流程转入** value **后紧跟的语句** statement（**块**）；**若表达式的值与任何一个** case **后的值** value **都不相同，则执行** default **后的语句** statement（**块**）；**若没有** default **子句，则什么都不执行。**

使用 switch **语句时，需要注意的是：**

（1）**各个** case **后的常量值** value **应各不相同。**

（2）**通常在每一种** case **情况后都应使用** break **语句，否则，遇到第一个相等情况后，下面** break **前的所有语句都会被执行（包括** default **后面的语句，若有的话）。可以试着分别加上或去掉** break **语句来执行例** 3.4**。**

（3）**各个分支的** statement **可以是一条或多条语句，不必使用复合语句。**

（4）**不同** case **后的语句** statement **相同时，可以合并多个** case **子句。**

（5）switch **语句在用表达式的值比较每一个** case **后的值** value **时，是从前往后顺序进行的。若各个值** value **互不相同，则** case **子句的顺序可任意。**switch **语句最后执行** default **子句，通常** default **子句放在** switch **结构的最后。**

【例 3.5**】 查看五级分制的成绩对应的百分制成绩的范围。设百分制与五级分制成绩的对应关系为：**

A： 90 ～ 100
B： 80 ～ 89
C： 70 ～ 79
D： 60 ～ 69
E： 60 **以下**

```
class SwitchTest{
  public static void main(String args[])throws java.io.IOException{
     char a;
     System.out.println("请输入大写字母 A ～ E: ");
     a=(char)System.in.read();
     switch(a){
       case 'A':System.out.println("范围 90 ～ 100！");break;
       case 'B':System.out.println("范围 80 ～ 89！");break;
       case 'C':System.out.println("范围 70 ～ 79！");break;
       case 'D':System.out.println("范围 60 ～ 69！");break;
       case 'E':System.out.println("60 分以下！");break;
       default:System.out.println("重新输入！");
     }
  }
```

```
    }
```

在代码中加 break 语句后，应明确知道程序将会发生的变化，并要确认程序没有转移到不想执行的代码上。

【例 3.6】 输入一个小写字母，判断是元音还是辅音。

因为元音有 5 个，都需要执行相同的语句，为避免重复，只需写出最后一个值 value 后的 statement 和 break 语句即可。

```
class EmptyCase {
 public static void main(String args[])throws java.io.IOException{
    // 从键盘输入一个小写字母
    char ch = (char)System.in.read();
    // 确定是元音还是辅音
    switch(ch) {
       case 'a':
       case 'e':
       case 'i':
       case 'o':
       case 'u':System.out.println(ch + "是元音");break;
       default:System.out.println(ch + "是辅音");
    }
  }
}
```

3.3 循环语句

循环语句的作用是使某一段程序根据需要重复执行多次。循环语句由循环体和循环条件两部分构成，循环体是要重复执行的语句，循环条件决定循环的开始、重复执行以及结束循环。循环语句实现的循环（或称重复）结构是一种封闭结构，当循环条件被满足时，重复执行循环结构内的操作，当循环条件不被满足时，退出循环结构。

一个循环一般包括四个部分：

（1）初始化部分，用来设置循环的一些初始条件，如累加器清零等。

（2）循环体部分，重复执行的一段程序，可以是一条语句，也可以是一块语句。

（3）迭代部分，在当前循环结束，下一次循环开始前执行的语句。常用形式为一个计数器值在增减。

（4）终止部分，一般为布尔表达式，每一次循环都要对该表达式求值，以检查是否满足循环终止条件。

Java 语言提供三种形式的循环语句：while 循环语句、do - while 循环语句和 for 循环语句，下面分别予以介绍。

3.3.1 while 语句

while 语句的一般格式为：

```
while(boolean-expression)
  statement
```

Java 执行 while 循环语句时，先检查 boolean 表达式（循环条件）的值是否为 true，若为 true，则执行给定语句 statement（即循环体），然后再检查 boolean 表达式的值，反复执行上述操作，直到 boolean 表达式的值为 false，就退出循环结构。

while 语句的执行是：先判断条件，根据条件再决定是否继续执行循环体（简称先判断后执行）。每执行一次循环体后，循环条件均应发生相应的变化，使得执行若干次循环后，循环条件会从 true 变为 false，以便能够结束循环。若执行循环时，循环条件总是为 true，则不能终止循环，这种死循环在程序设计中是要注意避免的。

若首次执行 while 语句时，循环条件为 false，则循环体一次也未执行，即 while 语句循环体最少执行次数为 0 次。

【例 3.7】　计算 1 + 2 + 3 + … + 100。

一组有规律的数据的连续加或连续乘等计算一般都用循环程序来解决。

```
class Sum1To100{
 public static void main(String args[]){
    int i,sum = 0;
     i=1;                    // 设循环初值
    while(i <= 100){   // 设循环条件为 i<=100
      sum += i ++;     // 在循环体中执行 i++，当 i 的值到 101 时，循环条件即为 false
     }
     System.out.println("1 到 100 的和为："+sum);
  }
}
```

程序运行结果如下：

```
1 到 100 的和为：5050
```

3.3.2 do - while 语句

while 语句在执行循环体前先检查 boolean 表达式（循环条件），有些情况下，不管条件表达式的值是 true 还是 false，都希望把循环体至少执行一次，那么就应使用 do - while 循环。do - while 循环语句的一般格式为：

```
do
  statement
while(boolean-expression);
```

do 循环语句首先执行给定的语句 statement（循环体），然后再计算 boolean 表达式（判断

循环条件），若表达式值为 false，则结束循环，否则重复执行循环体。do 语句的循环体至少被执行一次，这是 do 循环与 while 循环最大的区别。一般称 while 循环为“当型” 循环（先判断后执行），do 循环为“直到型”循环（先执行后判断）。

【例 3.8】 将键盘输入的数累加并显示累加和，直到输入 0 为止。

```
import java.io.*;
class ParseDemo {
  public static void main(String args[])throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
      String str;
      int i,sum=0;
      System.out.println("请输入数据，输入 0 结束。");
      do {
        str = br.readLine();
        i = Integer.parseInt(str);
        sum += i;
         System.out.println("当前的和是：" + sum);
    } while(i != 0);
   }
}
```

3.3.3 for 语句

for 循环语句在几种循环语句中的格式与用法最灵活，它的一般格式为：

```
for([expression1];[expression2];[expression3])statement
```

其中，表达式 expression1 指出 for 循环的循环初值；表达式 expression2 是一个关系或逻辑表达式，值为 false 时循环结束；表达式 expression3 指出每次循环时所进行的计算和更新。3 个表达式在使用中可根据需要，部分或全部不写均可。

执行 for 语句时，先计算表达式 expression1（只计算一次，默认时表示无初始内容），接着检查表达式 expression2 的值是 true 还是 false，若为 false，则不执行语句 statement（循环体），退出循环；若为 true，就执行给定的语句 statement，再计算表达式 expression3，然后，又检查 expression2 表达式的值，再根据值为 true 或为 false 决定是否执行循环体。

可以在 for 循环的表达式 expression1 中说明仅在循环中使用的变量，例如：

```
for(int i = 10;i >= 0;)i--;
```

【例 3.9】 用 for 循环语句按 10 度的增量打印出一个从摄氏 10~90 度到华氏温度的转换表。

```
class TempConversion{
  public static void main (String args[]){
     int fahr,cels;
```

```
        System.out.println("摄氏温度    华氏温度");
      for(cels = 10;cels <= 90;cels += 10){
        fahr = cels * 9 / 5 + 32;
         System.out.println("    " + cels + "            " + fahr);
        }
      }
}
```

程序的运行结果如下：

```
摄氏温度    华氏温度
    10          50
    20          68
    30          86
    40          104
    50          122
    60          140
    70          158
    80          176
    90          194
```

在理解了 for **语句的功能后，可灵活地使用该语句的各种形式来编写循环程序。例如，对温度变换的例** 3.9，for **语句可改写为下面几种形式（只写出有关部分）。**

（1）省略表达式 expression1

若 for **语句的初值部分在** for **循环外已全部设置，则** for **语句的相关部分可省略。**

```
    int fahr,cels = 10;
    System.out.println("摄氏温度    华氏温度");
   for(;cels <= 90;cels += 10){
     fahr = cels * 9 / 5 + 32;
      System.out.println("    " + cels + "            " + fahr);
    }
```

（2）在表达式 expression1 **中包含变量定义等更多的内容**

在 for **语句的初值部分可包含变量定义（仅在** for **循环中使用）、赋值等内容，各项之间用逗号分隔。**

```
   for(int fahr,cels = 10;cels <= 90;cels += 10){
     fahr = cels * 9 / 5 + 32;
      System.out.println("    " + cels + "            " + fahr);
    }
```

（3）省略表达式 expression3

将表达式 expression3 **写到循环体中，即省略了** expression3 **部分。**

```
for(int fahr,cels = 10;cels <= 90;){
  fahr = cels * 9 / 5 + 32;
  System.out.println("   " + cels + "          " + fahr);
  cels += 10;
}
```

（4）**在表达式** expression3 **中包含更多内容**

将循环体的部分或全部写到表达式 expression3 **部分，** Java **也能正确执行。**

```
for(int fahr,cels = 10;cels <= 90;
  fahr = cels * 9 / 5 + 32,      // 下面 3 行都属于表达式 expression3
   System.out.println("   " + cels + "          " + fahr),
   cels += 10)                   // 循环体为空
}
```

（5）**同时省略表达式** expression1 **和表达式** expression3

```
int fahr,cels = 10;
for(;cels <= 90;){
  fahr = cels * 9 / 5 + 32;
   System.out.println("   " + cels + "          " + fahr);
   cels += 10;
}
```

（6）**同时省略所有表达式** expression

当所有表达式都省略时，为 for **的无限循环形式，这时，可在循环体中添加其他控制语句来终止循环。**

```
int fahr,cels = 10;
for(;;){
  if(cels > 90)break;
  fahr = cels * 9 / 5 + 32;
   System.out.println("   " + cels + "          " + fahr);
   cels += 10;
}
```

当然，上述 for **的几种形式还有其他的写法。在充分理解** for **语句功能的基础上，写出** for **语句的更多种形式也是不难的。善于使用** for **循环语句，可使程序简洁清晰。**

3.3.4 循环嵌套

循环嵌套是指在某个循环语句的循环体中又包含另一个循环语句，也称多重循环。外面的循环语句称为“外层循环”，外层循环的循环体中的循环称为“内层循环”。

设计循环嵌套结构时，要注意内层循环语句必须完整地包含在外层循环的循环体中，不得出现内外层循环体交叉的情况。 Java **语言中的三种循环语句都可以组成多重循环。**

【例 3.10】　求 3～200 之间的所有素数。

```
public class PrimeNumber{
  public static void main(String args[]){
    System.out.println(" ** 3～200 之间的所有素数 **");
    int n = 0;
    for(int i = 3;i < 200;i += 2){ // 外层循环
      int k = (int)Math.sqrt(i);
      boolean isPrime = true;
      for(int j = 2;j <= k;j ++ ){ // 内层循环
        if( i%j == 0 )isPrime = false;
      }
      if(isPrime){
        System.out.print(" " + i);
        n ++;          // 控制每行打印 10 个素数
        if( n % 10 == 0 )System.out.println();
      }
    }
    System.out.println();
  }
}
```

程序运行结果如下：

```
** 3～200 之间的所有素数 **
 3 5 7 11 13 17 19 23 29 31
 37 41 43 47 53 59 61 67 71 73
 79 83 89 97 101 103 107 109 113 127
 131 137 139 149 151 157 163 167 173 179
 181 191 193 197 199
```

该程序结构是一个二重循环：内层循环对数据 *i* 判断是否为素数，若不是素数，则赋 isPrime 值为 false；外层循环使得数据 *i* 从 3 至 200 变化。两层循环配合就可求出 3 至 200 之间的所有素数。

3.4　标号和其他流程控制语句

3.4.1　标号

标号是一个标识符，用于给某程序块一个名字。格式如下：

```
label:{codeBlock}
```

label 是标号名，用标识符表示。标号名用冒号与其后面的语句（块）分开。

【例 3.11】　给 for 循环块一标号 Loop。

```
Loop:for(int i = 0,sum = 0;i < 10;i ++)sum += i;
```

由于 Java 语言未提供 goto 语句，故标号不与 goto 一起使用。下面介绍的 break 语句和 continue 语句经常需要使用标号。

3.4.2　break 语句

break 语句和下一节的 continue 语句可以看成结构化的 goto 语句。break 语句的功能是终止执行包含 break 语句的一个程序块。break 语句除了可应用于前面介绍的 switch 语句中外，还可应用于各种循环语句中。break 语句的格式如下：

```
break [label];
```

break 有两种形式：不带 label 和带 label。label 是标号名，它必须位于 break 语句所在的封闭语句块的开始处。

【例 3.12】　用 break 终止循环。

```
class BreakLoop{
  public static void main(String args[]){
    for(int i=0; i<100; i++) {
        if(i == 5) break;          // 若 i 为 5 则终止循环
      System.out.println("i: " + i);
     }
     System.out.println("Loop 完成。");
  }
}
```

程序运行结果如下：

```
i: 0
i: 1
i: 2
i: 3
i: 4
Loop 完成。
```

在这个例子中，循环测试表达式是 *i*<100，由于在循环体中加入了分支语句，当 *i* 的值为 5 时，执行到 break 语句，它使得程序控制跳出循环，继续执行 for 循环以后的语句。

break 语句只终止执行包含它的最小程序块，而有时希望终止更外层的块，用带标号的 break 语句就可实现这种功能，它使得程序流程控制转移到标号指定层次的结尾。

【例 3.13】　用 break 终止外层循环。

```
class BreakDemo {
  public static void main(String args[]) {
    boolean t = true;
```

```
        first: {          // 定义块 first
          second: {     // 定义块 second
            third: {   // 定义块 third
              System.out.println("在 break 之前。");
              if(t) break second; // break 终止 second 块
              System.out.println("本语句将不被执行。");
            }
            System.out.println("本语句将不被执行。");
          }
          System.out.println("在 second 块后的语句。");
        }
      }
    }
```

程序运行结果如下：

```
在 break 之前。
在 second 块后的语句。
```

标号提供了一种简单的 break **语句所不能实现的控制循环的方法，当在循环语句中遇到** break **时，不管其他控制变量，都会终止。正常的** break **只退出一重循环，可以用标号标出想退出的任一重循环。**

3.4.3 continue 语句

continue **语句只能用在循环中，它的功能是使得程序跳过循环体中** continue **语句后剩下的部分（即短路），终止当前这一轮循环的执行。** continue **语句的格式如下：**

```
continue [label];
```

continue **语句有带标号和不带标号两种形式。不带标号的** continue **语句在** while **或** do-while **语句中使流程直接跳到循环条件的判断上；在** for **语句中则直接计算表达式** 3 **的值，再根据表达式** 2 **的值是** true **或** false **决定是否继续循环。**

【例 3.14**】 以每行两个数据的格式输出数字** 0～9**。**

```
class ContinueDemo {
  public static void main(String args[]) {
    for(int i = 0; i < 10; i ++) {
      System.out.print(i + " ");
      if (i % 2 == 0) continue;
      System.out.println(); // 本语句在执行到 continue 时被跳过
    }
  }
}
```

程序运行结果如下：

```
0 1
2 3
4 5
6 7
8 9
```

这里当然也可以不用 continue**，只要改写成如下形式**

```
if(i  % 2 != 0)System.out.println();
```

即可。continue **语句提供另一种风格，也就是若** continue **后面多于一个语句，它比用** if **省掉一对花括号和一层嵌套。**

continue **语句和** break **语句一样，也可以与标号结合使用。这个标号名必须放在循环语句之前，用于标志这个循环体。执行了内循环体的** continue **语句后，将进行由标号标明的循环语句的下一轮循环。**

【例 3.15**】　带标号的** continue **语句。**

```
class ContinueLabel {
   public static void main(String args[]) {
      outer: for (int i = 0; i < 10; i ++) {
         for(int j = 1; j < 10; j ++) {
            if(j >= i) {
               System.out.println();
               continue outer;
            }
            System.out.print((i * j) + " ");
         }
      }
      System.out.println();
   }
}
```

程序运行结果如下：

```
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
```

在这个例子中，当满足 j>=i 的条件时，程序执行完相应的语句后跳转到外层循环，执行外层循环的迭代语句 i++，然后开始下一次循环。

3.4.4 return 语句

return 语句的功能是从当前方法中退出，返回到调用该方法的语句处，并从紧跟该语句的下一语句继续程序的执行。return 语句的格式如下：

```
return [expression];
```

或

```
return([expression]);
```

当用 void 定义了一个返回值为空的方法时，方法体中不一定要有 return 语句，程序执行完，它自然返回。若要从程序中间某处返回，则可使用 return 语句。若一个方法的返回类型不是 void 类型，那么就用带表达式 expressions 的 return 语句。表达式的类型应该同这个方法的返回类型一致或小于返回类型。例如，一个方法的返回类型是 double 类型时，return 语句表达式的类型可以是 double、float 或者是 short、int、byte、char 等。例如：

```
double exam(int x,double y,boolean b){
  if(b)
    return x;
  else
    return y;
}
```

3.5 方法的使用

通过前面介绍的程序可以看到，对简单的问题，程序也比较简单：一个程序是一个类，在类中包含一个 main()方法。但在解决一些比较复杂的问题时，按照现代模块化程序设计的思想，应仔细分析问题，善于将这些复杂问题分解成若干相对简单的问题，即划分为多个模块。这样，解决一个复杂问题就转化为逐个解决一些简单问题。对程序开发和维护而言，小模块比大程序更便于管理。

在 Java 语言中，类和方法就是程序的模块。在一个类中可以根据需要设计多个方法。在本节中，先介绍 Java 语言中的方法。在程序设计时，可将一个程序中完成特定功能的程序段定义为方法（类似其他语言的函数）。在需要使用这些功能时，可调用相应的方法，特别是在某些功能多次被使用时，采用方法可大大提高程序代码的可复用性。使用方法要掌握方法定义、方法调用、方法参数传送等方面的内容。

3.5.1 方法的定义与调用

1. 方法的定义

方法的定义是描述实现某个特定功能所需的数据及进行的运算和操作。定义形式如下：

```
[modifier] returnType methodName([Parameter list]){
  // methodBody 方法体
}
```

其中，用方括号括住的项目是可选的。方法的类型 returnType 指的是方法的返回值类型，若方法完成的功能是计算值，则计算结果值或计算值的表达式一般要书写到方法体里的 return 语句中，而且类型一般应与 returnType 指明的类型一致。返回值类型可以是基本类型、数组、类等。若方法完成的功能不返回值，returnType 处应为 void，而且方法体中的 return 语句不能带表达式或不用 return 语句。

方法名 methodName 是一个标识符，是对方法的命名。

可选的参数表必须用圆括号括起来，参数表 Parameter list 由 0 个或多个用逗号分隔的参数构成，每个参数由类型和参数名（标识符）组成，参数可以是基本数据，也可以是数组或类实例，参数类型可以是基本数据类型或类。方法用参数来与外界发生联系（数据传送）。

方法定义中{}括起来的部分称为方法体，在其中书写方法的实现语句，包括数据定义和执行语句。

方法定义前面的修饰符 modifier 用关键字表示，修饰符是可选的，用来说明方法的某些特性。可用的修饰符有 public、static、private 等。

Java 语言允许一个类中定义多个方法，方法定义形式为并列形式，先后顺序无关紧要。

【例 3.16】 定义一个计算圆面积的方法 area()。计算圆面积需要知道圆的半径 *r*，area()方法应从外部得到这个 *r*，所以可将 *r* 设置为一个参数，类型为 double。area()方法将计算出一个面积值，类型也应为 double，可将它设置为 area()返回值类型。area()方法可定义如下：

```
double area(double r){
  double s = Math.PI * r * r; // 方法参数在方法体中可直接引用
   return(s);
}
```

【例 3.17】 定义一个求三个整数中最大数的方法 max3()。该方法需要三个整数，因此，需设置三个整数参数。max3()的返回结果是一个整数值（即最大整数），可设置方法返回值类型为整型。max3()可定义如下：

```
int max3(int x,int y,int z){
  int big;
  big = Math.max(x,y);
  big = Math.max(big,z);
   return(big);
}
```

2，方法的调用

在程序中需要某个方法的功能时，要调用该方法。调用方法时，要用实际参数替换方法定义中的参数表中的形式参数（或称虚拟参数）。实际参数（简称实参）的个数、类型、顺序都必须与形式参数（简称形参）一致。

【例 3.18】　调用例 3.17 中定义的 max3()方法求变量 *a*、*b* 和 *c* 中的最大值。

```
class MethodDemo{
  public static void main(String args[]){
    int a = 4,b = 5,c = 2,big;
     big = max3(a,b,c); // 用 3 个实参调用方法 max3，方法的返回值存入 big 变量
      System.out.println(big);
    }
   static int max3(int x,int y,int z){   //方法 max3 有 3 个形式参数
      int big;
      big = Math.max(x,y);
      big = Math.max(big,z);
       return(big);
    }
}
```

在上例中，方法的返回值类型为基本类型，在调用方法中，用一个与方法返回值相同类型的变量 big 来接收返回值。也可以对方法调用的返回值直接输出，如上例中输出三个整数中的最大数，可改为 System.out.println(max3(a,b,c))。

注意：与例 3.17 不同，在方法 max3()的声明中用了修饰符 static，它说明该方法是一个类方法。与类方法 main()相同，类方法可以直接调用，而不需要创建实例对象，若方法未用 static 修饰符修饰，这个方法就是实例方法。实例方法不能像类方法那样被直接调用。

一般而言，有返回值的方法调用形式为表达式形式，即可以在允许表达式出现的地方使用方法调用。若方法无返回值（void 类型），方法调用的形式一般为表达式语句的形式，即调用方法的形式为单独的加分号的语句。

3.5.2　方法调用中的数据传送

调用方法与被调方法之间往往需要进行数据传送。例如，调用方法传送数据给被调方法，被调方法得到数据后进行计算，计算结果再传送给调用方法。一般说来，方法间传送数据有如下的几种方式：值传送方式、引用传送方式、返回值方式、实例变量和类变量传送方式。方法的参数可以是基本类型的变量、数组和类对象等。通过实参与形参的对应，数据传送给方法体使用。

1，值传送方式

值传送方式是将调用方法的实参的值计算出来赋予被调方法对应形参的一种数据传送方式。在这种数据传送方式下，被调方法对形参的计算、加工与对应的实参已完全脱离关系。当被调方法执行结束，形参中的值可能发生变化，但是返回后，这些形参中的值将不会带到对应的实参中。因此，值传送方式的特点是“数据的单向传送”。

使用值传送方式时，形式参数一般是基本类型的变量，实参是常量、变量，也可以是表达式。

2，引用传送方式

使用引用传送方式时，方法的参数类型一般为复合类型（引用类型），复合类型变量中存储的是对象的引用。所以在参数传送中是传送引用，方法接收参数的引用，因此任何对形参的改变都会影响到对应的实参。因此，引用传送方式的特点是“引用的单向传送，数据的双向传送”。

3，返回值方式

返回值方式不是在形参和实参之间传送数据，而是被调方法通过方法调用后直接返回值到调用方法中。使用返回值方式时，方法的返回值类型不能为 void，且方法体中必须有带表达式的 return 语句，其中表达式的值就是方法的返回值。在例 3.18 中，方法 max3()就是利用返回值方式将 3 个数据的最大值回送到 main()方法中的。这个方法的返回值为 int 类型，所以在定义方法时，方法的类型要定义为 int 类型。

4. 实例变量和类变量传送方式

实例变量和类变量传送方式也不是在形参和实参之间传送数据，而是利用在类中定义的实例变量和类变量是类中诸方法共享的变量的特点来传送数据。因类变量（static 变量）可直接访问，使用较简单，下面是一个使用类变量的例子。

【例 3.19】　求半径为 1.23 高为 4.567 的圆柱体体积。

```
public class CircleVolume{
   static double r = 1.23,h = 4.567,s,v;          // 定义类变量
   public static void main(String args[]){
      area();                                      // 调用 area 方法
      vol();                                       // 调用 vol 方法
      System.out.println("半径为 1.23 的圆面积 = " + s);
      System.out.println("半径为 1.23 高为 4.567 的圆柱体体积 = " + v);
   }
   static void area(){
      s = Math.PI*r*r;                  // 访问类变量 r 和 s
   }
   static void vol(){
      v = s * h;                        // 访问类变量 s、h 和 v
   }
}
```

程序的运行结果如下：

```
半径为 1.23 的圆面积 = 4.752915525615998
半径为 1.23 高为 4.567 的圆柱体体积 = 21.7065652054882
```

本程序利用类变量使得 main()方法、area()方法和 vol()方法之间传送了数据。

3.5.3 方法和变量的作用域

在 Java 语言中，方法与变量的作用域（有效使用范围）是清晰的，根据定义变量的位置不同，作用域也不全相同。当一个方法使用某个变量时，以如下的顺序查找变量：

当前方法、当前类，一级一级向上经过各级父类、import 类和包，若都找不到所要的变量定义，则产生编译错误。

下面对方法和变量作用域的讨论限于在一个类中的情况。多个类的情况见第 5 章相关章节。

1. 局部变量

局部变量是定义在块内、方法内的变量。这种变量的作用域是以块和方法为单位的，仅在定义该变量的块或方法内有效，而且要先定义赋值，然后再使用，即不允许超前引用。因为局部变量在查找时首先被查找，因此若某一局部变量与类的实例变量名或类变量名相同时，则该实例变量或类变量在方法体内被暂时“屏蔽”起来，只有退出这个方法后，实例变量或类变量才起作用。

【例 3.20】 说明实例变量和局部变量同名。

```
class A{
   int x = 8;    //  实例变量
   void f(){
      int x = 6;  //  局部变量与实例变量同名，屏蔽了实例变量
      System.out.println(" x = " + x);
   }
}
```

方法 f 输出的结果为：

```
x = 6（注：这个程序不能直接运行）
```

每调用一次方法，都要动态地为方法的局部变量分配内存并初始化。方法体内不能定义静态变量。方法体内的任何语句块内都可以定义新的变量，这些变量仅在定义它的语句块内起作用。当语句块有嵌套时，内层语句块定义的变量不能与外层语句块的变量同名，否则会出现编译错误。另外，方法的参数也属于局部变量，因此声明与参数同名的局部变量也会出错。

2. 实例变量和类变量

定义在类内、方法外的变量是实例变量，使用了修饰符 static 的变量是静态变量（或称类变量）。实例变量和类变量的作用域是以类为单位的。因为实例变量、类变量与局部变量的作用域不同，故可以与局部变量同名。

【例 3.21】 说明局部变量和实例变量、类变量。

```
class MyObject {
   static short s = 400;  //  类变量
```

```
    int i = 200;  // 实例变量
    void f() {
      System.out.println("s = " + s);
      System.out.println("i = " + i);
      short s = 300;      // 局部变量
      int i = 100;        // 局部变量，与实例变量同名
      double d = 1E100;  // 局部变量
      System.out.println("s = " + s);
      System.out.println("i = " + i);
      System.out.println("d = " + d);
    }
}
class LocalVariables {
    public static void main(String args[]) {
      MyObject myObject = new MyObject();
      myObject.f();
    }
}
```

程序运行结果如下：

```
s = 400
i = 200
s = 300
i = 100
d = 1.0E100
```

一般情况下，变量应该先定义后使用。但实例变量和静态变量可以超前引用，即在定义位置前引用，但静态变量不能超前引用静态变量。例如：

```
class B{
    static int x;
    static int z = y + 1;  // 静态变量不能超前引用静态变量，错
    int c = 0;
    void a(){
      int a = b + y;        // 超前引用 b 和 y，对
      System.out.println(" a = "+ a);
    }
    int b = 2;
    static int y;
    int c = b;                // c 重复声明，错
    int x = 0;                // x 重复声明，错
}
```

3，方法

实例方法和类方法在整个类内均是可见的，可以超前引用。除方法的重载外，声明两个同名实例方法或类方法是错误的，类方法与实例方法同名也是错误的。例如：

```
class C{
   void a(int x){
      b(x);                    // 超前引用方法 b()，对
   }
   void  a(){                  // 方法重载，对
      x = 0;                   // 超前引用实例变量，对
   }
   void b(int x){...}
   void  a(int  x){            // 方法 a(int x)重复声明了，错
      ...
   }
   static  void  b(int  x){    // 修饰符不能作为方法重载的标志，错
      ...
   }
   static  void  b(){          // 方法重载，对
      ...
   }
}
```

3.5.4 方法的嵌套和递归调用

1，嵌套调用

在一个方法的调用中，该方法的实现部分又调用了另外的方法，则称为方法的嵌套调用。

【例 3.22】　求 $C_n^x = \frac{n!}{x!(n-x)!}$，**当** $n = 11$，$x = 4,6,8,10$ **时的值。**

```
public class MethodNestDemo{
   public static void main(String args[]){
      int n = 11,x;
      for(x = 4;x < 11;x += 2)
         System.out.println("C(" + n + "," + x + ") = " + comb(n , x));
   }
   static int comb(int n,int x){
      return( fact( n ) / fact( x ) / fact( n – x ));
   }
   static int fact(int n){
```

```
        if(n == 1)
            return 1;
        else
            return fact(n - 1) * n;
    }
}
```

程序运行结果如下：

```
C(11,4) = 330
C(11,6) = 462
C(11,8) = 165
C(11,10) = 11
```

程序中包含三个类方法，在 main()**类方法中调用** comb()**类方法，而这个方法又调用** fact() **类方法，从而形成了方法调用嵌套。**

2. 递归

在一个方法有调用该方法自身的情况时，称为方法的递归调用。大多数情况是直接递归，即一个方法直接自己调自己。当一个实际问题可用递归形式描述时，该问题的求解用递归方法变得容易。例如，求 $n!$**，求前** n **个自然数的和，求第** n **个** Fibonacci **数等，它们的递归描述分别为：**

**** n **的阶乘：**

$$n! = \begin{cases} 1 & n = 1 \\ n(n-1)! & n > 1 \end{cases}$$

前 n **个自然数之和：**

$$s(n) = \begin{cases} 1 & n = 1 \\ n + s(n-1) & n > 1 \end{cases}$$

第 n **个** Fibonacci**数：**

$$f(n) = \begin{cases} 1 & n = 1,2 \\ f(n-1) + f(n-2) & n > 2 \end{cases}$$

根据上述描述，容易写出对应递归方法。在例 3.22 **中求阶乘的方法** fact **即为递归方法。下面是求第** n **个** Fibonacci**数的例子。**

【例 3.23**】　用递归调用求第** n **个** Fibonacci**数的方法求前** 20 **个** Fibonacci**数。**

```
public class Fibo{
    public static void main(String args[]){
        final int n = 20;
        for(int i = 1;i <= n;i++){
            System.out.print(f(i)+" ");
            if(i % 10 == 0)System.out.println();
        }
    }
    static long f(long n){
```

```
        if(n == 1||n == 2)return 1;
        else return f(n - 1) + f(n - 2);
    }
}
```

程序的运行结果如下：

```
1 1 2 3 5 8 13 21 34 55
89 144 233 377 610 987 1597 2584 4181 6765
```

从程序设计的角度来说，递归方法必须解决两个问题：一是递归计算的公式，二是递归结束的条件。对求第 *n* 个 Fibonacci数列的递归方法来说，这两个条件是：

递归计算公式： fib(*n*) = fib(*n*-1) + fib(*n*-2)

递归结束条件： fact(1) = 1

3.5.5 方法的重载

Java **语言允许在一个类中定义几个同名的方法，但要求这些方法具有不同的参数集合，即方法参数的个数、类型和顺序要不同。这种做法称为方法的重载。当调用一个重载的方法时，** Java **编译器可根据方法参数的个数、类型和顺序的不同，来调用相应的方法。**

Math **类中的方法** abs()、max()**和** min()**等都是重载的，它们具有** double、float、int、long **等参数和返回值类型的重载方法。** Java **根据方法名及参数集合的不同来区分不同的方法，若调用的两个同名方法中参数个数、类型及顺序均一样，仅仅返回值类型不同，则编译时会产生错误。**

重载方法可以有不同类型的返回值。

【例 3.24**】 用方法重载求圆、矩形、梯形面积。**

```
class Area{
    static double area(double r){
        return Math.PI * r * r;
    }
    static double area(double l,double w){
        return l * w;
    }
    static double area(double d1,double d2,double h){
        return (d1 + d2) * h / 2;
    }
    public static void main(String args[]){
        double s1 = area(3.0);
        System.out.println("圆面积 = " + s1);
        double s2 = area(3.0,4.0);
        System.out.println("矩形面积 = " + s2);
        double s3 = area(3.0,4.0,5.0);
```

```
        System.out.println("梯形面积 = " + s3);
    }
}
```

程序运行结果如下：

```
圆面积 = 28.274333882308138
矩形面积 = 12.0
梯形面积 = 17.5
```

在本例中，三个求面积的方法名都相同，但参数个数不同，在 main **方法中用不同的参数个数去调用** area **方法，可求不同的面积。**

3.6 异常处理

Java **是一个讲究安全性的语言。任何可能在程序运行过程中产生打断程序正常执行的事件都有用来保护的陷阱。**Java **异常处理机制提供了去检查及处理产生各种错误、异常事件的方法。**

3.6.1 异常概述

1. **异常的概念**

用 Java **的术语来说，在程序运行过程中发生的、会打断程序正常执行的事件称为异常（**Exception**），也称为例外。程序运行过程中可能会有许多意料之外的事情发生，例如，零用做了除数（产生算术异常** ArithmeticException**），在指定的磁盘上没有要打开的文件（产生文件未找到异常** FileNotFoundException**），数组下标越界（产生数组下标越界异常** ArrayIndexOutOfBoundsException**）等。对于一个实用的程序来说，处理异常的能力是一个不可缺少的部分，它的目的是保证程序在出现任何异常的情况下仍能按照计划执行。**

下面先来看一个 Java **系统对异常处理的例子。**

【例 3.25**】** Java **系统对异常的处理。**

```
public class SystemException{
    public static void main(String args[]){
        int a = 68;
        int b = 0;
        System.out.println(a / b);  // 0 用做了除数
    }
}
```

程序运行结果为：

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at SystemException.main(SystemException.java:5)
```

屏幕显示的信息指明了异常的类型：ArithmeticException: / by zero（**算术异常/用** 0 **除**）。

在这个程序中，未进行程序的异常处理。这是因为除数为零是算术异常，它属于运行时异常（RunTimeException），通常运行时异常在程序中不做处理，Java 运行时系统能对它们进行处理。Java 语言本身提供的 Exception 类已考虑了多种异常的处理。

2. Java 对异常的处理机制

Java 异常处理机制提供了一种统一和相对简单的方法来检查及处理可能的错误。例如，在程序中出现了除以零的错误时，即抛出（throw）一个 ArithmeticException 异常实例，异常处理程序捕获（catch）这个异常实例，并对它进行处理。如果系统找不到相应的异常处理程序，则由 Java 默认的异常处理程序来处理，即在输出设备上输出异常信息，同时程序停止运行。有时也可以在程序中写一些自己的例程来处理异常；也可以将错误交给 Java 系统去处理，从而可以使得程序安全地退出。

3. 异常类的层次和主要子类

Java 用面向对象的方法处理异常，所有的异常都是 Throwable 类的子类生成的对象，所有的异常类都是 Throwable 类的后代。Throwable 类的父类是 Java 的基类（Object），它有两个直接子类：Error 类和 Exception 类。运行时异常 RuntimeException 类是 Exception 类的子类。只有 Throwable 类的后代才可以作为一个异常被抛出。Java 语言的异常处理的主要子类见表 3.1～表 3.3。

表 3.1　Error 类

子　类　名	说　　明
AbstractMethodError	调用抽象方法错误
ClassFormatError	类文件格式错误
IllegalAccessError	非法访问错误
IncompatibleClassChangeError	非法改变一个类错误
InstantiationError	实例化一个接口或抽象类错误
InternalError	Java 内部错误
LinkageError	连接失败错误
NoClassDefFoundError	找不到类定义错误
NoSuchFieldError	域未找到错误
NoSuchMethodError	调用不存在的方法错误
OutOfMemoryError	内存溢出错误
StackOverflowError	堆栈溢出错误
ThreadDeadError	线程死亡错误
UnknownError	未知错误
UnstatisfiedLinkError	链接不满足错误
VerifyError	校验失败错误
VirtualMachineError	虚拟机错误

表 3.2　Exception 类

子　类　名	说　　明
ClassNotFoundException	类未找到异常
DataFormatException	数据格式异常
IllegalAceessException	非法存取异常
InstantiationException	实例化异常
InterruptedException	中断异常
NoSuchMethodException	调用不存在方法异常
RuntimeException	运行时异常

表 3.3　RuntimeException 类

子　类　名	说　　明
ArithmeticException	算术异常
ArrayIndexOutOfBoundException	数组越界异常
ArrayStoreException	数组存储异常
ClassCastException	类强制转换异常
IllegalArgumentException	非法参数异常
IllegalThreadstateException	非法线程状态异常
IndexOutOfBoundException	索引越界异常
NegativeArraySizeException	负值数组大小异常
NullPointerException	空引用异常
NumberFormatException	数值格式异常
SecurityException	安全异常
StringIndexOutOfBoundException	字符串越界异常

4，异常类的方法和属性

（1）异常类的构造方法

构造方法是一类方法名与类名相同的方法，用于创建并初始化该类的对象。Exception 类有四个重载的构造方法，常用的两个构造方法为：

- public Exception()创建新异常。
- public Exception(String message)用字符串参数 message 描述异常信息，创建新异常。

（2）异常类的方法

Exception 类常用的方法有：

- public String toString()返回描述当前异常对象信息的字符串。
- public String getMessage()返回描述当前异常对象的详细信息。
- public void printStackTrace()在屏幕上输出当前异常对象使用堆栈的轨迹，即程序中先后调用了哪些方法，使得运行过程中产生了这个异常对象。

【例 3.26】　运行时异常产生时的信息显示。

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

```
        String   s = "123.45";
        methodA(s);
    }
    static void methodA(String s) {
        Integer i = new Integer(s);
            System.out.println(i);
    }
}
```

程序运行结果如下：

```
Exception in thread "main" java.lang.NumberFormatException: 123.45
    at java.lang.Integer.parseInt(Integer.java:438)
    at java.lang.Integer.<init>(Integer.java:570)
    at MyClass.methodA(MyClass.java:7)
    at MyClass.main(MyClass.java:4)
```

本程序运行结果说明，程序运行时产生一个 NumberFormatException **数值格式异常。在用构造方法** Integer **将一个字符串转换为** Integer **数据时，参数字符串格式不对，所以产生了这个运行时异常，**Java **系统（即系统调用方法** printStackTrace()**将调用堆栈的轨迹打印了出来。输出的第一行信息也是** toString()**方法输出的结果，对这个异常对象进行简单说明。其余各行显示的信息表示了异常产生过程中调用的方法，最终是在调用** Integer.parseInt()**方法时产生的异常，调用的出发点在** main()**方法中。**

3.6.2 异常处理

在 Java **语言中，异常有以下几种处理方式：**

（1**）可以不处理运行时异常，由** Java **虚拟机自动进行处理。**

（2**）使用** try-catch-finally **语句捕获异常。**

（3**）通过** throws **子句声明抛弃异常，还可以自定义异常，用** throw **语句来抛出它。**

1，运行时异常

运行时异常是 Java **运行时系统在程序运行中检测到的，可能在程序的任何部分发生，而且数量可能较多，如果逐个处理，工作量很大，有可能影响程序的可读性及执行效率。因此，**Java **编译器允许程序不对运行时异常进行处理，而将它交给默认的异常处理程序，一般的处理方法是在屏幕上输出异常的内容以及异常的发生位置。如例** 3.25 **和例** 3.26 **所示。当然，在必要的时候，也可以声明、抛出、捕获运行时异常。**

2，try-catch-finally 语句

在 Java **语言中，允许自己来处理异常。**Java **语言提供** try-catch-finally **语句来捕获和处理异常，该语句的格式如下：**

```
try{
```

```
    statements                          // 可能产生异常的语句
  }catch(Throwable-subclass e){         // 异常参数
    statements                          // 异常处理程序
  }catch(Throwable-subclass e){         // 异常参数
    statements                          // 异常处理程序
  }…
  finally{
    statements
  }
```

try 语句块中是可能产生异常对象的语句，一旦其中的语句产生了异常，程序即跳到紧跟其后的第一个 catch 子句。try 程序块之后的 catch 子句可以有多个，也可以没有。

catch 子句中都有一个代表异常类型的形式参数，这个参数指明了这个 catch 程序块可以捕获并处理的异常类型。若产生的异常类型与 catch 子句中声明的异常参数类型匹配，则执行这个 catch 程序块中的异常处理程序。匹配的意思是指异常参数类型与实际产生的异常类型一致或是其父类。若不匹配，则顺序寻找下一个 catch 子句，因此 catch 语句的排列顺序应该从特殊到一般，否则，放在后面的 catch 语句将永远执行不到。

也可以用一个 catch 语句处理多个异常类型，这时它的异常类型参数应该是这多个异常类型的父类。

若所有 catch 参数类型与实际产生的异常类型都不匹配，则标准异常处理程序将被调用，即在输出设备上输出异常信息，同时程序停止运行。

【例 3.27】 捕获除数为零的异常，并显示相应信息。

```
class ExceptionDemo1{
  public static void main(String args[]) {
    int d, a;
    try { // 监控可能产生异常的代码块
      d = 0;
      a = 68 / d;
      System.out.println("本字符串将不显示。");
    } catch (ArithmeticException e) { // 捕获 divide-by-zero 错误
      System.out.println("产生用零除错误。");
    }
    System.out.println("在捕获语句后。");
  }
}
```

程序运行结果如下：

```
产生用零除错误。
在捕获语句后。
```

【例 3.28】 多个 catch 子句的 try 语句。

```
class ExceptionDemo2 {
  public static void main(String args[]) {
    try {
      int a = args.length;
      System.out.println("a = " + a);
      int b = 42 / a;
      int c[] = { 1 };
      c[4] = 99;
    }catch(ArithmeticException e) {  // 捕获算术运算异常
      System.out.println("Divide by 0: " + e);
    }catch(ArrayIndexOutOfBoundsException e) { // 捕获数组下标越界异常
      System.out.println("Array index oob: " + e);
    }
    System.out.println("After try/catch blocks.");
  }
}
```

程序运行结果如下：

```
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
```

catch **子句中异常参数的声明原则是从特殊到一般，若将一般（范围宽）的异常参数放到了前面，特殊（范围窄）的异常参数放到了后面，编译系统会指出下列错误：**

```
.....: catch not reached.
```

这是提示后面的 catch **子句根本不会被执行到，因为它能捕获的异常已经被前面的** catch **子句捕获了。**

try **语句中的** finally **子句的作用是说明必须执行的语句，无论** try **程序块中是否抛出异常，** finally **程序块中的语句都会被执行到。**

【例 3.29**】　有** finally **子句的** try **语句。**

```
class ExceptionDemo3{
  public static void main(String args[]){
    try{
      int x=0;
      int y=20;
      int z=y/x;
      System.out.println("y/x 的值是 :"+z);
    }catch(ArithmeticException e){
      System.out.println("捕获到算术异常：" + e);
    }finally{
```

```
            System.out.println("执行到 finally 块内!");
            try{
               String name = null;
               if(name.equals("张三")){   // 字符串比较，判断 name 是否为"张三"
                  System.out.println("我的名字叫张三。");
               }
            }catch(Exception e){
               System.out.println("又捕获到异常：" + e);
            }finally{
               System.out.println("执行到内层 finally 块内!");
            }
         }
      }
   }
```

程序运行结果如下：

```
捕获到算术异常：java.lang.ArithmeticException: / by zero
执行到 finally 块内!
又捕获到异常：java.lang.NullPointerException
执行到内层 finally 块内!
```

在 Java 语言中，try-catch-finally 语句允许嵌套。本例中是将内层的 try 嵌套在外层的 finally 块内。在程序执行到外层的 try 程序块时，由于分母为零而产生了算术异常，所以程序转移到第一个 catch 块。该 catch 捕获了这个算术异常，并进行了处理，之后程序转向必须执行的外层的 finally 程序块。因为该 finally 块产生空指针异常（一个 null 字符串和字符串“张三”进行比较），所以内层 catch 子句再次捕获到异常，最后程序转移到内层的 finally 程序块。

finally 块还可以和 break、continue 以及 return 等流程控制语句一起使用。当 try 程序块中出现了上述这些语句时，程序必须先执行 finally 程序块，才能最终离开 try 程序块。

【例 3.30】 break 与 finally 的联连。

```
class BreakAndFinally{
   public static void main(String args[]){
      for(;;)
         try{
            System.out.println("即将退出循环了!");
            break;
         }finally{
            System.out.println("finally 块总要被执行到!");
         }
   }
}
```

程序运行结果如下：

```
即将退出循环了！
finally 块总要被执行到！
```

3，throw 语句和 throws 子句

throw 语句可使得用户自己根据需要抛出异常。throw 语句以一个异常类的实例对象作为参数。一般的形式是：

```
methodName（arguments）throws 异常类{
  ...
  throw new 异常类();
}
```

new 运算符被用来生成一个异常类的实例对象。例如：

```
throw new ArithmeticException();
```

包含 throw 语句的方法要在方法头参数表后书写 throws 子句。它的作用是通知所有要调用此方法的其他方法，可能要处理该方法抛弃的那些异常。若方法中的 throw 语句不止一个，throws 子句应指明抛弃的所有可能产生的异常。

通常使用 throws 子句的方法本身不处理本方法中产生的异常，声明抛弃异常使得异常对象可以从调用栈向后传播，直到有合适的方法捕获它为止。对未用 throw 语句产生系统异常的方法，也可以使用 throws 子句来声明抛弃异常。例 2.9 和例 2.10 就是这种情况，在 main() 方法中不对异常进行处理。

为了能捕获 throw 抛出的异常，应在 try 块中调用包含 throw 语句的方法。

throw 语句和 throws 子句的使用见下例。

【例 3.31】 throw 语句和 throws 子句的使用。

```
class ThrowDemo{
  void inException()throws ArithmeticException{
    throw new ArithmeticException();
  }
  public static void main(String args[]){
    ThrowDemo s = new ThrowDemo();
    try{
      s.inException();
    }catch(Exception e){
      System.out.println("异常来了：" + e);
    }
  }
}
```

程序运行结果如下：

异常来了：java.lang.ArithmeticException

程序中通过 throw 语句抛出一个算术异常，包含 throw 语句的 inException()方法头中加入了 throws 子句，它指明要抛弃算术异常。inException()方法的调用发生在 try 块中以捕获抛出的异常，捕获异常的情况与前面见过的用零除情况一样。

需要注意的是，throw 语句一般应放入分支语句中，表示仅在满足一定条件后才被执行，而且 throw 语句后不允许有其他语句，否则将出现编译错误信息：unreachable statement。

4，创建自己的异常

在例 3.31 中，用 throw 抛出的异常类是系统提供的算术异常类。Java 语言还允许定义用户自己的异常类，从而实现用户自己的异常处理机制。使用异常处理使得自己的程序足够健壮，以从错误中恢复。定义自己的异常类要继承 Throwable 类或它的子类，通常是继承 Exception 类。

【例 3.32】 设计自己的异常。从键盘输入一个 double 类型的数，若不小于 0.0，则输出它的平方根，若小于 0.0，则输出提示信息“输入错误！”。

```
import java.io.*;
class MyException extends Exception{
   void test(double x)throws MyException{
      if(x < 0.0)throw new MyException();  // 条件成立时，执行 throw 语句
      else System.out.println(Math.sqrt(x));
   }
   public static void main(String args[])throws IOException{
      MyException n = new MyException();
      try{
         System.out.print("求输入实数的平方根。请输入一个实数：");
         BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
         String s = br.readLine();
         n.test(Double.parseDouble(s));
      }catch(MyException e){
         System.out.println("输入错误！");
      }
   }
}
```

程序的两次运行结果如下：

```
（1）求输入实数的平方根。请输入一个实数：123.5
     11.113055385446435
（2）求输入实数的平方根。请输入一个实数：-789
     输入错误！
```

在本程序中，自己的异常类通过 extends 继承了 Exception 异常类。在 test()方法中，用 throw

语句指定了可能抛出的异常，这个语句在调用 test()方法时，参数小于 0 时被执行，产生异常并抛出。

习 题 三

3.1 编程：输入一个三角形的 3 个边长，检查是否能构成一个直角三角形。

3.2 编程：用 random 方法产生 4 位数的年号，判断是否为闰年的年号。若是闰年，则输出“*xxxx* 年是闰年!”，否则输出“*xxxx* 年是平年!”。

3.3 编程：从键盘输入一个 0 至 99 999 之间的任意数，判断输入的数是几位数。

3.4 铁路运货的运费与路程远近及货物的重量有关，设有如下的单位运费：

不足 100 千米，每吨每千米为 1.00 元；

100 千米以上，不足 300 千米，每吨每千米为 0.90 元；

300 千米以上，不足 500 千米，每吨每千米为 0.80 元；

500 千米以上，不足 1000 千米，每吨每千米为 0.70 元；

1000 千米以上，每吨每千米为 0.60 元。

编程：对输入的货物重量和路程计算相应的运费。

3.5 设银行的活期存款年利率为 1.8%，编程求一笔钱 *x* 存活期 *y* 天后应得的利息。

3.6 用 while 循环语句编程：求 1～1 000 之间的偶数之和。

3.7 编程：输入一个四位整数，按与各位数字相反的顺序输出各位数字。例如，设输入为 1234，输出则为 4321。

3.8 编程：求 1!+2!+3!+…+10!。

3.9 编程：输出一数列的前 *n* 项。该数列的第 1、2、3 项为 1，以后的各项是其前 3 项的和。即：1，1，1，3，5，9，17，31，…。分别用递归和递推方式实现。

3.10 编程：解“百钱买百鸡”问题。即母鸡五元钱一只，公鸡三元钱一只，小鸡一元钱三只，现有 100 元钱想买 100 只鸡，共有多少种买法。

3.11 编程：求 2～*n* 之间的所有素数，素数是只能被 1 和该数本身整除的数，*n* 由键盘输入。

3.12 编程：用循环语句输出如下图（1）和图（2）所示的 *n* 行用符号“*”组成的三角形图形，*n* 从键盘输入。

```
    *                  ********
   ***                  ...
  *****                *****
   ...                  ***
 *******                 *
```

（1） （2）

3.13 编程：用循环语句输出如下图（1）和图（2）所示的 *n* 行图形，*n* 从键盘输入，是一个小于 10 的数。

A	1
BBB	121
CCCCC	12321
DDDDDDD	1234321
...	...
（1）	（2）

3.14　编程：用循环语句输出如下图（1）、图（2）和图（3）所示的 n（奇数）行菱形图形，n 从键盘输入，是一个小于 10 的奇数。（下面图形是 5 行的例）

*	A	A　　B
***	BBB	AAA　BBB
*****	CCCCC	AAAAABBBBB
***	BBB	AAA　BBB
*	A	A　　B
（1）	（2）	（3）

3.15　验证“角谷猜想”：对任意自然数，若是奇数，就对它乘以 3 再加 1；若是偶数，就对它除以 2，这样得到一个新数；对这个新数，再按上述奇数、偶数的计算规则进行计算，一直进行下去，最终将得到 1。

3.16　编程：求 1 到 500 之间的所有“完全数”。完全数是该数因子之和等于该数的数。例如 6 = 1 + 2 + 3，所以 6 是完全数。

3.17　编程：求“水仙花数”。水仙花数是一个三位数，它的各位数字的立方和等于该数。例如 $153 = 1^3 + 5^3 + 3^3$，所以 153 是一水仙花数。

3.18　一个整数的各位数字之和能被 9 整除，则该数也能被 9 整除。用此算法编程，检查整数 n 能否被 9 整除。

3.19　编程：求 1 到 100 之间的“同构数”。同构数是一种出现在它的平方数的右端的数。例如 5 的平方是 25，5 是 25 右端的数，5 就是同构数；25 也是一个同构数，它的平方是 625。

3.20　编程：不大于 n（n 从键盘输入）的自然数中，数字“1”有多少个？对输入小于 0 的数的情况进行异常处理。

3.21　编程：在两位整数中，十位比个位大的数有多少个？

3.22　编程验证下列结论：任何一个自然数 n 的立方都等于 n 个连续奇数的和。例如：$2^3 = 3 + 5$，$3^3 = 7 + 9 + 11$。

3.23　编程：计算 n 的阶乘，用递归方法实现。

3.24　编写对 n 个盘子移动的 Hanoi 塔递归程序。著名的 Hanoi 塔问题是这样的：有 n 个直径不同的圆盘套在一个针（源）上，现要将这些盘子移到另一根针（目标）上，移动时必须保证小盘子在大盘子上方，且一次移动一个盘子。为了使得移动能够进行，可借助一根辅助针。

3.25　编写一个方法，计算 $kkk\cdots k$（共 n 个 k，$n > 0$）的值。例如，$k = 2$，$n = 5$，则为 22222。调用这个方法，求下列 $s1$ 和 $s2$ 的值：

$s1 = 6 + 66 + 666 + 6666 + 66666$

$s2 = 8 + 888 + 88888 + 8888888$

3.26 **阅读分析下列程序，判断程序是否有错，若有错，请指出错误原因。**

```
class BreakDemo {
  public static void main(String args[]) {
    one: for(int i=0; i<10; i++) {
      System.out.print("Pass " + i + ": ");
    }
    for(int j=0; j<100; j++) {
      if(j == 10) break one;
      System.out.print(j + " ");
    }
  }
}
```

3.27 **什么是异常？试列出三个系统定义的运行时异常类。**

3.28 try-catch-finally **语句的执行顺序是怎样的？**

3.29 **对下面的代码段：**

```
try{
  run();
}catch(IOException e){
  System.out.println("Exception1");
  return;
}catch(Exception e1){
  System.out.println("Exception2");
  return;
}finally{
  System.out.println("finally");
}
```

若 run()**方法抛出一个空指针异常** NullPointerException，**显示器上将显示什么？请在下面的几个选项中选择一个答案：**

（1）**无输出**

（2）Exception1
finally

（3）Exception2
finally

（4）Exception1

（5）Exception2

第 4 章　数　　组

数组是一种最简单的复合数据类型。数组是一组同类型有序数据的集合，数组中的一个数据成员称为数组元素，数组元素可以用一个统一的数组名和下标（序号）来惟一确定。根据数组下标是一个还是多个，数组分为一维数组和多维数组。

4.1　一维数组

一维数组中的各个元素排成一行，通过数组名和一个下标就能访问一维数组中的元素。

4.1.1　一维数组的定义

数组的定义包括数组声明和为数组分配空间、初始化（创建数组）等内容，必要时，还要为数组元素分配空间或初始化。

1. 一维数组的声明

声明一个一维数组的一般形式为：

```
type arrayName[];
```

或

```
type[] arrayName;
```

其中，类型 type 可以是 Java 中任意的基本数据类型或引用类型，数组名 arrayName 是一个合法的标识符，[]指明该变量是一个数组变量。

例如：

```
int  intArray[];      (或  int[]  intArray;)       // 声明一个整型数组
double decArray[];(或  double[]  decArray;) // 声明一个双精度实型数组
String strArray[];  (或   String[] strArray;)  // 声明一个字符串数组
Button  btn[];       (或  Button[]  btn;)         // 声明一个按钮数组
```

一个数组声明语句可同时声明多个数组变量。此时，后一种声明格式写起来简单些。例如：

```
int[] a,b,c;
```

相当于：

```
int a[],b[],c[];
```

与其他高级语言不同，Java 在数组声明时并不为数组分配存储空间，因此，在声明的[]中不能指出数组中元素的个数（数组长度），而且对于如上声明的数组是不能访问它的任何元

素的，必须经过初始化、分配存储空间创建数组后，才能访问数组的元素。当仅有数组声明而未分配存储空间时，数组变量中只是一个值为 null 的空引用（指针）。

2，一维数组的空间分配

为数组分配空间有两种方法：数组初始化和使用 new 运算符。为数组分配空间后，数组变量中存储为数组存储空间的引用地址。

（1）数组初始化

数组初始化是指在声明数组的同时指定数组元素的初始值。一维数组初始化的形式如下：

```
type arrayName[] = {element1[,element2..]}
```

其中，element 为类型 type 的初始值。基本类型和字符串类型等可以用这种方式创建数组空间。

例如：

```
int intArray[] = {1,2,3,4,5};
double decArray[] = {1.1,2.2,3.3};
String strArray[] = {"Java","BASIC","FORTRAN"};
```

从上述例子可以看到，一维数组的初始化即在前面数组声明的基础上在大括号中给出数组元素的初值，系统将自动按照所给初值的个数计算出数组的长度并分配相应的存储空间。

（2）使用 new 运算符

通过使用 new 运算符可为数组分配存储空间和指定初值。若数组已经声明，为已声明数组分配空间的一般形式如下：

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

其中，arrayName 是已声明的数组变量，type 是数组元素的类型，arraySize 是数组的长度，可以为整型常量或变量。通过数组运算符 new 为数组 arrayName 分配 arraySize 个 type 类型大小的空间。

若数组未声明，则可在数组声明的同时用 new 运算符为数组分配空间：

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

例如：

```
int a[];
a = new int[10];                    // 给数组 a 分配 10 个整型数据空间
double b[] = new double[5];   // 给数组 b 分配 5 个双精度实型数据空间
String s[] = new String[2];     // 给数组 s 分配 2 个元素的引用空间
```

一旦数组初始化或用 new 分配空间以后，数组的长度即固定下来，不能变化，除非用 new 运算符重新分配空间。

在 Java 语言中，用 new 运算符为数组分配空间是动态的，即可根据需要随时用 new 为已分配空间的数组再重新分配空间。但需注意，对一个数组再次动态分配空间时，若该数组

的存储空间的引用没有另外的存储，则该数组的数据将会丢失。例如：

```
int a[]={1,2,3};
a = new int[5];   // 为 a 数组重新分配空间，原 a 数组的值 1,2,3 将丢失
```

用 new 进行数组的动态空间分配时，若未指定初值，则使用各类数据的默认初值。即对数值类型，其默认初值是相应类型的“0”；对字符型，默认初值为'\u0000'；对布尔型，默认初值为 false；对复合数据类型，默认初值为 null。

经过上述操作，就完成了基本类型的数组和已经初始化的字符串数组的定义，接着就可以访问数组或存取数组元素了。但对复合类型的数组，还需要对数组元素分配空间、初始化。

3. 复合类型数组元素的动态空间分配和初始化

一般情况下，复合类型的数组需要进一步对数组元素用 new 运算符进行空间分配并初始化操作。设已声明一个复合类型的数组：

```
type arrayName[];                    // type 是一个复合数据类型
```

对数组 arrayName 的动态空间分配步骤如下：

（1）为数组分配每个元素的引用空间：

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

（2）为每个数组元素分配空间：

```
arrayName[0] = new type(paramList);
…
arrayName[arraySize-1] = new type(paramList);
```

其中，paramList 参数表用于数组元素初值的指定。

例如，下面是一个图形界面应用程序中所用按钮数组的定义：

```
Button  btn[];                  // 声明一个 Button 按钮类型的数组 btn
btn  =  new  Button[2];         // 给数组 btn 分配 2 个元素的引用空间
btn[0] = new Button("确定");    // 为 btn[0]分配空间并赋显示文本“确定”
btn[1] = new Button("退出");    // 为 btn[1]分配空间并赋显示文本“退出”
```

当然，在比较简单的情况下，上述操作可简化为：

```
Button btn[] = {new Button("确定"),new Button("退出")};
```

4.1.2 一维数组的引用

一维数组的引用分为数组元素的引用和数组的引用，大部分时候都是数组元素的引用。一维数组元素的引用方式为：

```
arrayName[index]
```

其中，index 为数组下标，是 int 类型的量，也可以是 byte、short、char 等类型，但不允

许为 long 类型。下标的取值从 0 开始，直到数组的长度减 1。一维数组元素的引用与同类型的变量相同，每一个数组元素都可以用在同类变量被使用的地方。对前面建立的数组变量 intArray，有 5 个数组元素，通过使用不同的下标来引用不同的数组元素 intArray[0]、intArray[1]、…、intArray[4]。

Java 对数组元素要进行越界检查以保证安全性。若数组元素下标小于 0、大于或等于数组长度将产生 ArrayIndexOutOfBoundsException 异常。

Java 语言对于每个数组都有一个指明数组长度的属性 length，它与数组的类型无关。例如，a.length 指明数组 a 的长度。

对一维数组元素的逐个处理，一般用循环结构的程序。

【例 4.1】　设数组 a 中存有 10 个学生某门课程的成绩，输出这 10 个学生的成绩与平均成绩的差（低于平均成绩用负数表示）。

```
public class Score{
   public static void main(String args[]){
      int a[] = {90,87,67,83,88,94,76,98,95,72},i,sum = 0;
      double ave;
      for(i = 0;i < a.length;i ++) sum += a[i];
      ave = sum/10.0;
      System.out.println("Average = " + ave);
      for(i = 0;i < 10;i ++)
      System.out.print(a[i] + " ");
      System.out.println();
      for(i = 0;i < 10;i ++)System.out.print((a[i]-(int)ave) + " ");
   }
}
```

程序运行结果如下：

```
Average = 85.0
90 87 67 83 88 94 76 98 95 72
5 2 -18 -2 3 9 -9 13 10 -13
```

【例 4.2】　输入 10 个整数，按输入相反的顺序输出它们。

```
import java.io.*;
public class NumberInput {
   public static void main(String args[])throws IOException{
      InputStreamReader ir = new InputStreamReader(System.in);
      BufferedReader br =  new BufferedReader(ir);
      String s;
      int i,n,a[] = new int [10];
      for(i = 0;i < a.length;i ++){
         s = br.readLine();//数据输入按每行一个的方式进行
```

```
            a[i] = Integer.parseInt(s);
        }
        for(i = a.length - 1;i >= 0;i --)
            System.out.println(10 – i + ":" + a[i]);
    }
}
```

【例 4.3】　将 Fibonacci 数列的前 15 个数存入一维数组 a 中，并顺序输出数组的后 5 个数。

```
public class Fib{
    public static void main(String args[]){
        int i,x = 1,y = 1,z,a[] = new int [15];
        for(i = 0;i < 15;i ++){
            a[i] = x;
            z = x + y;
            x = y;
            y = z;
        }
        for(i = 10;i < 15;i ++)
            System.out.println((i + 1) + ":" + a[i]);
    }
}
```

程序运行结果如下：

```
11:89
12:144
13:233
14:377
15:610
```

输出的数据是 Fibonacci 数列的第 11 至 15 个数。

有时候需要将不同基本类型的数据组织到一个数组中，在 Java 这种面向对象的语言中，实现起来是很容易的。在第 2 章中已经介绍过与基本类型相应的包装类，在 Java 语言中，所有的类都是类 Object 的子类，因此可以定义类 Object 的数组，也就可以将不同基本类型的数据存入到一个数组。

【例 4.4】　类 Object 的数组。

```
class ObjectArray {
    public static void main(String args[]) {
        Object a[] = new Object[5];
        a[0] = new Integer(3);
        a[1] = new String("张三");
```

```
        a[2] = new Boolean("true");
        a[3] = new Character('F');
        a[4] = new Double(1345.68);
        System.out.println("编号    姓名      已婚    性别      工资");
        for(int i = 0;i < 5;i ++)System.out.print("  " + a[i] + "  ");
        System.out.println();
    }
}
```

程序运行结果如下：

```
编号    姓名      已婚    性别      工资
  3     张三      true      F      1345.68
```

当数组创建后，数组名中就存储为数组存储区的首地址（即一种引用），可以将此地址赋值给另一同类的数组，即数组的引用。

【例 4.5**】　数组的引用。**

```
public class RefDemo {
    public static void main(String[] args){
        int[] a = { 1,2,3 },b;
        b = a;
        for (int i = 0;i < b.length; ++i )
        System.out.print(b[i] + " ");
        System.out.println();
    }
}
```

由“int a[]={1,2,3},int b[];**”可知数组** a **已创建，数组变量存储为数组存储区的引用，数组** b **未创建，数组变量中为** null **值。若有“**b = a;**”，则使得数组** b **得到数组** a **的存储区引用。此时，对数组** a **元素的访问，可用数组** b **来进行，即** b[0]、b[1]、b[2]**的值分别为** 1、2、3**。若将数组** a **和** b **的赋值反过来：“**a = b;**”，则将得到一个编译错误：**

```
variable b might not have been initialized        //变量 b 还未初始化
```

【例 4.6**】　用递归调用打印数组元素。**

```
public class Recursion1 {
    static int values[] = {1,2,3,4,5};
    public static void main(String args[]) {
        for(int i = 0; i < 5; i ++) values[i] = i;
        printArray(5);
    }
    static void printArray(int i) {
        if(i == 0) return;
```

```
            else printArray(i - 1);
            System.out.println("a[" + (i - 1) + "] " + values[i - 1]);
        }
    }
```

程序运行结果如下：

```
a[0] 0
a[1] 1
a[2] 2
a[3] 3
a[4] 4
```

4.2 多维数组

Java 也支持多维数组。在 Java 语言中，多维数组被看做数组的数组。例如二维数组为一个特殊的一维数组，其中每个元素又是一个一维数组。下面的讨论主要针对二维数组，更高维数的数组情况类似于二维数组。

使用二维数组可方便地处理表格形式的数据。

4.2.1 二维数组的定义

二维数组的定义与一维数组类似，包括数组声明、为数组和数组元素分配空间、初始化等内容。

1，二维数组的声明

声明二维数组的一般形式为：

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

或

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

或

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

其中，type 是数组的类型，可以是简单类型，也可以是复合类型。

例如：

```
char c[][];   // 声明一个二维 char 类型的数组 c
float f[][];  // 声明一个二维 float 类型的数组 f
```

与一维数组时的情况一样，对数组的声明不分配数组的存储空间。

2，二维数组的空间分配

（1）二维数组的初始化

二维数组的初始化也是在声明数组的同时就为数组元素指定初值。例如：

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

Java 系统将根据初始化时给出的初始值的个数自动计算出数组每一维的大小。在这个例子中，二维数组 intArray 由三个一维数组组成，这三个一维数组的元素个数分别为 2、2、3。在 Java 语言中，由于把二维数组看做是数组的数组，数组空间不一定连续分配，所以不要求二维数组每一维的大小相同。

【例 4.7】　定义一个三角形二维数组。

```
class ArrayLengths {
  public static void main(String args[]) {
    int iarray[][] = {
      { 44 },
      { -22, 16 },
      { 11, -12, 99 }
    };
    System.out.println("iarray.length = " + iarray.length);
    System.out.println("iarray[0].length = " + iarray[0].length);
    System.out.println("iarray[1].length = " + iarray[1].length);
    System.out.println("iarray[2].length = " + iarray[2].length);
  }
}
```

程序运行结果如下：

```
iarray.length = 3
iarray[0].length = 1
iarray[1].length = 2
iarray[2].length = 3
```

注意：在这个程序中，length 用于二维数组的结果。

（2）使用 new 运算符

对二维数组，用 new 运算符分配空间有两种方法。

一种方法是直接为二维数组的每一维分配空间。若数组已经声明，为已声明数组分配空间的一般形式如下：

```
arrayName = new type[arraySize1][arraySize2];
```

其中，arrayName 是已声明的数组名，type 是数组元素的类型，arraySize1 和 arraySize2 分别是数组第一维和第二维的长度，可以为整型常量或变量。通过数组运算符 new 为数组 arrayName 分配 arraySize1×arraySize2 个 type 类型大小的空间。

若数组未声明，则可在数组声明的同时用 new 运算符为数组分配空间：

```
type arrayName[][] = new type[arraySize1] [arraySize2];
```

例如：

```
int a[][];
a = new int[3][4];                    // 给数组 a 分配 12 个整型数据空间
double b[][] = new double[2][5];      // 给数组 b 分配 10 个双精度实型数据空间
String s[][] = new String[2][2];      // 给数组 s 分配 4 个 String 元素的引用空间
```

另一种方法是从最高维开始，分别为每一维分配空间，格式为：

```
arrayName = new type[arrayLength1][];
arrayName[0] = new type[arrayLength20];
arrayName[1] = new type[arrayLength21];
…
arrayName[arrayLength1-1] = new type[arrayLength2n];
```

例如：

```
int a[][]= new int [2][];   // 定义二维数组 a 由两个一维数组构成
a[0] = new int [3];         // 二维数组 a 的第 1 个一维数组有 3 个元素
a[1] = new int [5];         // 二维数组 a 的第 2 个一维数组有 5 个元素
```

在 Java 语言中，必须首先为最高维分配引用空间，然后再顺次为低维分配空间。

注意：在用 new 进行二维数组动态空间分配时可以先只确定第一维的大小，其余维的大小可以在以后分配。在 Java 语言中，对二维数组不允许有如下的形式：

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

3. 复合类型数组元素的动态空间分配和初始化

与一维数组相同，对于复合类型的数组，要为每个数组元素单独分配空间。

例如：

```
String s[][] = new String [2][];
s[0] = new String[2];
s[1] = new String[2];
s[0][0] = new String("Java");
s[0][1] = new String("Program");
s[1][0] = new String("Applet");
s[1][1] = new String("Application");
```

在上述二维数组的定义中，数组的初始化方式和 new 方式是分别使用的，实际上，还经常见到这两种方式的混合使用方式。例如：

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

即二维数组 a 由三个一维数组组成，初始化时 new 的个数即一维数组个数，而每个一维数组的数据个数与存储空间用 new 运算符动态分配。

4.2.2 二维数组的引用

大多数情况是引用二维数组的元素。对二维数组中的每个元素，引用方式为：

```
arrayName[index1][index2]
```

其中，index1 和 index2 为下标，可用类型同一维数组，如 c[2][3]等。同样，每一维的下标都从 0 开始。

对二维数组元素的逐个处理，一般用嵌套循环结构的程序。

【例 4.8】 矩阵转置。矩阵是排列成若干行若干列的数据表，转置是将数据表的行列互换。即第一行变成第一列、第二行变成第二列等等。程序如下：

```
public class Matrixt{
  public static void main(String args[]){
    int a[][] = {{1,2,3,4},{2,3,4,5},{3,4,5,6}};
    int b[][] = new int [4][3];
    int i,j;
    for(i = 0;i < 3;i ++)
      for(j = 0;j < 4;j ++)b[j][i] = a[i][j];
    for(i = 0;i < 4;i ++){
      for(j = 0;j < 3;j ++) System.out.print(b[i][j] + " ");
      System.out.println();
    }
  }
}
```

程序运行结果如下：

```
1  2  3
2  3  4
3  4  5
4  5  6
```

输出为 4 行 3 列的矩阵，即已经进行了行和列的互换。

4.3 数组作为方法参数和返回值

在 Java 语言中，数组可作为方法参数和方法的返回值。因为数组是复合类型，数组变量存储的是数组存储区的引用，所以，传送数组或返回数组实际上在传送引用。在这个意义上来说，即使实际参数和形式参数数组变量名不同，但因为它们是相同的引用，若在被调方法中改变了形参数组，则该形参对应的实参数组也将发生变化。

【例 4.9】　数组作为方法参数。

```
class ArrayArgument {
   public static void main(String args[]) {
      int x[] = { 11, 12, 13, 14, 15 };
      display(x);
      change(x);
      display(x);
   }
   public static void change(int x[]) {
      int y[] = { 21, 22, 23, 24, 25 };
      x = y;
   }
   public static void display(int x[]) {
      for(int i = 0; i < x.length; i++)
         System.out.print(x[i] + " ");
      System.out.println("");
   }
}
```

程序运行结果如下：

```
11 12 13 14 15
11 12 13 14 15
```

【例 4.10】　编写一个方法，求一组数的最大值、最小值和平均值。

该方法要将求出的最大值、最小值和平均值多个值作为结果返回，可将它们存储到一个数组中，再用 return 返回。

```
class ReturnArray{
   public static void main(String args[]) {
      double a[] = {1.1, 3.4, -9.8, 10};
      double b[] = max_min_ave(a);
      for(int i = 0;i < b.length;i ++)
      System.out.println("b[" + i +"]= " + b[i]);
   }
   static double [] max_min_ave(double a[]){
      double res[] = new double[3];
      double max = a[0],min = a[0],sum = a[0];
      for(int i = 1;i < a.length;i ++){
         if(max < a[i])max = a[i];
         if(min > a[i])min = a[i];
         sum += a[i];
```

```
        }
        res[0] = max;
        res[1] = min;
        res[2] = sum/a.length;
        return res;
    }
}
```

程序运行结果如下：

```
b[0]= 10.0
b[1]= -9.8
b[2]= 1.1749999999999998
```

4.4 数组操作的常用方法

在 Java 语言中提供了一些对数组进行操作的类和方法，掌握它们的用法，可方便数组程序的设计。

1. 类 System 的静态方法 arraycopy()

系统类 System 的静态方法 arraycopy()可用来进行数组复制，其格式和功能如下：

```
public static void arraycopy(Object src,int src_position,Object dst,int dst_position,int length)
```

从源数组 src 的 src_position 处，复制到目标数组 dst 的 dst_position 处，复制长度为 length。

【例 4.11】 用方法 arraycopy()复制数组。

```
class ArrayCopy {
    public static void main(String args[]) {
        int array1[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        int array2[] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
        System.arraycopy(array1, 0, array2, 0, 5);
        System.out.print("array2: ");
        for(int i=0;i<array2.length;i++)System.out.print(array2[i] + " ");
    }
}
```

程序运行结果如下：

```
array2: 0 1 2 3 4 0 0 0 0 0
```

2. 类 Arrays 中的方法

java.util.Arrays **类中提供了对数组的排序** sort()、**二分查找** binarySearch()**等静态方法。**

（1）void sort(Object[] a)

sort()方法有重载，以适应对不同类型数组 a 的递增排序。

【例 4.12】 使用 sort()方法对一整型数组递增排序。

```
import java.util.*;
public class ArraySort{
   public static void main(String args[]){
      int a[]={8,6,7,3,5,4},i;
      Arrays.sort(a);
      for(i = 0;i < a.length;i ++)System.out.println(" " + a[i]);
   }
}
```

程序运行结果如下：

```
3 4 5 6 7 8
```

sort()方法的另一类重载为数组的部分元素的递增排序，其格式为：

```
void sort(Object [] a,int fromIndex,int toIndex)
```

排序范围为 fromIndex 至 toIndex - 1。例如：

```
int a[]={8,6,7,3,5,4};
Arrays.sort(a,2,5);    // a 数组元素的顺序为 8 6 3 5 7 4
```

（2）int binarySearch(Object[] a,Object key)

binarySearch()方法有重载，以适应对不同类型已排序数组 a 的二分 key 查找。若找到，则返回找到元素的位置。

【例 4.13】 binarySearch()方法的使用。

```
import java.util.*;
public class BinarySearch{
   public static void main(String args[]){
      int  a[]={3,4,5,6,7,8},i;                    //  数组要求已排序
      i=Arrays.binarySearch(a,6);                  //  在数组中查找数据 6
      System.out.println(i);
   }
}
```

程序的运行结果为：

```
3
```

另外，java.lang.reflect.Array 类提供了一些静态方法来动态创建和存取 Java 数组。具体内容可参考 Java SDK 的说明文档。

4.5 数组应用举例

【例 4.14】 数据排序。将一组数按从小到大的顺序输出（sort）。数据排序有很多种方法，如冒泡排序、选择排序、快速排序等等。在 Java 语言中提供了数据排序的方法 sort，为了熟悉数组的使用，这里不用 Java 的 sort 方法，而自己编写排序程序。冒泡排序方法的思想是这样的：

n 个数据排序要进行 *n*-1 轮相邻元素的比较。若是从小到大排序（递增），则前面的元素大于后面的元素时两者进行交换；若是从大到小排序（递减），则前面的元素小于后面的元素时两者进行交换。比较 *n*-1 次后比较完第一轮，可以保证最大数或最小数已经排到最后，接着对剩下的 *n*-1 个数进行第二轮的比较，需要交换时进行交换，比较 *n*-2 次后比较完第二轮。此时，本次比较数据的最大或最小数已经排到本次比较范围的最后，如此继续下去，经过 *n*-1 轮比较，就完成了 *n* 个数据的排序。冒泡排序的程序如下：

```
public class Sort{
   public static void main(String[] args){
      int[] a = { 32,87,3,589,12,1076,2000,8,622,127 };
      for (int i = 1;i<a.length;++i ){ // 进行 n-1 轮比较
         for (int j = 1; j <= a.length-i; j++){
            if (a[j-1] > a[j]){ // 比较
               int t = a[j-1];  // 交换
               a[j-1] = a[j];
               a[j] = t;
            }
         }
      }
      for (int i = 0; i < a.length; i++)  // 输出
         System.out.print(a[i] + " ");
      System.out.println();
   }
}
```

程序执行结果如下：

```
3 8 12 32 87 127 589 622 1076 2000
```

【例 4.15】 两个矩阵相乘。设矩阵 $C_{mn}=A_{mp}\times B_{pn}$，$A$ 矩阵的列数要与 B 矩阵行数相同才能进行相乘，矩阵 C 的元素 c_{ij} 等于 A 矩阵的第 i 行各元素与 B 矩阵第 j 列的各对应元素乘积之和，即 $c_{ij}=a_{ik}\times b_{kj}$ ($k=1,\cdots,p$，$i=1,\cdots,m$，$j=1,\cdots,n$)。

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

```
        int a[][]=new int[2][3];
        int b[][] = { {1,5,2,8},{5,9,10,-3},{2,7,-5,-18} };
        int c[][] = new int[2][4];
        for( i=0; i<2; i++)
           for( j=0; j<3; j++)a[i][j]=(i+1)*(j+2);
        for( i=0; i<2; i++){
           for( j=0; j<4; j++){
              c[i][j]=0;
              for( k=0; k<3; k++)
              c[i][j]+=a[i][k]*b[k][j];
           }
        }
        System.out.println("*** Matrix A ***");
        for( i=0; i<2; i++){
           for( j=0; j<3; j++)System.out.print(a[i][j]+" ");
           System.out.println();
        }
        System.out.println("*** Matrix B ***");
        for( i=0; i<3; i++){
           for( j=0; j<4; j++)System.out.print(b[i][j]+" ");
           System.out.println();
        }
        System.out.println("*** Matrix C ***");
        for( i=0; i<2; i++){
           for( j=0; j<4; j++)System.out.print(c[i][j]+" ");
           System.out.println();
        }
     }
  }
```

程序执行结果如下：

```
*** Matrix A ***
2 3 4
4 6 8
*** Matrix B ***
1 5 2 8
5 9 10 -3
2 7 -5 -18
*** Matrix C ***
25 65 14 -65
```

50 130 28 -130

习 题 四

4.1 从键盘输入 20 个整数，将奇数和偶数分别存入不同的数组中，并按先奇数后偶数的顺序输出这两个数组中的数据。

4.2 设在数组 a 中存有 10 个整数，现从键盘输入一个数，检查该数是否在数组中：若在数组中，则输出该数在数组中的位置；若在数组中未找到该数，则输出“Not Found!”信息。

4.3 有一个数组，内放 10 个整数，要求找出最小的数和它的下标，然后将它和数组中最前面的元素对换。

4.4 有 17 个人围成一圈（编号 0～16），从第 0 号的人开始从 1 报数，凡报到 3 的倍数的人离开圈子，然后再数下去，直到最后只剩下一个人为止。问此人原来的位置是多少号？

4.5 输入一个 n 阶方阵各元素的值，求出两条对角线元素值之和（交叉位置的元素只计算一次）。

4.6 以下面的格式打印杨辉三角形的前 9 行：

```
    1
   1  1
  1  2  1
 1  3  3  1
.....
```

4.7 编写一个程序，输出奇数阶幻方（设阶数为 n）。幻方是一个行列数相等的方阵，它由 1 至 $n \times n$ 的数据组成，要求其每行、每列和对角线上的数据之和相等。一种算法如下：

（1）第一个数 1 放在方阵的最后一行的中央；

（2）下一个数放在本数的右下方。有几个特殊的情况：

若当前这个数是 n 的倍数，则下一个数放在本数的上方；若右下方出了方阵的行，则下一个数放在同列的第一行；若右下方出了方阵的列，则下一个数放在同行的第一列。

例如，用本算法排出的 3 阶幻方为：

```
4  9  2
3  5  7
8  1  6
```

4.8 下列程序的输出结果是什么？

```
class Ex4_8{
   public static void main(String args[]){
      int[] a={1,2,3},b=a;
      a[0]=a[1]+1;a[1]=a[2]+1;a[2]=a[0]+1;
      for(int i=0;i<3;i++)System.out.print(a[i]+" ");
      System.out.println();
```

```
        for(int i=0;i<3;i++)System.out.print(b[i]+" ");
        System.out.println();
    }
}
```

4.9 **分析下列程序，判断程序是否有错。若有错，请指出错误原因。**

```
class Ex4_9{
    public static void main(String args[]){
        int a[][]={{1,2},{3,4,5}},i,j;
        for(i=0;i<2;i++){
            for(j=0;j<3;j++)System.out.print(a[i][j]+" ");
            System.out.println();
        }
    }
}
```

4.10 **编译并运行如下程序后，结果是什么？请在给出的选项中选择出正确的答案。**

```
class Array{
    public static void main(String[] args){
        int length = 100;
        int[] d = new int[length];
        for (int index = 0; index < length; index++)
            System.out.println(d[index]);
    }
}
```

（1）**因为** int[]**数组声明不正确，程序不能被编译。**

（2）**程序被编译，但在运行时将抛出一个** ArrayIndexOutOfBoundsException **异常。**

（3）**程序将显示数字** 0 **到** 99，**然后将抛出一个** ArrayIndexOutOfBoundsException **异常。**

（4）**程序被编译，但** println()**方法将抛出一个** NoSuchMethodException **异常。**

（5）**程序正常运行，并显示** 100 **个** 0**。**

第 5 章　面向对象程序设计

前面几章介绍了 Java 语言中的基本数据类型、一些系统定义的类和特定的类的使用，很多时候用户程序还需要针对特定问题的特定逻辑来定义自己的类。本章将完整地介绍 Java 语言中重要的自定义的复合数据类型——类，以及与之相关的对象、接口和包等概念及其使用。

5.1　类（class）

Java 语言作为一种面向对象的程序设计语言，具备面向对象技术的基本属性。对象是面向对象技术的一个基本概念。类是组成 Java 程序的基本元素，它封装了一系列的变量（即数据成员，也称为“域 field”）和方法（即成员方法 method），是一类对象的原型。创建一个新的类，就是创建一个新的数据类型。实例化一个类，就得到一个对象。因此，对象就是一组变量和相关方法的集合，其中变量表明对象的状态、属性，方法表明对象所具有的行为。

对一个用户自定义的类，要为类取一个名字，并指明类中包含哪些变量和方法以及相应的类型、实现等，这称为类的定义。然后用已定义的类来创建该类的对象，对对象进行各种操作、运算，使得程序能完成所要完成的功能。

5.1.1　类定义

类定义语句的一般形式为：

```
[modifier]class NameOfClass [extends Super] [implements interfaces] {
   member variable declarations      // 成员变量声明
   method  declarations              // 方法成员声明
}
```

其中，class、extends 和 implements 都是 Java 的关键字。modifier、extends 和 implements 是有关修饰符、继承和接口的内容，以后逐渐介绍。类定义中必须要写的内容是类名 NameOfClass，应该使用 Java 语言合法的标识符来对自定义的类命名。大括号{}中是定义类体的地方，指明该类中包含的数据成员和成员方法。若类体中只包含数据成员，则类的定义类似传统语言中的结构体或记录。在 Java 语言中也允许定义没有任何成员的空类。

【例 5.1】　定义一个名为 Rect 表示长方形的类，它仅包含 double 类型的长 length 和宽 width 两个数据成员。

```
class Rect {
   double length;
   double width;
}
```

从这个例子可以看到，在类中进行成员变量的声明与一般的变量的声明形式完全相同。成员变量的类型可以任意，成员变量的名称在一个类中保证惟一性即可。

在第 3 章中，已经介绍了在 Java 语言中如何定义和使用方法。在类的定义中，可加入对数据成员进行操作的方法成员。

【例 5.2】　定义一个名为 Rectangle 表示长方形的类，它包含 double 类型的长 length 和宽 width 两个数据成员以及设置长方形长宽值的方法 setDim()、计算长方形面积的方法 area()。

```
class Rectangle {
   double length;
   double width;
   double area() {
      return length * width; // 长 * 宽的值是 double 型，所以方法返回值为 double 型
   }
   void setDim(double w, double l) {   // 设置长方形的大小
      width = w;
      length = l;
   }
}
```

在一个程序中，有时需要定义多个类。多个类的定义形式有两种：并列和嵌套。

常见的多个类定义形式为并列定义，即一个类接着一个类进行定义，它们之间是并列的关系；另一种形式为嵌套定义，即在一个类中定义另外的类，它们之间是包含和被包含的关系，可分别称为包含类和内部类（或嵌套类）。采用何种形式定义多个类，由类之间的访问关系确定。

类定义了一个类型（type）。与 Java 语言提供的几种基本类型一样，类型用来声明、定义该类型的变量。例如下面的语句：

```
Rect rect1;
```

声明变量 rect1 的类型为类 Rect。类型为类的变量与基本类型变量有所不同，类是一种引用（Reference）类型。实际上，rect1 是一个对类型为类 Rect 的对象的引用，rect1 不是对象本身，可理解为一个指针，上述声明仅生成一个空（null）引用。

5.1.2　类对象

一旦定义了所需的类，就可以创建该类的变量，创建类的变量称为类的实例化，类的变量也称为类对象、类的实例等。

类的对象是在程序运行中创建生成的，其所占的空间在程序运行中动态分配。当一个类的对象完成了它的使命，为节省资源，Java 的垃圾收集程序就会自动收回这个对象所占的空间，即类对象有自己的生命周期。

1，创建对象

创建类的对象需用 new 运算符，一般形式为：

```
objectName = new className()
```

new 运算符用指定的类在内存中分配空间，并将存储空间的引用存入语句中的对象变量 objectName。例如：

```
rect1 = new Rect();
```

new 运算符也可以与类声明一起使用，来创建类的对象。例如：

```
Rect rect1 = new Rect();
```

2，引用对象

在创建了类的对象后，就可以对对象的各个成员进行访问，进行各种处理。访问对象成员的一般形式为：

```
objectName.fieldName
objectName.methodName()      // 方法名带圆括号
```

运算符“.”在这里称为成员运算符，在对象名 objectName 和成员名 fieldName、methodName()之间起到连接的作用，指明是哪个对象的哪个成员。

例如，设已经定义了例 5.2 中的 Rectangle 类，可以用如下的方法来引用对象的成员：

```
Rectangle r = new Rectangle();     // 定义类的对象
r.length,r.width     // 引用对象的数据成员
r.area()             // 引用对象的成员方法
```

【例 5.3】 利用例 5.1 定义的类 Rect，计算长和宽分别为 10 和 20 的长方形面积。

```
class RectDemo {
  public static void main(String args[]) {
    Rect rect1 = new Rect();    // 为对象 rect1 分配存储空间
    double area;
    rect1.width = 10;           // 向数据成员赋值
    rect1.length = 20;
    area = rect1.width * rect1.length;    // 对数据成员进行运算
    System.out.println("长方形面积是：" + area);
  }
}
```

本程序由两个类构成，设已经编译类 Rect 文件 Rect.class 和 RectDemo.class，程序的运行结果如下：

```
长方形面积是：200.0
```

若程序需要更多的类对象，可用 new 运算符多次为不同的对象分配空间，但各个对象有各自的存储空间，它们是互不相干的。

【例 5.4】 利用例 5.1 中定义的类 Rect，分别计算长、宽分别为 10、20 和 3、6 的两个长方形面积。

```
class RectDemo1{
  public static void main(String args[]){
    Rect rect1 = new Rect();
    Rect rect2 = new Rect();
    double area;
    rect1.width = 10;
    rect1.length = 20;
    area = rect1.width * rect1.length;
    System.out.println("长=20，宽=10 的长方形面积是：" + area);
    rect2.width = 3;
    rect2.length = 6;
    area = rect2.width * rect2.length;
    System.out.println("长= 6，宽= 3 的长方形面积是：" + area);
  }
}
```

程序运行结果如下：

```
长=20，宽=10 的长方形面积是：200.0
长= 6，宽= 3 的长方形面积是：18.0
```

在这个例子中，为了处理两个长方形，计算长方形面积和输出面积值在程序中使用了两次，显得程序不紧凑。在类的定义中加入可对成员数据进行操作的成员方法，可使程序更有效率。

【例 5.5】 用例 5.2 中定义的类 Rectangle 分别计算长、宽分别为 10、20 和 3、6 的两个长方形面积。

```
class RectDemo2 {
  public static void main(String args[]) {
    Rectangle rect1 = new Rectangle();
    Rectangle rect2 = new Rectangle();
    double ar;
    rect1.setDim(10, 20);      // 初始化每个长方形
    rect2.setDim(3, 6);
    ar = rect1.area();         // 调用 area 方法得到第一个长方形的面积
    System.out.println("第一个长方形的面积是： " + ar);
    ar = rect2.area();         // 调用 area 方法得到第二个长方形的面积
    System.out.println("第二个长方形的面积是： " + ar);
```

```
    }
}
```

程序运行结果如下：

```
第一个长方形的面积是： 200.0
第二个长方形的面积是： 18.0
```

在这个程序例子中，类体中定义 setDim()**方法时用了** double **类型的** w **和** l **两个虚拟参数，程序中对两个类对象调用了两次** setDim()**方法：第一次调用** setDim()**方法时，用了实际参数** 10 **和** 20**；第二次调用时给了实际参数** 3 **和** 6**，即分别对两个对象的数据成员赋不同的长宽值。**

像这样将数据和对数据进行计算、操作的方法封装到类中，正是面向对象程序设计方法的特征。

5.1.3 构造方法

在 Java **中，任何变量在被使用前都必须先设置初值。**Java **提供了为类的成员变量赋初值的专门功能：构造方法（**constructor**）。构造方法是一种特殊的成员方法，它的特殊性反映在如下几个方面：**

（1**）构造方法名与类名相同。**

（2**）构造方法不返回任何值，也没有返回类型。**

（3**）每一个类可以有零个或多个构造方法。**

（4**）构造方法在创建对象时自动执行，一般不能显式地直接调用。**

【例 5.6**】 分别计算长、宽分别为** 10、20 **和** 3、6 **的两个长方形面积。本程序用构造方法来初始化长方形的大小。**

```
class RectConstructor{
   double length;
   double width;
   double area() {
      return length * width;
   }
   RectConstructor(double l, double w) {   // 构造方法，设置长方形的大小
      length = l;
      width = w;
   }
}
class RectDemo3 {
   public static void main(String args[]) {
      RectangleRC rect1 = new RectangleRC(10,20);      // 初始化每个长方形
      RectangleRC rect2 = new RectangleRC(3,6);
      double ar;
      ar = rect1.area();      // 调用 area 方法得到第一个长方形的面积
```

```
        System.out.println("第一个长方形的面积是： " + ar);
        ar = rect2.area();      // 调用 area 方法得到第二个长方形的面积
        System.out.println("第二个长方形的面积是： " + ar);
    }
}
```

程序运行结果如下：

```
第一个长方形的面积是： 200.0
第二个长方形的面积是： 18.0
```

需要注意的是，当方法虚拟参数名与成员变量名相同时，使用时会产生混淆，在 Java 语言中，可用 this 关键字表示本对象。例如：

```
Rectangle(double length, double width) {     // 使用 this 避免命名空间冲突
    this.length = length;                    // 很明确：参数向成员变量赋值
    this.width = width;
}
```

在有多个构造方法时，一个构造方法可以调用另一个构造方法，调用的方法是：

```
this(实际参数表)
```

这个语句调用与 this 中参数匹配的构造方法。

没有参数的构造方法叫做无参数构造方法。一个类若没有任何用户定义的构造方法，Java 会自动提供一个空无参数构造方法，在创建对象时，使用这个无参的构造方法为类对象的成员变量赋数据类型的默认值。一旦用户定义了自己的构造方法，无参构造方法就不能再被使用。

在 Java 语言中也允许构造方法重载，即定义多个构造方法。

【例 5.7】　构造方法的重载。

```
class RectOverload{
    double length;
    double width;
    double area(){
        return length * width;
    }
    RectOverload(double l, double w) {
        length = l;
        width = w;
    }
    RectOverload(double s) {
        length = s;
        width = s;
    }
```

```
}
class RectDemo4 {
  public static void main(String args[]) {
    RectOverload rect1 = new RectOverload(10,20);  // 初始化一个长方形
    RectOverload rect2 = new RectOverload(6);      // 初始化一个正方形
    double ar;
    ar = rect1.area();     // 调用 area 方法得到一个长方形的面积
    System.out.println("长方形的面积是： " + ar);
    ar = rect2.area();     // 调用 area 方法得到一个正方形的面积
    System.out.println("正方形的面积是： " + ar);
  }
}
```

程序运行结果如下：

```
长方形的面积是： 200.0
正方形的面积是： 36.0
```

使用构造方法并非只是为了给对象置初值方便，更重要的是，它是确保一个对象有正确起始状态的必要手段。另外，通过使用非 public 的构造方法，可以防止程序被其他人错误地使用与扩展。

5.1.4 类和成员的修饰符

在类和类的成员定义时可以使用一些修饰符（modifier），来对类和成员的使用做某些限定。一般将修饰符分为两类：访问控制符和非访问控制符。访问控制符有 public、protected、private 等，它们的作用是给予对象一定的访问权限，实现类和类中成员的信息隐藏。非访问控制符作用各不相同，有 static、final、native、volatile、abstract 等。某些修饰符只能应用于类的成员，某些修饰符既可应用于类，也可应用于类的成员。对访问控制符的讨论见 5.4 节。这里先介绍一些非访问控制符。

1，static 修饰符

在第 3 章中，已经使用 static 修饰符来修饰类的成员变量和方法成员，使它们成为静态成员，也称为类成员。静态成员存储于类的存储区，属于整个类，而不属于一个具体的类对象。因为静态成员属于整个类，所以它被所有该类对象共享。在不同的类对象中访问静态成员，访问的是同一个。

对静态成员的使用要注意以下两点：

（1）静态方法不能访问属于某个对象的成员变量，而只能处理属于整个类的成员变量，即静态方法只能处理静态变量。

（2）可以用两种方式调用静态成员，它们的作用相同。

变量：类名.变量、类对象.变量。

方法：类名.方法名()、类对象.方法名()。

【例 5.8】　静态成员的使用。

```
class StaticDemo {
   static int a = 42;      // 静态变量
   static int b = 99;
   static void callme() {   // 静态方法
      System.out.println("a = " + a);
   }
}
class StaticByName {
   public static void main(String args[]) {
      StaticDemo.callme();      // 不需创建对象，通过类名直接调用静态方法
      System.out.println("b = " + StaticDemo.b);  //通过类名直接调用静态变量
   }
}
```

程序运行结果如下：

```
a = 42
b = 99
```

2. final 修饰符

final 修饰符可应用于类、方法和变量。final 的意义为终极或最终。final 类不能被继承，即 final 类无子类。final 方法不能被覆盖，即子类的方法名不能与父类的 final 方法名相同。final 变量实际上是 Java 语言的符号常量，可在定义时赋初值或在定义后的其他地方赋初值，但不能再次赋值，习惯上使用大写的标识符表示 final 变量。例如：

```
final double PI = 3.1416;
final double G = 9.18;
```

因为 final 变量不能改变，没有必要在每个对象中进行存储，可以将 final 变量声明为静态的，以节省存储空间。例如：

```
static final double PI = 3.1416;
```

3. abstract 修饰符

abstract 修饰符可应用于类和方法，称为抽象类和抽象方法。抽象类需要继承、抽象方法需要在子类中实现才有意义。进一步的讨论见 5.2.2 节抽象类和抽象方法。

4. native 修饰符

native 修饰符一般用来声明用其他语言如 C、C++、FORTRAN、汇编等书写方法体并具体实现方法功能的特殊方法。由于 native 方法的方法体使用其他语言编写，所以所有的 native 方法都没有方法体。native 方法可应用于实时性强、执行效率高，运行速度要求较高的场合。

5. synchronized 修饰符

synchronized 修饰符可应用于方法或块，在多线程程序中，对用于共享的方法和块加互斥锁，使得任一时刻，synchronized 方法或块只能由一个线程执行或操作。详细讨论见本书第 10 章。

5.1.5 类的使用

下面从几个方面来讨论类的使用情况。

1，私有成员的访问

为了降低类间的偶合性，可以为类成员指定 private 修饰符，表示该成员只能在该类内部访问。若需要在其他类中访问私有成员，只能通过取数和送数的方法来访问。这样的方法常命名为 getXxx()和 setXxx()等。

【例 5.9】 私有成员的访问。

```
class RectangleRC {
   private double length;
   private double width;
   double getLength() {    // 定义取长方形边长的方法
      return length;
   }
   double getWidth() {
      return width;
   }
   RectangleRC(double l, double w) {
      length = l;
      width = w;
   }
}
class RectDemo5 {
   public static void main(String args[]) {
      RectangleRC rect1 = new RectangleRC(8,7);
      double ar = rect1.getLength()*rect1.getWidth();// 获得长、宽
      System.out.println("长方形的面积是： " + ar);
   }
}
```

程序运行结果如下：

```
长方形的面积是： 56.0
```

2，方法参数是类的对象

在 Java 语言中，方法的参数类型除了可以是基本类型外，还可以是引用类型——类。因为在类的对象中实际存储为对象的引用，因此在调用类参数时方法间传送的是引用。尽管 Java 采用值传送，引用从调用方法单向传送到被调方法，但由于调用方法与被调方法对应类参数的引用相同，它们引用同一对象，所以，若在被调方法中修改了引用类型形式参数的取值，则调用方法对应的实际参数也将发生相应的变化。即调用方法与被调方法之间是“引用单向传送，数据双向传送”。

应用引用类型的方法参数，可在方法间传送数据。

【例 5.10】　引用类型的方法参数是方法间传送数据的桥梁。

```
class RefParameter{
   double width,length,area;
   RefParameter(double w,double l){
      width = w;
      length = l;
   }
   void calArea(RefParameter r){
      r.area = r.width * r.length;
   }
}
class PassObject{
   public static void main(String args[]){
      RefParameter rr = new RefParameter(10,20);
      rr.calArea(rr);
      System.out.println("长方形面积为 ：" + rr.area);
   }
}
```

程序运行结果如下：

```
长方形面积为 ：200.0
```

本例中，calArea()方法的参数类型声明为引用类型。在 main()方法中调用该方法时，实际参数 rr 与形式参数 r 对应结合，rr 的引用单向传送给 r，但发生数据双向传送。类对象 rr 中的 width 和 length 传送给 r，在方法 calArea()中求出面积 area，又传回给 rr。输出 rr 的 area 即为所求。

3. 方法返回值为类的对象

在 Java 语言中，方法的返回值类型也可以为引用类型，例如类。

【例 5.11】　方法的返回值类型为引用类型。

```
class RetClass{
```

```
    double width,length,area;
    RetClass(double w,double l){
      width = w;
      length = l;
    }
    RetClass calArea(RetClass r){    // 声明方法的返回值类型为引用类型
      r.area = r.width * r.length;
      return r;                      // 返回值为引用类型的对象
    }
  }
  class ReturnObject{
    public static void main(String args[]){
      RetClass rr = new RetClass(10,20);
      rr = rr.calArea(rr);
      System.out.println("长方形面积为 ：" + rr.area);
    }
  }
```

程序运行结果如下：

```
长方形面积为 ：200.0
```

当方法返回值类型声明为引用类型时，方法中 return **语句的表达式类型也应为该引用类型，** return **的对象应是该类的对象。**

4，类对象作为类的成员

类的数据成员也可以是引用类型的数据，如数组、字符串和类等。若一个类的对象是一个类的成员时，要用 new **运算符为这个对象分配存储空间。在包含类数据成员的类及类的实例中可以访问类数据成员的成员。**

【例 5.12**】 类对象作为类的成员。**

```
  class RectC{
    double width,length;
    RectC(double w,double l){
      width = w;
      length = l;
    }
  }
  class RectangleC{                  // 具有两个成员的类
    RectC r = new RectC(10,20);      // 类成员要分配空间
    double area;                     // 基本类型成员
  }
```

```
class ClassMember{
  public static void main(String args[]){
    RectangleC rr = new RectangleC();
    rr.area = rr.r.width * rr.r.length;
    System.out.println("长方形面积为 ：" + rr.area);
  }
}
```

程序运行结果如下：

```
长方形面积为 ：200.0
```

注意 rr.r.width **和** rr.r.length **的意义：** rr **有成员** r，r **有成员** width、length。

5，**内部类**（Inner class）

内部类，也称为嵌套类，被附加到 JDK1.1 **及更高版本中。内部类允许一个类定义被放到另一个类定义里、一个语句块里或一个表达式内部。内部类是一个有用的特征，因为它们允许将逻辑上同属性的类组合到一起，并在另一个类中控制一个类的可视性。**

【例 5.13**】　在一个类中定义类（内部类）。**

```
class RectDemo6 {
  public static void main(String args[]) {
    double ar;
    class RectangleR {
      double length;
      double width;
      double area(){
        return length * width; // 返回面积
      }
      void setDim(double w, double l){  // 设置长方形的大小
        width = w;
        length = l;
      }
    }
    RectangleR rect1 = new RectangleR();
    RectangleR rect2 = new RectangleR();
    rect1.setDim(10, 20);     // 初始化每个长方形
    rect2.setDim(3, 6);
    ar = rect1.area();          // 调用 area 方法得到第一个长方形的面积
    System.out.println("第一个长方形面积是： " + ar);
    ar = rect2.area();          // 调用 area 方法得到第二个长方形的面积
    System.out.println("第二个长方形面积是： " + ar);
```

```
    }
}
```

程序运行结果如下：

```
第一个长方形面积是： 200.0
第二个长方形面积是： 18.0
```

因为内部类定义在一个类中，因此内部类的名称不能与所嵌套的类相同，而且只能在定义的范围中使用。内部类具有下面一些特性：

（1）内部类可以被定义在方法中。它可以访问嵌套类的方法的 final 变量。

（2）内部类可以使用所嵌套类的类变量和实例变量以及所嵌套的块中的本地变量。

（3）内部类可以被定义为 abstract 抽象类。

（4）内部类可以被声明为 private 或 protected，以便防护它们，使之不受来自外部类的访问。访问保护不阻止内部类使用其他类的任何成员。

（5）一个内部类可以作为一个接口，由另一个内部类实现。

（6）声明为 static 的内部类成为顶层类。这些内部类失去了在本地范围和其他内部类中使用数据或变量的能力。

（7）内部类不能声明任何 static 成员，只有顶层类可以声明 static 成员。因此，一个需求 static 成员的内部类必须使用来自顶层类的成员。

5.2 类的继承（inheritance）

继承是面向对象程序设计的另一个重要特色，类继承也称为类派生，是指一个类可以继承其他类的非私有成员，实现代码复用。被继承的类称为父类或超类，父类包括所有直接或间接被继承的类；继承父类或超类后产生的类称为派生类或子类。

类的继承反映了客观世界的层次关系，Java 语言以 Object 类作为所有类的父类，所有的类都是直接或间接地继承 Object 类得到的。Java 还提供不同层次的标准类，使用户可根据需要派生自己的类。

在 Java 语言中，只允许单继承。所谓单继承是指每个类只有一个父类，不允许有多个父类。但一个类允许同时拥有多个子类，这时这个父类实际上是所有子类的公共成员变量和公共方法成员的集合，而每一个子类则是父类的特殊化，是对公共成员变量和方法成员的功能、内涵方面的扩展和延伸。Java 语言的多继承可通过接口来实现。

类继承不改变成员的访问权限，父类中的成员为公有的或被保护的，则其子类的成员访问权限也继承为公有的或被保护的。

5.2.1 类继承的实现

Java 中的继承是通过 extends 关键字来实现的，在定义新类时使用 extends 关键字指明新类的父类，就在两个类之间建立了继承关系。

1. 定义子类

通过继承系统类定义子类的例子已经见过的有例 1.2 的小程序和例 3.32 的自定义异常处

理程序，其类头分别为：

```
public class HelloApplet extends Applet{.....}
class MyException extends Exception {.....}
```

一般地，在类声明中，加入 extends 子句来创建一个类的子类。extends 后即为父类名，若父类名又是某个类的子类，则定义的类也是该类的（间接）子类。若无 extends 子句，则该类为 java.lang.Object 的子类。

2，类继承的传递性

类继承具有传递性，即子类继承父类的所有非私有成员，也继承父类的父类直至祖先所有的非私有成员。

3，类的成员覆盖

在类的继承中，若子类（派生类）新增的成员名称与父类（超类）成员相同，则称为成员覆盖（overriding）。

在子类中定义与父类同名成员的目的是修改父类的属性和行为，在子类中，通过名称仅能直接访问本身的成员，若有必要访问父类的同名成员，可用关键字 super。

（1）成员变量的覆盖

若子类声明了与父类同名的变量，则父类的变量被隐藏起来，直接使用的是子类的变量，但父类的变量仍占据空间，可通过 super 或父类名来访问。

【例 5.14】 在子类中定义了与父类同名的变量，从而隐藏了父类成员变量，这种隐藏变量可加 super 前缀来访问。

```
class A{
   int x = 100;
}
class B extends A{
   int  x  =  200;                                  // 在子类中定义与父类同名变量 x
   void print(){
      System.out.println("Subclass  :  "  +  x);          // 直接输出为子类变量
      System.out.println("Superclass : " + super.x);     // 父类变量用 super 访问
   }
   public static void main(String args[]){
      (new B()).print();
   }
}
```

程序运行结果如下：

```
Subclass : 200
Superclass : 100
```

需要注意的是，super 不能用于静态方法中，因为静态方法只能访问静态变量。在静态方

法中，父类的静态变量可通过类名前缀来引用。

（2）成员方法覆盖

同子类可以定义与父类同名的成员变量以实现对父类成员变量的隐藏的情况一样，子类也可以定义与父类同名的方法，实现对父类方法的覆盖。方法成员的覆盖与成员变量的隐藏的不同之处在于：子类隐藏父类的成员变量只是使得它不可见，父类的同名成员变量在子类对象中仍然占据自己的存储空间；而子类成员方法对父类同名方法的覆盖将清除父类方法占用的内存空间，从而使得父类的方法在子类对象中不复存在。

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

下述规则适用于覆盖方法：

- 覆盖方法的返回类型必须与它所覆盖的方法相同。
- 覆盖方法不能比它所覆盖的方法访问性差。
- 覆盖方法不能比它所覆盖的方法抛出更多的异常。

4. 派生类的初始化

在创建派生类的对象时，使用派生类的构造方法对其初始化，不但要对自身的成员变量赋初值，还要对父类的成员变量赋初值。因为成员变量赋初值通常在构造方法中完成，因此在 Java 语言中，允许派生类继承父类的构造方法。构造方法的继承遵循如下的原则：

（1）若父类是无参数的构造方法，则子类无条件地继承该构造方法。

（2）若子类无自己的构造方法，则它将继承父类的无参构造方法作为自己的构造方法；若子类有自己的构造方法，则在创建子类对象时，它将先执行继承自父类的无参构造方法，然后再执行自己的构造方法。

（3）若父类是有参数的构造方法，子类可以通过在自己的构造方法中使用 super 关键字来调用它，但这个调用语句必须是子类构造方法的第一个可执行语句。

【例 5.15】 继承例 5.6 中的类 RectConstructor 时对父类的成员变量赋初值。

```
class Cube extends RectConstructor{
   double height;
   Cube(double l,double w,double h){
      super(l,w); // 调用父类的构造方法，是子类构造方法中第一个可执行语句
      height = h;
   }
   void vol(){
      System.out.println("长方体体积 = "+ area() * height);
   }
}
public class CubeDemo {
   public static void main(String args[]){
      Cube c = new Cube(7,6,5);
      c.vol();
```

```
    }
}
```

本程序中，子类 Cube 继承了父类的变量 length、width 和父类的方法 area()，代码复用使得程序简洁高效。程序运行结果如下：

```
长方体体积 = 210.0
```

在本例派生类的构造方法 Cube()中，用 super()调用父类的构造方法对父类的成员变量赋初值，注意该 super()语句必须是派生类构造方法中第一条语句，不能将它放到“height = h;”语句后。

5.2.2 抽象类和抽象方法

abstract 修饰的抽象类需要子类继承，在派生类中实现抽象类中的抽象方法。抽象类被派生、抽象方法被子类实现后才有实际意义。抽象方法是只有返回值类型、方法名、方法参数而不定义方法体的一种方法。抽象方法的方法体在子类中才编写实现。注意：不能用 abstract 修饰构造方法、静态方法和私有（private）方法，也不能覆盖父类中的抽象方法。

抽象方法必须定义在抽象类中。抽象类是一种未实现的类，抽象类不能用 new 实例化一个对象。

【例 5.16】 抽象类必须要派生子类。

```
abstract class Shape {    // 定义抽象类 Shape 和抽象方法 display
    abstract void display();
}
class Circle extends Shape {
    void display() {          // 实现抽象类的方法
        System.out.println("Circle");
    }
}
class Rectangle extends Shape {
    void display() {          // 实现抽象类的方法
        System.out.println("Rectangle");
    }
}
class Triangle extends Shape {
    void display() {          // 实现抽象类的方法
        System.out.println("Triangle");
    }
}
class AbstractClassDemo {
    public static void main(String args[]) {
        (new Circle()).display();
```

```
        (new Rectangle()).display();
        (new Triangle()).display();
    }
}
```

程序的运行结果如下：

```
Circle
Rectangle
Triangle
```

5.2.3 类对象之间的类型转换

与基本数据类型之间的强制类型转换类似，在 Java 语言中，有继承关系的父类对象和子类对象之间也可以在一定条件下相互转换。父类对象和子类对象的转换需要注意以下原则：

（1）子类对象可以被视为是其父类的一个对象，反之则不可。

（2）若一个方法的形式参数定义的是父类的对象，则调用该方法的实际参数可以使用子类对象。

（3）若父类对象引用指向的实际是一个子类对象（在以前的某个时候根据（1）将子类对象的引用赋值给这个父类对象的引用），则这个父类对象的引用可以用强制类型转换转化成子类对象的引用。

5.3 接口（interface）

接口是若干完成某一特定功能的没有方法体的方法（抽象方法）和常量的集合。接口仅提供了方法协议的封装。为了获取接口功能和真正实现接口功能，需要使用类来继承该接口。在继承接口的类中，通过定义接口中抽象方法的方法体（即抽象方法的覆盖）来实现接口功能。

Java 语言使用接口来实现类间多重继承的功能，从而弥补了 Java 语言只支持类间单重继承，描述复杂实际问题处理不方便的不足。

5.3.1 接口的定义和实现

1，接口的定义

在 Java 语言中，用关键字 interface 来定义接口。接口有类似类的结构，其定义格式如下：

```
[modifier] interface interfaceName [extends superInterfaceNames]{
    //  interfaceBody
}
```

从接口定义的格式可以看到，接口定义包括两个方面的内容：定义接口名和接口体。接口名 interfaceName 是一个合法的标识符。接口体 interfaceBody 同抽象类相似，为变量和抽象方法的集合，但没有构造方法和静态初始化代码。接口体中定义的变量均为终极的（final）、

静态的（static）和公共的（public）。接口体中定义的方法均为抽象的和公共的。由于接口所有成员均具有这些特性，相关的修饰符可以省略。

在 Java 系统中也定义了不少的接口，例如，用于数据输入输出的 DataInput 接口和 DataOutput 接口，用于事件处理的 ActionListener 接口等，这些都是本书后面章节要学习使用的接口。

2. 接口的实现

在某个继承接口的派生类中为接口中的抽象方法书写语句并定义实在的方法体，称为实现这个接口。派生类实现哪个或哪些接口用 implements 说明，不能用 extends 说明。

派生类在实现接口时还要注意：若实现接口的类不是抽象类，则在该类的定义部分必须实现指定接口的所有抽象方法。方法体可以由 Java 语言书写，也可以由其他语言书写。因为是覆盖方式，所以方法头部分应该与接口中的定义完全一致，即有完全相同的参数表和返回值。

【例 5.17】 接口的实现。

```
interface Irect{      // 定义接口
  double w=3,l=4;
  void compute();
}
class Crect implements Irect{     // 定义实现接口的类
  public void compute(){
    System.out.println("边长分别为 3 和 4 的长方形面积为："+w*l);
  }
}
public class InterfaceDemo{        // 定义主类，创建接口类对象
  public static void main(String args[]){
    Crect r = new Crect();
    r.compute();
  }
}
```

程序运行结果如下：

```
边长分别为 3 和 4 的长方形面积为：12.0
```

5.3.2 接口的继承和组合

接口也可以通过关键字 extends 继承其他接口。子接口将继承父接口中所有的常量和抽象方法。此时，子接口的非抽象派生类不仅需要实现子接口的抽象方法，而且需要实现继承来的抽象方法。不允许存在未被实现的接口方法。

【例 5.18】 接口的继承。

```
interface A{                    // 定义接口 A
```

```
    String a = "在接口 A 中";
    void showA();
}
interface B extends A{       // 定义接口 B，它继承接口 A
    String b = "在接口 B 中";
    void showB();
}
interface C extends B{      // 定义接口 C，它继承接口 B
    String c = "在接口 C 中";
    void showC();
}
class InterfaceABC implements C{      // 定义实现接口 C 的类
    public void showA(){System.out.println(a);}    // 实现 public 方法
    public void showB(){System.out.println(b);}
    public void showC(){System.out.println(c);}
}
public class UseInterface2{
    public static void main(String args[]){
        InterfaceABC i = new InterfaceABC();
        i.showA();i.showB();i.showC();
    }
}
```

程序运行结果如下：

```
在接口 A 中
在接口 B 中
在接口 C 中
```

若在派生类中有未实现的方法时，编译出错信息将显示：应该声明该类为抽象类；它没有实现 XXX 方法。

在本例中，实现接口 C 的派生类 InterfaceABC 中定义抽象方法的方法体时，一定要声明方法为 public 的，否则编译将显示如下的出错信息：

```
attempting to assign weaker access privileges;
```

它的中文含义为：企图缩小方法访问权限范围。

接口继承不允许循环继承或继承自己。接口与类有些方面不同，例如，所有类的根类为类 Object，而接口没有所谓的共同根接口；接口可以同时继承多个接口，还可以通过 extends 将多个接口组合成一个接口。例如：

```
public interface Myall extends interface1,interface2{
    void doSomethingElse();
}
```

5.3.3 接口的多态

接口的使用使得方法的描述说明和方法功能的实现分开考虑，这有助于降低程序的复杂性，使程序设计灵活，便于扩充修改。这也是 Java 面向对象程序设计方法中多态特性的体现。

【例 5.19】 定义接口并实现接口，说明接口的多态。

在本程序中，定义一个接口 OneToN。在接口体中，包含一个未定义体的方法 disp()。在类 Sum 和 Pro 中分别用不同的代码实现了接口 OneToN 中的 disp()方法。在 Sum 的方法中计算 1 至 *n* 的和，在 Pro 的方法中计算 1 至 *n* 的乘积。

```
interface OneToN{
   int disp(int n);
}
class Sum implements OneToN{     // 继承接口
   public int disp(int n){       // 实现接口中的 disp 方法
      int s = 0,i;
      for(i = 1;i <= n;i ++)s += i;
      return s;
   }
}
class Pro implements OneToN{     // 继承接口
   public int disp(int n){       // 实现接口中的 disp 方法
      int m = 1,i;
      for(i = 1;i <= n;i ++)m *= i;
      return m;
   }
}
public class UseInterface{
   public static void main(String args[]){
      int n = 10;
      Sum s = new Sum();
      Pro p = new Pro();
      System.out.println("1 至 n 的和 = " + s.disp(n));
      System.out.println("1 至 n 的积 = " + p.disp(n));
   }
}
```

程序的运行结果如下：

```
1 至 n 的和 = 55
1 至 n 的积 = 3628800
```

5.3.4 接口类型的使用

在 Java 语言中，某些系统类的方法返回值类型为接口类型，这说明接口可以作为一种类型来使用。在前面介绍 Java 语言类型分类时，也已知道接口与数组、类一样，是一种引用类型。

在 Java 语言中，任何实现接口的类的实例都可以存储在该接口类型的变量中。通过这些变量可以访问类所实现的接口中的方法。Java 运行时系统动态地确定应该使用哪个类中的方法。

将接口作为一种数据类型可以不需要了解对象所对应的具体的类，而着重于它的交互界面。例 5.20 以例 5.19 所定义的接口 OneToN 以及实现该接口的类 Sum 和 Pro 为例，其中以 OneToN 作为引用类型来使用。

【例 5.20】 接口类型的使用。

```
public class UseInterface1{
  public static void main(String args[]){
    int n = 10;
    OneToN otn;
    Sum s = new Sum();
    otn = s;  // 在接口类型变量中存储 Sum 类的实例
    System.out.println("1 至 n 的和 = " + otn.disp(n));
    Pro p = new Pro();
    otn = p;  // 在接口类型变量中存储 Pro 类的实例
    System.out.println("1 至 n 的积 = " + otn.disp(n));
  }
}
```

程序的运行结果与例 5.19 相同。

5.4 包（package）

一组相关的类和接口集合称为包。包体现了 Java 语言面向对象特性中的封装机制，包将 Java 语言的类和接口有机地组织成层次结构，这个层次结构与具体的文件系统的目录树结构层次一致。因此，Java 包就是具有一定相关性，在文件系统中可准确定位的 Java 文件的集合。

5.4.1 创建包

包由包语句 package 创建，其语法格式如下：

```
package [package1[.package2[.[...]]]]
```

关键字 package 后的 package1 是包名，在 package1 下允许有次一级的子包 package2，package2 下可以有更次一级的子包 package3 等等。各级包名之间用“.”号分隔。通常情况下，包名称的元素被整个地小写。

在 Java 程序中，package 语句必须是程序的第一条非空格、非注释语句。通过 package 语句，可将 Java 程序分层次地存放在不同的目录下，目录名称与包的名称相同。

【例 5.21】　在例 5.1 的类定义前加语句 package 创建包。

```
package ch05;  // Rect.java 文件名
```

编译程序完成生成 Rect.class 文件后，可将当前目录的 Rect.class 文件复制或移动到创建的 ch05 子目录中。

5.4.2　使用包

将类组织为包的目的是为了更好地利用包中的类。一般情况下，一个类只能引用与它在同一个包中的类。

在 Java 程序中，若要用到某些包中的类或接口，一种方法是在程序的开始部分写出相应的引入（import）语句，指出要引入哪些包的哪些类。另一种方法不用引入语句，直接在要引入的类和接口前给出其所在包名。无论采用哪种方法，使用系统类的前提是这个系统类应该是用户程序可见的类。

1，使用 import 语句

import 语句用于灵活地实现在编译单元中使用外部类和接口的引入机制，引入语句不必考虑类和接口的引入顺序以及是否被多次引入。

import 语句的格式与意义如下：

```
import PackageName；                // 引入 PackageName 包
import PackageName.Identifier；     // 引入 PackageName 包中的类和接口
import PackageName.*；              // 引入 PackageName 包中的全部类和接口
```

2，直接使用包

这种方法一般用在程序中引用类和接口次数较少的时候，在要引入的类和接口前直接给出其所在包名。例如：

```
java.applet.Applet ap = new java.applet.Applet();
```

在一些 Java 程序中，还使用全局惟一包名（Globally Unique Package Name）的引用形式。全局是相对于 Internet 和 Intranet 而言的。全局惟一包名通常用一个 Internet 域名经过简单变换命名。例如，sun.com 和 ibm.com 等，将域名前后颠倒，得到 com.sun 和 com.ibm 等，这些作为引用包名的前缀，再加上组织部门、项目、硬件系统名称等。例如：

```
com.sun.java.io.*;
```

3，使用 CLASSPATH 环境变量

CLASSPATH 环境变量的作用与 DOS 的 PATH 和 APPEND 命令作用类似，当一个程序找不到它所需要的其他类的.class 文件时，系统会自动到 CLASSPATH 环境变量所指明的路径中去查找。

通过 SET 命令设置 CLASSPATH，可设置程序对类的搜索路径。若设置错误，Java 程序将不能正常执行。下面是一个设置 CLASSPATH 的 SET 命令：

```
SET CLASSPATH = .;c:\jdk1.4\lib;c:\jdk1.4\lib\classes.zip
```

它将 Java 类搜索路径设置为当前目录、c:\jdk1.4\lib 目录和 c:\jdk1.4\lib\classes.zip。

对 Java 应用程序，还可以通过设置 Java 解释器开关参数来指定类文件的搜索路径。例如，对于 Sun 公司的 Jave SDK 解释器 java.exe，有开关参数-classpath；对于 Microsoft 公司的 Visual J++中的 Java 解释器 jview.exe，有参数-cp。

例如，若需要解释执行的 Hello.class 文件不在当前目录，而在 D 盘根目录下 jfile 目录中，则可以使用如下的命令行语句来执行 SDK 解释器：

```
java Hello -classpath d:\jfile
```

【例 5.22】 对例 5.3 中类 RectDemo 引用例 5.21 中的 ch05 包中的类 Rect 的使用，可在例 5.3 的类定义前添加语句：

```
package ch05;
import ch05.*;
```

编译完成产生 class 文件后，将其从当前目录复制或移动到 ch05 子目录下，可在当前目录下用如下的命令来执行：

```
java ch05.RectDemo
```

程序运行结果同例 5.3 一样。

5.4.3 类及类成员的访问权限

Java 程序将数据和对数据的处理代码封装为类，并以类为程序的基本单位，但类又被封装在包中。要访问类或封装在类中的数据和代码，必须清楚在什么情况下它们是可访问的。

一个类总可以访问和调用自己的变量和方法，但这个类之外的程序其他部分是否能访问这些变量和方法，则由该变量和方法以及它们所属类的访问控制符决定。

1，类成员的访问权限

Java 将类的成员可见性（可访问性）划分为五种情况，按照可见性的范围大小从小到大列出如下：

（1）仅在本类内可见。

（2）在本类及其子类可见。

（3）在同一包内可见。

（4）在同一包内及其子类（不同包）可见。

（5）在所有包内可见。

类成员的可访问性与定义时所用的修饰符 private（私有）、protected（保护）、private protected（私有保护）和 public（公共）有关。声明为 private 的类成员仅能在本类内被访问；声明为 protected 的类成员可以在本类、本包、本类的子类被访问；声明为 private protected 的类成员可以在本类、本类的子类被访问；声明为 public 的类成员可以在所有包内被访问；未用修饰符声明的类成员，则隐含为在本包内可被访问。

2. 类的访问权限

类通常只用两种访问权限：默认和 public。类声明为 public 时，可以被任何包的代码访问；默认时，可被本包的代码访问。因为类封装了类成员，因此，类成员的访问权限也与类的访问权限有关。例如，public 访问权限的类成员封装在默认修饰符的类中，则该类成员只能在本包内被访问。

为清楚起见，将类成员的可访问性总结在表 5.1 中。其中，“√”表示允许使用相应的变量和方法。注意：表中列出的类成员可访问性是针对 public 类的。

表 5.1 类成员的可访问性

	无修饰符	private	private protected	protected	public
同类	√	√	√	√	√
同包，子类	√		√	√	√
同包，非子类	√			√	√
不同包，子类			√	√	√
不同包，非子类					√

5.4.4 Java 的应用程序接口（API）

Java 的应用程序接口 API 是以包的形式提供的，每个包内包含大量相关的类、接口和异常。这些是 Java 程序设计时要充分利用的资源。编写 Java 程序不需要从头开始，只需针对所要解决的问题，用自己编写的类来继承系统提供的有关标准类，这样可以提高编程效率，降低代码出错的可能。因此，学习 Java 的一个重要任务是了解、掌握、运用 Java 的标准类库。

下面介绍几个 Java API 的主要包。

（1）java.lang

java.lang 是 Java 语言的核心包，有 Java 程序所需要的最基本的类和接口，包括 Object 类、基本数据类型包装类、数学类、异常处理类、线程类、字符串处理类、系统与运行类和类操作类等。这个包由编译器自动引入。

（2）java.applet

java.applet 包是用来实现运行于 Internet 浏览器中的 Java Applet 的工具类库，它包含少量的几个接口和一个非常有用的类 java.applet.Applet。

（3）java.awt

java.awt 包是 Java 抽象窗口工具箱包，包含许多字体和颜色设置、几何绘图、图像显示、图形用户接口操作的类和接口。

（4）java.io

java.io 包是 Java 语言的标准输入/输出类库，包含实现 Java 程序与操作系统、外部设备以及其他 Java 程序做数据交换所使用的类，例如基本输入/输出流、文件输入/输出流、过滤输入/输出流、管道输入/输出流、随机输入/输出流等，还包含了目录和文件管理类等。

（5）java.net

java.net 是 Java 网络包，实现网络功能。

（6）java.util

java.util 包包含了 Java 语言中的一些低级的实用工具，如处理时间的 Date 类，处理变长数组的 Vector 类，实现栈和杂凑表的 Stack 类和 HashTable 类等。

使用包中系统类的方法有三种：一种是继承系统类，在用户程序中创建系统类的子类，例如 Java Applet 程序的主类作为 java.applet 包中 Applet 类的子类；第二种方法是创建系统类的对象，例如创建包装类的对象；最后一种方法是直接使用系统类，例如程序中常用的 System.out.println()方法，就是系统类 System 的静态属性 out 的方法。

习 题 五

5.1 定义一个表示学生信息的类 Student，要求如下：

（1）类 Student 的成员变量（根据实际数据的情况，自定类型）：

sNO 表示学号

sName 表示姓名

sSex 表示性别

sAge 表示年龄

sJava 表示 Java 语言课程的成绩

（2）类 Student 的方法成员：

getNo() 获得学号

getName() 获得姓名

getSex() 获得性别

getAge() 获得年龄

getJava() 获得 Java 课程成绩

5.2 编程：按照题 5.1 中学生类 Student 的定义，创建两个该类的对象，存储并输出两个学生的信息（数据初始化不使用构造方法），计算并输出这两个学生 Java 语言成绩的平均值。

5.3 编程：重做题 5.2，学生数据初始化使用构造方法来实现。

5.4 编程：对题 5.1 的 Student 学生类，创建 5 个 Student 学生对象，求它们 Java 语言课程成绩的最高分和最低分。

5.5 什么是继承？什么是父类或超类？什么是子类或派生类？什么是单重继承或多重继承？什么是终极类？

5.6 什么是成员变量的隐藏？什么是方法的覆盖？方法的覆盖与成员变量的隐藏有何不同？方法的覆盖与方法的重载有何不同？

5.7 父类对象与子类对象相互转换的条件是什么？如何实现它们的相互转换？

5.8 根据下面的要求实现圆类 Circle。

（1）圆类 Circle 的成员变量：

radius 表示圆的半径

（2）圆类 Circle 的方法成员：

Circle() 构造方法，将半径置 0

Circle(double r) 构造方法，创建 Circle 对象时将半径初始化为 r

double getRadius() 获得圆的半径值

double getPerimeter() 获得圆的周长

double getArea() 获得圆的面积

void disp() 将圆的半径、圆的周长和圆的面积输出到屏幕

5.9 通过继承题 5.8 中的圆 Circle 类，派生圆柱体类 Cylinder。要求如下：

（1）圆柱体类 Cylinder 的成员变量：

height 表示圆柱体的高

（2）圆柱体类 Cylinder 的方法成员：

Cylinder(double r，double h) 构造方法，创建 Cylinder 对象时将圆半径初始化为 r，圆柱高初始化为 h

double getHeight() 获得圆柱体的高

double getVol() 获得圆柱体的体积

void dispVol() 将圆柱体的体积输出到屏幕

5.10 编程：按照题 5.8 的 Circle 类，接收键盘的输入值作为圆的半径，计算圆的周长和面积。

5.11 编程：按照题 5.9 的 Cylinder 类，接收键盘的输入值作为圆的半径和圆柱体的高，计算圆柱体的体积。

5.12 什么是接口？接口的功能是什么？接口与类有何异同？

5.13 什么是包？包有何作用？

5.14 如何引用包中的某个类？如何引用整个包？

5.15 下列类定义中哪些是合法的抽象类定义？

（1）class Animal { abstract void growl(); }

（2）abstract Animal {abstract void growl();}

（3）class abstract Animal {abstract void growl();}

（4）abstract class Animal {abstract void growl();}

（5）abstract class Animal {abstract void growl() {System.out.println("growl");}}

5.16 定义一个不能被继承的名为 Abc 的类的正确方法是什么？

（1）class Abc {}

（2）native class Abc {}

（3）abstract final class Abc {}

（4）class Abc {final;}

（5）final class Abc {}

5.17 考查下列的两个类后回答问题：

```
public class ClassA {
    public void method1(int i) {}
    public void method2(int i) {}
    public static void method3(int i) {}
    public static void method4(int i) {}
}
public class ClassB extends ClassA {
    public static void method1(int i) {}
    public void method2(int i) {}
    public void method3(int i) {}
    public static void method4(int i) {}
}
```

（1）**哪个方法覆盖了父类的方法**?
（2）**哪个方法隐藏了父类的方法**?
（3）**其他方法做什么**?

第 6 章　字符串处理

字符串是字符的序列，它是组织字符的基本数据结构，对于绝大多数程序来说都是很重要的。Java 将字符串当做对象来处理，它提供了一系列的方法对整个字符串进行操作，使得字符串的处理更加容易和规范。在本章中，将讨论字符串的表示、生成、访问、修改以及一些处理字符串的方法。

Java 语言中的包 java.lang 中封装了 final 类 String 和 StringBuffer，其中类 String 对象是字符串常量，建立后不能改变。而类 StringBuffer 对象类似于一个字符缓冲区，建立后可以修改。

6.1　类 String 字符串

6.1.1　类 String 字符串的定义

String 类是字符串常量类，String 对象建立后不能修改。以前使用的每个字符串常量（用双引号括起来的一串字符）实际上都是 String 对象，如字符串“Java”在编译后即成为 String 对象。因此，可以用字符串常量直接初始化一个 String 对象。例如：

```
String s = "Hello World. ";
```

由于每个字符串常量对应一个 String 类的对象，所以对一个字符串常量可以直接调用类 String 中提供的方法，例如：

```
int len = "Hello World! ".length();
```

将返回字符串的长度 12，字符串的长度即字符串中字符的个数。

通过类 String 提供的构造方法，可以生成一个空字符串（不包含任何字符的字符串），也可以由字符数组或字节数组来生成一个字符串对象。默认的构造方法不需要任何参数，它生成一个空字符串。例如：

```
String s = new String(); // 建立一个空字符串对象
```

其他创建 String 对象的构造方法有：

- String(String value)用已知串 value 创建一个字符串对象。
- String(char chars[])用字符数组 chars 创建一个字符串对象。
- String(char chars[],int startIndex,int numChars)用字符数组 chars 的 startIndex 位置开始的 numChats 个字符，创建一个字符串对象。
- String(byte ascii[],int hiByte)用字节数组 ascii 创建一个字符串对象，Unicode 字符的高位字节为 hiByte，通常应该为 0。
- String(byte ascii[],int hiByte,int startIndex,int numChars)用字节数组 ascii 创建一个字符串对象。其参数的意义同上。

由于在 Internet 上通常使用的字符都为 8 位的 ASCII 码，Java 提供了从字节数组来初始化字符串的方法，并且用 hiByte 来指定每个字符的高位字节。对 ASCII 码来说，hiByte 应为 0，对于其他非拉丁字符集，hiByte 的值应该非 0。

【例 6.1】　类 String 构造方法的使用。

```
public class StringConstructors{
   public static void main(String args[]){
      String s,s1,s2,s3,s4,s5,s6,s7;
      byte byteArray[] = {(byte)'J',(byte)'a',(byte)'v',(byte)'a'};
      char charArray[] = {'程','序','设','计'};
      StringBuffer sb = new StringBuffer("欢迎");;
      s = new String("Hello!");
      s1 = new String();
      s2 = new String(s);
      s3 = new String(sb);
      s4 = new String(charArray,2,2);
      s5 = new String(byteArray,0);
      s6 = new String(charArray);
      s7 = new String(byteArray,0,0,1);
      System.out.println("s   = "+ s );
      System.out.println("s1 = "+ s1);
      System.out.println("s2 = "+ s2);
      System.out.println("s3 = "+ s3);
      System.out.println("s4 = "+ s4);
      System.out.println("s5 = "+ s5);
      System.out.println("s6 = "+ s6);
      System.out.println("s7 = "+ s7);
   }
}
```

程序运行结果如下：

```
s   = Hello!
s1 =
s2 = Hello!
s3 = 欢迎
s4 = 设计
s5 = Java
s6 = 程序设计
s7 = J
```

6.1.2 类 String 的常用方法

类 String 中提供的访问 String 字符串的方法很多，大体上可分为类转换、子字符串、比较、修改等几类。

1. 类 String 字符串的比较

类 String 中提供了一些方法，用来进行字符串的比较。

（1）boolean equals(Object anObject)和 equalsIgnoreCase(String anotherString)

方法 equals()和 equalsIgnoreCase()用来比较两个字符串的值是否相等，不同的是后者忽略字母的大小写。例如：

```
System.out.println("abc".equals("abc"));                    // 输出为 true
System.out.println("abc".equalsIgnoreCase("ABC"));    // 输出为 true
```

注意：它们与运算符“==”实现的比较是不同的。运算符“==”比较两个字符串对象是否引用同一个实例对象，而 equals()和 equalsIgnoreCase()则比较两个字符串中对应的每个字符是否相同。例如：

```
String s = "abc";
String s1= "abc";
System.out.println( s == s1 );                  // 输出为 false
System.out.println("abc" == "abc");          // 输出为 true
```

（2）int compareTo(String anotherString)

为了比较两个字符串的大小，类 String 中实现了方法 compareTo()，通过返回的整数值指明当前字符串与参数字符串的大小关系。若调用的串对象比参数大，返回正整数。反之，返回负整数。相等则返回 0。

若比较的两个字符串有不同的字符，则从左边数起的第一个不同字符的大小即两个字符串的大小，字符的大小建立在 Unicode 字符集基础上。方法的返回值为：

```
this.charAt(k)-anotherString.charAt(k)
```

例如：

```
System.out.println("this".compareTo("that"));          // 输出为 8
```

若比较的两个字符串各个位置的字符都相同，仅长度不同，则方法的返回值为：

```
this.length()-anotherString.length()
```

例如：

```
System.out.println("abc".compareTo("abcd") );          // 输出为 -1
```

方法 int compareToIgnoreCase(String str)是与上述方法功能相同的方法，但忽略字母大小写。

（3）boolean startWith(String prefix)和 endWith(String suffix)

这两个方法均有重载：

- boolean startWith(String prefix,int offset)
- boolean endWith(String suffix,int offset)

方法 startWith()和 endWith()用来比较当前字符串的起始字符（或子字符串）prefix 和终止字符（或子字符串）suffix 是否和调用的字符（或字符串）相同，重载的方法中同时还可以指定比较的开始位置 offset。

（4）boolean regionMatches(int toffset,String other,int ooffset,int len)

该方法有重载：

```
boolean regionMatches(boolean ignoreCase,int toffset,String other,int ooffset,int len);
```

regionMatches 方法的功能是比较两个字符串中指定区域的子字符串是否相同。

其中，toffset 和 ooffset 分别指明当前字符串和参数字符串中所要比较的子字符串的起始位置，len 指明比较的长度，而 ignoreCase 指明比较时是否区分大小写。无此参数，比较是区分大小写的。

2，类 String 子字符串和定位

（1）char charAt(int index)

该方法的功能是返回字符串 index 处的字符，index 的值从 0 到串长度减 1。例如：

```
System.out.println("China".charAt(1));   //  输出为 h
```

（2）int indexOf(int ch)和 lastIndexOf(int ch)

方法 indexOf()有重载：

- int indexOf(int ch,int fromIndex)
- int indexOf(String str)
- int indexOf(String str,int fromIndex)

以上方法的功能是返回字符串对象中指定的字符和子串首次出现的位置，从串对象开始处或从 fromIndex 处开始查找。若未找到，则返回-1。

方法 lastIndexOf()也有重载：

- int lastIndexOf(int ch,int fromIndex)
- int lastIndexOf(String str)
- int lastIndexOf(String str,int fromIndex)

这几个方法与相应的 indexOf 方法相比，不同之处是从串对象的右端开始查找。

【例 6.2】 lastIndexOf 的使用。

```
class LastNumber {
  public static void main(String args[]) {
    String s1 = "67.89,55.87,-56.45,11.22,78.9";
    int i1 = s1.lastIndexOf(',');
    String s2 = s1.substring(i1 + 1);
    System.out.println(s2);
  }
```

```
}
```

程序运行结果为:

```
78.9
```

（3）String substring(int beginindex)**和** substring(int beginindex,int endindex)

该方法的功能是返回子字符串。前者返回从位置 beginindex **开始处到串尾的子字符串；后者是返回从** beginindex **开始到** endindex-1 **为止的子字符串，子串长度为** endindex - beginindex**。**

【例 6.3**】 类** String **方法** indexOf **和** substring **的使用。**

```
class DollarAmount {
  public static void main(String args[]) {
    String s1 = "这块显示卡的售价为$45.60";
    int i1 = s1.indexOf('$');
    String s2 = s1.substring(i1);
    System.out.println(s2);
  }
}
```

程序运行结果为:

```
$45.60
```

【例 6.4**】 显示一周星期日至星期六的名称。**

```
class WeekName{
  public static void main(String args[]){
    String xq = "日一二三四五六",s;
    int i;
    for(i=0;i<7;i++){
      s=xq.substring(i,i+1);
      System.out.println("  星期"+s);
    }
  }
}
```

程序运行结果为:

```
星期日
星期一
星期二
星期三
星期四
星期五
星期六
```

3，类 String 字符串对象的修改

这里，修改的含义是将修改后的字符串对象赋给新的对象或直接输出。

（1）String toLowerCase()**和** String toUpperCase()

方法 toLowerCase()**和** toUpperCase()**的功能是分别将字符串中的字母转换为小写和大写。例如：**

```
System.out.println("abcde".toUpperCase()); // 输出为 ABCDE
System.out.println("ABCDE".toLowerCase()); // 输出为 abcde
```

（2）replace(char oldChar,char newChar)

该方法的功能是将字符串中 oldChar **字符用** newChar **字符替换。例如：**

```
System.out.println("41177".replace('7','8')); // 输出为 41188
```

（3）String trim()

该方法的功能是截去字符串两端的空白字符（whitespace，**不大于**'\u0020'**的字符**）。**例如：**

```
String s = "    Java    ";
String s1 = "他 10 岁。";
System.out.println(s.trim()+s1);        // 输出为：Java 他 10 岁。
```

4，类 String 与其他类的转换

（1）static String valueOf(Object obj)

该方法的功能是将其他类的参数转换为字符串对象返回。为了方便起见，类 String **中提供了重载的静态方法** valueOf()，**用来把其他类型的对象转换为字符串对象。重载方法有：**

- static String valueOf(char data[])
- static String valueOf(char data[],int offset,int count)
- static String valueOf(boolean b)
- static String valueOf(char c)
- static String valueOf(int i)
- static String valueOf(long l)
- static String valueOf(float f)
- static String valueOf(double d)

例如：

```
System.out.println(String.valueOf(Math.PI)); // PI 转换为 String 对象
```

注：其他类也提供了方法 valueOf()**把一个字符串转换为相应的类对象。例如：**

```
String piStr = "3.14159";
Float pi = Float.valueOf(piStr);   // String 对象转换为 Float 对象
```

（2）toString()

java.lang.Object **中提供了方法** toString()**把对象转换为字符串，这一方法通常被重写以适**

合子类的需要。下面是一个类 Integer 使用 toString()方法的例子。

方法 static String toString(int i,int radix)用来将整数 i 转换为 radix 进制的数字字符串。例如:

```
String s = Integer.toString(123,8);        // 将十进制的 123 转换为八进制数据
System.out.println(s);                     // 输出为 173
```

在实际应用中，也经常需要将字符串转换为其他类型的数据。下面是一个例子。

【例 6.5】 String 数字字符串转换为数值。

```
class StringToInt {
  public static void main(String args[]) {
    String s = "125";
      Integer obj = Integer.valueOf(s);
    int i = obj.intVa lue();
    // 上两行也可合写为 int i = Integer.valueOf(s).intValue();
    i += 10;
    System.out.println(i);
  }
}
```

5，类 String 的其他方法

（1）int length()

方法 length()的功能为返回类 String 字符串对象的长度。

【例 6.6】 求几个给定字符串的平均长度。

```
class StringAverage {
  public static void main(String args[]) {
    String array[] = new String[3];
    array[0] = "Short string";
    array[1] = "This is a complete sentence!";
    array[2] = "This is the longest " +
                    "element in the array";
    int total = array[0].length();
    total += array[1].length();
    total += array[2].length();
    System.out.println("平均字符串长度为： " + total/3);
  }
}
```

程序运行结果如下：

```
平均字符串长度为： 26
```

（2）String concat(String str)

方法 concat()**的功能是将** str **连接到调用串对象的后面，即进行字符串的连接操作。例如：**

```
System.out.println("41177".concat("abc"));// 输出为 41177abc
```

在 Java **语言中，重载的运算符“+”也可用来实现字符串的连接或字符串与其他类对象的连接。例如：**

```
String s = "Java" + "程序设计";          // s = "Java 程序设计"
int age = 20;
String s1 = "他" + age + "岁。";          // s1 = "他 20 岁。"
```

6.2 类 StringBuffer 字符串

6.2.1 类 StringBuffer 字符串的定义

类 StringBuffer **的对象是一种可以改变（如增长、修改）的字符串对象，使用起来比** String **类更加方便。在** Java **语言中支持字符串的加运算，实际上就是使用了** StringBuffer **类。**

类 StringBuffer **提供了下面几种构造方法对一个可变的字符串对象进行初始化。**

- StringBuffer()**建立空的字符串对象。**
- StringBuffer(int len)**建立长度为** len **的字符串对象。**
- StringBuffer(String s)**建立一个初值为** String **类对象** s **的字符串对象。**

例如：

```
StringBuffer sb1 = new StringBuffer();
StringBuffer sb2 = new StringBuffer(30);
StringBuffer sb3 = new StringBuffer("StringBuffer");
```

若不给任何参数，则系统为字符串分配 16 **个字符大小的缓冲区，这是默认的构造方法。参数** len **则指明字符串缓冲区的初始长度。参数** s **给出字符串的初始值，同时系统还要再为该串分配** 16 **个字符大小的空间。**

这里有两个概念：长度 length **和容量** capacity**，前者是类** StringBuffer **对象中包含字符的数目，而后者是缓冲区空间的大小。例如，对上述初始化的** sb3**，它的** length **为** 12（"StringBuffer" **的长度），但它的** capacity **为** 28**。**

6.2.2 类 StringBuffer 的常用方法

1，设置/获取类 StringBuffer 字符串对象长度和容量

（1）void ensureCapacity(int minimumCapacity)

本方法的功能是设定字符串缓冲区的大小。保证缓冲区的容量至少为指定的值 minimumCapacity**。若当前缓冲区容量小于参数值，则分配新的较大的缓冲区。新容量取以下两者中的大者：**

- **参数** minimumCapacity **的值。**

▪ **老容量的两倍加** 2。

例如:

```
StringBuffer sb = new StringBuffer("china");
sb.ensureCapacity(30);
System.out.println(sb.capacity());    // 输出为 44(老容量的两倍加 2)
```

(2) void setLength(int newLength)

本方法的功能是指明字符串的长度,这时字符串缓冲区中指定长度以后的字符值均为零。例如:

```
StringBuffer sb = new StringBuffer("china");
sb.setLength(3);                              // 输出为 chi
```

(3) int length()

本方法的功能是获取字符串对象的长度。

(4) int capacity()

本方法的功能是获取缓冲区的大小。

【例 6.7**】** StringBuffer **字符串长度** Length **和容量** Capacity **意义的区别。**

```
class StringBufferDemo {
  public static void main(String args[]) {
    StringBuffer sb1 = new StringBuffer();
    StringBuffer sb2 = new StringBuffer(30);
    StringBuffer sb3 = new StringBuffer("abcde");
    System.out.println("sb1.capacity = " + sb1.capacity());
    System.out.println("sb1.length = " + sb1.length());
    System.out.println("sb2.capacity = " + sb2.capacity());
    System.out.println("sb2.length = " + sb2.length());
    System.out.println("sb3.capacity = " + sb3.capacity());
    System.out.println("sb3.length = " + sb3.length());
  }
}
```

程序运行结果为:

```
sb1.capacity = 16
sb1.length = 0
sb2.capacity = 30
sb2.length = 0
sb3.capacity = 21
sb3.length = 5
```

2，修改类 StringBuffer 字符串对象

修改类 StringBuffer **对象的常用方法有** append、insert、delete **等。**

（1）StringBuffer append(Object obj)

方法 append()**用于在串缓冲区的字符串末尾添加各种类型的数据。重载的方法如下：**

- StringBuffer append(boolean b)
- StringBuffer append(int i)
- StringBuffer append(long l)
- StringBuffer append(float f)
- StringBuffer append(double d)

【例 6.8**】** StringBuffer **字符串连接操作。**

```
class StringBufferAppend {
  public static void main(String args[]) {
    tringBuffer sb = new StringBuffer("abcde");
    sb.append("fgh");
    sb.append("ijklmnop");
    System.out.println(sb);
    System.out.println("sb.capacity = " + sb.capacity());
    System.out.println("sb.length = " + sb.length());
  }
}
```

程序运行结果如下：

```
abcdefghijklmnop
sb.capacity = 21
sb.lengt h = 16
```

（2）StringBuffer insert()

方法 insert()**用于在串缓冲区的指定位置** offset **处插入各种类型的数据。重载的方法有：**

- StringBuffer insert(int offset,int i)
- StringBuffer insert(int offset,long l)
- StringBuffer insert(int offset,float f)
- StringBuffer insert(int offset,double d)

【例 6.9**】** StringBuffer **字符串插入操作。**

```
class StringBufferInsert {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("abcde");
    sb.insert(0, "012345");
    System.out.println(sb);
  }
```

```
}
```

程序运行结果如下：

```
012345abcde
```

（3）StringBuffer delete(int start,int end)**和** deleteCharAt(int index)

方法 delete()**用来从** StringBuffer **字符串对象中删去从** start **开始到** end-1 **结束的子字符串。例如：**

```
StringBuffer sb = new StringBuffer("aabbcc");
System.out.println(sb.delete(2,4));                    // 输出为 aacc
```

方法 deleteCharAt()**用来删除** index **处的字符。例如：**

```
StringBuffer sb = new StringBuffer("china");
System.out.println(sb.deleteCharAt(4));                // 输出为 chin
```

（4）StringBuffer reverse()

方法 reverse()**用于对** StringBuffer **类字符串对象进行颠倒操作。设原字符串的长度为** n，**新字符串中** k **位置的字符在原字符串中的位置为** $n-k-1$。

【例 6.10**】　对** StringBuffer **字符串进行颠倒操作。**

```
class StringBufferReverse {
   public static void main(String args[]) {
      StringBuffer sb1 = new StringBuffer("abcde");
      sb1.append("abcdefghij");
      StringBuffer sb2 = sb1.reverse();
      System.out.println(sb1);
      System.out.println(sb2);
   }
}
```

程序运行结果如下：

```
jihgfedcbaedcba
jihgfedcbaedcba
```

（5）void setCharAt(int index,char ch)

本方法的功能是设置指定位置 index **的字符值** ch**。例如：**

```
StringBuffer sb = new StringBuffer("aaAaa");
sb.setCharAt(2,'中');       // sb 为 aa 中 aa
```

（6）StringBuffer replace(int start,int end,String str)

该方法的功能是将 StringBuffer **对象字符串从** start **至** end -1 **处，用字符串** str **替换。例如：**

```
StringBuffer sb = new StringBuffer("********");
System.out.println(sb.replace(2,6,"Java"));   // 输出为 **Java**
```

3，类 StringBuffer 对象的子字符串和字符

（1）void getChars(int srcBegin,int srcEnd,char[] dst,int dstBegin)

该方法的功能是将 StringBuffer 对象字符串中字符复制到目标字符数组中去。复制的字符从 srcBegin 处开始，到 srcEnd-1 处结束。复制字符的个数为 srcEnd-srcBegin。字符被复制到目标数组的 dstBegin 至 dstBegin + (srcEnd -srcBegin) -1 处。例如：

```
char a[]= {'*','*','*','*','*','*','*','*'};
StringBuffer sb = new StringBuffer("Java");
sb.getChars(0,4,a,2);
System.out.println(a);// 输出为 **Java**
```

（2）String substring(int start)

该方法的功能是获取 StringBuffer 字符串对象从 start 处开始的子串。重载的方法 String substring(int start,int end)用来获取 StringBuffer 字符串对象从 start 至 end-1 处之间的子串。例如：

```
StringBuffer sb = new StringBuffer("Java Programming");
System.out.println(sb.substring(5));     // 输出为 Programming
System.out.println(sb.substring(0,4));   // 输出为 Java
```

4，类型转换

在 java.lang.Object 中提供了方法 toString 将指定对象转换为字符串，与类 Character、Integer、Boolean 等一样，类 StringBuffer 也重写了方法 toString()，可将类 StringBuffer 的对象转换成类 String 的对象。例如：

```
StringBuffer sb = new StringBuffer("aabbcc");
String s = sb.toString();
System.out.println(s);              // 输出为 aabbcc
```

6.3 main 方法的参数

通过前面 Java 应用程序实例的学习可以看到，在类方法 main()名后的括号中有 String 类的数组参数 args，args 可以用来接收 Java 命令行传送给 Java 应用程序的数据。所谓命令行参数，是指在执行字节码文件时，在命令行上字节码文件名后给出的内容。例如：

```
java HelloEcho How are you? 123
```

这里“How are you? 123”就是命令行参数，在 Java 应用程序中通过 args 来得到并处理这些内容。命令行参数有多个时，一般用空格来分隔，如上述的命令行参数有 4 个。在 Java 应用程序中，将这些参数一个一个按顺序存入类 String 的数组 args，第一个参数存入 args[0]，第二个参数存入 args[1]等等。按照上述命令行的写法，“How”存入 args[0]，“are” 存入 args[1]，“you?”存入 args[2]，“123”存入 args[3]。注意：它们都是 String 类的对象。

若要将包含多个空格的单词作为一个字符串，可用引号括起来。例如：

```
java HelloEcho " How are you?" 123
```

【例 6.11】　显示所有命令行参数。

```
class CommandLine {
  public static void main(String args[]) {
    for(int i=0; i<args.length; i++)
    System.out.println("args[" + i + "]: " +
    args[i]);
  }
}
```

若执行程序的命令为：

```
java CommandLine how are you?
```

则程序运行结果为：

```
how are you?
```

【例 6.12】　用符号“#”画一个宽为 *w* 高为 *h* 的空心方框。*w* 和 *h* 用命令行参数方式提供。

```
public class DoRect{
  public static void main(String args[]){
    int w = Integer.valueOf(args[0]).intValue();
    int h = Integer.valueOf(args[1]).intValue();
    Rectangle myrect = new Rectangle(w,h);
    myrect.drawRect();
  }
}
class Rectangle{
  int width,height,area;
  public Rectangle(int w,int h){
    width = w;
    height = h;
    area = getArea(w,h);
  }
  protected int getArea(int w,int h){
    int a;
    a = w * h;
    return a;
  }
  public void drawRect(){
```

```
        int i,j;
        for(i = width;i > 0;i --)System.out.print("#");
        System.out.println();
        for(i = height - 2;i > 0;i --){
            System.out.print("#");
            for(j = width - 2;j > 0;j --)System.out.print(" ");
            System.out.print("#");
            System.out.println();
        }
        for(i = width;i > 0;i --)System.out.print("#");
        System.out.println("");
    }
}
```

用 javac 编译该程序后，可以用 java 解释器来执行它，具体过程如下：

```
编译：javac DoRect.java
运行：java DoRect 20 5
```

这里命令行参数“20　5”，它存入 DoRect 的 main()方法的 args 变量。其中 args[0]的值为“20”，args[1]的值为“5”，Integer.valueOf(String string).intValue 的功能是把字符串 string 中的数字转化为一个整型值。

参数是 20 5 时的程序执行结果为：

```
####################
#                  #
#                  #
#                  #
####################
```

即用符号“#”画一个宽为 20 高为 5 的方框。

注意：main()方法的参数必须提供，否则，Java 应用程序能通过编译，但在运行时会产生如下的异常：

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

6.4　字符串应用举例

【例 6.13】　用命令行方式提供 1 至 3 个整数，按顺序分别为年、月、日数据。若仅提供一个整数，则为年号，程序判断该年是否为闰年；若提供两个整数，则为年号、月号，程序输出为该年月的天数；若提供三个整数，则为年、月、日数据，程序输出这一天是星期几。若未提供任何数据或提供的数据个数超过三个，则程序输出出错信息并给出程序的使用方法。

在本程序中使用了两个公式：对给定年号判断是否为闰年和计算该年的元旦是星期几。为方便阅读程序，这里列出相应的计算方法。

（1）判断是否为闰年

满足下列条件之一即为闰年：

- 年号能被 400 整除；
- 年号能被 4 整除，但不能被 100 整除。

（2）计算元旦是星期几

设年号为 y，先计算 s = y + (y -1)/ 4 - (y -1)/ 100 + (y -1)/ 400（除法运算全部取整数商），再对 s 取除以 7 的余数，则余数为 0 至 6 之间的数，即为星期数（0 为星期日）。例如，计算 2002 年的元旦是星期几，s = 2002 + 500 -20 + 5 = 2487，s 除以 7 的余数为 2，所以，2002 年的元旦是星期二。

```
class StringExam{
  public static void main(String args[]){
    String weekname = "日一二三四五六",s,leap;
    int y,m,d,t = 0,w = 0,i;
    if(args.length < 1 || args.length > 3){
      System.out.println("参数个数错误！");
      System.out.println("请重新启动程序，" +
                         "在命令行上提供 1 至 3 个整数参数！");
    }
    else{
      y = Integer.valueOf(args[0]).intValue();
      leap = (y % 400 == 0 || y % 4 ==0 && y % 100 != 0)? "闰" : "平";
      if(args.length == 3){
        m = Integer.valueOf(args[1]).intValue();
        d = Integer.valueOf(args[2]).intValue();
        w = y + (y-1)/ 4 - (y-1)/ 100 + (y-1)/ 400;
        for(i = 1;i <= m - 1; i++)t += month_num(i,leap);
        w = (w + t + d - 1) % 7;
        s = weekname.substring(w,w + 1);
        System.out.println(y + " 年 "+ m +" 月 "+ d +" 日是星期"+ s +"。" );
      }
      else if(args.length == 2){
        m = Integer.valueOf(args[1]).intValue();
        t = month_num(m,leap);
        System.out.println(y + " 年 "+ m +" 月有 "+ t +" 天。");
      }
      else {
        System.out.println(y + " 年是" + leap + "年。");
      }
```

```
        }
    }
    static int month_num(int m,String b){ // 求各月的天数
        String big = "1 3 5 7 8 10 12",s = String.valueOf(m);
        int n,w;
        if(m == 2)
            n = (b == "闰") ? 29 : 28;
        else{
            w = big.indexOf(s);                    // 大月天数为 31 天
            n = 31;
            if(w == -1) n--;                       // 小月在大月的天数上减去 1 天
        }
        return n;
    }
}
```

下面是三次运行程序的结果：

（1）java StringExam 2002

程序输出：2002 **年是平年。**

（2）java StringExam 2002 2

程序输出：2002 **年** 2 **月有** 28 **天。**

（3）java StringExam 2002 2 12

程序输出：2002 **年** 2 **月** 12 **日是星期二。**

习 题 六

6.1 **编译运行下面程序时有何种结果？请选择一个正确的答案。**

```
class Test {
    public static void main() {
        System.out.println("hello");
    }
}
```

（1）**程序不能编译，因为** main()**方法定义不正确。**

（2）**程序能编译，但不能运行，因为** class **未声明为** public **的。**

（3）**程序能无错地编译并运行，但运行时产生异常。**

（4）**程序运行时显示“**hello**”。**

6.2 **编程：输入一个字符串，将其中的大写英文字母改为小写字母，小写字母改为大写字母，然后输出。**

6.3 **编程：检查输入的字符串是否为“回文”。当一个字符串顺读倒读时都一样，则这样的字符串就是“回文”。例如，“**level**”、“**madam**”，“**123321**”等都是“回文”。**

6.4　编程：从输入的字符串中删去输入的所有子字符串。例如，若输入的字符串为“this is a string.”，输入的子字符串为“is”，则删除后的字符串为“th　a string.”。

6.5　编程：从命令行方式输入的字符串中删去所有重复的字符（每种字符只保留一个）。例如，若输入“hello”，则删除后的字符串为“helo”。

6.6　编程：以每行 5 个数据的格式输出 n 和 m 之间的所有素数。n 和 m 用命令行方式提供。

6.7　编程：统计一个字符串中给定字符出现的频率。

6.8　编程：统计一个字符串中给定子字符串出现的频率。

6.9　编写一个方法：将一个表示十进制数的字符串转换为以逗号分隔的字符串，从右边开始每三个数字标一个逗号。例如，给定一个字符串“1234567”，该方法返回“1,234,567”。

6.10　改写 6.9 题的方法，可以接收参数以指定分隔字符串和分隔字符之间数字的数目。

第 7 章　输入输出处理

7.1　输入/输出概述

7.1.1　输入/输出流概念

为了使得一个 Java 程序能与外界交流数据信息，Java 语言必须提供输入/输出的功能。例如，从键盘读取数据，从文件中读数据或向文件中写数据，将数据输出到打印机以及在一个网络连接上进行读写操作等。输入/输出时，数据在通信通道中流动。所谓“数据流（stream）”，指的是所有数据通信通道之中数据的起点和终点。例如，执行程序通常会输出各种信息到显示器，使用户可以随时了解程序的状态信息，而这些信息的通道就是一个数据流，其中的数据就是要显示的信息，数据的源（起点）就是执行的程序，而数据的终点就是显示器。又例如，一个程序在打开某一文件时，程序和文件之间就建立起一个数据流，文件的内容就是数据流中的数据。若这个文件是程序所要读取的文件，那么数据流的源就是文件，而目的就是程序；若要对文件进行写入操作时，情况相反。总之，只要是数据从一个地方“流”到另外一个地方，这种数据流动的通道都可以称为数据流。

从程序设计的角度看，从数据流的概念编写程序也是比较简单的。当程序是数据流的源时，一旦建立起数据流后，便可以不去理会数据流的目的是什么（可能是显示器、打印机、网络系统中的远端客户等等），可以将对方看成是一个会接受数据的“黑匣子”，程序只负责提供数据就可以了。而程序若是数据流的终点目的，那么等到数据流建立完成后，也同样不必关心数据流的起点是什么，只要索取自己想要使用的数据就可以了。

输入/输出是相对于程序来说的。程序在使用数据时所扮演的角色有两个：一个是源，一个是目的。若程序是数据流的源，即数据的提供者，这个数据流对程序来说就是一个“输出数据流”（数据从程序流出）。若程序是数据流的终点，这个数据流对程序而言就是一个“输入数据流”（数据从程序外流向程序）。

7.1.2　输入/输出类

在 java.io 包中提供了 60 多个类（流），从功能上分为两大类：输入流和输出流。输入数据用输入流，输出数据用输出流。从流结构上可分为字节流（以字节为处理单位或称面向字节）和字符流（以字符为处理单位或称面向字符）。字节流的输入流和输出流基础是 InputStream 和 OutputStream 这两个抽象类，字节流的输入输出操作由这两个类的子类实现。类 RandomAccessFile 是一个例外，它允许对文件进行随机访问，可以同时对文件进行输入（读）或输出（写）操作。字符流是 Java 1.1 版后新增加的以字符为单位进行输入输出处理的流，字符流输入输出的基础是抽象类 Reader 和 Writer。下面对一些常用的输入输出流类进行介绍。

1，字节流 InputStream 和 OutputStream 类

（1）InputStream

InputStream 中包含一套字节输入流需要的方法，可以完成最基本的从输入流读入数据的功能。当 Java 程序需要外设的数据时，可根据数据的不同形式，创建一个适当的 InputStream 子类类型的对象来完成与该外设的连接，然后再调用执行这个流类对象的特定输入方法，如 read()，来实现对相应外设的输入操作。InputStream 类的类层次如下所示。

```
InputStream
├ FileInputStream
├ ByteArrayInputStream
├ PipedInputStream
├ SequenceInputStream
├ StringBufferInputStream
└ FilterInputStream
    ├ DataInputStream （实现 DataInput 接口）
    ├ LineNumberInputStream
    ├ BufferedInputStream
    └ PushbackInputStream
```

InputStream 类子类对象自然也继承了 InputStream 类的方法。常用的方法有：读数据的方法 read()，获取输入流字节数的方法 available()，定位输入位置指针的方法 skip()、reset()、mark()等。

（2）OutputStream

OutputStream 中包含一套字节输出流需要的方法，可以完成最基本的输出数据到输出流的功能。当 Java 程序需要将数据输出到外设时，可根据数据的不同形式，创建一个适当的 OutputStream 子类类型的对象来完成与该外设的连接，然后再调用执行这个流类对象的特定输出方法，如 write()，来实现对相应外设的输出操作。OutputStream 类的类层次如下所示。

```
OutputStream
├ ObjectOutputStream
├ ByteArrayOutputStream
├ FileOutputStream
├ PipedOutputStream
└ FilterOutputStream
    ├ DataOutputStream （实现 DataOutput 接口）
    ├ BufferedOutputStream
    └ PrintStream
```

OutputStream 类子类对象也继承了 OutputStream 类的方法。常用的方法有：写数据的方法 write()，关闭流方法 close()等。

在 InputStream 类和 OutputStream 类子类中，FilterInputStream 和 FilterOutputStream 过滤

流抽象类又派生出 DataInputStream 和 DataOutputStream 数据输入输出流类等子类。过滤流的主要特点是在输入输出数据的同时能对所传输的数据做指定类型或格式的转换，即可实现对二进制字节数据的理解和编码转换。数据输入流 DataInputStream 中定义了多个针对不同类型数据的读方法，如 readByte()、readBoolean()、readShort()、readChar()、readInt()、readLong()、readFloat()、readDouble()、readLine()等。数据输出流 DataOutputStream 中定义了多个针对不同类型数据的写方法，如 writeByte()、writeBoolean()、writeShort()、writeChar()、writeInt()、writeLong()、writeFloat()、writeDouble()、writeChars()等。这些方法大大方便了数据的输入输出操作。

2，字符流 Reader 和 Writer 类

（1）Reader

Reader 中包含一套字符输入流需要的方法，可以完成最基本的从输入流读入数据的功能。当 Java 程序需要外设的数据时，可根据数据的不同形式，创建一个适当的 Reader 子类类型的对象来完成与该外设的连接，然后再调用执行这个流类对象的特定输入方法，如 read()，来实现对相应外设的输入操作。Reader 类的类层次如下所示。

```
Reader
├ BufferedReader
│   └ LineNumberReader
├ CharArrayReader
├ FilterReader
│   └ PushbackInputStream
├ InputStreamReader
│   └ FileReader
├ PipedReader
└ StringReader
```

（2）Writer

Writer 中包含一套字符输出流需要的方法，可以完成最基本的输出数据到输出流的功能。当 Java 程序需要将数据输出到外设时，可根据数据的不同形式，也要创建一个适当的 Writer 子类类型的对象来完成与该外设的连接，然后再调用执行这个流类对象的特定输出方法，如 write()，来实现对相应外设的输出操作。Writer 类的类层次如下所示。

```
Writer
├ BufferedWriter
├ CharArrayWriter
├ FilterWriter
├ OutputStreamWriter
│   └ FileWriter
├ PipedWriter
├ StringWriter
└ PrintWriter
```

7.1.3 标准输入/输出

为方便使用计算机常用的输入输出设备，各种高级语言与操作系统对应，都规定了可用的标准设备（文件）。所谓标准设备（文件），也称为预定义设备（文件），在程序中使用这些设备（文件）时，可以不用专门的打开操作就能简单地应用。一般地，标准输入设备是键盘，标准输出设备是终端显示器，标准错误输出设备也是显示器。

Java 语言的系统类 System 提供访问标准输入输出设备的功能。System 类是继承 Object 类的终极类，它有三个类变量：in、out 和 err，分别表示标准输入、标准输出和标准错误输出流。

（1）标准输入

System 类的类变量 in 表示标准输入流，其定义为：

```
public static final InputStream in
```

标准输入流已打开，做好提供输入数据的准备。一般这个流对应键盘输入，可以使用 InputStream 类的 read()和 skip(long n)等方法来从输入流获得数据。read()从输入流中读一个字节，skip(long n)在输入流中跳过 n 个字节。

（2）标准输出

System 类的类变量 out 表示标准输出流，其定义为：

```
public static final PrintStream out
```

标准输出流也已打开，做好接收数据的准备。一般这个流对应显示器输出，可以使用 PrintStream 类的 print()和 println()等方法来输出数据，这两个方法支持 Java 的任意基本类型作为参数。标准输出允许输出重定向。

（3）标准错误输出

System 类的类变量 err 表示标准错误输出流，其定义为：

```
public static final PrintStream err
```

标准错误输出流已打开，做好接收数据的准备。一般这个流也对应显示器输出，与 System.out 一样，可以访问 PrintStream 类的方法。标准错误输出不能重定向。

【例 7.1】 读写标准文件。将键盘输入的字符输出到屏幕并统计输入的字符数。

```
import java.io.*;
class MyType{
   public static void main(String args[])throws IOException{
      int b,count = 0;
      System.out.println("请输入：");
      while((b = System.in.read()) != -1){
         count++; System.out.print((char)b);
      }
      System.out.println(); // 输出换行
      System.err.println("\n 输入了"+count+"个字符。");// 使用标准错误输出设备
```

```
    }
}
```

程序运行时，显示“请输入：”，这时可在光标提示处输入任意字符，按回车键则显示输入字符。输入结束按组合键<Ctr>+<z>，则立即显示输入的字符数（若输入过程中按了回车键，则一个回车键计为两个字符，即回车、换行字符）。

7.2 文件的顺序访问

由于 Java 输入输出流类种类繁多，本节开始仅能介绍一些主要的输入输出流类的使用，未介绍的内容应能举一反三加以应用。输入输出流操作的一般步骤如下：

（1）使用引入语句引入 java.io 包：“import java.io.*;”。

（2）根据不同数据源和输入输出任务，建立字节流或字符流对象。

（3）若需要对字节或字符流信息组织加工为数据，在已建字节流或字符流对象上构建数据流对象。

（4）用输入输出流对象类的成员方法进行读写操作，需要时设置读写位置指针。

（5）关闭流对象。

其中步骤（2）～（5）要考虑异常处理。

7.2.1 字节流（InputStream 类和 OutputStream 类）

前面已经介绍过，InputStream 类和 OutputStream 类都是抽象类，不能直接生成对象，要通过继承类来生成程序中所需要的对象。在继承类中，一般将 InputStream 类和 OutputStream 类中的方法重写，以提高效率或为了特殊流的需要。

1. FileInputStream 和 FileOutputStream 类

类 FileInputStream 和 FileOutputStream 分别直接继承于 InputStream 和 OutputStream，它们重写或实现了父类中的一些方法以顺序访问本地文件，是字节流操作的基础类。

（1）创建字节输入文件流 FileInputStream 类对象

若需要以字节为单位顺序读出一个已存在文件的数据，可使用字节输入流 FileInputStream。可以用文件名、文件对象或文件描述符建立字节文件流对象。FileInputStream 类构造方法有：

- FileInputStream(String name) 用文件名 name 建立流对象。

例如：

```
FileInputStream fis = new FileInputStream ("c:/config.sys");
```

- FileInputStream(File file)用文件对象 file 建立流对象。

例如：

```
File myFile = new File("c:/config.sys");
FileInputSteam fis = new FileInputStream(myFile);
```

若创建 FileInputStream 输入流对象成功，就相应地打开了该对象对应的文件，接着就可

以从文件读取信息了。若创建对象失败，将产生异常 FileNotFoundException，这是一个非运行时异常，必须捕获和抛出，否则编译会出错。

（2）读取文件信息

从 FileInputStream 流中读取字节信息，一般用 read()成员方法，该方法有重载：

- int read()读流中一个字节，若流结束则返回-1。
- int read(byte b[])从流中读字节填满字节数组 b，返回所读字节数，若流结束则返回-1。
- int read(byte b[],int off, int len)从流中读字节填入 b[off]开始处，返回所读字节数，若流结束则返回-1。

（3）创建字节输出文件流 FileOutputStream 类对象

FileOutputStream 可表示一种创建并顺序写的文件。在构造此类对象时，若指定路径的文件不存在，会自动创建一个新文件；若指定路径已有一个同名文件，该文件的内容将被保留或删除。

FileOutputStream 对象用于向一个文件写数据。像输入文件一样，也要先打开这个文件后才能写这个文件。要打开一个 FileOutputStream 对象，像打开一个输入流一样，可以将字符串或文件对象作为参数。FileOutputStream 类的构造方法有：

- FileOutputStream(String name) 用文件名 name 创建流对象。

例如：

```
FileOutputStream fos = new FileOutputStream("d:/out.dat");
```

- FileOutputStream(File file) 用文件对象 file 建立流对象。

例如：

```
File myFile = new File("d:/out.dat");
FileOutputStream fos = new FileOutputStream(myFile);
```

上述两种格式的构造方法还允许使用第二个参数：boolean append。若这个参数的值为 true，则向文件尾输出字节流（指定文件已存在时，该文件原有内容将保留），即为添加数据方式使用字节流。

（4）向输出流写信息

向 FileOutputStream 中写入信息，一般用 write()方法，该方法有重载：

- void write(int b)将整型数据的低字节写入输出流。
- void write(byte b[])将字节数组 b 中的数据写入输出流。
- void write(byte b[],int off,int len)将字节数组 b 中从 off 开始的 len 个字节数据写入输出流。

（5）关闭 FileInputStream

当完成一个文件的操作，可用两种方法关闭它：显式关闭和隐式关闭。隐式关闭是让系统自动关闭它，Java 有自动垃圾回收的功能。显式关闭是使用 close()方法，例如：

```
fos.close();
```

【例 7.2】 完成文件复制功能。用命令行方式提供源文件名和目标文件名。

```
//用法：java CopyFile 源文件名 目标文件名
```

```
import java.io.*;
class CopyFile {
   public static void main(String args[])throws IOException{
      int i;
      FileInputStream fin;
      FileOutputStream fout;
      try {
         try {
            fin = new FileInputStream(args[0]);    // 打开输入文件
         } catch(FileNotFoundException e) {
            System.out.println("输入文件未找到！");
            return;
         }
         try {
         fout = new FileOutputStream(args[1]); // 打开输出文件
         } catch(FileNotFoundException e) {
            System.out.println("打开输出文件错误！");
            return;
         }
      } catch(ArrayIndexOutOfBoundsException e) {
         System.out.println("用法: CopyFile  源文件  目标文件");
         return;
      }
      try {    // 复制文件
         while((i = fin.read())!= -1) fout.write(i);
      } catch(IOException e) {
         System.out.println("文件错误！");
      }
      fin.close();
      fout.close();
   }
}
```

【例 7.3】　在终端上显示指定文本文件的内容，文本文件名通过命令行方式提供。

```
//用法：java ShowFile TEST.TXT
import java.io.*;
class ShowFile {
   public static void main(String args[])throws IOException{
      int i;
      FileInputStream fin = null;
```

```
        try {
            fin = new FileInputStream(args[0]);
        } catch(FileNotFoundException e) {
            System.out.println("文件未找到！");
            System.exit(-1);
        } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("用法: java ShowFile 文件名");
            System.exit(-2);
        }
        // 读字符直到遇到 EOF
        while(( i = fin.read()) != -1)System.out.print((char) i);
        fin.close();
    }
}
```

2，DataInputStream 和 DataOutputStream 类

字节文件流 FileInputStream **和** FileOutputStream **只能提供纯字节或字节数组的输入/输出，如果要进行基本数据类型如整数和浮点数的输入/输出，则要用到过滤流类的子类二进制数据文件流** DataInputStream **和** DataOutputStream **类。这两个类的对象必须和一个输入类或输出类联系起来，而不能直接用文件名或文件对象建立。**

使用数据文件流的一般做法是分两步：首先用前面的方法建立字节文件流对象，然后基于字节文件流对象建立数据文件流对象，再用此对象的方法对基本类型的数据进行输入/输出。

DataInputStream **类的构造方法如下：**

- DataInputStream(InputStream in)**创建过滤流** FilterInputStream **对象并为以后的使用保存** InputStream **参数** in**。**

DataOutputStream **类的构造方法如下：**

- DataOutputStream(OutputStream out)**创建输出数据流对象写数据到指定的** OutputStream**。**

【例 7.4**】 在** C **盘根目录下建立文件** fib.dat**，存储** Fibonacci**数列的前** 20 **个数。**Fibonacci **数列的前两个数是** 1**，从第三个数开始，是其前两个数之和。即** 1,1,2,3,5,8, 13,21,…

```
import java.io.*;
class FibOut {
    public static void main(String args[]) {
        try {
            // 创建字节文件输出流
            OutputStream fos = new FileOutputStream("c:/fib.dat");
            DataOutputStream dos = new DataOutputStream(fos); // 创建数据输出流
            // 写 20 个 Fibonacci 数至数据输出流
            int count = 0,i = 1,j = 1;
```

```
            for(;count < 20; count++){
              dos.writeInt(i);
              int k = i + j;
              i = j;
              j = k;
            }
            fos.close();   // 关闭文件输出流
          }catch(Exception e) {
            System.out.println("Exception: " + e);
          }
          System.out.println("文件创建成功!");
        }
      }
```

程序运行正常结束后，可以在 C 盘的根目录见到创建的文件 fib.dat，这是二进制数据文件，用系统的 type 命令查看其内容是无意义的，可用下面的例 7.5 程序来查看其内容。

【例 7.5】　从例 7.4 建立的文件中读取 Fibonacci 数据并显示到屏幕上。

```
import java.io.*;
import java.text.*; // 数据格式化输出需要
class FibIn {
  public static void main(String args[]) {
    DecimalFormat df = new DecimalFormat("0000   ");      // 数据格式化类
    try {
      // 创建文件输入流
      FileInputStream fis = new FileInputStream("c:/fib.dat");
      DataInputStream dis = new DataInputStream(fis);      // 创建数据输入流
      for(int i = 0; i < 20; i++) {                        // 读出数据并显示
        if(i % 10 == 0)System.out.println();
        System.out.print(df.format(dis.readInt()));
      }
      fis.close();                                         // 关闭文件输入流
    }
    catch(Exception e) {
      System.out.println("Exception: " + e);
    }
  }
}
```

本程序中使用了 DecimalFormat 类对象的 format()方法，它将整数参数按 DecimalFormat 类对象的格式转换为字符串输出。为对齐数据，在数字前输出了前导的 0，每个数字后有两个空字符。

程序的运行结果如下：

```
0001  0001  0002  0003  0005  0008  0013  0021  0034  0055
0089  0144  0233  0377  0610  0987  1597  2584  4181  6765
```

上述两例中使用了整数的输入方法 readInt()**和输出方法** writeInt()**，其他可用的输入/输出方法见表** 7.1**。**

表 7.1　DataInputStream **和** DataOutputStream **类的方法**

数 据 类 型	DataInputStream	DataOutputStream
byte	readByte	writeByte
short	readShort	writeShort
int	readInt	writeInt
long	readLong	writeLong
float	readFloat	writeFloat
double	readDouble	writeDouble
boolean	readBoolean	writeBoolean
char	readChar	writeChar
String	readUTF	writeUTF
byte[]	readFully	

对于其他基本类型数据的输入/输出方法，从上面整数输入/输出方法的命名规则，可以容易地写出相应的方法名。

3. BufferedInputStream **和** BufferedOutputStream **类**

若处理的数据量较多，为避免每个字节的读写都对流进行，可以使用过滤流类的子类缓冲流。缓冲流建立一个内部缓冲区，输入输出数据先读写到缓冲区中进行操作，这样可以提高文件流的操作效率。

缓冲输出流 BufferedOutputStream **类提供和** FileOutputStream **类同样的写操作方法，但所有输出全部写入缓冲区中。当写满缓冲区或关闭输出流时，它再一次性输出到流，或者用** flush() **方法主动将缓冲区输出到流。**

当创建缓冲输入流 BufferedInputStream **时，一个输入缓冲区数组被创建，来自流的数据填入缓冲区，一次可填入许多字节。**

（1）**创建** BufferedOutputStream **流对象**

若要创建一个 BufferedOutputStream **流对象，首先需要一个** FileOutputStream **流对象，然后基于这个流对象创建缓冲流对象。**

BufferedOutputStream **类的构造方法如下：**

- BufferedOutputStream(OutputStream out)**创建缓冲输出流，写数据到参数指定的输出流，缓冲区设为默认的** 512 **字节大小。**
- BufferedOutputStream(OutputStream out, int size)**创建缓冲输出流，写数据到参数指定的输出流，缓冲区设为指定的** size **字节大小。**

例如，下面的代码可创建一个缓冲输出流 bos**：**

```
FileOutputStream fos = new FileOutputStream("/user/dbf/stock.dbf");
BufferedOutputStream bos = new BufferedOutputStream(fos);
```

（2）用 flush()方法更新流

要想在程序结束之前将缓冲区里的数据写入磁盘，除了填满缓冲区或关闭输出流外，还可以显式调用 flush()方法。flush()方法的声明为：

```
public void flush() throws IOException
```

例如：

```
bos.flush();
```

（3）创建 BufferedInputStream 流对象

BufferedInputStream 类的构造方法如下：

- BufferedInputStream(InputStream in) 创建 BufferedInputStream 流对象并为以后的使用保存 InputStream 参数 in，并创建一个内部缓冲区数组来保存输入数据。
- BufferedInputStream(InputStream in, int size) 用指定的缓冲区大小 size 创建 BufferedInputStream 流对象，并为以后的使用保存 InputStream 参数 in。

（4）缓冲流类的应用

缓冲流类一般与另外的输入输出流类一起配合使用。对例 7.5，可以将流对象定义修改为：

```
FileInputStream fis = new FileInputStream("c:/fib.dat");
BufferedInputStream bis = new BufferedInputStream(fis);
DataInputStream dis = new DataInputStream(bis);
```

4. PrintStream 类

过滤流类的子类 PrintStream 类提供了将 Java 的任何类型转换为字符串类型输出的能力。输出时，可应用经常使用的方法 print()和 println()。

创建 PrintStream 流也需要 OutputStream 流对象。PrintStream 类的构造方法有：

- public PrintStream(OutputStream out)创建一个新的打印流对象。
- public PrintStream(OutputStream out,boolean autoFlush)创建一个新的打印流对象。布尔值的参数 autoFlush 为 true 时，当写一个字节数组、引用 println()方法或写 newline 字符或写字节('\n')时，缓冲区内容将被写到输出流。

【例 7.6】 使用 PrintStream 流。

```
import java.io.*;
import java.awt.*;
public class PrintStreamDemo{
  public static void main(String args[])throws IOException{
    FileOutputStream fos = new FileOutputStream("PrintStream");
    PrintStream ps = new PrintStream(fos);
    Button b1 = new Button("Button"); // 创建任一对象，这里是创建一个按钮
    ps.println(123);
```

```
        ps.println(3.1415926);
        ps.println("123"+456);
        ps.println(123==123.0);
        ps.println(b1);   //  打印对象时，调用对象的 toString()方法
        ps.close();
        fos.close();
    }
}
```

程序运行后，在系统提示符下执行下面的 type 命令：

```
type printstream
```

屏幕显示的结果如下：

```
123
3.1415926
123456
true
java.awt.Button[button0,0,0,0x0,invalid,label=Button]
```

7.2.2 字符流（Reader 类和 Writer 类）

由于 Java 采用 16 位的 Unicode 字符，因此需要基于字符的输入/输出操作。从 Java 1.1 版开始，加入了专门处理字符流的抽象类 Reader 和 Writer，前者用于处理输入，后者用于处理输出。这两个类类似于 InputStream 和 OuputStream，也只是提供一些用于字符流的接口，本身不能用来生成对象。

从 7.1 节的 IO 类层次图可以看到，Reader 和 Writer 类也有较多的子类，与字节流类似，它们用来创建具体的字符流对象进行 IO 操作。这里介绍一些常用的字符流子类及其主要方法。字符流的读写等方法与字节流的相应方法都很类似，但读写对象使用的是字符。

1，InputSteamReader 和 OutputStreamWriter 类

这是 java.io 包中用于处理字符流的基本类，用来在字节流和字符流之间搭一座“桥”。这里字节流的编码规范与具体的平台有关，可以在构造流对象时指定规范，也可以使用当前平台的默认规范。

InputSteamReader 和 OutputStreamWriter 类的构造方法如下：

- public InputSteamReader(InputSteam in)
- public InputSteamReader(InputSteam in,String enc)
- public OutputStreamWriter(OutputStream out)
- public OutputStreamWriter(OutputStream out,String enc)

其中 in 和 out 分别为输入和输出字节流对象，enc 为指定的编码规范（若无此参数，表示使用当前平台的默认规范，可用 getEncoding()方法得到当前字符流所用的编码方式）。

读写字符的方法 read()、write()，关闭流的方法 close()等，与 Reader 和 Writer 类的同名

方法用法都是类似的。

2，FileReader 和 FileWriter 类

FileReader 和 FileWriter 类是 InputSteamReader 和 OutputStreamWriter 类的子类，利用它们可方便地进行字符输入/输出操作。

FileReader 类的构造方法有：

- FileReader(File file) 对指定要读的 file 创建 FileReader 对象。
- FileReader(String fileName) 对指定要读的 fileName 创建 FileReader 对象。

FileWriter 类的构造方法有：

- FileWriter(File file) 对指定的 file 创建 FileWriter 对象。
- FileWriter(String fileName) 对指定的 fileName 创建 FileWriter 对象。

这里列出的 FileWriter 类的两个构造方法都可带第二个布尔值的参数 append，当 append 为 true 时，为添加到输出流。

FileReader 类中可用的方法有：read()返回输入字符，read(char[] buffer)输入字符到字符数组中等。

FileWriter 类中常用的方法有：write(String str)和 write(char[] buffer)输出字符串，write(int char)输出字符，flush()输出缓冲字符，close()在执行 flush 后关闭输出流，getEncoding()获得文件流字符的编码等。

【例 7.7】 使用 FileWriter 类输出字符。

```
import java.io.*;
class FileWriterDemo {
   public static void main(String args[]){
      FileWriter out = null;
      try{
         out = new FileWriter("FileWrite.txt");
         System.out.println("Encoding:" + out.getEncoding());
         out.write("Java Programming.");
         out.close();
      }catch(IOException e){}
   }
}
```

程序运行结果如下：

```
Encoding:GBK
```

执行系统命令 >type filewriter.txt 后显示如下：

```
Java Programming.
```

3. BufferedReader 和 BufferedWriter 类

缓冲字符流类 BufferedReader 和 BufferedWriter 的使用，可提高字符流处理的效率。它们

的构造方法如下：

- public BufferedReader(Reader in)
- public BufferedReader(Reader in,int sz)
- public BufferedWriter(Writer out)
- public BufferedWriter(Writer out,int sz)

其中 in **和** out **分别为字符流对象，** sz **为缓冲区大小。从上述构造方法的声明可以看出，缓冲流的构造方法是基于字符流创建相应的缓冲流。**

在 BufferedReader **和** BufferedWriter **类中，除了** Reader **和** Writer **中提供的基本读写方法外，增加了对整行字符的处理方法** readLine()**和** newLine()**。前者从输入流中读取一行字符，行结束标志为回车符和换行符；后者向字符输出流中写入一个行结束标记，该标记是由系统定义的属性** line.separator**。**

【例 7.8**】 创建顺序文本文件。使用** FileWriter **类和** BufferedWriter **类并用** write()**方法写文件。**

```
import java.io.*;
class BufferedWriterDemo {
  public static void main(String args[]) {
    try {
      FileWriter fw = new FileWriter(args[0]);          // 创建字符输出流对象
      BufferedWriter bw = new BufferedWriter(fw);  // 创建缓冲字符输出流对象
      for(int i = 0; i < 10; i++) {                          // 将字符串写至文件
        bw.write("Line " + i + "\n\r");
      }
      bw.close();                                                 // 关闭缓冲字符输出流
    }catch(Exception e) {
      System.out.println("Exception: " + e);
    }
  }
}
```

若程序编译成功，运行命令为：

```
java BufferedWriterDemo p7-8.txt   // 指明创建的输出文件为 p7-8.txt
```

程序运行时，没有任何屏幕上的输出，但程序运行完成后，可见到当前目录下已经创建了 p7-8.txt **文件，这是一个文本文件，可用系统命令** type **查看其内容，内容为** Line 0 **至** Line 9 **共** 10 **行信息。**

【例 7.9**】 读顺序文本文件。使用** FileReader **类和** BufferedReader **类并用** readLine()**方法读文件。**

```
import java.io.*;
class BufferedReaderDemo {
  public static void main(String args[]) {
```

```
        try {
            FileReader fr = new FileReader(args[0]);          // 创建文件字符流对象
            BufferedReader br = new BufferedReader(fr);  // 创建缓冲字符流对象
            String s;
            while((s = br.readLine()) != null)
                System.out.println(s);
            fr.close();   // 关闭流
        }
        catch(Exception e) {
            System.out.println("Exception: " + e);
        }
    }
}
```

4. PrintWriter 类

PrintWriter 类提供字符流的输出处理。由于该类的对象可基于字节流或字符流来创建，写字符的方法 print()、println()可直接将 Java 基本类型的数据转换为字符串输出，用起来很方便。

PrintWriter 类的构造方法如下：

- PrintWriter(OutputStream out)
- PrintWriter(OutputStream out, boolean autoFlush)
- PrintWriter(Writer out)
- PrintWriter(Writer out, boolean autoFlush)

例如，为文件 test.txt 创建 PrintWriter 对象 pw 的语句可为：

```
PrintWriter pw = new PrintWriter(new FileOutputStream("test.txt"));
```

或

```
PrintWriter pw = new PrintWriter(new FileWriter("test.txt"));
```

【例 7.10】 将键盘输入的字符存储到命令行参数指定的文件中。

```
import java.io.*;
public class ReadKey{
    public static void main(String args[]){
        File f = new File(args[0]);
        try{
            BufferedReader br =
                new BufferedReader(new InputStreamReader(System.in));
            PrintWriter pw = new PrintWriter(new FileWriter(f));
            String s;
            System.out.println("请输入文本：(按 Ctrl+z 结束输入。)");
```

```
            while((s = br.readLine())!=null)pw.println(s);
            br.close();
            pw.close();
          }catch(IOException e){
            e.printStackTrace();
          }
        }
    }
```

7.3 文件的随机访问

有时读文件不是从头至尾顺序读的，也可能想将一个文本文件当做一个数据库，读完一个记录后，跳到另一个记录，这些记录在文件的不同地方；或者对一个文件进行又读又写的操作。Java 提供的 RandomAccessFile 类可进行这种类型的输入输出。

RandomAccessFile 类直接继承于 Object，但由于实现了 DataInput 和 DataOutput 接口而与同样实现该接口的 DataInputStream 和 DataOutputStream 类方法很类似。

1，建立随机访问文件流对象

建立 RandomAccessFile 类对象类似于建立其他流对象，RandomAccessFile 类的构造方法如下：

- RandomAccessFile(File file, String mode)
- RandomAccessFile(String name, String mode)

其中，name 为文件名字符串，file 为 File 类的对象，mode 为访问文件的方式，有“r”或“rw”两种形式。若 mode 为“r”，则文件只能读出，对这个对象的任何写操作将抛出 IOException 异常；若 mode 为“rw”并且文件不存在，则该文件将被创建。若 name 为目录名，也将抛出 IOException 异常。

例如，打开一个数据库后更新数据：

```
RandomAccessFile rf = new RandomAccessFile ("/usr/db/stock.dbf", "rw");
```

2，访问随机访问文件

RandomAccessFile 对象的读写操作和 DataInput/DataOutput 对象的操作方式一样，可以使用在 DataInputStream 和 DataOutputStream 里出现的所有 read()和 write()方法。

3，移动文件指针

随机访问文件的任意位置的数据记录读写是通过移动文件指针指定文件读写位置来实现的。与文件指针有关的常用方法有：

- public long getFilePointer()throws IOException 返回文件指针的当前字节位置。
- public void seek(long pos) throws IOException 将文件指针定位到一个绝对地址 pos。pos 参数指明相对于文件头的偏移量，地址 0 表示文件的开头。例如，将文件 rf 的文件指针

移到文件尾，可用语句：

```
rf.seek(rf.length());
```

其中，public long length()throws IOException **返回文件的长度。地址“**length()**”表示文件的结尾。**

- public int skipBytes(int n)throws IOException **将文件指针向文件尾方向移动** n **个字节。**

4，向随机访问文件增加信息

可以用访问方式“rw**”打开随机访问文件后，向随机访问文件增加信息。例如：**

```
rf = new RandomAccessFile("c:/config.sys","rw");
rf.seek(rf.length());              //任何顺序写将添加到文件
```

【例 7.11**】 使用随机访问文件读写数据。**

```
import java.io.*;
public class RandomIODemo {
    public static void main(String args[]) throws IOException {
      RandomAccessFile rf = new RandomAccessFile("random.txt","rw");
      rf.writeBoolean(true);
      rf.writeInt(123456);
      rf.writeChar('j');
      rf.writeDouble(1234.56);
      rf.seek(1);
      System.out.println(rf.readInt());
      System.out.println(rf.readChar());
      System.out.println(rf.readDouble());
      rf.seek(0);
      System.out.println(rf.readBoolean());
      rf.close();
    }
}
```

程序的输出结果如下：

```
123456
j
1234.56
true
```

【例 7.12**】 显示指定文本文件最后** n **个字符。文本文件名和数字** n **用命令行参数的方式提供。**

```
import java.io.*;
class RandomIODemo2{
```

```
    public static void main(String args[]) {
      try {
        RandomAccessFile rf = new RandomAccessFile(args[0], "r");
        long count = Long.valueOf(args[1]).longValue();
        long position = rf.length();
        position -= count;
        if(position < 0) position = 0;
        rf.seek(position);
        while(true) {
          try {
            byte b = rf.readByte();
            System.out.print((char)b);
          }catch(EOFException eofe) {
            break;
          }
        }
      }catch(Exception e) {
        e.printStackTrace();
      }
    }
}
```

若程序正确编译后，执行命令为:

```
java RandomIODemo2 RandomIODemo2.java 50
```

这个命令表示显示程序 RandomIODemo2.java **的最后** 50 **个字符，程序的运行结果如下：**

```
ch(Exception e) {
e.printStackTrace();
}
}
}
```

7.4 目录和文件管理

java.io **包中的** File **类提供了与具体平台无关的方式来描述目录和文件对象的属性功能，其中包含大量的方法可用来获取路径、目录和文件的相关信息，并对它们进行创建、删除、改名等管理工作。因为不同的系统平台对文件路径的描述不尽相同，为做到平台无关，在** Java **语言中使用抽象路径等概念。** Java **自动进行不同系统平台的文件路径描述与抽象文件路径之间的转换。**

File **类的直接父类是** Object **类。**

7.4.1 目录管理

目录操作的主要方法为：

- public boolean mkdir() **根据抽象路径名创建目录。**
- public String[] list() **返回抽象路径名表示路径中的文件名和目录名。**

7.4.2 文件管理

在进行文件操作时，常需要知道一个关于文件的信息。Jave **的** File **类提供了一个成员方法来操纵文件和获得一个文件的信息。另外，** File **类还可以对目录和文件进行删除、属性修改等管理工作。**

1，创建一个新的文件对象

可用 File **类的构造方法来生成** File **对象。** File **类的构造方法有：**

- File(String pathname) **通过给定的路径名变换的抽象路径创建文件对象。**
- File(File parent, String child) **从父抽象路径（目录）和子路径字符串创建文件对象。**
- File(String parent, String child) **从父路径和子路径字符串创建文件对象。**

这些构造方法取决于访问文件的方式。例如，若在应用程序里只用一个文件，第一种创建文件的结构是最容易的。但若在同一目录里打开数个文件，则后两种方法更好一些。

2，文件测试和使用

一旦创建了一个文件对象，便可以使用下述方法来获得文件相关信息。

（1）获得文件名。

- public String getName() **得到一个文件名。**
- public String getParent() **得到一个抽象路径的父路径名。**
- public String getPath() **得到一个文件的路径名。**
- public String getAbsolutePath() **得到一个抽象路径的绝对路径名。**

（2）文件重命名。

- public boolean renameTo(File dest) **将抽象路径文件名重命名为给定的新文件名。**

（3）文件删除。

- public boolean delete() **删除抽象路径表示的文件或目录。**

（4）文件测试。

- public boolean exists() **检查抽象路径表示的文件是否存在。**
- public boolean canWrite() **检查抽象路径表示的文件是否可写。**
- public boolean canRead() **检查抽象路径表示的文件是否可读。**
- public boolean isFile() **检查抽象路径表示的文件是否为正常文件（非目录）。**
- public boolean isDirectory() **检查抽象路径表示的是否为目录。**
- public boolean isAbsolute() **检查抽象路径表示的是否为绝对路径。**

（5）获得一般文件信息。

- public long lastModified() **得到抽象路径表示的文件最近一次修改的时间。**
- public long length() **得到抽象路径表示的文件的长度。**

【例 7.13】 获取文件信息。显示所给文件的基本信息，文件名通过命令行参数方式提供。

```
import java.io.*;
public class fileInfo{
  public static void main(String args[]) throws IOException{
    File fileToCheck;
    if(args.length > 0){
      for(int i = 0;i<args.length;i++){
        fileToCheck = new File(args[i]);
        info(fileToCheck);
      }
    } else{
    System.out.println("命令行未给出文件！");
    }
  }
  public static void info (File f) throws IOException {
    System.out.println("绝对路径: "+f.getAbsolutePath());
    System.out.println("路　径: "+f.getPath());
    System.out.println("父路径（目录）: "+f.getParent());
    System.out.println("文件名: "+f.getName());
    if(f.exists()){
      System.out.println("文件存在。");
      System.out.println("是目录？：" + f.isDirectory());
      System.out.println("可读？：" + f.canRead());
      System.out.println("可写？：" + f.canWrite());
      System.out.println("文件长度：" + f.length() + " 字节。");
      System.out.println("文件最后修改时间：" + f.lastModified());
    }else{
      System.out.println("文件不存在！");
    }
  }
}
```

设运行字节码文件的命令行是（本程序可以输入多个命令行参数）：

```
java FileInfo d:\lu\!jiaocai\java 程序\FileInfo.java
```

则程序的运行结果如下：

```
绝对路径: d:\lu\!jiaocai\java 程序\FileInfo.java
路　径: d:\lu\!jiaocai\java 程序\FileInfo.java
父路径（目录）: d:\lu\!jiaocai\java 程序
```

文件名: fileInfo.java
文件存在。
是目录？：false
可读？：true
可写？：true
文件长度：925 **字节。**
文件最后修改时间：1014198308000

注意：程序输出中路径（getPath()**的输出**）**和父路径**（getParent()**的输出**）**两行内容与命令行参数写法有关。若只给出文件名，则路径为文件名，父路径为** null**。**

【例 7.14**】 显示工作目录下的文件名和目录名信息。**

```
import java.io.*;
class DirTree {
   public static void main(String args[]) {
      File dir = new File(System.getProperty("user.dir"));
      if(dir.isDirectory()){
         System.out.println("Directory of "+ dir);
         String listing[]= dir.list();
         for(int i = 0;i<listing.length()&&i<10;i++)
            System.out.println("\t"+listing[i]);
      }
   }
}
```

本程序中，System.getProperty("user.dir")**的作用是获得用户的工作目录，将**"user.dir"**改为**"file.separator"**或**"path.separator"**可获得文件分隔符或路径分隔符。程序的输出结果如下（程序中控制只显示前** 10 **个目录和文件名）：**

```
Directory of D:\lu\!jiaocai\Java 程序
        LinkExample.jar
        ButtonDemoSelf.java
        FontDemo.java
        FontDemo.class
        FontDemo.html
        Drawings.html
        Drawings.java
        ColorString.java
        Demo.java
        Beeper.java
```

*7.5 其他常用流处理

7.5.1 管道流

管道是一种数据流的形式，是线程（见第 10 章）之间传输数据的通道。在 java.io 包中，类 PipedInputStream 和 PipedOutputStream 描述了管道的输入和输出。将一个线程中的管道输出流连接到另一个线程的管道输入流中，就可以通过管道在两个线程中传送数据了。

管道流的构造方法如下：

- PipedInputStream() 创建未连接管道输出流的管道输入流对象。
- PipedInputStream(PipedOutputStream src) 创建连接到管道输出流 src 的管道输入流对象。
- PipedOutputStream()创建未连接管道输入流的管道输出流对象。
- PipedOutputStream(PipedInputStream snk) 创建连接到管道输入流 snk 的管道输出流对象。

由于可能引起系统死锁的原因，在单线程的程序中一般不使用管道操作。在多线程程序中使用管道操作的例子请参看其他资料。

7.5.2 内存的访问

在 Java 语言中，为了保证安全性而禁止直接操作内存，但 Java 语言提供了 ByteArrayInputStream 和 ByteArrayOutputStream 类来利用内存。可将字节数组中的数据视为内存数据进行操作，通过 ByteArrayInputStream 类将数组中的数据以流方式从内存读出，或将数据通过 ByteArrayOutputStream 类以流方式写入内存（数组）中暂时保存。这两个类的构造方法如下：

- public ByteArrayInputStream(byte[] buf)创建一个字节数组输入流，buf 用做缓冲区数组。
- public ByteArrayInputStream(byte[] buf,int offset,int length)创建一个字节数组输入流，buf 用做缓冲区数组，offset 为数组读出开始位置，length 为读出字节数。
- public ByteArrayOutputStream()创建字节数组输出流，缓冲区容量初始化为 32 字节。
- public ByteArrayOutputStream(int size)创建字节数组输出流，缓冲区容量指定为 size 个字节。

【例 7.15】 使用 ByteArrayInputStream 和 ByteArrayOutputStream 类访问内存。

```
import java.io.*;
public class AccessMemory{
   public static void main(String args[]) throws IOException{
      int ch;
      String str = "Creates a ByteArrayInputStream";
      byte bt[] = str.getBytes();
      ByteArrayInputStream bais = new ByteArrayInputStream(bt,8,6);
      System.out.println("合计:" + bais.available());
```

```
            while((ch = bais.read()) != -1)System.out.print((char)ch);
        }
    }
```

程序运行结果如下：

```
合计:6
a Byte
```

7.5.3 顺序流

SequenceInputStream **类可以将两个或多个输入流合并为一个输入流，若要将多个文件的内容一次读到内存中，就可以应用这个类来完成。**SequenceInputStream **类的构造方法为：**

- public SequenceInputStream(Enumeration e)**用枚举类的对象来创建顺序输入流。**
- public SequenceInputStream(InputStream s1, InputStream s2)**用两个输入流来创建顺序输入流。**

【例 7.16**】 用顺序流将两个** java **程序文件** p1.java **和** p2.java **先后显示到屏幕上。**

```
import java.io.*;
public class SequenceInputStreamDemo{
    public static void main(String args[]){
        FileInputStream fis1,fis2;
        try{
            fis1 = new FileInputStream("p1.java");
            fis2 = new FileInputStream("p2.java");
            SequenceInputStream sis = new SequenceInputStream(fis1,fis2);
            int buf = 0;
            while((buf = sis.read())>0)System.out.print((char)buf);
            fis1.close();fis2.close();sis.close();
        }catch(IOException e){
            System.out.println(e);
            System.exit(-1);
        }
    }
}
```

习 题 七

7.1 **在** Java **语言中，输入输出处理需要引入的包是_____，面向字节的输入输出类的基类是_____和_____，面向字符的输入输出类的基类是_____和_____。**

7.2 **下列语句中哪些是正确的语句?**

（1）File f = new File("autoexec.bat");

（2）DataInputStream d = new DataInputStream(System.in);

（3）OutputStreamWriter o = new OutputStreamWriter(System.out);

（4）RandomAccessFile r = new RandomAccessFile("OutFile");

7.3 **下列语句中哪些是正确的语句**?

（1）RandomAccessFile raf=new RandomAccessFile("myfile.txt","rw");

（2）RandomAccessFile raf=new RandomAccessFile(new DataInputStream());

（3）RandomAccessFile raf=new RandomAccessFile("myfile.txt");

（4）RandomAccessFile raf=new RandomAccessFile(new File("myfile.txt"));

7.4 **要读一个较大的文件，下列创建对象的方法中哪个是最合适的方法**?

（1）new FileInputStream("file.name");

（2）new InputStreamReader(new FileInputStream("file.name"));

（3）new BufferedReader(new InputStreamReader(new FileInputStream("file.name")));

（4）new RandomAccessFile raf=new RandomAccessFile("myfile.txt","+rw");

7.5 **下列创建** InputStreamReader **对象的方法中哪些是正确的方法**?

（1）new InputStreamReader(new FileInputStream("data"));

（2）new InputStreamReader(new FileReader("data"));

（3）new InputStreamReader(new BufferedReader("data"));

（4）new InputStreamReader("data");

（5）new InputStreamReader(System.in);

7.6 **过滤流类** FilterOutputStream **是** BufferedOutputStream、DataOutputStream **和** PrintStream **类的父类，下列各类中哪些是** FilterOutputStream **类构造方法有效的参数**?

（1）InputStream （2）OutputStream （3）File （4）RandomAccessFile

7.7 **编程：求** 2～200 **之间的所有素数，将求得的结果保存到** PRIME.DAT **文件中。**

7.8 **编程：检查** C **盘根目录下** CONFIG.SYS **文件是否存在，若存在，则显示该文件的名称和内容；若不存在，则显示相应信息。**

7.9 **编程：输入** 5 **个学生的信息（包含学号、姓名、**3 **科成绩），统计各学生的总分，然后将学生信息和统计结果存入二进制数据文件** STUDENT.DAT **中。**

7.10 **编程：从** 7.9 **题中建立的** STUDENT.DAT **文件中读取数据，寻找平均分最高的学生并输出该生的所有信息。**

7.11 **编程：从** 7.9 **题中建立的** STUDENT.DAT **文件中读取数据，按学生的总分递减排序后，显示前** 3 **个学生的学号、姓名和总分，并将显示的数据存入** STUSORT.DAT **文件中。**

第 8 章　Java Applet

Java Applet 是用 Java 编写的、含有可视化内容并被嵌入 Web 页中由浏览器解释执行的小程序。本章将详细介绍超文本标记语言 HTML 和 Java Applet 及其 AWT 绘图等内容。

8.1　Applet 概述

8.1.1　Java Applet 的特点

Java Applet 是一种特殊的 Java 应用程序。英文后缀-let 是小的意思，Java Applet 即 Java 小应用程序，常简称为 Java 小程序或 Java 小应用。Applet 被嵌入在一个 HTML 文件中，在网上传播，在一个网络浏览器的支持下可下载并运行。Java Applet 运行在一个窗口环境中，提供基本的绘画功能，动画和声音的播放功能，可实现内容丰富多彩的动态页面效果、页面交互功能，实现网络交流能力。

在 Java 应用程序中，必须有一个 main()方法。当程序开始运行时，解释器首先查找 main()方法并执行。而 Applet 则没有 main()方法，它必须嵌入在 HTML 文件中，由支持 Java Applet 的浏览器或 Java SDK 中模拟浏览器环境的 appletviewer.exe 来运行。从某种意义上来说，Applet 有些类似于组件，它实现的功能是不完全的，它必须借助于浏览器中预先设计好的功能和已有的图形界面。Applet 所要做的，是接收浏览器发送给它的消息和事件，并做出及时的反应。另外，为了协调与浏览器的合作过程，Applet 中有一些固定的只能由浏览器在特定时刻和场合调用的方法。

8.1.2　HTML 语言

在第 1 章中已介绍了一个非常简单的 Java Applet 程序，由于 Java Applet 嵌入在 HTML 语言中执行，这里先简单介绍 HTML 语言。

HTML 语言是 HyperText Markup Language 的简写，称为超文本标记语言。它是一种排版语言，在给出具体信息的同时，也用各种标记（tag）来指出这些信息的显示格式。WWW 浏览器可以理解这些标记，并按照标记的要求在浏览器的显示页面中把 HTML 文件中的信息显示出来。

HTML 语言的一个最明显特点是它的标记大都是成对使用，例如，<HTML>和</HTML>、<BODY>和</BODY>等。也有些标记可以单独使用，例如，<CENTER>、<P>等。常用的 HTML 标记及其意义见表 8.1。

表 8.1　常用 HTML 标记及其意义

标　记	说　明
<HTML>..</HTML>	标志整个 HTML 文件的开始和结束
<TITLE>..</TITLE>	括起部分为窗口标题
<HEAD>..</HEAD>	括起部分为窗口头部内容

续表

标　记	说　明
<BODY>…</BODY>	括起部分为页面内容
<I>…</I><B>…</B><S>…</S><U>…</U>	文字斜体，粗体，删除线，下划线显示
<H1>…</H1>至<H6>…</H6>	括起文字用指定标题样式显示
<LEFT>，<RIGHT>，<CENTER>	对齐方式设置为左，右，居中对齐
<FONT COLOR="RED" FACE="Arial" SIZE=3>	设置文本的颜色，字体，大小等
<A HREF="FILENAME.HTM">…</A>	链接到指定的 HTM 文件
<IMG SRC="IMAGENAME.GIF">…</IMG>	显示指定的 GIF 图片文件
<HR LENGTH="n">	显示长度为 n 的水平分割线
<P>	开始一新的段落

【例 8.1】　一个简单 HTML 文件（文件的扩展名为.html 或.htm）。HTML 文件是文本文件，可用任一文本文件编辑器进行编辑，HTML 语言标记名不区分大小写。

```
<HTML>
<HEAD>
<TITLE>Java HTML Test</TITLE>
</HEAD>
<BODY>
<CENTER>
<FONT SIZE="4" COLOR="RED">This is a HTML Test.</FONT>
<HR WIDTH="200"><P>
<A HREF="APPLET.HTM"><I>Java Applet</I></A><P>
<A HREF="APPLICATION.HTM"><I> Java Application</I></A><P>
<HR WIDTH="200"></CENTER>
</BODY></HTML>
```

在 IE 浏览器中显示的结果如图 8.1 所示。

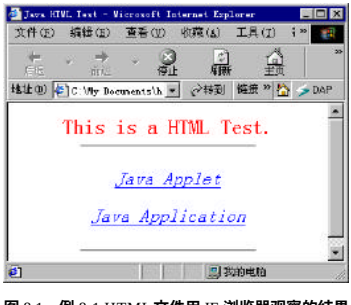

图 8.1　例 8.1 HTML 文件用 IE 浏览器观察的结果

8.1.3 在 HTML 文件中嵌入 Applet

在例 8.1 的 HTML 文件中，未嵌入 Java Applet。将 Java Applet 嵌入网页是通过<APPLET>标记来实现的。在一个 Applet 标记单元中，有如下的标记内容。

```
< APPLET
  [CODEBASE = codebaseURL]
  CODE = AppletFile
  [ALT = alternateText]
  [NAME = AppletInstanceName]
  WIDTH = pixels
  HEIGHT = pixels
  [ALIGN =alignment]
  [VSPACE = pixels]
  [HSPACE = pixels]
>
< PARAM NAME = AppletAttribute VALUE =value>
< PARAM NAME = AppletAttribute VALUE =value>
...
alternateHTML
</APPLET>
```

加方括号的参数是可选的，Applet 标记的参数解释如下。

（1）CODEBASE = codebaseURL

决定 Applet 的 URL 位置和目录，若默认，则使用当前 HTML 页面的路径。例如：

```
CODEBASE = "http://java.sun.com/Applets"
```

（2）CODE = AppletFile

指出 Applet 的 Applet 子类名字。它的路径是相对于 CODEBASE 而言的，不能为绝对路径。例如：

```
CODE = "Clock.class"
```

（3）ALT = alternateText

指明当浏览器不能执行 Applet 时所显示的文字。

（4）NAME = AppletInstanceName

指出 Applet 实例的名称，多用于在多个 Applet 的通信中做标识符。

（5）WIDTH = pixels 和 HEIGHT = pixels

指定 Applet 显示区域的大小，单位为像素点数。例如：

```
WIDTH = 200 HEIGHT = 150
```

（6）ALIGN =alignment

设置 Applet 在页面上的排列对齐方式，有 LEFT、RIGHT、CENTER、TOP、TEXTTOP、

MIDDLE、ABSMIDDLE、BASELINE、BOTTOM、ABSBOTTOM 等多种。例如：

```
ALIGN = CENTER
```

（7）VSPACE = pixels 和 HSPACE = pixels

设置 Applet 与周围文本之间的间距，单位是像素点数。例如：

```
VSPACE = 15 HSPACE = 15
```

（8）PARAM NAME = AppletAttribute VALUE =value

从外界获取参数。PARAM 的属性包括 NAME 和 VALUE 两个，其中 NAME 给出参数名，VALUE 给出参数值。一个 Applet 单元可以包含多个 PARAM 单元。

（9）alternateHTML

标识的文字在不支持 Applet 标记的浏览器中显示，代替 Applet。

例如，要把一个名为 Myapplet.class 的字节码文件嵌入 HTML 文件中，可以采用如下格式：

```
<Applet CODE = Myapplet.class WIDTH = 200 HEIGHT = 50></Applet>
```

注意：HTML 标记名字不分大小写，但值分大小写。

8.2 Applet 的创建和执行

8.2.1 Applet 的类层次及框架结构

1. Applet 类的类层次

所有的 Java Applet 都必须声明为 java.applet.Applet 类的子类或 javax.swing.JApplet 类的子类。通过这个 Applet 类或 JApplet 类的子类，才能完成 Applet 与浏览器的配合。

Applet 类有如下的继承关系。

```
java.lang.Object（Object 类是所有类的根类）
  └java.awt.Component（抽象组件类）
    └java.awt.Container（抽象容器类）
      └java.awt.Panel（非抽象面板类，实现了 Container 所有方法）
        └java.applet.Applet
          └javax.swing.JApplet
```

从类层次可以了解到，Applet 类除了可以拥有自己的方法外，还可以继承它的父类的方法。注意：父类都属于 java.awt 包，Applet 属于 java.applet 包。javax.swing.JApplet 类是 java.applet.Applet 的扩展版，它提供了对基础类库 JFC/Swing 组件体系结构的支持，它属于 javax.swing 包。

Applet 和 JApplet 类为 public 类，编写的 Applet 的主类也必须声明为 public，因此文件名必须与类名相同（包括大小写），源文件名必须为类名加.java。

2，Applet 的框架结构

Applet 的一般结构框架形式如下：

```
import java.awt.*;
import java.applet.*;
public class appletName extends Applet {
  public void init() {// 初始化、设置字体、装载图片等
  }
  public void start() {// 启动或恢复执行
  }
  public void stop() {// 执行被挂起
  }
  public void destroy() {// 执行关闭活动
  }
  public void paint(Graphics g) {// 进行绘图操作等
  }
}
```

在上面列出的各种 Applet 方法中，最基本的是 init()和 paint()方法。

若为继承 JApplet 类的小程序，方法 init()、start()和 stop()等都未变，但在绘图、加入组件等方面与继承 Applet 类的小程序有所不同（见第 9 章）。继承 JApplet 类的小程序应使用下面的引入语句和类声明：

```
import javax.swing.*;
public class appletName extends JApplet { ...}
```

8.2.2 Applet 的执行

1，用 appletviewer 执行

Java SDK 开发环境中提供的小程序观察器 appletviewer.exe 是一个浏览 Applet 的简易工具，它仅仅使用 HTML 文件中需要运行 Applet 的各种信息，而其他内容将不会被显示出来。

appletviewer 通过命令行方式运行，它弹出一个类似浏览器的画面，包括一个 Applet 菜单、一个图形显示区域和一个位于显示区域下方的状态行。appletviewer 运行时要指定一个嵌入 Applet 字节码的 HTML 文件，使得给定的 Applet 可以在显示区域中运行。

2，在浏览器中执行

支持 Java Applet 的浏览器运行嵌入 Applet 单元的 HTML 文档即可运行其中的 Applet 程序，但常见的浏览器只能支持 JDK 1.1 版的 Java Applet。若需要使用新版本的 Java SDK 来开发浏览器可运行 Java Applet，可安装 Java SDK 相应版本的插件，并需用 HtmlConverter.exe 程序来变换原 HTML 文档为新的 HTML 文档格式。

3，在网页编辑器 FrontPage 2000 中执行

Microsoft FrontPage 2000 是一种常用的网页设计工具，它集成在 Microsoft Office 2000 中。使用 FrontPage 2000 可以用所见即所得的方式进行网页设计，且不需要网页设计者写任何 HTML 代码。若正在用 FrontPage 2000 进行网页设计，其中需要 Java Applet 程序的功能，可在网页编辑器中直接插入 Java Applet（用 JDK 1.1 版开发）。下面以 Microsoft FrontPage 2000 为例，说明在网页设计中应用 Java Applet 的方法。

（1）启动 FrontPage。

（2）选择“插入”菜单下“高级”菜单项的级联菜单，在其中选择“Java 小程序”命令，将出现一个名为“Java 小程序属性”的对话框。见图 8.2。

（3）在对话框中填入信息：在“Java 小程序源”文本框中输入 Java Applet 程序名（大小写要一致），例如 Clock.class。在“含有 Java 小程序的 URL”文本框中输入 Java Applet 程序的路径（若在与网页相同文件夹中，可不写路径）。根据需要，在“浏览器不支持 Java 时显示的消息”文本框和“Java 小程序参数”列表框中输入相应信息（可用“添加”、“修改”和“删除”命令按钮进行操作）。在“布局”和“大小”相应文本框中设置信息。

（4）选择“确定”按钮。设计页面上将出现一个 Java Applet 程序的图标。

（5）选择“预览”标签或选择“在浏览器中浏览”按钮，运行 Java Applet。

图 8.2　FrontPage 2000 中的“Java 小程序属性”对话框

8.2.3　Applet 的主要方法及生命周期

在浏览器中运行 Applet 程序，从运行开始到运行结束，Applet 程序表现为一些不同的行为，例如，初始化、绘图、退出等。每一种行为都对应一个相关的方法，在 Java Applet 中有五种相对重要的方法：初始化 init()、开始执行 start()、停止执行 stop()、退出 destroy()、绘画 paint()。前四种方法分别对应 Applet 从初始化、启动、暂停到消亡的生命周期的各个阶段。

1，public void init()初始化

在整个 Applet 生命周期中，初始化只进行一次。当第一次浏览含有 Applet 的 Web 页时，浏览器将进行下面的工作：

（1）下载该 Applet。

（2）创建一个该 Applet 主类的实例对象。

（3）调用 init()对 Applet 自身进行初始化。

在 init()方法中可设置 Applet 初始状态、载入图形或字体、获取 HTML 中 Applet 标记单元中<PARAM>设定的参数等。

2，public void start()启动

在整个 Applet 生命周期中，启动可发生多次。在下列情况下，浏览器会调用 start()方法：

（1）Applet 第一次载入时。

（2）离开该 Web 页后，再次进入时（用 back,forward）。

（3）reload 该页面时。

（4）在浏览含有 Applet 的 Web 页时用浏览器右上角缩放按钮缩放浏览窗口大小时。

在 start()方法中可启动一个线程来控制 Applet，给引入类对象发送消息，或以某种方式通知 Applet 开始运行。

3，public void stop()停止执行

在整个 Applet 生命周期中，停止执行可发生多次。 在下列情况下，浏览器会调用 stop()方法：

（1）离开 Applet 所在 Web 页时（用 back,forward）。

（2）Reload 该页面时。

（3）在浏览含有 Applet 的 Web 页时用浏览器右上角缩放按钮缩放浏览窗口大小时。

（4）close 该 Web 页（彻底结束对该页面的访问），exit 结束浏览器运行时（从含有该 Applet 的 Web 页退出时）。

stop()挂起 Applet，可释放系统处理资源，否则当浏览者离开一个页面时，Applet 还将继续运行。

4，public void paint(Graphics g)绘制

发生下列情况时，浏览器会调用 paint()方法，而且可根据需要产生多次调用。

（1）Web 页中含有 Applet 的部分被卷入窗口时。

（2）Applet 显示区域在视线内时调整浏览窗口大小、缩放浏览窗口、移动窗口或 reload 等需要重绘窗口时都会调用 paint()方法。

与前几个方法不同的是，paint()中带有一个参数 Graphics g，它表明 paint()需要引用一个 Graphics 类的实例对象。

在 Applet 中不用编程者操心，浏览器会自动创建 Graphics 对象并将其传送给 paint()方法。但编程者应在 Applet 中引入 Graphics 类所在的包或该类：

```
import java.awt.Graphics;
```

5. public void destroy()退出（撤销）

在整个 Applet 生命周期中，退出只发生一次。在彻底结束对该 Web 页的访问和结束浏览器运行时（close exit）调用一次。

destroy()是 java.applet.Applet 类中定义的方法，只能用于 Applet。可在该方法中编写释放系统资源的代码。但除非用了特殊的资源如创建的线程，否则不需重写 destroy()方法，因为 Java 运行系统本身会自动进行“垃圾”处理和内存管理。

【例 8.2】 Applet 的方法调用。

```
import java.awt.*;
import java.applet.Applet;
public class AppletLife extends Applet{
   public static int colors=10;
   Font font;
   public void init(){
      System.out.println("Now init");
      font = new java.awt.Font("TimesRoman", Font.PLAIN, 36);
   }
   public void start(){
      System.out.println("Now start");
   }
   public void stop(){
      System.out.println("Now stop");
   }
   public void paint(Graphics g){
      int red = (int)(Math.random() * 50);
      int green = (int)(Math.random() * 50);
      int blue = (int)(Math.random() * 256);
      g.setFont(font);
      g.setColor(new Color((red + colors * 30) % 256,
            (green + colors / 3) % 256, blue));
      colors = colors + 10;
      System.out.println("Now paint");
      g.drawString("hello",30,30);
   }
   public void destroy(){
      System.out.println("Now destroy");
   }
}
```

程序运行时，在 Applet 显示区域显示 Hello（重画时再改变颜色），在后台窗口中显示各个方法中 println()输出的内容，可以试着进行最小化 Applet 窗口、切换窗口等操作，可以看

到后台窗口中 Applet 各方法执行的顺序和时机。

除了在上述框架结构中给出的方法外，还有 repaint()方法来重画 Applet，showStatus()方法在 Applet 显示区域的下方状态行显示给定的字符串等。

8.2.4 Applet 和 Application

由于程序结构不同的原因，在前面的讨论中，Java 小程序 Applet 和 Java 应用程序 Application 是分别介绍的，实际上，可以编写出既可作为小程序，又可作为应用程序的 Java 程序，这样的程序可以独立地在操作系统下运行，又可在浏览器中运行。下面是一个简单的例子。

【例 8.3】 既可作为 Applet，又可作为 Application 运行的程序。

```
import javax.swing.*;
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
public class App2 extends Applet{
   public static void main(String args[]){
      JFrame frame=new JFrame("Application");
      App2 app = new App2();
      frame.getContentPane().add(app,BorderLayout.CENTER);
      frame.setSize(150,100);
      frame.setVisible(true);
      frame.addWindowListener(new WindowControl(app));
      app.init();
      app.start();
   }
   public void paint(Graphics g){
      g.drawString("Hello,World!",25,25);
      g.drawRect(20,10,80,20);
   }
   public void destroy(){
      System.exit(0);
   }
}
class WindowControl extends WindowAdapter{
   Applet c;
   public WindowControl(Applet c){
      this.c=c;
   }
   public void WindowControl(WindowEvent e){
```

```
        c.destroy();
    }
}
```

程序作为 Application 运行时的结果如图 8.3 所示，作为 Applet 运行时的结果如图 8.4 所示。

图 8.3　例 8.3 运行结果（一）

图 8.4　例 8.3 运行结果（二）

8.2.5　HTML 参数传送

与 Application 可从命令行获得系统传送的参数的情况类似，Applet 则可以通过 HTML 文件来得到外部参数，这是通过 HTML 文档中 PARAM 参数标记来实现的。PARAM 部分的 name 是参数名，而 value 是参数的值，它是字符串类型。在 Applet 中，通过方法：

```
public String getParameter(String name)
```

可返回 HTML 中 PARAM 参数标记参数名为 name 的参数的值。

例如，若 Applet 被指定为使用的 HTML 文件为：

```
<Applet code = "Clock.class" width = 50 height = 50>
<param name = Color value = "blue">
</Applet>
```

那么，调用 getParameter("Color")则返回值“blue”。

【例 8.4】　Applet 运行时从 HTML 文件的 Applet 单元获取参数，参数及其相应的值见程序中的注释部分。

```
/*
<Applet code="ParamDemo" width=300 height=80>
<param name=fontName value=Courier>
<param name=fontSize value=14>
<param name=leading value=2>
<param name=accountEnabled value=true>
</Applet>
*/
import java.awt.*;
import javax.swing.*;
public class ParamDemo extends JApplet{
   String fontName;
   int fontSize;
```

```
    float leading;
    boolean active;
    public void start() {
      String param;
      fontName = getParameter("fontName");
      if(fontName == null)fontName = "Not Found";
      param = getParameter("fontSize");
      try {
        if(param != null) // 如果未找到
          fontSize = Integer.parseInt(param);
        else
          fontSize = 0;
      } catch(NumberFormatException e) {
        fontSize = -1;
      }
      param = getParameter("leading");
        try {
          if(param != null) // 如果未找到
            leading = Float.valueOf(param).floatValue();
          else
            leading = 0;
        } catch(NumberFormatException e) {
          leading = -1;
        }
        param = getParameter("accountEnabled");
        if(param != null)active = Boolean.valueOf(param).booleanValue();
      }
      public void paint(Graphics g) {     // 显示参数
        g.drawString("Font name: " + fontName, 10, 20);
        g.drawString("Font size: " + fontSize, 10, 35);
        g.drawString("Leading: " + leading, 10, 50);
        g.drawString("Account Active: " + active, 10, 65);
      }
    }
```

由于 getParameter()**方法返回字符串，若希望得到数值参数，则需要进行类型转换。程序运行结果如图** 8.5 **所示。**

图 8.5　**例** 8.4 **运行结果**

8.3　Applet **的** AWT **绘制**

AWT **是** Abstract Window Toolkit **的缩写，中文意义是抽象窗口工具包。所谓抽象，是指**

Java 作为一种跨平台的语言，要求 Java 程序应能在不同的平台系统上运行。为了达到这个目的，集中了很多绘图、图形图像、颜色、字体、图形界面组件等类的 AWT 类库中的各种操作被定义成在一个并不存在的“抽象窗口”中运行。“抽象窗口”使得开发人员所设计的图形界面程序能够独立于具体的界面实现，可以适用于所有的平台系统。

因为 Applet 本身就是一个图形界面的程序，应用 AWT 类库中的类能较容易地进行图形界面的设计。在这里，先介绍字体、颜色、绘图等基本的图形界面元素的使用，其他内容在以后各章中再陆续介绍。

8.3.1 AWT 绘制基础

要进行 AWT 绘制工作，要应用 java.awt.Component 类中的几个方法，这些方法在绘图过程中作用各不相同。

1. void paint(Graphics g)方法

在本方法中进行绘图的具体操作，需编写绘图的程序段。在第一次显示或显示图形被破坏需修复时，paint()方法被自动调用。paint()方法需要一个 Graphics 类的参数 g，可认为它代表了 Applet 显示区域的背景，字形、颜色、绘图都针对这个对象进行。参数 g 由系统自动提供，不能自己创建 Graphics 类的对象，但有时在需要 Graphics 对象时，可通过 getGraphics()方法获得一个 Graphics 对象：

```
Graphics g = getGraphics();
```

2. void update(Graphics g)方法

本方法用于更新图形。它首先清除背景，然后设置前景，再调用 paint()方法完成组件中的具体绘图。

3. void repaint()方法

本方法用于重绘图形。在组件外形发生变化（如调整窗口大小）时，repaint()方法立即被系统自动调用。它调用 update()方法以实现对组件的更新。repaint()方法有几种重载的方法，分别调用不同的 repaint()方法，可实现对组件的局部重绘、延时重绘等功能。

8.3.2 应用字体

字体是各种字母和符号的大小和外观的完整集合。字体定义了字符的外观、大小和字体类型（类型包括粗体、斜体或者普通体，字体类型也称为字形、字体风格等）。

字体是通过字的轮廓来创建的。字的轮廓是个位图，它定义了字符或者符号的外观。同一类字体家族具有类似的外观，因为它们是通过同一种字形轮廓的集合创建的。同样，不同的字体家族使用不同的轮廓集合，它们的外观就有明显的区别。

在 Java 语言中，应用字体用 Font 类实现，一个 Font 类的对象表示了一种字体显示效果，包括字体、字形和字号等内容。Font 类的构造方法为：

```
Font(String name,int style,int size)
```

其中，字体名 name 有 Courier、Dialog、Helvetica、Monospaced、SansSerif、Serif、TimesRoman 等。字形 style 指的是字的外观，有三个字形的静态变量：Font.PLAIN（正常）、Font.BOLD（粗体）、Font.ITALIC（斜体）。粗体和斜体可组合为粗斜体：Font.BOLD+Font.ITALIC。字体大小 size 以像素点数来度量，一个像素点（point）是 1/72 英寸。

例如，下面的语句可创建一个 Font 类的对象：

```
Font myFont = new Font("TimesRoman",Font.BOLD,18);
```

myFont 被设置为 TimesRoman 字体的 18 点粗体字。

若希望使用该 Font 对象，可以使用 Graphics 类或需要设置字体的组件类的 setFont()方法，该方法使用 Font 对象作为参数。例如，对于 Graphics 类对象 g：

```
g.setFont(myFont);
```

另外，与 setFont()方法对应的方法是 getFont()方法，它的作用是获得当前所用的字体。其他常用的 Font 类方法有：

- public int getStyle()　获得当前字形。
- public int getSize()　获得当前字体大小。
- public String getName()　获得当前字体名称。
- public String getFamily()　获得当前字体家族名称。
- public boolean isPlain()　测试当前字体是否为正常字形。
- public boolean isBold()　测试当前字体是否为粗体。
- public boolean isItalic()　测试当前字体是否为斜体。

【例 8.5】　字体设置和显示效果。

```
import javax.swing.JApplet;
import java.awt.*;
public class FontDemo extends JApplet {
   public void paint(Graphics g) {
      int baseline = 60;            // 设置写字的基线
      g.setColor(Color.black);   // 设置写字的颜色为黑色
      g.drawLine(0, baseline, 200, baseline);  // 绘出基线
      g.setFont(new Font("Serif", Font.BOLD+Font.ITALIC, 36));   //设置字体字形字号
      g.drawString("中国 Wxyz", 10, baseline);  // 输出字符串
   }
}
```

程序运行结果如图 8.6 所示。

图 8.6　例 8.5 运行结果

图中说明了基线（baseline，图中文字下方的一横线）的意义：基线是输出大写字母下边的对齐线。drawString()等方法中输出字符串的坐标都是以基线为基准（Y 坐标）输出的。

8.3.3　应用颜色

若希望程序的界面五彩斑斓，可应用 java.awt 包中的类 Color。Color 类提供了 13 种颜色

常量，多种创建颜色对象的构造方法以及多种获取和设置颜色的方法。Java 采用 24 位颜色标准，每种颜色由红（R）、绿（G）、蓝（B）三种颜色的不同比例值组合而成，RGB 的取值范围为 0～255。理论上可组合成 1600 万种以上的颜色。实际上要考虑设备的限制和需要。

Color 类的 13 种 public final static 的颜色常量见表 8.2。注意：颜色常量大小写均可（表中只给出小写的颜色常量）。

表 8.2　Color 类的 13 种颜色常量

颜色常量	颜　色	RGB 值
Color.black	黑色	0,0,0
Color.blue	蓝色	0,0,255
Color.green	绿色	0,255,0
Color.cyan	蓝绿色（青色）	0,255,255
Color.darkGray	深灰色	64,64,64
Color.gray	灰色	128,128,128
Color.lightGray	浅灰色	192,192,192
Color.red	红色	255,0,0
Color.magenta	洋红色	255,0,255
Color.pink	粉红色	255,175,175
Color.orange	橙色	255,200,0
Color.yellow	黄色	255,255,0
Color.white	白色	255,255,255

Color 类常用的构造方法如下：

- public Color(int r,int g,int b)
- public Color(float r,float g,float b)
- public Color(int rgb)

其中，int 类型的 r、g、b 分别表示红、绿、蓝的含量，取值范围是 0～255。float 类型的 r、g、b 取值范围是 0.0～1.0。int 类型的 rgb 值按二进制位来分配红、绿、蓝的含量，16～23 位是红的含量，8～15 位是绿的含量，0～7 位是蓝的含量。

Color 类的颜色常量和实例对象可提供给需要设置颜色的对象方法，来改变当前的颜色。例如，在 Graphics 类中，可用下面的方法来改变当前颜色：

```
public void setColor(Color c);
```

设 g 是一个 Graphics 的对象，设置 g 当前颜色为红色的方法可以为下列方法之一。

```
g.setColor(Color.red);                  // 使用 Color 常量
g.setColor(new Color(255,0,0));         // 使用 Color 对象
```

对于 GUI 的组件，可用与颜色有关的下列方法分别设置和获取组件的背景色和前景色：

- public void setBackground(Color c);
- public Color getBackground();
- public void setForeground(Color c);
- public Color getForeground();

若需要颜色的分量值，可用 Graphics 类的方法 getColor()或 Color 类的如下方法：

- public int getRed();
- public int getGreen();
- public int getBlue();

【例 8.6】　绘制有颜色的字符串。

```
import javax.swing.JApplet;
import java.awt.*;
public class ColorString extends JApplet{
   private int red,green,blue;
   private float rf,gf,bf;
   private int ci;
   String str;
   public void init(){
      red = 200;green = 20; blue = 200;
      rf = 0.6f;gf = 0.4f;bf = 0.2f;
      ci = 0x00ff0000;     // 设置为红色
      str = "Hello,Java Programmer.";
   }
   public void paint(Graphics g){
      Color c = new Color(red,green,blue);
      g.setColor(c);
      g.drawString(str,30,30);
      showStatus("当前颜色：" + g.getColor());
      g.setColor(new Color(rf,gf,bf));
      g.drawString(str,30,50);
      g.setColor(new Color(ci));
      g.drawString(str,30,70);
   }
}
```

程序运行结果如图 8.7 所示。

图 8.7　例 8.6 运行结果

8.3.4　绘制图形

Java 语言的类 Graphics 包含各种绘图方法，用于绘制直线、矩形、多边形、圆和椭圆等图形和进行简单的图形处理。

绘图时，总假设坐标原点在图的左上角，坐标为（0,0）。沿 X 轴水平向右方向为正方向。沿 Y 轴垂直向下为正方向，度量单位为像素点个数。

绘图都是用 Graphics 类的对象来完成的。在 Applet 中，Graphics 对象自动产生，并作为参数传送给 update()或 paint()方法。

1. 写字符串

- public void drawString(string str,int x,int y)在基线（x,y）坐标处写字符串 str。
- public void drawChars(char[] data,int offset,int length,int x,int y)在基线（x,y）坐标处写开始偏移为 offset、长度 length 的字符数组 data 内容。
- public void drawBytes(byte[] data,int offset,int length,int x,int y)在基线（x,y）坐标处写开始偏移为 offset、长度 length 的字节数组 data 内容。

2，画直线

- drawLine(x1,y1,x2,y2)从点（x1,y1）到（x2,y2）画一条直线。

3. 画矩形

- drawRect(x,y,width,height)以（x,y）为原点，即矩形的左上角，画宽为 width，高为 height 的矩形。
- fillRect(x,y,width,height)画实心的矩形，参数同上。
- drawRoundRect(x,y,width,height,x',y')画圆角矩形。前四个参数与上面的 drawRect()方法参数相同，x' 和 y' 指明了正好包含角的圆弧的矩形的宽和高。
- fillRoundRect(x,y,width,height,x',y')画实心的圆角矩形，参数同上。
- public void draw3DRect(int x,int y,int width,int height, boolean raised)画立体感矩形。前四个参数同 drawRect()方法。boolean 取值为 true 或 false，表明该矩形是从平面突起的还是凹陷的。
- fill3Drect(x,y,width,height,boolean)画实心的立体感矩形，参数同上。

4，画椭圆

- public abstract void drawOval(int x,int y,int width,int height)画以（x,y）为原点，即矩形的左上角，宽为 width，高为 height 的矩形的内切椭圆。
- fillOval(x,y,width,height)画实心的椭圆，参数同上。

5. 画圆弧

- public abstract void drawArc(int x,int y,int width,int height,int startAngle,int arcAngle)考虑以（x,y）为原点，宽为 width，高为 height 的矩形内切椭圆。画它上面从 startAngle 角度开始，掠过 sweepAngle 角度的线段。
- fillArc(x,y,width,height,startAngle,arcAngle)实心，参数同上。

6，画多边形

- public abstract void drawPolygon(int[] xPoints,int[] yPoints,int nPoints) xPoint 和 yPoint 是两个数组，分别定义多边形顶点的 *x* 坐标和 *y* 坐标。nPoint 为顶点个数。
- fillPolygon(xPoint,yPoint,nPoint)实心的，参数同上。

7．限定作图区域

▪ public abstract void clipRect(int x,int y,int width,int height)这四个参数划定了一个矩形区，使得所有的绘图操作只能在这个矩形区域内起作用，超出范围则无效。

在使用这些方法绘图时，一般将正确的参数提供给这些方法就能绘出所需图形。下面给出一个综合了这些方法的一个例子。

【例 8.7】 绘制各种图形。

```
import java.awt.*;
import javax.swing.JApplet;
public class DrawFigures extends JApplet{
  public void paint(Graphics g){
    g.drawLine(30,5,40,150);
    g.drawRect(40,10,50,20);
    g.fillRect(60,30,70,40);
    g.drawRoundRect(110,10,130,50,30,30);
    g.drawOval(150,120,70,40);
    g.fillOval(190,160,50,40);
    g.drawOval(90,100,50,40);
    g.fillOval(130,100,50,40);
    drawMyPolygon(g);
    g.drawString("这些图形由 Graphics 类的方法绘出。",40,220);
  }
  public void drawMyPolygon(Graphics g){
    int x[] = { 30,50,65,110,120};
    int y[] = { 100,140,120,170,200};
    g.drawPolygon(x,y,5);
  }
}
```

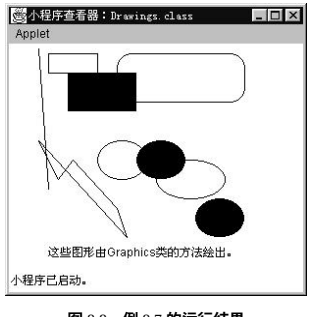

图 8.8 例 8.7 的运行结果

程序运行结果如图 8.8 所示。

8.4 Applet 的通信

8.4.1 同页 Applet 间的通信

嵌入在同一个 HTML 文件中的 Applet 程序（同页）可以通过 java.applet 包中提供的接口、类方法进行通信。有用的接口为 Applet 环境上下文接口 AppletContext，在该接口中定义了如下的方法：

▪ public Applet getApplet(String name) 查找和返回由 Applet 上下文指明的 HTML 文档

中具有指定 name 的 Applet，name 用 HTML 标记中 name 属性设置，是一个 Applet 字节码文件的名字。

- public Enumeration getApplets()查找由 Applet 上下文指明的文档中所有的 Applet。它返回的是枚举 Enumeration 对象，该对象枚举了页中所有的 Applet，这时可通过接口 Enumeration 提供的方法 hasMoreElements()和 nextElement()来获得同页中所有的 Applet 对象。
- public boolean hasMoreElements() 测试该枚举对象是否包含更多的元素。
- public Object nextElement()返回该枚举对象的下一个元素。若无更多的元素存在，则抛出异常“NoSuchElementException”。

【例 8.8】 使用 getApplet()方法获取同页 Applet 信息，显示获得的 Applet 类名。

```
import java.awt.*;
import javax.swing.*;
public class GetAppletDemo extends JApplet{
   JApplet ap ;
   public void paint(Graphics g){
      ap = (JApplet)getAppletContext().getApplet("绘图");
      g.drawString("--"+ap.getClass().getName(),30,30);
   }
}
```

为运行这个 Applet，编写一个 HTML 文件如下：

```
<Html>
<Head></Head><Body>
<Applet code = GetAppletDemo.class width = 400 height = 400>
</Applet></Body>
<Applet code = Drawings.class width = 400 height = 400 name = "绘图">
</Applet></Body>
</Html>
```

图 8.9 例 8.8 运行结果

将本例的 Applet 和 HTML 文件中指明的另一个 Applet 类文件存放到 HTML 文件所在目录，用 appletviewer 运行 HTML 文件，就可见到两个 Applet 运行窗口，其中本程序的 Applet 运行窗口中即显示了另一个 Applet 的类名。运行显示情况如图 8.9 所示。

若需要同页所有 Applet 类文件的信息，可调用 getApplets()方法。

【例 8.9】 使用 getApplets()方法获取同页所有 Applet 信息，显示获得的 Applet 类名。

```
import java.awt.*;
import javax.swing.*;
import java.util.Enumeration;
public class GetAppletsDemo extends JApplet{
   JApplet ap;
```

```
    public void paint(Graphics g){
      int i = 0;
      Enumeration e = getAppletContext().getApplets();
      while(e.hasMoreElements()){
        ap= (JApplet)e.nextElement();
        i = i + 20;
        g.drawString("--"+ap.getClass().getName(),30,i);
      }
    }
}
```

图 8.10　例 8.9 运行结果

程序运行结果如图 8.10 所示。

8.4.2　Applet 与浏览器之间的通信

在 Applet 类中提供了多种方法，使之可以与浏览器进行通信。前面介绍的 Applet 从 HTML 文件获得参数，实际上也是一种与浏览器之间的通信。下面再介绍一些 Applet 类中与浏览器通信的方法。

- public URL getCodeBase()得到 Applet 自身的 URL 地址。URL 的概念在下一节中介绍。
- public URL getDocumentBase()返回嵌入 Applet 的 HTML 文档的绝对 URL 地址。例如，设 Applet 包含在下述文档中：

```
http://java.sun.com/products/jdk/1.4/index.html
```

则 HTML 文档的 URL 地址为：

```
http://java.sun.com/products/jdk/1.4/
```

- public String[][] getParameterInfo()返回本 Applet 参数的信息，返回值为字符串数组，它的每个元素是包含三个字符串的一维数组，三个字符串分别为名称、类型和描述。
- public String getAppletInfo()返回本 Applet 的作者、版本、版权等信息。

【例 8.10】　使用 getCodeBase()和 getDocumentBase()方法来获得 Applet 程序所在路径和 HTML 文档名。

```
import java.awt.*;
import java.applet.*;
import java.net.*;
public class Bases extends  Applet {
  public void paint(Graphics g) {
    String msg;
    URL url = getCodeBase();
    msg = "Code base: " + url.toString();
    g.drawString(msg, 10, 20);
    url = getDocumentBase();
    msg = "Document base: " + url.toString();
```

```
        g.drawString(msg, 10, 40);
    }
}
```

程序运行结果如图 8.11 所示。

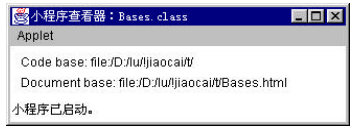

图 8.11　例 8.10 运行结果

另外，在 AppletContext 接口中提供了方法 showDocument()，来请求浏览器显示一个 URL 地址所对应的 HTML 文件。

8.5　Applet 的应用

8.5.1　访问 WWW 资源

Applet 程序经常有访问 WWW 资源的需要，下面简单介绍在 Applet 程序中如何访问 WWW 资源及其相关的一些概念。

1．URL 统一资源定位器

URL（Uniform Resource Locator）是统一资源定位器的简称，它表示 Internet 上某一资源的地址。URL 包括两方面的内容：协议名和资源名，中间用冒号隔开。例如：

```
http://www.wuhan.net.cn
```

其中，协议名指明资源所使用的传输协议，如 http、ftp 等，资源名指明资源的地址，包括主机名、端口号、文件名等。有时 URL 中也包含资源文件名，例如：

```
http://www.wuhan.net.cn/index.html
```

2，URL 类

为了表示 URL，在 java.net 包中提供了类 URL 来支持 URL。类 URL 有六种构造方法，下面是一些常用的构造方法，它们可用来初始化一个 URL 对象。

- URL(String spec)从字符串 spec 创建一个 URL 对象。
- URL(String protocol, String host, int port, String file)从指定的协议、主机、端口号和文件创建一个 URL 对象。
- URL(String protocol, String host, String file)从指定的协议、主机和文件创建一个 URL 对象。

3，获取 URL 对象的属性

一个 URL 对象生成后，其属性是不能改变的，但可以通过类 URL 的方法来获取这些属性。

- Object getContent(Class[] classes) 返回 URL 对象的内容。
- String getFile() 返回 URL 对象的文件名。
- String getHost() 返回 URL 对象的主机名。
- String getPath() 返回 URL 对象的路径。
- int getPort() 返回 URL 对象的端口号。若端口未设置，则返回-1。
- String getProtocol()返回 URL 对象的协议名。

有些协议的 URL 地址并不完全具备这些属性，但是，类 URL 仍然提供了这些方法，这是因为类 URL 是基于 http 协议的，而 http 协议下的 URL 包含上述的这些属性。

8.5.2 访问网络资源

1，IP 地址（Internet Protocol）

IP 地址是网络上的通信地址，是计算机、服务器、路由器的端口地址，每一个 IP 地址在全球是惟一的，是运行 TCP/IP 协议的基础。

IP 地址 4 字节长，每个字节对应一个小于 256 的十进制数，字节之间用点号分隔。例如，上海热线的 IP 地址是 202.96.209.5 和 202.96.209.133 等。

IP 地址分为两部分内容：一部分为网络标识，另一部分为主机标识。根据第 1 个字节的数字范围不同，可分为 A～E 五类 IP 地址。

从用户上网的地址角度考虑，IP 又可分为动态地址和静态地址两类。动态地址是指一台计算机与 Internet 连接，成为一台 Internet 上的主机，但每次连接得到的 IP 地址不一定是相同的，这由当时连接的网络服务器的情况而定。这对一般用户来讲，是无关紧要的。有时需要知道本机的 IP 地址或连接的对方计算机的 IP 地址，可使用 Java 语言的 InetAddress 类。

2，InetAddress 类

java.net.InetAddress 类继承于 Object 类，可用于描述一个 IP 地址。常用的方法有：

- public static InetAddress getByName(String host)throws UnknownHostException 确定所给主机名的主机 IP 地址。
- public static InetAddress getLocalHost()throws UnknownHostException 返回本地主机和 IP 地址。

【例 8.11】 获得本机主机名和 IP 地址（本例是拨号上网的结果）。

```
import javax.swing.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
public class LocalHost extends JApplet{
   InetAddress HostAddr = null;
```

```
    public void start(){
      try{
        HostAddr = InetAddress.getLocalHost();
      }catch(UnknownHostException e){
        System.err.println(e.getMessage());
      }
      repaint();
    }
    public void paint(Graphics g){
      g.drawString("Host name / ip = " + HostAddr.toString(),10,30);
    }
  }
```

程序运行结果如图 8.12 **所示。**

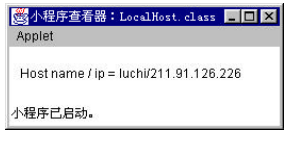

图 8.12　**例** 8.11 **运行结果**

习　题　八

8.1　**在** Applet **的几种常用方法中，哪些方法只运行一次？哪些方法可运行多次？**

8.2　**最小化** Applet **运行窗口时，将运行什么方法？从最小化** Applet **窗口图标恢复到原窗口大小或最大化，将运行什么方法？**

8.3　Applet **中** paint()**方法、** repaint()**方法和** update()**方法三者之间有什么区别和联系？**

8.4　**编写** Applet **在不同行上显示自己姓名、性别、家庭地址的各种颜色的字符串。**

8.5　**编写** Applet **显示字符串，字符串及其显示位置、字体、字形、大小和颜色通过** HTML **文件中的** PARAM **参数来传送。**

8.6　**编写** Applet **画多个嵌套的方框，对它们赋以不同的特色。**

8.7　**编写** Applet **绘出** $N \times N$ **的方格，在其中放入整数** 1 **至** N **的平方，使得方格的横向、纵向、两条对角线上的数字之和都相同。一种算法见习题** 4.7**。这里** N **是一个奇数，从** HTML **文件中的** PARAM **参数传送。**

8.8　**编写** Applet**，用绘画的方法画出自己的姓名。**

8.9　**如何将一个圆半径** radius **的值传送给一个** Java Applet**？选择一个答案。**

（1**）**public void init() {
String s = getParameter("radius"); doSomethingWithRadius(s); }

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

String s = args[0]; DoSomethingWithRadius(s); }

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

String s = getParameter("radius"); DoSomethingWithRadius(s); }

（4）public void init() {

int radius = getParameter("radius"); doSomethingWithRadius(radius); }

（5）public void init() {

int radius = getParameter(); doSomethingWithRadius(radius); }

第 9 章　图形用户界面(GUI)设计

对一个优秀的应用程序来说，良好的图形用户接口是必不可少的。缺少良好的图形用户接口，将会给用户理解和使用应用程序带来很多不便。Java 的抽象窗口工具集 AWT 和组件集 Swing 中包含了很多的类来支持 GUI 的设计。

9.1　图形用户界面设计概述

对使用过 Windows 操作系统的人员来说，GUI 程序是再熟悉不过的了。在 GUI 应用程序中，各种 GUI 元素有机结合在一起，它们不但提供漂亮的外观，而且提供了与用户交互的各种手段。在 Java 语言中，这些元素主要通过 java.awt 包和 javax.swing 包中的类来进行控制和操作。

9.1.1　GUI 支持包和简单 GUI 程序例

1，java.awt 包

Java 语言在 java.awt 包中提供了大量的进行 GUI 设计所使用的类和接口，包括绘制图形、设置字体和颜色、控制组件、处理事件等内容，AWT 是 Java 语言进行 GUI 程序设计的基础。

java.awt 包中主要的类及其层次关系如下所示。

```
java.lang.Object        java 所有类的超类
  ├ Font                字体类
  ├ Color               颜色类
  ├ Graphics            几何绘图类
  ├ Component           组件类
  │   ├ Label           标签类
  │   ├ Button          按钮类
  │   ├ TextComponent   文本组件类
  │   │   ├ TextField   单行文本框类
  │   │   └ TextArea    多行文本框类
  │   ├ List            列表类
  │   ├ Container       容器类
  │   │   ├ Panel       面板类
  │   │   │   └ Applet  小程序类
  │   │   └ Window      窗口类
  │   │       ├ Frame   框架类
  │   │       └ Dialog  对话框类
  │   └ Checkbox        单选按钮与复选按钮类
```

```
├ CheckboxGroup        按钮组合类
├ MenuComponent        菜单组件类
|   ├ MenuBar          菜单条类
|   └ MenuItem         菜单项类
├ FlowLayout           流式布局管理类
└ BorderLayout         边界布局管理类
```

在第 8 章中已经介绍了 java.awt 包中设置字体和颜色、进行几何绘图等类的使用。本章将介绍 AWT 布局管理器、鼠标事件处理、键盘事件处理等方面的内容。

2，javax.swing 包

Swing 包是 Java 基础类库（Java Foundation Classes——JFC）的一部分。Swing 提供了从按钮到可分拆面板和表格的所有组件。Swing 组件是 Java 语言提供的第二代 GUI 设计工具包，它以 AWT 为基础，在 AWT 内容的基础上新增或改进了一些 GUI 组件，使得 GUI 程序功能更强大，设计更容易、更方便。“Swing”是开发新组件的项目代码名，现在，这个名字常用来引用新组件和相关的 API。

Swing 包首先出现在 JDK 1.1 中。以前的版本中使用 AWT 组件，在新的 Java 版本中仍然支持 AWT 组件，但几乎所有的 AWT 组件都有对应的新的、功能更强的 Swing 组件，所以现在开发 GUI 程序时，一般建议用 Swing 组件代替 AWT 组件。AWT 组件和对应的 Swing 组件，从名称上很容易记忆和区别。例如，AWT 的框架类、面板类、按钮类和菜单类，被命名为 Frame、Panel、Button 和 Menu，而 Swing 对应的组件类被命名为 JFrame、JPanel、JButton 和 JMenu。与 AWT 组件相比，Swing 组件的名前多一个“J”字母。另外，AWT 组件在 java.awt 包中，而 Swing 组件在 javax.swing 包中。

本书主要介绍 Swing 组件的使用。

3. 一个 Java GUI 简单程序

在介绍 Java GUI 程序设计方法前，下面先看一个简单的 Swing GUI 应用程序的例子。

【例 9.1】 一个简单的 Swing GUI 应用程序。在一个框架窗口中显示两个标签和一个按钮：上面的标签显示一串固定的文字信息，选择按钮后在下面的标签上显示系统现在的时间。

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;
// 继承 JFrame 类并实现 ActionListener 接口
public class SwingDemo extends JFrame implements ActionListener{
   JButton  b1;              // 声明按钮对象
   JLabel  l1,l2;            // 声明标签对象
   SwingDemo(){              // 定义构造方法
      super("Swing 应用程序的例子");    // 调用父类的构造方法
      l1=new JLabel("一个 GUI 应用程序的例子",JLabel.CENTER);// 定义标签，文字居中
```

```
        l2=new JLabel(" ");                    // 定义无文字标签
        b1=new JButton("现在时间[T]");          // 定义按钮
        b1.setMnemonic(KeyEvent.VK_T);         // 设置按钮的快捷键
        b1.setActionCommand("time");           // 设置按钮事件的控制名称
        b1.addActionListener(this);            // 注册按钮事件
        getContentPane().add(l1,BorderLayout.NORTH);    // 向内容窗格添加标签 l1
        getContentPane().add(l2,BorderLayout.CENTER);   // 向内容窗格添加标签 l2
        getContentPane().add(b1,BorderLayout.SOUTH);    // 向内容窗格添加标签 b1
    }
    // 对按钮引发事件编程
    public void actionPerformed(ActionEvent e){     // 捕获按钮事件
        Calendar c1 = Calendar.getInstance();       // 获取系统日期和事件
        if(e.getActionCommand().equals("time")){    // 判断是否为所需的按钮事件
            l2.setText("现在时间是"+c1.get(Calendar.HOUR_OF_DAY) // 设置标签文字
              +"时"+c1.get(Calendar.MINUTE)+"分");
            l2.setHorizontalAlignment(JLabel.CENTER);  // 设置标签标签文字居中对齐
        }else System.exit(0);
    }
    public static void main(String args[]){            // 主方法
        JFrame frame = new SwingDemo();                 // 创建 JFrame 对象，初始不可见
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  // 设置框架关闭按钮事件
        frame.pack();                                   // 压缩框架的显示区域
        frame.setVisible(true);                         // 显示框架主窗口
    }
}
```

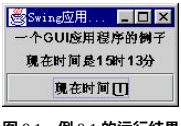

图 9.1　例 9.1 的运行结果

程序运行后显示的形式如图 9.1 所示。

这是 Swing GUI 应用程序的最简单的例子。语句不多，但说明了 Java Swing GUI 应用程序中的基本代码。

（1）引入合适的包和类

一般的 Swing GUI 应用程序应包含程序中的前三个引入语句，它们分别表示引入 awt 包、awt 事件处理包和 swing 包。例中的第四个引入语句引入的 util 包是程序中类 Calendar 所需要的。

由于 Swing 组件使用 AWT 的结构，包括 AWT 的事件驱动模式，所以，使用 Swing 组件的程序一般需要使用 awt 包。

（2）使用默认的观感或设置自己的观感（Look and Feel）

本程序使用了默认的观感。

（3）设置一个顶层的容器

本程序的顶层容器为 JFrame 框架。

（4）根据需要，使用默认的布局管理器或设置另外的布局管理器

本程序使用 JFrame 的默认布局管理器 BorderLayout。

（5）定义组件并将它们添加到容器

本程序使用两个标签（JLabel 类的对象）和一个按钮（JButton 类的对象）。

（6）对组件或事件编码

本程序对按钮事件进行了注册和捕获，对按钮事件的处理编写了代码。

9.1.2 容器、组件、布局和观感

一般地，从一个 GUI 程序的外貌可以见到一些对界面起到装饰美化作用的圆、矩形等几何图形和图像，也可以见到如按钮、列表等一些可进行人机交互的组件。在 Java GUI 程序中，这些界面元素应处于一个容器中，其中组件在容器中的摆放位置和大小由容器的布局管理器决定。设置不同的观感，可得到不同的组件外观和形态。

1，容器（Container）和组件（Component）

一个 Java 的图形用户界面的最基本元素是组件，组件是可以以图形化的方式显示在屏幕上并能与用户进行交互的对象，如一个按钮、一个文本框等。在 Java 语言中，通常将组件放在一定的容器内使用。容器实际上是一种具有容纳其他组件和容器的功能的组件。抽象类 Container 是所有容器的父类，其中包含了很多有关容器的功能和方法。而类 Container 又是 Java 语言的组件类 Component 的子类。

在类 Container 中包含了将组件加入容器的方法 add()，将组件移出容器的方法 remove() 和方法 removeAll()，获得组件的方法 getComponent()等。

Container 可以引发 ContainerEvent 类代表的容器事件。当容器中加入和移出组件时，容器将引发 COMPONENT_ADDED 和 COMPONENT_REMOVED 两种容器事件。若要编写响应容器事件的程序，应实现容器事件的监听器接口 ContainerListener，并在监听器内部实现该接口中的两个抽象方法。

- void componentAdded(ContainerEvent e) 向容器加入组件时被调用。
- void componentRemoved(ContainerEvent e) 当组件从容器移出时被调用。

在这两个方法内部，可以调用 ContainerEvent 类的实际参数 e 的有关方法：

- Component getChild()返回引发事件发生的组件。
- Container getContainer()返回引发事件的容器。
- String paramString()返回标识该事件的参数字符串。

每一个使用 Swing GUI 的应用程序都必须包含至少一个顶层 Swing 容器组件。这样的容器有三种：JFrame、JDialog 和 JApplet（用于 Java 小程序）。每一个 JFrame 对象实现一个主窗口，每一个 JDialog 对象实现一个第二窗口（依赖另一窗口的窗口），每一个 JApplet 对象实现一个 Java 小程序的显示区域。

在例 9.1 中，SwingDemo 程序有一个顶层容器 JFrame。frame 是 JFrame 类的一个实例对象，它是一个有边界、标题、关闭按钮的窗口。具有 GUI 的应用程序一般使用至少一个框架。

SwingDemo 程序也包含三个组件：两个标签组件，一个按钮组件。这些组件的排列形式由容器所用的布局管理器决定，各种容器都有自己默认的布局管理器。

另外，在例 9.1 中，为了要实现单击窗口的关闭按钮关闭窗口，使用了下面的代码：

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

JFrame 提供 setDefaultCloseOperation 方法来构造当用户单击关闭按钮时的动作。对单个框架窗口的应用程序来说，大多数情况下是退出应用程序，这由常量 EXIT_ON_CLOSE 指定。若使用的是 Java 平台的早期版本，需要实现一个事件监听来通过选择窗口关闭按钮退出应用程序：

```
frame.addWindowListener(new WindowAdapter() {
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
});
```

2. 布局管理器（Layout Manager）

为了使得图形用户界面具有良好的平台无关性，在 Java 语言中提供了布局管理器这个工具来管理组件在容器中的布局，而不使用直接设置组件位置和大小的方式。容器中的组件定位由布局管理器决定。每个容器都有一个默认的布局管理器，当容器需要对某个组件进行定位或判断其大小尺寸时，就会调用其相应的布局管理器。但也可以不用默认的布局管理器，在程序中指定其新的布局管理器。

Java 平台提供多种布局管理器，常用的有 FlowLayout、BorderLayout、GridLayout、CardLayout、BoxLayout 和 GridBagLayout 等。使用不同的布局管理器，组件在容器上的位置和大小都是很不一样的。

在程序中安排组件的位置和大小时，应该注意：

（1）容器中的布局管理器负责各个组件的大小和位置，因此用户无法在这种情况下直接设置这些属性。若试图使用 Java 语言提供的 setLocation()、setSize()、setBounds()等方法，则都会被布局管理器覆盖。

（2）若用户确实需要亲自设置组件的位置和大小，则应取消该容器的布局管理器，方法为：

```
setLayout(null);
```

随后，用户必须使用 setLocation()、setSize()、setBounds()等方法为组件设置大小和位置，但这种方法将会导致程序的系统相关。

在一个 GUI 应用程序的界面上，除了可以见到上述的容器、组件等标准 GUI 元素外，还可以见到一些几何图形、图案、图像等内容，这些内容通常是不能被系统识别进行人机交互的，在界面上只起到装饰、美化界面的作用。

3. 观感（Look and Feel）

Java Swing 的一个重要特征是它的可插入的“观感”体系。一个 Swing 应用程序或一个最终用户可指明所需要的观感，使得 Swing 应用程序的外观和行为都可以被定制。Swing 运行一个默认的 Java 观感（也称为 Metal 观感），还实现了模仿 Motif 和 Windows 的观感。这样，一个 Swing 程序可拥有 Java 程序的独特外观，也可以拥有熟悉的 Windows 操作系统外观。在图 9.2、图 9.3 和图 9.4 中分别显示了 Windows、Motif 和 Metal 三种观感。

图 9.2　Windows 观感

图 9.3　Motif 观感

图 9.4　Java（Metal）观感

一般在应用程序的 JFrame 的构造方法中设置或在 JApplet 的 init()方法中进行观感的设置。

【例 9.2】　设置观感。

```
import javax.swing.*;
import java.awt.*;
public class SetLAF{
  public static void setNativeLookAndFeel() {
    try {
      UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
    } catch(Exception e) {
      System.out.println("设置 native LAF 错误: " + e);
    }
  }
  public static void setJavaLookAndFeel() {
    try {
      UIManager.setLookAndFeel(UIManager.getCrossPlatformLookAndFeelClassName());
    } catch(Exception e) {
      System.out.println("设置 Java LAF 错误: " + e);
    }
  }
  public static void setMotifLookAndFeel() {
    try {
      UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
    } catch(Exception e) {
      System.out.println("设置 Motif LAF 错误: " + e);
    }
  }
}
```

因为 setLookAndFeel()方法抛出异常，因此设置观感的代码应捕捉异常。本例创建的类 SetLAF 可在以后的程序中设置观感时使用。

在例 9.1 中未设置观感，即使用默认的 Java（Metal）观感。

9.1.3 事件处理

在一个 GUI 程序中，为了能够接收用户的输入、命令的按键和鼠标操作，程序系统首先应该能够识别这些操作并做出相应的响应。通常一个键盘和鼠标操作将引发一个系统预先定义好的事件，用户程序只要编写代码定义每个事件发生时程序应做出何种响应即可。这些代码会在它们对应的事件发生时由系统自动调用，这就是 GUI 程序中事件和事件响应的基本原理。

在 Java 语言中，除了键盘和鼠标操作，系统的状态改变也可以引发事件。可能产生事件的组件称为事件源，不同事件源上发生的事件种类是不同的。若希望事件源上引发的事件被程序处理，需要将事件源注册给能够处理该事件源上那种事件类型的监听器。监听器具有监听和处理某类事件的功能，它可以是包容事件源的容器，也可以是另外的对象。也就是说，事件源和事件处理是分开的，一般组件都不处理自己的事件，而将事件处理委托给外部的处理实体，这种事件处理模型称为授权处理模型。

事件的行为多种多样，由不同的监听器处理。编写事件处理程序首先应确定关注的事件属于何种监听器类型。例如，选择按钮或普通菜单项引发的事件由 ActionListener 监听器处理。选择其他组件的事件也都有相应的监听器。在 AWT 中，提供 11 种标准的监听器类型，见表 9.1。在 Swing 中，也提供了较多类型的监听器，请参看有关资料。

表 9.1　AWT 标准监听器、适配器和相应的注册方法

监 听 器	适 配 器 类	注 册 方 法
ActionListener		addActionListener
AdjustmentListener		addAdjustmentListener
ComponentListener	ComponentAdapter	addComponentListener
ContainerListener	ContainerAdapter	addContainerListener
FocusListener	FocusAdapter	addFocusListener
ItemListener		addItemListener
KeyListener	KeyAdapter	addKeyListener
MouseListener	MouseAdapter	addMouseListener
MouseMotionListener	MouseMotionAdapter	addMouseMotionListener
TextListener		addTextListener
WindowListener	WindowAdapter	addWindowListener

在确定监听器类型后，要用事件源类的注册方法来注册一个监听器类的对象。这样，事件源产生的事件会传送给已注册的处理该类事件的监听器对象，该对象将自动调用相应的事件处理方法来处理该事件。具体的注册方法是：用监听器类的对象作为参数调用事件源本身的 addXxxListener()方法。该方法的参数是一个监听器类的对象，有多种形式，例如：

（1）分离的监听器类，该类通常为继承相应事件适配器类的子类，类中包含了事件处理方法。参数是该类的一个对象。

（2）实现监听器接口，参数为 this，表示本对象就是一个监听器类的对象。在本类中包含事件处理方法。

（3）有名内部类，参数形式为继承事件适配器类的子类对象，在子类中包含事件处理方法。

（4）匿名内部类，参数形式为用 new 开始的一个无名的类定义。其中包含事件处理方法。

在表 9.1 中，也列出了与监听器类型对应的适配器类和注册方法。

下面利用例 9.1 来说明授权处理模型的处理机制。

在例 9.1 中，关注的事件是选择按钮事件，选择按钮后，应显示现在的时间。按钮事件是 ActionEvent 事件，由 ActionListener 监听器处理，实现 ActionListener 接口的类定义为：

```
public class SwingDemo extends JFrame implements ActionListener{
```

即类 SwingDemo 实现监听器接口 ActionListener，需要重写接口中惟一 actionPerformed()方法来具体处理按钮引发的事件（这里是显示时间）。方法的参数是 ActionEvent 对象，该对象给出了事件和事件源的信息。为处理按钮事件，需对按钮注册一个监听器类的对象，程序例中是如下的语句：

```
b1.addActionListener(this);
```

b1 为按钮，this 为类 SwingDemo（本类）的实例对象。

事件处理过程如下：当用户选择按钮时，按钮引发一个动作事件 ActionEvent，因为按钮注册到 ActionListener 类的对象，所以按钮事件导致动作监听器的 actionPerformed()方法被调用，显示系统当前的时间。

若有多个事件源产生同类事件，可在事件处理方法中用 getSource()或 getActionCommand()方法判别引发事件的事件源。

Swing 组件可产生多种类型的事件，不同的组件引发的事件也不完全相同。

9.2 布局管理器

在容器中所有组件的布局（位置和大小）由布局管理器来控制。在 Java 语言中提供了 FlowLayout、BorderLayout、GridLayout、CardLayout 和 GridBagLayout 等多种布局管理器。

每种容器都有自己默认的布局管理器。在默认的情况下，JPanel 使用 FlowLayout，而内容窗格 ContentPane（JApplet、JDialog 和 JFrame 对象的主容器）使用 BorderLayout。如果不希望使用默认的布局管理器，则可使用所有容器的父类 Container 的 setLayout()方法来改变默认的布局管理器。例如，下面是使得 JPanel 使用 BorderLayout 的代码：

```
JPanel pane = new JPanel();
pane.setLayout(new BorderLayout());
```

当向面板或内容窗格等容器添加组件时，add 方法的参数个数和类型是不同的，这依赖于正在使用的容器的布局管理器。

下面介绍几种常用的布局管理器。

1，FlowLayout

FlowLayout 布局是一种最基本的布局。这种布局指的是把组件一个接一个从左至右、从上至下地依次放在容器上，每一行中的组件默认为居中对齐。当容器的尺寸改变后，组件的大小不变，但布局将会随之变化。

FlowLayout 是 Applet 和 JPanel 的默认布局管理器。FlowLayout 类的构造方法如下：

- FlowLayout() 创建每行组件对齐方式为居中对齐、组件间距为 5 个像素单位的对象。

▪ FlowLayout(int align) 创建指定每行组件对齐方式、组件间距为 5 个像素单位的对象，align 可取三个静态常量 LEFT、CENTER 和 RIGHT 之一（分别表示左、中、右对齐方式）。

▪ FlowLayout(int align, int hgap, int vgap) 创建指定每行组件对齐方式的对象，该对象也使用参数 vgap 和 hgap 指定了组件间的以像素为单位的纵横间距。

向使用 FlowLayout 布局的容器添加组件可简单地使用下面的语句：

```
add(组件名);
```

2，BorderLayout

BorderLayout 是内容窗格的默认布局管理器。内容窗格是框架 JFrame、小程序 JApplet 和对话框 JDialog 的主容器。BorderLayout 将容器的布局分为五个区：北区、南区、东区、西区和中区。这几个区的分布规律是“上北下南，左西右东”。当容器的大小改变时，容器中的各个组件相对位置不变，其中间部分组件的尺寸会发生变化，四周组件宽度固定不变。

BorderLayout 类的构造方法如下：

▪ BorderLayout() 创建组件间无间距的 BorderLayout 对象。

▪ BorderLayout(int hgap, int vgap) 创建有指定组件间距的对象。

向 BorderLayout 布局的容器添加组件时，每添加一个组件都应指明该组件加在哪个区域中。add()方法的第二个参数指明加入的区域，区域东南西北中可用五个静态常量表示：BorderLayout.EAST、BorderLayout.SOUTH、BorderLayout.WEST、BorderLayout.NORTH 和 BorderLayout.CENTER。

【例 9.3】 将五个按钮加入 BorderLayout 的五个区。

```
import java.awt.*;
import javax.swing.*;
public class BorderLayoutDemo extends JApplet {
  public void init() {
    Container c = getContentPane();
    c.add(new Button("北 North"), BorderLayout.NORTH);
    c.add(new Button("南 South"), BorderLayout.SOUTH);
    c.add(new Button("东 East"), BorderLayout.EAST);
    c.add(new Button("西 West"), BorderLayout.WEST);
    c.add(new Button("中 Center"), BorderLayout.CENTER);
  }
}
```

程序运行的结果如图 9.5 所示。

3，GridLayout

GridLayout 布局是将容器的空间分成若干行和列的一个个网格，可以给出网格的行数和列数，组件添加到这些网格中。当改变容器的大小后，其中的组件相对位置不变，但大小改变。容器中各个组件同高度、同宽度。各个组件默认的排列方式为：从上到下，从左到右。

GridLayout 类的构造方法如下：

- public GridLayout()创建单行每个组件一列的 GridLayout 对象。
- public GridLayout(int rows, int cols) 创建指定行列数的 GridLayout 对象。
- public GridLayout(int rows, int cols, int hgap, int vgap)创建指定行列数的 GridLayout 对象。

因为没有容器默认使用 GridLayout，因此在使用 GridLayout 前，要用 setLayout()方法将容器的布局管理器设置为 GridLayout。

在向 GridLayout 添加组件时，组件加入容器要按序进行，每个网格中都必须加入组件，若希望某个网格为空，可以为该网格加入一个空的标签：add(new JLabel())。

【例 9.4】 网格布局。

```
import java.awt.*;
import javax.swing.*;
public class GridLayoutDemo extends JApplet {
  public void init() {
    Container c = getContentPane();
    c.setLayout(new GridLayout(3,2));
    c.add(new Button("1"));
    c.add(new Button("2"));
    c.add(new Button("3"));
    c.add(new Button("4"));
    c.add(new Button("5"));
    c.add(new Button("6"));
  }
}
```

程序运行的结果如图 9.6 所示。

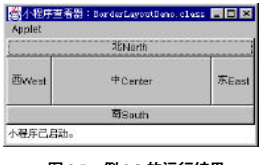

图 9.5 例 9.3 的运行结果

图 9.6 例 9.4 的运行结果

4，CardLayout

CardLayout 布局管理器能够使得多个组件共享同一显示空间，这些组件之间的关系像一叠重叠的扑克牌，只有最上面的组件是可见的。注意：在一个显示空间（卡片）中只能显示

一个组件，因此，可使用容器嵌套的方法来显示多个组件。

CardLayout 类的构造方法如下：

- CardLayout()创建间距为零的对象。
- CardLayout(int hgap, int vgap)创建带有水平 hgap 和垂直 vgap 间距的对象。

为了使用叠在下面的组件，可以为每个组件取一名字，名字在用 add()方法向容器添加组件时指定，需要某个组件时通过 show()方法指定该组件的名字来选取它。也可以顺序使用这些组件，或直接指明选取第一个组件（用 first()方法）或最后一个组件（用 last()方法）。

【例 9.5】 CardLayout 布局。

```
import java.awt.*;
import javax.swing.*;
public class CardLayoutDemo extends JApplet{
  CardLayout cl =new CardLayout(20,40);  // 组件在卡片中有边界
  JButton b1=new JButton("卡片一");
  JButton b2=new JButton("卡片二");
  JButton b3=new JButton("卡片三");
  public void init(){
    getContentPane().setLayout(cl);
      getContentPane().add("card1",b1);
      getContentPane().add("card2",b2);
      getContentPane().add("card3",b3);
  }
}
```

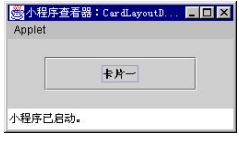

图 9.7 例 9.5 运行结果

程序运行结果如图 9.7 所示。程序中的三个按钮组件顺序添加到卡片布局管理器的各个卡片上，它们共享同一显示区域，因此只能见到最上面的“卡片一”按钮。

5，GridBagLayout

GridBagLayout 是最复杂也最灵活的布局管理器。这个布局管理器将组件放入单元格中，但允许一些组件跨越单元格。

可用 GridBagLayout 类的构造方法 GridBagLayout()来创建一个 GridBagLayout 布局管理器。因 GridBagLayout 布局设置比较复杂，这里就不介绍了，请读者参看 API 说明或其他资料。

9.3 常用 Swing 组件

9.3.1 容器组件

1，JFrame 框架

框架是 JFrame 类的对象，是 Swing GUI 应用程序的主窗口，窗口有边界、标题、关闭

按钮等。对 Java 应用程序，应至少包含一个框架，例 9.1 的应用程序即使用了框架。有时，小程序也使用框架。JFrame 类继承于 Frame 类。JFrame 类的构造方法如下：

- JFrame() 创建无标题的初始不可见框架。
- JFrame(String title) 创建标题为 title 的初始不可见框架。

例如，创建带标题“Java GUI 应用程序”的框架对象 frame，可用语句：

```
JFrame frame = new JFrame("Java GUI 应用程序");
```

要显示框架对象代表的框架窗口，可使用方法 setVisible()语句：

```
frame.setVisible(true);
```

可使得 JFrame 类对象 frame 表示的框架窗口显示到屏幕上。

一般在显示框架前，可设置框架的初始显示大小，可使用 setSize()方法或 pack()方法。例如：

```
frame.setSize(200,150);      // 设置框架窗口初始大小为 200×150 点
frame.pack();                // 设置框架窗口初始大小为刚好只显示出所有的组件
```

在向框架添加组件时，并不直接添加组件到框架，而是添加到内容窗格（content pane），改变其他特性（布局管理器、背景色等）的同时也对内容窗格进行了相应的改变。要存取内容窗格，可通过 getContentPane()方法，若希望用自己的容器替换掉内容窗格（例如用 JPanel），可以使用 setContentPane()方法。

选择框架的关闭按钮后，框架窗口将自动关闭，但若是应用单个框架的应用程序，为了在选择框架的关闭按钮时能退出程序，应添加 WindowListener 监听器或书写下列代码：

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

JFrame 的默认观感为 Java（Metal），若要使用其他的观感，必须显式设置。JFrame（应该说是内容窗格）的默认布局管理器是 BorderLayout。

2，JPanel 面板

JPanel 是一种添加到其他容器使用的容器组件，可将组件添加到 JPanel，然后再将 JPanel 添加到某个容器。JPanel 也提供一个绘画区域，可代替 AWT 的画布 Canvas（没有 Jcanvas）。

javax.swing.JPanel 类继承于 javax.swing.JComponent 类，其构造方法有：

- public JPanel() 创建具有默认 FlowLayout 布局的 JPanel 对象。
- public JPanel(LayoutManager layout) 创建具有指定布局管理器的 JPanel 对象。

将 JPanel 作为画布的绘画区域使用时，要使用下面的两个步骤：首先，设置画布绘图区域的大小；其次，使用 paintComponent()方法（不是 paint()方法）来绘图，在该方法体中，首先调用方法 super.paintComponent()来清除显示区域。例如：

```
public void paintComponent(Graphics g) {
  super.paintComponent(g);
  ...
}
```

JPanel 可指定边界，可用的边界有 titled、etched、beveled 、line、matte、compound 和 empty 等，也可以创建自己的边界。可用 JComponent 类的 setBorder()方法设置边界。其用法如下：

```
public void setBorder(Border border)
```

其中，Border 类的参数可用 javax.swing.BorderFactory 类中的方法获得。获取各种相应边界的方法为：createTitledBorder()、createEtchedBorder()、createBevelBorder()、createRaisedBevelBorder()、createLoweredBevelBorder()、createLineBorder()、createMatteBorder()、createCompoundBorder()和 createEmptyBorder()。例 9.6 说明了边界的设置操作。

注意：也可以对其他组件如 JSlider 滚动条等设置边界。

【例 9.6】 使用 JPanel。

```
import java.awt.*;
import javax.swing.*;
class JPanelDemo extends JPanel {
   JButton b1 = new JButton("JPanel");
   JButton b2 = new JButton("Demo");
   public JPanelDemo() {
      setBackground(Color.white);
      add(b1);
      add(b2);
   }
   public static void main(String[] args) {
      JPanel jp = new JPanelDemo();
      jp.setBorder(BorderFactory.createTitledBorder("Hello,Border"));   //设置边界
      JFrame frame = new JFrame("JPanelDemo");
      frame.setSize(200, 150);
      frame.setContentPane(jp);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

程序运行结果如图 9.8 所示。

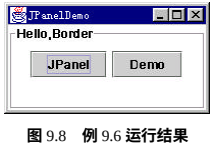

图 9.8 例 9.6 运行结果

3. JApplet

javax.swing.JApplet 类是 java.applet.Applet 类的子类。使用 Swing 组件的小程序需继承 JApplet 类。

除了所处的 java 包不同外，JApplet 与 Applet 的主要区别介绍如下。

（1）默认的布局管理器不同

Applet 默认的布局管理器是 FlowLayout，而 JApplet（内容窗格）默认的布局管理器是 BorderLayout。

（2）加入组件的方法不同

Applet 可直接加入组件，而 JApplet 默认使用内容窗格 ContentPane 作为主容器。加入 Swing 组件时，要先使用 JApplet 的方法 getContentPane()获得一个 Container 对象，再调用这个对象的 add()方法将 Swing 组件加入到 JApplet 的容器中。

4，JTabbedPane

javax.swing.JTabbedPane 类继承于 javax.swing.JComponent，它的对象反映为一组带标签的面板，每个面板都可以存放组件，因此 JTabbedPane 是一个容器组件。

JTabbedPane 类的构造方法有：

- JTabbedPane()创建空对象，该对象具有默认的标签位置 JTabbedPane.TOP 和默认的布局策略 JTabbedPane.WRAP_TAB_LAYOUT。
- JTabbedPane(int tabPlacement) 创建空对象，该对象具有指定的标签位置 JtabbedPane.TOP、JTabbedPane.BOTTOM、JTabbedPane.LEFT 和 JTabbedPane.RIGHT 以及默认的布局策略 JTabbedPane.WRAP_TAB_LAYOUT。
- JTabbedPane(int tabPlacement, int tabLayoutPolicy)创建空对象，该对象具有指定的标签位置和布局策略。

【例 9.7】 使用 JTabbedPane 容器。

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
public class JTabbedPaneDemo{
   public static void main(String args[]){
      new MyTabbedPane();
   }
}
class MyTabbedPane extends JFrame implements ChangeListener,ActionListener{
   JTabbedPane jt;
   JButton jb[];
   int index = 0;
   MyTabbedPane(){
      super("使用标签面板容器");
      jt = new JTabbedPane();
      jb = new JButton[5];
      for(int i = 0;i<5;i++){
         jb[i] = new JButton("第" + i + "页面板");
         jb[i].addActionListener(this);
```

```
            jt.addTab("页标签" + i,jb[i]);
        }
        jt.addChangeListener(this);
        getContentPane().add(jt,BorderLayout.CENTER);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300,200);
        setVisible(true);
    }
    public void stateChanged(ChangeEvent e){
        if(e.getSource()==jt){
            int i = ((JTabbedPane)e.getSource()).getSelectedIndex();
            jb[index].setVisible(false);
            jb[i].setVisible(true);
            index = i;
        }
    }
    public void actionPerformed(ActionEvent e){
        setTitle("响应单击"+((JButton)e.getSource()).getText());
    }
}
```

程序运行结果如图 9.9 **所示。**

图 9.9　**例** 9.7 **运行结果**

9.3.2　按钮（JButton）

按钮是 GUI **中非常重要的一种基本组件。按钮一般对应一个事先定义好的事件、执行功能、一段程序。当使用者单击按钮时，系统自动执行与该按钮联系的程序，从而完成预定的功能。**

JButton **类提供对按钮的支持，它的类层次关系如下：**

java.awt.Container - javax.swing.JComponent - javax.swing.AbstractButton - javax.swing.JButton。

JButton **类有如下的构造方法：**

- JButton()　**创建空按钮。**
- JButton(Icon icon)　**创建带图标的按钮。**
- JButton(String text)　**创建带文字的按钮。**

▪ JButton(String text, Icon icon) 创建带文字和图标的按钮。

JButton 组件与 AWT 的 Button 组件相比，增加了显示文本中可用 HTML 标记，可带图标等功能。

在 JButton 按钮的使用中，常用到继承来的 setMnemonic()（设置快捷字母键）、setActionCommand()（设置动作命令）方法等。

JButton 组件引发的事件为 ActionEvent，可实现 ActionListener 监听器接口的 actionPerformed()方法，用 addActionListener()方法注册，用 getActionCommand()或 getSource()方法确定事件源。

【例 9.8】 设计一个 GUI 应用程序，有两个标签 l1、l2 和三个按钮 b1、b2、b3。l1 标签显示固定的文字，l2 标签的文字随选择不同的按钮而变化：选择 b1 按钮时，l2 标签显示为“欢迎进入 Java 世界”，选择 b2 按钮时，l2 标签显示当前的日期，选择 b3 按钮时，退出该应用程序。程序如下：

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;
public class JButtonDemo extends JPanel implements ActionListener{
  JButton b1,b2,b3;
  static JLabel l1,l2;
  JButtonDemo(){
    l1 = new JLabel(" 这是一个演示按钮动作的程序",JLabel.CENTER);
    l2 = new JLabel(" ",JLabel.CENTER);
    b1 = new JButton("欢迎[w]");
    b1.setMnemonic(KeyEvent.VK_W);  // 设置按钮的快捷键
    b1.setActionCommand("welcome");
    b2 = new JButton("日期[d]");
    b2.setMnemonic(KeyEvent.VK_D);   // 设置快捷字符为 D
    b2.setActionCommand("date");
    b3 = new JButton("退出[q]");
    b3.setMnemonic(KeyEvent.VK_Q);   // 设置快捷字符为 Q
    b3.setActionCommand("quit");
    b1.addActionListener(this);
    b2.addActionListener(this);
    b3.addActionListener(this);
    add(b1);add(b2);add(b3);
  }
  public void actionPerformed(ActionEvent e){
    Calendar c1 = Calendar.getInstance();
    if(e.getActionCommand().equals("welcome"))
      l2.setText("欢迎进入 Java 世界。");
```

```
        else if(e.getActionCommand().equals("date"))
          l2.setText("今天是"+c1.get(Calendar.YEAR)+"年"+(c1.get(Calendar.MONTH)+1)+
              "月" + c1.get(Calendar.DATE) + "日");
        else System.exit(0);
        l2.setHorizontalAlignment(JLabel.CENTER);    // 标签文字水平居中
      }
      public static void main(String args[]){
        JFrame frame=new JFrame("使用 JButton");
        frame.getContentPane().add(new JButtonDemo(),BorderLayout.SOUTH);
        frame.getContentPane().add(l1,BorderLayout.NORTH);
        frame.getContentPane().add(l2,BorderLayout.CENTER);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
      }
    }
```

本程序将两个标签和一个包含三个按钮的面板（使用 FlowLayout 布局）添加到框架上（使用 BorderLayout 布局）。命令按钮设置了快捷字母键，可用鼠标单击或按 Alt + 快捷字母来选择按钮。

程序运行启动后选择“欢迎”按钮和选择“日期”按钮后显示的情况，见图 9.10 和图 9.11。

图 9.10　例 9.8 运行结果（一）

图 9.11　例 9.8 运行结果（二）

【例 9.9】　带图形和 HTML 文字的按钮。

```
import java.awt.*;
import javax.swing.*;
public class JButtonDemo1 extends JFrame{
  public static void main(String[] args){
    new JButtonDemo1();
  }
  public JButtonDemo1(){
    super("Using JButton");
    Container content = getContentPane();
    content.setBackground(Color.white);
    content.setLayout(new FlowLayout());
    JButton button1 = new JButton("<html><h2><font color = blue>Java");
    content.add(button1);
```

```
        ImageIcon im = new ImageIcon("images/newssms.gif");
        JButton button2 = new JButton(im);
        content.add(button2);
        JButton button3 = new JButton("Java", im);
        content.add(button3);
        JButton button4 = new JButton("Java", im);
        button4.setHorizontalTextPosition(SwingConstants.LEFT);
        content.add(button4);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setVisible(true);
      }
    }
```

程序中使用了类 IconImage **和指定图标图像文件名来创建图标图像对象，程序的运行结果如图** 9.12 **所示。**

图 9.12　**例** 9.9 **的运行结果**

9.3.3　标签（JLabel）

标签是用户不能修改只能查看其内容的组件，常用来在界面上输出信息。JLabel **类提供了对标签的支持，它的类层次关系为：**javax.swing.JComponent - javax.swing.JLabel**。**

JLabel **类的构造方法有：**

- JLabel()　**创建一个空标签。**
- JLabel(Icon image)　**创建一个带指定图像的标签。**
- JLabel(Icon image, int horizontalAlignment)**创建一个带指定图像和水平对齐方式的标签。**
- JLabel(String text)　**创建一个带文字的标签。**
- JLabel(String text, Icon icon, int horizontalAlignment)**创建一个带文字、图像和指定水平对齐方式的标签。**
- JLabel(String text, int horizontalAlignment)**创建一个带文字和指定水平对齐方式的标签。**

其中，horizontalAlignment **水平对齐方式可以使用表示左对齐、右对齐、居中对齐的常量** JLabel.LEFT、JLabel.RIGHT **和** JLabel.CENTER**。**

【例 9.10**】　具有文字对齐的标签。**

```
import javax.swing.*;
import java.awt.*;
public class JLabelAlignDemo extends JApplet {
   public void init(){
      Container c = getContentPane();
      c.add(new JLabel("文字左对齐标签",JLabel.LEFT),BorderLayout.NORTH);
```

```
        c.add(new JLabel("文字右对齐标签",JLabel.RIGHT),BorderLayout.CENTER);
        c.add(new JLabel("文字居中标签",JLabel.CENTER),BorderLayout.SOUTH);
    }
}
```

程序运行结果如图 9.13 **所示。**

JLabel **类常用方法有：**

- public void setText(String text)**定义这个组件将显示的单行文字。**
- public String getText() **返回标签显示的文字。**
- public Icon getIcon() **返回标签显示的图像。**
- public void setIcon(Icon icon)**定义这个组件将显示的图标。**

【例 9.11**】 使用带图标的标签。**

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JLabelDemo extends JApplet {
    public void init() {
        Container c = getContentPane();
        Icon icon = new ImageIcon("images/cup.gif");
        JLabel label = new JLabel("Swing!", icon, JLabel.CENTER);
        c.add(label, BorderLayout.CENTER);
    }
}
```

程序运行结果如图 9.14 **所示。**

图 9.13 **例** 9.10 **运行结果**

图 9.14 **例** 9.11 **的运行结果**

9.3.4 **复选框**（JCheckBox）

JCheckBox **类提供复选框按钮的支持。复选框按钮是具有开关或真假状态的按钮。**

JCheckBox **类的层次关系如下：**

javax.swing.AbstractButton - javax.swing.JToggleButton - javax.swing.JcheckBox。

JCheckBox **类的构造方法如下：**

- JCheckBox()**创建无文本的初始未选复选框按钮。**
- JCheckBox(Icon icon)**创建有图像无文本的初始未选复选框按钮。**

- JCheckBox(Icon icon, boolean selected)**创建带图像和选择状态但无文本的复选框按钮。**
- JCheckBox(String text)**创建带文本的初始未选复选框按钮。**
- JCheckBox(String text, boolean selected)**创建具有指定文本和状态的复选框按钮。**
- JCheckBox(String text, Icon icon)**创建具有指定文本和图标图像的初始未选复选框按钮。**
- JCheckBox(String text, Icon icon, boolean selected)**创建具有指定文本、图标图像、选择状态的复选框按钮。**

其中，构造方法的参数 selected **若为真，则表示按钮初始状态为选中。**

JCheckBox **类常用的方法有继承来的方法** isSelected()**，其格式为：**

```
public boolean isSelected()
```

当复选框按钮选中时返回 true**，否则返回** false**。**

JCheckBox **类的选择事件是** ItemEvent**，可实现** ItemListener **监听器接口的** itemStateChanged()**方法来处理事件，用** addItemListener()**方法注册。**

【例 9.12**】 选择粗体、斜体复选框按钮，改变文本框中显示文字的字形。**

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class JCheckBoxDemo extends JApplet implements ItemListener{
   private JTextField t;
   private JCheckBox bold,italic;
   public void init(){
      t = new JTextField("观察这里文字字形的变化",40);
      t.setFont(new Font("Serif",Font.PLAIN,20));
      getContentPane().add(t,BorderLayout.NORTH);
      bold = new JCheckBox("粗体 Bold");
      bold.addItemListener(this);
      getContentPane().add(bold,BorderLayout.CENTER);
      italic = new JCheckBox("斜体 Italic");
      italic.addItemListener(this);
      getContentPane().add(italic,BorderLayout.SOUTH);
   }
   public void itemStateChanged(ItemEvent e){
      int b = bold.isSelected()?Font.BOLD:Font.PLAIN;
      int i = italic.isSelected()?Font.ITALIC:Font.PLAIN;
      t.setFont(new Font("Serif",b + i,20));
   }
}
```

程序运行结果如图 9.15 **所示。**

图 9.15　例 9.12 的运行结果

9.3.5　单选按钮（JRadioButton）

在一组单选按钮中，可进行选择其中一个的操作，即进行“多选一”。JRadioButton 类的类层次和构造方法的参数构成都与前面介绍的 JCheckBox 类相同，这里不再列出 JRadioButton 类的这些内容。

因为单选按钮是在一组按钮中选择一个，因此必须将单选按钮分组，即指明在一个组中包含哪些按钮。可用 ButtonGroup 创建按钮组对象，应用对象的 add()方法顺序加入各个单选按钮。

在单选按钮中也可以使用 HTML 代码，这是 Java SDK 1.3 版新增的功能。

单选按钮的选择事件是 ActionEvent 类事件。

【例 9.13】　使用单选按钮来设置 Swing 应用程序的不同观感。

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class JRadioButtonDemo extends JPanel {
   static JFrame frame;
   static String metal= "Metal";
   static String motif = "Motif";
   static String windows = "Windows";
   JRadioButton metalButton, motifButton, windowsButton;
   public JRadioButtonDemo(){
      JButton button = new JButton("Hello, world");
      button.setMnemonic('h');
      metalButton = new JRadioButton(metal);
      metalButton.setMnemonic('o');
      metalButton.setActionCommand(metal);
      motifButton = new JRadioButton(motif);
      motifButton.setMnemonic('m');
      motifButton.setActionCommand(motif);
      windowsButton = new JRadioButton(windows);
      windowsButton.setMnemonic('w');
      windowsButton.setActionCommand(windows);
      // 将单选按钮设置为一组
      ButtonGroup group = new ButtonGroup();
      group.add(metalButton);
      group.add(motifButton);
      group.add(windowsButton);
      // 对单选按钮设置监听器
      RadioListener myListener = new RadioListener();
```

```
        metalButton.addActionListener(myListener);
        motifButton.addActionListener(myListener);
        windowsButton.addActionListener(myListener);
        add(button);
        add(metalButton);
        add(motifButton);
        add(windowsButton);
    }
    /** ActionListener 监听器监听单选按钮*/
    class RadioListener implements ActionListener {  // 嵌套类
        public void actionPerformed(ActionEvent e) {
            if((e.getActionCommand()).equals(metal))
                SetLAF.setJavaLookAndFeel();
            else if((e.getActionCommand()).equals(motif))
                SetLAF.setMotifLookAndFeel();
            else SetLAF.setNativeLookAndFeel();
            SwingUtilities.updateComponentTreeUI(frame);
            frame.pack();
        }
    }
    public static void main(String s[]) {
        JRadioButtonDemo panel = new JRadioButtonDemo();
        frame = new JFrame("使用 JRadioButton 选择观感");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add("Center", panel);
        frame.pack();
        frame.setVisible(true);
    }
}
```

程序运行时可用单选按钮选择三种不同的观感，运行结果如图 9.2、**图** 9.3 **和图** 9.4 **所示。在本程序中，使用了类的嵌套，类** RadioListener **嵌套在类** JRadioButtonDemo（**称为包含类**）**中，嵌套类可以访问包含类中的成员。另外，在程序中使用了** SwingUtilities **类的静态方法** updateComponentTreeUI()**用设置的当前观感来更新界面特性。注意：本程序编译时要先编译例** 9.2**。**

9.3.6 文本框

Java **语言提供了单行文本框、口令框和多行文本框等文本框形式，它们都是人机交互的主要组件。**

1，单行文本框（JTextField）

单行文本框一般用来让用户输入如姓名、地址这样的信息，它是一个能够接收用户的键盘输入的单行文本区域。类 JTextField 提供对单行文本框的支持，它的类层次如下：

javax.swing.JComponent - javax.swing.text.JTextComponent - javax.swing.JtextField。

JTextField 类有如下的几种构造方法：

- JTextField() 创建一个新的单行文本框。
- JTextField(int columns) 创建具有指定长度的空单行文本框。
- JTextField(String text) 创建带初始文本内容的单行文本框。
- JTextField(String text, int columns) 创建带初始文本内容并具有指定长度的单行文本框。

JTextField 类的常用方法有：

- public void setText(String s)在文本框中显示字符串 s。
- public String getText()获得文本框中的字符串。

当用户在文本框里按回车键时，就产生了一个 ActionEvent 事件。当用户在文本框中移动文本光标时，就产生 CaretEvent 事件，可注册 addCaretListener 监听器，实现 CaretListener 的 caretUpdate()进行事件处理。

【例 9.14】 编写温度变换的应用程序，将摄氏温度转换为华氏温度。

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class CelsiusConverter implements ActionListener {
   JFrame frame;
   JPanel panel;
   JTextField tempCelsius;
   JLabel celsiusLabel, fahrenheitLabel;
   JButton convertTemp;
   public CelsiusConverter() { // 构造方法
      frame = new JFrame("温度变换");
      panel = new JPanel();
      panel.setLayout(new GridLayout(2, 2));
      tempCelsius = new JTextField();
      celsiusLabel = new JLabel("摄氏温度 = ");
      convertTemp = new JButton("变换");
      fahrenheitLabel = new JLabel("? 华氏温度");
      convertTemp.addActionListener(this); // 注册监听变换按钮事件
      panel.add(tempCelsius);
      panel.add(celsiusLabel);
      panel.add(convertTemp);
      panel.add(fahrenheitLabel);
```

```
        frame.getContentPane().add(panel, BorderLayout.CENTER);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();// 显示变换程序
        frame.setVisible(true);
    }
    // 实现监听器接口
    public void actionPerformed(ActionEvent event) {
        int tempF;
        tempF = (int)((Double.parseDouble(tempCelsius.getText())) * 1.8 + 32);
        fahrenheitLabel.setText(tempF + " 华氏温度");
    }
    public static void main(String[] args) {
        CelsiusConverter converter = new CelsiusConverter();
    }
}
```

程序运行显示如图 9.16 所示。

本程序启动后，在文本框中输入摄氏温度后，单击“变换”按钮，将在一个标签中显示对应的华氏温度值。要处理按钮事件，因此向按钮添加事件监听器。

图 9.16　例 9.14 的运行结果

2. 口令框（JPasswordField）

单行口令文本框 JPasswordField 类是 JTextField 类的子类。在 JPasswordField 对象中输入的文字会被其他字符替代，这个组件常用来在 Java 程序中输入口令。

JPasswordField 类的构造方法与 JTextField 类的构造方法类似，常用的其他方法有：

- char[] getPassword()　返回输入的口令。
- char getEchoChar()　返回输入文本时回显在框中的字符。回显字符默认为字符“*”。
- void setEchoChar(char c)　设置回显字符。

【例 9.15】　使用口令框。

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class JPasswordFieldDemo extends JApplet implements ActionListener{
    JLabel jl1,jl2;
    JPasswordField jp1,jp2;
    JButton jb1,jb2;
    public void init(){
        Container c = getContentPane();
        jl1 = new JLabel("<html><h3>请输入您的密码:" ,JLabel.CENTER);
        jl2 = new JLabel("<html><h3>请再次输入密码:" ,JLabel.CENTER);
```

```
        jp1 = new JPasswordField(8);
        jp2 = new JPasswordField(8);
        jb1 = new JButton("<html><h3>提交");
        jb2 = new JButton("<html><h3>取消");
        c.setLayout(new GridLayout(3,2));
        c.add(jl1);
        c.add(jp1);
        c.add(jl2);
        c.add(jp2);
        c.add(jb1);
        c.add(jb2);
        jb1.addActionListener(this);
        jb2.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e){
        if(e.getSource()==jb1)
        if(jp1.getPassword().length>0)
            if(String.valueOf(jp1.getPassword()).equals
                (String.valueOf(jp2.getPassword())))
              showStatus("密码输入成功！");
            else{
              showStatus("两次输入的密码不同，请重新输入！");
              jp1.setText("");
              jp2.setText("");
            }
        else showStatus("密码不能为空！");
        if(e.getSource()==jb2){
          jp1.setText("");
          jp2.setText("");
          showStatus("");
        }
    }
}
```

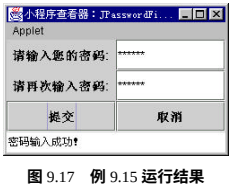

图 9.17　例 9.15 运行结果

程序运行显示如图 9.17 所示。

3. 多行文本框（JTextArea）

JTextField 是单行文本框，不能显示多行文本，如果想要显示大段的多行文本，可以使用类 JTextArea 支持的多行文本框。JTextArea 有六个构造方法，常用的有四个：

- JTextArea() 创建空多行文本框。

- JTextArea(int rows, int columns) 创建指定行列数的多行文本框。
- JTextArea(String text) 创建带初始文本内容的多行文本框。
- JTextArea(String text, int rows, int columns) 创建带初始文本内容和指定大小的多行文本框。

其中，text 为 JTextArea 的初始化文本内容；rows 为 JTextArea 的高度，以行为单位；columns 为 JTextArea 的宽度，以字符为单位。例如，构造一个高 5 行，宽 15 个字符的多行文本框的语句为：

```
textArea = new JTextArea(5, 15);
```

多行文本框默认是不会自动折行的（但可以输入回车符换行），可以使用类 JTextArea 的 setLineWrap(boolean wrap)方法设置是否允许自动折行。wrap 为 true 时允许自动折行。多行文本框会根据用户输入的内容自动扩展大小。若不自动折行，那么多行文本框的宽度是由最长的一行文字确定的；若数据行数超过了预设的行数，则多行文本框会扩展自身的高度去适应。换句话说，多行文本框不会自动产生滚动条。这时，可用滚动窗格（JScrollPane）来为多行文本框增加滚动条。

滚动窗格是一个能够自己产生滚动条的容器，通常只包容一个组件，并且根据这个组件的大小自动产生滚动条。这种性质正是 JTextArea 组件所需要的：当数据行数超过包含在 JScrollPane 中的 JTextArea 预设区域大小时，JTextArea 的扩展就会反映到滚动窗格的滚动条上。

多行文本框里文本内容的获得和设置，同样可以使用 getText()和 setText()两个方法来完成。还可以用 setEditable(boolean)确定是否可对多行文本框的内容进行编辑。

【例 9.16】 使用带滚动条的多行文本框。

```
import javax.swing.*;
import java.awt.*;
public class JTAreaAndJSPaneDemo extends Jframe  {
   public JTAreaAndJSPaneDemo() {
      super("使用带滚动条的多行文本框");
      JTextArea textArea = new JTextArea(5, 15);
      textArea.setLineWrap(true);
      JScrollPane jsp = new JScrollPane(textArea);
      getContentPane().add(jsp, BorderLayout.CENTER);
      pack();
      setDefaultCloseOperation(EXIT_ON_CLOSE);
   }
   public static void main(String[] args) {
      JTAreaAndJSPaneDemo tp = new JTAreaAndJSPaneDemo();
      tp.setVisible(true);
   }
}
```

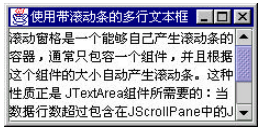

图 9.18　例 9.16 运行结果

程序运行显示如图 9.18 所示。

9.3.7　列表框（JList）

JList 类支持的列表框是允许用户从一个列表中选择一项或多项的组件，显示一个数组和向量的表是很容易的。列表框使用户易于操作大量的选项。列表框的所有项目都是可见的，如果选项很多，超出了列表框可见区的范围，则列表框的旁边会有一个滚动条。

JList 类的构造方法如下：

- JList()　创建空模式的对象。
- JList(Object[] listData)　构造显示指定数组 listData 中元素的 JList 对象。

例如，创建一个显示数组 data 中字符串的 JList：

```
String[] data = {"one", "two", "three", "four"};
JList dataList = new JList(data);
```

JList 类的常用方法见例 9.17。

【例 9.17】　使用 JList。

```
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;
public class JListDemo extends JFrame {
   public static void main(String[] args) {
      new JListDemo();
   }
   private JList jl;
   private JTextField valueField;
   public JListDemo() {
      super("一个简单的 JList");
      WindowUtilities.setNativeLookAndFeel();
      addWindowListener(new ExitListener());
      Container content = getContentPane();
      String[] entries = { "北京","上海","天津","重庆","武汉"};
      jl = new JList(entries);  // 创建 JList 对象
      jl.setVisibleRowCount(4); // 设置 JList 显示行数
      Font displayFont = new Font("Serif", Font.BOLD, 18);
     jl.setFont(displayFont);
     jl.addListSelectionListener(new ValueReporter()); // 注册监听器
      JScrollPane listPane = new JScrollPane(jl);  // 增添 JList 的滚动条
      JPanel listPanel = new JPanel();
```

```
        listPanel.setBackground(Color.white);
        Border listPanelBorder = BorderFactory.createTitledBorder("JList 数据");
        listPanel.setBorder(listPanelBorder);   //  设置面板边界
        listPanel.add(listPane);
        content.add(listPanel, BorderLayout.CENTER);
        JLabel valueLabel = new JLabel("选择为:");
        valueLabel.setFont(displayFont);
        valueField = new JTextField("", 6);
        valueField.setFont(displayFont);
        JPanel valuePanel = new JPanel();
        valuePanel.setBackground(Color.white);
        Border valuePanelBorder = BorderFactory.createTitledBorder("JList 选择");
        valuePanel.setBorder(valuePanelBorder);
        valuePanel.add(valueLabel);
        valuePanel.add(valueField);
        content.add(valuePanel, BorderLayout.SOUTH);
        pack();
        setVisible(true);
    }
    private class ValueReporter implements ListSelectionListener {
        public void valueChanged(ListSelectionEvent event) {
            if (!event.getValueIsAdjusting())
                valueField.setText(jl.getSelectedValue().toString());
        }
    }
}
```

程序的运行结果如图 9.19 **所示。**

图 9.19　**例** 9.17 **运行结果**

9.3.8　组合框（JComboBox）

在 Java **语言中，组合框有可编辑的和不可编辑的两种不同的形式，默认是不可编辑的组合框。这里对不可编辑的组合框进行介绍。组合框用于在多项选择中选择一项的操作。在未选择组合框时，组合框显示为带按钮的一个选项的形式，当对组合框按键或单击时，组合框会打开可列出多项的一个列表，提供给用户选择。**

类 JComboBox **提供组合框的支持，其相关类的层次如下：**

javax.swing.JComponent - javax.swing.JComboBox。

类 JComboBox()**的构造方法有四种，常用的有如下两种：**

- JComboBox()**用默认的数据模式创建组合框。**

- JComboBox(Object[] items)**用指定数组创建组合框。**

创建组合框后，可用方法 setSelectedIndex(int anIndex)**选定指定下标** anIndex **处的项目；可用方法** getSelectedIndex()**获得选定项目的数组下标；可用方法** getSelectedItem()**获取选定的项目。**

组合框事件是 ActionEvent **事件，事件处理方法与其他处理同类事件的方法类似。**

【例 9.18**】 使用** JComboBox**。**

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class JComboBoxDemo extends JPanel {
  JLabel picture,text;
  public JComboBoxDemo() {
    String[] pStrings = {"cup","cat","boy","girl"};
    JComboBox pList = new JComboBox(pStrings);
    pList.setSelectedIndex(0);
    pList.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        JComboBox cb = (JComboBox)e.getSource();
        String pName = (String)cb.getSelectedItem();
        picture.setIcon(new ImageIcon("images/" + pName + ".gif"));
        text.setText(pName);
        text.setHorizontalAlignment(JLabel.CENTER);
      }
    });
    picture = new JLabel(new ImageIcon("images/" +
        pStrings[pList.getSelectedIndex()] +".gif"));
    picture.setBorder(BorderFactory.createEmptyBorder(10,0,0,0));
    picture.setPreferredSize(new Dimension(180, 140));
    text = new JLabel(pStrings[pList.getSelectedIndex()],JLabel.CENTER);
    setLayout(new BorderLayout());
    add(pList,BorderLayout.NORTH);
    add(picture,BorderLayout.CENTER);
    add(text,BorderLayout.SOUTH);
    setBorder(BorderFactory.createEmptyBorder(20,20,20,20));
  }
  public static void main(String args[]) {
    JFrame frame = new JFrame("使用 JComboBox");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setContentPane(new JComboBoxDemo());
    frame.pack();
```

```
        frame.setVisible(true);
    }
}
```

程序的运行结果如图 9.20 所示。

图 9.20　例 9.18 运行结果

9.3.9　滚动条（JSlider）

在某些程序中，需要调整线性的值，这时就需要滚动条。滚动条提供了易于操作的值的范围或区的范围。javax.swing.JSlider 类提供了对滚动条的支持，它的父类是 javax.swing.JComponent 类。

可用下列 JSlider 类的构造方法生成 JSlider 对象：

- JSlider()　创建范围为 0 至 100，初值为 50 的水平滚动条。
- JSlider(int orientation)　创建范围为 0 至 100，初值为 50 的水平或垂直滚动条。表示方向的值可为常量 JSlider.HORIZONTAL 和 JSlider.VERTICAL，分别表示水平和垂直方向。
- JSlider(int min, int max)　创建范围为 min 至 max，初值为 min 和 max 平均值的水平滚动条。
- JSlider(int min, int max, int value)　创建范围为 min 至 max，初值为 value 的水平滚动条。
- JSlider(int orientation, int min, int max, int value)　创建范围为 min 至 max，初值为 value 的水平或垂直滚动条。

对创建的 JSlider 对象可显示数据刻度和数据值，也可设置边界等。与其他接口组件一样，滚动条产生一个可以控制的事件 ChangeEvent。例 9.19 说明了 JSlider 的使用。

【例 9.19】　使用 JSlider。

```
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;
public class JSliderDemo extends JFrame implements ChangeListener{
   JLabel l1;
     int v1,v2;
   JSlider sl1,sl2;
   public static void main(String[] args) {
      new JSliderDemo();
   }
   public JSliderDemo() {
      super("使用 JSlider");
      Container c = getContentPane();
      c.setBackground(Color.white);
      sl1 = new JSlider(JSlider.VERTICAL,100,200,100);  // 创建垂直滚动条
      sl1.setMajorTickSpacing(50);     // 设置大刻度间隔
```

```
        sl1.setMinorTickSpacing(10);    // 设置小刻度间隔
        sl1.setPaintTicks(true);        // 显示刻度
        sl1.setPaintLabels(true);       // 显示标注
        sl1.addChangeListener(this);    // 注册监听器
        c.add(sl1, BorderLayout.WEST);
        sl2 = new JSlider();            // 创建水平滚动条
        sl2.setBorder(BorderFactory.createTitledBorder("JSlider(具有带标题的边框、刻度和标注)"));
                                        // 设置组件带标题的边界
        sl2.setMajorTickSpacing(20);
        sl2.setMinorTickSpacing(5);
        sl2.setPaintTicks(true);        // true 表示显示产生滚动条上的刻度
        sl2.setPaintLabels(true);       // true 表示显示刻度上的数据
        sl2.addChangeListener(this);
        c.add(sl2,BorderLayout.SOUTH);
        l1=new JLabel("垂直滚动条的值为："+100+"水平滚动条的值为："+50);
        c.add(l1,BorderLayout.CENTER);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // 设置框架关闭按钮事件
        pack();
        setVisible(true);
    }
    public void stateChanged(ChangeEvent e){
        if(e.getSource()==sl1)
            v1=((JSlider)e.getSource()).getValue();
        else if(e.getSource()==sl2)
            v2=((JSlider)e.getSource()).getValue();
        l1.setText("垂直滚动条的值为："+v1+",水平滚动条的值为："+v2);
    }
}
```

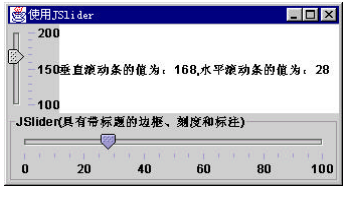

图 9.21　例 9.19 的运行结果

程序运行结果如图 9.21 所示。

9.3.10　菜单

菜单将一个应用程序的命令按层次化管理并组织在一起，是一种常用的 GUI 组件。常见的菜单为下拉式菜单和弹出式菜单（快捷菜单）等。

下拉式菜单包含有一个菜单条（也称为菜单栏，MenuBar），在菜单条上安排有若干个菜单（Menu），每个菜单又包含若干菜单项（MenuItem），每个菜单项对应了一个命令或子菜单项。它们构成一个应用程序的菜单系统。用鼠标或键盘选择对应一个命令的菜单项与选择一个按钮类似。使用菜单系统可方便地向程序发布命令。

在构建一个自己的菜单系统时，可按照菜单系统的层次一步一步地进行。

（1）用类 JMenuBar 创建菜单条

可简单地用 JMenuBar()构造方法类创建一个新菜单条。例如:

```
JMenuBar aMenuBar  = new JMenuBar();
```

（2）用类 JMenu 创建菜单

用类 JMenu 的构造方法来创建菜单，其构造方法有:

- public JMenu()构造一个无文本的菜单。
- public JMenu(String s)用字符串 s 作为文本来构造一个菜单。

例如:

```
JMenu  aMenu = new JMenu("文件"); // 创建“文件”菜单
```

（3）用类 JMenuItem 创建菜单项

类 JMenuItem 的构造方法有:

- public JMenuItem()创建一个菜单项，但不设置文本和图标。
- public JMenuItem(Icon icon)创建一个带图标的菜单项。
- public JMenuItem(String text)创建一个具有指定文本的菜单项。
- public JMenuItem(String text, Icon icon)创建具有文本和图标的菜单项。
- public JMenuItem(String text,int mnemonic)创建具有文本和快捷字母的菜单项。

例如:

```
JMenuItem aMenuItem = new JMenuItem("新建");  // 创建“新建”菜单项
```

（4）将菜单项加入到菜单中，将菜单加入到菜单条中

可用 JMenuBar 类和 JMenu 类的 add()方法完成添加工作。例如:

```
aMenuBar.add(aMenu );
aMenu.add(aMenuItem);
```

另外，可用 addSeparator()方法向菜单添加分隔线。

（5）将菜单条加入容器中

可向实现了 MenuContainer 接口的容器（如框架）加入菜单系统。在框架 JFrame 类中有方法:

```
public void setJMenuBar(JMenuBar menubar)
```

可为框架设置菜单条。例如:

```
JFrame aFrame = new JFrame();
aFrame.setJMenuBar(aMenuBar);
```

（6）处理菜单项选择事件

为了检测对菜单项做出的选择，要监听菜单项 ActionEvent 事件，选择一个菜单项正如选择了一个 JButton 按钮一样。事件处理的相关代码可见例 9.20。

【例 9.20】　使用下拉菜单系统。

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JMenuDemo extends JFrame implements ActionListener{
   JLabel j1 = new JLabel("请选择菜单:",JLabel.CENTER);
   JMenuItem aaMenuItem,baMenuItem;
   JMenuDemo(){
      super("使用 JMenu");
      JMenuBar aMenuBar = new JMenuBar();
      Icon i1 =  new ImageIcon("images/girl.gif");
      Icon i2 =  new ImageIcon("images/boy.gif");
      JMenu aMenu = new JMenu("菜单 A");
      JMenu bMenu = new JMenu("菜单 B");
      JMenuItem aaMenuItem = new JMenuItem("菜单项 AA",i1);
      JMenuItem abMenuItem = new JMenuItem("菜单项 AB",i2);
      JMenuItem baMenuItem = new JMenuItem("菜单项 BA");
      aMenuBar.add(aMenu);
      aMenuBar.add(bMenu);
      aMenu.add(aaMenuItem);
      aMenu.addSeparator();
      aMenu.add(abMenuItem);
      bMenu.add(baMenuItem);
      aaMenuItem.addActionListener(this);
      abMenuItem.addActionListener(this);
      baMenuItem.addActionListener(this);
      setJMenuBar(aMenuBar);
      getContentPane().add(j1,BorderLayout.CENTER);
   }
   public void actionPerformed(ActionEvent e){
      JMenuItem source = (JMenuItem)(e.getSource());
      j1.setText("选择了菜单:"+source.getText());
      j1.setHorizontalAlignment(JLabel.CENTER);
   }
   public static void main(String args[]){
      JFrame frame = new JMenuDemo();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
      frame.pack();
   }
}
```

程序运行时选择了菜单 A 后的情况如图 9.22 所示。

图 9.22 例 9.20 运行结果

9.3.11 对话框

对话框是一种大小不能变化、不能有菜单的容器窗口，对话框不能作为一个应用程序的主框架，而必须包含在其他的容器中。Java 语言提供多种对话框类来支持多种形式的对话框。JOptionPane类支持简单、标准的对话框；JDialog 类支持定制用户自己的对话框；JFileChooser 类支持文件打开、保存对话框；ProgressMonitor 类支持操作进度条控制对话框等。

对话框依赖于框架。当框架撤销时，依赖该框架的对话框也撤销。当框架图标化时，依赖它的对话框也从屏幕上消失。当框架窗口恢复时，依赖框架的对话框又返回屏幕。

下面介绍较简单的JOptionPane和 JFileChooser 类支持的对话框。

1. JOptionPane 对话框

JOptionPane 提供的对话框是模式对话框。当模式对话框显示时，它不允许用户输入到程序的其他的窗口。使用 JOptionPane，可以创建和自定义问题、信息、警告以及错误等几种类型的对话框。JOptionPane 提供标准对话框的布局支持、图标、指定对话框的标题和文本、自定义按钮文本、允许自定义组件的对话框显示、指定对话框在屏幕上的显示位置等特性。

javax.swing.JOptionPane 类继承于 javax.swing.JComponent 类，有七种构造方法，常用的有五种：

- JOptionPane()创建具有测试信息的 JOptionPane 对象。
- JOptionPane(Object message) 创建显示 message 和默认选项的 JOptionPane 对象。
- JOptionPane(Object message, int messageType) 创建以 messageType 类型显示 message 和使用默认选项的 JOptionPane 对象。
- JOptionPane(Object message, int messageType, int optionType)创建显示指定类型信息和指定选项类型的 JOptionPane 对象。
- JOptionPane(Object message, int messageType, int optionType, Icon icon) 创建显示指定类型信息和指定选项类型、图标的 JOptionPane 对象。

要显示简单的模式对话框，可以使用 showMessageDialog()或 showOptionDialog()方法。它们的格式为（showMessageDialog()方法有重载）：

- public static void showMessageDialog(Component parentComponent,Object message, String title,int messageType,Icon icon)throws HeadlessException
- public static int showOptionDialog(Component parentComponent,Object message,String title,int optionType,int messageType,Icon icon,Object[] options,Object initialValue) throws HeadlessException

前者显示一个带一个“Ok”或“确定”按钮的对话框。后者显示自定义的对话框——可以显示具有定制文本的各种按钮，并可包含一个标准的文本信息或组件集合。例如：

```
JOptionPane.showMessageDialog(frame,"请输入一个整数。\n" + "无效输入。 ");
```

showXxxDialog 方法的所有参数和 JOptionPane 构造方法都是标准化的，下面对这些参数进行说明。

- parentComponent 对每个 showXxxDialog 方法的第一个参数总是父组件，必须是一个框架、一个框架中的组件或 null 值。JOptionPane 构造方法不包含这个参数。
- message 这个必须的参数指明要显示的对话框。一般是一个字符串，显示在对话框的一个标签中。
- title 标题。
- optionType 指定出现在对话框底部的按钮集合，可以选择下面 4 个标准集合之一：DEFAULT_OPTION， YES_NO_OPTION， YES_NO_CANCEL_OPTION， OK_CANCEL_OPTION。
- messageType 确定显示在对话框中的图标。从下列值中选择一个：PLAIN_MESSAGE（无 icon），ERROR_MESSAGE，INFORMATION_MESSAGE，WARNING_MESSAGE，QUESTION_MESSAGE。
- icon 指明了在对话框中显示用户定义图标。
- options 进一步指明任选对话框底部的按钮。一般地，对按钮指定一个字符串数组。
- initialValue 指明选择的初始值。

showMessageDialog()和 showOptionDialog()方法返回用户选择的一个整数，整数值为 YES_OPTION、NO_OPTION、CANCEL_OPTION、OK_OPTION 和 CLOSED_OPTION 之一。除了 CLOSED_OPTION 的每一个选项都对应用户单击的按钮，当 CLOSED_OPTION 返回时，说明用户关闭了对话框窗口。

即使用户改变了标准对话框上按钮显示的字符串，返回值仍然为预定义的整数值之一。例如，一个 YES_NO_OPTION 对话框总是返回下列值之一：YES_OPTION、NO_OPTION 或 CLOSED_OPTION。

【例 9.21】 使用对话框。

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JOptionPaneDemo{
   public JFrame aFrame;
   public JButton aButton;
   public JOptionPane anOptionPane;
   static JLabel aLabel;
   public JOptionPaneDemo(){
      aFrame = new JFrame("使用 JOptionPane");
      aButton = new JButton("显示对话框");
      aLabel = new JLabel("您作出的选择是：",JLabel.CENTER);
      aButton.addActionListener(new OptionListener());
      anOptionPane = new JOptionPane();
      aFrame.setSize(300,200);
      aFrame.getContentPane().add(aButton,BorderLayout.NORTH);
```

```
        aFrame.getContentPane().add(aLabel,BorderLayout.SOUTH);
        aFrame.setVisible(true);
        aFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    public class OptionListener implements ActionListener{
        public void actionPerformed(ActionEvent e){
            String[] choices = {"喜欢", "不喜欢"};
            int n = anOptionPane.showOptionDialog(aFrame,"您喜欢音乐吗？","请选择",
                JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE, null,
                choices,choices[0]);
            if(n == 0)JOptionPaneDemo.aLabel.setText("喜欢音乐");
            else JOptionPaneDemo.aLabel.setText("不喜欢音乐");
        }
    }
    public static void main( String[ ] args){
        new JOptionPaneDemo();
    }
}
```

程序运行时显示的对话框如图 9.23 所示。

图 9.23　例 9.21 运行结果

2. JFileChooser 对话框

JFileChooser 类提供对文件的打开、关闭等文件操作的标准对话框。

JFileChooser 类继承于 JComponent 类，其构造方法有：

- JFileChooser()构造一个指向用户默认目录的 JFileChooser 对象。
- JFileChooser(File currentDirectory)构造一个以给定 File 为路径的 JFileChooser 对象。

构造 JFileChooser 对象后，要利用该类的方法 showOpenDialog()或 showSaveDialog()来显示文件打开或文件关闭对话框。它们的格式为：

- public int showOpenDialog(Component parent)throws HeadlessException
- public int showSaveDialog(Component parent)throws HeadlessException

它们的参数都是包含对话框容器的对象，返回值为下面几种情况：

- JFileChooser.CANCEL_OPTION 表示选择了“撤销”按钮。
- JFileChooser.APPROVE_OPTION 表示选择了“打开”或“保存”按钮。
- JFileChooser.ERROR_OPTION 表示出现错误。

在打开或关闭文件对话框中做出选择后，可用 JFileChooser 类的方法 getSelectedFile()返回选取的文件名（File 类的对象）。

【例 9.22】　使用文件打开、关闭对话框（JFileChooser）。将选择的文件名显示到文本区域中。

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.filechooser.*;
public class JFileChooserDemo extends JFrame {
  public JFileChooserDemo() {
    super("使用 JFileChooser");
    final JTextArea ta = new JTextArea(5,20);
    ta.setMargin(new Insets(5,5,5,5));
    ta.setEditable(false);
    JScrollPane sp = new JScrollPane(ta);
    final JFileChooser fc = new JFileChooser();
    JButton openBtn = new JButton("打开文件...");
    openBtn.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent e) {
        int returnVal = fc.showOpenDialog(JFileChooserDemo.this);
        if (returnVal == JFileChooser.APPROVE_OPTION) {
          File file = fc.getSelectedFile();
          ta.append("打开: " + file.getName() + ".\n");
        } else ta.append("取消打开命令.\n");
      }
    });
    JButton saveBtn = new JButton("保存文件...");
    saveBtn.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent e){
        int returnVal = fc.showSaveDialog(JFileChooserDemo.this);
        if (returnVal == JFileChooser.APPROVE_OPTION){
          File file = fc.getSelectedFile();
          ta.append("Saving: " + file.getName() + ".\n");
        } else ta.append("取消保存命令。 \n");
      }
    });
    JPanel buttonPanel = new JPanel();
    buttonPanel.add(openBtn);
    buttonPanel.add(saveBtn);
    openBtn.setNextFocusableComponent(saveBtn);
    saveBtn.setNextFocusableComponent(openBtn);
    Container c = getContentPane();
    c.add(buttonPanel, BorderLayout.NORTH);
```

```
        c.add(sp, BorderLayout.CENTER);
    }
    public static void main(String[] args) {
        JFrame frame = new JFileChooserDemo();
        frame.setDefaultCloseOperation(EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
}
```

程序运行的开始界面如图 9.24 **所示，选择“打开文件”按钮后，即出现如图** 9.25 **所示的打开文件对话框，选择文件后将选择文件名显示到文本区域中。选择“保存文件”按钮，将出现文件保存对话框。**

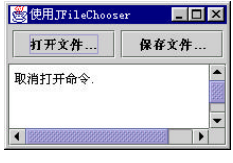

图 9.24　**例** 9.22 **运行结果（一）**

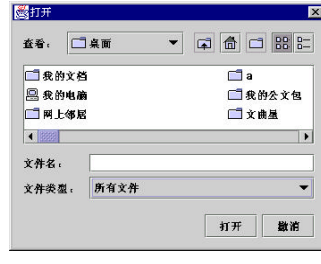

图 9.25　**例** 9.22 **运行结果（二）**

9.4　鼠标和键盘事件

在 GUI **程序中，通常使用鼠标来进行人机交互操作。鼠标的移动、单击、双击等都会引发鼠标事件。为了输入数据、操作命令等，也使用键盘。键盘的按下、释放也会引发键盘事件。下面简单介绍** AWT **的鼠标和键盘事件的处理。**

9.4.1　鼠标事件

处理鼠标事件的程序要实现在 java.awt.event **包中定义的两个接口** MouseListener **和** MouseMotionListener**，在这两个接口中定义了未实现的鼠标事件处理方法。**

接口 MouseListener **中的方法为：**

- public void mousePressed(MouseEvent e)**处理按下鼠标左键。**
- public void mouseClicked(MouseEvent e)**处理鼠标单击。**
- public void mouseReleased(MouseEvent e)**处理鼠标按键释放。**
- public void mouseEntered(MouseEvent e)**处理鼠标进入当前窗口。**
- public void mouseExited(MouseEvent e)**处理鼠标离开当前窗口。**

接口 MouseMotionListener **中的方法为：**

- public void mouseDragged(MouseEvent e)**处理鼠标拖动。**
- public void mouseMoved(MouseEvent e)**处理鼠标移动。**

对应上述接口，对应的注册监听器的方法是 addMouseListener() **和** addMouseMotionListener()**。**

另外，类 MouseEvent **的方法** getX()、getY()**常用来获取鼠标当前所在位置的坐标，它们的格式如下：**

- public int getX()
- public int getY()

【例 9.23**】 通过单击鼠标来画蓝色的圆点。**

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class MouseEventDemo extends JApplet{
   public void init(){
      addMouseListener(new CircleListener());
      setForeground(Color.blue);
      setBackground(Color.white);
   }
}
class CircleListener extends MouseAdapter {
   private int radius = 10;
   public void mousePressed(MouseEvent e) {
      JApplet app = (JApplet)e.getSource();
      Graphics g = app.getGraphics();
      g.fillOval(e.getX()-radius,e.getY()-radius,2*radius,2*radius);
   }
}
```

程序运行结果如图 9.26 **所示。**

图 9.26 **例** 9.23 **运行结果**

9.4.2 键盘事件

处理键盘事件的程序要实现在 java.awt.event **包中定义的接口** KeyListener**，在这个接口中定义了未实现的键盘事件处理方法。键盘事件处理方法为：**

- public void KeyPressed(KeyEvent e)**处理按下键。**
- public void KeyReleased(KeyEvent e)**处理松开键。**
- public void KeyTyped(KeyEvent e)**处理敲击键盘。**

为识别引发键盘事件的按键，常用到 KeyEvent **类的如下方法：**

- public char getKeyChar()**返回该事件中键的字符。例如，**shift +<a>**按键事件返回值为“**A**”。**
- public static String getKeyText(int keyCode)**返回描述键代码的字符串，例如“**HOME**”、“**F1**”或“**A**”等。**

【例 9.24**】 综合鼠标事件和键盘事件处理的程序，模拟一个电子白板，可以用鼠标在上面绘画，可用键盘在上面写字。**

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class MouseAndKeyDemo extends JApplet {
  protected int lastX = 0, lastY = 0;
  public void init() {
    setBackground(Color.white);
    setForeground(Color.blue);
    addMouseListener(new PositionRecorder());
    addMouseMotionListener(new LineDrawer());
    addKeyListener(new CharDrawer());
  }
  protected void record(int x, int y) {
    lastX = x;
    lastY = y;
  }
  private class PositionRecorder extends MouseAdapter {
    public void mouseEntered(MouseEvent e) {
      requestFocus();
      record(e.getX(), e.getY());
    }
    public void mousePressed(MouseEvent e) {
      record(e.getX(), e.getY());
    }
  }
  private class LineDrawer extends MouseMotionAdapter {
```

```
        public void mouseDragged(MouseEvent e) {
            int x = e.getX();
            int y = e.getY();
            Graphics g = getGraphics();
            g.drawLine(lastX, lastY, x, y);
            record(x, y);
        }
    }
    private class CharDrawer extends KeyAdapter {
        public void keyTyped(KeyEvent event) {
            String s = String.valueOf(event.getKeyChar());
            getGraphics().drawString(s, lastX, lastY);
            record(lastX + 11, lastY);
        }
    }
}
```

程序中，在类 MouseAndKeyDemo **中定义了三个私有嵌套类，两个类对鼠标事件进行处理，一个类对键盘事件进行处理。程序的运行情况如图** 9.27 **所示。图中的线用鼠标随手画出，文字用键盘输入。**

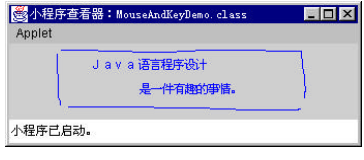

图 9.27　**例** 9.24 **的运行显示**

习　题　九

9.1　**哪一种布局管理器是从上到下、从左到右安排组件，当移动到下一行时是居中的？选择一个正确的答案。**

（1）BorderLayout　（2）FlowLayout　（3）GridLayout　（4）CardLayout　（5）GridBagLayout

9.2　**下面一行代码做什么？选择一个正确的答案。**

```
JTextField tf = new JTextField(30);
```

（1）**代码不合法。在** JTextField **中，没有这样的构造方法。**

（2）**创建了一个** 30 **行的** JTextField **对象，但未进行初始化，它是空的。**

（3）**创建了一个** 30 **列的** JTextField **对象，但未进行初始化，它是空的。**

（4）**创建一个有** 30 **行文本的** JTextField **对象。**

（5）创建一个有 30 列文本的 JTextField 对象。

9.3 如何在一个容器的底部放 3 个组件？选择一个正确的答案。

（1）设置容器的布局管理器为 BorderLayout，并添加每个组件到容器的南部。

（2）设置容器的布局管理器为 FlowLayout，并添加每个组件到容器。

（3）设置容器的布局管理器为 BorderLayout，并添加每个组件到使用 FlowLayout 的另一容器，然后将该容器添加到第一个容器的南部。

（4）设置容器的布局管理器为 GridLayout，并添加每个组件到容器。

（5）不使用布局管理器，将每个组件添加到容器。

9.4 编程：创建有一个文本框和三个按钮的小程序。当按下每个按钮时，使不同的文字显示在文本框中。

9.5 编程：增加一个复选框到题 9.4 创建的程序中，使得选中或不选中复选框时插入的文字不同。

9.6 编程：编写一个 Application，接受用户输入的账号和密码，给三次输入机会。

9.7 编程：编写成人标准身高和体重互查的程序。身高和体重在两个不同的文本框中输入，要求输入一个。输入身高则输出体重，输入体重则输出身高。用一个按钮启动互查。互查的公式为：

$$体重 = 身高 - 100$$

9.8 编程：创建三个 JSlider 滚动条和一个 JLabel 标签对象，滚动条用来调整红、绿、蓝三色的比例，每个滚动条加标题边界，设置刻度（自定）和标注。当拖动滑块修改三色比例时，变化标签的背景色。

9.9 编程：显示前 n 个 Fibonacci 数的 GUI 程序。n 在一个文本框中输入，选择一个按钮开始计算并显示结果，结果输出到一个 JTextArea 中。

9.10 编程：创建一个文本框、三个单选按钮、一个标签和一个按钮，文本框用来输入自然数 n，根据选择单选按钮的不同，分别计算：

$1 + 2 + \cdots + n$ 或 $1 \times 2 \times \cdots \times n$ 或 $1 + 1/2 + 1/3 + \cdots + 1/n$

计算结果在标签中显示。

第 10 章　Java 的多线程

到目前为止，大多数程序设计时习惯上考虑该程序如何从头至尾顺序执行各项任务的设计方法，即一个程序只有一条执行路线。但现实世界中的很多过程都是同时发生的，对应这种情况，可编写有多条执行路径的程序，使得程序能够同时执行多个任务（并行）。

10.1　多线程概述

10.1.1　多线程的概念

Java 语言支持程序并行执行的多线程编程，实现了一般传统语言难以实现的某些功能，如实时交互能力、多 CPU 的支持等。

线程是程序中的一条执行路径。多线程是指程序中包含多条执行路径。在一个程序中可以同时运行多个不同的线程来执行不同的任务，即允许单个程序创建多个并行执行的线程来完成各自的任务。浏览器程序就是一个多线程的例子，在浏览器中可以在下载 Java 小程序或图像的同时滚动页面，在访问新页面时，播放动画和声音，打印文件等。

多线程编程的含义是将程序任务分成几个并行的子任务。例如，在网络编程中，会发现很多功能是可以并发执行的。设某个程序需要进行网络传输和用户输入的工作，在单线程的情况下，执行读网络数据或等待用户输入，只有等读完数据或用户输入完成才能进行其他的处理。若网络传输速度不快或用户输入速度较慢，程序的执行效率是很低的，程序大部分时间在等待，在等待的过程中程序什么也不能干。在多线程的情况下，可以用两个独立的线程去完成这些功能，而不影响正常的显示或其他功能。多线程是与单线程比较而言的，在读网络数据和等待用户输入时有很多时间处于等待状态，多线程利用这个特点将任务分成多个并发任务后就可以提高系统执行效率。

在多线程程序中，多个线程可共享一块内存区域和资源。例如，当一个线程改变了所属应用程序的变量时，则其他线程下次访问该变量时将看到这种改变。线程间可利用共享特性来实现数据交换、实时通信等。

总之，多线程编程能提高程序的运行效率，增强程序的交互性，降低复杂任务的难度，也符合人们的自然习惯。但也要认识到线程本身可能影响系统性能的不利方面和多线程程序需要注意的地方，以正确使用多线程。

线程需要占用内存，也需要 CPU 时间跟踪线程。当线程间有共享资源的情况时，要注意解决竞用共享资源的问题，防止死锁的情况发生等。

Java 的多线程运行与操作系统紧密相关，操作系统的线程执行效率越高，程序的执行效率就越高。

Java 的线程是通过 java.lang.Thread 类来实现的，在该类中封装了虚拟的 CPU。

10.1.2 Java 对多线程的支持

Java 语言里，线程表现为线程类，Thread 线程类封装了所有需要的线程操作控制。在设计程序时，必须很清晰地区分开线程对象和运行线程，可以将线程对象看做是运行线程的控制面板。在线程对象里有很多方法来控制一个线程是否运行、睡眠、挂起或停止。线程类是控制线程行为的惟一的手段。一旦一个 Java 程序启动后，就已经有一个线程在运行。可通过调用 Thread.currentThread 方法来查看当前运行的是哪一个线程。

【例 10.1】 演示如何操纵当前线程。

```
class testthread{
  public static void main(String args[]){
    Thread t = Thread.currentThread();
    t.setName("单线程");    // 对当前线程取名为"单线程"
    t.setPriority(8);// 设置线程优先级为 8，最高为 10，最低为 1，默认为 5
    System.out.println("The running thread: " + t); // 显示线程信息
    try{
      for(int i=0;i<3;i++){
        System.out.println("Sleep time " + i);
        Thread.sleep(100);    // 睡眠 100 毫秒
      }
    }catch(InterruptedException e){  // 捕获 sleep()方法的异常
      System.out.println("thread has wrong");
    }
  }
}
```

程序执行结果如下：

```
The running thread: Thread[单线程,8,main]
Sleep time 0
Sleep time 1
Sleep time 2
```

在 Java 语言中，若创建了一个 Thread 类（多数情况是其子类）的实例对象，就形成了一个新的线程。运行线程的执行代码书写在 run()方法中，通过调用线程对象的不同方法，可以控制线程的启动、暂时挂起、终止等。

下面是一些常用线程类的方法。

（1）类方法

以下是 Thread 的类（静态）方法，即可以直接从 Thread 类调用：

- currentThread() 返回正在运行的 Thread 对象。
- yield() 停止运行当前线程，将 CPU 控制权主动移交到下一个可运行线程。
- sleep(int n) 让当前线程睡眠 n 毫秒，n 毫秒后线程可以再次运行。

（2）实例方法

以下方法必须用 Thread 的实例对象来调用：

- start() 为本线程建立一个执行环境，然后调用本线程的 run()方法。
- run() 在其中书写运行本线程的将要执行的代码，也是 Runnable 接口的惟一方法。当一个线程初始化后，由 start()方法来调用它，一旦 run()方法返回，本线程也就终止了。
- stop() 让某线程马上终止，系统将删除本线程的执行环境。
- suspend() 将线程挂起，暂停运行，但系统不破坏线程的执行环境，可以用 resume()来恢复本线程的执行。
- resume() 恢复被挂起的线程进入可运行状态。
- setPriority(int p) 给线程设置优先级 $1 \leqslant p \leqslant 10$。
- getPriority() 返回线程的优先级。
- setName(String name) 赋予线程一个名字 name。
- getName() 取得由 setName()方法设置的线程名字的字符串。
- wait(long timeout) 停止当前线程，直到另外的线程对这个对象使用 notify()或 notifyAll()方法。
- notify()或 notifyAll() 唤醒指定对象的一个或所有线程。

10.1.3 线程的状态和生命周期

一个线程从创建、启动到终止期间的任何时刻，总是处于下面五个状态中的某个状态。

1，创建状态

用 new 运算符创建一个 Thread 类或子类的实例对象时，形成的新线程就进入创建状态，但此时还未对这个线程分配任何资源，没有真正执行它。

2，可运行状态

在创建线程后，若要执行它，系统要对这个线程进行登记并为它分配系统资源，这些工作由 start()启动方法来完成。线程启动后，将进入线程队列排队等待 CPU 时间片，成为可运行状态（或称为就绪状态）。此时线程已经具备了运行的条件，一旦它获得 CPU 等资源时就可以脱离开创建它的主线程而独立运行。

3，运行状态

当可运行状态的线程被调度并获得 CPU 等资源时，便进入运行状态。

4，阻塞状态

由于人为或系统的原因，线程必须停止运行，以后还可以恢复运行的状态称为阻塞状态。发生以下几种情况之一后，线程进入阻塞状态：

（1）调用了该线程的 sleep()休眠方法。

（2）该线程正在等待 I/O 操作完成。

（3）调用了 wait()等待方法。

（4）调用了 suspend()挂起方法。

5，终止状态

运行 run()方法完成后调用 stop()或 destroy()方法就进入线程的终止态（或称为死亡状态）。处于这种状态的线程不具有继续运行的能力。

由于线程是一个动态的概念，所以它有一个从创建产生到终止消亡的生命周期，生命周期的中期是可运行态、运行态和阻塞态的顺序轮流转化。这些转化及线程生命周期的演进是由系统运行的状态、同时存在的其他线程和线程本身的算法所共同决定的。

10.1.4 线程的调度和优先级

处于可运行状态的线程进入线程队列排队等待 CPU 等资源时，同一时刻在队列中的线程可能有多个，它们完成各自任务的轻重缓急程度是不同的。为了体现上述差别，多线程系统会给每个线程自动分配一个线程的优先级。任务较重要或紧急的线程，分配较高的优先级，在可运行态的线程队列中就往前排；否则，就分配较低的优先级。优先级低的线程只能等到优先级高的线程执行完后才被执行。对于优先级相同的线程，则遵循队列的“先进先出”原则，即先到的线程先获得系统资源来运行。

在 Java 语言中，对一个新建的线程，系统会分配一个默认的线程优先级：继承创建这个线程的主线程的优先级（一般为普通优先级）。Thread 类也提供了方法 setPriority()来修改线程的优先级。该方法的参数一般可用 Thread 类的优先级静态常量：

PRIORITY.NORM_PRIORITY 表示普通优先级（5）
PRIORITY.MIN_PRIORITY 表示普通优先级（1）
PRIORITY.MAX_PRIORITY 表示普通优先级（10）

当一个在可运行状态队列中排队的线程被分配到 CPU 等资源而进入运行状态后，这个线程就称为是被“调度”或被线程调度管理器选中了。线程调度管理器负责管理线程排队和 CPU 等资源在线程间的分配。

10.2 多线程的实现方法与控制

10.2.1 多线程的实现方法

编写多线程程序需要创建线程，线程的创建包括定义线程体和创建线程对象两个方面的内容。线程的行为由线程体决定，线程体是由 run()方法定义的，运行系统通过调用 run()方法实现线程的具体行为。

可以通过继承 Thread 类或实现 Runnable 接口这两种途径来构造自己的 run()方法。

1，继承 Thread 类

可通过继承 Thread 类并重写其中的 run()方法来定义线程体以实现线程的具体行为，然后创建该子类的对象以创建线程。

在继承 Thread 类的子类 ThreadSubclassName 中重写 run()方法来定义线程体的一般格式为：

```
public class ThreadSubclassName extends Thread{
  public ThreadSubclassName(){
    ..... // 编写子类的构造方法，可默认
  }
  public void run(){
    ..... // 编写自己的线程代码
  }
}
```

用定义的线程子类 ThreadSubclassName 创建线程对象的一般格式为：

```
ThreadSubclassName ThreadObject = new ThreadSubclassName();
```

然后，就可启动该线程对象表示的线程：

```
ThreadObject.start();                    //启动线程
```

在下面的例 10.2 中应用继承类 Thread 的方法创建了三个单独的线程，它们分别打印自己的“Hello World!”。

【例 10.2】 应用继承类 Thread 的方法实现多线程的程序。

```
class ThreadDemo extends Thread{
  private String whoami;
  private int delay;
  public ThreadDemo(String s,int d){
    whoami=s;
    delay=d;
  }
  public void run(){
    try{
      sleep(delay);
    }catch(InterruptedException e){
    }
    System.out.println("Hello World!" + whoami + " " + delay);
  }
}
public class MultiThread{
  public static void  main(String args[]){
    ThreadDemo t1,t2,t3;
    t1 = new ThreadDemo("Thread1",(int)(Math.random()*2000));
    t2 = new ThreadDemo("Thread2",(int)(Math.random()*2000));
    t3 = new ThreadDemo("Thread3",(int)(Math.random()*2000));
    t1.start();
    t2.start();
```

```
        t3.start();
    }
}
```

程序的一次运行结果如下：

```
Hello World!Thread3 365
Hello World!Thread1 1361
Hello World!Thread2 1417
```

程序启动时总是调用 main()**方法执行主线程，因此** main()**是创建和启动线程的地方。程序中，在创建三个线程时，传送了线程的名称和线程在打印信息之前的延时时间（作为** sleep()**方法的参数）。因为在这里直接控制线程，必须用** start()**直接启动它们。**

【例 10.3**】　应用继承类** Thread **的方法实现多线程的程序。在本例中应用了线程组。**

```
class MyThread extends Thread {
    public void run(){
        try {
            for(int i = 0; i < 3; i++) {
                int msec = (int)(300 + 500 * Math.random());
                Thread.sleep(msec);
                System.out.println(getName()+" x");
            }
        }catch(InterruptedException ex){
            ex.printStackTrace();
        }
    }
}
class ThreadsDemo {
    private static int NUMTHREADS = 3;
    public static void main(String args[]) {
        // 创建线程
        MyThread threads[] = new MyThread[NUMTHREADS];
        for(int i = 0; i < NUMTHREADS; i++) {
            threads[i] = new MyThread();
        }
        // 启动线程
        for(int i = 0; i < NUMTHREADS; i++) {
            threads[i].start();
        }
        // 等待线程完成
        for(int i = 0; i < NUMTHREADS; i++) {
```

```
            try {
               threads[i].join();
            }catch(Exception e){
               e.printStackTrace();
            }
         }
         System.out.println("所有线程已经完成。");
      }
   }
```

程序的一次运行结果如下：

```
Thread-3 x
Thread-1 x
Thread-2 x
Thread-3 x
Thread-1 x
Thread-2 x
Thread-3 x
Thread-2 x
Thread-1 x
所有线程已经完成。
```

2，实现 Runnable 接口

编写多线程程序的另一种方法是实现 Runnable **接口。在一个类中实现** Runnable **接口（以后称实现** Runnable **接口的类为** Runnable **类），并在该类中定义** run()**方法，然后用带有** Runnable **参数的** Thread **类构造方法创建线程。**

创建线程对象可用下面的两个步骤来完成。

- **生成** Runnable **类** ClassName **的对象。**

```
ClassName RunnableObject = new ClassName();
```

- **用带有** Runnable **参数的** Thread **类构造方法创建线程对象。新创建的线程的指针将指向** Runnable **类的实例。用该** Runnable **类的实例为线程提供** run()**方法——线程体。**

```
Thread ThreadObject = new Thread(RunnableObject);
```

然后，就可启动线程对象 ThreadObject **表示的线程：**

```
ThreadObject.start();
```

在 Thread **类中带有** Runnable **接口的构造方法有：**

- public Thread(Runnable target);
- public Thread(Runnable target, String name);
- public Thread(String name);

- public Thread(ThreadGroup group，Runnable target);
- public Thread(ThreadGroup group，Runnable target，String name);

其中，参数 Runnable target **表示该线程执行时运行** target **的** run()**方法，** String name **以指定名字构造线程，** ThreadGroup group **表示创建线程组。**

【例 10.4**】 用** Runnable **接口实现的多线程。**

```
class TwoThread implements Runnable{
  TwoThread(){
    Thread t1 = Thread.currentThread();
    t1.setName("第一主线程");
    System.out.println("正在运行的线程：" + t1);
    Thread t2 = new Thread(this,"第二线程");
    System.out.println("创建第二线程");
    t2.start();
    try{
      System.out.println("第一线程休眠");
      Thread.sleep(3000);
    }catch(InterruptedException e){
      System.out.println("第一线程有错");
    }
    System.out.println("第一线程退出");
  }
  public void run(){
    try{
      for(int i = 0;i < 5;i++){
        System.out.println("第二线程的休眠时间：" + i);
        Thread.sleep(1000);
      }
    }catch(InterruptedException e){
      System.out.println("线程有错");
    }
    System.out.println("第二线程退出");
  }
  public static void main(String args[]){
    new TwoThread();
  }
}
```

程序运行结果如下：

```
正在运行的线程：Thread[第一主线程,5,main]
创建第二线程
```

第一线程休眠
第二线程的休眠时间：0
第二线程的休眠时间：1
第二线程的休眠时间：2
第一线程退出
第二线程的休眠时间：3
第二线程的休眠时间：4
第二线程退出

main 线程用 new Thread(this,"第二线程")创建了一个 Thread 对象，通过传递第一个参数来标明新线程调用 this 对象的 run()方法。然后调用 start()方法，它将使线程从 run()方法开始执行。

3. 多线程的 Applet

如果想让线程作为 Java 小程序的一部分而运行，通常采用实现 Runnable 接口的方法。这是因为 Java 语言不支持多继承，即子类只能有一个父类，而小程序必须继承 Applet 类或 JApplet 类，此时可用前面介绍的实现 Runnable 接口的方法来实现 Applet 的多线程。

也可用下列方法来编写多线程的 Applet 程序：

（1）定义一线程类，实现线程体。

（2）在 Java 小程序中生成该类的对象并启动运行。

例如：

```
class MyThread extends Thread{       // 生成线程类
                    M
  public void run(){                 // 定义线程体
                    M
  }
                    M
}
public class MyApplet extends Applet{
  MyThread thread;
  public void init(){
    thread = new MyThread();         // 生成线程类的实例
    thread.start();                  // 启动线程
  }
  pubic void start(){
    thread.resume();
  }
  pubic void stop(){
    thread.suspend();
  }
}
```

10.2.2 多线程的控制

1，操作线程

如果创建线程正常，可在线程的 run()方法里控制线程。一旦进入 run()方法，便可执行里面的任何程序。run()好像 main()一样，一旦 run()执行完，这个线程也就结束了。若想推迟一个线程的执行，应使用 sleep(delay)休眠方法，delay 是休眠的时间（毫秒）。当线程休眠时并不占用系统资源，其他线程可继续工作。

2，暂停一个线程

经常需要挂起一个线程而不指定多少时间。例如，若创建了一个含有动画线程的小程序，也许让用户暂停动画直到想恢复为止。若不想将动画线程扔掉，但想让它停止，像这种类似的线程可用 suspend()方法来挂起。

用 suspend()方法挂起线程并不永久地停止线程，这可用 resume()方法重新激活线程。

3，停止一个线程

线程的最后一个控制是停止方法 stop()，用它可以停止线程的执行。

注意：这并没有消灭这个线程，但它停止了线程的执行，并且这个线程不能用 start()方法重新启动。一般不用 stop()停止一个线程，只简单地让它执行完而已。很多复杂的线程程序将需要控制每一个线程，在这种情况下会用到 stop()方法。如果需要，可以测试线程是否被激活。一个线程已经启动而且没有停止被认为是激活的。如果线程 t1 是激活的，t1.isAlive()将返回 true。

10.3 多线程的互斥

程序中的多个线程一般是独立运行的，各个线程有自己的数据和方法。但有时需要在多个线程之间共享一些资源对象，这些资源对象的操作在某些时候必须要在线程间很好地协调，以保证它们的正确使用。不考虑协调性，就可能产生错误。

下面以两个线程共享一个堆栈的情况来说明在线程之间协调对共享资源操作的重要性。

堆栈是内存中的一段存储区域，它的一端固定，一端活动，活动端称为栈顶。存取数据均在栈顶进行，并遵从“先进后出（FILO）”（或“后进先出（LIFO）”）的原则。对每个堆栈有专门的变量保存栈顶的位置信息。数据进栈涉及两个操作：变动栈顶和在栈顶存入数据。数据出栈也涉及两个动作：取出栈顶数据和变动栈顶。

考虑一种特殊的情况，若线程 A 对一个栈进行进栈操作，仅变动了栈顶，就因为系统线程调度的原因暂时挂起，启动了共享这个堆栈的线程 B。该线程进行出栈操作，因为此前线程 A 还未在栈顶存入数据，将导致线程 B 不能从栈顶取出数据，发生堆栈操作错误。错误原因是分离了线程 A 的进栈操作的两个动作，破坏了进栈操作的完整性。

在 Java 语言中，为保证线程对共享资源操作的完整性，用关键字 synchronized 为共享资源加锁来解决这个问题。这个锁使得共享资源对线程是互斥操作的，称为互斥锁。

synchronized 可修饰一个代码块或一个方法，使修饰对象在任一时刻只能有一个线程访

问，从而提供了程序的异步执行功能。使用 synchronized 的形式为：

```
synchronized( this ){.....} // 修饰一个代码块
synchronized methodName( parameters ){  // 修饰一个方法
.....
}
```

使用 synchronized 的实例见下一节。

10.4 多线程的同步

因为多线程提供了程序的异步执行的功能，所以在必要时还必须提供一种同步机制。同步也是一种各线程间对共享资源使用的协调。例如，对异步操作中所用的堆栈操作的情况来说，设线程 A 和 B 分别对堆栈进行进、出栈的操作。若线程 A 锁住共享堆栈，一直进栈或进栈快，将导致堆栈溢出（满）。若线程 B 锁住共享堆栈，一直出栈或出栈快，将导致栈空。因此，必须考虑遇到堆栈满或空时，怎样使得线程能够自动避免错误操作的问题。

实现同步需要在这些线程之间相互通信。Java 提供了方法 wait()和 notify()等来使线程之间相互交谈。一个线程可以进入某一个对象的 synchronized 方法进入等待状态，直到其他线程显式地将它唤醒。可以有多个线程进入同一个方法并等待同一个唤醒消息。

当堆栈空时，应该让线程 B 释放堆栈的互斥锁（调用 wait()方法），使其进入该锁的等待队列，等待线程 A 进行进栈操作。

同理，当堆栈满时，应该让线程 A 释放堆栈的互斥锁（调用 wait()方法），使其进入该锁的等待队列，等待线程 B 进行出栈操作。

当堆栈正常时，调用 notify()或 notifyAll()方法唤醒线程恢复运行。

线程同步的一个重要特点是它们之间可以互相通信。可以设计线程使用共享对象，每个线程都可以独立操作共享对象。典型的线程间通信、同步执行建立在生产者-消费者模型上：一个线程生产，另一个线程消费。

下面是一个简单的“HotDog（热狗）”生产者和相应的消费者模型的例子。生产者生产 6 个热狗（字母），放在可放 2 个热狗的盘子（缓冲区）里，盘子满就暂停生产；消费者吃盘子中的热狗，吃光则等待生产。

【例 10.5】 “HotDog（热狗）”生产者和相应的消费者模型。

```
class HotDogPAC{
   public static void main(String args[]){
      HotDog h = new HotDog();
      Producer p1 = new Producer(h);
      Consumer c1 = new Consumer(h);
      p1.start();
      c1.start();
   }
}
class HotDog{
```

```
    private char buffer[] = new char[2];
    private int next = 0;
    private boolean isFull = false;
    private boolean isEmpty = true;
    public synchronized char eat(){
        while(isEmpty == true){
            try{
                wait();
            }catch(InterruptedException e){
        }
    }
    //吃热狗，减去热狗的数量
    next--;
    // 是否为最后一个热狗?
    if(next==0){
        isEmpty=true;
    }
    // 刚吃一个热狗，盘子不满了
    isFull = false;
    notify();
    return(buffer[next]);
}
// 将生产的热狗放到盘子里(buffer)
public synchronized void add(char c){
// 等待盘子有空
    while(isFull == true){
        try{
            wait();   // 盘子满暂停生产
        }catch (InterruptedException e){
        }
    }
    // 将热狗放到盘子里
    buffer[next] = c;
    // 盘子里可以放两个热狗，盘子还有空吗？
    next ++;
    // 盘子满了
    if(next == 2){
        isFull = true;
    }
    isEmpty = false;
```

```
        notify();
    }
    }
    class Consumer extends Thread{
        private HotDog hotdog;
            public Consumer(HotDog h){
            hotdog = h;
        }
        public void run(){
            char c;
            // 要吃六个热狗
            for(int i=0;i<6;i++){
                c = hotdog.eat();
                // 打印输出吃掉的 HotDog 的一个字母
                System.out.println("吃热狗: " + c);
                try{
                    sleep((int)(Math.random()*2000));
                }catch(InterruptedException e){
                }
            }
        }
    }
    class Producer extends Thread{
        private HotDog hotdog;
        private String alphabet = "HotDog";
        public Producer(HotDog h){
            hotdog = h;
        }
        public void run(){
            char c;
            // 要生产六个热狗
            for(int i=0;i<6;i++){
                c = alphabet.charAt(i);
                hotdog.add(c);
                System.out.println("生产热狗 ：" + c +" ，放到盘子中。");
                try{
                    sleep((int)(Math.random()*1000));
                }catch(InterruptedException e){
                }
            }
```

```
    }
}
```

程序的一次运行结果如下：

```
生产热狗 ：H ，放到盘子中
吃热狗: H
生产热狗 ：o ，放到盘子中
生产热狗 ：t ，放到盘子中
吃热狗: t
生产热狗 ：D ，放到盘子中
吃热狗: D
生产热狗 ：o ，放到盘子中
吃热狗: o
生产热狗 ：g ，放到盘子中
吃热狗: g
吃热狗: o
```

程序中 HotDog **类执行监视两个线程之间传输信息的功能。监视是多线程中不可缺少的一部分，因为它保持了通信的流动。**

HotDog **类包含两个重要特征：数据成员** buffer[]**是私有的，方法** add()**和** eat()**是公有的。**

生产者-消费者模型程序经常用来实现远程监视功能，它让消费者看到生产者同用户的交互或同系统其他部分的交互。例如，在网络中，一组生产者线程可以在很多工作站上运行。生产者可以打印文档，文档打印后，一个标志将保存下来。一个（或多个）消费者将保存标志并在晚上报告白天打印活动的情况。另外还有例子，在一个工作站上分出几个独立的窗口，一个窗口用做用户输入（生产者），另一个窗口做出对输入的反应（消费者）。

10.5 多线程的应用

多线程在网络和 GUI **设计中都有较多的应用，一些系统开销很大的并发进程程序也可以修改为多线程程序，以节省系统资源。下面是一个在小程序中使用多线程的例子。**

【例 10.6**】 一个** Java **小程序。在屏幕上显示时间，每隔一秒钟刷新一次。为使小程序不影响其他程序的运行，使用了多线程。**

```
import java.awt.*;
import java.applet.*;
import java.util.Date;
public class Clock extends Applet implements Runnable{
   Thread clockThread;
   Font font;
   public void init(){
      font=new Font("TimesRoman",Font.BOLD,64);
   }
```

```
    public void start(){
      if (clockThread==null){
        clockThread = new Thread(this,"Show time");
        clockThread.start();
      }
    }
    public void run(){
      while (clockThread!=null) {
        repaint();
        try {
          clockThread.sleep(1000);
        }catch (InterruptedException e){
        }
      }
    }
    public void paint(Graphics g){
      Date now = new Date();
      g.setFont(font);
      g.setColor(Color.red);
      g.drawString(now.getHours() + ":" + now.getMinutes() + ":"    +
            now.getSeconds(),5,50);
    }
    public void stop(){
      clockThread.stop();
    }
  }
```

程序运行结果如图 10.1 **所示。**

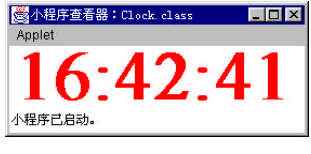

图 10.1　**例** 10.6 **的运行情况**

习　题　十

10.1　**在** Java **语言中多线程程序设计的两种方法是**_____**和**_____**。**

10.2　**实现** Runnable **接口的多线程设计方法中，**Thread **类构造方法的参数应为**_____**。**

10.3　在 Java 语言中，线程的实现部分（线程体）应书写在方法_____中。

10.4　在 Java 语言中，实现线程的异步执行要用到关键字_____。

10.5　在 Java 语言中，实现线程间的通信可用_____和_____方法。

10.6　编程：用继承 Thread 类的方法实现一个多线程程序，该程序先后启动三个线程，每个线程首先打印出一条线程创建信息，然后休眠一个随机时间，最后打印出线程结束信息退出。

10.7　编程：用实现 Runnable 接口的方法重做上题。

10.8　编程：在一个线程中求 100 以内的素数，求出一个素数后休眠一个随机时间。在另一个线程中求水仙花数，求出一个水仙花数后也休眠一个随机时间。输出数据时应有提示，指明是哪个线程输出的数据。

10.9　下列程序的输出是什么？

```
class SychTest1{
   private int x;
   private int y;
   public void setX(int i){ x = i;}
   public void setY(int i){ y = i;}
   public synchronized void setXY(int i){
      setX(i);
      setY(i);
   }
   public synchronized boolean check(){
      return  x!=y;
   }
}
public class SychTest{
   public static void main(String args[]){
      SychTest1 s = new SychTest1();
      s.setX(5);
      s.setY(6);
      System.out.println(s.check());
   }
}
```

10.10　下列程序的输出是什么？

```
public class X implements Runnable{
   private int x;
   private int y;
   public static void main(String[] args){
      X that = new X();
      (new Thread(that)).start();
```

```
        (new Thread(that)).start();
    }
    public synchronized void run(){
        for(;;){
            x ++;
            y ++;
            System.out.println("x=" +  x + ",y = "+ y);
        }
    }
}
```

第 11 章　Java 多媒体技术

最近几年来，随着计算机硬件技术的飞速发展，需要计算机硬件支撑的计算机多媒体技术如图形图像处理、动画、声音处理等也得到迅猛的发展。作为 20 世纪 90 年代的计算机语言，Java 在多媒体技术的支持方面一直在不断发展之中。Java 语言提供的多媒体功能效果好且使用很灵活、方便。在 Web 页面设计中，正由于在 Java 动画中灵活地运用图像和声音媒体，才使得 Web 页面更具魅力。本章介绍 Java 应用程序和小程序在图像处理、动画和播放声音等方面的应用。

11.1　图像

Java 支持的图像文件格式有 GIF 和 JPEG 等。下面分别简单介绍在 Java Applet 和应用程序中显示图形图像的方法和步骤。

1．在 Applet 中显示图像

（1）图像的装载

要显示图像，先要进行图像的装载。在 Applet 类中提供了 getImage()方法来将指定路径的图像文件装载到 Applet 中。由于 Java 语言是面向网络的，因此文件的存储位置不局限于本地机器的磁盘。大部分情况是存取网络中 Web 服务器上的图像文件。getImage()方法的声明如下：

- public Image getImage(URL url)获得绝对 URL 地址指定的 image 对象。
- public Image getImage(URL url,String name)　获得绝对 URL 地址指定的 image 对象，name 指明相对于 URL 的 image 图像文件名。

其中，url 参数经常使用 getCodeBase()方法和 getDocument()方法。例如：

```
Image im1 = getImage(getCodeBase(),"images/dog.gif");
Image im2 = getImage(getDocument(),"images/dog.gif");
Image im3 = getImage("c:/images/cup.gif");
```

getImage()方法在调用后立即返回，它不检查图像文件是否存在，图像的真正装载要到第一次显示时才发生。若需要知道图像的装载进程，可使用 AWT 提供的两种跟踪图像装载的方法：使用 MediaTracker 类和使用 ImageObserver 接口。

使用 MediaTracker 类的方法需创建一个 MediaTracker 类的对象，用它来跟踪一个或多个图像的装载，就可以在程序中获得图像的状态信息。

使用 ImageObserver 接口的方法允许更进一步跟踪图像的装载。Component 类就使用 ImageObserver 接口在图像装载时重绘它的对象。实现 ImageObserver 接口需重写 imageUpdate()方法。

（2）图像的显示

在获得 Image 对象后，可用 Graphics 类的 drawImage()方法显示图像。drawImage()方法有 6 种不同的重载形式，常用的声明形式为：

- public abstract boolean drawImage(Image img,int x,int y,ImageObserver observer)在点坐标（*x*,*y*）处（图像位于该点的右下方）将图像文件 img 按原样大小显示出来。
- public abstract boolean drawImage(Image img,int x,int y,int width,int height, ImageObserver observer)在坐标（*x*,*y*）处将图像文件 img 按 width 和 height 指定的大小显示出来，这种格式能够放大或缩小图像。
- public abstract boolean drawImage(Image img,int dx1,int dy1,int dx2,int dy2,int sx1,int sy1,int sx2,int sy2,ImageObserver observer)从源图上截取以点（*sx*1,*sy*1）和点（*sx*2,*sy*2）为顶点的矩形块显示到点（*dx*1,*dy*1）和点（*dx*2,*dy*2）为顶点的矩形区域内。这种格式能够裁剪图像。

drawImage()方法在显示完图像的信息后就会返回。

若需要使用裁剪效果，也可以使用 Graphics 类的 setClip()方法或 clipRect()方法。这些方法的声明如下：

- public abstract void setClip(int x,int y,int width,int height)
- public abstract void clipRect(int x,int y,int width,int height)

它们的功能均为以坐标（*x*,*y*）为左上角，大小为 width 和 height 的矩形设置裁剪区域。

【例 11.1】 在 Applet 中装载并显示图像。

```
import java.applet.Applet;
import java.awt.*;
public class ImageDemo extends Applet {
   private Image im;
   public void init() {
      im = getImage(getCodeBase(),"images/dog.gif");
   }
   public void paint(Graphics g) {
      g.drawImage(im, 0, 0, this);
   }
}
```

图 11.1 例 11.1 运行结果

程序运行时，需要在 Applet 程序所在目录的下一级目录 images 中存储有 dog.gif 文件，程序的运行结果如图 11.1 所示。

2. 在应用程序中显示图像

在 Java 语言中，也可以在应用程序中显示图像，但显示方法与 Applet 有所不同。

在应用程序中，装载图像需要使用 Toolkit 类来实现。一般通过 Toolkit 类的 getDefaultToolkit()方法或通过调用 Component 类的 getToolkit()方法来获得一个 Toolkit 对象。这两个方法的声明如下：

- public static Toolkit getDefaultToolkit() // Toolkit **类的方法**
- public Toolkit getToolkit() // Component **类的方法**

【例 11.2**】 在应用程序中装载并显示图像。**

```
import java.awt.*;
import javax.swing.*;
class ImageDemo1 extends JPanel {
  private Image im;
  public ImageDemo1() {
    String imageF = System.getProperty("user.dir") + "/sunflowr.jpg";
    im = getToolkit().getImage(imageF);
    setBackground(Color.white);
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.drawImage(im, 0, 0, this);
  }
  public static void main(String[] args) {
    JPanel panel = new ImageDemo1();
    JFrame frame = new JFrame("向日葵");
    frame.setBackground(Color.white);
    panel.setBackground(Color.white);
    frame.setSize(200, 150);
    frame.setContentPane(panel);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

程序的运行结果如图 11.2 **所示。**

图 11.2 **例** 11.2 **运行结果**

若在程序中显示图像前增加设置裁剪区域的语句，即：

```
g.clipRect(0,10, 200, 160);  // 增加这个语句
g.drawImage(im, 0, 0, this);
```

则程序的运行结果如图 11.3 **所示。**

图 11.3 **设置裁剪区域的显示效果**

11.2 动画

动画技术的原理是多幅图画每隔一定时间间隔连续进行显示。

【例 11.3**】 一个简单的文字滚动的** Applet**。这个** Applet **创建一个滚动信息的线程，信息从** Applet **的显示区域自右至左滚动显示。**

```
import java.applet.Applet;
import java.awt.*;
public class SimpleBanner extends Applet implements Runnable {
   String msg = " A Simple Moving Banner.";
   Thread t = null;
   int state;
   boolean stopFlag;
   public void init() {  // 设置颜色
      setBackground(Color.cyan);
      setForeground(Color.red);
      setFont(new Font("",Font.BOLD,24));
   }
   public void start() {// 启动线程
      t = new Thread(this);
      stopFlag = false;
      t.start();
   }
   public void run() {// 跑动 banner 线程的进入点
      char ch;
      for( ; ; ) {// 显示 banner
         try {
            repaint();
            Thread.sleep(500);
            ch = msg.charAt(0);
            msg = msg.substring(1, msg.length());
            msg += ch;
            if(stopFlag)break;
         } catch(InterruptedException e) {}
      }
   }
   public void stop() {// 暂停 banner
      stopFlag = true;
      t = null;
   }
```

```
    public void paint(Graphics g) {  // 显示 banner
      g.drawString(msg, 50, 30);
    }
}
```

程序运行结果如图 11.4 所示。

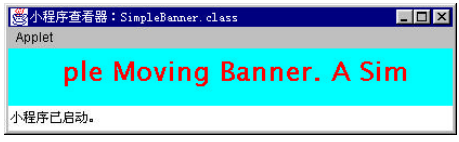

图 11.4　例 11.3 运行结果

【例 11.4】　在 Applet 中显示动画。一系列 gif 文件顺序连续显示就得到动画效果。本程序中，在显示完一个 gif 文件后，用 sleep()暂停一秒钟。gif 文件个数、各 gif 文件名和暂停时间作为 HTML 传送给 Applet 的参数。gif 文件存储在程序文件目录下的 images 子目录中。

```
import java.applet.*;
import java.awt.*;
public class ImageAnimation extends Applet implements Runnable {
  int count;
  Image images[];
  int msec;
  Thread thread;
  int index;
  public void init() {
    // 读图像文件个数参数
    String str = getParameter("count");
    count = Integer.parseInt(str);
    // 分配和初始化图像文件名数组
    images = new Image[count];
    for(int i = 0; i < count; i++) {
      String filename = getParameter("file" + i);
      images[i] = getImage(getDocumentBase(),filename);
    }
    // 读 sleep()方法参数
    str = getParameter("msec");
    msec = Integer.parseInt(str);
  }
  public void start() {
    // 创建和启动线程
    thread = new Thread(this);
```

```
        thread.start();
    }
    public void paint(Graphics g) {
        g.drawImage(images[index], 10, 10, this);
    }
    public void run() {
        try {
            while(true)    {
                Thread.sleep(msec);
                ++ index;
                if(index >= count)index = 0;
                repaint();
            }
        }catch(Exception e) {
        }
    }
    public void stop() {
        // 终止线程
        thread.stop();
        thread = null;
    }
}
```

相应的 HTML **文档内容为:**

```
<applet code = "ImageAnimation" width = 150 height = 150>
<param name = "count" value = "4">
<param name = "file0" value = "images/phone0.gif">
<param name = "file1" value = "images/phone1.gif">
<param name = "file2" value = "images/phone2.gif">
<param name = "file3" value = "images/phone3.gif">
<param name = "msec" value = "1000">
</applet>
```

图 11.5　**例** 11.4 **运行结果**

程序运行结果如图 11.5 **所示。**

一般的动画都是单独成为一个线程，不要直接在 Applet.paint() **方法中画图像。**

对从网络下载图像，需要注意，无论浏览器还是 Applet，**下载图像是异步的。图像只是在显示时才下载，真正下载图像的时间是在运行方法** drawImage()**时。当图像较大或网速较慢时，图像显示可能断断续续的。**

若希望在图像没有下载完之前关闭显示，到显示时连续显示，可使用 Java **在** java.awt **包**

中提供的类 MediaTracker，**它可跟踪资源的下载。**

【例 11.5】 **修改例** 11.4 **使用类** MediaTracker **跟踪** Applet **资源。这里省略了与例** 11.4 **相同的部分。**

```
.....
public class ImageAnimation extends Applet implements Runnable {
  .....
  MediaTracker tracker ;  // 定义资源跟踪对象
  public void init() {
    .....
    tracker = new MediaTracker(this);
    for(int i = 0; i < count; i++) {
      String filename = getParameter("file" + i);
      images[i] = getImage(getDocumentBase(),filename);
      tracker.addImage(images[i],i); // 加入跟踪内容和跟踪内容的标识符
    }
    .....
  }
  public void run() {
    try {
      tracker.waitForAll(); // 等待所有的资源下载
      while(true)  {
        .....
      }
    }catch(Exception e) {
  }
}
```

【例 11.6】 **这里再给出一个包含动画线程的** Java **小程序。**

```
import java.awt.*;
import java.awt.image.ImageProducer;
import java.applet.Applet;
public class ThreadsAnimator extends Applet implements Runnable{
  Image images[];
  MediaTracker tracker;
  int index = 0;
  Thread animator;
  int maxWidth,maxHeight;
  Image offScrImage;
  Graphics offScrGC;
  boolean loaded = false;
```

```
// 初始化 Applet
public void init(){
    tracker = new MediaTracker(this);
    maxWidth = 100;
    maxHeight = 100;
    images = new Image[10];
    try{
        offScrImage = createImage(maxWidth,maxHeight); // 生成后台缓冲区
        offScrGC = offScrImage.getGraphics();
        offScrGC.setColor(Color.lightGray);
        offScrGC.fillRect(0,0,maxWidth,maxHeight);
        resize(maxWidth,maxHeight);
    }catch(Exception e){
        e.printStackTrace();
    }
    for(int i = 0;i < 10;i++) {
        String imageFile = new String("images/Duke/T" +
            String.valueOf(i+1) +".gif");
        images[i] = getImage(getDocumentBase(),imageFile):
        tracker.addImage(images[i],i);
    }
    try{
        tracker.waitForAll();
    }catch(InterruptedException e){
    }
    loaded = true;
}
public void paint(Graphics g){
    if(loaded) {
        g.drawImage(offScrImage,0,0,this);
    }
}
public void start(){
    if(tracker.checkID (index)){
        offScrGC.drawImage(images[index],0,0,this);
    }
    animator = new Thread(this);
    animator.start();
}
public void run(){
```

```
        Thread me = Thread.currentThread();
        while((animator != null)&&(animator == me)){
          if(tracker.checkID(index)){
            offScrGC.fillRect(0,0,100,100);
            offScrGC.drawImage(images[index],0,0,this); index++;
            if(index>=images.length){
              index = 0;
            }
          }
          try{
            animator.sleep(200);
          }catch(InterruptedException e){
          }
          repaint();
        }
      }
    }
```

本程序使用了双缓冲区技术来消除动画显示的闪烁。程序运行的显示屏幕如图 11.6 所示。

图 11.6　例 11.6 运行屏幕显示

11.3　声音

Java 的一大特色是对声音的直接支持，尤其是在动画中配上声音效果。Java 平台采用了新的声音引擎，并支持应用程序和 Applet 中的音频。

它支持下列音频文件格式：WAV、AIFF 和 AU（Sun 公司支持）。由于 AU 格式的声音仅有 8 kHz 的采样频率且不支持立体声效果，所以音质不太好。但 AU 声音文件的尺寸较小，有利于网上传播。

此外，它还支持下列基于 MIDI（Musical Instrument Digital Interface，乐器数字接口）的歌曲文件格式：MIDI TYPE 0、MIDI TYPE 1 和 RMF（Rich Music Format）。

下面也从在 Applet 和应用程序两种不同的程序中如何播放声音来介绍声音媒体的使用。

1，在 Applet 中播放声音

（1）Applet 的 play()方法

Java 语言在 Applet 类中提供的方法 play()可以将指定声音文件的装载和播放一并完成，play()方法的声明如下：

- public void play(URL url)播放指定绝对地址 URL 处的声音文件。
- public void play(URL url,String name)播放给定 URL 相对地址和文件名为 name 的声音文件。

其中 url 参数也经常使用 getCodeBase()或 getDocument()方法。例如：

```
play(getCodeBase(),"audio.au");
```

将加载和播放 Applet 文件所在目录的声音文件“audio.au”。方法 getCodeBase()获取的工作目录正是 play()方法需要的。

play()方法只能将声音播放一遍，若想将某声音作为背景声音循环播放，需要用到功能更强的 java.applet 包中的 AudioClip 接口，它能更有效地管理声音的播放操作。

（2）AudioClip 中的播放声音方法

在 AudioClip 接口中声明了三个需实现的方法：

- public void play()开始播放声音文件，每次调用这个方法，都从头开始播放声音文件。
- public void loop()在一个循环中开始播放声音文件。
- public void stop()停止播放这个声音文件。若未用该方法停止声音的播放，即使离开这一 Web 页面，声音也不会停止。

【例 11.7】 在 Applet 中播放指定 URL 处的 au、wav 和 mid 声音文件。

```
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.applet.*;
public class SoundDemo extends Applet{
  AudioClip aau;
  public void init(){
    aau = getAudioClip(getDocumentBase(),"spacemusic.au");
    this.addMouseListener(new MouseAdpt(this));
  }
  public void paint(Graphics g){
    g.drawString("鼠标进入显示区播放 spacemusic.au",20,60);
    g.drawString("在显示区鼠标按键播放 tada.wav",20,80);
    g.drawString("鼠标移出显示区播放 trippygaia1.mid",20,100);
  }
  class MouseAdpt extends MouseAdapter{
    Applet ap;
    MouseAdpt(Applet p){
      ap = p;
    }
    public void mouseEntered(MouseEvent e){
      ((SoundDemo)ap).aau.play();
    }
    public void mousePressed(MouseEvent e){
      ap.play(ap.getDocumentBase(),"tada.wav");
    }
    public void mouseExited(MouseEvent e){
```

```
            ap.play(ap.getDocumentBase(),"trippygaia1.mid");
        }
    }
}
```

程序运行的界面如图 11.7 所示。

图 11.7　例 11.7 运行显示界面

程序中使用两种方法播放三个声音文件：一种方法是应用 AudioClip 对象自身的方法 play()；另一种方法是应用 Applet 对象的 play()方法。当用户的鼠标进入 Applet 显示区时，使用第一种方法播放 spacemusic.au 文件；当在显示区中对鼠标按键时，用第二种方法播放 tada.wav 声音文件；当鼠标光标移出 Applet 显示区时，也用第二种方法播放 trippygaia1.mid 文件。若在播放一个声音文件时就对鼠标进行播放其他声音文件的操作，可听到声音文件同时播放的混合声音。

本程序要求播放的声音文件与 Applet 位于相同的文件夹。play()一旦被调用，立刻开始恢复和播放声音。如果找不到指定的声音文件，将不会有出错信息和任何声音出现。

2，在应用程序中播放声音

随着 Java 2 的引入，应用程序也能用 Applet 类的 newAudioClip()方法装入声音文件并播放声音。该方法的声明如下：

```
public static final AudioClip newAudioClip(URL url)
```

可用该方法的调用返回值来创建一 AudioClip 对象。例如：

```
AudioClip au = Applet.newAudioClip("tada.wav");
```

在创建 AudioClip 对象 au 后，可以对该对象调用 play()、loop()以及 stop()方法来控制声音文件的播放。若 getAudioClip()或 newAudioClip()方法不能找到指定的声音文件，AudioClip 对象的值将是空的。播放空对象会导致出错，所以标准的过程首先是对该条件进行检测。

【例 11.8】　在应用程序中播放声音。

```
import java.net.*;
import java.applet.*;
import javax.swing.*;
import java.awt.*;
public class SoundDemo1 extends JPanel{
   Applet ap;
   URL cb;
   AudioClip aau;
   JLabel l1 = new JLabel("<html><h3>您正在欣赏 canyon.mid 音乐！ ");
   SoundDemo1(){
      setBackground(Color.white);
```

```
        add(l1);
        try {
            cb = new URL("file:"+System.getProperty("user.dir")+"/canyon.mid");
            aau = Applet.newAudioClip(cb);
            aau.play();
        } catch (MalformedURLException e){
            System.err.println(e.getMessage());
        }
    }
    public static void main(String args[]){
        JFrame f = new JFrame("在应用程序中播放声音");
        f.getContentPane().add(new SoundDemo1());
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(300,100);
        f.show();
    }
}
```

本程序运行时有一个界面（如图 11.8 所示），在界面上只有一个标签，程序运行后可听到程序指定播放的声音文件的音乐。程序未对声音的播放进行控制。

图 11.8　例 11.8 运行界面

习 题 十 一

11.1　Java **语言支持的图像的格式为**_____**和**_____**。**

11.2　**创建** Image **图像对象时，可用方法**_____**。**

11.3　drawImage()**方法有显示图像、**_____**和**_____**的功能。**

11.4　**除了** Graphics **类的** drawImage()**方法可裁剪图像外，另外可用于裁剪图像的** Graphics **类的方法是**_____**和**_____**。**

11.5　**在应用程序中装载图像可借助于**_____**类的对象。**

11.6　**在应用程序中播放声音可用**_____**方法获得一个** AudioClip **对象。**

11.7　**编程：当鼠标进入、退出小程序显示区域时显示不同的图像。**

11.8　**编程：设计一个** GUI **应用程序，在窗口框架上有一个文本框和一个按钮，在文本框中输入声音文件名，选择按钮后播放其声音。**

第 12 章　Java 应用实例

本章根据前面所学的 Java 语言知识，设计一个简单的图像浏览和播放声音的 Java 应用程序。在这个程序中，应用了 GUI 设计方法，综合了一些方面的程序设计技术。为帮助读者回顾以前的知识并理解程序，在源程序中给出了较多的注释。

程序在一个框架窗口中创建了一个具有两个卡片页标签的 JTabbedPane 组件，在两个卡片页标签上各添加一个按钮组件，选择不同的按钮将打开文件对话框选择要浏览的图像文件或声音文件，选择错误则出现一个信息对话框，消除信息对话框后可重新选择文件。为简单起见，选择的图像文件显示在按钮中，选择的声音文件名也显示在按钮中，同时在声音设备中播放声音。若要继续显示图像或听另一个声音文件，可选择按钮再次打开文件对话框。

【例 12.1】　一个能够浏览图像文件和播放声音文件的 Java GUI 应用程序。

```
// 引入程序需要的包
import java.io.*;
import java.applet.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
// 定义主类
public class ImageAndSound{
   public static void main(String args[]){
      new Myis();
   }
} // 主类结束
// 定义继承框架 JFrame 类的子类 Myis，并实现两个监听器接口
class Myis extends JFrame implements ChangeListener,ActionListener{
   JTabbedPane jt;
   JButton jb[];
   int index = 0;
   JFileChooser fc;
   URL cb;
   AudioClip au;
   Myis(){   // 构造方法
      super("图像和声音");        // 调用父类的构造方法
      jt = new JTabbedPane();     // 创建 JTabbedPane 对象
```

```
    jb = new JButton[2];        // 创建按钮数组
    fc = new JFileChooser();   // 创建文件选择器
    // 创建按钮对象并注册监听器
    jb[0] = new JButton("<html><h1><font color=blue>请选择图片文件");
    jb[1] = new JButton("<html><h1><font color=blue>请选择声音文件");
    jb[0].addActionListener(this);
    jb[1].addActionListener(this);
    // 在 JTabbedPane 上加入标签页
    jt.addTab("<html><h2><font color=blue>浏览图片",jb[0]);
    jt.addTab("<html><h2><font color=blue>播放声音",jb[1]);
    // 对两个 gif 文件创建两个 ImageIcon 对象
    ImageIcon jtim1 = new ImageIcon("images/gifIcon.gif");
    ImageIcon jtim2 = new ImageIcon("images/sound.gif");
    // 将图像添加到卡片页标签上
    jt.setIconAt(0,jtim1);
    jt.setIconAt(1,jtim2);
    // 对 JTabbedPane 对象注册监听器
    jt.addChangeListener(this);
    // 将 JTabbedPane 对象添加到 JFrame 的主容器 ContentPane 上，放在中央
    getContentPane().add(jt,BorderLayout.CENTER);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // 注册窗口关闭事件
    setSize(400,300);   // 设置框架窗口的初始显示大小
    setVisible(true);   // 显示框架窗口
  }
  // 选择 JTabbedPane 标签时执行的方法
  public void stateChanged(ChangeEvent e){
    if(e.getSource()==jt){
      // 获取标签页序号
      int i = ((JTabbedPane)e.getSource()).getSelectedIndex();
      // 使选择卡片页上的按钮可见，另一卡片页上按钮隐去
      jb[index].setVisible(false);
      jb[i].setVisible(true);
      index = i;  // 记下当前选择的卡片页序号
    }
  }
  // 选择标签中按钮时执行的方法
  public void actionPerformed(ActionEvent e){
    int returnVal = fc.showOpenDialog(Myis.this);      // 打开文件打开对话框
    if (returnVal == JFileChooser.APPROVE_OPTION) {    // 若选择了文件
      File file = fc.getSelectedFile(); // 用选择的文件创建 File 类的对象
```

```
    String ext = null;
    String s = file.getName();              // 获取文件名
    // 在文件名字符串中查找文件名和扩展名间的分隔符
    int i = s.lastIndexOf('.');
    if (i > 0 &&  i < s.length() - 1)
      ext = s.substring(i + 1).toLowerCase();          // 将文件扩展名转换为小写
    if(index  ==  0){                                  // 若希望打开的是图像文件
      // 判断是否为 Java 支持的图像文件
      if(ext.equals("gif")||ext.equals("jpg")||ext.equals("jpeg")){
        // 用图像文件创建 ImageIcon 对象
        ImageIcon im = new ImageIcon(file.getPath());
        // 在窗口标题上显示提示
        setTitle("文件："+file.getName()+"->单击图像继续");
        jb[0].setText("");   // 清除按钮上的提示文本
        jb[0].setIcon(im);   // 在按钮上显示图像
      }
      else{      // 若打开文件是 Java 不支持的图像文件
        // 显示信息对话框
        JOptionPane.showMessageDialog(this, "打开文件不支持！");
      }
    }else{   // 若打开的是声音文件
      // 判断是否为 Java 支持的声音文件
      if(ext.equals("mid")||ext.equals("wav")||ext.equals("au")){
        try {
          // 用声音文件构造 URL 对象
          cb = new URL("file:"+file.getPath());
          // 用 URL 对象调用 Applet 类的 newAudioClip()方法
          au = Applet.newAudioClip(cb);
          au.play();                    // 播放声音文件
          jb[1].setText("<html><h1><font color=blue>您正在欣赏"+
              file.getName());    // 更新按钮上的显示文本
          setTitle("单击按钮继续");  // 在窗口标题上显示提示
        } catch (MalformedURLException em){    // 处理 URL 异常
          System.err.println(em.getMessage());
        }
      }else{   // 若不是 Java 支持的声音文件
        JOptionPane.showMessageDialog(this, "打开文件不支持！");
      }
    }
  }
```

```
    }
} // Myis 类结束
```

程序运行的两个显示界面如图 12.1 和图 12.2 所示。

图 12.1　例 12.1 浏览图像的显示

图 12.2　例 12.1 播放声音时的显示

从本程序的结构看，程序由两个类构成。在主类 ImageAndSound 中，仅有一个 main() 方法，它通过创建 Myis 类的对象而执行 Myis 类的构造方法。程序的主要任务在类 Myis 中完成。在类 Myis 中包含了三个方法：Myis 类的构造方法、stateChanged(ChangeEvent e)方法和 actionPerformed(ActionEvent e)方法。它们的作用和用法读者可自己分析。

第 13 章　实验内容与安排

Java 语言作为一种面向对象的程序设计语言，在程序设计思想和方法上与传统的程序设计语言有较大的不同。要学好 Java 语言程序设计，一方面要学习 Java 语言程序设计的概念和方法，自己多动手编写程序；另一方面要重视这门课程的实践性环节，多上机调试自己所编写的程序。两个方面结合起来，才能熟练地掌握 Java 程序设计的方法和技巧，才能真正学好本课程。在有条件的时候，要多安排上机时间。在上机的时候，提高上机效率并保证实验效果也是很重要的。因此，特对本课程的实验做出如下说明和要求：

（1）对于实验安排中布置的实验内容和习题，应自己动手按自己的算法编制程序，上机调试运行通过。

（2）上机前应做好充分的准备工作。对选定的实验内容要先编好程序，并进行认真的静态检查，减少错误的发生。同时，还要预先对程序的运行结果和范围做出估计，做到心中有数。

（3）上机前应熟悉上机操作步骤和各种操作调试命令。

（4）为了解 Java 系统对程序出错的信息提示，可有意识地人为设置一些程序错误，观察编译程序的反应，逐步获得对程序错误的处理经验，提高程序设计能力。

（5）下机后要对实验结果进行整理，分析程序调试过程中所出现的各种情况，对未通过的程序更要认真分析原因，总结经验教训。

（6）完成实验报告。通过实验报告，记下每次实验的目的与要求、实验内容、程序有关算法、源程序清单、运行结果、在调试过程中的体会、经验教训等。

下面的实验安排与实验内容仅供参考，使用者可根据实验总学时的多少，灵活变动实验的次数和内容。

实验一　语言环境和简单程序设计

1. 实验目的与要求

以 Java 2 SDK 为例，掌握 Java 开发工具的安装和简单使用，熟悉 Java 语言的基本数据类型，并能掌握 Java 应用程序和 Java 小程序的开发方法，为以后的实验打下基础。

2. 实验内容

在学习了教材第 1～2 章后进行。

（1）Java 开发工具的安装

将教师预先准备好的 Java 2 SDK 标准版（目前，sun 公司网站上最新的是 2002 年 2 月推出的 1.4.0 版）安装到自己的实验计算机上。观察并记下开发系统安装位置、目录分布和在安装前后硬盘空间的变化情况等有关信息。

（2）Java 开发工具的设置和使用

安装完成后，可设置 Java SDK 开发环境，为以后的实验自己制定和创建一个工作目录或按照教师指定的目录，使得自己的实验内容与别人的实验能够容易区别开。为方便开发工作，可将自己的工作目录设为当前目录，为 Java 的工具目录设置搜索路径，必要时，还要设置 classpath 类搜索路径。可将这些工作创建到一个 bat 批处理文件中，文件名可自定。具体做法可参见教材第 1 章相关部分。

（3）Java 应用程序和小程序的简单程序设计。

选择第 1 章习题部分的应用程序和小程序编程题各一题，用安装的 Java 开发工具调试通过。注意保存时源文件名与程序的主类名要完全一致，文件类型名为 java。

对第 2 章基本数据类型，可根据自己的学习掌握情况，选择 1 至 2 个例题调试运行通过以加深理解。

实验二　控制语句和数组程序设计

1，实验目的与要求

在熟悉 Java 基本数据类型和数组的基础上，能够运用 Java 语言的分支、循环等流程控制语句进行简单的程序设计。能够初步掌握方法、异常处理的程序设计技术。

2，实验内容

在学习了教材第 3～4 章后进行。

实验题可选择教材习题 3.19、3.25、4.4 和 4.10（验证自己判断的结果）。

实验三　面向对象程序设计和字符串程序设计

1，实验目的与要求

掌握面向对象程序以数据为核心的设计思想，学习和掌握使用 String 类、StringBuffer 类和其他常用系统类，创建和使用自己的类等程序设计方法。

2，实验内容

在学习了教材第 5～6 章后进行。

实验题可选择教材习题 5.4、6.3 和 6.6。

实验四　输入输出程序设计

1，实验目的与要求

通过 Java 语言字节流和字符流处理知识的学习，掌握在程序中进行输入输出和文件处理方面的方法和技术，使自己的程序向实用性迈进一步。

2，实验内容

在学习了教材第 7 章后进行。
实验题可选择教材习题 7.9 至 7.11。

实验五　Java Applet 设计

1，实验目的与要求

了解 Java 小程序的运行机制，会用 appletviewer.exe 小程序查看器运行嵌入 Java 小程序的 HTML 文档（若使用 JDK 1.1 版，可在 IE 等浏览器中运行）。

2，实验内容

在学习了教材第 8 章后进行。
实验题为：
（1）教材习题 8.4、8.5。
（2）查看并运行 Java SDK 自带的小程序例，它们均在开发工具所在目录的子目录 demo\applets 中。

实验六　图形用户界面设计

1，实验目的与要求

通过 Java 语言 GUI 知识的学习，在了解 GUI 相关概念的基础上，掌握 Java GUI 程序设计的方法，并能够正确理解和使用 Java 的事件处理机制。

2，实验内容

在学习了教材第 9 章后进行。
实验题可选择教材习题 9.9 和 9.10。

实验七　Java 线程和多媒体程序设计

1，实验目的与要求

了解 Java 语言的多线程的程序设计方法以及在多媒体程序设计中的应用，掌握在 Java 应用程序和小程序中浏览图像、播放声音的方法。

2，实验内容

在学习了教材第 10～11 章后进行。
实验题为：

（1）第 11 章一个动画例题，自选 4 至 5 个 GIF 或 JPEG 图像文件实现它们的动画（应用程序或小程序均可）。

（2）习题 11.7 和 11.8。