第 1 章　Java 语言概述

面向对象的编程语言使程序能够比较直观地反映客观世界的本来面目，并且使软件开发人员能够运用人类认识事物所采用的一般思维方法进行软件开发，是当今计算机领域中软件开发的主流技术。所有面向对象的程序设计语言都支持对象、类、消息、封装、继承、多态等诸多概念，而这些概念是人们在进行软件开发、程序设计的过程中逐渐提出来的。

美国加州 Sun Microsystem 公司于 1995 年正式推出纯面向对象(object-oriented，简称 OO)的 Java 程序设计语言，具有面向对象、与平台无关、安全、稳定和多线程等优良特性，是目前软件设计中极为健壮的编程语言。由于它很好地解决了网络编程语言中的诸多问题，因此一经推出，便受到了计算机界的普遍欢迎和接受，并得到了广泛的应用和发展，成为目前最为流行的面向对象程序设计语言之一。

1.1　Java 语言简介

Java 语言的出现是源于对平台无关语言的需要，希望这种语言能编写出嵌入各种家用电器等设备的芯片上，且易于维护的程序。但是，人们发现当时的编程语言，比如：C、C++等都有一个共同的缺点，那就是只能对特定的 CPU 芯片进行编译。这样，一旦电器设备更换了芯片就不能保证程序正确运行，就可能需要修改程序并针对新的芯片重新进行编译。1990 年 Sun 公司成立了由 James Gosling 领导的开发小组，开始致力于开发一种可移植的、跨平台的语言，该语言能生成正确运行于各种操作系统，各种 CPU 芯片上的代码。因此，Gosling 决定自行开发一种新的语言，并将该语言命名为 Oak（橡树），这便是 Java 语言的前身。

Java 语言的快速发展得益于 Internet 和 Web 的出现。1993 年 7 月，伊利诺斯大学的 NCSA 推出了一个在 Internet 上广为流行的 WWW 浏览器 Mosaic 1.0 版。然而，这时的 WWW 页面虽然内容丰富，比如可以实现声、图、文并茂，但它却是静态的，若想增强 WWW 的动感，需要通过一种机制来使它具有动态性。其解决方案显然是嵌入一种既安全可靠，同时又非常简练的语言，这时的 Oak 完全能满足这一要求。到 1994 年， Sun 公司的创始人之一 Bill Joy 的介入，使 Oak 成为 Java 而得以走红。

由于 Java 确实是一种分布式、安全性高、内部包含的编译器非常小、并且适合网络开发环境的语言，因而一经发布，立即得到包括 Netscape 在内的各 WWW 厂商的广泛支持。工业界一致认为：“Java 是（20 世纪）80 年代以来计算机界的一件大事”。微软总裁 Bill Gates 认为：“Java 是长期以来最卓越的程序设计语言”。而今，Java 已成为最流行的面向对象的网络编程语言。

Java 具有简单、面向对象、可移植、分布式、多线程、安全、动态等特点：

简单性：Java 语言产生于 C++语言之后，是完全面向对象的编程语言，充分吸取了 C++语言的优点，采用了程序员所熟悉的 C 和 C++语言的许多语法，同时又去掉了 C 语言中指针、内存申请和释放等影响程序健壮性的部分，并提供自动垃圾回收机制以简化内存管理。因此，可以说 Java 语言是站在 C++语言这个“巨人的肩膀上”前进。

面向对象：Java 是纯面向对象的程序设计语言，支持封装、多态性、和继承。本书将在第 4 章开始详细地讨论 Java 语言的面向对象特性。

可移植：Java 语言的可移植性来源于它的平台无关性，这是 Java 语言最大的优势。Java 将源程序编译成字节码（二进制代码），这种字节码通过 Java 解释器来解释执行。任何一台机器，只要配备了 Java 解释器，就可以运行 Java 字节码，而不管这种字节码是在何种平台上生成的。这不仅使开发的 Java 源代码是可移植的，甚至源代码经过编译之后形成的二进制代码——字节码，也同样是可移植的。因为 Java 源程序编译生成的字节码不是直接在操作系统平台上运行，而是在一个称为 Java 虚拟机（JVM）的平台上运行。利用 JVM 把 Java 字节码跟具体的软硬件平台分隔开来，这就保证在任何机器上编译的 Java 字节码文件都能在不同的机器上运行。

分布式：基于 Java 开发的应用程序可以借助 URL 开启和存取网络对象，实现了数据的分布性；同时，Java 语言编写的 Applet 可以从服务器下载到客户端执行，服务器的部分计算因而能被转移到客户端进行，实现了操作的分布性。Java 还提供了一整套网络类库，开发人员可以利用类库进行网络程序设计，

进而实现了 Java 语言的分布式特性。

多线程：首先，Java 环境本身就是多线程的；其次，Java 语言内置对多线程的控制，可以大大简化多线程应用程序开发。实际上多线程使人产生多个任务在同时执行的错觉，因为，目前的计算机的处理器在同一时刻只能执行一个线程，但处理器可以在不同的线程之间快速地切换，由于处理器速度非常快，远远超过了人接收信息的速度，所以给人的感觉好象多个任务在同时执行。C++没有内置的多线程机制，因此必须调用操作系统的多线程功能来进行多线程程序的设计。

安全：指针和释放内存等原 C++功能在 Java 中被删除，从而避免了非法内存操作。比如，当你准备从网络上下载一个程序时，你最大的担心是程序中含有恶意的代码，比如试图读取或删除本地机上的一些重要文件，甚至该程序是一个病毒程序等。当你使用支持 Java 的浏览器时，你可以放心地运行 Java 的小应用程序 Java Applet 而不必担心病毒的感染和恶意的企图，因为 Java 小应用程序将限制在 Java 运行环境中，不允许它访问计算机的其它部分。

动态：Java 的动态特性是其面向对象设计方法的发展，它允许程序动态地装入运行过程中所需要类，这就使得 Java 可以在分布环境中动态地维护程序及类库，而不像 C++那样，每当其类库升级之后，相应的程序都必须修改，重新编译。

1.2 Java 运行环境与开发环境

1.2.1 Java 三种平台简介

Java 运行平台主要分为下列 3 个版本：

（1）J2SE：称为 Java 标准版或 Java 标准平台。J2SE 提供了标准的 SDK 开发平台。利用该平台可以开发 Java 桌面应用程序和低端的服务器应用程序，也可以开发 Java Applet 程序。

（2） J2EE：称为 Java 企业版或 Java 企业平台。J2EE 可以构建企业级的服务应用，J2EE 平台包含了 J2SE 平台，并增加了附加类库，以便支持目录管理，交易管理和企业级消息处理等功能。

（3）J2ME：称为 Java 微型版或 Java 小型平台。J2ME 是一种很小的 Java 运行环境，用于嵌入式的消费产品中，如移动电话，掌上电脑或其他无线设备等。

1.2.2 Java SDK 开发环境

1. 安装 Java Development Kit(JDK)

本书基于J2SE的SDK1.5 版来学习Java。可以登录到Sun的网站 http://java.sun.com/，免费下载SDK1.5（jdk-1_5_0-windows-i586.exe）。安装的时候可以选择安装到任意的硬盘驱动器上，例如安装到D:\jdk1.5.0 目录下。正确安装后，在JDK目录下有bin、demo、lib、jre等子目录，如图 1_1，其中bin目录保存了javac、java、appletviewer等命令文件，demo目录保存了许多java的例子，lib目录保存了Java的类库文件，jre目录保存的是Java的运行时环境（JRE）。在安装SDK的同时，计算机就安装上了Java运行环境平台。

图 1_1 SDK 目录结构

2. 环境变量的设置

设置环境变量的目的是为了能够正常使用所安装的 JDK 开发包。通常，我们需要设置三个环境变量：JAVA_HOME、PATH 和 CLASSPATH。

JAVA_HOME

该环境变量的值就是 Java 所在的目录，一些 Java 版的软件和一些 Java 的工具需要用到该变量，设置 PATH 和 CLASSPATH 的时候，也可以使用该变量以方便设置。

PATH

该环境变量指定一个路径列表，用于搜索可执行文件。执行一个可执行文件时，如果该文件不能在当前路径下找到，则依次寻找 PATH 中的每一个路径，直至找到待执行的文件、或者找完 PATH 中的路径也不能找到，则报错。Java 的编译命令（javac），执行命令（java）和一些工具命令（javadoc、jdb 等）都在其安装路径下的 bin 目录中。

CLASSPATH

该环境变量也指定一个路径列表，是用于搜索 Java 编译或者运行时需要用到的类。在 CLASSPATH 列表中除了可以包含路径外，还可以包含.jar 文件。Java 查找类时会把这个.jar 文件当作一个目录来进行查找。通常，我们需要把 JDK 安装路径下的 jre\lib\rt.jar 包含在 CLASSPATH 中。

设置环境变量有三种方法：

1）在系统特性中设置 PATH 和 CLASSPATH。

在 Windows2000、Windows2003、Windows XP 系统中，右击“我的电脑”，在弹出的快捷菜单中，选择“属性”命令，弹出“系统特性”对话框，再单击该对话框中的“高级”选项，然后单击“环境变量”按钮，添加如下的环境变量：

变量名：JAVA_HOME

变量值：d:\jdk1.5.0 （其内容应根据 JDK 安装目录变化）

变量名：PATH

变量值：.;%JAVA_HOME% \bin;（其内容应根据 JDK 安装目录变化）

变量名：CLASSPATH

变量值：.; %JAVA_HOME%\jre\lib\rt.jar;（其内容应根据实际情况变化）

2） 在 MS-DOS 命令行窗口设置

也可以在 MS-DOS 命令行键入下列命令后,回车确认，例如

set JAVA_HOME=d:\jdk1.5.0;

set PATH=. ; %JAVA_HOME% \bin;

set CLASSPATH = .; %JAVA_HOME% \jre\lib\rt.jar;

注意，这种方式设置的环境变量只对本 DOS 窗口有效。

3）修改系统自动批处理文件

用记事本编辑 Autoexec.bat 文件，将上面 MS-DOS 命令行窗口输入的语句加入即可：

注意： path 及 classpath 环境变量设置中的“.;”是对应应用程序的当前目录。

1.3 一个 Java 程序的开发过程

1.3.1 Java 程序的开发的过程

Java 程序分成三类: Application（应用程序）、Applet（小程序）、servlet（服务器端小程序）。应用程序在计算机中单独运行，而小程序只能嵌在 HTML 网页中由浏览器执行。servlet 是运行在服务器端的小程序，它可以处理客户传来的请求，然后将执行结果传给客户端。无论哪种 Java 程序，其基本开发过程都如图 1_2 所示。

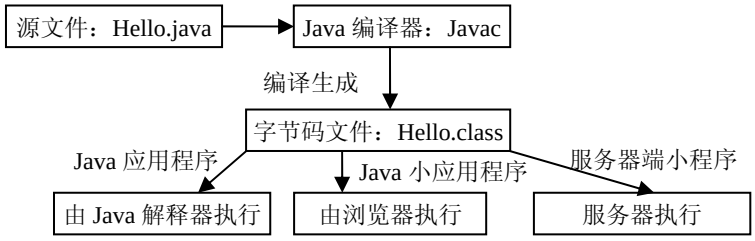

图 1_2 Java 程序的开发过程

（1）编写源文件：使用一个文字编辑器，如 UltraEdit 或记事本，来编写源文件。不可以使用 Word 或者写字板等带格式控制的编辑器，因为它们含有不可见字符。然后将编好的源文件以扩展名. java 保存起来。

（2）编译 Java 源程序：使用 javac.exe 编译源文件得到字节码文件，其扩展名为.class。

（3）运行 Java 程序：如前所叙，Java 程序分为三类：Java 应用程序和 Java 小应用程序（Applet）、servlet（服务器端小程序），Java 应用程序必须通过 java 解释器 java.exe 来解释执行其字节码文件，Java 小应用程序必须通过支持 Java 标准的浏览器来解释执行，而 servlet 必须在应用服务器中执行。下面对 Java 应用程序和小应用程序的开发过程作简单介绍。

1.3.2 一个简单的 Java 应用程序的开发

1. 编写源文件

```
public class HelloWorld
{
    public static void main(String args[])
    {
        System.out.println(“Hello World!”);
    }
}
```

注意：

（1）一个 Java 源程序是由若干个类组成的。上面的这个 Java 应用程序简单到只有一个类，类的名字是叫 HelloWorld。

（2）class 是 Java 的关键字，用来定义一个类。public 也是关键字，说明 Hello 是一个 public 类。第一个大括号和最后一个大括号以及它们之间的内容叫做类体。我们将会在后而后章节中系统的学习类的定义和使用。

（3）public static void main(String args[ ])是类体中的一个方法，main 方法是程序开始执行的位置。在一个 Java 应用程序中 main 方法必须被说明为 public static void。

（4）String args[]声明一个字符串类型的数组 args[]（注意 String 的第一个字母是大写的，它是 main 方法的参数，以后会学习怎样使用这个参数）。

（5）将源文件命名为 HelloWorld.java。源文件的命名规则是这样的：如果源文件中有多个类，那么只能有一个类是 public 类；如果有一个类是 public 类，那么源文件的名字必须与这个 public 类的名字完全相同（大小也必须相同），扩展名是.java。如果源文件没有 public 类，那么源文件的名字只要和某个类的名字相同并且扩展名是.java 就可以了。

2. 编译

当创建了 HelloWorld.java 这个源文件后，就要使用 Java 编译器 javac.exe 对其进行编译：

```
javac  Hello.java
```

编译完成后在当前目录下生成一个 HelloWorld.class 文件，该文件称为字节码文件。

如果 Java 源程序中包含了多个类，那么用编译器 javac 编译完源文件后将生成多个扩展名为.class 的文件，每个扩展名是.class 的文件中只存放一个类的字节码，其文件名与该类的名字相同。这些字节码文件将被存放在与源文件相同的目录中。

3. 执行

Java 应用程序必须通过 Java 虚拟机中的 Java 解释器（java.exe）来解释执行其字节码文件。 Java 应用程序总是从主类的 main 方法开始执行。因此，必须如下方式运行 Java 应用程序：

```
java  HelloWorld
```

MS-DOS 命令行窗口将显示如下信息：

```
Hello World!
```

注意：当 Java 应用程序中有多个类时，运行 java.exe 时，跟在其后的类名必须是包含了 main 方法的那个类的名字，即主类的名字。

1.3.3 一个简单的 Java 小应用程序的开发

Java 小应用程序的建立及运行可分为下述四个步骤：

1. 建立 Java 小应用程序源程序文件。

```
import java.awt.*;
import java.applet.*;
public class HelloWorldApplet extends Applet
```

```
{
    public void paint(Graphics g)
    {
        g.drawString("Hello World!",25,25);
    }
}
```

Java 小应用程序不再需要 main 方法，但必须继承自 Applet 类，即它是 Applet 类的子类。

上述源程序中我们使用了 import 语句，这是因为我们要使用系统提供给我们的 Applet 类。Applet 类在包 java.applet 中。关于包以后还会讲解。

2. 利用 Java 编译器（Javac）编译该 Java Applet。产生.class 字节码文件。

```
javac  HelloWorldApplet.java
```

3. 建立一个 HTML 文件，在其中嵌入 Java 字节码文件。

在记事本中建立名为 Test.html 的 HTML 文件，并输入下面两行语句。

```
<APPLET CODE="HelloWorld.class" width=150  height=100>
</APPLET>
```

4. 用 WWW 浏览器或 appletviewer 装入 HTML 文件，使该 Applet 运行。

图 1_3 HelloWorldApplet 运行结果

图 1_3 是用 appletviewer test.html 的运行结果。

1.4 Java 开发工具简介

Java程序的开发，首先需要用到文本编辑器来编辑源文件，最简单的Windows系统自带的记事本。不过，建议大家还是安装一个专门的编辑工具，常用的比如UltraEdit，当前的版本是 16.0，其下载地址是：http://www.ultraedit.com/。

除了纯文本编辑器外，Java 也有许多集成开发环境，我们给大家推荐两款免费下载的集成开发环境比如：Gel、Eclipse。

1. Gel

Gel 号称世界上最轻巧的 Java IDE，这个软件是由 GExperts 开发，下载地址：

http://www.gexperts.com/products/gel/download.asp。

2. Eclipse

Eclipse 是一款非常受欢迎的 Java 开发工具，国内的用户越来越多，其最大特点是它能接受由 Java 开发者自己编写的开放源代码插件，这类似于 Sun 公司的 NetBeans 平台。目前 Eclipse 联盟已推出其 Eclipse3.5 版软件，用户可从 http://www.eclipse.org 下载最新版本。

小结

本章主要介绍了 Java 语言的产生、特点及 Java 程序的开发环境和过程。上述内容对于掌握 Java 语言编程非常重要。通过这些内容的学习，不仅可以了解 Java 语言的特点和开发、运行环境，而且为进一步学习 Java 语言打下了坚实的基础。

习题

一、基本概念

1、 简述 Java 语言的的基本特征

2、 为什么 Java 程序的运行采用了虚拟机机制？

3、 简述 Java 程序程序的运行过程。

二、编程实践

1、安装并配置 Java 开发环境。

2、分别用 UltraEdit 和 Eclipse 完成本章示例中 Java 应用程序和小程序的开发。

第 2 章　Java 语言基础

Java 语言程序设计中的最基本问题包括：标识符、关键字、数据类型、常量、变量、表达式、赋值语句、控制语句、数据的输入与输出等。

2.1　基本数据类型与变量

符号是构成语言和程序的基本单位。Java 语言不采用通常计算机语言系统所采用的 ASCII 代码集，而是采用更为国际化的 Unicode 字符集。在这种字符集中，每个字符用两个字节即 16 位表示。这样，整个字符集中共包含 65535 个字符。其中，前面 128 个字符表示 ASCII 码，使 Java 语言的字符集对 ASCII 码具有兼容性，其他字符用来表示汉字、日文的片假名和平假名、朝鲜文、希腊字母等其他语言中的符号。但是，Unicode 只用在 Java 平台内部，当涉及打印、屏幕显示、键盘输入等外部操作时，仍由计算机的具体操作系统决定其表示方法。例如，使用英文操作系统时，仍采用 8 位二进制表示的 ASCII 码。Java 编译器接收到用户程序代码后，可将它们转换成各种基本符号元素。

2.1.1　标识符

程序员对程序中的各个元素加以命名时使用的命名记号称为标识符（identifier）。在Java语言中对标识符的定义有如下规定：

- 标识符的长度不限。但在实际命名时不宜过长，过长会增加录入的工作量。
- 标识符可以由字母、数字、下划线“_”和美元符号“$”组成，但必须以字母、下划线或美元符号开头。
- 同一个字母的大、小写被认为是不同的标识符，即区分字母的大小写。例如：Result , result , RESULT 代表不同的标识符。

通常情况下，为提高程序的可读性和可理解性，在对程序中的任何一个对象命名时，应该取一个能反映该对象含义的名称作为标识符。此外，作为一种习惯，标识符的开头或标识符中出现的每个单词的首字母通常大写，其余字母小写。例如，TestPoint，getArea。

2.1.2　保留字

具有专门的意义和用途，不能当作一般的标识符使用，这些标识符称为保留字（reserved word），也称为关键字，Java 语言中的保留字均用小写字母表示。下面列出了 Java 语言中的所有关键字：

- 用于数据类型的关键字：
  byte　short　int　long　float　double　char　boolean　enume
- 用于流程控制语句的关键字
  if　else　switch　case　default　do　while　for　break　continue
- 方法、类型、变量的修饰关键字
  private　public　protected　final　static　abstract　synchronized　native　volatile
- 异常处理关键字
  try　catch　finally　throw　throws
- 对象接口相关关键字
  new　extends　implements　class　interface instanceof　this　super
- 逻辑值常量关键字
  false　true　null
- 方法相关关键字
  return　void
- 包相关关键字
  package　import

2.1.3　空白与注释

程序的易读性和易理解性是软件质量评价的重要指标之一，程序中的空白和注释可以帮助提高易读

性和易理解性，并且注释对于交流和软件的维护具有重要的作用。

空格、制表符、回车、换行以及注释统称为空白。对编译器而言，空白的惟一作用是分隔单词，一个空格与连续三个空格或一个换行的作用相同的。

程序中的注释对编译和执行来说是不起作用的，编译系统将忽略注释的内容，而程序中的多个空白与一个空白的作用相同。

Java 语言中使用三种方式给程序加注释：

（1）//注释内容。表示从“//”开头直到此行末尾均作为注释。例如：

```
//comment line
```

（2）/*注释内容*/。表示从“/*”开头，直到“*/”结束均作为注释，可占多行。例如：

```
/* comment on one or more line */
```

（3）/**注释内容*/。表示从“/**”开头，直到“*/”结束均作为注释，可占多行。例如：

```
/** documenting comment having many line
*/
```

在编程时，如果只注释一行，则选择第一种。若注释内容较多，一行写不完时，既可选择第一种方式，在每行注释前加“//”；也可选择第二种方式，在注释段首尾分别加“/*”和“*/”。第三种方式主要用于创建 HTML 文件时，Java 的文档生成器能从这类注释中提取信息，并将其规范化后用于建立 Web 页面。

2.1.4 基本数据类型

数据在计算机中总是以某种特定的格式存放在计算机的存储器中，不同的数据占用存储单元的多少不同，而且不同的数据其操作方式也不尽相同。在计算机中，将数据的这两方面性质抽象为数据类型这个概念。因此，数据类型在程序中就具有两个方面的作用：一是确定了该类型数据的取值范围，这实际上是由给定数据类型所占存储单元的多少确定的；二是确定了允许对这些数据的操作方式。

Java 的基本数据类型为字符型、整型、浮点型和布尔型。表 2-1 中给出了所有的基本数据类型。对于字符串，Java 预定义 String 类代表字符串，但它不是基本类型，本书后续部分将介绍 String 类。

表 2-1　基本数据类型

类型名	值类型	占用内存	能表示的范围
boolean	true 或 false	1 字节	仅限 true 或 false
char	单个 Unicode 字符	2 字节	所有 Unicode 字符
byte	整数	1 字节	-2^7～(2^7-1)
shor	整数	2 字节	-2^{15}～$(2^{15}-1)$
int	整数	4 字节	-2^{31}～$(2^{31}-1)$
long	整数	8 字节	-2^{63}～$(2^{63}-1)$
float	浮点数	4 字节	约在 10^{-38}～10^{38}
double	浮点数	8 字节	约在 10^{-308}～10^{308}

2.1.5　常量

1. 字符型常量

字符型常量的写法是用单引号括住一个字符，例如'A'、'9'、'$'、'_'（下划线）、' '（空格）等。单引号中的字符大小写是有区别的，即 'A' 和 'a' 表示两个不同的字符常量。

Java 语言采用转义字符表示单引号本身以及 ASCII 编码表中不可显示的控制字符（编码小于 20H 的字符），即通过转变其他字符的含义来表示这些特殊的字符。转义字符也由单引号括住，均以反斜杠（'\'）开头。由于反斜杠专门用作转义，所以反斜杠符号本身也需采用转义表示。

常用的转义字符有：'\n' 表示换行符；'\t' 表示水平制表符；'\7' 表示响铃符（输出该字符时会发出一声铃响）；'\\' 表示一个反斜杠；'\'' 表示一个单引号；'\"' 表示一个双引号。

2. 整型常量

Java 语言的整数常量支持三种数制：十进制、八进制和十六进制。0 开头的表示八进制数，其后只允许出现 0 至 7；0x 开头的表示十六进制数，其中允许出现 0 至 9 和 A 至 F；其余表示十进制数。表 2-2 给出同一整数值的三种表示。

表 2-2 同一整数值的三种表示

十进制	八进制	十六进制
0	0	0x0
7	07	0x7
10	012	0xA
82	0122	0x52
126	0176	0x7E

3. 浮点型常量

浮点数常量只能是十进制数。Java 语言允许按日常使用小数点的习惯书写浮点数，并且小数点前或后只要有一个数字即可。例如：

2.14　2.0　0.025　+32.3　-4.27518　2.　.025　-.5

等均是合法的浮点常量。

为了能在参与运算时可获得最大的精度，Java 语言规定上述形式书写的浮点常量的类型均为双精度浮点数类型，即上述浮点数常量每个占用 8 个字节。

4. 布尔型常量

布尔类型有两个常量：true 表示真，false 表示假。

5. 符号常量

有一种特殊的常量是用标识符表示的，称为符号常量。符号常量常用于帮助记忆，提高程序的可读性和可维护性。譬如某一程序经常用到圆周率 π，如果源程序中每次出现 π 我们都写 3.14，那么想提高圆周率精度到 3.1415926 时，就必须修改程序中的每一个 3.14。我们可以将圆周率表示为一个符号常量 PI 并初始化为 3.14，程序中每次使用圆周率都以 PI 表示。如果要提高程序的计算精度，只要修改 PI 的初始值即可。

Java 语言本身并没有提供标准的符号常量，我们可根据应用需要自己动手声明符号常量。常量的值一经确定就不可以改变，并且常量标识符一般大写。Java 中用保留字 final 来定义与实现常量标识符。

final　数据类型　常量名 1=值 1 [,常量名 2=值 2…];

比如：final int PI=3.14159，NUM=100；

2.1.6 变量

与常量不同，变量中存储的数据是可以改变的，且每个变量必须在使用前声明，它的定义包括变量名、变量类型和作用域几个部分。其定义格式如下：

数据类型　变量名[=值 1[,变量名[=值 2]…];

如：int count；　char c='a'；声明变量时，相当于告诉编译器要在该变量中存储什么样的数据。

例 2.1　基本数据类型及变量的声明

```
public class Example2_1{
    public static void main(String args[]){
        int x , y ;//定义x，y两个整型变量
        float z = 1.234f ; //指定变量z为float型，且赋初值为1.234
        double w = 1.234 ;//指定变量w为double型，且赋初值为1.234
        boolean flag = true; //指定变量flag为boolean型，且赋初值为true
        char c ; //定义字符型变量c
        c = 'A'; //给字符型变量c赋值'A'
```

```
        x = 12; //给整型变量x赋值为12
        y = 300;//给整型变量y赋值为300
    }
}
```

2.2 运算符与表达式

对各种类型的数据进行加工的过程成为运算，表示各种不同运算的符号称为运算符，参与运算的数据称为操作数，由操作数（变量、常量）和运算符一起构成表达式，表达式可以是变量、常量和运算符的组合，也可以是方法调用或者对象分配。其中，由变量、常量和运算符构成的符号串是最常见的一种表达式。

运算符用于标识程序执行的运算类型。在 Java 中主要包括算术运算符、关系运算符、逻辑运算符、位运算符和赋值运算符等运算符。

2.2.1 算术运算

表达式的类型由运算符和操作数确定。算术表达式是由算术运算符或者位运算符将操作数连接组成的表达式。Java 语言的算术运算符如表 2-3 所示：

表 2-3 Java 中的算术运算符

	运算符	用法	含义
二元运算符	+	op1+op2	加法
	-	op1-op2	减法
	*	op1*op2	乘法
	/	op1/op2	除法
	%	op1%op2	模运算(求余)
一元运算符	+	+op1	正数
	-	-op1	负数
	++	++op1,op1++	自增
	--	--op1, op1++	自减

参与算术运算的操作数可以是整数类型、浮点类型、双精度类型等，算术表达式求值得到的类型是相应的这些类型。假设变量 x、y 和 z 均声明为 double 类型，则 x+y 是一个算术表达式，表达式类型为 double 类型；(x+y)*z 也是算术表达式，其类型仍为 double 类型。

需要注意的是：

（1）两个整数类型的数据做除法时，结果只保留整数部分，小数部分被舍弃（注意不是四舍五入）。如：2/3 的结果为 0。

（2）运算符“%”的两个操作数只能是整数类型。取模的含义是只取整除后的余数，例如 17 % 3 的求值结果为 2、28 % 5 的求值结果为 3。求模的结果不可能大于或等于第二个操作数。

（3）自增与自减运算符只适用于变量（不能是浮点型和双精度类型），且变量位于运算符的哪一侧有不同的效果。例如，下面的三个语句清楚地说明了这一点。

```
int a1=2,a2=2;
int b=(++a1)*2;     //执行后 b 的值是 6，a1 的值是 3，称++a 为前加
int c=(a2++)*2;     //而 c 的值是 4，a2 的值是 3，称 a++为后加
```

（4）当操作数类型不相同时，为尽最大可能地保持计算的准确性，保证运算结果的精度，Java 会自动进行相应的类型转换，将不同的数据类型转变为精度最高的数据类型。

2.2.2 关系运算

利用关系运算符连接的式子称为关系表达式。关系运算实际上就是我们常说的比较运算，它有 6 个运算符号，列于表 2-4 中。关系运算容易理解，但需注意两点：

（1）关系表达式的运算结果是一个逻辑值“真”或“假”，在 Java 中用 true 表示“真”；用 false 表

示“假”。

（2）注意等于运算符“==”和赋值运算符“=”的区别。

表 2-4 Java 中的关系运算符

运算符	含义	运算符	含义
==	等于	!=	不等于
>	大于	<	小于
>=	大于等于	<=	小于等于

2.2.3 逻辑运算

利用逻辑运算符将操作数连接的式子称为逻辑表达式，逻辑表达式的运算结果是布尔型型。逻辑运算符如表 2-5 所示：

表 2-5 Java 中的逻辑运算符

运算符	例子	含义
&&	x && y	与运算，若 x 和 y 均为 true 则结果为 true，否则结果为 false。
‖	x ‖ y	或运算，若 x 或 y 其中之一为 true 则结果为 true，否则为 false。
!	! x	非运算，若 x 为 true 则结果为 flase，否则为 true。

逻辑表达式在控制程序的执行流向时起着重要作用。例如可用逻辑表达式(score < 90)&& (score >= 80)判断表示学生成绩是否为“良”，该表达式的两个子表达式(score < 90)和(score >= 80)均为布尔类型；当变量 score 的值为 80 分（含 80 分）至 90（不含 90 分）时，该表达式的求值结果为 true；否则该表达式的求值结果为 false。在下一章我们将学习如何利用逻辑表达式的不同求值结果控制程序的执行流向。

2.2.4 赋值运算

赋值运算符组成的表达式称为赋值表达式。 Java 语言中，赋值运算符是“=”。赋值运算符的作用是将赋值运算符右边的一个数据或一个表达式的值赋给赋值运算符左边的一个变量。注意赋值号左边必须是变量。例如：

```
double s=6.5+45; //将表达式 6.5+45 的和值赋给变量 s
```

需要注意的是赋值中的类型转换，在赋值表达式中，如果运算符两侧的数据类型不一致，但赋值符左边变量的数据类型较高时，系统会自动进行隐式类型转换，当然也可以人为地进行强制类型转换；如果赋值符左边变量的数据类型低于右边表达式值的类型，则必须进行强制类型转换，否则编译时报错。例如：

```
int a=65603;
float b;
char c1, c2='A';
b=a;    //正确的隐式类型转换，运算时先将 a 的值转换为 b 的类型，再赋给 b
b=c2;   //正确的隐式类型转换
c1=a;   //不正确的隐式类型转换，编译器会报错
c2=(char)a;   //正确的显式类型转换。运算时 Java 将 a 的值按 char 类型的宽度削窄（抛弃高位的两个字节）再赋给 c2，使 c2 的值为字符’C’。
```

在赋值运算符“=”之前加上其它运算符，则构成复合赋值运算符。Java 常用复合赋值运算符见表 2-6。

表 2-6 Java 中常用复合运算符

复合赋值运算符	举例	等效于	复合赋值运算符	举例	等效于
+=	x+=y	x=x+y	—=	x—=y	x=x-y
*=	x*=y	x=x*y	/=	x/=y	x=x/y

%=	x%=y	x=x%y	^=	x^=y	x=x^y

2.2.5 位运算

位运算是对整数的二进制表示的每一位进行操作，位运算的操作数和结果都是整型量。位运算符如表 2-7 所示：

表 2-7 Java 中的位运算符

运算符	含义	例子	运算规则 （设 x=11010110，y=01011001，n=2）	运算结果
~	位反	~x	将 x 按比特位取反	00101001
&	位与	x&y	x,y 按位进行与操作	01010000
|	位或	x|y	x,y 按位进行或操作	11011111
^	位异或	x^y	x,y 按位进行异或操作	10001111
<<	左移	x<<n	x 各比特位左移 n 位，右边补 0	01011000
>>	右移	x>>n	x 各比特位右移 n 位,左边按符号位补 0 或 1	11110101
>>>	不带符号的右移	x>>>n	x 各比特位右移 n 位，左边的空位一律填零	00110101

2.2.6 数据类型转换

数据类型转换就是将一种类型的数据转换为另一种类型的数据。关于数据类型转换，通常分为隐式类型转换和显式类型转换两种。

在编译器内部通常只执行两个类型相同的操作数的二元运算，例如两个整数相乘或两个浮点数相乘。因而当二元运算的两个操作数类型不同时，编译器会转换其中一个操作数的类型以保证两个操作数的类型一致后再作运算，这种由编译器自动完成的类型转换工作称为隐式类型转换。

例如 score 表示学生某门课程的平时成绩的整数类型变量，下面的表达式计算平时成绩的 30%：

score* 0.3

该表达式中二元运算符的两个操作数类型不一致，左边是 int 类型，右边是 double 类型。编译器会先将 score 转换 double 类型，然后再求两 double 类型数据的乘法，表达式求值结果也为 double 类型。

因此，在从低精度到高精度的数据类型转换可以自动进行，即从表达范围小的类型向表达范围大的类型发生的转换可以自动进行，称之为隐式类型转换。

Java 语言中算术运算、关系运算和逻辑运算的隐式类型转换规则比较简单：如果二元运算符两边操作数的类型不一致，总是将取值范围较小的类型转换为取值范围较大的类型，二元运算的求值结果当然是转换后的类型。这种类型转换规则的目的在于保证类型转换过程不丢失信息。由前面所学的内容可知，取值范围从小到大分别是 byte、int、float 和 double，

注意：byte，short 和 char 在一起运算时，首先转换为 int 类型进行运算。

对于赋值运算 Java 语言规定了不同的类型转换规则：当赋值运算的左、右操作数类型不一致时，右操作数首先转换为左操作数的类型，然后将转换结果赋值给左操作数。并且这一规则只允许左操作数类型的取值范围大于右操作数类型的取值范围，例如将一个 int 类型的表达式赋值给一个 float 类型的变量。如果左操作数类型的取值范围小于右操作数的类型，例如将一个 double 类型的表达式赋值给一个 int 类型或 float 类型的变量，则类型转换过程可能丢失部分信息，所以编译器会报告源程序有错并拒绝生成目标代码。如果你确实需要用到这样的赋值运算，则必须使用显式的类型转换。

显式类型转换需要人为地在表达式前面指明所需要转换的类型，系统将按这一要求把某种类型强制性地转换为指定的类型，强制类型转换的形式是在表达式前面加上由圆括号括住的类型名字。其一般形式如下：

（<类型名>）<表达式>

例如有两个 int 类型的变量 length 和 area 分别表示矩形的长和面积，假设 area 的值为 20、length 的值为 8，则可通过以下语句

System.out.println(area / length);

计算并输出矩形的宽度。由于这两个变量均为 int 类型，故除法运算均不会产生隐式类型转换，输出结果为 2 而不是预期的 2.5。产生误差的主要原因是由于两个操作数均为整数类型，运算符“/”执行整除运算。可利用强制类型转换改进上述语句，让“/”执行普通除法。以下两条语句均可达到目的：

System.out.println((double) area / length);

System.out.println(area / (double) length);

但以下语句在整除运算后才作强制类型转换，输出还是错误结果 2：

System.out.println((double) (area / length));

例 2.2 计算某一学生的数、理、化三科总分与平均分

```
public class Example2_2 {
    public static void main(String args[])
    {
        int math = 92; // 数学成绩
        int phys = 88; // 物理成绩
        int chem = 95; // 化学成绩
        int total; // 三科总分
        int average; // 三科平均分
        //计算三科总分
        total = math + phys + chem;
        //计算三科平均分
        average = total / 3;
        //在屏幕上输出计算结果
        System.out.println("数理化三科总分为" + total);
        System.out.println("数理化三科平均分为" + average);
    }
}
```

程序运行结果：

```
数理化三科总分为275
数理化三科平均分为 91
```

从运行结果可以看出，整除是舍去全部小数部分而不是四舍五入，大家思考怎样才能实现四舍五入呢？

2.2.7 运算符的优先级

运算符的优先级决定了表达式中不同运算执行的先后次序。优先级高的先进行运算，优先级低的后进行运算。在优先级相同的情况下，由结合性决定运算的顺序，表 2-8 是 Java 所有运算符的优先级和结合性。最基本的规律是：域和分组运算优先级最高，接下来依次是单目运算，双目运算，三目运算，赋值运算的优先级最低。

表 2-8 运算符的优先级和结合性

优先级	描述	运算符	结合性
1	分隔符	[] () . , ;	
2	对象归类、自增自减、逻辑非	instance of ++ --	右到左
3	算术乘除运算	* / %	左到右
4	算术加减运算	+ -	左到右
5	移位运算	>> << >>>	左到右
6	大小关系运算	< <= > >=	左到右

7	相等关系运算	== !=	左到右
8	按位与运算	&	左到右
9	按位异或运算	^	左到右
10	按位或运算	|	左到右
11	逻辑与运算	&&	左到右
12	逻辑或运算	||	左到右
13	三目条件运算	?:	左到右
14	赋值运算	=	右到左

2.3 数组

数组是相同类型的数据按顺序组成的一种复合数据类型。通过数组名加数组下标来使用数组中的数据（称为数组中的元素），数组具有下列特点：

- 数组的每一元素均属于同一类型，该类型称为数组的基类型；
- 数组的元素个数一经确定后就保持不变，我们称它为数组的长度；
- 数组的每一元素均能被直接访问，我们用数组下标来标识数组的元素；
- 数组的元素允许是数组类型，从而形成二维数组、多维数组等结构。

数组中的元素通过下标访问，不过要注意，Java 语言规定，数组下标从 0 开始存取。

2.3.1 一维数组

1. 一维数组的声明

声明一个数组就是要确定数组名、数组的维数和数组元素的数据类型。数组名是符合 Java 标识符定义规则的用户自定义标识符，它是数组类对象的引用类型变量的名字，引用类型变量的值是数组类某个对象的句柄。声明引用类型变量时，系统只为该变量分配引用空间，其值为 null，并未创建一个具体的对象。数组的维数用方括号(“[ ]”)的个数来确定，对于一维数组来说，只需要一对方括号。类型标识符是指数组元素的数据类型，它可以是 Java 的基本类型和引用类型，如 int、float、double、char、类(class)、接口(interface)等。一维数组的声明格式如下：

类型标识符 数组名[ ];

或

类型标识符[ ] 数组名;

例如：

```
int abc[ ];
double[ ] example2;
```

都是正确的一维数组声明语句。

2 一维数组的初始化

声明一个数组仅仅为这个数组指定了数组名和数组元素的类型，并不为数组元素分配实际的存储空间。数组经过初始化后，其长度(即可存放的元素个数)就不可再改变。Java 数组的初始化可以通过直接指定初值的方式来完成，也可以用 new 操作符来完成。

（1）直接指定初值的方式

用直接指定初值的方式对数组初始化，是指在声明一个数组的同时将数组元素的初值依次写入赋值号后的一对花括号内，给这个数组的所有数组元素赋上初始值。例如：

```
int[] a1={23,-9,38,8,65};
```

（2）用关键字 new 初始化数组

用关键字 new 初始化数组，只为数组分配存储空间而不对数组元素赋初值。用关键字 new 来初始化数组有两种方式：

1）先声明数组，再初始化数组。这实际上由两条语句构成，格式如下：

类型标识符 数组名[ ];

数组名=new 类型标识符[数组长度];

其中，第一条语句是数组的声明语句，第二条语句是初始化语句。应该注意的是：两条语句中的数组名、类型标识符必须一致。数组长度通常是整型常量，用以指明数组元素的个数。例如：

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

2）在声明数组的同时用 new 关键字初始化数组。这种初始化实际上是将上面所述的两条语句合并为一条语句罢了。格式如下：

类型标识符 数组名[ ]=new 类型标识符[数组长度];

或

类型标识符[ ] 数组名=new 类型标识符[数组长度];

例如：

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

3 一维数组的引用

当数组经过初始化后，就可通过数组名与下标来引用数组中的每一个元素。一维数组元素的引用格式如下：

数组名[数组下标]

其中：数组名是经过声明和初始化的标识符；数组下标是指元素在数组中的位置，数组下标的取值范围是 0～(数组长度－1)，下标值可以是整数型常量或整数型变量表达式。例如，在有了“int[ ] a=new int[10];”声明语句后，下面的两条赋值语句是合法的：

```
a[3]=25;
a[3+6]=90;
```

但

```
a[10]=8；//是错误的，因为下标越界。
```

有一个重要的表示一维数组长度（即元素的个数）的格式：数组名.length。比如对于上面声明的一维数组 a，用 a.length 的方式将得到数组 a 的长度 10。

2.3.2 二维数组

日常生活中处理的许多数据，从逻辑上看是由若干行、若干列组成的，比如图 2_1 中的矩阵 A 。为适应存放这样一类数据，Java 采用二维数组来存取。

$$A=\begin{bmatrix}13 & 16 & 9\\ 26 & 15 & 32\\ 66 & 36 & 73\end{bmatrix}$$

图 2_1 矩阵 A

1. 二维数组的声明

二维数组的声明与一维数组类似，只是需要给出两对方括号，其格式如下

类型说明符 数组名[ ][ ];

或

类型说明符[ ][ ] 数组名;

例如：

```
int arr[ ][ ];
```

或 `int [ ][ ] arr;`

其中：类型说明符可以是 Java 的基本类型、类或接口；数组名是用户根据标识符定名规则给出的一个标识符；两个方括号中前面的方括号表示行，后面的方括号表示列。

2. 二维数组的初始化

二维数组声明同样也是为数组命名和指定其数据类型，不为数组元素分配内存。二维数组的初始化也有直接指定初值的方式和用 new 操作符的方式两种。

（1）用 new 操作符初始化数组

用 new 操作符来初始化数组有两种方式：

1）先声明数组再初始化数组。在数组已经声明以后，可用下述两种格式中的任意一种来初始化二维数组。

数组名=new 类型说明符[数组长度][]；

或

数组名=new 类型说明符[数组长度][数组长度]；

其中：对数组名、类型说明符和数组长度的要求与一维数组一致。例如：

int arra[ ][ ];

arra=new int[3][4]；

上述两条语句声明并创建了一个 3 行 4 列的数组 arra。实际上共有 12 个元素，共占用 12×4=48 个字节的连续存储空间。这里的语句：

arra=new int[3][4]；

实际上相当于下述 4 条语句：

arra=new int[3][]；创建一个有 3 个元素的数组，且每个元素也是一个数组。

arra[0]=new int[4]；创建 arra[0]元素的数组，它有 4 个元素。

arra[1]=new int[4]；创建 arra[1]元素的数组，它有 4 个元素。

arra[2]=new int[4]；创建 arra[2]元素的数组，它有 4 个元素。

上述语句的作用如下图所示：

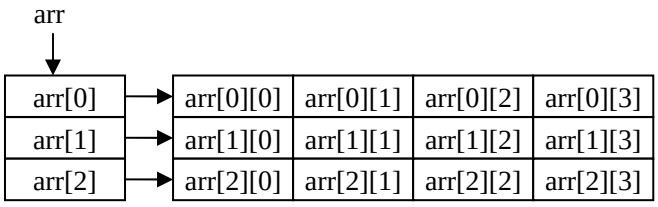

图 2_2 语句 arra=new int[3][4]的作用

从图 2_2 可以看出，Java 语言把二维数组中每一行看成一个数组元素，则有三行的数组可看成只有三个数组元素，只不过这三个元素又是由四个元素组成的数组的。因此，二维数组实际上是每个元素都是一维数组的一个一维数组。

2）在声明数组的同时初始化数组。格式如下：

类型说明符[ ][ ] 数组名=new 类型说明符[数组长度][ ]；

或

类型说明符 数组名[ ][ ]=new 类型说明符[数组长度][数组长度]；

例如：

int[ ][ ] arr=new int[4][ ];

int arr[ ][ ]=new int[4][3];

但是，不指定行数而指定列数是错误的。例如，下面的初始化是错误的。

int[ ][ ] arr=new int[][4];

（2）直接指定初始值的方式

在数组声明时对数据元素赋初值就是用指定的初值对数组初始化。例如

int[ ][ ] arr1={{3,-9,6},{8,0,1},{11,9,8} };

声明并初始化数组 arr1，它有 3 个元素，每个元素又都是有 3 个元素的一维数组。

用指定初值的方式对数组初始化时，各子数组元素的个数可以不同。例如：

```
int[ ][ ] arr1={{3,-9},{8,0,1},{10,11,9,8} };
```

它等价于：

```
int[ ][ ] arr1=new int[3][ ];
int arr1[0]={3,-9};
int arr1[1]={8,0,1};
int arr1[2]={10,11,9,8};
```

小结

类型决定了数据的表示方式、取值范围和可用操作。每一类型的数据都可分为常量与变量两大类。常量的类型由其书写形式决定；变量的类型由程序员显式声明，遵循“先声明、后使用”原则。

表达式用于对数据进行运算，由运算符与操作数组成。一个表达式的求值结果有两重含义：一是表达式的值，二是表达式的类型。表达式的求值结果不仅取决于运算符，而且还与运算符执行的次序密切相关，运算符执行次序由运算符的优先级与结合性质决定。

当二元运算的两个操作数类型不一致时，编译器会自动完成隐式类型转换。程序员自己也可显式地强制类型转换。

本章仅对数组的声明和数组元素的使用方法作了介绍，数组在程序中的应用将在后面的章节作详细介绍。

习题

一、基本概念

1、什么是标识符？标识符的命名规则是什么?

2、什么是关键字？Java 语言有哪几类关键字?

3、Java 语言有哪些基本数据类型?

4、如何定义并且初始化一维数组和二维数组?

二、编程实践

1、输入下面的程序，注意观察输出结果并思考为什么？

```
public class Example2_3 {
    public static void main(String[] args)
    {
        int x,y,n=2;
        x = 214;//x=11010110b;
        y = 89;// y=01011001b
        System.out.println(~x);
        System.out.println(x&y);
        System.out.println(x|y);
        System.out.println(x^y);
        System.out.println(x<<n);
        System.out.println(x>>n);
        System.out.println(x>>>n);
    }
}
```

2、下面程序的输出结果是什么？

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

```
        {
            long[] a = {1,2,3,4};
            long[] b = {100,200,300,400,500};
            b = a;
            System.out.println("数组b的长度:"+b.length);
            System.out.println("b[0]="+b[0]);
        }
    }
```

3、下面的程序输出所有一些字母，输入这段代码，并思考类型转换在这段程序中的应用

```
    public class GreekAlphabet
    {
        public static void main (String args[ ])
        {
            int startPosition=0,endPosition=0;
            char cStart='α',cEnd='ω';
            startPosition=(int)cStart;//cStart做int型转换据运算，并将结果赋值给
startPosition
            endPosition=(int)cEnd ;//cEnd做int型转换运算，并将结果赋值给endPosition
            System.out.println("希腊字母\'α\'在unicode表中的顺序位置:"+(int)'α');
            System.out.println("希腊字母表：");
            for(int i=startPosition;i<=endPosition;i++ )
            {
                char c='\0';
                c=(char)i;//i做char型转换运算，并将结果赋值给c
                System.out.print(" "+c);
                if((i-startPosition+1)%10==0)
                    System.out.println("");
            }
        }
    }
```

第 3 章　Java 语言流程控制语句

Java 程序通过控制语句来执行程序流，完成一定的任务。程序流是由若干个语句组成的，语句可以是单一的一条语句，如 c=a+b,也可以是用大括号{}括起来的一个复合语句。Java 中的控制语句有以下几类：

◇ 分支语句：if-else，switch

◇ 循环语句：while，do-while，for

◇ 与程序转移有关的跳转语句：break，continue，return

◇ 例外处理语句：try-catch-finally，throw

使用分支语句编写的程序称为选择结构程序；使用循环语句编写的程序称为循环结构程序。

3.1　分支语句

Java 语言提供了两种基本的分支选择语句：if 语句和 switch 语句。用这两个语句可以形成三种形式的选择结构。

①双分支选择结构：if/else 语句用来判定一个条件（布尔表达式），当条件为真（true）时执行一个操作，条件为假（false）时执行另一个操作。

②单分支选择结构：省略了 else 的 if 语句在条件为真时执行一个操作，条件为假时则跳过该操作。

③多分支选择结构：switch 语句则是根据表达式的值来决定执行许多不同操作中的某一个操作。当然，使用嵌套 if 语句也可以实现多分支选择结构。

3.1.1　if 语句

if 语句的一般形式为：

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

程序执行这种形式的 if 语句时，如果布尔表达式求值结果为 true 则执行语句块 1，否则执行 else 后面的语句块 2。这种形式的 if 语句常用于对不同条件分别作不同处理的情况。

说明：这里的“语句块 1”和“语句块 2”是指一个语句或多个语句构成的复合语句，当为多个语句时，一定要用一对花括号“{”和“}”将其括起，使之成为一个复合语句。

例 3.1 基本结构的 if-else 语句

```
public class Example3_1 {
    public static void main(String args[]) {
        int math=65 ,english=85;
        if(math>60) {
           System.out.println("数学及格了");
        }
        else {
            System.out.println("数学不及格");
        }
        if(english>90) {
              System.out.println("英语是优");
         }
        else {
           System.out.println("英语不是优");
        }
        System.out.println("我在学习if-else语句");
```

```
        }
    }
```
运行结果是:

数学及格了

英语不是优

我在学习 if-else 语句

在 if 语句中，可以省略 else 子句以形成单分支结构。程序执行这种形式的 if 语句时，如果布尔表达式的求值结果为 true 则执行语句块 1，否则跳过语句块 1 中的语句（这时 if 语句相当于一条空语句）。

例 3.2 缺少 else 子句的 if 语句

```
public class Example3_2 {
    public static void main(String args[]) {
        int a=95 ,b=85,c;
        int temp;
        if(a>b) {
            temp = a;
            a = b;
            b = temp;
            System.out.println("交换数a与b,以保证a<=b");
        }
        System.out.println("数a="+a+"小于数b="+b);
    }
}
```
运行结果是:

交换数a与b,以保证a<=b

数 a=85 小于数 b=95

在例 3.2 中，演示了利用中间变量如何交换两个变量值的方法。

在 if 语句中的“语句块 1”和“语句块 2”中可以是包含任何合法的 Java 语句，当然也包括 if 语句本身。因此，如果在 if 语句的语句区块中仍然是 if 语句，则构成 if 语句的嵌套结构，从而形成多分支选择结构的程序。嵌套 if 语句必须更小心地处理语句编排的格式，既要避免缩进层次太深，又要突出程序的逻辑结构。程序 3.3 演示了一种较常用的 if 语句嵌套缩进方式，该例首先计算三门课程的平均成绩，然后根据平均成绩来得到总评成绩和应发奖学金。

例 3.3 if 语句的嵌套

```
import java.io.*;
public class Example3_3 {
    public static void main(String[] args) {
        int math=90, phys=80, chem=76; // 数理化三科成绩
        // 计算三科平均成绩
        int average = (int) ((math + phys + chem) / 3.0 + 0.5);//加0.5实现四舍五入
        // 根据平均成绩计算总评成绩与奖学金
        String score; // 总评成绩
        double studentship; // 奖学金
        if (average >= 90) {
            score = "优";
```

```
            studentship = 500.00;
        } else if (average >= 80) {
                score = "良";
                studentship = 200.00;
            } else if (average >= 70) {
                    score = "中";
                    studentship = 55.50;
                } else if (average >= 60) {
                    score = "及格";
                    studentship = 0.00;
                } else {
                    score = "不及格";
                    studentship = 0.00;
                }
        // 输出计算结果
        System.out.println("总评成绩为\"" + score + "\"。");
        System.out.println("奖学金为 " + studentship + " 元。");
    }
}
```

运行结果是：

总评成绩为"良"。

奖学金为 200.0 元。

3.1.2 条件运算符

对于一些简单的 if/else 语句，可用条件运算符来替代。

例如，若有以下 if 语句：

```
if (x>y)
    m=x;
else
    m=y;
```

则可用下面的条件运算符来替代：m=(x>y) ? x : y

其中，“? :”被称为条件运算符，“(x>y)? x : y”被称为条件表达式。条件表达式的语义是：若（x>y）条件为 true，则表达式的值取 x 的值，否则表达式的值取 y 的值。条件表达式的一般形式为：

布尔表达式 1？ 表达式 2： 表达式 3

在条件表达式中：

（1）表达式 2 和表达式 3 的类型必须相同。

（2）条件运算符的执行顺序是：先求解表达式 1，若值为 true 则执行表达式 2，此时表达式 2 的值作为整个条件表达式的值，否则求解表达式 3，将表达式 3 的值作为整个条件表达式的值。

在实际应用中，常常将条件运算符与赋值运算符结合起来，构成赋值表达式，以替代比较简单的 if/else 语句。条件运算符的优先级高于赋值运算符，因此，其结合方为“自右自左”。

3.1.3 switch 语句

当要从多个分支中选择一个分支去执行，虽然可用 if 嵌套语句来解，但当嵌套层较数多时，使程序的可读性大大降低。Java 提供的 switch 语句可清楚地处理多分支选择问题。switch 语句根据表达式的值来执行多个操作中的一个，其格式如下：

switch (表达式){

```
    case value1 : 语句块 1;   break;    //分支 1
    case value2 : 语句块 2;   break;    //分支 2
      ……
    case valueN : 语句块 N;   break;   //分支 N
    [default :默认语句块; ]              //分支 N+1
}
```

注意：

◇ 表达式的返回值类型必须是这几种类型之一：int,byte,char,short；

◇ case 子句中的值 valueN 必须是常量，而且所有 case 子句中的值应是不同的。当表达式的值与某个 case 后面的常量值相等时，就执行此 case 后面的语句块；

◇ break 语句用来在执行完一个 case 分支后，使程序跳出 switch 语句，即终止 switch 语句的执行；

◇在一些特殊情况下，多个不同的 case 值要执行一组相同的操作，这时可以不用 break。在这种情况下，如果某个 case 中没有使用 break 语句，一旦表达式的值和该 case 后面的常量值相同，程序不仅执行该 case 里的语句块，而且继续执行后继 case 里的语句块，直到遇到 break 语句为止；

◇ default 子句是可选的。

例 3.4 根据数字判断星期几

```
public class Example3_4
{
    public static void main(String[] args)
    {
        for (int i = 1; i <= 7; i++)
        switch (i) {
            case 1:
               System.out.println(i+":星期一"); break;
            case 2:
               System.out.println(i+":"星期二"); break;
            case 3:
               System.out.println(i+":"星期三"); break;
            case 4:
               System.out.println(i+":"星期四"); break;
            case 5:
               System.out.println(i+":"星期五"); break;
            default:
               System.out.println(i+":"周末");break;
            }
    }
}
```

运行结果是：

```
1:星期一
2:星期二
3:星期三
4:星期四
5:星期五
6:周末
```

7:周末

例 3.4 演示当 i 从 1 变化到 7 时，程序的执行情况，注意，当 i 为 6 或者 7 的时候，与所有 case 子句后面的值都不匹配，所以执行 default 语句块。

下面的例子演示当有些 case 子句后面无 break 时，程序的执行情况。

例 3.5 有些 case 语句缺少 break 的情况

```
public class Example3_5
{
   public static void main(String[] args)
   {
      int month = 5;
      String season;
      switch (month) {
          case 12:
          case 1:
          case 2: season = "冬天"; break;
          case 3:
          case 4:
          case 5: season = "春天"; break;
          case 6:
          case 7:
          case 8: season = "夏天"; break;
          case 9:
          case 10:
          case 11:season = "秋天"; break;
          default:season = "错误的输入";
       }
       System.out.println(month+"月是" + season + ".");
   }
}
```

运行结果是：

5 月是春天.

例 3.6 给定一个日期判断是星期几

```
public class Example3_6
{
   public static void main(String[] args)
   {
      int year = 2009, month = 4, day = 13, total, week, i;
      boolean leap = false;
      leap = (year % 400 == 0) | (year % 100 != 0) & (year % 4 == 0);
      week = 1;//起始日1979-12-31是monday
      total = year - 1980 + (year - 1980 + 3) / 4;
      for (i = 1; i <= month - 1; i++) {
         switch (i) {
             case 1:
```

```
                case 3:
                case 5:
                case 7:
                case 8:
                case 10:
                case 12:
                   total = total + 31;  break;
                case 4:
                case 6:
                case 9:
                case 11:
                    total = total + 30;  break;
                case 2:
                    if (leap)
                        total = total + 29;
                    else
                        total = total + 28;
                    break;
            }
        }
        total = total + day;
        week = (week + total) % 7;
        System.out.print("日期 " + year + '-' + month + "-" + day + " 是 ");
        switch (week) {
            case 0:
                System.out.println("星期日");  break;
            case 1:
                System.out.println("星期一");  break;
            case 2:
                System.out.println("星期二");  break;
            case 3:
                System.out.println("星期三");  break;
            case 4:
                System.out.println("星期四");  break;
            case 5:
                System.out.println("星期五");  break;
            case 6:
                System.out.println("星期六");  break;
        }
    }
}
```

运行结果是：

```
日期 2009-4-13 是 星期一
```

3.2 循环语句

循环语句的作用是反复执行一段程序代码，直到满足终止条件为止。Java 语言提供的循环语句有：while 语句、do-while 语句和 for 语句。这些循环语句各有其持点，用户可根据不同的需要选择使用。

3.2.1 while 语句

while 语句的一般形式为：

```
[初值表达式]
while (循环终止布尔表达式){
    循环体程序语句区块;
    [改变循环变量;]
}
```

while 语句中各个成分的执行次序是：先判断循环终止布尔表达式的值，若值为假，则跳过循环体，执行循环体后面的语句；若布尔表达式的值为 true，则执行循环体中的语句区块，然后再回去判断布尔表达式的值，如此反复，直至布尔表达式的值为 false，跳出 while 循环体。

例 3.7 用 while 循环求 1+2+3+…+20=?

```
public class Example3_7 {
   public static void main(String args[]) {
      int sum=0;
      int i=1,n=20;
      while(i<=n) {
          sum=sum+i;
          i=i+1;
      }
      System.out.println("sum="+sum);
   }
}
```

运行结果是：

```
sum=210
```

3.2.2 do-while 语句

do-while 语句的一般形式为：

```
[初值表达式]
do {
    循环体程序语句区块;
    [改变循环变量;]
} while (循环终止布尔表达式);
```

do-while 语句中各个成分的执行次序是：先执行一次循环体语句区块，然后再判断循环终止布尔表达式的值，若值为 false 则跳出 do-while 循环，执行后面的语句；若值为 true 则再次执行循环体语句区块。如此反复，直到布尔表达式的值为 false，跳出 do-while 循环。

do-while 循环语句与 while 循环语句的区别仅在于 do-while 循环中的循环体至少执行一次，而 while 循环中的循环体可能一次也不执行。

例 3.8 用 do-while 循环求 1+2+3+…+20=?

```
public class Example3_8
{
   public static void main(String[] args)
   {
      int sum = 0, i = 0;
```

```
        do {
            i++;
            sum += i;
        } while (i < 20);
        System.out.println("sum= " + sum);
    }
}
```

3.2.3 for 语句

for 语句的一般形式为：

for (初值表达式; 循环条件布尔表达式; 循环过程表达式){
    循环体程序语句区块;
}

其中：初值表达式对循环变量赋初值；循环条件布尔表达式用来判断循环是否继续进行；循环过程表达式完成修改循环变量、改变循环条件的任务。

for 语句执行过程是：

（1）求解初值表达式；

（2）求解循环条件布尔表达式，若值为真，则执行循环体语句区块，然后再执行第（3）步；若值为假，则跳出循环体语句。

（3）求解循环过程表达式，然后转去执行第（2）步

例 3.9 用 for 循环求 1+2+3+…+20=?

```
public class Example3_9
{
    public static void main(String[] args)
    {
        int sum = 0, i;
        for(i=1;i<=20;i++)
        {
            sum += i;
        }
        System.out.println("sum= " + sum);
    }
}
```

需要注意的是，在 for 语句中，for 头的构件——括号内的三个表达式均可省略，但两个分号不可省略。当在 for 头的构件中省略了任何一个表达式时，应该注意将其写在程序中的其它位置，否则会形成“死循环”等问题。

例 3.10 正向打印大写字母表，反向打印小写字母表

```
public class Example3_10 {
    public static void main(String[] args) {
        char ch; // 循环变量
        // 正向打印字母表
        ch='A';
        for (; ch <= 'Z';)
        {
            System.out.print(ch + " ");
```

```
            ch = (char) (ch + 1);
        }
        System.out.println();
        // 反向打印小写字母表
        ch=(char)(ch+31);
        for (;ch >= 'a' ;)
        {
            System.out.print(ch + " ");
            ch = (char) (ch - 1);
        }
        System.out.println();
    }
}
```

运行结果是：

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
z y x w v u t s r q p o n m l k j i h g f e d c b a
```

务请牢记，如果不是万不得已，最好不要使用省略这些表达式的形式，因为省略这些表达式的形式实际上已经失去了使用 for 语句的意义。

此外，在 for 结构头的构件中，“初值表达式”和“循环过程表达式”中还可以使用逗号运算符。

3.2.4 增强的 for 语句

对于数组类型和集合（Collection）类，Java 新增了一种叫增强的 for 流程的语法，可以使 for 循环更紧凑和易读。例 3.11 演示了增强 for 语句的使用方法。

例 3.11

```
public class Example3_11 {
    public static void main(String[] args) {
        int sum = 0;
        int a[] = {0,1,2,3,4,5};
        for (int i = 0; i < a.length; i++)
            System.out.println("This is "+i);
        // 增强的for语句,达到上面for循环一样的效果
        System.out.println("下面是for-each的执行结果");
        for (int e : a)//把数组a中的每个元素依次取出并赋给e
            System.out.println("This is " + e);
    }
}
```

运行结果是：

```
This is 0
This is 1
This is 2
This is 3
This is 4
This is 5
下面是for-each的执行结果
This is 0
```

```
This is 1
This is 2
This is 3
This is 4
This is 5
```

3.2.5 循环嵌套

循环嵌套是指在循环体内包含有循环语句的情形。Java 语言提供的三种循环结构都可以自身嵌套，也可以相互嵌套。循环嵌套时应该注意的是：无论哪种嵌套关系都必须保证每一个循环结构的完整性，不能出现交叉。

例 3.12 求 1!+2!+3!+…+10!=?

```
public class Example3_11 {
    public static void main(String[] args) {
       int sum=0;
       int mid=1;
       for(int i=1;i<=10;i++)
       {
          mid=1;
          for(int j=1;j<=i;j++)
          {
              mid=mid*j;
          }
          sum=sum+mid;
       }
        System.out.println("总和是："+sum);
    }
}
```

运行结果是：

```
总和是：4037913
```

3.3 跳转语句

Java 语言提供的 break，continue，return 和 throw 都可以实现流程的跳转以改变程序的执行流程。本节只讲 break、continue 和 return 三个转移语句，throw 在后面的章节介绍。

3.3.1 break 语句

break 语句通常有下述不带标号和带标号的两种形式：

（1） break；

（2） break Label;

其中：break 是关键字；Label 是用户定义的语句块标号。

break 语句虽然可以独立使用，但通常用于 switch 结构和循环结构中，以控制程序的执行流程转移。可有下列三种情况：

（1）break 语句用在 switch 语句中，其作用是强制退出 switch 结构，执行 switch 结构后的语句。这一功能已在 3.1.2 节中介绍。

（2）break 语句用在单层循环结构的循环体中，其作用是强制退出循环结构。若程序中有内外两重循环，而 break 语句写在内循环中，则执行 break 语句只能退出内循环，而不能退出外循环。若想要退出外循环，可使用带标号的 break 语句。

（3）break Label 语句用在循环语句中，必须在外循环入口语句的前方写上 Label 标号，可以使程

序流程退出标号所指明的外循环。

3.3.2 continue 语句

continue 语句只能用于循环结构中，其作用是使循环短路，即 continue 语句用来结束本次循环，跳过循环体中下面尚未执行的语句，接着进行终止条件的判断，以决定是否继续循环。对于 for 语句，在进行终止条件的判断前，还要先执行迭代语句。continue 语句有下述两种形式：

（1） continue;

（2） continue Label;

其中：continue 是关键字；Label 为用户定义的语句块标号。

说明：

（1）continue 语句也称为循环的短路语句。用在循环结构中，使程序执行到 continue 语句时回到循环的入口处，执行下一次循环，而使循环体内写在 continue 语句后的语句不执行。

（2）当程序中有嵌套的多层循环时，为从内循环跳到外循环，可使用带标号的 continue lab 语句。此时应在外循环的入口语句前方加上标号。

例如：

```
outer: for( int i=0; i<10; i++ ){       //外层循环
inner:    for( int j=0; j<10; j++ ){    //内层循环
              if( i<j ){
                  ……
                  continue outer;
              }
              ……
          }
          ……
      }
```

3.3.3 返回语句 return

return 语句从当前方法中退出，返回到调用该方法的语句处，并从紧跟该语句的下一条语句继续程序的执行。返回语句有两种格式：

（1） return expression ;

（2） return;

return 语句通常用在一个方法体的最后，也可用在 if-else 语句中，除此之外，将会产生编译错误。

例 3.13 求 1+3+5+7+9=? 以及 10 以内的素数

```
public class Example3_13 {
   public static void main(String args[]) {
      int sum=0,i,j;
      for( i=1;i<=10;i++) {
         if(i%2==0) {           //计算1+3+5+7+9
            continue;
         }
         sum=sum+i;
      }
      System.out.println("sum="+sum);
      for(j=2;j<=10;j++) {    //求50以内的素数
         for( i=2;i<=j/2;i++) {
```

```
            if(j%i==0)
               break;
          }
          if(i>j/2) {
            System.out.println(""+j+"是素数");
          }
        }
      }
    }
```

运行结果是：

```
sum=25
2是素数
3是素数
5是素数
7是素数
```

3.4 循环控制要点及循环语句比较

前面通过示例程序讲述了三种循环语句的用法。仔细分析前面的示例程序，可以看出，循环控制主要有两种办法：一是用计数器控制循环，另一种是用标记控制循环。计数器控制循环指事先确切知道所需执行次数的循环，比如求自然数 1~10 之和。标记控制循环主要适用于哪些事先无法知道循环次数的事务处理。例如统计选票就是这样一类问题，只知道有许多人参加投票，但不能确切地知道选票数。在这种情况下可以使用一个叫做标记值的特殊值作为“数据输入结束”的标志，用户将所有合法的数据都输入之后，就输入这个标记值，表示最后一个数据已经输入完了。循环控制语句得到这个标记值后，结束循环。标记控制循环通常也称作不确定循环，因为在循环开始执行之前并不知道循环的次数。比如在下面的例 3.15 的折半查找过程中，程序使用 low<=high 作为循环控制标记。

一般情况下，不管哪种循环控制，Java 提供的三种循环语句是可以相互替代的。然而，由于 for 语句头中包含了控制循环所需要的各个构件，因此，对于同样的问题，使用 for 循环编写的程序最简洁清晰。比如求自然数 1～10 之和的问题分别用三种循环结构写出，就可以清楚地看到这一事实。对于那些只知道某些语句要反复执行多次(至少执行一次)，但不知道确切执行次数的问题，使用 do-while 循环会使程序更清晰。对于那种某些语句可能要反复执行多次，也可能一次都不执行的问题，使用 while 循环当然是最好了。

最后，循环结构程序在书写时要把握下面几个要点：

（1）从问题中抽取出参与循环运算的的循环不变式。

（2）进入循环前的初始值设定，包含循环控制变量和参与循环的变量的初始值的设定。

（3）设定控制循环终止的条件（即是否继续进行循环）。

（4）如何改变循环控制变量。

例 3.14 选择排序

```
public class Example3_14 {
    public static void main(String args[])
    {
        int []a={15,20,13,39,56,28,45,36};
        int n = a.length;
        int t,flag=-1;
        for(int i=0;i<n-1;i++)
        {
```

```
                t=i;
                for(int j=i+1;j<n;j++)
                {
                    if(a[j]<a[i])
                    {
                        flag=1;
                        t = j;
                    }
                }
                if(flag==1)
                {
                    flag=a[t];
                    a[t]=a[i];
                    a[i]=flag;
                    flag=-1;
                }
            }
            for(int i=0;i<n;i++)
                System.out.print(a[i]+"\t");
        }
    }
```

运行结果是：

```
    13  15  20  36  39  28  45  56
```

例 3.15 折半查找

```
    public class Example3_14 {
        public static void main(String args[])
        {
            int []a={13,15,20,28,36,39,45,56};//一个按升序排列的数组
            int low=0,high=a.length-1,mid=(low+high)/2;
            int x=36;               //待查找的数字
            int num=-1;
            while(low<=high)       //循环结束的条件low<=high
            {
                if(a[mid]==x)
                {
                    num=mid;
                    break;
                }
                if(a[mid]<x) low=mid+1;
                if(a[mid]>x) high=mid-1;
                mid=(low+high)/2;
            }
            if(num!=-1)
                System.out.println("数组a["+num+"]的值等于"+x);
```

```
            else
                System.out.println("数组a中找不到分量值为"+x+"的元素");
        }
    }
```

运行结果是：

数组 a[4]的值等于 36

小结

控制结构可用于描述实体的复杂行为，一个结构化程序只需三种基本控制结构，即顺序、选择和循环。设计控制结构时常用流程图等结构化设计工具，然后将这种形式的算法转换为 Java 语言编写的源程序。源程序中各种控制结构应采用合理的缩进，使程序的版面形式能直接体现程序的逻辑结构。

Java 语言最重要的选择结构是 if 语句，该语句有两种形式：不带 else 的 if 语句与带 else 的 if 语句。if 嵌套时的需要注意：Java 编译器是将 else 与离它最近的 if 组合在一起，除非用花括号{}指定不同的匹配方式。switch 语句主要用于多分支条件判断的情形。

Java 语言的循环结构包括 while 语句、do_while 语句和 for 语句。while 语句是最基本的循环形式，它先判断循环条件，再执行循环体；do_while 语句则先执行一次循环体后，才根据循环条件决定是否终止循环；for 语句是 while 语句的精简写法，适用于编写固定次数的循环。此外，循环体中还可用 break 语句强制跳出循环，或用 continue 语句提前开始下一次循环。

习题

一、基本概念

1、试说明 if-else 语句和 switch 语句各自适合用于怎样的分支结构。

2、试说明 switch 语句中 break 的作用

3、试说明 for、while、do-while 语句的特点。

4、试说明 Java 语言中 break 与 continue 语句的用途。

二、编程实践

1、编写一个 Java 程序，从键盘输入一元二次方程的 3 个系数 a、b 和 c，输出这个方程的解。

2、用 $\frac{\pi}{4}=1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\ldots$ 公式求 π

3、编写应用程序，输出满足 1+3+5+7+…+n<999 的最大整数 n。

4、用辗转相除法求两个正整数 m 和 n 的最大公约数和最小公倍数。

5、已知一个含有 20 个数值的整数序列，编写一个 Java 程序，将这个数列中的所有质数交换到前面，非质数交换到后面，并输出处理后的结果。

6、用二分法求下面方程在（-10，10）之间的根。

$$2x^3-4x^2+3x-6=0$$

7、打印出以下的杨辉三角形（要求打印出 10 行）。

```
1
1    1
1    2    1
1    3    3    1
1    4    6    4    1
1    5    10   10   5    1
```

第 4 章　Java 面向对象编程基础

4.1　面向对象技术基础

面向对象是一种新兴的程序设计方法，或者是一种新的程序设计规范，其基本思想是使用对象、类、继承、封装、消息等基本概念来进行程序设计。从现实世界中客观存在的事物（即对象）出发来构造软件系统，并且在系统构造中尽可能运用人类的自然思维方式。

4.1.1　面向对象的基本概念

1．对象的基本概念

对象是系统中用来描述客观事物的一个实体，它是构成系统的一个基本单位。一个对象由一组属性和对这组属性进行操作的一组服务组成。从更抽象的角度来说，对象是问题域或实现域中某些事物的一个抽象，它反映该事物在系统中需要保存的信息和发挥的作用；它是一组属性和有权对这些属性进行操作的一组服务的封装体，而客观世界正是由对象和对象之间的联系组成的。

2．类的基本概念

把众多的事物归纳、划分成一些类是人类在认识客观世界时经常采用的思维方法。分类的原则是抽象。类是具有相同属性和服务的一组对象的集合，它为属于该类的所有对象提供了统一的抽象描述，其内部包括属性和服务两个主要部分。在面向对象的编程语言中，类是一个独立的程序单位，它应该有一个类名并包括属性说明和服务说明两个主要部分。类与对象的关系就如模具和铸件的关系，类的实例化结果就是对象，而对一类对象的抽象就是类。

3．消息

消息就是向对象发出的服务请求，它应该包含下述信息：提供服务的对象标识、服务标识、输入信息和回答信息。服务通常被称为方法或函数。

4.1.2　面向对象的基本特征

1．封装性

封装性就是把对象的属性和服务结合成一个独立的相同单位，并尽可能隐蔽对象的内部细节，封装包含两个含义：

◇ 把对象的全部属性和全部服务结合在一起，形成一个不可分割的独立单位（即对象）。

◇ 信息隐蔽，即尽可能隐蔽对象的内部细节，对外形成一个边界（或者说形成一道屏障），只保留有限的对外接口使之与外部发生联系。

封装的原则在软件上的反映是：要求使对象以外的部分不能随意存取对象的内部数据（属性），从而有效的避免外部错误对它的“交叉感染”，使软件错误能够局部化，大大减少查错和排错的难度。

2．继承性

特殊类的对象拥有其一般类的全部属性与服务，称作特殊类对一般类的继承。例如，轮船、客轮；人、大人。一个类可以是多个一般类的特殊类，它从多个一般类中继承了属性与服务，这称为多继承。例如，客轮是轮船和客运工具的特殊类。在 Java 语言中，通常我们称一般类为父类（superclass，超类），特殊类为子类(subclass)。

3．多态性

对象的多态性是指在一般类中定义的属性或服务被特殊类继承之后，可以具有不同的数据类型或表现出不同的行为。这使得同一个属性或服务在一般类及其各个特殊类中具有不同的语义。例如：对“几何图形”来说，同样是 “绘图”方法， “椭圆”和“多边形”都是“几何图”的子类，但其“绘图”方法功能不同。

4.2　Java 中的类

在 Java 程序中，描述实体抽象概念的程序成分称为“类”，描述实体具体个体的程序成分称为“对象”。实体的属性被定义为类的数据成员，实体的行为被定义为类的方法成员。

4.2.1　类的定义

类是对一组具有相同属性、表现相同行为的对象的描述。Java 中类的定义格式如下：

[类修饰符] class 类名 [extendes 父类名] [implements 接口列表]

{

数据成员；

成员方法；

}

◇类修饰符（qualifier）：用于规定类的一些特殊性，比如访问限制、抽象类等。

◇extendes 父类名：指明新定义的类是由已存在的父类派生出来的。这个新定义的类就可以继承一个已存在类—父类的某些特征。

◇implements 接口列表： Java 本来只支持单继承，为了给多重继承的软件开发提供方便，它提供了这一接口机制。

例 4.1 一个简单的类——圆

```
class Circle {
    //数据成员
    double r;    //半径
    double area; //面积
    //成员方法
    void Area(double r){
        area = 3.14*r*r;
    }
    double getArea() {
        return area;
    }
}
```

4.2.2 类及其成员的访问控制符

在 Java 中，通过访问修饰符说明对类及其成员的访问限制。Java 提供了四种访问控制方式：缺省修饰符、public、private 及 protected。一个类或者成员可以没有修饰符，也可以用其他几种不同的修饰符，它们的作用是不同的，这将在后面章节逐一介绍，这里先给出所有访问控制方式的访问权限，见表 4-1。

表 4-1 Java 类成员的访问权限控制

访问控制符	同一个类中	同一个包中	不同包的子类	不同包的非子类
private	可以访问	×	×	×
default(缺省)	可以访问	可以访问	×	×
protected	可以访问	可以访问	可以访问	×
public	可以访问	可以访问	可以访问	可以访问

在声明 Java 类时，如果类名前无访问权限修饰符，则这个类只能被同一个包里的类使用。而 Java 规定，同一个程序文件中的所有类都在同一个包中。这也就是说，无修饰符的类可以被同一个程序文件中的类使用。

如果一个类的修饰符是 public，则这个类是公共类。公共类不但可供它所在包中的其它类使用，也可供其它包中的类使用。在程序中可以用 import 语句引用其它包中的 public 类。Java 规定，在一个程序文件中，只能定义一个 public 类，并且文件名应该同 public 类的名字相同，否则，编译时会报错。

4.2.3 数据成员的声明

数据成员的声明形式如下：

[修饰符] 数据成员类型 数据成员名表；

其中：

（1）修饰符是可选的，它是指访问权限修饰符 public、private、protected、缺省和非访问权限修饰符 static、final 等；

（2）数据成员类型可以是诸如 int、float 等 Java 中已定义基本类型和构造类型，也可以是用户自定义的构造类型；

（3）数据成员名表是指一个或多个数据成员名，即用户自定义标识符，当同时声明多个数据成员名时，彼此间用逗号分隔。

4.2.4 成员方法的声明

在 Java 程序中，成员方法的声明只能在类中进行，格式如下：

[修饰符] 返回值的类型 成员方法名（形式参数表）throws ［异常表］
{
    数据说明部分；
    执行语句部分；
}

成员方法的声明包括成员方法头和方法体两部分。其中，成员方法头确定成员方法的名字、形式参数的名字和类型、返回值的类型、访问限制和异常处理等；方法体由包括在花括号内的说明部分和执行语句部分组成，它描述该方法功能的实现。

在成员方法头中：

（1）修饰符。修饰符可以缺省，也可以是 public、private、protected 等访问权限修饰符，还可以有静态成员方法修饰符 static、最终成员方法修饰符 final、抽象成员方法修饰符 abstract 等非访问权限修饰符。访问权限修饰符指出满足什么条件时该成员方法可以被访问。非访问权限修饰符指明数据成员的使用方式；

（2）返回值的类型。返回值的类型可以用 Java 允许的各种数据类型关键字（例如，int，float 等）指明成员方法完成其所定义的功能后，要返回的运算结果值的数据类型。若成员方法没有返回值，则在返回值的类型处应写上 void 关键字，以表明该方法无返回值；

（3）成员方法名，为用户自定义标识符。

（4）形式参数表。成员方法可分为有参成员方法和无参成员方法两种。对于无参成员方法来说，没有形式参数表这一项，但成员方法名后的一对圆括号不可省略；而对于有参成员方法来说，形式参数表指明调用该方法所需要的参数个数、参数的名字及其参数的数据类型，其格式为：

（形式参数类型 1 形式参数名 1，形式参数类型 2 形式参数名 2，……）

（5）throw 异常表。它指出当该方法遇到一些设计者未曾想到的问题时如何处理。

例 4.2 一个银行帐户类

```
class Account {
    // 银行账户的属性
    private double balance = 0; // 存款余额
    // 向账户中存款，存款金额为amount
    public void deposit(double amount) {
        balance = balance + amount;
    }
    // 从账户中取款，取款金额为amount；取款成功返回true，否则返回false
    public boolean withdraw(double amount) {
        if (amount <= balance) {
            balance = balance - amount;
            return true;
        } else
```

```
            return false;
        }
        // 查询账户的当前余额
        public double getBalance() {
            return balance;
        }
    }
```

类成员（数据成员和成员方法）的访问权限修饰符见 4.2.2 节。这里，补充说明，在 private 后声明的成员称为私有成员，这些成员仅能由该类中声明的方法成员来访问。例如，例 4.2 中类 Account 的属性 balance 被声明为私有的，故只有在该类中声明的 deposit()、withdraw()和 getBalance()等方法体中才可访问该属性，在其他类中声明的方法体或在 main()方法体中均禁止访问该属性。

在 public 后声明的成员称为公有成员，这些成员是类与外部世界的接口，除该类本身声明的成员方法可访问这些成员外，其他类中声明的方法成员或 main()方法均可访问这些成员。例如，例 4.2 中类 Account 的行为 deposit()、withdraw()和 getBalance()被声明为公有的，因而其他程序片段都可访问这些行为。

如果类成员前无访问权限修饰符，则表示缺省访问权限，即该员仅在同一个包中可以访问。而 Java 规定，同一个程序文件中的所有类都在同一个包中，例如，例 4.1 中类 Circle 的所有数据成员和方法。这也就是说，无修饰符的类成员可以被同一个程序文件中的类使用，但不能被其它包中的程序文件使用。

设计类成员访问控制的最常用策略是：将所有数据成员声明为私有的，以实现信息隐藏；将所有方法成员声明为公有的，以供其他程序片段使用。这种设计模式的好处是，当一个类的属性的表示方法发生变化时，只需修改类中一些成员方法的方法体代码，只要该类能保持所有成员方法的接口及其实现的功能不变，那么上述变化就不会影响到使用该类的其他程序。

如果你设计的类中有一个数据成员需要对外公开，就要仔细研究其他程序片段是仅仅需要读取该数据成员的值，还是需要修改该数据成员的值。若是前者则仍应将数据成员声明为私有的，然后提供一个返回该数据的公有方法成员，比如例 4.1 中的 getArea()方法和例 4.2 中的 getBalance()方法；若是后者则有两种处理方式：一是将数据成员声明为公有的，二是将数据声明为私有的并提供一个公有的访问方法和一个公有的修改方法。

返回私有数据成员的那些公有方法成员又称 getter 方法，因为这些公有方法通常命名为 get…()（注意对布尔类型返回值通常命名为 is…()）；修改私有数据成员的那些公有方法成员又称 setter 方法，因为这些方法通常命名为 set...()。

例 4.3 学生类的定义

```
class Student {
    //数据成员
    String num;  //学号
    String name; //姓名
    int  age;  //年龄
    //针对数据成员的get方法和set方法
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getName() {
```

```
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getNum() {
        return num;
    }
    public void setNum(String num) {
        this.num = num;
    }
}
```

4.3 对象

若一个程序仅由这些类组成就无法完成任何功能。那么程序中应如何使用这些类呢？正确做法是利用这些类创建若干对象实例，让这些对象实例相互协作完成指定功能。

4.3.1 对象的创建与使用

程序中定义的类不仅可将复杂程序划分为一个个模块，而且每一个类都可作为一种用户自定义的类型使用。既然类是一种类型，那么它也可以像 int、double 等基本数据类型一样声明具体的实例，这些实例称为对象。类中定义的数据成员决定了该类的每一个对象实例的表示方法，方法成员决定了该类的每一个对象实例的可用操作。

类是创建对象的模板。当使用一个类创建对象时，我们也说我们给出了这个类的一个实例。通常的格式为：

类名 对象名 = new 类名([参数列表]);

Type objectName=new Type([parameterList]);

创建一个对象包括对象的声明、为对象分配内存空间和赋初值三个步骤。

1、对象的声明：

类的名字 对象名字;

如

People zhangPing;

或

People zhangPing, LiYong;

这里 People 是一个类的名字，zhangPing 是我们声明的对象的名字.

2、为声明的对象分配内存.

使用 new 运算符和类的构造方法为声明的对象分配内存，如果类中没有构造方法，系统会调用默认的构造方法。默认的构造方法是无参数的，构造方法的名字必须和类名相同。用 new 可以为一个类实例化多个不同的对象，这些对象分别占用不同的内存空间，因此改变其中一个对象的状态不会影响其他对象的状态。

3、最后一步是执行构造方法，进行初始化。

zhangPing=new People(“20040101”); //实例化一个对象

上面三个步骤，通常可以写成如下简洁的形式：

People zhangPing=new People(“20040101”);

当用一个类创建了一个对象之后，该对象就可以通过“.”运算符操作自己的变量、使用类中的方法。

例 4.4 生成例 4.2 中 Account 类的对象并使用成员方法

```
public class Example4_4 {
```

```
    public static void main(String[] args) {
        //  为张三开设一个账户
        Account zhang3 = new Account();
        //  张三的账户存入500元后又取出100元
        zhang3.deposit(500);
        if (!zhang3.withdraw(100)) System.out.println("余额不足，取款失败！");
        //  为李四开设一个账户
        Account li4 = new Account();
        //  从张三的账户取出150元存入李四的账户
        if (!zhang3.withdraw(150)) System.out.println("余额不足，转账失败！");
        else li4.deposit(150);
        //  查询张三和李四的账户余额
        System.out.println("张三的账户余额为 " + zhang3.getBalance());
        System.out.println("李四的账户余额为 " + li4.getBalance());
    }
}
```

运行结果是：

```
张三的账户余额为 250.0
李四的账户余额为 150.0
```

例 4.5 简单的商品入出库

```
// 单一商品的库存
class Inventory {
    private int quantity = 0; // 存货的数量
    // 验收入库，入库数量为qty
    public void checkIn(int qty) {
        quantity += qty;
    }
    // 领料出库，出库数量为qty
    public int checkOut(int qty) {
        // 如果数量余额不足则不作出库
        if (quantity < qty) return 0;
        // 减少库存数量与金额
        quantity -= qty;
        return 1;
    }
    // 查询当前的存货数量余额
    public int getQuantity() {
        return quantity;
    }
}
public class Example4_5{
    public static void main(String args[])
    {
        Inventory inv = new Inventory();
```

```
        System.out.println("入库前库存为:"+inv.getQuantity());
        inv.checkIn(100);
        System.out.println("入库100件后库存为:"+inv.getQuantity());
        inv.checkOut(20);
        System.out.println("出库20件后库存为:"+inv.getQuantity());
    }
}
```

运行结果是：

```
入库前库存为:0
入库100件后库存为:100
出库 20 件后库存为:80
```

4.3.2 构造方法

每当由类构造对象时都要调用该类的构造方法，在 Java 中，每个类都至少有一个构造方法。构造方法可以确保用户正确地构造类的对象，同时，构造方法也会对对象作初始化工作。

构造方法说明形式如下：

[构造方法修饰符] 方法名（[形式参数列表]） [throws 异常列表]

{ 方法体 }

构造方法修饰符与一般方法修饰符相同，可参见 4.2.2 小节。

一个类的构造方法可以有多个，它们都具有相同的方法名，即类名。编译器根据参数的类型来决定调用哪个构造方法。这就是构造方法的多态。

构造方法分为缺省的构造方法（不带参数）和带参数的构造方法。

（1）缺省的构造方法 如果类的定义没有编写构造方法，Java 语言会自动为用户提供。这个由 Java 自动提供的构造方法就是所谓的缺省构造方法。缺省的构造方法确保每个 Java 类都至少有一个构造方法，该方法应符合方法的定义。

例如在例 4.1 中的 Circle 类、例 4.2 中的 Account 类中，我们都没有定义构造方法，则 Java 自动提供了一个缺省的构造方法，如下：

```
public CircleEmpInfo(){}
public Account(){}
```

（2）带参数的构造方法

带有参数的构造方法能够实现这样的功能：当构造一个新对象时，类构造方法可以按需要将一些指定的参数传递给对象的变量。

例 4.6 带参数的构造方法——雇员类的构造与使用

```
class EmpInfo {
    String name;         // 雇员的姓名
    String designation;  // 雇员的职务
    String department;   // 雇员的部门
    // 带参数的构造方法
    public  EmpInfo(String name,String designation,String department){
        this.name=name;
        this.designation=designation;
        this.department=department;
    }
    void print() {
        System.out.println(name + " is " + designation + " at " + department);
```

```
    }
    public static void main(String argv[]){
        EmpInfo employee =new EmpInfo("张三","经理","茶餐厅");
        employee.print();
    }
}
```

运行结果是：

```
张三 is 经理 at 茶餐厅
```

下面我们构造一个信用卡类，信用卡与我们前面构造的普通银行账户的最大区别是允许透支，因此，我们引入一个表示透支限额的属性 maxOverdraft，该属性在构造信用卡对象时，就自动赋一个初值作为透支额度。

例 4.7 带参数的构造方法——信用卡类的构造

```
class CreditCard {
    // 信用卡账户的属性
    private double maxOverdraft; // 透支限额
    private double balance; // 存款余额
    // 构造方法，将透支限额maxOverdraft设置为max
    public CreditCard(double max) {
        maxOverdraft = max;
    }
    // 向账户中存款，存款金额为amount
    public void deposit(double amount) {
        balance = balance + amount;
    }
    // 从账户中取款，取款金额为amount；取款成功返回true，否则返回false
    public boolean withdraw(double amount) {
        if (amount <= balance + maxOverdraft) {
            balance = balance - amount;
            return true;
        } else return false;
    }
    // 查询账户的当前余额
    public double getBalance() {
        return balance;
    }
    //查询账户的当前余额
    public double getMaxOverdraft() {
        return maxOverdraft;
    }
    public static void main(String args[])
    {
        CreditCard cc = new CreditCard(1000);
        double money=cc.getBalance()+cc.getMaxOverdraft();
        System.out.println("余额为:"+cc.getBalance()+"透支额度
```

```
为:"+cc.getMaxOverdraft()+"可支取:"+money);
            cc.deposit(500);
            money=cc.getBalance()+cc.getMaxOverdraft();
            System.out.println("余额为:"+cc.getBalance()+"透支额度
为:"+cc.getMaxOverdraft()+"可支取:"+money);
            cc.withdraw(1200);
            money=cc.getBalance()+cc.getMaxOverdraft();
            System.out.println("余额为:"+cc.getBalance()+"透支额度
为:"+cc.getMaxOverdraft()+"可支取:"+money);
        }
    }
```

运行结果：

```
余额为:0.0透支额度为:1000.0还可支取:1000.0
余额为:500.0透支额度为:1000.0还可支取:1500.0
余额为:-700.0 透支额度为:1000.0 还可支取:300.0
```

从上面的例子中可以看出，构造方法不能像一般的方法那样被直接调用，它是在构造类的实例的时候被 new 关键字调用的。当我们构造一个类的实例的时候，编译器主要完成以下三件事情：

（1）为对象分配内存空间；

（2）初始化对象中的实例变量的值，初始值可以是缺省值，或者变量按指定的值初始化；

（3）调用对象的构造方法。

下面的例子测试构造方法执行前后的变化。

例 4.8

```
class Variable{
    int x=0,y=0,z=0; //类成员变量
    void init(int x,int y){
        this.x=x;
        this.y=y;
        int z=5;  // 局部变量
        System.out.println("****正在初始化****");
        System.out.println(" x="+x+" y="+y+" z="+z);
    }
}
public class Example4_8{
    public static void main(String args[]){
        Variable v=new Variable();
        System.out.println("****初始化前****");
        System.out.println(" x="+v.x+" y="+v.y+" z="+v.z);
        v.init(20,30);
        System.out.println("****初始化后****");
        System.out.println(" x="+v.x+" y="+v.y+" z="+v.z);
    }
}
```

运行结果：

```
****初始化前****
```

```
 x=0 y=0 z=0
****正在初始化****
 x=20 y=30 z=5
****初始化后****
 x=20 y=30 z=0
```

4.4 访问权限

对象操作自己的变量和使用类中的方法是有一定限制的，即类中成员是有访问权限的许可。所谓访问权限是指对象是否可以通过“.”运算符操作自己的变量或使用类中的方法。访问限制修饰符有 private、protected 和 public 以及缺省，具体作用在 4.2.2 节的表 4-1 中给出。本节介绍 public 和 private 修饰符的使用，protected 和缺省修饰符在后面章节介绍。

4.4.1 public 修饰符

如果将一个成员变量或成员方法定义为 public 类型，则在同一类、子类、同一包中的类、不同的包中的类均可以访问该成员变量或成员方法。

```
class Demopub1{
    public int a;          //公共类型的成员变量
    public void method()   //公共类型的成员方法
    {
        System.out.println();
    }
}
```

Demopub1 类中的数据成员 a 和方法 method()都是 public 的，所以在该类的对象在任何地方都可以访问。

4.4.2 private 修饰符

用关键字 private 修饰的数据成员和方法称为私有变量和私有方法。如果一个变量或成员声明为私有类型，则该变量或方法只能在同一类中被访问。如：

```
class DemoPrivate{
    public int a; //  公共类型的成员变量
    private int b; //  私有类型的成员变量
    public int getA(){ //公共类型的成员方法
            return a;
    }
   public int getB(){    //私有类型的成员方法
        return b;
   }
    public DemoPrivate(int a,int b) { //构造方法
        this.a = a;
        this.b = b;
    }
}
```

DemoPrivate 类中，因为数据成员 b 变声明为私有，所以 DemoPrivate 的任何对象都不能直接通过“.”运算符访问变量 b（变量 a 可直接通过“.”运算符访问），如果要想获取 b 的值，该类的对象只能通过“.”运算符使用 getB()方法获取。

4.5 方法的重载

方法重载（method overloading）就是一个类中可以有多个方法具有相同的名字，但这些方法的参数

必须不同，或者是参数的个数不同，或者是参数的类型不同。 这也是面向对象的程序设计中的奇妙之处，重载反映了大千世界的变化。

从另外的含义上来讲，重载也可以看成是同一个方法具有不同的版本，每个版本之间在参数特征方面有差别。重载是 Java 实现多态性的一种方式。

当调用一个重载方法时，JVM 自动根据当前对方法的调用形式在类的定义中匹配形式符合的成员方法，匹配成功后，执行参数类型、数量均相同的成员方法。方法重载在 Java 的 API 类库中得到大量的使用。

例 4.9 关于成员方法重载的例子

```
class Demo{
   int a,b;
   int method(){          //成员方法一
        return a+b;
   }
   int method(int c){     //成员方法二
         return a+b+c;
   }
   int method(int c,int d){ //成员方法三
         return a+b+c+d;
    }
   Demo(int a,int b) {     //构造方法
        this.a = a;
        this.b = b;
    }
}
public class Example4_8{
    public static void main(String args[]){
       Demo aDemo = new Demo(1,2); //实例化
       int a = aDemo.method(); //调用成员方法一
       System.out.println(a);
       int b = aDemo.method(3); //调用成员方法二
       System.out.println(b);
       int c = aDemo.method(3,4);//调用成员方法三
       System.out.println(c);
    }
}
```

运行结果：

```
3
6
10
```

从上面的程序中可以看到成员方法的重载。在该程序中，方法 method()被定义了三次，每次参数不同。在调用该方法时，由于使用的是同一方法名，因此需要根据方法的参数才能找到需要运行哪个方法。

例 4.10 构造方法的重载

```
class Demo2{
   int a,b,c;           //成员变量
```

```
    public Demo2(){}     //构造方法1
    public Demo2(int a) {  //构造方法2
         this.a = a;
    }
    public Demo2(int a,int b) { //构造方法3
         this.a = a;
     this.b = b;
  }
    public Demo2(int a,int b,int c) {//构造方法4
         this.a = a;
     this.b = b;
     this.c = c;
  }
  }
  public class Example4_9{
  public static void main(String args[]){
        //应用第一种构造方法
    Demo2 aDemo = new Demo2();
       System.out.println("方法一成员变量a："+aDemo.a);
       System.out.println("方法一成员变量b："+aDemo.b);
       System.out.println("方法一成员变量c："+aDemo.c);
      //应用第二种构造方法
       Demo2 bDemo = new Demo2(1);
       System.out.println("方法二成员变量a："+bDemo.a);
       System.out.println("方法二成员变量b："+bDemo.b);
       System.out.println("方法二成员变量c："+bDemo.c);
     }
  }
```

运行结果：

```
方法一成员变量a：0
方法一成员变量b：0
方法一成员变量c：0
方法二成员变量a：1
方法二成员变量b：0
方法二成员变量 c：0
```

从上面的程序中可以看到构造方法的重载。在该程序中，方法 Demo2()被定义了四次，每次参数不同。第一个构造方法，没有参数，也没有方法体，它和系统的缺省构造方法是一致的。缺省的构造方法确保每个 Java 类都至少有一个构造方法。如果程序中给出了带参数的构造方法，而没有给出缺省构造方法，这时调用缺省构造方法将导致错误。

在调用构造方法时，由于使用的是同一方法名，因此根据构造方法的参数才能找到需要运行哪个方法。

4.6 static、final 修饰符

4.6.1 static 修饰符

被 static 修饰符修饰的数据成员或成员方法被称为类成员，是类的静态成员（变量被称为类变量，

方法被称为类方法)。无 static 修饰的成员称为实例成员（变量被称为实例变量，方法被称为实例方法)。

在生成每个类的实例对象时，Java 运行时系统会为每个对象的实例变量分配一块内存，然后可以通过该对象来访问这些实例变量。不同对象的实例变量是不同的。而对于类变量来说，在生成类的第一个实例对象时，Java 运行时系统对这个对象的每个类变量分配一块内存，以后再生成该类的实例对象时，这些实例对象将共享同一个类变量，每个实例对象对类变量的改变都会直接影响到其他实例对象。

类变量可以通过类名直接访问，也可以通过实例对象来访问，因为它类变量被保存在类的内存区的公共存储单元中，而不是保存在某个对象的内存区中。因此，一个类的任何对象访问它时，存取到的都是相同的数值；两种方法的结果是相同的。

例 4.11 类变量的使用

```
class Demo3{
    static int a;
    public void display(){
        System.out.print(" a="+a);
    }
}
public class Example4_11{
    public static void main(String args[]){
    Demo3 a1=new Demo3(); a1.a=10;
    Demo3 a2=new Demo3(); a2.a=20;
    Demo3.a=50;
    a1.display();
    a2.display();
    }
}
```

运行结果：

```
a=50 a=50
```

实例方法可以对当前对象的实例变量进行操作，也可以对类变量进行操作，但类方法不能访问实例变量。实例方法必须由实例对象来调用，而类方法除了可由实例对象调用外，还可以由类名直接调用。另外，在类方法中不能使用 this 或 super。与类方法相比，实例方法没有这些限制。

例 4.12 类方法和类变量的使用

```
class member {
    static int classVar;
    int instanceVar;
    static void setClassVar(int i){
        classVar=i;
        // instanceVar=i; //在类方法中不能引用实例成员。
    }
    static int getClassVar( ){
        return classVar;
    }
    void setInstanceVar(int i ){
        classVar=i;
        instanceVar=i;
    }
```

```
        int getInstanceVar(){
            return instanceVar;
        }
    }
    public class Example4_12{
        public static void main(String args[]){
            member  m1=new member( );
            member  m2=new member( );
            m1.setClassVar(1);
            m2.setClassVar(2);
            System.out.println("m1.classVar="+m1.getClassVar( )+ "
m2.classVar="+m2.getClassVar( ));
            m1.setInstanceVar(11);
            m2.setInstanceVar(22);
            System.out.println("m1.InstanceVar="+m1.getInstanceVar( )+ "
m2.InstanceVar="+m2.getInstanceVar( ));
        }
    }
```

运行结果：

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

4.6.2 final 修饰符

1．final 修饰成员变量

如果一个成员变量前面有 final 修饰，那么这个成员变量就变成了常量，一经赋值，就不允许在程序的其他地方修改。定义方式如下：

final type variableName；

例如

```
class ConstTimeExpress{
      final int MaxHour=23;
      final int MaxMinute=59;
  final int MaxSecond=59;
}
```

2．final 修饰方法

方法的 final 修饰符表明方法不能被子类覆盖。带有 final 修饰符的方法称为最终方法。Java 的方法除非被说明为最终方法，否则方法是可以被覆盖的。Java 之所以这样规定，主要是因为 Java 的纯面向对象特性，它把覆盖当作面象对象的重要特性，给予了最大限度的实现。

把方法声明为最终方法有时可增加代码的安全性。

使用方式如下。

```
 final   returnType   methodName(paramList){
 }
```

例如：

```
 final int getLength(String s){
    ……
 }
```

3．final 修饰类

final 类不能被继承。由于安全性的原因或者是面向对象的设计上的考虑，有时候希望一些类不能被继承，例如，Java 中的 String 类，它对编译器和解释器的正常运行有很重要的作用，不能轻易改变它，因此把它修饰为 final 类，使它不能被继承，这就保证了 String 类型的唯一性。同时，如果你认为一个类的定义已经很完美，不需要再生成它的子类，这时也应把它修饰为 final 类。

当用 final 修饰符修饰类时，所有包含在 final 类中的方法，都自动成为 final 方法。

定义一个 final 类的格式如下：

final class finalClassName{

……

}

4.7 参数的传递

4.7.1 方法参数的传递

当方法被调用时，如果方法有参数，参数必须要实例化，即参数变量必须有具体的值。在 Java 中，仅提供了一种参数传递方式，即按值调用，但由于在 Java 语言的对象模型中所有对象名都表示对象的引用而不是对象本身，所以对象作为参数传递的效果与按引用调用的参数传递方式相同。

Java 的数据类型我们可以分为基本数据类型和引用数据类型两大类，基本数据类型作为参数传递时传递的是值。引用数据类型包括对象、数组以及接口，当参数是引用类型时，“传值”传递的是变量的引用而不是变量所引用的实体。

下面的例子演示了 Java 中基本数据类型和引用类型参数的传递。

例 4.13 基本数据类型和引用类型参数的传递

```
class People
{
    int money;
    void setMoney(int n){
        money = n;
    }
}
class A
{
    void f(double y,People p){
        y=y+1;
        p.setMoney(200);
        System.out.println("在方法f中改变参数y的值为"+y);
        System.out.println("在方法f中参数对象p的money改变为"+p.money);
    }
}
public class Example4_13 {
    public static void main(String args[])
    {
        double y=0.8;
        People zh=new People();
        zh.setMoney(1888);
        A a = new A();
        System.out.println("在方法f被调用之前zh的money是:"+zh.money);
```

```
        System.out.println("在方法f被调用之前main中y的值是:"+y);
        a.f(y, zh);
        System.out.println("在方法f被调用之后zh的money变为:"+zh.money);
        System.out.println("在方法f被调用之后main中y的值仍然是:"+y);
    }
}
```

运行结果：

```
在方法f被调用之前zh的money是:1888
在方法f被调用之前main中y的值是:0.8
在方法f中改变参数y的值为1.8
在方法f中参数对象p的money改变为200
在方法f被调用之后zh的money变为:200
在方法f被调用之后main中y的值仍然是:0.8
```

4.7.2 命令行参数的使用

在 Java 应用程序中我们必须写 public static void main(String[] args)主方法。main 方法中有一个参数是字符串数组 args，这个数组的元素 args[0],args[1]，…，args[n]的值都是字符串。args 就是命令行的参数。在 Java 解释器解释用户的字节码文件时，可以包括需要传给 main 方法的参数。一般形式为：

java 类文件名 字符串 1 字符串 2 …… 字符串 n

类文件名和各字符串间用空格分隔。在下面的例子中我们展示如何使用 main()方法的这个参数。

例 4-14 从命令行输入参数

```
public class Example4_14{
   public static void main(String[] args)  {
     for(int i=0;i<args.length;i++)  {
       System.out.println("args[" +i+"]:"+args[i]);
     }
   }
}
```

在控制台输入 java MainArgument One Two Three 后的运行结果：

```
args[0]:One
args[1]:Two
args[2]:Three
```

从上面的例子可以看出，形如“java 类文件名 字符串 1 字符串 2 …… 字符串 n”的命令行运行方式中，可以通过 args[]数组访问命令行中的各个字符串参数，并且其对应关系是字符串 1 的值保存在 args[0]中，字符串 2 的值保存在 args[1]，依此类推。同时还应注意，命令行中的参数都是以字符串形式获取的，如果需要转换为其他类型，必须用显示转换。

小结

本章主要阐述了面向对象的抽象性和封装性在 Java 程序中的实现技术，其中包括类的定义、对象的创建和访问，类成员的访问权限控制，并给出了成员变量、成员方法、构造方法、静态成员等一系列概念的应用。

类是构造程序的基本单位，它有两方面作用：一是作为封装和信息隐藏机制，二是作为类型定义机制。访问控制决定了类的哪些成员可由外部访问，哪些成员禁止外部访问。通常做法是将描述实体属性的数据成员定义为私有的，将描述实体行为的方法成员定义为公有的。

每一对象实例都占用一些存储空间以保存其内部状态，创建对象时系统将自动调用类的构造方法来初始化对象，程序员自己无法显式地调用构造方法。静态数据成员让一个类的所有对象实例共享同一存

储空间，从而使得这些对象之间可以通信。

与基本数据类型的参数不同，对象作为参数时实际参数的对象状态可能在方法调用后被改变，而基本数据类型的实际参数在方法调用后其值不会被改变。

要注意区别 Java 语言中的变量和对象实例，变量由 JVM 自动创建和撤销，对象实例是由 new 创建的一片匿名存储区域，只能通过引用它的变量对其进行操作。对象实例的撤销由 JVM 的垃圾回收机制自动完成。

习题

一、基本概念

1、试用现实生活中的例子说明类与对象这两个概念的区别，简述 Java 语言是如何支持面向对象的抽象和封装概念？

2、在 Java 程序中可以通过哪几个途径对成员进行初始化？

3、阐述默认访问属性、public 访问属性、private 访问属性和 protected 访问属性的应用场合。

4、什么是静态成员？如何对静态成员变量进行初始化？如何对静态成员进行操作？举例说明在什么情况下选择使用静态成员？

5、举例说明类变量与实例变量的区别。

6、试从类型、声明、初始化和引用方式等方面比较 Java 语言的对象和变量的异同?

7、什么是方法重载？构造方法可以重载吗？

二、编程实践

1、以下三个程序能否通过编译？如果不能通过编译，请指出产生错误的原因以及改正的办法。

(1)
```
public class Location {
    public void Location(int a, int b) { x = a; y = b; }
    public int getX() { return x; }
    public int getY() { return y; }
    public static void main(String[] args) {
        Location airport = new Location(5, 8);
        System.out.println(airport.getX());
        System.out.println(airport.getY());
    }
    private int x;
    private int y;
}
```

(2)
```
public class Square {
    private double length;
    private double width;
    public Square(double x, double y) { length = x; width = y; }
    public static double perimeter() { return 2 * (length + width); }
    public static double area() { return length * width; }
    public static void main(String[] args) {
        Square myHome = new Square(5.5, 4.2);
        System.out.println(myHome.perimeter());
        System.out.println(myHome.area());
    }
}
```

(3)
```
public class Circle {
```

```
    private double xPos = 0;
    private double yPos = 0;
    private double radius = 1;
    public Circle(double x, double y, double r) { xPos=x; yPos=y; radius=r;}
    public double perimeter() { return 2 * 3.14 * radius; }
    public double area() { return 3.14 * radius * radius; }
    public static void main(String[] args) {
        Circle figure = new Circle();
        System.out.println(figure.perimeter());
        System.out.println(figure.area());
    }
}
```

2、设计一个一元二次方程类 Equation，然后再编写一个 Java 应用程序，对该类对象表示的一元二次方程进行创建、显示和求解等操作。

3、试设计一个身份证类，其中包含姓名、性别、民族、出生年月日、住址、身份证编码、签证机关和有效期限。

4、试设计一个记录某门课程的所有同学的考试成绩的成绩单类 Score Report，其中除了应该包含课程编码、课程名称、考试日期、教师姓名以及每个学生的学号、姓名、成绩等信息外，还应该包含与之相关的所有行为方法。

5、设计一个带有实例计数器的类，该类最多只允许创建 3 次对象实例，如果创建过的对象实例则在屏幕上打印出错信息。用一个 main()方法演示该类的用法。（提示：引入静态数据成员 instanceCount 使同一类的对象实例之间可共享信息；题目仅要求约束创建对实例的次数不超过 3 次，并未限制当前正在使用的对象实例仅有 3 个）

第 5 章　Java 面向对象编程进阶

5.1 继承

在日常生活中，我们常用“IS-A”关系来组织与表达知识，从而将知识组织成一种有层次、可分类的结构。这种层次结构的分类方法在许多领域中都应用得非常广泛，例如生物学家用界、门、纲、目、科、属、种七个主要层次对数以百万计的生物进行分类，每种分类都用若干特征刻划。在每个层次中同一分类的生物具有共同的特征，并可以与其他分类区别开来。

例如猫是一种猫科动物，而猫科动物是一种哺乳动物，哺乳动物又是一种脊椎动物，脊椎动物是一种动物。即使我们没有见过名为 Tom 和 Mimi 的两只猫，也大概对它们有所了解，因为它们都具备了猫的外形、哺乳动物的行为等。但 Tom 和 Mimi 作为两个特定的个体，在某些方面有自己的个性，如体重、毛色、眼睛颜色等，这是我们在第四章讨论过的类与对象实例之间的区别。

因此，继承是一种由已有的类创建新类的机制。利用继承，我们可以先创建一个拥有共同属性的一般类，根据该一般类再创建具有特殊属性的新类。由继承而得到的类称为子类(subclass)，被继承的类称为父类（或叫超类，superclass）。直接或间接被继承的类都是父类。子类继承父类的状态和行为，同时也可以修改父类的状态或重写父类的行为，并添加新的状态和行为。与 C++不同的是，Java 语言不支多重继承。

作为类型构造机制，如果类 B 继承类 A，则所有要求对象为 A 类型的地方也可以接受 B 类型的对象，这时我们称 B 类型为 A 类型的子类型。正如“苹果”是一种“水果”，所有出现“水果”的地方用“苹果”来代替是合理的；但反之则不然，出现“苹果”的地方不一定能用“水果”代替。

作为类的构造机制，继承是面向对象程序设计语言中支持软件重用的一种重要方式，而作为类型构造机制，继承是实现动态多态性的基础。

5.1.1 Java 语言的继承语法

继承机制的基本原则是从已有的类（超类 / 父类）定义新的类（子类），使得子类拥有父类的属性和方法，这些属性和方法成为子类的继承成员。在定义子类时，还可对这些继承成员进行调整，例如重定义其中的某些方法等。因此要学习某种面向对象语言的继承机制，需要掌握如何定义子类，如何在子类中使用继承成员，以及如何调整继承成员等主要内容。

在 Java 语言中，一个名为 SubClass 的类继承另一个已有类 ParentClass 的语法形式为：

```
public class SubClass extends ParentClass {
    成员列表;
};
```

其中，类 ParentClass 称为类 SubClass 的基类，新定义的类 SubClass 称为 ParentClass 的派生类，extends 是 Java 语言的保留字，指明类之间的继承关系。

花括号中定义的是 SubClass 中新增加的数据成员与成员函数。第四章已经讨论过由保留字 public 与 private 定义的类成员访问控制方式，在 Java 语言中还有一种类成员访问控制方式，用保留字 protected 定义，在 protected 后定义的数据成员或成员方法称为受保护成员。

受保护成员具有公有成员与私有成员的双重角色：它可以被后代类的成员方法访问，但不可以被其他不相关的类的成员方法所访问。后面我们还会详细地讨论继承对成员访问控制规则的影响。

我们通过一个实例进一步了解 Java 语言的继承机制。假定我们根据学生的三门学位课程的分数决定其是否可以拿到学位，对于本科生，如果三门课程的平均分超过 60 分即表示通过，而对于研究生，则需要平均分超过 80 分才能通过。下面先给出基类 Student 表示学生的共同特征：

例 5.1

```
//Sudent类
public class Student {
    public final static int NUM_OF_SCORE = 3; // 每个学生的分数个数
```

```
    private String name; // 姓名，私有成员
    protected int[] scores; // 分数，受保护成员
    public Student() {
        name = "";
        scores = new int[NUM_OF_SCORE];
    }
    public void setName(String newName) {
        name = newName;
    }
    public String getName() {
        return name;
    }
    public void setScore(int scoreNum, int score) {
        // 检查参数的正确性
        if (scoreNum < 0 || scoreNum >= NUM_OF_SCORE) return;
        if (score < 0 || score > 100) return;
        scores[scoreNum] = score;
    }
    public int getScore(int scoreNum) {
        if (scoreNum < 0 || scoreNum >= NUM_OF_SCORE) return -1;
        return scores[scoreNum];
    }
}
// 本科生类，继承学生类
public class UndergraduateStudent extends Student {
    public String computeGrade() {
        int total = 0;
        for (int i = 0; i < NUM_OF_SCORE; i++)
            total = total + scores[i];
        if (total / NUM_OF_SCORE > 60) return "PASS";
        else return "NO PASS";
    }
}
// 研究生类，继承学生类
public class GraduateStudent extends Student {
    public String computeGrade() {
        int total = 0;
        for (int i = 0; i < NUM_OF_SCORE; i++)
            total = total + scores[i];
        if (total / NUM_OF_SCORE > 80) return "PASS";
        else return "NO PASS";
    }
}
```

上述两个子类 UndergraduateStudent、GraduateStudent 的成员方法 computeGrade()中都访问了父类的

受保护成员 scores，这是合法的访问，因为子类的成员方法可以访问父类的受保护成员。下面程序简单地演示了这几个类的用法。

例 5.2

```
public class StudentDemo {
    public static void main(String[] args) {
        GraduateStudent student1 = new GraduateStudent();
        student1.setName("张三");
        student1.setScore(0, 78);
        student1.setScore(1, 92);
        student1.setScore(2, 72);
        System.out.println("研究生 " +
student1.getName()+student1.computeGrade());
        UndergraduateStudent student2 = new UndergraduateStudent();
        student2.setName("李四");
        student2.setScore(0, 80);
        student2.setScore(1, 78);
        student2.setScore(2, 75);
        System.out.println("本科生
"+student2.getName()+student2.computeGrade());
    }
}
```

运行结果：

研究生 张三不能毕业

本科生 李四可以毕业

在例 5.2 中，通过对象 student1 和 student2 访问了父类 student 的公有成员方法 setName()和 setScore()，这是合法的访问，因为子类继承了父类的成员，而且父类的公有成员被子类继承后仍然是公有成员。

5.1.2 继承与成员访问控制

继承机制引入了受保护成员，提供了一种新的成员访问控制级别，因此我们需要进一步了解 Java 语言的类成员访问控制规则。通过第四章的学习我们知道，一个类自身的成员方法可以访问它所有的成员，而其他类的成员方法则只能访问这个类的公有成员，不能访问这个类的私有成员，这是 Java 语言最基本的成员访问控制规则。

在引入继承机制后，子类继承了父类除构造方法以外的所有成员，这些成员称为子类的继承成员。注意，继承成员不仅包括在父类中定义的公有、受保护、私有成员，还包括父类的继承成员，即父类继承自它的祖先类得到的成员。在子类的内部，不仅能够访问子类自己定义的所有成员，也能够访问父类中公有和受保护的成员，但不能访问父类中私有的成员。

例如，在程序例 5.1 的子类 UndergraduateStudent 继承了父类 Student，它从父类 Student 得到的继承成员包括父类的私有成员 name，受保护成员 scores，公有成员 NUM_OF_SCORE、setName()、getName()、setScore()和 getScore()，而方法 computeGrade()则是它自己定义的成员。在子类 UndergraduateStudent 的内部既能访问它自己定义的成员 computeGrade()，也能访问继承成员 scores、NUM_OF_SCORE、setName()、getName()、setScore()和 getScore()，但不能访问继承成员 name。

子类的某些继承成员也能被使用子类的程序访问，而哪些成员能被使用子类的程序访问则由继承成员在子类中的访问控制决定。继承成员在子类的访问控制与它们在父类中的访问控制相同，即原先在父类是公有的成员，被子类继承后仍然是子类的公有（继承）成员，原先在父类是受保护的成员，被子类

继承后仍然是子类的受保护（继承）成员，原先在父类是私有的成员，被子类继承后仍然是父类的私有成员（注意父类的私有继承成员即使是在子类内部也不能访问）。

因此，使用子类的程序能够访问子类的公有成员（包含继承的公有成员），但不能访问子类的受保护成员，更不能访问子类的私有（继承）成员。

例 5.2 的类 StudentDemo 就是使用子类 UndergraduateStudent 的程序，它能通过子类 UndergraduateStudent 的对象引用 student2 访问它的公有继承成员 NUM_OF_SCORE、setName()、getName()、setScore()和 getScore()，但不能直接访问受保护的继承成员 scores，更不能直接访问私有继承成员 name。

总之，子类可继承父类的所有成员，这些成员是子类的继承成员，但在子类内部只能访问公有和受保护的继承成员，但不能直接访问私有的继承成员。

例 5.3

```
class A {
   private int x;
   protected int y;
   public void setX(int x) {
      this.x=x;
   }
   public int getX() {
      return x;
   }
    protected int getY() {
        return y;
    }
    protected void setY(int y) {
        this.y = y;
    }
}
class B extends A {
   int z=12;
   public void setZ(int z)
   {
       //下面的对x的访问非法，子类虽然继承了父类的x属性,但不能直接访问父类的私有成员
       //this.y=y+x;
       //正确的访问方式,子类可以访问父类的公有和保护成员
       this.z = z+getX()+y;
   }
   public int getZ() {
       return z;
   }
}
public class Example5_3 {
    public static void main(String args[]) {
        B b=new B();
        b.setX(10);
```

```
            System.out.println("子类对象继承的父类的私有成员x的值是(不能直接访
问):"+b.getX());
            b.setY(20);
            System.out.println("子类对象继承的父类的保护成员y的值是:"+b.y);
            b.setZ(30);
            System.out.println("子类对象的实例变量z的值是:"+b.getZ());
        }
    }
```

运行结果:

```
子类对象继承的父类的私有成员x的值是(不能直接访问):10
子类对象继承的父类的保护成员y的值是:20
子类对象的实例变量 z 的值是:60
```

5.1.3 java.lang.Object 类介绍

Java 的类继承层次是一种单根层次结构，Java 程序中的每个类都是类 Object 的后代类。如果一个类没有指明其超类，则它的超类隐含地是类 Object。因此例 5.1 的类 Student 是类 Object 的子类，它继承了类 Object 的成员，这样类 UndergraduateStudent 继承类 Student 的继承成员除了上面所给出的成员之外也包括了类 Object 的成员。

类 java.lang.Object 处于 java 开发环境的类层次的根部，其它所有的类都是直接或间接地继承了此类。该类定义了一些最基本的状态和行为。下面，我们介绍一些常用的方法见表 5-1。

表 5-1 java.lang.Object 类的常用方法

方法名	说明
clone( )	创建与该对象的类相同的新对象
equals( )	比较两对象是否相等同
finalize( )	用于在垃圾收集前清除对象
getClass( )	返回对象运行时所对应的类表示，从而可得到相应的信息
hashCode( )	返回该对象的散列码值
toString( )	返回该对象的字符串表示
notify( )	激活等待队列中的一个线程
notifyAll( )	激活等待队列中的全部线程
wait( )	等待该对象另一更改线程的通知

instanceof 运算符

instanceof 运算符是一个双目运算符，左面的操作数是一个对象，右面是一个类。当左面的对象是右面的类创建的对象时，该运算符运算的结果是 true，否则是 false。例如：

```
if(b1 instanceof Button)  doDealB1();
```

其中 b1 是对象，Button 是一个类名。

例 5.4 getClass 和 Instanceof 方法的使用

```
class SubClass {}
class SuperClass extends SubClass {}
public class ClassAndInstance {
    public static void main(String[] args) {
        test(new SubClass());
        test(new SuperClass());
    }
```

```
        static void test(Object x) {
            System.out.println("Testing x of type " + x.getClass());
            System.out.println("x instanceof SubClass " + (x instanceof SubClass));
            System.out.println("x instanceof SuperClass " + (x instanceof
SuperClass));
        }
    }
```

运行结果：

```
    Testing x of type class SubClass
    x instanceof SubClass true
    x instanceof SuperClass false
    Testing x of type class SuperClass
    x instanceof SubClass true
    x instanceof SuperClass true
```

5.1.4成员变量的隐藏和方法的重写

有时从父类继承下来的成员不能满足子类的要求，虽然子类可引入新的成员数据或成员方法来代替，但如果能对继承成员进行调整以满足子类的要求就更为方便。根据需要对继承成员进行调整是继承机制的一个重要方面。

重新定义是对原有的继承成员赋予新的含义，也称为重写。例如，重新定义继承的数据成员的类型，重新实现继承的成员方法等。一般来说，重新实现继承的成员方法是最常见的重新定义。

当我们在子类中定义的成员变量和父类中的成员变量同名时，称子类重新定义了父类的该变量，也称子类的成员变量隐藏了父类的该成员变量。

子类重新定义父类的方法是指在子类定义一个与父类（或祖先类）的某个方法有完全相同接口的方法，这时称子类的这个方法重新定义了父类（或祖先类）相同接口的方法。所谓方法接口完全相同是指返回类型、方法名和方法参数列表完全相同，方法参数列表相同指参数的个数以及相应位置上的参数类型相同（参数名可以不同），我们也称子类重写了父类的该方法。因此，子类通过成员变量的隐藏和方法的重写可以把父类的状态和行为改变为自身的状态和行为。例如，

假定我们有下述父类：

```
    public class SuperClass {
        int super;
        public void method(int x, double y) {
            System.out.println("SuperClass -> method");
        }
    }
```

下面的子类 SubClassOne 定义的方法 method()和成员变量 super 就是对父类 SuperClass 中方法 method()和成员变量 super 的重新定义：

```
    public class SubClassOne extends SuperClass {
        String super;       //父类变量super被隐藏
        public void method(int z, double w) { //重定父类的method(int,double)方法
            System.out.println("SubClassOne -> method");
        }
    }
```

而下面述子类 SubClassTwo 所定义的方法 method()和成员变量 sub 就不是对父类 SuperClass 中方法 method()和成员变量 super 的重定义，因为它们的返回类型和参数列表不相同：

```
public class SubClassTwo extends SuperClass {
    int sub;  //因为变量名不变
    public int method(int x) { //因为参数不区配
        System.out.println("SubClassTwo -> method");
        return 1;
    }
}
```

这时 SubClassTwo 拥有两个 method()方法，一个是自己定义的返回类型为 int 的 method(int)方法，而另一个是从父类继承的返回类型为 void 的 method(int ,double)方法。

注意，Java 语言规定，在重定义时不能将静态成员方法改为非静态成员方法，也不允许将非静态成员方法改为静态成员方法。只要子类的方法与父类的方法接口相同就认为是重定义。

另外 Java 语言也不允许重定义一个使用 final 保留字修饰的祖先类方法。

例 5.5

```
class People {
    public double x;
    public void setX(double x) {
      this.x=x;
    }
    public double getDoubleX() {
      return x;
    }
}
class Employer extends People {
    String x;
    public String getX() {
      return x;
    }
}
public class Example5_4 {
    public static void main(String args[]) {
        Employer e=new Employer();
        e.x="abc"; //合法，子类对象的x是String型
        System.out.println("对象e的x的值是:"+e.getX());
        //e.x=98.98; //非法，因为子类对象的x已经不是double型
        e.setX(98.98); //子类对象调用继承的方法操作隐藏的double型变量x
        double m=e.getDoubleX();//子类对象调用继承的方法操作隐藏的double型变量x
        System.out.println("对象e隐藏的x的值是:"+m);
    }
}
```

运行结果：

```
对象e的x的值是:abc
对象e隐藏的x的值是:98.98
```

5.1.5 super 与 this

关键字 this 用来指向当前对象或类实例。例如：

```
public class MyDate {
   private int day, month, year;
   public void tomorrow() {
      this.day = this.day + 1;
  }
}
```

这里，this.day 指的是当前对象的 day 字段。或者在成员方法定义时，我们使用的形式参数与成员变量名称相同，这时我们需要用到 this。例如：

```
class Demothis{
    int a;          //成员变量
    public Demo(int a)
    {
        this.a = a;      //“＝”左边的a是对象的成员变量a，而右边的是形参a
    }
}
```

子类在隐藏了父类的成员变量或重写了父类的方法后，常常还希望能用到父类的成员变量，或者在重写的方法中使用父类中被重写的方法以简化代码的编写，这时就需要访问父类的被重写的成员变量或调用父类被重写的方法。

Java 中通过 super 来实现对父类成员的访问。前面我们已经知道，this 用来引用当前对象，与 this 类似，super 用来引用当前对象的父类。

super 的使用可以分为下面三种情况：

1)用来访问父类被隐藏的成员变量，如：super.variable

2)用来调用父类中被重写的方法，如：super.Method([paramlist])；

3)用来调用父类的构造方法，如：super([paramlist])；

注意，在构造方法中，super 必须是子类构造方法中的第一条语句

例 5.6 调用父类的构造方法

```
class Parent { //类A
    public int n;   //公共类型的成员变量
    public Parent( ){ }
    public Parent(int n){ this.n = n; }
    int method( ){  return n; }
}
public class Sub extends Parent { //类B
    public Sub(){
        super(10);
    }
    public static void main(String args[]){
        Parent  aInstance = new Sub( );
        int b=aInstance.method( );
        System.out.println("类A中的成员变量："+b);
    }
}
```

运行结果是：

```
类 A 中的成员变量：10
```

例 5.7 调用父类的构造方法和访问被重写的成员

```
class superClass {
    int y;
    superClass( ) {
        y=30;
        System.out.println("父类的方法中:y="+y);
    }
    void doPrint(){
        System.out.println("执行父类的doPrint()方法");
    }
}
class subClass extends superClass{
    int y;
    subClass( ){
        super();  //调用父类的构造方法
        y=50;
        System.out.println("在子类中:y="+y);
    }
    void doPrint( ){
        super.doPrint();  // 调用父类被重写的方法
        System.out.println("执行子类的doPrint()方法");
        System.out.println("super.y="+super.y+"  sub.y="+y);
    }
}
public class inviteSuper{
    public static void main(String args[]){
        subClass subSC=new subClass();
        subSC.doPrint( );
    }
}
```

运行结果:

```
父类的方法中:y=30
在子类中:y=50
执行父类的doPrint()方法
执行子类的doPrint()方法
super.y=30  sub.y=50
```

5.1.6 对象的上转型对象

假设类 A 是类 B 的父类，即

class B extends A {…… }

当我们用子类创建一个对象，并把这个对象的引用放到父类的对象中时，例如

A a;

A a=new B();

或者：

A a;

B b=new B();
a=b;

称这个父类对象 a 是子类对象 b 的上转型对象，上转型对象在 Java 编程中是常见的。

上转型对象的实例是子类负责创建的，但上转型对象会失去原对象的一些属性和功能，其特点归纳如下：

◇如果子类重写了父类的某个方法后，当上转型对象调用这个方法时一定是调用子类中这个重写的方法。因为程序在运行时知道，这个上转型对象的实体是子类创建的。

◇上转型对象不能操作子类新增的成员变量和子类新增的方法。

◇可以将对象的上转型对象再强制转换到一个子类对象，这时，该子类对象又具备了子类所给的所有属性和功能，也能操作子类新增的成员变量和子类新增的方法了。

例 5.8 上转型对象

```
class  Mammal{   //哺乳动物类
    private int n=40;
    void speak(String s) {
        System.out.println("哺乳动物类中的speak()方法:"+s);
    }
}
public class Monkey extends Mammal{   // 猴子类
    void computer(int aa,int bb) {
        int cc=aa*bb;
        System.out.println("猴子类中的computer()方法执行结果:"+cc);
    }
    void speak(String s) {
        System.out.println("执行猴子类中的speak()方法:"+s+"**");
    }
    public static void main(String args[]){
        Mammal mammal=new Monkey();  // mammal是Monkey类的对象的上转型对象.
        //虽然子类重写了父类的方法后，但上转型对象调用这个方法时一定是调用子类中这个重写的方法。
        mammal.speak("我喜欢这个游戏");
        //上转型对象不能访问子类新增的方法,下面的访问是错误
        //mammal.computer(10,10);
        //将上转型对象再强制转换到为其子类对象后,则具备子类对象的所有特点
        Monkey monkey=(Monkey)mammal; //把上转型对象强制转化为子类的对象.
        monkey.computer(10,10);
    }
}
```

运行结果：

执行猴子类中的speak()方法:我喜欢这个游戏**

猴子类中的 computer()方法执行结果:100

5.2 多态性

多态(Polymorphism)的意思就是用相同的名字来定义不同的方法。在 Java 中，普通类型的多态为重载，这就意味着可以用相同的名字来表示几个不同的方法，这些方法以参数的个数不同、参数的类型不同等方面来进行区分，以使得编译器能够进行识别。也可以这样讲，重载是同一个方法具有不同的版本，

每个版本之间在参数特征方面有差异。

例如：family()方法可以有三个版本，如下：

```
family() {  }
family(String ch) { address=ch; }
family(String ch,float n) { address=ch; pay=n; }
```

这些方法并存于程序中，编译时，编译器根据实参的类型和个数来区分从而调用那个方法。如果这些方法作为函数或过程同时出现在其它语言的程序中，如 C，那将导致灾难性的错误。

由方法重载实现的多态性称为静态多态性，也称编译时多态。在编译阶段，具体调用哪个被重载的方法，编译器会根据参数的不同来静态确定调用相应的方法，重载是 Java 实现多态性的方式之一。在第四章 4.5 节对方法重载作了详细介绍，这里再给一个由方法重载实现的静态多态性。

例 5.9 方法重载实现的静态多态性

```
class P {
    String str="Word";
    //构造方法重载
    P(){ }
    P(String a) {
        str=a;
    }
    //方法重载
    int add(int a,int b)
    {
        return a+b;
    }
    double add(double a,double b)
    {
        return a+b;
    }
    public String display(){
        return("Str="+ str);
    }
}
public class Example5_9{
    public static void main(String[] args) {
        P ko1=new P();
        P ko2=new P("Excel");
        System.out.println(ko1.display()+"add(1,2)="+ko1.add(1,2));
        System.out.println(ko2.display()+"add(2.3,32.)="+ko2.add(2.3,3.2));
    }
}
```

运行结果：

```
Str=Word    add(1,2)=3
Str=Excel   add(2.3,3.2)=5.5
```

除静态多态性外，在 Java 语言中，多态性还体现在方法重写实现的动态多态性，也称运行时多态。由于子类继承了父类所有的属性，所以子类对象可以作为父类对象使用。程序中凡是使用父类对象的地

方，都可以用子类对象来代替。一个对象可以通过引用子类的实例来调用子类的方法。如果子类重写了父类的方法，那么重写方法的调用原则如下：

◇ Java 运行时系统根据调用该方法的实例，来决定调用哪个方法。对子类的一个实例，如果子类重写了父类的方法，则运行时系统调用子类的方法；

◇ 如果子类继承了父类的方法（未重写），则运行时系统调用父类的方法。

另外，方法重写时应遵循的原则如下：

◇ 改写后的方法不能比被重写的方法有更严格的访问权限，比如，某方法在父类中是 public 的，那么在子类中是不可以更改为 protected 或者 private，也不可以缺省访问控制修饰符。

◇ 改写后的方法不能比被重写的方法产生更多的异常。

进行方法重写时必须遵从这两个原则，否则编译器会指出程序出错。

例 5.10 方法重写实现的动态多态性

```
class  Animal {
    void cry() {  }
}
class Dog extends Animal {
   void cry() {
      System.out.println("这是Dog的叫声：汪汪...汪汪");
   }
}
class Cat extends Animal {
   void cry() {
      System.out.println("这是Cat的叫声：喵喵...喵喵...");
   }
}
public class Example5_10 {
   public static void main(String args[]) {
      Animal animal=new Dog();   //animal是Dog的上转型对象
      animal.cry();
      animal=new Cat();       //animal是Dog的上转型对象
      animal.cry();
   }
}
```

运行结果：

```
这是Dog的叫声：汪汪...汪汪
这是 Cat 的叫声：喵喵...喵喵...
```

总之，在将方法调用绑定到方法体时都经过两个步骤，一个步骤是在编译时静态完成的，确定该方法调用的可访问且可应用的最精确方法，另一个步骤是在运行时动态完成的，根据对象引用的动态类型真正调用这个最精确方法可能被重定义的版本。因此，从某种意义上说，Java 语言的方法调用都是动态绑定的，但只有在满足以下条件时，运行时实际调用的方法版本才与编译时确定的方法版本不同：

1）被调用的方法是非静态方法，静态方法的调用不会产生运行时多态；

2）被调用的方法在祖先类定义，而且被后代类重写；

3）使用静态类型为祖先类的引用变量调用方法，但它的动态类型却是后代类。

知道 Java 里绑定的所有方法都通过后期绑定具有多态性以后，就可以相应地编写自己的代码，令其与基础类沟通。此时，所有的子类都保证能用相同的代码正常地工作。或者说，我们可以认为是“将一

条消息发给一个对象，让对象自行判断要做什么事情”。

例 5.11

```
class Shape {
    void draw() {}
    void erase() {}
}
class Circle extends Shape {
    void draw() {
        System.out.println("Circle.draw()");
    }
    void erase() {
        System.out.println("Circle.erase()");
    }
}
class Square extends Shape {
    void draw() {
        System.out.println("Square.draw()");
    }
    void erase() {
        System.out.println("Square.erase()");
    }
}
class Triangle extends Shape {
    void draw() {
        System.out.println("Triangle.draw()");
    }
    void erase() {
        System.out.println("Triangle.erase()");
    }
}
public class Example5_11{
    public static void main(String[] args) {
        Shape[] s = new Shape[9];
        // 用各种图形填充数组s
        for(int i = 0; i < s.length; i++)
        {
            //随机生成一个形状赋给Shape对象
            switch((int)(Math.random() * 3)) {
                default: // To quiet the compiler
                case 0: s[i]= new Circle();break;
                case 1: s[i]= new Square();break;
                case 2: s[i]= new Triangle();break;
            }
        }
```

```
        // 动态多态性
        for(int j = 0; j < s.length; j++)
            s[j].draw();
    }
}
```

运行结果：

```
Triangle.draw()
Circle.draw()
Circle.draw()
Circle.draw()
Triangle.draw()
Circle.draw()
Square.draw()
Triangle.draw()
Circle.draw()
```

上面的例子中，有一个基础类名为 Shape，用来作为所有形状的父类，另外还有大量子类：Circle（圆形），Square（方形），Triangle（三角形），这些对象的上转型对象可用类似下面这个语句简单地表现出来：

```
Shape s = new Circle();
```

在这里，我们创建了 Circle 对象，并将结果赋给一个 Shape 类。这表面看起来似乎属于错误操作（将一种类型分配给另一个），但实际是完全可行的——因为按照继承关系，Circle 属于 Shape 的一种。因此编译器认可上述语句，不会向我们提示一条出错消息。

当我们调用其中一个基础类方法时（该方法已在子类里覆盖）：

```
s.draw();
```

大家也许认为会调用 Shape 的 draw()，因为这毕竟是一个 Shape 对象。那么编译器怎样才能知道该做其他任何事情呢？但此时实际调用的是 Circle.draw()，因为动态绑定已经介入。

针对从 Shape 派生出来的所有东西，Shape 建立了一个通用接口——也就是说，所有（几何）形状都可以描绘和删除。子类重写了这些定义，为每种特殊类型的几何形状都提供了独一无二的行为。

main()包含了 Shape 的一个数组，其中的对象随机生成。我们知道自己拥有 Shape，但不知除此之外任何具体的情况（编译器同样不知）。然而，当我们为每个数组元素调用 draw()的时候，与各类型有关的正确行为会魔术般地发生，就象上面这个运行结果那样。

当然，由于几何形状是每次随机选择的，所以每次运行都可能有不同的结果。之所以要突出形状的随机选择，是为了让大家深刻体会这一点：编译器毋需获得任何特殊的信息也能发出正确的调用。对所有对象对draw()的调用都是通过动态绑定进行的。

5.3 最终类最终方法与抽象类抽象方法

继承机制允许对现有的类进行扩充，但有些时候，我们也许想把一个类的功能固定下来，不再允许定义它的子类对其进行扩充，Java 语言称这样的类为最终类(final class)，例如基本数据类型的包装类，如 Byte、Double 等，都是最终类。

要把一个类定义为最终类，只要在声明时用保留字 final 修饰即可，例如：

```
public final class MyFinalClass {
......
}
```

如果把最终类声明为某个类的超类，编译程序将报告错误。

我们也可把一个（普通）类的某些成员方法声明为最终方法，这使得该方法不可以被子类重写，例

如：

```
public class MyClass {
    public final void myFinalMethod() { ... ... }
......
}
```

最终类是类层次结构的叶子节点，它们不能再被继承和扩充。但 Java 语言还有一种特殊的类，它们只能被继承和扩充，而不能用于创建自己的对象实例，这种类称为抽象类(abstract class)。抽象类用于建模现实世界中一些没有具体对象的纯粹抽象概念，例如，“鸟”可以认为是一个纯粹抽象的概念，它本身没有任何具体的对象实例，任何具体的鸟儿都是由“鸟”经过特殊化形成的某个具体的种类的对象，也就是说，“鸟”是具体的鸟儿经过抽象形成的一个概念。

Java 语言在声明一个类时使用保留字 abstract 修饰则使得它成为抽象类，例如：

```
public abstract class MyAbstractClass {
    ... ...
}
```

使用抽象类创建对象实例，编译器会报告错误，例如下述语句不能通过编译：

```
MyAbstractClass obj = new MyAbstractClass(); // 编译错误!
```

Java 语言也可定义某些成员方法为抽象方法，例如：

```
public abstract class MyAbstractClass {
    public abstract void myAbstractMethod(); // 没有方法体!
    ... ...
}
```

抽象方法没有方法体，直接跟分号表示结束。注意，只能将抽象类的成员方法定义为抽象方法，不能将非抽象类的成员方法定义为抽象方法。从某种意义上，正是因为抽象类有一些成员方法是抽象方法，没有方法体，不能直接调用，所以抽象类才不能创建对象实例。

显然不能将一个类（方法）同时用 final 和 abstract 修饰，因为禁止类被派生，方法被重定义，而抽象类只能被派生才有可能创建对象实例，抽象方法只有被重定义才能给出其方法体。

抽象类不能创建对象实例，因此它通常作为某些类的超类。抽象类的子类应该重定义抽象类的抽象方法，给出它们的具体实现，这时这个子类方法不要再用 abstract 修饰。如果抽象类的某个抽象方法没有被它的子类重定义给出具体实现，则这个子类也是抽象类，在声明这个子类时必须使用 abstract 修饰。

下面，我们举一个比较完整的例子来说明抽象类的使用。假定我们要为某个公司编写雇员工资支付给程序。这个公司有各种类型的雇员，不同类型的雇员按不同的方式支付工资：经理(Manager)每月获得一份固定的工资；销售人员(Salesman)在基本工资的基础上每月还有销售提成；一般工人(Worker)则按它每月工作的天数计算工资。在这里有各种各样的雇员，按照单选原则，我们应该设计一个类——雇员(Employee)描述所有雇员的共同特性，例如姓名(name)等。这个类还应提供一个计算工资的抽象方法 computeSalary()使得我们可通过这个类计算所有雇员的工资，这个方法是抽象方法，因为无法为一个没有明确类型的雇员计算工资。经理、销售人员、一般工人对应的类都继承这个父类，并重定义计算工资的方法 computeSalary()，给出它的具体实现。下面的程序例 5.12 定义了父类 Employee、子类 Manager、Salesman 和 Worker。为简单起见，这些类我们都采用了最简单的设计：

例 5.12

```
//雇员抽象类
abstract class Employee {
    private String name;
    public Employee(String name) {
        this.name = name;
```

```
        }
        public String getName() {
            return name;
        }
        // 计算雇员月工资的抽象方法
        public abstract double computeSalary();
    }
    //经理类
    class Manager extends Employee {
        private double monthSalary; // 月工资额
        public Manager(String name, double monthSalary) {
            super(name); // 调用父类Employee的构造方法
            this.monthSalary = monthSalary;
        }
        //重定义父类的抽象方法computeSalary，下面几个类的方法类似。
        //注意：方法重定义只要求方法的返回类型与形式参数列表完全一致。
        public double computeSalary() {
            return monthSalary;
        }
    }
    //销售人员类
    class Salesman extends Employee {
        private double baseSalary; // 基本工资额
        private double commision; // 每件产品的提成额
        private int quantity; // 销售的产品数量
        public Salesman(String name, double baseSalary, double commision, int
quantity) {
            super(name); // 调用父类Employee的构造方法
            this.baseSalary = baseSalary;
            this.commision = commision;
            this.quantity = quantity;
        }
        public double computeSalary() {
            return baseSalary + commision * quantity;
        }
    }
    //工人类
    class Worker extends Employee {
        private double dailySalary; // 每天工资额
        private int days; // 每月工作的天数
        public Worker(String name, double dailySalary, int days) {
            super(name); // 调用父类Employee的构造方法
            this.dailySalary = dailySalary;
            this.days = days;
```

```
    }
    public double computeSalary() {
        return dailySalary * days;
    }
}
public class Example5_12 {
    public static void main(String[] args) {
        //以共同的父类声明记录雇员的数组
        Employee[] data = new Employee[4];
        //创建一些雇员对象，这时雇员数组是一个多态数据结构，可以存放各种类型的雇员
        data[0] = new Manager("Manager", 10000);
        data[1] = new Salesman("Salesman", 3000, 200, 12);
        data[2] = new Worker("Worker Zhang", 200, 25);
        data[3] = new Worker("Worker Li", 250, 26);
        displaySalary(data);
    }
    //显示所有雇员的工资额，不同种类的雇员都放在数组data中，因为Employee抽象了所有雇员的共同特性，
    //因此这里只要以Employee作为参数即可，而无需关心具体种类的雇员。
    public static void displaySalary(Employee[] data) {
        for (int i = 0; i < data.length; i++) {
        //下面的调用data[i].computeSalary()将根据data[i]所引用的对象实例调用相应的computeSalary()方法来计算该雇员的月工资额
            System.out.println("雇员 " + data[i].getName() + " 的月工资是： "+data[i].computeSalary());
        }
    }
}
```

运行结果：

```
雇员 Manager 的月工资是： 10000.0
雇员 Salesman 的月工资是： 5400.0
雇员 Worker Zhang 的月工资是： 5000.0
雇员 Worker Li 的月工资是： 6500.0
```

例 5.12 中，雇员数组 data 是一个典型的多态数据结构，里面可存放各种类型的雇员（实际上是雇员对象实例的引用），在 displaySalary()中的调用 data[i].computeSalary()会根据 data[i]的动态类型调用相应雇员的 computeSalary()方法计算其月工资额。这种动态多态性使得程序的扩充十分方便，假设我们要将一般雇员再细分为计时雇员(DailyWorker)和计件雇员(PieceWorker)两类，计时雇员按天计算工资额，而计件工资按其生产的产品件数计算工资额，那么我们可取消类 Worker，重新派生两个类 DailyWorker 和 PieceWorker，重定义其中的 computeSalary()方法，再重新编译这两个类即可。

5.4 接口

类是构成 Java 程序的一种基本单位，一个 Java 程序至少含有一个类。接口（interface）是构成 Java 程序的另一基本单位。我们可从两方面理解 Java 语言接口的作用：一方面，接口也是构成 Java 程序的一个基本单位；另一方面，接口也定义了 Java 程序的一种类型。

作为构成 Java 程序的基本单位之一，从本质上讲，接口是一种特殊的抽象类，这种抽象类中只包

含常量和方法的定义，而没有变量和方法的实现，因此，我们可以把接口认为就是方法定义和常量值的集合。通过接口使得处于不同层次，甚至互不相关的类可以具有相同的行为。在接口中，只能定义公有的静态常量数据成员，不能定义其他形式的数据成员，只能定义公有的抽象方法，不能定义其他形式的成员方法。因此接口表达了设计级的内容，表示通过该接口进行访问的使用者与实现该接口的实现者之间的某种约束，没有任何实现，也不能用于创建对象实例。

类可以实现接口，这是类与接口之间最基本的关系，这时这个类称为该接口的实现者，而该接口称为这个类的超接口(super interface)。从某种意义上说，接口是观察实现该接口的类的一种角度。例如，电视机、空调等可以从它们都是可遥控的这个角度去观察，因此可得到“可遥控的”这个接口；又例如，人、老虎等可以从它们都是可活动的这个角度去观察，因此可得到“可活动的”这个接口等。类实现一个接口就是承诺可以从某个角度对它进行观察。

一个类可以实现多个接口，因为现实世界的实体总是可以从多个角度去观察，例如电视机这个实体既可以从可遥控的角度观察，也可从可欣赏的角度观察。一个接口也可为多个类所实现，因为人们可通过同一个角度去观察不同的实体。

接口定义了 Java 程序的一种类型，可声明某接口的引用变量。虽然不能通过该变量创建对象实例，但可通过该变量引用实现该接口的类的对象实例，从而调用该接口所定义的方法来观察该对象实例，而该对象实例则通过实现接口的方法支持这种观察。在这个意义下，实现某接口的类所对应的类型是该接口所定义的类型的子类型，即任何需要该接口类型的变量的地方可以用实现它的类的变量去代替。这种代替是安全可行的，因为类实现一个接口就承诺支持对它的这种观察。

实现接口的类类型是该接口的子类型，这是 Java 语言中子类型关系的一种重要形式。同时，Java 语言的接口也可被继承，从而接口之间也存在子类型关系：接口 A 继承接口 B，则使得 A 类型是 B 类型的子类型。

与类之间的继承相比，Java 语言的接口继承允许多继承，而不仅仅是单继承，这是因为接口不涉及实现，概念上较简单，容易解决名字冲突问题，且不会带来重复继承的问题。 Java 语言的接口不是单根的，不存在一个接口是所有接口的祖先接口。

5.4.1 接口声明

接口通过使用关键自 interface 来声明，完整的接口定义格式如下：

```
[public] interface interfaceName [extends  listOfSuperInterface]{
    [常量定义]
    [方法定义]
}
```

其中 public 修饰符指明任意类均可以使用这个接口，缺省情况下，只有与该接口定义在同一个包中的类才可以访问这个接口。extends 子句与类声明中的 extends 子句基本相同，不同的是一个接口可以有多个父接口，用逗号隔开，而一个类只能有一个父类。子接口继承父接口中所有的常量和方法。

接口体中包含常量定义和方法定义两部分。接口的常量定义只能是公有的静态常量数据，因此数据成员的声明不能使用除 public、static、final 以外的修饰符，接口的方法定义只能是公有的抽象方法，因此成员方法的声明不能使用除 public、abstract 以外的修饰符。例如：

```
interface Summaryable {
    final int MAX=50;       // MAX具有public、static、final属性
    void  printone(float x);   //printone(float x) 具有public、abstract属性
    float sum(float x ,float y);
}
```

接口的数据成员缺省认为是公有静态常量，成员方法缺省认为是公有抽象方法，在声明接口的成员时也可以不使用象 public、static、final、abstract 之类的修饰符，比如上面例子中，MAX 为公有静态常量，printone()和 sum()方法为公有抽象方法。

接口中的公有静态常量都必须使用常量表达式进行初始化，否则会出现编译错误。

5.4.2 接口与类

只有接口的 Java 程序是不可运行的。接口表达的是设计级的内容，它没有实现体，接口的意义在于它可以被类实现。类实现接口，接口提供一种观察类的角度或方式，这是类和接口之间的基本关系。

一个类通过使用关键字 implements 声明自己使用（或实现）一个或多个接口。如果实现多个接口，用逗号隔开接口名。例如：

```
class Calculate implements Summary,Substractable{
    ……
}
```

类 Calculate 使用了 Summary 和 Substractable 接口。

如果一个类使用了某个接口，那么这个类必须实现该接口的所有方法，即为这些方法提供方法体。需要注意的如下：

◇ 在类中实现接口的方法时，方法的名字，返回类型，参数个数及类型必须与接口中的完全一致。

◇ 接口中的方法被默认是 public，所以类在实现接口方法时,一定要用 public 来修饰。

例 5.13 使用多重接口的例子

```
interface I1 {
    abstract void test(int i);
}
interface I2 {
    abstract void test(String s);
}
class MultInterfaces implements I1, I2 {
    public void test(int i) {
        System.out.println("In MultInterfaces.I1.test "+i);
    }
    public void test(String s) {
        System.out.println("In MultInterfaces.I2.test "+s);
    }
}
public class Example5_13{
    public static void main(String[] a) {
        MultInterfaces  t = new MultInterfaces ();
        t.test(42);
        t.test("Hello");
    }
}
```

运行结果：

```
In MultInterfaces.I1.test 42
In MultInterfaces.I2.test Hello
```

类实现接口使得类所对应的类型成为接口所对应的类型的子类型，因此，虽然不能为接口创建对象实例，但可以声明接口的引用变量，而且可以使这种引用变量引用实现该接口的类的对象实例，进而调用接口方法在这个对象实例的实现版本，这是接口的最基本用法，例 5.14 中将 Television 类型对象上转型为 Controllable，例 5.15 中 LininCorp、TebuCorp 的对象上转型为 Advertisement。

例 5.14

```
interface Controllable {
    int OFF = 0;
    int ON = 1;
    void setPower(int onOrOff);
    boolean isPowerOn();
}
class Television implements Controllable {
    // OFF是接口Controllable的公有静态成员，因此可以通过接口名访问
    private int power = Controllable.OFF;
    private String tradeMark = ""; // 类Television自己声明的属性
    public Television(String mark) {
        tradeMark = mark;
    }
    // 类Television自己定义的方法
    public String getTradeMark() {
        return tradeMark;
    }
    //实现接口Controllable声明的方法，必须声明为公有方法
    public void setPower(int onOrOff) {
        if (onOrOff == Controllable.ON) {
            System.out.println("开启电视电源...");
            power = Controllable.ON;
        } else if (onOrOff == Controllable.OFF) {
            System.out.println("关闭电视电源...");
            power = Controllable.OFF;
        }
    }
    //实现接口Controllable声明的方法，必须声明为公有方法
    public boolean isPowerOn() {
        return (power == Controllable.ON);
    }
}
public class Example5_14 {
    public static void main(String[] args) {
        Television obj = new Television("长虹");
        /* 类Television实现接口Controllable，使得类型Television是Controllable的子类型，因此，对象obj可以隐式类型转换为Controllable类型*/
        Controllable ref = obj;
        ref.setPower(Controllable.ON);
        /* 不能使用ref直接直接访问getTradeMark()方法，因为该方法不是接口Controllable的成员，但可以将变量ref强制转换为Television类型，然后再调用getTradeMark()方法。这种强制转换是可行的，因为ref引用的对象实例本来就是类Television的实例*/
        if (ref.isPowerOn())
```

```
            System.out.println(((Television)ref).getTradeMark()+"牌电视的电源已
经开启！");
        else
            System.out.println(((Television)ref).getTradeMark()+"牌电视的电源已
经关闭！");
    }
}
```

运行结果：

```
开启电视电源...
长虹牌电视的电源已经开启！
```

例 5.15

```
interface Advertisement {
    public void showAdvertisement();
    public String getCorpName();
}
class LininCorp implements Advertisement { // LininCorp实现Avertisement接口
    public void showAdvertisement(){
        System.out.println("**************");
        System.out.println("一切皆有可能");
        System.out.println("**************");
    }
    public String getCorpName() {
        return "李宁" ;
    }
}
class TebuCorp implements Advertisement { //TebuCorp实现Avertisement接口
    public void showAdvertisement(){
        System.out.println("----------------------");
        System.out.println("飞一般的感觉");
        System.out.println("----------------------");
    }
    public String getCorpName() {
        return "特步" ;
    }
}
class AdvertisementBoard {
    public void show(Advertisement adver) {
        System.out.println("广告牌显示"+adver.getCorpName()+"公司的广告词：");
        adver.showAdvertisement();
    }
}
public class Example5_15 {
    public static void main(String args[]) {
        AdvertisementBoard board = new AdvertisementBoard();
```

```
            board.show(new TebuCorp());
            board.show(new LininCorp());
        }
    }
```

运行结果：

```
广告牌显示特步公司的广告词：
----------------------
飞一般的感觉
----------------------
广告牌显示李宁公司的广告词：
**************
一切皆有可能
**************
```

可以把实现某一接口的类创建的对象的引用赋给该接口声明的接口变量中，那么该接口变量就可以调用被类重写的接口方法，我们称为接口回调。实际上，当接口变量调用被类重写的接口方法时，就是通知相应的对象调用这个方法。

例 5.16 接口回调

```
interface  ShowMessage {
    void ShowMark(String s);
}
class TV implements ShowMessage {
    public void ShowMark(String s) {
        System.out.println(s);
    }
}
class PC implements ShowMessage {
    public void ShowMark(String s) {
        System.out.println(s);
    }
}
public class Example5_16 {
    public static void main(String args[]) {
        ShowMessage sm;         //声明接口变量
        sm=new TV();           //接口变量中存放对象的引用
        sm.ShowMark("长虹电视机");//接口回调。
        sm=new PC();           //接口变量中存放对象的引用
        sm.ShowMark("联想PC机"); //接口回调
    }
}
```

运行结果：

```
长虹电视机
联想 PC 机
```

5.4.3 使用接口的优点

接口的优点主要体现在下面几个方面：

（1）通过接口可以实现不相关类的相同行为，而不需要考虑这些类之间的层次关系。

（2）通过接口可以指明多个类需要实现的方法。

（3）通过接口可以了解对象的交互界面，而不需了解对象所对应的类。

（4）接口把方法的定义和类的层次区分开来，通过它可以在运行时动态地定位所调用的方法。同时接口中可以实现“多重继承”，且一个类可以实现多个接口。正是这些机制使得接口提供了比多重继承（如 C++等语言）更简单、更灵活、而且更强劲的功能。

5.5 包

由于 Java 编译器为每个类生成一个字节码文件，且文件名与类名相同，因此同名的类有可能发生冲突。为了解决这一问题，Java 提供包来管理类名空间。包是多个类或接口的集合，这些类和接口可分别定义在不同的.java 文件中，每个 Java 文件称为一个编译单元。包实际上提供了一种命名机制和可见性限制机制。

Java 语言引入包以更层次化地组织一个大型的 Java 程序。一个 Java 程序可以定义多个包，包使得不同的类或接口可处于不同的层次。这时类或接口的名字有简单名和全名之分，所谓全名就是给出这个类或接口所在的包层次的名称，简单名就是类或接口的名字。例如根类 Object 的全名是 java.lang.Object，它处在 java.lang 包中，其中 lang 是包 java 的子包。这正如文件的全名和简单名，在 Windows 操作系统中，一个文件的全名是指带目录路径的名称。文件全名的目录路径中用斜杠分割目录名，而 Java 语言的类或接口的全名中使用句点分割包。

总之，包使得大型 Java 程序可以具有清晰的层次化结构。包、子包、类或接口构成 Java 程序的逻辑组织结构，而目录、子目录、文件是 Java 程序在大多数 Java 运行平台下的物理组织结构。包也使得类或接口的名字层次化，减少了类或接口的名字冲突，使得 Java 程序的重用更为方便。

5.5.1 package 语句

package 语句作为 Java 源文件的第一条语句，指明该文件中定义的类所在的包。（若缺省该语句，则默认为无名包，一般保存到当前目录）。在开发小的或者临时的应用程序或者刚刚开始开发时，用无名包是非常方便的。

声明包的格式为：

package pkg1[．pkg2[．pkg3…]]；

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

package java.awt.image;　//属于该包的类文件存储在 java/awt/image 目录下

package myPackage;　　//属于该包的类文件存储在 myPackage 目录下

以上包的声明中，当包名是一个单一的标识符时就定义了一个包，当它是由句点分割的几个标识符就定义了具有子包关系的多个包，每个句点后面的包是句点前面包的子包。

包层次的根目录路径是由环境变量 classpath 来确定的。

Java 的基础类库其实就是 JDK 安装目录下面 jre\lib\rt.jar 这个压缩文件。基础类库里面的类非常多。但是真正对于我们来说最核心的只有几个，例如 java.lang.*; java.io.*; java.util.*; java.sql.*;等。

例如下述 Java 程序源文件 Example5_17.java 的第一条语句声明该文件属于包 p1，

该声明也定义了包 packageOne：

例 5.17 一个简单的包

```
package p1;
public class Example5_17{
    public static void main(String[] args) {
        System.out.println("类Example5_17属于包p1!");
    }
}
```

注意，包声明语句必须放在源文件除注释和空白之外的第一条语句，否则会出现编译错误。

声明所属的包时在保留字 package 后也可使用句点分割的几个标识符，例如下述文件 Example5_18.java 声明该文件属于包 p2.subPackage：

例 5.18 含有子包的包

```
package p2.subPackage;
public class Example5_18 {
    public static void main(String[] args) {
        System.out.println("类Example5_18属于包p2.subPackage!");
    }
}
```

注意，在 Java 语言中，不能认为属于某个包的子包的类或接口也属于该父包。例如对于 Example5_18.java 文件，我们不能说其中定义的类 Example5_18 属于包 p2，只能说这个类属于包 subPackage，或更完整地说属于包 p2.subPackage。

5.5.2 import 语句

包使得 Java 程序的组织结构层次化，也使得类名层次化，即一个类有简单名和全名之分。当两个类处于不同的包时，一个类要访问另一个类就不能简单的使用简单名访问，例如上述例子中包 p1 的类 Example5_17 就不能简单地使用名字 Example5_18 访问包 p2.subPackage 的类 Example5_18。这时类 Example5_17 有两种途径访问类 Example5_18，一种途径是使用类 Example5_18 的全名，即名字 p2.subPackage. Example5_18，例如在类 Example5_17 的 main()方法中可使用下述语句调用类 Example5_18 的 main()方法：

p2.subPackage.Example5_18.main(args);

另一途径是在文件 Example5_17.java 中引入包 p2.subPackage 的类 Example5_18，然后就可使用简单名 Example5_18 访问这个类。引入的方式也有两种：一种方式是只引入 Example5_18 这个类，另一种方式则可引入包 p2.subPackage 的所有类或接口。前一种方式是在文件 ClassOne.java 中使用下述语句：

import p2.subPackage. Example5_18;

其中 import 是 Java 语言的保留字，后面跟要引入的类的全名。有了该语句之后，则可使用下述语句直接调用类 Example5_18 的 main()方法：

ClassTwo.main(args);

第二种方式则使用如下语句引入包的所有类或接口：

import p2.subPackage.*;

这里使用星号"*"表示引入包 p2.subPackage 的所有类或接口。

注意，前面已经提到 p2.subPackage 中的类或接口并不属于包 p2，所以使用下述语句不能引入 p2.subPackage 的类或接口：

import p2.*;

因此，为了能使用 Java 中已提供的类，我们需要用 import 语句来引入所需要的类。其格式为

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

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

```
import   java.util.Date;          //引入 java.util.Date 类
import   java.util.*;             //引入 java.util 包中的所有类
```

5.5.3 编译和运行包

使用下述命令编译例 5.1 中的源文件：

> javac Example5_17.java

访命令在当前目录生成 ClassOne.class 文件，这时如果试图使用下面的命令运行此程序将出错：

> java Example5_17

这是类 Example5_17 属于该包 p1，在操作系统的命令行中必须使用全名访问这个类。上述类 Example5_17 的全名为 p1.Example5_17，但若使用下述命令运行此程序也将出错：

> java p1.ClassOne

这是因为上述命令在装载类时会在目录 p1 下查找 ClassOne.class 文件，但该文件却处在当前目录，解决的办法是在当前目录下建立子目录 p1，并将 Example5_17.class 文件拷贝至该目录，在使用上述命令运行就可以得到正确的输出。

程序员也可使用-d 参数运行 javac，让它自动建立包对应的子目录，并将生成的字节码文件放在该目录下。例如，可使用下述命令编译文件 Example5_17.java：

> javac –d . Example5_17.java

其中-d 参数可给出一个目录名来指定字节码文件所存放的目录，当源文件属于某个包时，编译器会在-d参数后指定的目录再建立包所对应的子目录，并将该源文件编译得到的字节码文件放在该子目录下。上面命令-d 参数后的句点在 Windows 操作系统下代表当前目录，意味着相对于当前目录存放字节码文件，例如上述命令编译后的 Example5_17.class 文件就将放在当前目录中自动建立子目录 p1 下。

这时，可通过下述命令运行此程序：

> java p1.Example5_17

例 5.2 定义了两个包：p2 和 subPackage，并声明 subPackage 是 p2 的子包。如果 Example5_18.java 放在当前目录下，使用命令：

> javac –d . Example5_18.java

编译此文件，会自动在当前目录下创建目录 p2，在 p2 目录下再创建子目录 subPackage，然后编译得到的字节码文件 Example5_18.class 放在目录 subPackage 下，这时需要使用下述命令运行此程序：

> java p2.SubPackage.Example5_18

5.5.4 包级访问控制

我们已经学习了类成员三种访问控制级别：公有(public)、受保护(protected)和私有(private)，公有成员可以被任意类访问，受保护成员只可以由类本身或子类访问，私有成员则只能在类本身内部访问。

包引进了第四种访问控制级别：包级访问控制。当声明某个类成员时，若不使用保留字 public、protedted 或 private 中任何一个进行修饰则该成员就具有包级访问控制级别（因此包级访问控制级别也是 Java 语言的缺省访问控制方式），具有包级访问控制级别的成员只能由同一个包的类访问。

包级访问控制的开放程度介于私有和受保护之间。Java 语言规定受保护成员不仅能被后代类（不管该后代类是否在同一个包）访问，还能被同一个包的非后代类访问，因此受保护的访问控制级别的开放程度比包级访问控制级别要高。第四章 4.2.2 节的表 4-1 总结了 Java 类成员的访问权限控制方式。

下面使用例 5.19 说明这些访问控制方式。假定有两个包 packageOne 和 packageTwo，包 packageOne 有三个文件，分别定义类 SuperClass 、SamePackageSubClass 和 SamePackageClass，其中 SamePackageSubClass 是 SuperClass 的子类。程序包 packageTwo 有两个文件，分别定义类 OtherPackageClass 、OtherPackageSubClass， 其中 OtherPackageSubClass 也是 SuperClass 的子类，但类 SamePackageClass 和 OtherPackageClass 都不是类 SuperClass 的后代类。类 SuperClass 中定义有私有、默认、受保护和公有成员，这五个类都定义了主方法 main()访问这些成员，其中因为不能访问相应成员而会产生语法错误的语句在下面的程序中已经被注释掉。

例 5.19

```
//SuperClass.java
package packageOne;
public class SuperClass {
    public String publicMember = "类SuperClass的公有成员";
```

```
    protected String protectedMember = "类SuperClass受保护成员";
    String defaultMember = "类SuperClass的默认成员";
    private String privateMember = "类SuperClass的私有成员";
    public static void main(String args[]) {
        String canDisplay = "在类SuperClass中可以显示";
        SuperClass obj = new SuperClass();
        System.out.println(canDisplay + obj.publicMember);
        System.out.println(canDisplay + obj.protectedMember);
        System.out.println(canDisplay + obj.defaultMember);
        System.out.println(canDisplay + obj.privateMember);
    }
}
//SamePackageSubClass.java
package packageOne;
public class SamePackageSubClass extends SuperClass {
    public static void main(String args[]) {
        String canDisplay =  "与父类SuperClass同一包的子类中可以访问";
        String canNOTDisplay ="在父类SuperClass同一包的子类中不可以访问";
        SamePackageSubClass subObj = new SamePackageSubClass();
        //不能访问父类的私有成员
        // System.out.println(canNOTDisplay + subObj.privateMember);
        System.out.println(canDisplay + subObj.publicMember);
        System.out.println(canDisplay + subObj.protectedMember);
        System.out.println(canDisplay + subObj.defaultMember);
    }
}
// SamePackageClass.java
package packageOne;
public class SamePackageClass {
    public static void main(String args[]) {
        String canDisplay =  "在与类SuperClass同一包的非后代类中可以访问";
        String canNOTDisplay ="在与类SuperClass同一包的非后代类中不可以访问";
        SuperClass obj = new SuperClass();
        System.out.println(canDisplay + obj.publicMember);
        System.out.println(canDisplay + obj.protectedMember);
        System.out.println(canDisplay + obj.defaultMember);
        //不能访问其他类的私有成员
        //System.out.println(canNOTDisplay + obj.privateMember);
    }
}
// OtherPackageSubClass.java
package packageTwo;
public class OtherPackageSubClass extends packageOne.SuperClass {
    public static void main(String args[]) {
```

```
        String canDisplay ="与父类SuperClass在不同包的子类中可以访问";
        String canNOTDisplay ="在与类SuperClass不同包的子类中不可以访问";
        OtherPackageSubClass subObj = new OtherPackageSubClass();
        System.out.println(canDisplay + subObj.publicMember);
        System.out.println(canDisplay + subObj.protectedMember);
        //不能访问不在同一包的父类的默认成员
        // System.out.println(canNOTDisplay + subObj.defaultMember);
    }
}
// OtherPackageClass.java
package packageTwo;
public class OtherPackageClass {
    public static void main(String args[]) {
        String canDisplay ="与类SuperClass在不同包非后代类中可以访问";
        String canNOTDisplay ="与类SuperClass在不同包非后代类中不可以访问";
        packageOne.SuperClass obj = new packageOne.SuperClass();
        System.out.println(canDisplay + obj.publicMember);
        //非后代类不能访问不同包的类的保护成员
        // System.out.println(canNOTDisplay + obj.protectedMember);
        //非后代类不能访问不同包的类的缺省成员
        // System.out.println(canNOTDisplay + obj.defaultMember);
    }
}
```

简单地说，私有成员不能被类以外的地方访问，公有成员可以被任何类访问，不使用任何访问控制保留字修饰的默认成员可以被同一个程序包的任何类访问，而如果希望一个成员可以被当前程序包以外的类访问，但仅仅允许该成员所属类的后代类访问，则声明该成员为受保护成员。

5.6 嵌套类

嵌套类（nested class）是 Java 类的嵌套形式，一个类可以在另一个类中定义，例如：

```
Class OuterClass{
    ……
    Class NestedClass{
        ……
     }
}
```

嵌套类有静态成员类和非静态的成员类两种。静态成员类和非静态成员类的最大区别是，非静态成员类的对象实例必须在有定义它的外围类的对象实例存在的情况下才能创建，其对象实例保存有指向它的直接外围类对象实例的引用，静态成员类的对象实例创建则不受此限制。非静态成员类内部不能定义静态成员，而静态成员类则可以定义任意形式的成员。

使用嵌套类可以提高代码的可读性和可维护性，增强封装必，并增加类之间的逻辑联系。

5.6.1 内部类

非静态成员类又叫内部类（inner class）。包含内部类的类称为外部类。与一般的类相同，内部类具有自己的成员变量和成员方法。通过建立内部类的对象，可以存取其成员变量和调用其成员方法，也可以访问外部类的成员变量和方法。例如：

```
class OuterClass{
```

```
    ……
    class InnerClass{     //声明内部类
        ……
    }
}
```

要创建内部类对象，必须先创建外部类 OuterClass 的对象 outerObject，然后用下列语法形式创建内部类对象：

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

Java将内部类作为外部类的一个成员，就如同成员变量和成员方法一样。因此外部类与内部类的访问原则是：在外部类中，通过一个内部类的对象引用内部类中的成员；反之，在内部类中可以直接引用它的外部类的成员，包括静态成员、实例成员及私有成员。

例 5.20 内部类和外部类之间的访问

```
class GroupOne{
    private int count;    //外部类的私有成员变量
    //声明内部类
    class Student {
        String name;
        public Student(String n1) {
            name=n1;
            count++;    //存取其外部类的成员变量
        }
        public void Output(){
            System.out.println(this.name);
        }
    }
    //外部类的实例成员方法
    public void output(){
        Student s1=new Student("张三");    //建立内部类对象
        s1.Output();    //通过s1调用内部类的成员方法
        System.out.println("count="+this.count);
    }
}
public class Example5_20{
    public static void main(String args[]){
        GroupOne g2=new GroupOne();
        g2.output();
    }
}
```

运行结果:

```
张三
count=1
```

上面的例子中，类 GroupOne 中声明了成员变量 count、内部类 Student、实例方法 output 方法，在内部类 Student 中声明了构造方法和 output 方法，构造方法存取了外部类 GroupOne 的成员变量 count。这个例子演示嵌套的两个类之间的访问规则，即在外部类 GroupOne 中，通过一个内部类 Student 的对象

s1 可以引用内部类中的成员；反之，在内部类 Student 中可以直接引用它的外部类的成员，如 count。

本例的外部类 GroupOne 中有实例方法 output()，内部类 Student 中也有实例方法 output()，两者虽然同名，却表达不同含义。使用时，外部类 GroupOne 的对象调用 GroupOne 的 output，如 g2.output()，内部类 Student 的对象调用 Student 的 output，如 s1.output()。

内部类又有两种特殊形式，一种是在方法体内定义的内部类，称为局部内部类（local inner class）；另一种是在方法体内定义的没有类名的内部类，称为匿名内部类（anonymous inner class）。下面分别介绍这两种特殊形式。

5.6.2 局部内部类

局部内部类不是在外部类定义成员的地方定义，而是在某个方法体中定义的，例如：

```
class OutreClss{
    ……
    void myMethod(){
        class LocalInnerClass{
            ……
        }
    }
}
```

上面的局部内部类定义在方法 myMethod()中，因此不能在 myMethod()方法外部访问，所以更加隐蔽。

例 5.21 局部内部类

```
interface Destination {
    String readLabel();
}
public class Goods {
    //成员方法
    public Destination dest(String s) {
        //局部内部类
        class GDestination implements Destination {
            private String label;
            private GDestination(String whereTo) {
                label = whereTo;
                System.out.println(readLabel());
            }
            public String readLabel() { return label; }
        }
        return new GDestination(s);
     }
    public static void main(String[] args) {
        Goods g= new Goods ();
        Destination d = g.dest("送到目的地北京");
    }
}
```

运行结果：

```
送到目的地北京
```

5.6.3 匿名内部类

匿名内部类比局部内部类更隐蔽，它连名字都没有，例如：

```
public class OuterClass{
    public Abc abc(){
        return new Abc(){
                        private int x = 1;
                        public int add(){ return ++x;}
                       };
    }
}
```

匿名内部类可以用来继承一个类或实现一个接口，但不能同时实现这两种功能，上面的例子是用一个类实现了接口 Abc，但这个类是匿名的。

例 5.22 匿名内部类

```
interface Speak {
    void speakHello();
}
class Teacher {
    void f(Speak sp) {
        sp.speakHello();
    }
}
public class Example5_22 {
    public static void main(String args[]) {
        Speak speak=new Speak() {
            public void  speakHello() { //实现speakHello()方法
                System.out.println("大家好，祝工作顺利！");
            }
        }; //实现了speakHello()方法的匿名类
        speak.speakHello();  //调用匿名类中的方法
        Teacher teacher=new Teacher();
        teacher.f(new Speak() {
            public void  speakHello() { //实现speakHello()方法
                System.out.println("I am teacher,how are you");
            }
        }//匿名类作为参数
        );
    }
}
```

运行结果：

```
大家好，祝工作顺利！
I am teacher,how are you
```

5.6.4 静态嵌套类

静态嵌套类在外部类定义成员的地方定义，例如：

```
Class OuterClass{
```

```
        ……
        Static class StaicNestedClass{
            ……
        }
    }
```

静态嵌套类可以定义自己的静态成员，只能访问外部类的静态成员，外部类要访问静态嵌套类只能通过类名，如 OuterClass.StaticNestedClass。创建静态嵌套类对象的语法是：

```
    OuterClass.StaticNestedClass nestedObject = new
OuterClass.StaticNestedClass();
```

例 5.23 静态嵌套类

```
    class Statistics {
        public static class Result {
            int count = 0;
            int sum = 0;
            double average = 0;
            public void display() {
                System.out.println("count = " + count);
                System.out.println("sum = " + sum);
                System.out.println("average = " + average);
                // 在静态类的方法中不能使用Statistics.this来访问外围类Statistics的成员
            }
        }
        public static Result scan(int data[]) {
            Result r = new Result();
            for (int i = 0; i < data.length; i++) {
                r.count++;
                r.sum += data[i];
            }
            if (r.count > 0)
                r.average = (double)r.sum / r.count;
            return r;
        }
    }
    public class Example5_23 {
        public static void main(String[] args) {
            int[] data = {90, 77, 89, 83, 92};
            // 使用类Statistic的静态成员类Result，必须使用带修饰的类名，即
Statistics.Result
            Statistics.Result r = Statistics.scan(data);
            r.display();
            // 可以使用如下方式创建静态类的对象实例
            Statistics.Result obj = new Statistics.Result();
        }
    }
```

运行结果：

```
count = 5
sum = 431
average = 86.2
```

总之，如果嵌套类要访问外围类的非静态成员时，则要设计为非静态类（内部类），否则可设计为静态类。因为内部类要维护指向与它相关联的外围对象实例的引用，这种维护既要消耗空间，也要消耗时间，当嵌套类无需访问外围对象实例的非静态成员时，嵌套类与外围类的这种关联完全没有必要存在。

内部和匿名类是 Java 为我们提供的两个出色的工具。它们提供了更好的封装，结果就是使代码更容易理解和维护，使相关的类都能存在于同一个源代码文件中（这要归功于内部类），并能避免一个程序产生大量非常小的类（这要归功于匿名类）。

小结

本章主要阐述了面向对象的继承性和多态性在 Java 程序中的实现技术，其中包括子类的定义、子类构造方法的调用规则、类成员的隐藏和重载、抽象类、接口、包及嵌套类等一系列概念的应用。

在引入继承机制后，子类继承了父类除构造方法以外的所有成员，在子类的内部，不仅能够访问子类自己定义的所有成员，也能够访问父类中公有和受保护的成员，但不能访问父类中私有的成员。

Java 语言中，多态性包括方法重写实现的动态多态性和方法重载实现的静态多态性。重载是同一个方法具有不同的版本，每个版本之间在参数特征方面有差异。方法重写是指子类重新定义父类的方法，即在子类中定义一个与父类的某个方法有完全相同接口的方法，所谓方法接口完全相同是指返回类型、方法名和方法参数列表完全相同。

最终类是类层次结构的叶子节点，它们不能再被继承和扩充。抽象类只能被继承和扩充，而不能用于创建自己的对象实例(abstract class)。抽象类用于建模现实世界中一些没有具体对象的纯粹抽象概念

作为构成 Java 程序的基本单位之一，从本质上讲，接口是一种特殊的抽象类，这种抽象类中只包含常量和方法的定义，而没有变量和方法的实现，因此，我们可以把接口认为就是方法定义和常量值的集合。

Java 语言引入包以更层次化地组织一个大型的 Java 程序。使用嵌套类可以提高代码的可读性和可维护性，增强封装必，并增加类之间的逻辑联系。

习题

一、基本概念

1、阐述 Java 语言是如何实现继承性的？继承性给程序开发带来什么好处？

2、阐述 Java 程序如何实现多态性？多态性给程序开发带来什么好处？

3、举例说明什么是对象的上转型对象？

4、什么叫接口？接口有何用途？如何实现接口？

5、举例说明什么是接口回调？

6、什么叫抽象类？什么叫抽象方法？能够创建一个抽象类的实例吗？

7、什么叫包？包有何用途？如何创建包？如何将一个类放入给定包中？

8、什么叫嵌套类，有哪几种，各有什么特点？

二、编程实践

1、试定义一个商品类，然后在此基础上再定义一个食品子类，一个服装子类。假设任何商品都应该有商品编号、商品名称、出场日期、产品厂家名称等信息，除此之外，食品应该有保质期、主要成分等信息，服装品应该有型号、面料托信息。

2、试为教师工作证和学生证设计一个类层次结构，尽可能地保证代码的重用率。假设工作证包括编号、姓名、性别、出生年月、职务和签发工作证日期；学生证包括编号、姓名、性别、出生年月、系别、入校日期及每学年的注册信息。

4、试为普通矩阵、三角矩阵、对角矩阵和稀疏矩阵设计一个类层次结构，其中三角矩阵、对角矩阵和

稀疏矩阵应该采用相应的压缩形式存储。在这个题目中应该如何对各种类进行抽象？如何利用多态性？
5、设计一个描述图形的类，并定义图形的几个子类，描述图形的类中要包括成员变量、构造方法和成员方法，定义的子类要重写父类的成员方法；并构造各种子类的实例对象来调用父类的成员方法；并测试向上转型对象的使用，根据运行结果进行分析总结。
6、编写程序以验证接口中的数据均会被自动设为 static 和 final。

第 6 章 异常

程序可能按我们的意愿终止（例如使用 return 语句），也可能因为程序中发生了错误而终止。例如程序执行时遇到除数为 0 或数组下标越界、用户输入错误、需要的文件不存在、文件打不开、内存不足等，这时将产生系统中断，导致正在执行的程序提前终止，这些情况我们称为发生了异常（Exception）。为处理这些情况，常用的典型方法是让被调用方法返回某一个特别的值，而外层的调用程序则检查这个错误标志，从而确定是否产生了某一类型的错误；另一种典型方法是当错误发生时跳出当前的方法体，控制转向某个专门的错误处理程序，从而中断了正常的控制流。不过，这两种方法都是权宜之计，不能形成强有力的结构化异常处理模式。

异常处理（exception handling）机制是用于管理程序运行期间错误的一种结构化方法，所谓结构化是指程序的控制不会由于异常的产生而随意跳转。异常处理机制将程序中的正常处理代码与异常处理代码显式区别开来，提高了程序的可读性。在专门提供了异常处理机制的程序设计语言中，程序员能够取得对于异常情况的控制。异常处理使得某些确定的信息可以在处于不同控制层次的函数之间传递，同时提供了一种不同于通常函数调用返回的控制转移方法。

异常（exception）不是错误（error），它可能是某些很少出现的特殊事件，检测到这些事件发生的方法体由于缺乏足够的信息不能对其进行最适当的处理，只能报告给其他程序对此事件进行处理。我们强调检测异常事件发生的方法不能进行最适当处理有两方面的含义：一是若该方法如果能够进行最适当的处理则可认为是正常的情况；二是因为不能进行最适当处理才要将异常的检测与处理分开。因此，一个好的应用程序，除了应具备用户要求的功能外，还应具备能预见程序执行过程中可能产生的各种异常的能力，并把处理异常的功能包括在用户程序中。也就是说，我们设计程序时，要充分考虑到各种意外情况，不仅要保证应用程序的正确性，而且还应该具有较强的容错能力。这种对异常情况给予恰当处理的技术就是异常处理。在软件开发过程中，很多情况都将导致异常的产生，例如：

（1）打开的文件不存在；

（2）网络连接中断；

（3）操作数超出预定范围；

（4）正在装载的类文件丢失；

（5）访问的数据库打不开；

Java 语言的特色之一是面向对象的异常处理机制，通过该机制，可以减少编程人员的工作量，增加程序的灵活性，增强程序的可读性和可靠性。

6.1 异常类

6.1.1 Java 异常类的类层次

Java 语言的异常以类的层次结构组织，类 Throwable 是所有异常类的祖先类，它是根类 Object 的子类。类 Throwable 有两个子类 Exception 和 Error。类 Exception 及其后代类标识上述意义上的“异常”，即很少发生的一些特殊事件，而类 Error 及其后代类标识一些导致程序完全无法执行下去的“错误”，例如耗尽内存资源，JVM 内部发生错误等，这些错误当然也是一些很少会发生的事件，但与异常不同，被认为是 Java 程序完全无法恢复和处理的事件。Java 语言定义了类 Exception 的一个子类 RuntimeException，用于标识 Java 程序在运行时可能会发生的一些常见的异常，例如被 0 除、数组下标越界等。Java 语言的异常类的层次结构组织如图 6_1 所示。

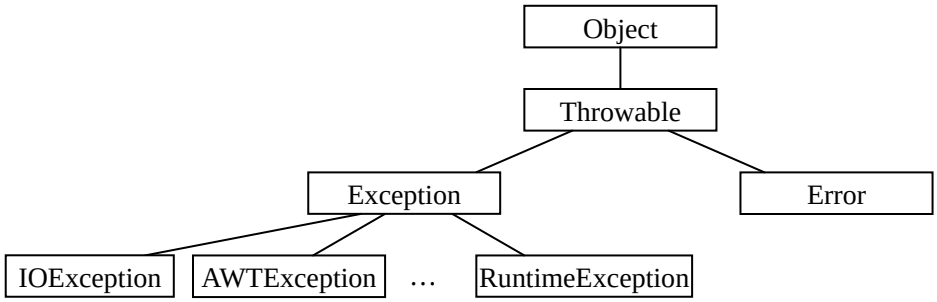

图 6_1 Java 异常类的层次结构

6.1.2 运行时异常与非运行时异常

可以将异常分为运行时异常和非运行时异常。对于非运行时异常，程序中必须要作出处理，或者捕获，或者声明抛弃。而对于运行时异常则可以不做处理。

1. 常见的运行时异常如下：

（1） 类型转换异常 ClassCastException，如：

```
String strName=new string(“123”);
int nNumber=(int)strName;       //试图把字符串强制转换为整数
```

（2）数组超界异常 ArrayIndexOutBoundsException，如：

```
int[] b=new int[10];
b[10]=1000;
```

（3）指定数组维数为负值异常 NegativeArraySizeException，如：

```
b[-1]=1001;
```

（4）算术异常 ArithmeticException，如：

```
int b=0;
a=500/b;
```

（5）Java 系统内部异常 InternalException，比如 JVM 抛出的异常。

（6）类型不符合异常 IncompatibleTypeException，如：

```
int n=12345;
String s=(String)n;
```

（7）内存溢出异常 OutOfMemeoryException

（8）没有找到类定义异常 NoClassDefFoundException

```
aClass aa=new aClass(); //但 aClass 类未定义。
```

（9）空指针异常 NullPointerException

```
int b[ ];
b[0]=99;     //没有实例化，就访问，将产生空指针。
```

2. 常见的非运行时异常如下：

（1）ClassNotFoundException：找不到类或接口所产生的异常

（2）CloneNotSupportedException：使用对象的 clone 方法但无法执行 Cloneable 所产生的异常。

（3）IllegalAccessException：类定义不明确所产生的异常。例如：类不为 public ，或是包含一个类定义在另一个类库内。

（4）IOException：在一般情况下不能完成 I/O 操作所产生的异常。

（5）EOFException：打开文件没有数据可以读取所产生的异常

（6）FileNotFoundException：在文件系统中，找不到文件名称或路径所产生

（7）InterruptedIOException：目前线程等待执行，另一线程中断目前线程 I/O 运行所产生的异常

在 Sun 公司提供的各种 API 包中，如 java.io,java.net,java.awt 等，都提供不同情况下可能产生的异常。

由于异常的种类非常多，需要实际运用中逐渐掌握。

6.1.3 创建用户异常类

如果 Java 提供的系统异常类型不能满足程序设计的需求，我们可以设计自己的异常类型。Java 语言允许程序员自定义异常类，以标识在特定的应用程序环境下可能发生的异常。从 Java 异常类的结构层次可以看出，Java 异常的公共父类为 Throwable。因此，用户定义的异常类型必须是 Throwable 的直接或间接子类。Java 推荐用户的异常类型以 Exception 为直接父类。创建用户异常的方法如下：

```
class UserException extends Exception{
    UserException(){
        super();
        …… //其它语句
    }
}
```

下面的代码用于创建自定义异常——缺钱异常 OutOfMoneyException：

```
class OutOfMoneyException extends Exception {
    public OutOfMoney() {
        super("Your account have not enough money!");
    }
    public OutOfMoney(String msg) {
        super(msg);
    }
}
```

6.2 异常的捕获与处理

6.2.1 Java 异常处理机制

异常处理机制包括检测、报告和处理。Java 语言提供两种处理异常的机制：

（1）捕获异常并处理

在 Java 程序运行过程中系统得到一个异常对象时，它将会沿着方法的调用栈逐层回溯，寻找处理这一异常的代码。找到能够处理这种类型异常的方法后，运行时系统把当前异常对象交给这个方法进行处理，这一过程称为捕获(catch)异常。这是一种积极的异常处理机制。如果 Java 运行时系统找不到可以捕获异常的方法，则运行时系统将终止，相应的 Java 程序也将退出。

（2）声明抛出异常

当 Java 程序运行时系统得到一个异常对象时，如果一个方法并不知道如何处理所出现的异常，则可在方法声明时，声明抛弃（throws）异常。Java 语言只支持异常的逐层报告，不支持异常的越级报告，也不支持在处理异常之后重新唤醒被异常中断执行的方法。

6.2.2 try-catch-finally

Java 语言将可能发生异常的语句放在由保留字 try 引导的块（try 语句块）中，其后使用保留字 catch 引导的块（catch 语句块）捕获这些语句所抛出的异常并进行处理，其基本形式如下：

```
try {
    ... // 可能抛出异常的语句;
} catch (异常类型 1 异常对象 1) {
    ... // 对该类型的异常进行处理的语句;
} catch (异常类型 2 异常对象 2) {
    ... // 对该类型的异常进行处理的语句;
}
... // 可以有多个 catch 块
```

```
catch (异常类型 n  异常对象 n) {
    ... // 对该类型的异常进行处理的语句;
}finally {
    ... // 执行异常处理后收尾工作的语句
}
```

1、try

捕获异常的第一步是用 try{…}选定捕获异常的范围，由 try 所限定的代码块中的语句在执行过程中可能会生成异常对象并抛出。

2、catch

一个 try 语句块可有一个或多个 catch 语句块，当 try 语句块抛出某种类型的异常时，程序将根据 catch 语句块书写的顺序逐个进行匹配，当发现某个 catch 后声明的异常类是 try 语句块抛出的异常类的祖先类时，就表示找到了匹配的 catch 语句块，程序将执行该语句块中的语句。

catch 块中的代码用来对异常对象进行处理，与访问其它对象一样，可以访问一个异常对象的变量或调用它的方法。getMessage( )是类 Throwable 所提供的方法，用来得到有关异常事件的信息。类 Throwable 还提供了方法 printStackTrace( )用来跟踪异常事件发生时执行堆栈的内容。例如：

```
try{
    ......
}catch( FileNotFoundException e ){
    System.out.println( e );
    System.out.println( "message: "+e.getMessage() );
    e.printStackTrace( System.out );
}catch( IOException e ){
    System.out.println( e );
}
```

3、catch 语句的顺序

捕获异常的顺序和 catch 语句的顺序有关，当捕获到一个异常时，剩下的 catch 语句就不再进行匹配。因此，在安排 catch 语句的顺序时，首先应该捕获最特殊的异常，然后再逐渐一般化。也就是一般先安排子类，再安排父类。例如上面的程序如果安排成如下的形式：

```
try{
    ......
}catch(IOException e ){
    System.out.println( e );
    System.out.println( "message: "+e.getMessage() );
    e.printStackTrace( System.out );
}catch(FileNotFoundException e ){
        System.out.println( e );
}
```

由于第一个 catch 语句首先得到匹配，第二个 catch 语句将不会被执行。编译时将出现“catch not reached”的错误。

4、finally

捕获异常的最后一步是通过 finally 语句为异常处理提供一个统一的出口，使得在控制流转到程序的其它部分以前，能够对程序的状态作统一的管理，一般用于收拾现场，做一些收尾工作。例如：当对象占用某些资源后，在执行某方法时发生异常要意外终止对象的声明周期，这时需要对象释放这些资源以保证它们可以被再利用。对象的收尾工作就是释放该对象在生命周期内所占用的各种资源，例如内存、

文件、网络连接等等。JVM 的垃圾回收机制可以自动回收无用对象所占用的内存，但这些对象所占用的其它资源仍需程序员编写程序进行回收。

正如 while 语句、if 语句可以嵌套一样，try...catch...finally...语句也可以嵌套，即在 try 语句块、catch 语句块或 finally 语句块中都可再使用 try...catch...finally...语句。

例 6.1 处理一个算术运算异常

```
public class Example6_1 {
    public static void main(String args[])
    {
        try{
            int a=0;
            int b=5/a;        //发生异常
        }catch(ArithmeticException e){
            System.out.println("发生了被0除异常,具体信息:"+e.getMessage());
        }
    }
}
```

运行结果是：

```
发生了被 0 除异常,具体信息:/ by zero
```

上例中的 catch()语句块对异常的处理是输出异常信息，getMessage()方法返回异常对象的详细信息，如果该对象没有详细信息则返回 null。类似的还有 printStackTrace()方法，可以把对异常的跟踪情况打印到标准错误流。

例 6.2 处理数据转换异常

```
public class Example6_2 {
   public static void main(String args[ ]) {
      int n=0,m=0,t=6666;
      try{  m=Integer.parseInt("8888");
            n=Integer.parseInt("ab85"); //发生异常,转向catch
            t=9999;  //t没有机会赋值
      }
      catch(NumberFormatException e) {
            System.out.println("发生异常:"+e.getMessage());
            n=123;
      }
      System.out.println("n="+n+",m="+m+",t="+t);
    }
}
```

运行结果是：

```
发生异常:For input string: "ab85"
n=123,m=8888,t=6666
```

例 6.3 异常的匹配与收尾

```
public class Example6_3 {
    public static void main(String[] args) {
        String[] members = new String[4];
        for (int count = 0; count < 3; count++) {
```

```
            try {
                int x;
                if (count == 0) x = 1 / 0;
                if (count == 1) members[4] = "George Martin";
                if (count == 2) return;
            } catch (ArrayIndexOutOfBoundsException exc) {
                System.out.println("发生了数组下标越界异常！");
            } catch (ArithmeticException exc) {
                System.out.println("发生了被零除异常！");
                continue;
            } finally {
                System.out.println("finally语句块中的语句总是会执行！");
            }
        }
    }
}
```

运行结果是：

```
发生了被零除异常！
finally语句块中的语句总是会执行！
发生了数组下标越界异常！
finally语句块中的语句总是会执行！
finally 语句块中的语句总是会执行！
```

从上面的运行结果看出，例 6.3 用一个循环来演示不同的执行流程。第一次循环 try 语句块将抛出被零除的异常，该异常被第二个 catch 语句块捕获，在执行该语句块的打印语句后，虽然有 continue 语句（有 break 语句也一样），但仍将执行 finally 语句块的语句。第二次循环 try 语句块将抛出数组下标越界的异常，该异常被第一个 catch 语句块捕获，在执行该语句块的打印语句后会跳过第二个 catch 语句块，但仍将执行 finally 语句块的语句。第三次循环没有任何异常发生并且应该执行 return，但在 main() 方法返回之前仍将执行 finally 语句块的语句。

注意：当 try 语句块的语句抛出一个不与任何 catch 后面声明的异常类型相匹配的异常或者根本没有抛出异常时，仍将执行 finally 语句块中的语句，然后才将控制转向捕获该异常的（上层）方法。

6.2.3 抛弃异常

如果在一个方法中产生了一个异常，但是这一方法并不确切地知道该如何对这一异常事件进行处理，这时，该方法就应该声明抛弃异常，由调用者捕获与处理。但调用者在捕获后，如果还没有足够信息进行最适当处理可以再转发给调用者的调用者，逐层报告，直到最适合处理该异常的方法。

当方法要抛出需要检查的异常时，具体来说，当出现下述两种情况之一时，该方法需要在方法体之前使用 throws 语句声明要抛出该异常类或该异常类的祖先类：

（1）该方法用 throws 语句显式抛出某种需要检查的异常；或者，

（2）该方法所调用的方法会抛出这种异常，而该方法又没有使用 catch 语句块对其进行处理。

如果要抛出异常，只需在 catch 语句块中重新抛出所捕获的异常即可：

```
public void someMethod(...) throws CannotHandleException{
    ... // 其它语句
    try {
        ... // 可能引发各种异常的语句
    } catch (CanHandleException exc) {
```

```
            ... // 处理能够处理的异常
        } catch (CannotHandleException exc) {
            throw exc; // 转发不能处理的异常
        }
        ... // 其它语句
    }
```

在方法声明中的 throws 子句中，表示该方法执行过程中可能会抛出异常及该异常的类型，throws 子句中同时可以指明多个异常，之间由逗号隔开。例如：

```
    public void someMethod(…) throws    IOException, IndexOutOfBoundsException {
        …
    }
```

例 6.4 抛出异常

```
    //自定义异常类
    class OutOfMoneyException extends Exception {
        public OutOfMoneyException() {
            super("你的帐户金额不足!");
        }
        public OutOfMoneyException(String msg) {
            super(msg);
        }
    }
    //定义帐户类
    class Account {
        private double balance = 1000;
        public void transfer(double amount) throws OutOfMoneyException {
            if (balance < amount)  //抛出金额不足异常
                throw new OutOfMoneyException("[余额为:" + balance + " < 要取出金额:"
+ amount + "]");
            balance = balance - amount;
        }
        public double getBalance() {
            return balance;
        }
    }
    public class Example6_4 {
        public static void main(String[] args) {
            Account obj = new Account();
            System.out.println("帐户总金额为:"+obj.getBalance());
            double amount = 800;
            for (int count = 1; count < 4; count++) {
                System.out.println("第"+count+"取款");
                try {
                    obj.transfer(amount);
                    System.out.println("取出金额: " + amount +"后, 余额为:
```

```
"+obj.getBalance());
                } catch (OutOfMoneyException exc) {
                    exc.printStackTrace();
                }
            }
        }
    }
```

运行结果是：

```
帐户总金额为:1000.0
第1取款
取出金额: 800.0后, 余额为: 200.0
第2取款
OutOfMoneyException: [余额为:200.0 < 要取出金额:800.0]
    at Account.transfer(Example6_4.java:13)
    at Example6_4.main(Example6_4.java:28)
第3取款
OutOfMoneyException: [余额为:200.0 < 要取出金额:800.0]
    at Account.transfer(Example6_4.java:13)
    at Example6_4.main(Example6_4.java:28)
```

例 6.5 try-catch-finally 嵌套的例子

```
class MyoneException extends Exception { } //自定义异常
public class Example6_5 {
    public static void main(String[] args) {
        System.out.println("进入第一层try");
        try {
            System.out.println("进入第二层try");
            try {
                throw new MyoneException(); //生成并抛出了个异常
            } finally {
                System.out.println("第二层try语句块中的善后处理");
            } //try-catch-finally嵌套在try限定的范围内。
        } catch (MyoneException e) {
            System.err.println("捕获第一层try语句块中的异常");
        } finally {
            System.err.println("第二层try语句块中的善后处理");
        }
    }
}
```

运行结果是：

```
进入第一层try
进入第二层try
第二层try语句块中的善后处理
捕获第一层try语句块中的异常
第二层 try 语句块中的善后处理
```

最后，我们在使用异常时，有以下几点建议需要注意：

（1）对于运行时例外，如果不能预测它何时发生，程序可以不做处理，而是让 Java 虚机去处理它。

（2）如果程序可以预知运行时例外可能发生的地点和时间，则应该在程序中进行处理，而不应简单地把它交给运行时系统。

（3）在自定义异常类时，如果它所对应的异常事件通常总是在运行时产生的，而且不容易预测它将在何时、何处发生，则可以把它定义为运行时异常，否则应定义为非运行时异常。

小结

面向对象程序设计语言往往利用异常处理机制帮助提高程序的可靠性。异常处理机制是用于管理程序运行期间异常的一种结构化方法。

本章系统地介绍了 Java 语言提供的一种处理机制，包括异常的概念、Java 语言中的异常分类、异常的抛出、异常的捕获、异常的处理等。

Java 语言异常处理机制的基本思想是将异常的检测与处理分离。Java 语言将异常定义为类，可像普通类一样创建异常类的对象实例。Java 语言将异常类组织成层次结构，并划分为需要检查的和无需检查的异常类，编译器将对需要检查的异常类进行检查以确定抛出这种异常类的方法会捕获或声明抛出该异常类。强制方法声明它所抛出的需要检查的异常对提高程序的健壮性很有帮助。

当方法检测到异常条件存在但无法确定相应处理方法时，用 throw 语句引发异常。方法的直接或间接调用者用 try 语句和 catch 语句捕获并处理这个异常，并可使用 finally 语句执行一些必要的收尾工作。

习题

一、基本概念

1、什么叫异常？简述 Java 语言的异常处理机制。

2、简述抛出异常、捕获异常或处理异常的基本过程。

4、如何自己声明异常？在处理这种异常时应该注意什么？

5、如何将成员方法中没有捕获的异常抛给调用该方法的方法？

二、编程实践

1、设计已给一元一次方程类，并为这个异常类添加异常处理。

2、编写一个程序，首先定义一个异常类，其次定义一个带主方法 main()的类，在 main()的 try 语句块抛出该异常类，在相应的 catch 语句块捕获该异常，并打印调用栈，使用 finally 语句块，在该语句块打印一些信息，以验证该语句块的语句总是会执行。

3、编写一个程序，首先定义两个异常类，其次定义一个类，其中有两个方法，一个方法抛出你所定义的某个异常类，另一个方法调用前一个方法，并捕获其抛出的异常，在处理时抛出另外一个异常类。

第 7 章　字符串处理

字符串是多个字符的序列，是编程中常用的数据类型。从某种程度上看，字符串有些类似于字符数组。实际上在 C 语言中，字符串就是用字符数组来实现的。但在纯面向对象的 Java 语言中，将字符串数据类型封装为字符串类，无论是字符串常量还是字符串变量，都是用类的对象来实现的。

Java 语言提供了两种具有不同操作方式的字符串类：String 类和 StringBuffer 类。它们都是 java.lang.Object 的子类。用 String 类创建的对象在操作中不能变动和修改字符串的内容，因此也被称为字符串常量。而用 StringBuffer 类创建的对象在操作中可以更改字符串的内容，因此也被称为字符串变量。也就是说，对 String 类的对象只能进行查找和比较等操作，而对于 StringBuffer 类的对象可以进行添加、插入、修改之类的操作。

7.1　String 类

String 类（字符串类）的对象是一经创建便不能变动内容的字符串常量。在 Java 语言中，字符串常量通常是作为 String 类的对象而存在，有专门的数据成员来表明它的长度。

7.1.1 创建 String 对象

Java 语言规定字符串常量必须用双引号括起，一个串可以包含字母、数字和各种特殊字符，如+、-、*、/、$等。

例如：　System.out.println("OK！");

中的"OK！"就是字符串常量。Java 的任何字符串常量都是 String 类的对象，只不过在没有用明确命名时，Java 自动为其创建一个匿名 String 类的对象，所以，它们也被称为匿名 String 类的对象。我们可以用下面的方法创建 String 类的对象。例如：

String c1="Java";

上面的语句创建 String 类的对象 c1，并通过赋值号将匿名 String 类的对象"Java"赋值给 c1 引用。String 类的对象一经创建，便有一个专门的数据成员来记录它的长度。

String 类中提供了多种构造方法来创建 String 类的对象。见表 7-1

表 7-1　String 类的构造方法

构造方法	说明
String( )	创建一个空字符串对象
String(String value)	用串对象 value 创建一个新的字符串对象，value 可以是字符串或 String 类的对象。
String(char value[ ])	用字符数组 value[]来创建字符串对象。
String(char value[ ],int offset,int count)	从字符数组 value 中下标为 offset 的字符开始，创建有 count 个字符的串对象。
String(byte ascii[ ])	用 byte 型字符串数组 ascii，按缺省的字符编码方案创建串对象。
String(byte ascii[ ]，int offset int count))	从字节型数组 ascii 中下标为 offset 的字符开始，按缺省的字符编码方案创建 count 个字符的串对象。
String(StringBuffer Buffer)	构造一个新的字符串，其值为字符串的当前内容。

例 7.1　String 类的 7 种构造方法的使用。

```
import java.io.*;
public class Example7_1
{
    public static void main(String[] args)
    {
        char charArray[]={'b','i','r','t','h',' ','d','a','y'};
```

```
        byte byteArray[]={-61,-26,-49,-14,-74,-44,-49,-13};
        StringBuffer buffer;
        String s,s1,s2,s3,s4,s5,s6,s7,s8;
        s=new String("hello");
        s8="ABC";
        buffer=new StringBuffer("Welcom to java programming!");
        s1=new String();
        s2=new String(s);
        s3=new String(charArray);
        s4=new String(charArray,6,3);
        s5=new String(byteArray);
        s6=new String(byteArray,2,4);
        s7=new String(buffer);
        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);
        System.out.println("s8="+s8);
        System.out.println("buffer="+buffer);
    }
}
```

运行结果如下：

```
s1=
s2=hello
s3=birth day
s4=day
s5=面向对象
s6=向对
s7=Welcom to java programming!
s8=ABC
buffer=Welcom to java programming!
```

7.1.2 String 类的常用方法

创建一个 String 类的对象后，使用相应类的成员方法对创建的对象进行处理，以完成编程所需要的功能。Java.lang.String 常用成员方法如下表所示：

表 7-2 Java.lang.String 常用成员方法

成员方法	功能说明
int length()	返回当前串对象的长度
char charAt(int index)	返回当前串对象下标 int index 处的字符
int indexof(int ch)	返回当前串内第一个与指定字符 ch 相同的下标，若找不到，则返回-1
int indexOf(String str,int fromIndex)	从当前下标 fromIndex 处开始搜索，返回第一

	个与指定字符串 str 相同的第一个字母在当前串中的下标，若找不到，则返回-1
String substring(int beginIndex)	返回当前串中从下标 beginIndex 开始到串尾的子串
String substring(int beginIndex,int endIndex)	返回当前串中从下标 beginIndex 开始到下标 endIndex-1 的子串
boolean equals(Object obj)	当且仅当 obj 不为 null 且当前串对象与 obj 有相同的字符串时，返回 true 否则返回 flase
boolean equalsIgnoreCase(String s)	功能与 quals 类似，equalsIgnoreCase 在比较字符串时忽略大小写
int compareTo(String another_s)	比较两字符串的大小。返回一个小于、等于或大于零的整数。返回的值取决于此字符串是不小于、等于或大于 another_s
String concat(String str)	将字符串 str 连接在当前串的尾部，返回新的字符串
String replace(char oldCh,char newCh)	将字符串的字符 oldCh 替换为字符串 newCh
String toLowerCase()	将字符串中的大写字符转换为小写字符
String toUpperCase()	将字符串中的小写字符转换为大写字符
static String valueOf(type variable)	返回变量 variable 值的字符串形式。Type 可以是字符数组
static String valueOf(char[] data, int offset,int count)	返回字符数组 data 从下标 offset 开始的 count 个字符的字符串
static String valueOf(Object obj)	返回对象 obj 的字符串
String toString ()	返回当前字符串

1. 访问字符串对象

用于访问字符串对象的信息常用到的成员方法有 length()、charAt(int index)、indexOf(int ch)、substring(int beginIndex,int endIndex)等，例 7.2 演示它们的使用方法和作用。

例 7.2 访问字符串对象

```
public class Example7_2
{
    public static void main(String args[])
    {
        String s1="Java Application";
        int len1=s1.length();
        int len2="ABCD".length();
        char c1="12ABG".charAt(3);
        char c2=s1.charAt(3);
        int n1="abj".indexOf('a');
        int n2=s1.indexOf('J');
        int n3="abj".indexOf("bj",0);
        int n4=s1.indexOf("va",1);
        String s2="abcdefg".substring(4);
        String s3=s1.substring(3,9);
        System.out.println("s1是"+s1+" 的长度为="+len1);
```

```
            System.out.println("ABCD  的长度为="+len2);
            System.out.println("\"12ABG\".charAt(3)="+c1+"
s1.chartAt(3)="+c2);
            System.out.println("\"abj\".indexOf('a')="+n1+"
s1.indexOf('J')="+n2);
            System.out.println("\"abj\".indexOf(\"bj\",0)="+n3+"
s1.indexOf(\"va\")="+n4);
            System.out.println("\"abcdefg\".substring(4)="+s2);
            System.out.println("s1.substring(3,9)="+s3);
        }
    }
```

运行结果：

```
s1是Java Application 的长度为=16
ABCD  的长度为=4
"12ABG".charAt(3)=B    s1.chartAt(3)=a
"abj".indexOf('a')=0    s1.indexOf('J')=0
"abj".indexOf("bj",0)=1  s1.indexOf("va")=2
"abcdefg".substring(4)=efg
s1.substring(3,9)=a Appl
```

2. 字符串比较

常用的字符串比较成员方法有：equals()、equalsIgnoreCase()及 compareTo()。

例 7.3 字符串比较运算成员方法的使用

```
public class Example7_3
{
    public static void main(String args[])
    {
        String s1="Java";
        String s2="java";
        String s3="Java";
        boolean b1=s1.equals(s2);
        boolean b2=s1.equals(s3);
        boolean b3=(s1==s2);
        boolean b4=s1.equalsIgnoreCase(s2);
        int n1=s1.compareTo(s2);
        int n2=s2.compareTo(s1);
        int n3=s1.compareTo(s3);
        System.out.println("s1="+s1+"\t s2="+s2+"\t s3="+s3);
        System.out.println("s1.equals(s2)返回的值为:"+b1);
        System.out.println("s1.equals(s3)返回的值为:"+b2);
        System.out.println("(s1==s3)返回的值为:"+b3);
        System.out.println("s1.equalsIgnoreCase(s2)返回的值为:"+b4);;
        System.out.println("s1.compareTo(s2)返回的值为:"+n1);;
        System.out.println("s2.compareTo(s1)返回的值为:"+n2);
        System.out.println("s1.compareTo(s3)返回的值为:"+n3);
```

```
        }
    }
```

运行结果是：

```
s1=Java s2=java    s3=Java
s1.equals(s2)返回的值为:false
s1.equals(s3)返回的值为:true
(s1==s3)返回的值为:false
s1.equalsIgnoreCase(s2)返回的值为:true
s1.compareTo(s2)返回的值为:-32
s2.compareTo(s1)返回的值为:32
s1.compareTo(s3)返回的值为:0
```

上例中，值得注意的是字符串 s1 与字符串 s3 的比较，我们用了两种方式：s1.equals(s3)返回值为 true，而 s1==s2 的返回值为假，这是因为字符串 s1、s3 仅是对象引用，内存示意图如图 7_1 所示。

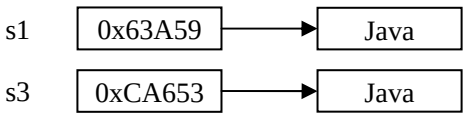

图 7_1 字符串变量的内存示意图

当用 s1==s2 进行比较时，比较的是 s1 与 s2 的地址，显然不等。所以判断字符串是否相等是不能用“==”的。

例 7.4 字符串数组的排序

```
public class Example7_5{
    public static void main(String args[]){
        String a[]={"Java","Basic","C++","Fortran","SmallTalk"};
        for(int i=0;i<a.length-1;i++){
            for(int j=i+1;j<a.length;j++){
                if(a[j].compareTo(a[i])<0){
                    String temp=a[i];
                    a[i]=a[j];
                    a[j]=temp;
                }
            }
        }
        for(int i=0;i<a.length;i++) {
            System.out.print(" "+a[i]);
        }
    }
}
```

运行结果是：

```
Basic C++ Fortran Java SmallTalk
```

3. 字符串操作

字符串操作是指用已有的字符串对象产生新的字符串对象。常用的成员方法有：concat( )、replace( )、toLowerCase( )、toUpperCase( )等。

例 7.5 字符串的连接、替换和字母大小写转换操作。

```
public class Example7_4 {
    public static void main(String args[])
    {
        String s1="Java";
        String s2="Welcome";
        String sc1=s2.concat(s1);
        String sr1=s2.replace('e','r');
        String w1=s1.toLowerCase();
        String u2=s1.toUpperCase();
        System.out.println("s1="+s1+"\ts2="+s2);
        System.out.println("s2+s1="+sc1);
        System.out.println("s2.replace('e','r')="+sr1);
        System.out.println("s1转换为小写="+w1);
        System.out.println("s1转换为大写="+u2);
    }
}
```

运行结果是：

```
s1=Java s2=Welcome
s2+s1=WelcomeJava
s2.replace('e','r')=Wrlcomr
s1转换为小写=java
s1 转换为大写=JAVA
```

7.1.3 其它数据类型与字符串的相互转换

String 类中的 valueOf(参数)成员方法可以将参数类型的数据转换成字符串。这些参数的类型可以是：boolean，char，int，long，float，double 和对象。另外，因为所有类都默认是 java.lang.Object 类的子类或间接子类，Object 类有一个 public 方法 toString()，各个子类可以根据情况对它进行重写，所有类的对象通过调用该方法可以获得该对象的字符串表示。

例 7.6 其它数据类型转换为字符串

```
public class Example7_6
{
    public static void main(String args[])
    {
        int nInt = 10;
        float fFloat = 3.14f;
        double dDouble = 3.1415926;
        boolean bBoolean = true;
        Integer obj1 = new Integer(nInt);
        Float obj2 = new Float(fFloat);
        Double obj3 = new Double(dDouble);
        Boolean obj4 = new Boolean(bBoolean);
        //调用toString方法转换为字符串
        String s1 = obj1.toString();
        String s2 = obj2.toString();
        String s3 = obj3.toString();
```

```
        String s4 = obj4.toString();
        //使用String类中的valueOf(参数)成员方法转换为字符串
        String s5 = String.valueOf(nInt);
        String s6 = String.valueOf(fFloat);
        String s7 = String.valueOf(dDouble);
        String s8 = String.valueOf(bBoolean);
        System.out.println(s1+"\t"+s5);
        System.out.println(s2+"\t"+s6);
        System.out.println(s3+"\t"+s7);
        System.out.println(s4+"\t"+s8);
    }
}
```

运行结果是：

```
10  10
3.14    3.14
3.1415926  3.1415926
true    true
```

Boolean、Byte、Double、Float、Integer、Long 等类也分别提供了静态方法 parseDouble(String)、parseFloat(String)、parseInt(String)、 parseLong(String) 等方法将对象转换成其他简单数据类型的方法。其方法声明格式如下：

static boolean parseBoolean(String s)　字符串转换为 boolean
static int parseInt(String s[, int radix])　字符串转换为 int
static byte parseByte(String s)　字符串转换为 byte
static double parseDouble(String s)　字符串转换为 double
static float　parseFloat(String s)　字符串转换为 float

例 7.7 字符串转换为其他数据类型

```
public class Example7_7{
    public static void main(String args[]){
        int nInt;
        float fFloat;
        double dDouble;
        //生成相应的数据类型
        String strString = new String("I love Java");
        String strInteger = new String("314");
        String strFloat = new String("3.14");
        String strDouble = new String("3.1416");
        //分别调用各类中的静态方法
        nInt = Integer.parseInt(strInteger);
        System.out.println(nInt);
        fFloat = Float.parseFloat(strFloat);
        System.out.println(fFloat);
        dDouble = Double.parseDouble(strDouble);
        System.out.println(dDouble);
    }
```

```
}
```
运行结果是：
```
314
3.14
3.1416
```
7.2 StringBuffer 类

StringBuffer 类(字符串缓冲器类)与 String 类不同，StringBuffer 类是一个在操作中可以更改其内容的字符串类，即一旦创建了 StringBuffer 类的对象，那么在操作中便可以更改和变动字符串的内容。也就是说，对于 StringBuffer 类的对象，不仅能进行查找和比较等操作，还可以进行添加、插入、修改之类的操作。

7.2.1 创建 StringBuffer 对象

StringBuffer 类提供了三种构造方法来创建其对象，其格式如下：

- StringBuffer()　构造一个不包含字符的字符串缓冲区，其初始的容量设为 16 个字符。
- StringBuffer(int len) 构造一个不包含字符的字符串缓冲区，其初始容量由参数 len 设定。
- StringBuffer(String str)　构造一个字符串缓冲区，来表示和字符串参数相同的字符序列。字符串缓冲区的初始容量为 16 加上字符串 str 的长度。

7.2.2　StringBuffer 类的常用方法

StringBuffer 类主要用于完成字符串的动态添加、插入、替换等操作，其常用成员方法见表 7-3。

表 7-3　Java.lang.StringBuffer 常用成员方法

成员方法	功能说明
int length()	返回当前缓冲区中字符串的长度
char charAt (int index)	返回当前缓冲区中字符串下标 index 处的字符
void setcharAt (int index,char ch)	将当前缓冲区中字符串下标 index 处的字符改变成字符 ch 的值
int capacity()	返回当前缓冲区长度
StringBuffer append(Object obj)	将 obj.toString()返回的字符串添加到当前字符串的末尾
StringBuffer append(type variable)	将变量值转换成字符串再添加到当前字符串的末尾。type 可以是字符数组、串和各种基本类型
StringBuffer append (char[]str,int offset,int len)	将数组从下标 offset 开始的 len 个字符依次添加到当前字符串的末尾
StringBuffer insert(int offset, Object obj)	将 obj.toString()返回的字符串插入当前字符下标 offset 处
StringBuffer insert(int offset, type variable)	将变量值转换成字符串，插入到当前字符数组下标为 offset 的位置处
StringBuffer delete(int start，int end)	删除字符串缓冲区中起始序号为 start、终止序号为 end-1 的字符
StringBuffer deleteCharAt(int index)	删除字符串缓冲区中指定位置的字符
StringBuffer replace(int start,int end,String str)	将字符串缓冲区中起始位置为 start、终止位置为 end 的字符替换为由字符串 str 指定的内容。
String toString()	将可变字符串转化为不可变字符串

1．添加操作

该功能主要由 StringBuffer 类中成员方法 append 完成，其作用就是将一个字符添加到另一个字符串缓冲区的后面。在应用中，如果添加字符的长度超过字符串缓冲区的的长度，则字符串缓冲区自动将长

度进行扩充。

该方法可用来向字符串缓冲区添加逻辑变量、字符、字符数组、双精度数、浮点数、整型数、长整型数、对象类型的字符串和字符串等。上述方法的返回类型均为 StringBuffer。

例如：

StringBuffer sbfSource=new StringBuffer("1+2=”)；

int nThree=3；

sbfSource.append(nThree)；

System.out.println(sbfSource.toString( ))；

输出结果为： 1+2=3

2．插入操作

字符串缓冲区 StringBuffer 的插入操作主要用于动态地向 StringBuffer 中添加字符。根据构造方法中的参数类型，可以向字符串缓冲区插入逻辑变量、字符、字符数组、双精数、浮点数、整型数、长整型数、对象类型的字符串和字符串等。上述方法的返回类型为 StringBuffer。

例如：

StringBuffer sbfSource=new StringBuffer("1+=2")；

int nOne=1；

sbfSource.insert(2，nOne)；

System.out.println(sbfSource.toString())；

输出结果为：1+1=2

3. 删除字符

下面的代码段为 delete 方法的例子：

StringBuffer sbfSource=new StringBuffer("You are the best")；

sbfSource.delete(0,3)； //结果为： are the best!

4．内容替换

下面的代码段为 replace 方法的例子：

StringBuffer sbfSource=new StringBuffer("You are the best!")；

Stringstr=new String("I'm")；

sbfSource.replace(0,7,str)； //结果为：I'm the best!

其他方法见表 7-3 的介绍。

例 7.8 字符串反串

```
public class Example7_8 {
    public static void main(String args[]){
        String strSource = new String("I love Java");
        String strDest = reverseIt ( strSource );
        System.out.println(strDest);
    }
    public static String reverseIt(String source) {
        int i, len = source.length();
        StringBuffer dest = new StringBuffer(len);
        for (i = (len - 1); i >= 0; i--)
            dest.append(source.charAt(i));
        return dest.toString();
    }
}
```

运行结果是：

```
avaJ evol I
```

7.3 StringTokenizer 类

有时需要分析字符串并将字符串根据分隔符分解成可被独立使用的单词，这时可以使用 StringTokenizer 类，该类提供三种形式的构造函数：

StringTokenizer(String str)

StringTokenizer(String sb，String delim)　// delim 为分隔符号

StringTokenizer(String Sb，String delim，boolean returnTokens)

在对一个字符串进行解析的时候，在字符串中必须包括一个用于解析的分隔符号。Java 置默认的分隔符为空格、制表符('\t')、换行符('\n')、回车符('\r')。如果在程序计中想采用自定义的分隔符，可以通过在构造函数中指定参数 delim 项来设置用户分隔符。另外，如果 returnTokens 标志为 true, 则分隔符字符也被作为标记返回。每个分隔符作为长度为一的字符串返回。如果标志为 false, 则跳过分隔符字符，且把它作为标记之间的分隔符。

例如：

```
StringTokenizer fenxi=new StringTokenizer("we are student");
StringTokenizer fenxi=new StringTokenizer("we ,are ; student", ", ; ");
StringTokenizer fenxi=new StringTokenizer("we ,are ; student", ", ; ",true);
```

相应地，在 StringTokenizer 类中提供了相应的成员方法：

1．统计分隔符数量

- public int countTokens()

该方法返回的是字符串中的当前单词的数量，为整数。例如：

```
String str=new String("I love Java");
StringTokenizer st=new StringTokenizer(str);
int nTokens=st.countTokens();　// 值为 3
```

2．匹配和寻找分隔符

通常，下面的两个组合方法均可以用来完成分隔符的寻找和匹配：

- public boolean hasMoreTokens()　hasMoreTokens()方法判断在字符串中是否还有已经定义的分隔符。如果有，除分隔符后、到下一个分隔符之前的内容进行一次循环。如果没有则终止循环。
- public String nextToken()　nextToken()方法逐个获取字符串中的单词，每当调用 nextToken()时，都将在字符串中获得下一个单词，每当获取到一个单词，字符串分析器中的负责计数的变量的值就自动减一，该计数变量的初值等于字符串的单词。

例 7.9 分析字符串，输出字符串中的单词，并统计单词个数

```
import java.util.*;
public class Example7_9{
    public static void main(String args[]){
        String s="This is a Java programming, I like Java";
        StringTokenizer fenxi=new StringTokenizer(s," ,"); //空格和逗号做分
        int number=fenxi.countTokens();
        while(fenxi.hasMoreTokens())
        {
            String str=fenxi.nextToken();
            System.out.print (str +"\t");
            System.out.println("还剩"+fenxi.countTokens()+"个单词");
        }
```

```
            System.out.println("s共有单词"+number+"个");
        }
    }
```

运行结果是：

```
This    还剩7个单词
is  还剩6个单词
a   还剩5个单词
Java    还剩4个单词
programming     还剩3个单词
I   还剩2个单词
like    还剩1个单词
Java    还剩0个单词
s共有单词8个
```

小结

用 String 类创建的对象在操作中不能变动和修改字符串的内容，因此也被称为字符串常量。而用 StringBuffer 类创建的对象在操作中可以更改字符串的内容，因此也被称为字符串变量。

String 类中的 valueOf(参数)成员方法可以将参数类型的数据转换成字符串。这些参数的类型可以是：boolean，char，int，long，float，double 和对象。

需要分析字符串并将字符串根据分隔符分解成可被独立使用的单词，这时可以使用 StringTokenizer 类。

习题

一、基本概念

1、在 Java 中，字符串用两个类描述： String 类与 StringBuffer 类，各有什么特点？

2、举例如何用 StringTokenizer 类分析字符串。

二、编程实践

1、编程测试 String 类提供的各种方法的使用。

2、编程测试 StringBuffer 类提供的各种方法的使用。

3、编程测试得用 StringTokenizer 类将字符串分解为用特定分隔符分隔的单词

4、编写一个 Java 程序，从键盘输入两个字符序列，第一个字符序列给 string1，第二个字符序列给 string2，查找在 string1 中是否存在 string2，并输出相应的信息。

第 8 章 线程

以往开发的程序大多是单线程，即一个程序只有一条从头至尾的执行线索。然而现实世界中的很多过程都具有多条线索同时动作的特性：例如，我们可以一边上网聊天，一边听音乐。Java 程序通过流控制来执行线索，程序中单个顺序的流控制称为线程，那么，多线程则指的是在单个程序中可以同时运行多个不同的线程，执行不同的任务，这就意味着同时存在几个执行体，按几条不同的执行线索共同工作，能同时处理多个任务，使得它们看上去几乎在同一时间内同时运行，但这仅是一种错觉，是由于 Java 能够快速地把控制从一个线程切换到另一个线程。

8.1 线程的概念

线程与进程相似，是一段完成某个特定功能的代码，是程序中单个顺序的流控制；但与进程不同的是，同类的多个线程共享一块内存空间和一组系统资源，而线程本身的数据通常只有微处理器的寄存器数据，以及一个供程序执行时使用的堆栈。所以系统在产生一个线程，或者在各个线程之间切换时，负担要比进程小的多，正因如此，线程被称为轻负荷进程（light-weight process）。一个进程中可以包含多个线程。一个线程是一个程序内部的顺序控制流。主要区别如下：

（1）进程：每个进程都有独立的代码和数据空间（进程上下文），进程切换的开销大。

（2）线程：轻量的进程，同一类线程共享代码和数据空间，每个线程有独立的运行栈和程序计数器（PC），线程切换的开销小。

（3）多进程：在操作系统中，能同时运行多个任务程序。

（4）多线程：在同一应用程序中，有多个顺序流同时执行。

使用多线程进行程序设计具有如下优点：

（1）多线程编程简单，效率高（能直接共享数据和资源，多进程不能）

（2）适合于开发服务程序（如 Web 服务，聊天服务等）

（3）适合于开发有多种交互接口的程序（如聊天程序的客户端，网络下载工具）

（4）减轻编写交互频繁、涉及任务多的程序的困难（如监听网络端口）

（5）程序的吞吐量会得到改善（同时监听多种设备，如网络端口、串口、并口以及其他外设）

（6）有多个处理器的系统，可以并发运行不同的线程（否则，任何时刻只有一个线程在运行）

8.2 线程的控制与调度

每个 Java 程序都有一个缺省的主线程。对于 Application，主线程是 main( )方法执行的线程索。对于 Applet，主线程指挥浏览器加载并执行 Java 小程序。要想实现多线程，必须在主线程中创建新的线程对象。

8.2.1 创建线程

在 Java 语言中创建线程对象有两种途径：一是以创建 Thread 类的子类为途径，二是以实现 Runnable 接口为途径。用实现 Runnable 接口的方式创建线程与用继承 Thread 类的方式创建线程无本质差别，但是，由于 Java 不支持多继承，因此任何类如果已经继承了某一类时，就无法再继承 Thread 类，这时只能通过实现接口 Runnable 的方式创建线程对象。例如，因为小应用程序已经继承了 Applet 类，所以不能再继承 Thread 类，而只能通过 Runnable 接口实现多线程。

1. 采用继承创建线程

在 Java 语言中创建线程对象的途径之一是创建 Thread 类的子类。Thread 类（线程类）是 java.lang 包中的一个专门用来创建线程和对线程进行操作的类。创建 Thread 类的子类时，首先应声明子类的构造方法，其次应用自己定义的 run( )方法去覆盖 Thread 类的 run( )方法，即将自己要执行的程序区块写入 run( )方法中，因为父类的 run()方法中没有任何操作语句。

Java 在 Thread 类中定义了许多方法，这些方法可分为四组：

（1）构造方法。用于创建用户的线程对象，表 8-1 列示了 Thread 类的构造方法。

（2）run()方法。用于定义用户线程所要执行的的操作。

（3）改变线程状态的方法。如 start()、sleep()、stop()、suspend()、resume()、yield()和 wait()方法等。这是最常用的一组方法。

（4）其它方法。如 setPriority()、setName()等。

表 8-1 Thread 类的构造方法

构造方法	说明
Thread()	构造一个新线程，用此方式创建的线程必须覆盖 run()方法
Thread(Runnable target)	构造一个新线程，使用指定对象 target 的 run()方法
Thread(ThreadGroup group,Runnable target)	在指定的线程组 group 中构造一个新的线程，使用指定对象 target 的 run()方法
Thread(String name)	用指定字符串名 name 构造一个新线程
Thread(ThreadGroup group,String name)	在指定的线程组 group 中用指定字符串名 name 构造一个新线程
Thread(Runnable target,String name)	用指定字符串名 name 构造一个新线程，使用指定对象 target 的 run()方法
Thread(ThreadGroup group, Runnable target,String name)	在指定的线程组 group 中使用字符串名 name 构造一个新线程，并使用指定对象 target 的 run()方法

表 8-2 Java.lang.Thread 的常用方法

成员方法	说明
void run()	此线程的线程体，在启动该线程后调用此方法。可以通过使用 Thread 类的子类来重载此方法
synchronized void start()	启动线程的执行，此方法引起 run()方法的调用，调用后立即返回。如果已经启动此线程，就抛出 IllegalThreadedStateException 异常
static Thread currentThread()	返回当前处于运行状态的 Thread 对象
static void yield()	使当前执行的 Thread 对象退出运行状态，使其进入等待队列
static void sleep(long millis)throws InterruptedException	使当前执行的线程睡眠 millis 毫秒。如果另一个线程已经中断了此线程，就抛出 InterruptedException 异常
static void sleep(long millis,int nanos)	使当前执行的线程睡眠 millis 毫秒和附加的 nanos 毫秒。如果另一个线程已经中断了这个线程，则抛出 InterruptedException 异常
final void stop()	停止线程的执行
final synchronized void stop(Throwable o)	通过抛出对象停止线程的执行。正常情况下，用户应该在调用 stop 方法时不用任何参数。但是，在某些特殊的环境中，通过 stop 方法来结束线程，并可抛出另一个对象
void interrupt()	中断一个线程
static boolean interrupted()	询问线程是否已经被中断
boolean isInterrupted()	询问另一个线程是否已经被中断

void destroy()	消毁一个线程
final boolean isAlive()	返回表示该线程活动状态的 boolean 值
final void suspend()	挂起这个线程的执行
final void resume()	恢复这个线程的执行，此方法仅在使用 suspend() 后才有效
final void setPriority(int newPriority)	设置该线程的优先级，如果优先级不在 MIN-PRIORITY 到 MAX-PRIORITY 的范围内，就抛出 IllegalArgumentException 异常
final int getPriority()	获取并返回此线程的优先级
final void setName(String name)	设置该线程名为 name
final String getName()	获取并返回此线程名
final ThreadGroup getThreadGroup()	获取并返回此线程组
static int activeCount()	返回此线程组中当前活动的线程数量
static int enumerate(Thread tarray[])	将此线程组中的每一个活动线程拷贝到指定的数组 tarray 中，返回放到此数组中线程的数量
final synchronized void join(long millis)	等待此线程死亡，可以指定一个以 millis 给出的等待时间（以毫秒为单位），如果为 0 则表示永远等待。如果另一个线程已经中断这个线程，那么就抛出 InterruptedException 异常
final synchronized void join(long millis ,int nanos)	等待此线程死亡，可以给出 millis 的等待时间和 nanos 附加时间。如果另一个线程已经中断此线程，则抛出 InterruptedException 异常
final void join()	无限期等待此线程死亡，如果另一个线程已经中断此线程，则抛出 InterruptedException 异常
public final boolean isDaemon()	返回线程是否为 daemon 线程
void check Access()	检查当前线程是否允许访问此线程组。如果不允许当前线程访问此线程组，就抛出 SecurityException 异常
String toString()	返回这个线程的字符串表示，包括该线程的名字，优先级和线程组。覆盖 Objeet 类中的 toString 方法

例 8.1 通过继承 Thread 类创建线程

```
public class Example8_1 extends Thread
{
    int count = 0;
    public Example8_1(String name)
    {
        super(name);
    }
    public static void main(String[] args)
    {
        Example8_1 p1 = new Example8_1(" 线程2"); //创建线程实例p1
        Example8_1 p2 = new Example8_1(" 线程1"); //创建线程实例p2
        p1.start(); // 执行线程1
        p2.start(); // 执行线程2
```

```
    }
    //线程类重写父类run（）方法,实现线程的具体操作
    public void run()
    {
        while (true) {
            System.out.println(this.getName() + ":计数 " + count);
            if (++count == 4)return;
        }
    }
}
```

程序运行结果为：

```
线程2:计数 1
线程1:计数 1
线程2:计数 2
线程2:计数 3
线程1:计数 2
线程 1:计数 3
```

当一个 Java 程序中有多个线程同时执行时，由于程序的执行效果受 JVM 线程调度的影响，程序运行的一个显著特征是不确定性，因此，上述程序在不同的计算机运行或在同一台计算机反复运行的结果不尽相同。

2. 通过实现接口创建线程

创建线程对象的另一个途径是实现 Runnable 接口，而 Runnable 接口只有一个方法 run( )，用户新建线程的操作就由这个方法来决定。run( )方法必须由实现此接口的类来实现。定义好 run( )方法之后，当用户程序需要建立新线程时，只要以这个实现了 run( )方法的类为参数创建系统类 Thread 的对象，就可以把用户实现的 run( )方法继承过来。

例 8.2

```
public class Example8_2 implements Runnable
{
    int count = 1, number;
    public Example8_2(int i)
    {
        number = i;
        System.out.println("创建线程 " + number);
    }
    //通过实现Runable接口中的run()方法以实现线程的具体操作
    public void run()
    {
        while (true) {
            System.out.println("线程 " + number + ":计数 " + count);
            if (++count == 4)return;
        }
    }
    public static void main(String args[])
    {
```

```
        for (int i = 0; i < 3; i++)
            new Thread(new Example8_2(i + 1)).start();
    }
}
```

程序运行结果为：

```
创建线程 1
创建线程 2
线程 1:计数 1
创建线程 3
线程 1:计数 2
线程 1:计数 3
线程 2:计数 1
线程 2:计数 2
线程 2:计数 3
线程 3:计数 1
线程 3:计数 2
线程 3:计数 3
```

3. 关于 run()方法中的局部变量

对于具有相同目标对象的线程，当其中一个线程拥有 CPU 资源时，目标对象自动调用接口中的 run()方法，这时，run()方法中的局部变量被分配内存空间，当轮到另一个线程拥有 CPU 资源时，目标对象会再次调用接口中的 run()方法，那么，run()方法中的局部变量会再次分配内存空间。也就是说，run()方法已经启动运行了两次，分别运行在不同的线程中，即运行在不同的时间片内。不同线程的 run()方法中的局部变量互不干扰。我们可以把例 8.1 和例 8.2 中 count 变量声明的位置改到 run()方法中声明，观察程序运行过程生成的每个线程实例对 count 值的修改有没有影响到其他线程的 count 值。

8.2.2 线程生命周期

线程是动态的，具有一定的生命周期，分别经历从创建、执行、阻塞直到消亡的过程。在每个线程类中都定义了用于完成实际功能的 run 方法，这个 run 方法称为线程体（Thread Body）。新建的线程在它一个完整的生命周期中通常要经历新建、就绪、运行、阻塞和死亡五种状态，这五种状态之间的转换关系和转换条件如图 8_1 所示。

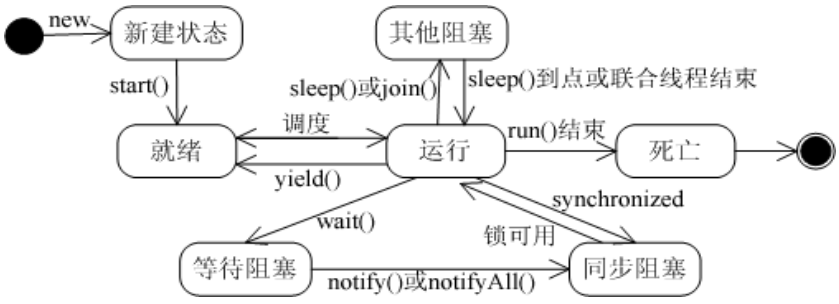

图 8_1 线程的生命周期

1．新建状态

当用 new 关键字和某线程类的构造方法创建一个线程对象后，这个线程对象处于新生状态，此时它已经有了相应的内存空间，并已被初始化。处于该状态的线程可通过调用 start( )方法进入就绪状态。

2．就绪状态

处于就绪状态的线程已经具备了运行的条件，但尚未分配到 CPU 资源，因而它将进入线程队列排队，

等待系统为它分配 CPU。一旦获得了 CPU 资源，该线程就进入运行状态，并自动地调用自己的 run 方法。此时，它脱离创建它的主线程，独立开始了自己的生命周期。

3．运行状态

进入运行状态的线程正在执行自己的 run 方法中的代码。若遇到下列情况之一，则将终止 run 方法的执行。

（1）终止操作。调用当前线程的 stop 方法或 destroy 方法进入死亡状态。

（2）等待操作。调用当前线程的 join(millis)方法或 wait(millis)方法进入阻塞状态。当线程进入阻塞状态时，在 millis 毫秒内可由其他线程调用 notify 或 notifyAll 方法将其唤醒，进入就绪状态。在 millis 毫秒内若不唤醒则须等待到当前线程结束。

（3）睡眠操作。调用 sleep(millis)方法来实现，该方法指定以毫秒为单位的一段时间作为参数，使得线程在指定的时间内进入阻塞状态，不能得到 CPU 时间，指定的时间一过，线程重新进入可执行状态。典型地，sleep() 被用在等待某个资源就绪的情形：测试发现条件不满足后，让线程阻塞一段时间后重新测试，直到条件满足为止。

（4）挂起操作。通过调用 suspend 方法来使得线程进入阻塞状态，并且不会自动恢复，必须其对应的 resume() 被调用，才能使得线程重新进入可执行状态。典型地，suspend() 和 resume() 被用在等待另一个线程产生的结果的情形：测试发现结果还没有产生后，让线程阻塞，另一个线程产生了结果后，调用 resume() 使其恢复。

sleep 方法和 suspend 方法都将使当前正在运行的线程进入阻塞状态，两者不同的是在线程运行过程中调用 sleep 方法后，该线程在不释放占用资源的情况下停止运行指定的睡眠时间。时间到达后，线程重新由 JVM 线程调度器进行调度和管理。而调用 suspend 方法后，线程将释放占用的所有资源，由 JVM 调度转入临时存储空间，直至应用程序调用 resume 方法恢复线程运行。

（5）退让操作。通过调用 yield 方法来实现。当前线程放弃执行，进入就绪状态。

（6）当前线程要求 I/O 时，则进入阻塞状态。

（7）若分配给当前线程的时间片用完时，当前线程进入就绪状态。当前线程的 run 方法执行完，则线程进入死亡状态。

8.2.3 线程的优先级

在 Java 系统中，运行的每个线程都有优先级。Java 将线程的优先级分为 10 个等级，分别用 1-10 之间的数字表示。数字越大表明线程的级别越高。相应地，在 Thread 类中定义了表示线程最低、最高和普通优先级的成员变量 MIN_PRIORITY、MAX_PRIORITY 和 NORMAL_PRIORITY，代表的优先级等级分别为 1、10 和 5。当一个线程对象被创建时，其默认的线程优先级是 5。

设置优先级是为了在多线程环境中便于系统对线程进行调度，优先级高的线程将优先得以运行。Java 线程的优先级是一个在 1～10 之间的正整数，数值越大，优先级越高，未设定优先级的线程其优先级取缺省值 5。Java 线程的优先级设置遵从下述原则：

（1）线程创建时，子线程继承父线程的优先级；

（2）线程创建后，可在程序中通过调用 setPriority()方法改变线程的优先级；

（3）线程的优先级是 1～10 之间的正整数，并用标识符常量 MIN_PRIORITY 表示优先级为 1，用 NORM_PRIORITY 表示优先级为 5，用 MAX_PRIORITY 表示优先级为 10。其他级别的优先级既可以直接用 1～10 之间的正整数来设置，也可以在标识符常量的基础上加一个常数。例如，下面的语句将线程优先级设置为 8。

```
setPriority(Thread.NORM_PRIORITY+3);
```

8.2.4 线程的调度

在单 CPU 的计算机上运行多线程程序，或者当线程数多于处理机的数目时，势必存在多个线程争用 CPU 的情况，这时需要提供一种机制来合理地分配 CPU，使多个线程有条不紊、互不干扰地工作，这种机制称为调度。

JVM 采用一种简单、确定的线程调度算法，称为固定优先级调度策略（fixed priority scheduling）。该算法基于多个可运行线程之间的相对优先级进行调度，即根据线程的优先级决定哪些可运行线程先投入运行。例如，在抢占式操作系统中优先级更高的线程会抢占 CPU 时间，而相同优先级的线程既可能顺序执行也可能分时执行，取决于本地操作系统的线程调度策略。

所谓抢占（pre-empt）是指当一个更高优先级的线程进入可运行状态时，JVM 线程调度程序将终止当前正运行线程而立即转去执行优先级更高的那个线程。虽然 JVM 声称两个相同优先级的线程采用循环执行策略（round-robin），但注意 JVM 本身并未实现相同优先级线程之间的时间分片，仅当运行在支持时间分片的操作系统平台时，这些线程才会有分时执行的效果。

当某一时刻有多个线程可运行时，JVM 将选择其中优先级最高的线程运行，这意味着优先级的绝对数值是无意义的，只起相对的作用。仅当正在运行的线程转换为非运行状态时，低优先级的线程才会投入运行。如果发生下列情况之一，JVM 会终止此线程的运行：

（1）本线程的线程体中调用了 yield( )方法，从而让出了对 CPU 的占有权；

（2）本线程的线程体中调用了 sleep( )方法，使线程进入睡眠状态；

（3）本线程由于 I/O 操作而进入阻塞状态；

（4）另一个具有更高优先级的线程从睡眠状态被唤醒，或其 I/O 操作完成而返回就绪状态。

例 8.3 高优先级线程抢占 CPU 时间

```
public class Example8_3 extends Thread
{
    public Example8_3(String name)
    {
        super(name);
    }
    public static void main(String[] args)
    {
        Example8_3 t1 = new Example8_3("线程1");
        t1.setPriority(Thread.MIN_PRIORITY);
        t1.start();
        Example8_3 t2 = new Example8_3("线程2");
        t2.setPriority(Thread.NORM_PRIORITY);
        t2.start();
        Example8_3 t3 = new Example8_3("线程3");
        t3.setPriority(Thread.MAX_PRIORITY);
        t3.start();
    }
    public void run()
    {
        for (int i = 0; i < 3; i++)
            System.out.println(this.getName() + " 正在运行!");
    }
}
```

程序运行结果：

```
线程3 正在运行!
线程3 正在运行!
线程3 正在运行!
```

```
线程2 正在运行!
线程2 正在运行!
线程2 正在运行!
线程1 正在运行!
线程1 正在运行!
线程 1 正在运行!
```

上例说明，一旦高优先级的线程进入可运行状态，JVM 线程调度程序将立即停止当前较低优先级线程的运行，而转去执行那个高优先级的线程，直至该线程执行完毕才重新执行低优先级的线程。

注意，在实际编程中，不提倡使用线程的优先级来保证算法的正确执行。要编写正确、跨平台的多线程代码，必须假设线程在任何时刻都有可能被剥夺 CPU 资源的使用权。

例 8.4 yield()方法只会将 CPU 时间交给相同或更高优先级的线程

```
public class Example8_3 extends Thread {
    int number;
    public Example8_3(int number) {
        this.number = number;
    }
    public void run()
    {
        for (int index = 1; index <= 50; index++)
        {
            if ((index % 10) == 0)
            {
                System.out.println("线程#" + number + ": " + index);
                yield();
            }
        }
    }
    static public void main(String[] args)
    {
        Thread t1 = new Example8_3(1);
        Thread t2 = new Example8_3(2);
        t1.setPriority(2);
        t2.setPriority(8);
        t1.start();
        t2.start();
    }
}
```

程序运行结果：

```
线程#1：10
线程#2：10
线程#2：20
线程#2：30
线程#2：40
线程#2：50
```

```
线程#1：20
线程#1：30
线程#1：40
线程#1：50
```

多次运行例 8.3 后会发现，线程 1、2 都创建完成后，如果线程 2 一旦投入运行，尽管也会调用 yield()方法，但线程 1 并不会因此而开始运行。这是因为调用 yield()方法只会将 CPU 时间交给具有相同或更高优先级的线程，如果当前可运行线程只有较低优先级的线程，则 JVM 线程调度程序会将刚才调用 yield()方法的线程重新投入运行。上例中，线程 1 的优先级比优先级低，故不会因为线程 2 调用了 yield()方法而开始运行。

8.2.5 守护线程

守护线程（daemon thread）是一类特殊的线程，它与普通线程的惟一区别在于这个线程具有最低的优先级。用于为系统中的其它对象和线程提供服务。典型的守护线程例子是 JVM 中的系统资源自动回收线程，它始终在低级别的状态中运行，用于实时监控和管理系统中的可回收资源。例如，所谓单线程 Java 程序亦有一个垃圾收集线程伴随着主线程一起运行，但当主线程结束时整个 Java 程序也同时结束，这是因为垃圾收集线程被设计为一个守护线程。

一个用户自定义的线程也可标记为是否守护线程：调用 Thread 类的 isDaemon()方法可判断当前线程是否一个守护线程，调用 setDaemon()方法可将一个线程设置为是否守护线程。由一个守护线程创建的新线程默认为是一个守护线程。守护线程的典型应用是在后台为其它线程提供服务，例如执行内存垃圾收集、处理 GUI 事件分派、登记系统运行日志等。

例 8.5 创建若干简单的守护线程，演示正在执行的守护线程无法影响应用程序的终止

```
public class Example8_5 extends Thread {
    public Example8_5() {
        //setDaemon()方法必须在执行该线程的start()方法之前调用
        setDaemon(true);
        start();
    }
    public void run()
    {
        while (true) {
            try {
                sleep(500);
            } catch (InterruptedException exc) {
                throw new RuntimeException();
            }
            System.out.println(this);
        }
    }
    public static void main(String[] args)
    {
        System.out.println("创建简单的守护线程 ...");
        for (int i = 0; i < 3; i++) new Example8_5();
        try {
            Thread.currentThread().sleep(1 * 1000);
        } catch (Exception exc) {}
```

```
        System.out.println("主程序运行结束 ...");
    }
}
```

程序运行结果：

```
创建简单的守护线程 ...
Thread[Thread-1,5,main]
Thread[Thread-0,5,main]
Thread[Thread-2,5,main]
主程序运行结束 ...
Thread[Thread-1,5,main]
Thread[Thread-0,5,main]
Thread[Thread-2,5,main]
```

从该程序的运行效果可看出，一旦主线程结束整个 Java 应用程序就立即结束，表明这些正在运行的无限循环守护线程无法影响整个应用程序的退出。

8.3 线程的互斥

两个或多个线程对同一共享资源的并发访问可能带来问题，特别是当一个或多个线程对资源进行“写”操作时。比如一个工资管理负责人正在修改员工工资表时，如果同时容许员工领取工资，必然出现混乱。因此，工资管理负责人正在修改工资表时，将不允许任何员工领取工资，也就是说这些员工必须等待直到工资修改完毕，在这个例子中，负责人和员工对工资表的访问必须是互斥的访问。

所以，当一个多线程应用程序同时执行多个任务时，这些任务之间有时需要通信与同步。线程的多种阻塞状态为实现不同应用背景的线程通信与同步提供了有力支持。例如，多个线程互斥地访问共享资源可使用锁阻塞方式，而一个线程等待另一线程执行完毕才继续执行则可使用 join()阻塞。多线程应用程序设计的复杂性主要体现在对线程通信与同步的审慎处理。

8.3.1 问题的提出

两个或多个线程对同一共享资源的并发访问可能带来问题，特别是当一个或多个线程对资源进行“写”操作时。比如一个工资管理负责人正在修改员工工资表时，如果同时容许员工领取工资，必然出现混乱。因此，工资管理负责人正在修改工资表时，将不允许任何员工领取工资，也就是说这些员工必须等待直到工资修改完毕，在这个例子中，负责人和员工对工资表的访问必须是互斥的访问。

两个线程对同一共享资源的并发访问可能带来问题，特别是当一个或多个线程对资源进行“写”操作时。下面的程序可帮助加深了解应用程序中多个线程对共享资源的并发访问处理不当时带来的问题。

例 8.6 多线程对同一共享资源的并发访问（这是错误的方式的例子）

```
class Account{
    double balance;
    public Account (double money){
        balance = money;
        System.out.println("帐户总金额为: "+balance);
    }
}
public class Example8_6 extends Thread {
    Account account;
    int delay;
    public Example8_6(Account account,int delay) {
        this.account =account;
        this.delay = delay;
```

```
    }
    public void run(){
        if(account.balance >= 100){
            try{
                sleep(delay);
                account.balance = account.balance - 100;
                System.out.println(this.currentThread()+"取100元成功");
            }catch(InterruptedException e)  { }
        }
        else
            System.out.println("取款失败!");
        }
    public static void main(String[] args) {
        Account account = new Account(100);
        Example8_6 accountThread1 = new Example8_6(account,1000);
        Example8_6 accountThread2 = new Example8_6(account,0);
        accountThread1.start();
        accountThread2.start();
    }
}
```

程序运行结果：

```
帐户总金额为: 100.0
Thread[Thread-1,5,main] 取100元成功
Thread[Thread-0,5,main] 取 100 元成功
```

该结果非常奇怪，因为尽管账面上只有 100 元，但是两个取钱线程都取得了 100 元钱，也就是总共得到了 200 元钱。出错的原因在哪里呢？原来，由于线程 1 在判断满足取钱的条件后，被线程 2 打断，还没有来得及修改余额。因此线程 2 也满足取钱的条件，并完成了取钱动作。从而使共享数据 balance 的完整性被破坏。

为保证多个并发线程可互斥地使用共享资源，Java 语言参考操作系统的管程（monitor）机制，为每一对象实例都引入一个管程，并称之为“锁”。由保留字 synchronized 定义的同步代码段受锁的控制：在进入同步代码段之前，线程必须获取指定对象实例的锁，倘若无法获得锁则线程进入该对象实例的锁等待队列中等候；同步代码段执行完毕或异常中断时，线程归还自己占有的锁并唤醒那些在队列中等待该锁的线程。由于在任何时刻一个对象实例的锁仅能由一个线程持有，锁机制可保证线程是否有权限进入某一特定的代码段，因而锁的典型用法是控制临界区（critical section）的访问，从而保证特定代码段之间的互斥。

Java 语言提供了保留字 synchronized.其基本用法如下：

```
synchronized(互斥对象){
    临界代码段
}
```

下面的例子中有两个线程：会计和出纳，他们共同拥有一个账本。他们都可以使用存取方法对账本进行访问，会计使用存取方法时，向账本写入存钱记录；出纳使用存取方法时，向账本写入取钱记录。因此，当会计正在使用账本时，出纳被禁止使用，反之也是这样。例 8.7 模拟会计向账本存入 270 万元，但分三次写入，每次因各种原因间隔一段时间，出纳从帐本取出 90 万元，也分三次取出，每次同样因各种原因间隔一段时间，间隔时间用 sleep(2000)模拟，该例用 synchronized 保证了会计、出纳其中一人

在使用账本时，另一个人必须等待。

例 8.7 多线程对同一共享资源的并发访问（正确的方式）——存取钱中的互斥问题

```
class Bank implements Runnable {
    int money=100;
    String name;
    public void setMoney(int n) {
        money=n;
    }
    public synchronized void deposite(int number) { //存款
         //会计使用存入270，每次存入90，稍歇一下
         for(int i=1;i<=3;i++) {
             try {
                 Thread.sleep(2000);
                 //会计还没使用完存取方法前出纳不能使用
                 money=money+number;
                 System.out.println("存入"+number+"万,"+"帐上现有"+money+"万,休
息一会再存");
             } catch(InterruptedException e){}
         }
     }
    public synchronized  void withdraw(int number){ //取款
        //出纳使用取出90，每次取出30，稍歇一下
        for(int i=1;i<=3;i++) {
             try {
                 Thread.sleep(2000);
                 //出纳还没使用完存取方法前会计不能使用
                 money=money-number;
                 System.out.println("取出"+number+"万,"+"帐上有"+money+"万,休息
一会再取");
             } catch(InterruptedException e){}
         }
    }
    public void run() {
        name=Thread.currentThread().getName();
        if(name.equals("会计"))
            deposite(90);
        else if(name.equals("出纳"))
            withdraw(30);
    }
}
public class Example8_7  {
    public static void  main(String args[]) {
        Bank bank;
        Thread 会计,出纳;
```

```
            bank=new Bank();
            会计=new Thread(bank);
            出纳=new Thread(bank);
            会计.setName("会计");
            出纳.setName("出纳");
            bank.setMoney(100);
            System.out.println("银行帐户原有100万");
            会计.start();
            出纳.start();
        }
    }
```

程序运行结果是：

```
银行帐户原有100万
存入90万,帐上现有190万,休息一会再存
存入90万,帐上现有280万,休息一会再存
存入90万,帐上现有370万,休息一会再存
取出30万,帐上有340万,休息一会再取
取出30万,帐上有310万,休息一会再取
取出 30 万,帐上有 280 万,休息一会再取
```

例 8.7 中，如果去掉 deposite()和 withdraw()方法前的 synchronized 修饰符，则程序运行过程中可能得到下面的错误结果：

```
银行帐户原有100万
存入90万,帐上现有160万,休息一会再存
取出30万,帐上有160万,休息一会再取
存入90万,帐上现有250万,休息一会再存
取出30万,帐上有220万,休息一会再取
取出30万,帐上有190万,休息一会再取
存入 90 万,帐上现有 280 万,休息一会再存
```

8.4 线程的同步

在前面我们研究了共享资源的访问问题。在实际应用中，多个线程之间不仅需要互斥机制来保证对共享数据的完整性，而且有时需要多个线程之间互相协作，按照某种既定的步骤来共同完成任务。一个典型的应用是称之为生产-消费者模型。该模型可抽象为：

（1）生产者负责产品，并将其保存到仓库中；

（2）消费者从仓库中取得产品；

（3）由于库房容量有限，因此只有当库房还有空间时，生产者才可以将产品加入库房；否则只能等待。只有库房中存在满足数量的产品时，消费者才能取走产品，否则只能等待。

实际应用中的许多例子都可以归结为该模型。如在操作系统中的打印机调度问题，库房的管理问题等。wait()方法与 notify()/notifyAll()方法配合使用，可控制一个线程进入阻塞状态后又恢复为可运行状态，从而解决线程的同步问题。

wait()方法的作用是：当一个线程执行了该方法，则该线程进入阻塞状态，同时让出同步对象的互斥锁，并自动进入互斥对象的等待队列。

notify() / notifyAll()的作用是：当一个线程执行了该方法，则拥有该方法的互斥对象的等待队列中的第一个/所有线程被唤醒，同时自动获得该互斥对象的互斥锁，并进入就绪状态等待调度。

一个对象实例的 wait()方法仅能由当前占有该实例的锁的那个线程发出调用，调用结果导致该线程

放弃当前占有的锁并转入阻塞状态，即进入到该对象实例的 wait()队列中等候唤醒。这些线程如何被唤醒取决于调用 wait()方法时的参数：如果没有指定时间参数，则仅当原对象实例的 notify()或 notifyAll()方法被调用时才重返可运行状态；如果指定了时间参数，则当原对象实例的 notify()或 notifyAll()方法被调用时，或经过的时间已超过指定时间长度（毫秒）时，该线程将重返可运行状态。

为了研究生产者-消费者问题，我们仍然存款与取款问题作为例子，假设存在一个账户对象(仓库)及两个线程：存款线程(生产者)和取款线程(消费者)，并对其进行如下的限制：

（1）只有当账户上的余额 balance=0 时，存款线程才可以存进 100 元；否则只能等待；

（2）只有当账户上的余额 balance=100 时，取款线程才可以取走 100 元；否则只能等待。

例 8.8 给出了生产者-消费者问题的解决方案。

例 8.8 存取钱的同步问题

```
class Account2
{
    double balance;
    public Account2(int money)
    {
        balance = money;
        System.out.println("帐户总金额为: " + balance);
    }
    public synchronized void withdraw(double money)
    {
        if (balance == 0)
            try {
                System.out.println("帐户金额不足,等待生产者存入......");
                wait();
            } catch (InterruptedException e) {
        }
        balance = balance - money;
        System.out.println("消费者取款100元成功");
        notify();
    }
    public synchronized void deposite(double money)
    {
        if (balance != 0)
            try {
                System.out.println("帐户还有足够金额,不需存入,等待消费");
                wait();
            } catch (InterruptedException e) {
        }
        balance = balance + money;
        System.out.println("生产者存款100元成功");
        notify();
    }
}
public class Example8_8 extends Thread
```

```
{
    Account2 account;
    String name;
    public Example8_8(Account2 account)
    {
        this.account = account;
    }
    public void run(){       {
        name=Thread.currentThread().getName();
        if(name.equals("consumer"))
            account.withdraw(100);
        if(name.equals("producer"))
            account.deposite(100);
    }
    public static void main(String[] args)
    {
        Account2 acount = new Account2(0);  //设初始帐户金额为0
        Example8_8 consumer = new Example8_8(acount);
        Example8_8 producer = new Example8_8(acount);
        consumer.setName("consumer");
        producer.setName("producer");
        consumer.start();
        producer.start();
    }
}
```

程序运行结果是：

```
帐户总金额为: 0.0
帐户金额不足,等待生产者存入......
生产者存款100元成功
消费者取款 100 元成功
```

线程同步设计中还需考虑的一个重要因素是线程安全性与程序性能之间的折衷。一个同步方法调用的开销大约是一个普通方法调用的 6.6 倍，因此，如果你选择的同步粒度太大，则会降低应用程序中的并发程度，从而导致程序性能的下降；如果同步粒度太小，则又担心无法保护共享资源，导致程序的线程安全性被破坏。因而一个好的线程同步设计应该是仅仅将那些真正需要保护的共享数据访问代码存放在同步代码段中，无关线程安全性的那些代码则不宜放在同步代码段中。

8.5 线程联合

一个线程 A 在占有 CPU 资源期间，可以让其他线程调用 join()方法和本线程联合，如： B .join()，则称线程 A 在运行期间联合了线程 B，线程 A 则进入阻塞状态，如何恢复运行则取决于调用时的参数：如果没有指定时间参数，则仅当另一线程执行完毕时才重返可运行状态；如果指定了时间参数，则当另一线程执行完毕，或经过的时间已超过指定时间长度，均会导致线程重返可运行状态。join()方法有多种重载形式，主要的形式有以下两种：

```
public final synchronized void join(long millis) throws InterruptedException { ... }
public final void join() throws InterruptedException { ... }
```

join()方法非常适合一个线程需等待另一线程执行完毕后再继续执行的应用背景。例 8.9 演示了两对

线程，其中每对线程都有一个线程（Waiter）在等待另一线程（Sleeper）执行结束后才继续运行。线程 Sleeper#2 在睡眠期间被唤醒，因而 Thread.sleep()方法返回一个 InterruptedException 异常，并且导致等待该线程的另一线程 Waiter#A 可继续执行；而线程 Sleeper#1 则在指定时间（3 秒）后才被唤醒，等待该线程执行结束的另一线程 Waiter#B 最后才运行完毕。

例 8.9

```
class Sleeper extends Thread {
    public Sleeper(String name) {
        super(name);
        start();
    }
    public void run() {
        try {
            Thread.sleep(3 * 1000);
        } catch (InterruptedException exc) {
            System.out.println(this.getName() + "睡眠期间被吵醒");
            return;
        }
        System.out.println(this.getName() + "睡觉睡到自然醒");
    }
}
class Waiter extends Thread {
    private Sleeper sleeper;
    public Waiter(String name, Sleeper sleeper) {
        super(name);
        this.sleeper = sleeper;
        start();
    }
    public void run() {
        try {
            sleeper.join();
        } catch (InterruptedException exc) {}
        System.out.println("等待" + sleeper.getName() + "的" + this.getName() +
"继续 ...");
    }
}
public class Example8_9
{
    public static void main(String[] args) {
        Sleeper s1 = new Sleeper("Sleeper#1");
        Sleeper s2 = new Sleeper("Sleeper#2");
        Waiter w1 = new Waiter("Waiter#A", s1);
        Waiter w2 = new Waiter("Waiter#B", s2);
        s2.interrupt();
    }
```

```
}
```

程序运行结果：

```
Sleeper#2睡眠期间被吵醒
等待Sleeper#2的Waiter#B继续 ...
Sleeper#1睡觉睡到自然醒
等待 Sleeper#1 的 Waiter#A 继续 ...
```

小结

线程相对于进程而言是一种轻量级的计算任务，Java 线程机制建立在本地平台的线程库基础上，创建一个 Java 线程有两种途径：要么继承 Thread 类，要么实现 Runnable 接口。

每一个线程都有其生存期，线程的可运行状态与正运行状态之间的切换由 JVM 线程调度程序负责。

线程优先级用于影响调度程序对线程的选择，可通过设置优先级使重要的任务优先执行；守护线程是一类特殊的线程，其优先级最低。

线程的可运行状态与正运行状态之间的切换由 JVM 线程调度程序负责，也可通过调用 yield()方法主动放弃 CPU 时间。线程阻塞是实现线程之间通信与同步的基础，Java 语言为线程提供了多种阻塞机制，其中由 synchronized 标识的同步代码段与 wait()/notify()机制是最重要的两类线程阻塞形式。

一个线程 A 在占有 CPU 资源期间，可以让其他线程调用 join()方法和本线程联合，如：Ｂ.join()，则称线程 A 在运行期间联合了线程Ｂ，线程 A 则进入阻塞状态。

习题

一、基本概念

1、线程与进程的主要区别是什么？为什么说线程是一种轻量级的计算任务？试列举两个适合多线程设计的应用背景。

2、为保证我们编写的多线程应用程序在不同平台之间的可移植性，在使用 Java 语言的线程机制时应避免使用哪些可能会影响程序可移植性的特性？

3、有几种方式创建线程？

4、Runnable 接口中包括哪些抽象方法？Thread 类有哪些主要成员变量和成员方法？

5、线程有哪几种基本状态？描述各状态之间进行相互转换的条件及实现方法。

6、阐述通过继承 Thread 类和通过实现 Runnable 接口来实现多线程的步骤有何不同？

7、用于控制线程的常用方法有哪些？

8、守护线程的优先级一定比普通线程的优先级低吗？试举例说明我们应将哪些线程设计为守护线程。

9、如何实现线程的同步与互斥。

二、编程实践

1、创建两个线程，每个线程打印出线程名字后在睡眠，给其他线程以执行的机会。主线程也要打印出名字后再睡眠。每个线程前后共睡眠 5 次，要求分别采用从 Thread 中继承和实现 Runnable 接口两种方式来实现。

2、编写一个 Java 应用程序，在主线程中创建 3 个线程：Counter、Printer 和 Storage。其中，Counter 线程的主要作用是从 0 开始计数，并将产生的每个数存放到 Storage 线程中。Printer 线程的作用是不断读取 Storage 线程中存放的数并打印出来，使用同步机制，确保每个只能被打印一次。

3、编写一个程序，创建两个线程模拟会计和出纳。两个线程共享变量 money。当 money 的值小于 150 时，会计线程会结束自己的运行；当 money 的值小于 0 时，出纳线程也结束自己的运行。

4、试模仿本章的生产者／消费者问题例子程序，编写一个基于消息队列的股票交易价格显示程序。该程序中的一个线程模拟从卫星通信、微波通信或股票信息发射台等不同途径获取最新的股票交易价格信息，并将价格数据写入一个消息队列中；程序中的另一个线程负责从消息队列中取出价格数据。（提示：设计本程序的关键是处理好消息队列的同步问题；以后还可考虑使用输入／输出流或 JDBC 实现消息队列的持久性。）

第 9 章 图形用户界面

用户界面(User Interface)是计算机的使用者(用户)与计算机系统交互的接口，用户界面的功能是否完善、使用是否方便，直接影响着用户对应用软件的使用。因此，设计和构造用户界面，是软件开发中的一项重要工作。

图形用户界面(Graphics User Interface，简称 GUI)就是为应用程序提供的一个图形化的界面。GUI 使用图形的方式，借助菜单、按钮等标准界面元素和鼠标操作，帮助用户方便地向计算机系统发出命令、启动操作，并将系统运行的结果同样以图形的方式显示给用户，使一个应用程序具有画面生动、操作简便的效果，省去了字符界面中用户必须记忆各种命令的麻烦，深受广大用户的喜爱和欢迎。GUI 已经成为目前几乎所有应用软件的既成标准。

9.1 Java 程序的图形用户界面

Java 程序可使用多种用户界面，包括这一章之前所使用的命令行方式，以及图形用户界面方式、Web 浏览器方式和小程序(applet)界面方式等。

9.1.1 Java 图形用户界面的组成

Java 应用程序的图形用户界面使用 Java API 提供的 java.awt 或 javax.swing 包中的组件实现。这些组件所构成的 GUI 系统通常包括以下几个部分：

（1）一些最基本的图形用户界面控件，如菜单、按钮、文本字段等，这些控件展示了系统可用的操作，用户通过使用鼠标或者键盘控制这些控件，激发系统完成相应的功能；

（2）容器组件，如窗口、面板等，这些容器组件容纳基本控件，将它们按照一定层次和空间结构组合在一起；

（3）布局管理组件，负责容器中组件的布局，使程序员可进一步美化图形用户界面；

（4）事件传播与处理模式，用户使用鼠标或者键盘控制用户界面的控件时在图形用户界面程序内部激发了事件，这些事件在图形用户界面程序内进行传播，由一些特定的图形用户界面组件监听并处理。GUI 系统必须确定这种事件的传播和处理模式，以简化图形用户界面程序的设计和实现。

（5）一些支持高级特性的控件，例如用于支持不同观感风格的控件。目前流行的计算机操作系统平台主要有 Windows、Macintosh、Unix 等，不同的操作系统平台上的图形用户界面的按钮、菜单等控件往往具有不同的外观和视觉效果。跨平台的应用程序需要支持不同平台上不同的观感风格。

9.1.2 AWT 和 Swing 组件

java.awt 或 javax.swing 包中的组件分别被称为 AWT 组件或 Swing 组件，AWT(Abstract Window Toolkit，抽象窗口工具箱)是 JDK 1.0 和 JDK1.1 用于支持编写 GUI 程序的组件，Swing 组件在 JDK1.1 中作为一种可选的插件，但在 JDK1.2 版以后作为 Java API 的核心部分组件用于支持 GUI 程序的开发。

Java API 提供 AWT 组件的初衷是支持编写跨平台的图形用户界面程序，使得在不同的操作系统平台可以具有相同的界面观感。但实际上，AWT 组件没能完全达到这一目标，许多 AWT 组件的源代码中包含有依赖特定平台的部分，使得基于 AWT 组件的用户界面程序在不同的平台仍具有不同的观感。因此 AWT 组件被称为重量级的 GUI 组件。

Swing 组件是在 AWT 组件基础上发展起来的新型 GUI 组件。与 AWT 组件相比，Swing 组件是轻量级 GUI 组件，完全由纯 Java 代码编写，不含任何依赖特定平台的代码，因此具有更高的平台无关性、更好的移植性。Swing 完善了 GUI 组件的功能，提供了滚动窗格、表格、树等新的组件，同时也为某些 GUI 控件，如按钮等提供了更多的特性，使得编写 GUI 程序更为方便和高效。

Swing 是构成 Java Foundation Classes（Java 基类，简称 JFC）的核心部分，采用 JFC 能够开发界面更加丰富的 GUI 程序。Swing 包括容器、原子控件等一些 Swing 组件和布局管理器、用作事件传播和处理的事件类和监听器接口，以及用于界面管理的辅助控件。Swing 组件是组成 GUI 的一些可见或不可见的区域，这些区域提供视觉效果及交互途径。事件类、实现监听器接口的用户类，以及运行于 Java 虚拟机内部的事件分发线程构成了 Swing 的事件处理系统，真正完成与用户的交互功能。

无论是 Swing 组件还是 AWT 组件，它们作为 GUI 系统都包含上面所说的几个基本组成部分，而且 Swing 组件在 AWT 组件基础上开发，具有 AWT 组件所具备的一切特性。Swing 组件与 AWT 组件也具有很好地对应关系，学习了 Swing 组件的用法也可容易地学会 AWT 组件的用法，因此这一章主要介绍 Swing 组件的用法。

注意，鉴于 Swing 组件具有更好的平台无关性，我们推荐程序员在编写 GUI 程序时使用 Swing 组件。不过由于目前有许多 Web 浏览器内嵌的 Java 虚拟机仍不支持 Swing，所以在编写某些 Applet 程序时可能仍需要使用 AWT 组件。

在 Java 中，设计用户界面需要经历下面 4 个基本步骤：

（1）创建和设置组件。在 Java 语言中，每一种组件都有相应的标准支持。所谓创建组件就是声明一个相应组件类的对象引用，并利用不着 new 运算符创建对象，随后可以通过调用组件类提供的成员方法设置组件的属性。

（2）将组件加入到容器中。在 Java 的图形用户界面中，所有的组件必须放在容器中，每种容器都提供了 add()成员方法，用于将某个组件添加到容器中。

（3）布局组件。Java 是编写网络应用程序的首选工具。由于运行网络程序时无法完全限制运行环境，所以显示方式也很难得知。为了保证能够将程序的用户界面完整地显示出来，在 Java 语言中，提出了布局管理器的概念，用户可以根据不同的需求，选择不同的布局管理器。布局管理器可以自动地布局放在容器中的每个组件的显示位置，控制其大小。当然，也可以不使用布局管理器，完全由用用户自行控制，但这样很难保证该应用程序在任何环境下都能将全部内容显示出来。

（4）处理由组件产生的事件。上面 3 个步骤完成了应用程序的用户界面外观设计，但没有任何操作行为。如果希望应用程序能够对用户通过交互界面发出的操作命令给予响应，就需要实现事件处理。

9.1.3 Java 的事件处理模式

Java 采用委托事件模型来处理事件。委托事件模型的特点是将事件的处理委托给独立的对象，而不是组件本身，从而将使用者界面与程序逻辑分开。整个“委托事件模型”由产生事件的对象（事件源）、事件对象本身以及监听器对象之间的关系所组成。

产生事件的对象——事件源（event source）会在事件产生时，将与该事件相关的信息封装在一个称之为“事件”（event）的对象中，并将该对象传递给事件监听器（event listener）对象，监听器对象根据该事件对象内的信息决定适当的处理方式。监听器对象要收到事件发生的通知，必须在程序代码中向产生事件的对象注册，当事件产生时，产生事件的对象就会主动通知监听器对象，监听器对象就可以根据产生该事件的对象来决定处理事件的方法。

例如，假设用户单击了应用程序界面中的一个按钮，这个按钮就是事件源，该事件由一个事件对象标识，并与被单击按钮的对象关联。这里的事件对象是一个属于 ActionEvent 类型的对象，其中包含了有关事件和事件源的信息。这个对象将作为参数传递给处理该事件监听器的成员方法。

Java 事件处理机制的一个重要特点是：事件处理不是由产生事件的类对象本身完成，而是委托另外一个称之为监听器的类对象专门负责事件处理，这样可以防止任务过于集中，易于规范事件处理的过程。监听器既可以由产生事件的类实现，也可以由其他类实现，甚至可以由内部类或匿名类实现。监听器中含有处理相应事件的成员方法，这些成员方法是对相应的监听接口中声明的成员方法的具体实现。也就是说，要想处理某种事件，必须创建某种事件的监听器。监听器类是一个实现某种监听接口的类。不同的事件类拥有不同的监听接口，每个监听接口中声明了处理这类事件的抽象方法。这种事件处理方式成为“委托”模式，即委托其他类对象处理事件。

一个事件监听器能注册到多个事件源，以便处理多个事件源所产生的相同类型的事件，事件监听器的事件处理程序可通过事件对象所携带的事件源信息区分某一时刻产生事件的事件源。一个事件源也可注册多个事件监听器，以处理多种类型的事件，或者对同一类型的事件做几种不同的处理。

Swing 和 AWT 定义了各种类型的事件，每一种事件有相应的事件监听器接口描述处理该事件应该实现的基本行为。通常若事件类名为 XxxEvent，则事件监听器接口命名为 XxxListener，而组件（事件

源)注册实现该事件监听器接口对象的方法则命名为 addXxxListener。例如,动作事件类名为 ActionEvent，动作事件监听器接口是 ActionListener，组件中添加该种监听器对象的方法为 addActionListener()。

综上所述，Java 的事件处理模式是，事件源注册监听器，事件监听器监听事件对象并对事件进行处理。为了能处理事件源产生的事件，首先需要使用 addXxxListener()方法为事件源注册相应的监听器，其次应该设计类实现相应的事件监听器接口。可设计一个类直接实现该监听器接口，但当监听器接口有多个方法需要实现，而程序又只需要其中一两个方法进行事件的处理时，则程序必须为其他方法提供冗余的空方法体。因此，为了方便处理，Swing 为某些具有多个方法的监听器接口提供了事件适配器类，这种类通常命名为 XxxAdapter，它以空方法体实现该监听器接口，可设计类继承事件适配器类，从而减少一些编程工作量。

AWT 和 Swing 定义的事件种类很多，后面在讨论 Swing 组件的用法时将给出该组件通常应监听的事件类型，并配合例子说明这种事件类型的基本处理方法。

9.2 Swing 容器

容器是用来放置其他组件的一种特殊部件，其中顶层容器和中间容器是常用的两类形式。下面分别介绍这些容器的主要用途及使用方式。

9.2.1 顶层容器

每一个应用 Swing 组件的应用程序都至少要有一个顶层容器。Swing 顶层容器提供最基本的窗口放置其它容器和组件，应用程序必须创建一个顶层容器之后才能放置其它 GUI 容器或组件。画框（JFrame）和对话框（JDialog）是常用的顶层容器，类 JApplet 用于小应用程序的编写。

1. JFrame

JFrame 是最常用的一种顶层容器，它的作用是创建一个顶层的 Windows 窗体。JFrame 的外观就像平常 windows 系统下见到的窗体，有标题、边框、菜单等等。

JFrame 类提供两种格式的构造方法：

JFrame(); // 不指定标题

JFrame(String title); // 指定标题

JFrame 类的其他常用方法见表 9-1。

表 9-1 JFrame 类的部分成员方法

成员方法	描述
void setTitle(String title)	设置窗体标题
String getTitle()	获取窗体标题
void pack()	根据窗口中组件的首选大小，将窗口整理到最合适的大小
void setDefaultCloseOperation(int op)	设置关闭窗口时所做的操作，op 可取的值包括： HIDE_ON_CLOSE: 关闭窗口时隐藏 ； EXIT_ON_CLOSE: 关闭窗口时退出应用程序，不可用于 Applet； DISPOSE_ON_CLOSE: 关闭窗口时隐藏窗口并释放资源； NOTHING_ON_CLOSE: 关闭时不做任何事情，通常配合监听器使用。
void setContentPane(Container contentPane)	设置窗口的内容窗格
Container getContentPane()	返回窗口的内容窗格
void setJMenuBar(JMenuBar menuBar)	设置窗口的菜单栏
JMenuBar getJMenuBar()	返回窗口的菜单栏
void setSize(int width,int height)	将窗口设置为宽 width，高 height
void setBounds(int xleft,int yleft,int width,int height)	将窗口的左上角位置设置为(xleft,yleft)，宽 width，高 height

在使用顶层容器时，需要注意以下几点：

- 为了能够在屏幕上显示，每个 GUI 组件必须位于一个容器中。
- 每个 GUI 组件只能添加到一个容器中。如果一个组件已经添加到一个容器中，又把它添加到另外一个容器中，则它将从第一个容器中删除，然后再移入第二个容器。
- 每个顶层容器都包含一个内容窗格（Content Panel），所有的可视组件都必须放在内容窗格中。可以调用顶层容器中的 getContentPane()方法得到当前容器的内容窗格，并使用 add()方法将组件添加到其中。
- 可以在顶层容器中添加菜单栏，它将位于顶层容器的约定位置，例如，在 Window 环境下，菜单栏位于窗口标题栏的下面。

例 9.1 创建与使用 JFrame

```
import java.awt.*;
import javax.swing.*;
public class Example9_1 {
    public static void main(String[] args) {
        //创建JFrame对象
        JFrame frame = new JFrame("JFrame演示");
        frame.setSize(300,200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // 在内容面板放置一背景为黄色的标签
        JLabel yellowLabel = new JLabel("JFrame演示");
        yellowLabel.setOpaque(true);
        yellowLabel.setBackground(Color.yellow);
        yellowLabel.setPreferredSize(new Dimension(200, 180));
        //将标签加入容器
        frame.getContentPane().add(yellowLabel, BorderLayout.CENTER);
        // 在内容面板放置一背景为蓝绿色的菜单条
        JMenuBar cyanMenuBar = new JMenuBar();
        cyanMenuBar.setOpaque(true);
        cyanMenuBar.setBackground(Color.cyan);
        cyanMenuBar.setPreferredSize(new Dimension(200, 20));
        frame.setJMenuBar(cyanMenuBar);
        frame.pack();
        frame.setVisible(true);
    }
}
```

运行这个程序后，将会在屏幕上看到如图 9_1 所示的结果。

图 9_1 例 9.1 运行结果

2. JDialog

对话框(JDialog)也是顶层容器，其特点是必须依赖于另一窗口（通常是 JFrame）。应用程序主要使用模态对话框，模态对话框常用来给予用户简单的信息提示或选择，它的出现会阻塞其它组件对事件的响应。创建模态对话框常使用类 JOptionPane 而不是 JDialog，类 JOptionPane 内部会自动创建一个 JDialog，并将自己添加到该对话框的内容窗格中，图 9_2 是使用类 JOptionPane 的方法 showOptionDialog()创建的对话框。

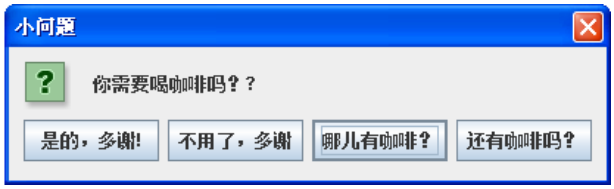

图 9_2 对话框演示

由 JOptionPane 提供的对话框具有较好的可定制特性，包括：

1. 允许指定不同的标题与消息文本；
2. 支持使用不同的提示图标；
3. 允许指定不同的按钮标签文字；
4. 允许指定几种不同的按钮模式（Yes、No、Cancel 等）。

这些定制特性以重载方法（使用不同的参数列表）实现，而且大多数操作都被定义为类方法，使用时非常方便（不必创建对象实例）。其中功能最强大的是 showOptionDialog()方法，例如，showMessageDialog()方法和 showConfirmDialog()方法等都通过调用该方法实现。该方法接口是：

```
int showOptionDialog(Component parentComponent, Object message,
                        String title, int optionType,int messageType,
                        Icon icon, Object[] options, Object initialValue)
```

其中参数的含义是（下面的常量都定义在类 JOptionDialog 中）：

parentComponent：包含对话框的父组件，该对话框依赖于该组件

message：通常是一个字符串，用于显示提示信息

title：对话框的标题

optionType：选项类型，决定按钮的缺省个数与按钮的文字，可取的值包括：

DEFAULT_OPTION：缺省类型，通常是只含 OK（确定）按钮

YES_NO_OPTION：含有 YES（是）和 NO（否）两个按钮

YES_NO_CANCEL_OPTION：含有 YES、NO 和 CANCEL（取消）三个按钮

OK_CANCEL_OPTION：含有 OK 和 CANCEL 两个按钮

messageType：信息类型，不同的信息有不同的缺省图标，可取的值包括：

ERROR_MESSAGE：错误信息

INFORMATION_MESSAGE：提示细心

WARNING_MESSAGE：警告信息
QUESTION_MESSAGE：提问信息
PLAIN_MESSAGE：普通信息

icon：图标，除使用缺省的图标外，也可用自己指定的图标

options：可选项，可用来指定按钮上面显示的内容

initialValue：可选，初始的选项，options 中的值

例 9.2 创建图 9.2 中的对话框

```
import javax.swing.*;
public class Example9_2 {
    public static void main(String[] args) {
        JFrame frame = new JFrame("对话框演示");
        Object[] options = {"是的，多谢!", "不用了，多谢", "哪儿有咖啡？", "还有咖啡吗？"};
        int n = JOptionPane.showOptionDialog(frame,"你需要喝咖啡吗？？","小问题",
                JOptionPane.YES_NO_OPTION, // 决定有几个按钮
                JOptionPane.QUESTION_MESSAGE, // 决定使用什么图标
                null, // 不使用定制图标
                options, // 按钮的文字
                options[2]); // 初始选中的按钮（用户直接按回车键选择的按钮）
        if (n >= 0 && n < options.length)
            JOptionPane.showMessageDialog(frame, "你的选择是: "+options[n],
                    "选择提示", JOptionPane.PLAIN_MESSAGE);
        else JOptionPane.showMessageDialog(frame, "你没有作选择!", "选择提示",
                JOptionPane.PLAIN_MESSAGE);
        System.exit(1);
    }
}
```

上面的例9.2在用户按下某个按钮后，会弹出一个消息对话框显示用户的选择。消息对话框是Swing定义的标准模态对话框，使用showMessageDialog()方法弹出，只有一个OK（确定）的按钮，可指定标题、消息串和图标（图标可标识消息串的类型），该方法接口为：

```
void showMessageDialog(Component parentComponent, Object message,
                       String title,int messageType, Icon icon)
```

例如，下述语句可弹出图 9_3 的消息对话框：

```
JOptionPane.showMessageDialog(frame,
                "喝咖啡的时间到了!", "提示", JOptionPane.WARNING_MESSAGE);
```

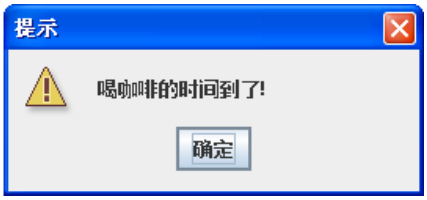

图 9_3 消息对话框演示

除消息对话框外，Swing 还定义了确认对话框和输入对话框。确认对话框使用showConfirmDialog()方法弹出，可指定按钮选项，标题、消息串、消息类型等。确认对话框通常用于让用户确认某些信息：

```
int showConfirmDialog(Component parentComponent, Object message,
                      String title,int optionType, int messageType, Icon icon)
```

该方法返回用户的选项，返回的值可取：

YES_OPTION：用户按了YES（是）按钮

NO_OPTION：用户按了NO（否）按钮

CANCEL_OPTION：用户按了CANCEL（取消）按钮

OK_OPTION：用户按了OK（确认）按钮

CLOSED_OPTION：用户直接关闭了对话框

例如下述语句可弹出图9_4的对话框：

```
int n = JOptionPane.showConfirmDialog(frame, "大家是否去喝咖啡呢？",
                    "询问", JOptionPane.YES_NO_CANCEL_OPTION, // 决定使用什么按钮
                    JOptionPane.QUESTION_MESSAGE); // 决定使用什么图标
```

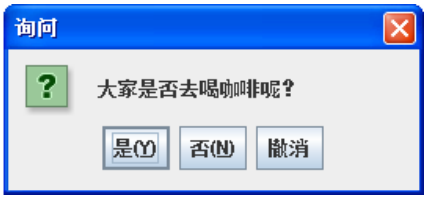

图9_4 确认对话框演示

输入对话框用showInputDialog()方法弹出，可让用户输入简单信息。showInputDialog()方法可指定标题、提示信息串、提示信息类型、输入的可选值及初始的选择值。该方法返回用户选择的对象：

```
Object showInputDialog(Component parentComponent, Object message,
                       String title, int messageType,Icon icon,
                       Object[] selectionValues, Object initialSelectionValue)
```

下面的方法是上述方法的重载方法，它让用户输入字符串：

```
String showInputDialog(Component parentComponent, Object message,
                                                  String title, int messageType)
```

例如下述语句可弹出图9_5的输入对话框：

```
String name = JOptionPane.showInputDialog(frame, "我们要去的咖啡厅是：",
                                  "输入咖啡厅名字", JOptionPane.QUESTION_MESSAGE);
```

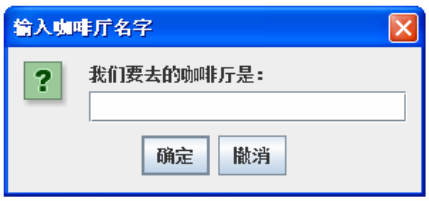

图 9_5 输入对话框演示

9.2.2. 中间容器

Swing 的中间容器包含了一些可以被使用在许多不同环境下的中间层容器。主要包括面板容器（类 JPanel）、分隔窗格（类 JSplitPane）、带滚动条的窗格容器（类 JScrollPane）、工具栏（类 JToolBar）等。中，它们都是 JComponent 的子类，通常放在其他容器类中。这里介绍具有代表性的面板容器。

面板容器时一种常用的容器种类。在默认情况下，除背景外不会自行绘制任何内容。当然，可以利用相应的成员方法方便地为它添加边框，或定制想要获知的内容。

在默认情况下，面板容器不透明，可以调用 setOpaque()方法将其设置为透明。如果面板容器是透明的，将没有背景，这样可以让位于该容器覆盖区域下面的组件显示出来。

JPanel 的默认布局管理器是 FlowLayout。所谓布局管理器是指能够布局容器中每个组件所放位置大

小的部件，不同的布局管理器对应不同的组件布局策略。可以在创建容器时制订布局管理器，或调用setLayout()成员方法更改布局管理器。另外，还可以直接调用 add()成员方法将组件放置到面板容器中。

在 Jpanell 类中提供了下面两种格式的构造方法。

JPanel()　这是无参数的构造方法，它将创建一个布局管理器为 FlowLayout 的面板容器。

JPanel（LayoutManager layout）　这个构造方法创造一个布局管理器为 layout 的面板容器。

除此之外，JPanel 类还提供了表 9-2 列出的有关管理逐渐的成员方法。

表 9-2　JPanel 类的部分成员方法

成员方法	描　述
void add(Component comp)	将组件 comp 添加到容器中
void add(Component comp ,int index)	将组件 comp 添加到容器中，其编号为 index,容器中每一个组件的编号为 0，第二个组件编号为 1，依此类推
int getComponentCount()	返回容器中的组件数量
Component getComponent(Point point	返回位于 point 坐标点的组件对象
void remove(Component comp)	从容器中移出 comp 组件

例 9.3 JPanel 示例

```
import javax.swing.*;
public class Example9_3 {
    public static void main(String[] args)
    {
        JFrame f = new JFrame("JPanel容器示例");
        JLabel label = new JLabel("这是JPanel容器示例");
        JTextField field = new JTextField(10);
        //生成JPanel实例
        JPanel cPanel = new JPanel();
        //指定JPanel的边界及大小
        cPanel.setBorder(BorderFactory.createEmptyBorder(20,20,20,20));
        //将label和field加入JPanel容器中
        cPanel.add(label);
        cPanel.add(field);
        //将JPanel容器加入顶层容器中
        f.setContentPane(cPanel);
        f.pack();
        f.setVisible(true);
    }
}
```

运行结果如图 9_6 所示。

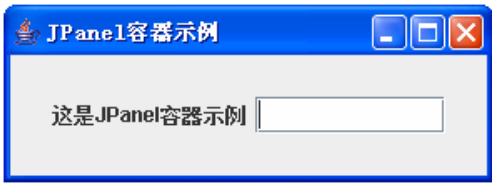

图 9_6　例 9.3 运行结果

面板容器最大的用途是将一个容器中的全部组件划分为如干组，每个组用一个面板作为容器，其中

设置若干个组件，这样就可以将顶层容器结构化，便于组件的布局和管理

9.3 布局管理器

前面讲述了 Swing 容器，以及将组件放入容器中的基本方法。但如何布局容器中各组件的位置还有一个未解决的问题。本节将详细阐述 Java 布局组件的基本策略。

首先，Swing 提供了 setLocation()，setSize()，setBounds() 等布局方法，但 Swing 的组件中存在一个默认的布局管理器，因此这些设置方法都会失效。如果需要设置组件大小或位置，则应取消该容器的布局管理器，方法为调用容器的 setLayout 方法，并将布局管理器设置为 null。

例 9.4 无布局管理器

```
import javax.swing.*;
public class Example9_4 extends JFrame {
    private JButton button = new JButton("JButton");;
    private JTextField textField = new JTextField("JTextField");
    public Example9_4() {
        setSize(300, 200);
        setLocation(400, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // 设置布局管理为null
        setLayout(null);
        // 设置按键的位置为(20,20)，宽100，高20
        button.setLocation(20, 20);
        button.setSize(100, 20);
        add(button);
        // 设置输入框的位置为(20,50)，宽200，高100
        textField.setBounds(20, 50, 200, 100);
        add(textField);
    }
    public static void main(String[] args) {
        Example9_4 frame = new Example9_4();
        frame.setVisible(true);
    }
}
```

在 Windows XP 上运行结果如图 9_7 所示。

图 9_7 例 9.4 运行结果

相对于基于布局管理器的布局方式在对组件的大小和位置的控制上较为灵活，但这种布局方式会导致平台相关，在不同的平台上可能产生不同的显示效果。如果想让 GUI 程序以一致的外观在不同的平台上运行，则需要采用基于布局管理器的布局方式。

Java 提供的布局管理器的类或接口都包含在 java.awt 包中，其中 LayoutManager 和 LayoutManager2 是两个接口，且 LayoutManager2 是 LayoutManager 的子接口，用于布局管理器的 FlowLayout、BorderLayout、GridLayout 这 3 个类实现了上述两个接口的类，利用它们可以控制组件在容器中的布局情况。

在表 9-3 中列出了 LayoutManager 接口中声明的部分有关布局管理器操作的抽象成员方法。

表 9-3 中 LayoutManager 接口中声明的部分成员方法

成员方法	描　述
void addLayoutComponent(String name,Component cp)	组件 cp 添加到布局管理器中，并用 name 与之关联
void removeLayoutComponent(Component cp)	从布局管理中删除组件 cp
Dimension preferredLayoutSize(Container parent)	返回给定容器的最佳尺寸
Dimension minimumLayoutSize(Container parent)	返回给定容器的最小尺寸

在 LayoutManager2 接口中，除了继承 LayoutManager 接口中的所有成员方法外，还补充声明了一些可以更加精确地控制组件布局的抽象方法。

根据不同的需要可以选择不同的布局方式。下面风别介绍 FlowLayout、BorderLayout 和 GridLayout 布局管理器的特点及使用方式。

9.3.1 FlowLayout 布局管理器

FlowLayout 是 JPane 容器的默认布局管理器。它按照从上到下，从左到右（或从右向左）的规则将添加到容器中的组件依次排序。如果一行中没有足够的空间放下一个组件，FlowLayout 换行后就将这个组件放到新的一行上。另外，在创建 FlowLayout 的队形是可以指定一行中组件的对齐方式。默认为居中，还可以指定每个组件之间的水平和垂直方向的间距大小，默认值为 5 个像素。这种布局管理器并不调整每个组件的大小，而是永远保持每个组件的最佳尺寸，剩余空间用空格补填。

在 FlowLayout 类中，提供了下面 3 种格式的构造方法：

◆ FlowLayout（）

这是无参数的构造方法。它将创建一个对齐方式为居中，水平和垂直方向间距为 5 个像素的布局管理器对象。

◆ FlowLayout（int align）

这个构造方法将创建一个对齐方式为 align 的布局管理器对象。align 是在 FlowLayout 类定义的常量 LEFT（居左）、RIGHT（居右）、CENTER（居中）、LEADING（沿容器左侧对其）和 TRAILING（沿容器右侧对其）中的一个。

◆ FlowLayout（ing align,int hgap,int vgap）

这个构造方法将创建一个对齐方式为 align、水平间距为 hgap 个像素、垂直间距为 vgap 个像素的布局管理器对象。

另外，在 FlowLayout 类中，除了实现 LayoutManager 接口的所有成员方法外，还提供了几个用于获取和设置对齐方式、组件间距等属性的成员方法。表 9-4 列出了其中的部分成员方法。

表 9-4 FlowLayout 类中的部分成员方法

成员方法	描　述
int getAlignment()	返回布局管理器的对齐方式。0—居左；1—居右；2—居中；3—沿容器左侧对齐；4—沿容器右侧对齐
void setAlignment(int align)	设置布局管理器的对齐方式为 align
int getHgap()	以像素为单位返回组件之间的水平间距。
void setHgap(int hgap)	将组件之间的水平间距设为 hgap 个象素
int getVgap()	以像素为单位返回组件之间的垂直间距。

void setVgap(int vgap)	将组件之间的垂直间距设为 vgap 个象素

例 9.5 FlowLayout 布局管理器

```
import java.awt.FlowLayout;
import javax.swing.*;
public class Example9_5 extends JFrame {
    private JButton button1 = new JButton("First Button");
    private JButton button2 = new JButton("Second Button");
    private JButton button3 = new JButton("Third Button");
    private JButton button4 = new JButton("Fourth Button");
    public Example9_5() {
        super("FlowLayout示例");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // 设置布局方式为FlowLayout
        getContentPane().setLayout(new FlowLayout());
        // 添加按键
        add(button1);
        add(button2);
        add(button3);
        add(button4);
        setSize(300, 130);  //设置窗口为300*130
        setLocation(400, 400);
    }
    public static void main(String arg[]) {
        Example9_5 frame = new Example9_5();//创建顶层窗口对象
        frame.setVisible(true); //显示窗口
    }
}
```

图 9_8 FlowLayout 布局管理器示例

在例 9.5 中，Example9_5 是 JFrame 的子类，因此是一个 JFrame 类型的顶层容器，利用 getContentPane() 方法可以获取它的内容同窗格，再利用 setLayout()方法可以将内容窗格的布局管理器设置为 FlowLayout，并将 4 个按钮添加到其中，它们的对齐方式为默认居中对齐，按钮之间的间距为 5 个像素。运行这个程序后将会在屏幕上看到如图 9_8 所示的结果。我们也可以试着改变窗口的大小，观察窗口中按键位置的变化情况。

9.3.2 BorderLayout 布局管理器

BorderLayout 是 JFrame 和 JDialog 两种容器的默认布局管理器，它将容器分为 5 个部分，分别命名为 North、South、West、和 Center。如图 9.9 所示。

由图 9.9 可以看出，在 5 个部分中有 4 个部分位于容器的 4 个周边，1 个位于中间。在使用这种布局管理器管理组件的排列时，需要为组件指明放的具体位置，默认位置为中间位置。如果将组件位置在 North 或 South,组件的宽度将延长至于容器一样宽，而高度不变。如果将组件放在 West 或 East，组件的

高度将延长至容器的高度减去 North 和 South 之后的高度，而宽度不变。不一定所有的区域都有组件，如果四周的区域（West、East、North、South 区域）没有组件，则由 Center 区域去补充。如果将两个组件放在同一位置。后面放置的组件将覆盖前面放置的组件。

BorderLayout 类提供了两种格式的构造方法：

◆ BorderLayout()

这是无参数的构造方法，它将创建一个组件之间水平和垂直间距均为零的布局管理器对象。

◆ BorderLayout(ing hgap,int vgap)

这个构造方法将创建一个组件之间水平竖决为 hagp,垂直间距为 vgap 的布局管理器对象。

另外，除了 BorderLayout 实现了 LayoutManager2 接口的全部方法外，还提供类似 FlowLayout 类中有关获取和设置组件间距的成员方法。

例 9.6 BorderLayout 布局管理器

```
import java.awt.BorderLayout;
import javax.swing.*;
public class Example9_6 extends JFrame {
    private JButton north = new JButton("North");
    private JButton south = new JButton("South");
    private JButton east = new JButton("East");
    private JButton west = new JButton("West");
    private JButton center = new JButton("Center");
    public Example9_6() {
        super("BorderLayout示例");
        setSize(300, 300);
        setLocation(400, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // 设置布局方式为BorderLayout
        getContentPane().setLayout(new BorderLayout());
        // 添加按键
        getContentPane().add(north, BorderLayout.NORTH);
        getContentPane().add(south, BorderLayout.SOUTH);
        getContentPane().add(east, BorderLayout.EAST);
        getContentPane().add(west, BorderLayout.WEST);
        getContentPane().add(center, BorderLayout.CENTER);
    }
    public static void main(String arg[]) {
        Example9_6 frame = new Example9_6();
        frame.setVisible(true);
    }
}
```

在这个程序中，创建了 5 个按钮，并将这 5 个按钮放在顶层窗口的内容窗格中。由于内容窗格的默认布局管理器为 BorderLayout，所以在放置按钮时需要给出具体的位置。运行这个程序后将会带屏幕上看到如图 9_9 所示的结果。

图 9_9 BorderLayout 布局管理器示例

9.3.3 GridLayout 布局管理器

GridLayout 布局管理器将整个容器划分成 N 行 M 列的网格平均占据容器的空间。布局时，按照组件加入的顺序优先考虑按行布局，当一行布局满之后再布局下一行（每行只能布局 M 个组件）。只有当行列不能满足指定的数值时（N×M 小于组件个数时），才按行扩展。

可以有两种将组件放入容器的方法：一是使用默认的布局顺序，即按照从上到下，从左到右的次序将组件放入容器的每个网格中：二是采用 add(Component comp,int index)方法将组件放入指定的网格中。

GridLayout 类提供了三种格式的构造方法：

GridLayout()

这是无参数的构造方法。它将创建一个一行内放置所有组件的网格布局管理器对象，组件之间没有间隙。

GridLayout（int rows,ing cols）

这个构造方法将创建一个 rows 行、cols 列的网格布局管理器对象，组件之间没有间隙。

GridLayout（int rows,ing cols,int hgap,int vgap）

这个构造方法将创建一个 rows 行、cols 列的网格布局管理器对象，组件之间的水平间隙为 hgap，垂直间隙为 vgap。

除此之外，这个类还提供了有关获取和设置行数和列数以及与 FlowLayout 一样的获取和设置组件间隙的成员方法

例 9.7　GridLayout 布局管理器

```
import java.awt.GridLayout;
import javax.swing.*;
public class Example9_7 extends JFrame {
    private JButton button1 = new JButton("First Button");
    private JButton button2 = new JButton("Second Button");
    private JButton button3 = new JButton("Third Button");
    private JButton button4 = new JButton("Fourth Button");
    public Example9_7() {
        super("GridLayout布局管理器示例");
        setSize(300, 200);
        setLocation(400, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // 设置布局方式为GridLayout,2行，2列
        getContentPane().setLayout(new GridLayout(2, 2));
        // 添加组件时不需要设置组件所在行、列
        getContentPane().add(button1);
        getContentPane().add(button2);
```

```
        getContentPane().add(button3);
        getContentPane().add(button4);
    }
    public static void main(String arg[]) {
        Example9_7 frame = new Example9_7();
        frame.setVisible(true);
    }
}
```

运行这个程序将会在屏幕上看到如图 9_10 所示的结果。

图 9_10 GridLayout 布局管理示例

9.4 Swing 组件

Swing 组件包括一些比较简单的组件，如标签、按钮、检查框、组合框以及列表，也包括一些比较复杂的组件如文本控件、菜单、表格、树、滑杆、进度条、文件选择器、颜色选择器等，他们都放在 javax.swing 包中。在 Swing 中，所有的组件都是 JComponent 类的子类，这个类为它的所有子类提供了下列功能：

- 工具提示。可调用 setToolTipText()成员方法指定的一个字符串，当光标停留在某个组件上时，将在组件附近开辟一个小窗口，并在窗口显示这个字符串作为提示信息。
- 绘画和设置边框。可以调用 setBorder()成员方法为组件设置一个边框。若要在组件内实现绘画功能，需要覆盖 paintConponent()成员方法。
- 控制显示外观。在每个组件的背后，都有一个对应的 ComponentUI 对象与之对应，由此完成组件的所有绘画、事件处理、定制尺寸任务。ComponentUI 对象与当前的现实外观有关，而显示外观可以通过调用 UIManager.setLookAndFeel()成员方法进行设置。
- 定制属性。对于任何 JComponent 组件，可以组合一个或多个属性。例如，一个布局管理器可以利用属性将多个约束条件组合在一起作用于一个组件上。调用 getClientProperty()成员方法将可获取组件的属性，调用 setClientProperty()成员方法将可以设置组件的属性。
- 支持拖拽功能。JComponent 类提供了 API，用来设置拖拽组件的操作。
- 支持双缓冲。为了解决刷新组件时容易产生闪烁的现象这个问题，提供了双缓冲的技术，它可以使屏幕刷新更加平滑。在默认情况下 JPanel 就是双缓冲。
- 点击绑定。当用户按下键盘键时，组件可以做出反应。例如，当一个按钮具有焦点时，按下“空格”键就等于在按钮上按下鼠标。

JComponent 不仅从 Component 和 Container 类中继承了大量的方法，还定义了许多新的成员方法，它们分别用来实现定制组件外观、设置或获取组件状态处理事件、绘制组件、处理组件层次、布局组件、获得组件大小和放置位置信息、指定组件绝对大小和位置等。

与 AWT 组件相比较，Swing 组件增加了以下几个新功能：

- 按钮和标签组件不仅可以显示文本框，还可以显示图片。
- 可以轻松地为大多数组件添加或更改边框。
- 可以通过调用成员方法或创建一个子类对象，改变 Swing 组件的外观和行为。
- Swing 组件不仅可以为矩形，还可以为其他类型。例如，可以创建一个圆形按钮。

下面介绍几种典型组件及其使用方式。

9.4.1 标签

标签是一种不响应任何事件的组件，它主要是被用来实现一些说明性的描述。在 Swing 中，用 JLable 类实现标签组件，并且显示形式得到了扩展，它不仅可以显示文字，还可以显示图片。图 9.11 显示了几种类型的标签。

在 JLable 类中，常用的方法包括：

1．构造方法。

JLable()

这是无参数的构造方法，它将创建一个内容为空的标签对象。

JLable(Icon icon)

这个构造方法将创建一个带有图标的标签对象。

JLable(Icon icon,int horizontalAlignment)

这个构造方法将创建一个带有图标 icon 的标签对象且图标的水平对齐方式由参数 horizontalAlignment 确定，他们可以使常量 LEFT、CENTER、RIGHT、LEADING 和 TRAILING。

JLable(String text)

这个构造方法将创建一个带有文本串 text 的标签对象。

JLable(String text,int horizontalAlignment)

这个构造方法将创建一个带有文本串 text 的标签对象且文本串的水平对齐方式由参数 horizontalAlignment 确定

JLable(String text,Icon icon,int horizontalAlignment)

这个构造方法将创建一个带有文本串 text、图标 icon 的标签对象且文本串的水平对齐方式由参数 horizontalAlignment 确定

2. 设置或获取图标，在标签可用和不可用时可使用不同的图标：

void setIcon(Icon image)

void setDisabledIcon(Icon image)

Icon getIcon()

Icon getDisabledIcon()

3. 设置或获取文字：

void setText(String text)

String getText()

4. 设置或获取图标和（或）文字的水平或垂直对齐方式：

void setHorizontalAlignment(int horizontalAlignment)

int getHorizontalAlignment()

void setVerticalAlignment(int verticalAlignment)

int getVerticalAlignment()

5. 设置或获取文本相对于图标的水平或垂直位置，水平位置的 textPosition 可取的值除了 LEFT、CENTER、RIGHT 之外还可取 LEADING(前导)、TRAILING（尾部，是缺省值），垂直位置的 textPosition 可取的值包括 TOP、CENTER 和 BOTTOM，这些常量都定义在接口 SwingConstants：

void setHorizontalTextPosition(int textPosition)

int getHorizontalTextPosition()

void setVerticalTextPosition(int textPosition)

int getVerticalTextPosition()

6. 设置或获取标签所关联的组件（通常是文字字段）：

void setLabelFor(Component c)

Component getLabelFor()

7. 设置或获取显示标签的快捷键(Mnemonic)，快捷键可指定是某个字符或某个键码，类 KeyEvent 使用常量定义了许多常用键的键码：

void setDisplayedMnemonic(int key)

void setDisplayedMnemonic(char aChar)

int getDisplayMnemonic()

因此使用标签主要关心的属性包括标签的文字、图标、文字相对于图标的位置、关联的组件等。例 9.8 用于创建图 9.11 所示的的标签。

例9.8 演示了标签的一些常用方法

```
import java.awt.FlowLayout;
import java.awt.GridLayout;
import javax.swing.*;
class DemoLabel extends JFrame{
    public DemoLabel(){
        super("标签示例");
        setSize(300, 200);
        setLocation(400, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // 利用图像创建一个图标
        ImageIcon icon = new ImageIcon("img/java.gif");
        // 创建第1个标签，并设置文本相对于图标的位置
        JLabel label1 = new JLabel("图标和文本", icon, JLabel.TRAILING);
        label1.setVerticalTextPosition(JLabel.CENTER);
        label1.setHorizontalTextPosition(JLabel.TRAILING);
        // 创建一个文本字段
        JTextField text1 = new JTextField("第一段文本");
        // 设置该标签关联的文本字段
        label1.setLabelFor(text1);
        // 创建第2个标签
        JLabel label2 = new JLabel("只有文本(O)", JLabel.CENTER);
        label2.setDisplayedMnemonic('O');
        // 创建一个文本字段
        JTextField text2 = new JTextField("第二段文本");
        // 设置该标签关联的文本字段
        label2.setLabelFor(text2);
        // 创建第3个标签
        JLabel label3 = new JLabel(icon, JLabel.LEADING);
        // 创建一个文本字段
        JTextField text3 = new JTextField("第三段文本");
        // 设置该标签关联的文本字段
        label3.setLabelFor(text3);
        // 使第3个标签不可用
        label3.setEnabled(false);
        // 改变放置的容器的布局管理器，设置为网格布局，共3行2列，水平间距10，垂直间距25
```

```
        GridLayout simpleLayout = new GridLayout(3, 2, 10, 25);
        getContentPane().setLayout(simpleLayout);
    // 将标签和文本字段加入面板
        getContentPane().add(label1); getContentPane().add(text1);
        getContentPane().add(label2); getContentPane().add(text2);
        getContentPane().add(label3); getContentPane().add(text3);
    }
}
public class Example9_8 {
    public static void main(String[] args) {
        DemoLabel demoLabel = new DemoLabel();
        demoLabel.setVisible(true);
    }
}
```

运行结果如图 9_11 所示。

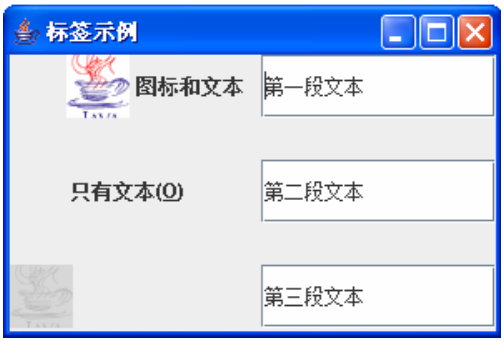

图 9_11 各种标签示例

9.4.2 按钮

按钮（Button）是种类最多，使用最频繁的组件。Swing 按钮既可以显示文字，也可以显示图像，并且每个按钮中的文字可以相对于图像显示在不同的位置，每个按钮中带下划线的字母是快捷键。例如，单击 Alt+M 等价于用鼠标单击中间的按钮。当按钮被禁用时，自动地变成禁用的外观，而且还可以提供一幅专门用于按钮禁用状态的图像。

在 Swing 中，按钮的父类是 AbstractButton。在抽象类 AbstractButton 中，定义了大量有关按钮组件操作的方法。表 9-5 列出了其中的一部分。

表 9-5 抽象类 AbstractButton 类的部分成员方法

方法	描述
boolean isSelected()	检测按钮是否被选中
void setSelected(boolean b)	设置按钮的被选中状态
String getText()	返回按钮的标签文本串
void setText(String text)	将按钮的标签文本设置为 text
Icon getIcon()	返回按钮的图标
void setIconb(Icon icon)	将按钮的图标设置为 icon
Icon getDisabledIcon()	返回按钮禁用时显示的图标
void setDisabledIcon(Icon icon)	将按钮禁用时的图标设置为 icon
Icon getPressedIcon()	返回按钮被按下时显示的图标
Void setPressedIcon(Icon icon)	将按钮被按下时的图标设置为 icon

使用按钮组建需要经过下列基本步骤：

① 创建按钮对象。

② 将按钮对象添加到容器中。

③ 设置响应单击按钮事件的操作。

下面介绍几种最常用的按钮形式：普通按钮、复选按钮和单选按钮，它们分别由 JButton、JCheckBox 和 JRadioButton 类实现。

JButton 类定义了最普通的按钮形式，用来响应用户的某项操作请求。在顶层容器中，如果有多个按钮，某一刻只能有一个默认按钮。默认按钮将呈现高亮度显示外观，并且当顶层容器获得输入焦点时，单击“回车”键与用鼠标单击该按钮获得同样的效果。可以调用 isDefaultButton()方法检测某个按钮是否为默认按钮，也可以利用 setDefaultButton(Button default)成员方法将 default 设置为默认按钮。除此之外，JButton 类还提供了下面 5 种格式的构造方法。

- JButton（）

这是无参数的构造方法， 它将创建一个没有文字和图标的按钮对象。

- JButton（Icon icon）

这个构造方法将创建一个图标为 icon 的按钮对象。

- JButton（String text）

这个构造方法将创建一个文本串为 text 的按钮对象。

- JButton（Action a）

这个构造方法将创建一个由 a 确定属性的按钮对象。

- JButton（String text，Icon icon）

这个构造方法将创建一个文本串为 text，图标为 icon 的按钮对象。

例9.9 演示普通按钮应用

```
import java.awt.FlowLayout;
import java.awt.event.KeyEvent;
import javax.swing.*;
public class Example9_9 extends JPanel {
    private JButton b1,b2,b3;
    public Example9_9() {
        b1 = new JButton("Disable");
        b1.setMnemonic(KeyEvent.VK_D); //快捷键Alt_D
        b1.setToolTipText("这是第一个按钮");  //设置鼠标停留时的提示信息
        b2 = new JButton("Middle");
        b2.setMnemonic(KeyEvent.VK_M); //快捷键Alt_M
        b3 = new JButton("Enable");
        b3.setMnemonic(KeyEvent.VK_E); //快捷键Alt_E
        b3.setEnabled(false);          //禁用此按钮
        add(b1);                     //将按钮放放面板中
        add(b2);
        add(b3);
    }
    public static void main(String[] args) {
        JFrame frame = new JFrame("按钮示例");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Example9_9 panel = new Example9_9();
```

```
        frame.getContentPane().add(panel);
        frame.setSize(300,80);
        frame.setVisible(true);
    }
}
```

运行这个程序后将在屏幕上看到如图 9_12 的结果。左侧和中间的两个按钮处于启用状态，右侧按钮处于禁用状态。

图 9_12 按钮示例

在这个例子中，调用按钮对象的 setMnmonic 为按钮设置了快捷键，按下了 Alt 及另外一个按键实现可以单击的操作。另外，还通过调用按钮对象的 setToolTipText()成员方法为 b1 按钮设置了提示信息，当鼠标移到按钮的显示区域并停留片刻后会显示。如图 9_12 所示的提示信息。

9.4.3 复选按钮

Swing 中的复选按钮用 JCheckBox 类实现。与 JButton 相同，可以为之设置文本串和图标。通常，将多个复选按钮组合成一组，对于处于一组中的复选按钮，每一时刻既可以选择一项，又可以选择多项。图 9.13 就是一个含有复选按钮的窗口。

JCheckBox 类提供了下面 7 种格式的构造方法：

- JCheckBox()

这是默认的构造方法。它将创建一个没有文本串、没有图标、没有被选中的复选按钮对象。

- JCheckBox（Icon icon）

它将创建一个图标为 icon 的复选按钮对象。

- JCheckBox（Icon icon，boolean selected）

它将创建一个图标为 icon 的复选按钮对象，是否被选中取决于 selected。如果 selected 为 true，复选按钮的初始状态为被选中。

- JCheckBox（String text）

它将创建一个文本串为 text 的复选按钮对象。

- JCheckBox（Action a）

它将创建一个由 a 确定属性的复选按钮对象。

- JCheckBox（String text，boolean selected）

它将创建一个文本串为 text 的复选按钮对象，是否被选中取决于 selected。如果 selected 为 true，复选按钮初始处于被选中状态。

- JCheckBox（String text,Icon icon）

它将创建一个文本串为 text、图标为 icon 的复选按钮对象。

对于复选框，可能监听的常见事件类型是 ItemEvent，选中或不选检查框时发生该类型的事件，创建复选框的类应使用实现接口 ItemListener 的类监听该类型的事件。

例 9.10 复选按钮的示例

```
import java.awt.GridLayout;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
```

```
import javax.swing.*;
class CheckBoxDemo extends JPanel implements ItemListener {
    private static final int NUM=5;
    //各种兴趣爱好
    private String interest[] = {"网球", "羽毛球", "乒乓球", "篮球", "其它"};
    private char mnemonic[] = {'T', 'B', 'P', 'K', 'O'}; // 对应的快捷键
    private String toolTip[] = {"美国网球公开赛", "世界羽毛球锦标赛", "世界乒乓球锦
标赛","美国职业篮球赛", "你还想看什么赛事？"}; // 复选框对应的提示信息
    private JCheckBox[] checkBox = new JCheckBox[NUM]; // 复选框
    private JLabel textLabel;
    //用于显示爱好和相应赛事字符串的文本域
    private   JTextArea interestTextArea;
    private JTextArea gameTextArea;
    public CheckBoxDemo(){
        super(new GridLayout(2,2));
        JPanel checkPanel = new JPanel(new GridLayout(0,1));
        for (int i = 0; i < NUM; i++) {
            checkBox[i] = new JCheckBox(interest[i]);//创建检查框
            checkBox[i].setMnemonic(mnemonic[i]); // 设置快捷键
            checkBox[i].setToolTipText(toolTip[i]); // 设置工具提示
            checkBox[i].addItemListener(this); // 设置这些检查框的事件监听者
            checkPanel.add(checkBox[i]);
        }
        textLabel = new JLabel("请选择你的兴趣和爱好：", JLabel.LEADING);
        interestTextArea = new JTextArea(getInterestString());//用初始文本创建
标签
        interestTextArea.setEditable(false); // 设置不可编辑
        gameTextArea = new JTextArea(getGameString());
        gameTextArea.setEditable(false);
        add(textLabel);
        add(interestTextArea);
        add(checkPanel);
        add(gameTextArea);
    }
    public void itemStateChanged(ItemEvent e) {
        //重新设置右边标签中显示爱好和赛事的字符串
        interestTextArea.setText(getInterestString());
        gameTextArea.setText(getGameString());
    }
    //生成显示爱好所用的字符串
    private String getInterestString() {
        StringBuffer temp = new StringBuffer("你选择的爱好是：");
        boolean isFirst = true;
        for (int i = 0; i < NUM; i++) {
```

```
            if (checkBox[i].isSelected()) {
                if (isFirst) { temp.append(interest[i]); isFirst = false; }
                else temp.append(", " + interest[i]);
            }
        }
        if (isFirst) temp.append("没有任何爱好!");
        return temp.toString();
    }
    //生成显示所选爱好对应赛事的字符串
    private String getGameString() {
        StringBuffer temp = new StringBuffer("你喜欢的赛事是：\n");
        boolean isFirst = true;
        for (int i = 0; i < NUM; i++) {
            if (checkBox[i].isSelected()) {
                temp.append(toolTip[i]+"\n");
                isFirst = false;
            }
        }
        if (isFirst) temp.append("不喜欢任何赛事!");
        return temp.toString();
    }
}
public class Example9_10{
    public static void main(String args[]){
        JFrame frame = new JFrame("复选框示例");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        CheckBoxDemo panel = new CheckBoxDemo();
        frame.getContentPane().add(panel);
        frame.setSize(400,200);
        frame.setVisible(true);
    }
}
```

运行这个程序后将在屏幕上看到如图 9_13 所示的结果。

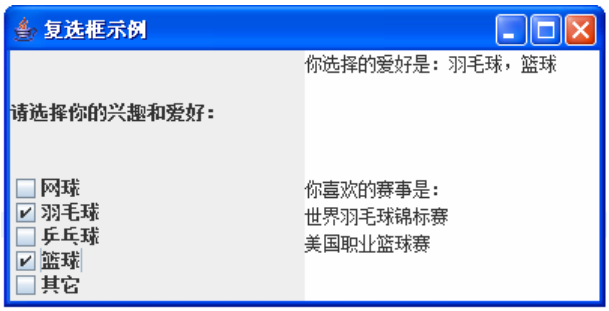

图 9_13 复选框示例

这是一个很有趣的例子，将 5 个复选按钮组织在一起放到一个面板中，最初 5 个复选框均被未被选中。一旦选中或取消选中某个复选项，就会从文本区域中看到相应的反应。如图 9.13 是选中羽毛球和篮

球的情况。

该例中类 CheckBoxDemo 实现了接口 ItemListener,因此它本身也是事件监听器，itemStateChanged()方法是该接口定义的惟一方法。事件监听器监听项目事件，项目事件通常是某个项目被选中或不被选中发生状态变化时所激发的事件。项目事件对象所属的类是 ItemEvent，这个类常用的方法包括：

- Object getSourse()

返回事件源对象，该方法从事件类的祖先类 java.util.EventObject 继承

- ItemSelectable getSelectable()

同样返回事件源，激发 ItemEvent 的事件源应该实现了接口 ItemSelectable

- Object getItem()

返回该事件所影响的项目

- int getStateChange()

返回项目选择状态，返回 ItemEvent.SELECTED 表示选中，否则表示没有选中

例 9.10 的事件处理程序并没有从事件对象获取检查框是否选中的信息，而是直接通过检查框的方法isSelected()判断某个检查框是否选中，根据这些信息动态修改文本区域的文本，反映用户所选择的爱好。

9.4.4 单选按钮

单选按钮通常成组出现，每一组的多个单选按钮，在每一时刻仅能有一个按钮被选中。Swing 用 JRadioButton 和 ButtonGroup 共同协作实现单选按钮的操作。由于 JRadioButton 派生于 AbstractButton，所以它也具有普通按钮的所有特性，例如，在单选按钮中显示指定图片等。图 9-14 是一个拥有一组单选按钮的窗口外观。

要想使某些单选按钮成为一组，就需要创建一个 ButtonGroup 对象，然后将每个单选按钮对象作为add()成员方法的参数添加到 ButtonGroup 对象表示的成组组件中。

JRadioButton 提供了以下 8 种格式的构造方法。

- JRadioButton（）

这是默认的构造方法，它将创建一个没有标签、没有图标、没有被选中的单选按钮。

- JRadioButton（String text）

它将创建一个标签为 text 的单选按钮。

- JRadioButton（String text，boolean selected）

它将创建一个标签为 text 的单选按钮。初始是否处于选中状态取决于 selected，如果 selected 为 true，初始被选中。

- JRadioButton（Icon icon）

它将创建一个图标为 icon 的单选按钮。

- JRadioButton（Icon icon，boolean selected）

它将创建一个图标为 icon 的单选按钮。初始是否处于选中状态取决于 selected，如果 selected 为 true，初始被选中。

- JRadioButton（String text，Icon icon）

它将创建一个标签为 text，图标为 icon 的单选按钮。

- JRadioButton（String text，Icon icon，boolean selected）

它将创建一个标签为 text，图标为 icon 的单选按钮。初始是否处于选中取决于 selected，如果 selected 为 true，初始被选中。

- JRadioButton（Action a）

它将创建一个由 a 确定属性的单选按钮。

对于单选按钮，可能监听的常见事件类型是 ActionEvent，选中或不选中单选按钮时发生该类型的事件，创建单选按钮的类应使用实现接口 ActionListener 的类监听该类型的事件

例 9.11 单选按钮的示例

```
    import java.awt.BorderLayout;
    import java.awt.GridLayout;
    import javax.swing.*;
    class RadioButtonDemo extends JPanel {
        private static final int NUM = 4;
        private String titles[] = {"皑皑白雪", "幽幽山野", "默默余辉", "婷婷荷花"};
        private String toolTip[] = {"snow.jpg", "forest.jpg", "sunset.jpg",
"water_lily.jpg"};// 各种图片对应的文件名
        private JLabel picture;
        public RadioButtonDemo(){
            super(new BorderLayout());
            JPanel left = new JPanel(new GridLayout(0,1));
            //为该面板创建一盒式布局管理器
            BoxLayout layout1 = new BoxLayout(left, BoxLayout.Y_AXIS);
            left.setLayout(layout1);
            //创建一标签提示用户选择图片
            JLabel label = new JLabel("请选择你想欣赏的图片：", JLabel.LEADING);
            left.add(label);
            //创建这些单选按钮的事件监听者
            java.awt.event.ActionListener listener = new
SimpleRadioButtonListener();
            //创建这些按钮所在的组
            ButtonGroup group = new ButtonGroup();
            //创建单选按钮
            for (int i = 0; i < NUM; i++) {
                JRadioButton button = new JRadioButton(titles[i]);// 创建单选按钮
                if (i == 0) button.setSelected(true); // 设置选择第一个广播按钮
                button.setToolTipText(toolTip[i]); // 设置工具提示
                button.setActionCommand(toolTip[i]); // 使用图片文件名作为动作命令名
                button.addActionListener(listener); // 设置这些单选按钮的事件监听者
                group.add(button); // 将它们加入到按钮组group中
                left.add(button); // 将它们加入到放置的窗格中
            }
            //分隔窗格的右边区域用来相应的图片
            ImageIcon icon = new ImageIcon("img/"+toolTip[0]);
            picture = new JLabel(icon);
            add(left,BorderLayout.WEST);
            add(picture,BorderLayout.EAST);
        }
        private class SimpleRadioButtonListener implements
java.awt.event.ActionListener{
            //事件监听程序
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                //实际上使用动作命令名称来传递事件处理时所需要的信息!
```

```
            picture.setIcon(new ImageIcon("img/"+evt.getActionCommand()));
        }
    }
}
public class Example9_11 {
    public static void main(String[] args) {
        JFrame frame = new JFrame("欢迎进入Java语言的GUI世界");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        RadioButtonDemo demo = new RadioButtonDemo();
        frame.getContentPane().add(demo);
        frame.setSize(800,600);
        frame.setVisible(true);
    }
}
```

运行这个程序后将会在屏幕上看到如图 9_14 所示的结果。

图 9_14 含有单选按钮的窗口

程序运行过程，可以看到右侧显示的图片将根据被选择的单选按钮发生变化。在程序中，调用 setActionCommand()成员方法的目的是为每个单选按钮的操作设置一个字符串，在事件监听程序中，当选中某个单选按钮时就会返回相应的字符串作为图片的文件名使用，随后可以根据这个字符串做出相应的操作。

9.4.5 文本框

文本框是接受用户输入的一种组件，Swing 将文本组件分为三大类：

1. 文本字段(text fields)，包括类 JTextField 和 JPasswordField，用于显示和编辑较短的、不带格式的一行文本。其中，类 JPasswordField 是类 JTextField 的子类，用于输入口令，它与类 JTextField 的区别在于用户输入的文字不会显示，而显示特定的字符作为掩码（mask）字符。文本字段的使用基于动作事件（即主要监听动作类型的事件 ActionEvent），与按钮等简单控件的用法类似；

2. 纯文本区域(plain text areas)，包括类 JTextArea，用于显示和编辑较长的、不带格式的多行文本，通常用于显示没有格式的帮助信息。纯文本区域的使用基于文档事件（即主要监听文档类型的事件 DocumentEvent），程序通常使用文档事件监听器监听用户对纯文本区域的改变；

3. 带样式文本区域(styled text areas)，包括类 JEditorPane 和 JTextPane，可用于显示和编辑复杂的、带样式的文本。JEditPane 类支持纯文本、HTML 和 RTF 的文本编辑；JTextPane 类进一步扩展了 JEditPane 类的功能，允许文本中嵌入图像或其他组件。

下面主要介绍 JTextField 和 JPasswordField 的使用方式。其他文本框的使用方式基本类似。

JTextField 类是一种经常使用的组件。它主要提供了下列几种构造方法。

- JTextField()

这是无参数的构造方法，它将创建一个初始为空，可显示字符列数为零的文本框对象。

◆ JTextField(String text)

这个构造方法将创建一个初始内容为 text 的文本框对象。

◆ JTextField(String text，int col)

这个构造方法将创建一个初始内容为 text，可显示字符列数为 col 的文本框对象。

◆ JTextField(int col)

这个构造方法将创建一个初始内容为空，可显示字符列数为 col 的文本框对象。

除此之外，这个类还提供很多成员方法，可以在程序中方便地获取或设置文本框组件的各个属性。表 9-5 列出了其中的一部分常用方法。

表 9-5 JTextField 类的部分成员方法

成员方法	描述
String setText()	返回文本框中的文本串
void setText(String text)	将文本框中的内容设置为 text
boolean isEditable()	检测文本框是否可编辑。如果返回 true，表示该文本框可编辑
void setEditable(boolean editable)	设置文本框的可编辑性。如果 editable 为 true，表示该文本框设置为可编辑
int getColunmns()	返回文本框所显示的字符列数
void setColumns(int col)	将文本框能够显示字符的列数设置为 col

类 JPasswordField 的用法与类 JTextField 基本相同，惟一区别是类 JPasswordField 创建的组件在输入时不回显用户输入的真正内容而使用掩码字符代替。JPasswordField 中用于设置或获取回显的掩码字符如下，以代替用户输入的真正内容：

void setEchoChar(char c)

char getEchoChar()

例 9.12 文本框示例

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.*;
public class Example9_12 {
    public static void main(String[] args)
    {
        final JFrame f = new JFrame("文本框示例");
        JLabel label = new JLabel("请输入密码:");
        JPasswordField pfield = new JPasswordField(10);
        pfield.setEchoChar('*');
        pfield.addActionListener(new ActionListener(){  //注册事件监听器
            public void actionPerformed(ActionEvent e){  //处理事件
                JPasswordField input = (JPasswordField)e.getSource();
                char[] password = input.getPassword();
                if(isPasswordCorrect(password)){
                    JOptionPane.showMessageDialog(f, "成功!你输入了正确的密码");
                    }else{
```

```
                        JOptionPane.showMessageDialog(f, "无效的密码,请再试一次","
错误信息",JOptionPane.ERROR_MESSAGE);
                    }
                }
            });
        JPanel contentPane = new JPanel(new BorderLayout());
        contentPane.setBorder(BorderFactory.createEmptyBorder(20,20,20,20));
        contentPane.add(label,BorderLayout.WEST);
        contentPane.add(pfield,BorderLayout.CENTER);
        f.setContentPane(contentPane);
        f.pack();
        f.setVisible(true);
    }
    private static boolean isPasswordCorrect(char[] input)
    {
        char[] correctPassword={'J','a','v','a'};
        if(input.length!=correctPassword.length)
            return false;
        for(int i=0;i<input.length;i++){
            if(input[i]!=correctPassword[i])
                return false;
        }
        return true;
    }
}
```

图 9_15 文本框示例

运行这个程序后，将会在屏幕上看到如图 9_15 所示的结果。如果输入了正确的密码"Java"，将显示“成功”对话框；否则，显示“失败”对话框。

9.4.6 组合框

组合框允许用户从若干个选项中选择一项。在 Swing 中用 JComboBox 类实现，它提供了两种不同形式的组合框。一种是不可编辑的组合框，它由一个按钮和下拉列表组成，这是默认形式；另一种是可编辑的组合框，它由一个可接收用户输入的文本框、按钮和下拉列表组成，用户既可以在文本框中输入文本串，也可以单击按钮，打开下拉列表。图 9_16 就是组合框应用的示例。

组合框占屏幕空间小，特别是可编辑形式的组合框，通过在文本框中输入文本串，可以突破只能选择列表中存在的选项的限制。与单选按钮一样，组合框也是用来处理成组选项的，但单选按钮适于用在选项比较少的时候，它更容易让用户一目了然；而组合框适用于屏幕空间比较紧张，选项较多的情形。

下面是 JComboBox 类提供的几个构造方法。

JComboBox()

这是无参数的构造方法，它将创建一个没有选项的空组合框对象。

JComboBox（Object[] item）

这个构造方法将创建一个组合框对象，其初始选项为 item 数组中的对象。

JComboBox（Vector item）

这个构造方法将创建一个组合框对象，其出事选项为 item 向量中的内容。

此外，还有几个用来获取或设置组合框属性的方法，其中一些被列在表 9.6 中。

表 9-6　JComboBox 类的部分成员方法

方法	描述
void addItem(Object item)	将 item 插入到组合框的尾部
void insertItemAt(Object item, int index)	将 item 插入到索引号为 index 的位置
Object getItemAt(int index)	返回索引号为 index 的选项对象
Object getSelectedItem()	返回被选中的对象
void removeItem(Object item)	从组合框中删除 item 选项
int getItemCount()	返回组合框中的选项数目

例 9.13　组合框应用示例

```
import java.awt.*;
import javax.swing.*;
class ComboBoxDemo extends JPanel {
    private static final int NUM = 4;
    private String titles[] = {"皑皑白雪", "幽幽山野", "默默余辉", "婷婷荷花"};
    private String files[] = {"snow.jpg", "forest.jpg", "sunset.jpg",
"water_lily.jpg"};//各种图片的文件名
    private JLabel picture;
    public ComboBoxDemo(){
        super(new BorderLayout());
        //窗格的上面区域用来放置组合框，创建一窗格来放置组合框
        JPanel north = new JPanel(new GridLayout(0,1));
        //为该面板创建一盒式布局管理器
        BoxLayout layout1 = new BoxLayout(north, BoxLayout.Y_AXIS);
        north.setLayout(layout1);
        //创建一标签提示用户选择图片
        JLabel label = new JLabel("请选择你想欣赏的图片：", JLabel.LEADING);
        north.add(label);
        //创建组合框的事件监听者
        java.awt.event.ActionListener listener = new SimpleComboBoxListener();
        //创建组合框
        JComboBox comboBox = new JComboBox(titles); // 创建组合框
        comboBox.setSelectedIndex(0); // 设置选择第一个组合框
        comboBox.addActionListener(listener); // 设置这些组合框的事件监听者
        comboBox.setAlignmentX(Component.LEFT_ALIGNMENT);
        north.add(comboBox); // 将它们加入到放置的窗格中
        //分隔窗格的右边区域用来相应的图片
        ImageIcon icon = new ImageIcon("img/"+files[0]);
        picture = new JLabel(icon);
        add(north,BorderLayout.NORTH);
```

```
            add(picture,BorderLayout.CENTER);
        }
        private class SimpleComboBoxListener implements
java.awt.event.ActionListener {
            //事件监听程序
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                JComboBox source = (JComboBox)evt.getSource();
                picture.setIcon(new
ImageIcon("img/"+files[source.getSelectedIndex()]));
            }
        }
    }
    public class Example9_13 {
        public static void main(String[] args) {
            JFrame frame = new JFrame("欢迎进入Java语言的GUI世界");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            ComboBoxDemo demo = new ComboBoxDemo();
            frame.getContentPane().add(demo);
            frame.setSize(800,600);
            frame.setVisible(true);
        }
    }
```

运行这个程序后将在屏幕上看到如图 9_16 所示的结果。

图 9_16 组合框应用示例

可以看到，当从组合框中选择一种情景，下面图片就会随之改变。

9.5 Java 事件类及处理

9.5.1 Java 事件类

AWT 和 Swing 定义了多种类型的事件，这些事件大体上可分为两大类型：

1. 低级事件：由窗口系统发生的事情或者底层输入而产生的事件，主要包括组件事件（跟踪组件位置、大小可见性的变化）、容器事件（反映容器中组件的添加和删除）、焦点事件（组件是否获得或失去键盘焦点，拥有键盘焦点的组件可响应键盘的输入）、窗口事件（反应窗口，画框或对话框的基本状态）、鼠标事件（用户使用鼠标进行交互）、键盘事件（用户使用键盘进行输入）等。

2. 语义事件：低级事件以外的，具有丰富含义的与组件有关的事件。如单击按钮，拖动滚动条等。这些事件源于图形用户界面，其含义由程序设计员赋予，例如，“确定”按钮将确认刚才的操作，“取消”按钮斜撤销刚才的操作。

在很多时候，这两种事件类型是重叠出现的，很难划分清楚。例如，当利用鼠标单击一个按钮时，既发生了单击鼠标的低级事件，又同时发生了单击按钮的语义事件。遇到这类情况应该如何处理，完全取决于程序设计。在大多数情况下，高级语义事件优先于低级主义事件。

程序应尽可能监听语义事件而不是低级事件，因为语义事件具有更明确的语义，例如事件源比较容易确定，应执行的事件处理动作比较明确等。低级事件通常有多个可能的事件源，例如就点击鼠标而言，用户界面的任何区域都可能点击鼠标，不同的区域需要进行不同的处理，不同的点击方式也可能要进行不同的处理，因此鼠标事件的监听往往十分复杂。语义事件则对这种低级事件进行了包装，使其目的更为明确，也更容易为编程人员所掌握。

绝大部分与图形用户界面有关的事件类都位于 java.awt.event 包中，其中包含了各种事件的监听接口，而在 javax.swing.event 包中定义了与 Swing 事件有关的事件类。

下面首先介绍低级事件和语义事件的处理。

1．低级事件

焦点事件、鼠标事件、键盘事件和窗口事件都属于低级事件，弄清它们的处理方式对于编写正确的程序至关重要，表 9-7 列出了低级事件的事件类名和事件描述。

表 9-7 低级事件的类名及描述

事件类名	描 述
FocusEvent	在组件获得焦点或失去焦点时产生的事件
MouseEvent	用户对鼠标操作所产生的事件
KeyEvent	用户对键盘操作所产生的事件
WindowEvent	用户对窗口操作所产生的事件

这几个类都位于 java.awt.event 包中，它们都是 ComponentEvent 类的子类。AWTEvent 类是 java.util.EventObject 的子类，该类提供了一个用于获得事件源对象的成员方法 getSource()。

在 AWTEvent 类中定义了一些用于标识事件的常量。表 9-8 列出了一些常见的事件常量。

表 9-8 事件常量

常 量	事 件
MOUSE_EVENT_MASK	鼠标事件。例如，按下鼠标，释放鼠标
KEY_EVENT_MASK	键盘事件。例如，按下键盘中的某个键
ITEM_EVENT_MASK	选择选项事件。例如，从列表组件中选择某项
WINDOW_EVENT_MASK	窗口事件。例如，关闭窗口
MOUSE_MOTION_EVENT_MASK	鼠标移动事件
FOCUS_EVENT_NASK	焦点事件

为了更有效地处理各种类型的事件，在 Java 中，每一种事件都对应一个监听器接口，负责处理事件的监听器必须实现对应的监听器接口。图 9_17 给出了对应低级事件的 5 种监听器接口关系图。其中，EventListener 接口没有声明任何内容，但所有的监听器接口都必须是它的子接口，这样，便于统一地扩展监听器的功能。如果想要处理上述事件，就要定义实现上述接口的监听器类，然后让相应的组件注册监听器。

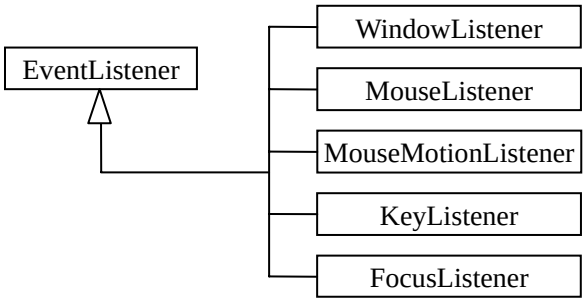

图 9_17 低级事件的监听器接口

2. 语义事件

语义事件是与组件有关的事件。表 9-9 列出了一部分描述语义事件的类，它们都是 AWTEvent 的子类，位于 java.event 包中。

表 9-9 语义事件类

事件类名	描述
ActionEvent	激活组件事件。当在文本框内按“回车”键、单击按钮时发生该事件
ItemEvent	选项事件。当选择了某些选项时发生该事件
ComponentEvent	组件事件。当组件被移动、拖放、显示或隐藏时产生该事件
ContainerEvent	窗口事件。当向窗口中添加组件或删除组件时发生该事件
TextEvent	文本框事件。当文本框内容发生变化时发生该事件

与低级事件一样，每一种事件都对应一个监听接口，这些监听器接口同样是 EventListener 接口的子接口。图 9_18 列出了这些接口之间的关系。

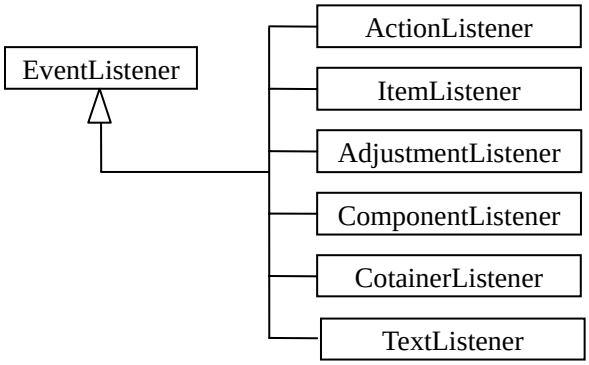

图 9_18 语义事件的监听器接口

同样，在这些接口中声明了处理每一种事件的成员方法。要想处理这些事件，需要设计一个实现这些接口的监听器类，稍后将举例说明具体的实施过程。

9.5.2 窗口事件的处理

窗口事件的标识是 WINDOW_EVENT_MASK，描述这个事件的类是 WindowEvent。也就是说，不管对窗口做了什么的操作，都会产生该标识的事件。由于与窗口操作相关的事件不止一种，所以对这类事件还附加了一个 ID，用来标识具体的窗口操作行为。这些 ID 被定义在 WindowEvent 类中，它们被列在表 9-10 中。

表 9-10 窗口事件的 ID 定义

事件 ID	描述
WINDOW_OPENED	窗口被打开时产生这个事件
WINDOW_CLOSING	单击“关闭”窗口图标或从系统菜单中选择“关闭”时产生这个事件
WINDOW_CLOSED	关闭窗口时产生这个事件
WINDOW_ACTIVATED	窗口被激活时产生这个事件

WINDOW_DEACTIVATED	窗口由激活状态变为失活窗台时产生这个事件
WINDOW_ICONFIED	窗口被极小化成图标时产生这个事件
WINDOW_DEICONIFIED	窗口由图标状态复原时产生这个事件

对应于窗口事件的监听器接口为 WindowListener，在这个接口中包含了每一种具体窗口事件的抽象方法。当某个事件发生时，将自动地调用下面相应的方法。

windowOpened(WindowEvent e) 处理 WINDOW_OPENED 事件。

windowClosing(WindowEvent e)处理 WINDOW_CLOSING 事件。

windowClosed(WindowEvent e) 处理 WINDOW_CLOSED 事件。

windoActivated(WindowEvent e)处理 WINDOW_ ACTIVATED 事件。

windowDeactivated(WindowEvent e) 处理 WINDOW_DEACTIVATED 事件。

WindowIconfied(WindowEvent e) 处理 WINDOWICONFIED 事件。

windowDeiconfied(WindowEvent e) 处理 WINDOW_DEICONIFIED 事件。

正如前面所说的，要想处理窗口事件就需要定义一个实现该接口的监听器类，并创建一个临听器，即监听器类的对象，然后让试图处理该事件的窗口对象注册该监听器即可。

下面是一个处理窗口事件的例子。在发生各自窗口事件时，应用程序将调用监听器中定义的相应成员方法，实现特定的操作。

例 9.14 处理窗口事件（监听器类可以是窗口类）

```
import java.awt.BorderLayout;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;
import javax.swing.*;
class WindowEventDemo extends JFrame implements WindowListener
{
    JLabel label;
    public WindowEventDemo()
    {
        super("WindowEvent示例");      //设置窗口标题
        label = new JLabel();         //创建标签组件
        getContentPane().add(label,BorderLayout.CENTER);
        this.addWindowListener(this); //为窗口注册监听器
        setSize(300,140);            //设置窗口大小
        setVisible(true);            //将窗口设置为可见
    }
    public void windowOpened(WindowEvent e) {   //打开窗口事件
        displayMessage("打开窗口",e);
    }
    public void windowClosing(WindowEvent e)        //处理关闭窗口事件
    {
        displayMessage("关闭窗口",e);
        dispose();
        System.exit(0);
    }
    public void windowDeactivated(WindowEvent e) {  //窗口失活事件
        displayMessage("窗口失去焦点",e);
```

```
    }
    public void windowActivated(WindowEvent e) {   //激活窗口事件
        displayMessage("窗口获得焦点",e);
    }
    public void windowClosed(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    //设置标签内容
    public void displayMessage(String prefix,WindowEvent e)
    {
        label.setText(prefix+"':"+e.getWindow()+"\n");
    }
}
public class Example9_14 {
    public static void main(String args[])
    {
        WindowEventDemo frame = new WindowEventDemo();
    }
}
```

运行上面这个程序，将在打开窗口、关闭窗口、激活窗口、使窗口失活时显示一个信息框。

监听器类可以是窗口类本身，也可以是一个独立的类，还可以是内部类或匿名类。上面的例 9.14 是将窗口类本身实现 WindowListener接口，使之成为监听器类，之后在构造方法中调用addWindowListener()方法将该对象注册到该类对象表示的窗口中。

同样还是这个程序，也可以单独定义一个监听器类。下面就是采用这种方式编写的程序代码。

例 9.15 处理窗口事件（监听器类是一个独立的类）

```
import java.awt.BorderLayout;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;
import javax.swing.*;
class WindowEventDemo2 extends JFrame   //窗口类
{
    JLabel label;
    public WindowEventDemo2()
    {
        super("WindowEvent示例2");
        label = new JLabel();
        getContentPane().add(label,BorderLayout.CENTER);
        this.addWindowListener(new WindowEventHandle()); //注册监听器
        setSize(300,140);
        setVisible(true);
    }
}
class WindowEventHandle implements WindowListener  //窗口事件的监听器类
{
```

```
        public void windowOpened(WindowEvent e) {   //打开窗口事件
            displayMessage("打开窗口",e);
        }
        public void windowClosing(WindowEvent e)        //处理关闭窗口事件
        {
            displayMessage("关闭窗口",e);
            ((WindowEventDemo2)e.getSource()).dispose();
            System.exit(0);
        }
        public void windowDeactivated(WindowEvent e) {  //窗口失活事件
            displayMessage("窗口失去焦点",e);
        }
        public void windowActivated(WindowEvent e) {   //激活窗口事件
            displayMessage("窗口获得焦点",e);
        }
        public void windowClosed(WindowEvent e) {}
        public void windowDeiconified(WindowEvent e) {}
        public void windowIconified(WindowEvent e) {}
        //设置标签内容
        public void displayMessage(String prefix,WindowEvent e)
        {

    ((WindowEventDemo2)e.getSource()).label.setText(prefix+"':"+e.getWindow()+"
\n");
        }
    }
    public class Example9_15 {
        public static void main(String args[])
        {
            WindowEventDemo2 frame = new WindowEventDemo2();
        }
    }
```

从这个例子可以看出，这种“委托”模式的事件处理方式需要为每个事件种类定义相应的监视器类，而这些监视器类需要实现相应的监听器接口，所以在监听器类中必须实现接口中的每个成员方法。例如，上面列举的窗口事件包含 7 种具体的事件。在接口中，每一种事件对应一个成员方法，因此，在监视器类的定义中就需要实现接口中的 7 个成员方法。在大多数情况下，可能只对其中的几种事件感兴趣。而 Java 语法的规定，监视器类需要实现接口中的全部方法，这样就需要将没有特别操作要求的那些事件对应的成员方法设计为空。显然，这会增加程序的复杂性。为了解决这个问题，Java 提出了适配器的概念。

所谓适配器是指 API 提供的一种实现了监听器接口的类。实际上，并不是每一个监听器接口都存在一个适配器。如果某个监听器类只包含一个成员方法，就没有必要提供适配器了。

WindowListener 是窗口事件的监听器类，其中包含 7 个成员方法。因此 API 提供了一个适配器 WindowAdapter 接口，该适配器不仅实现了 WindowListener 接口，还实现了 WindowFocusListener 接口及 WindowStateeListenerr 接口，因此，该适配器具有处理这 3 种事件的功能。下面是 WindowAdapter 类的源代码。

```
public abstract class WindowAdapter
          implements WindowListener,WindowStateListener,WindowFocusListener
{
    public void windowActivated(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowClosing(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowOpened(WindowEvent e) {}
    public void windowStateChanged(WindowEvent e) {}
    public void windowGainedFocus(WindowEvent e) {}
    public void windowLostFocus(WindowEvent e) {}
}
```

WindowAdapter 一个抽象类，其中包含属于 3 个监听器接口的总共 10 个成员方法的定义，由于在此无法确定每个事件的具体操作行为，所以所有的方法体均为空。

有了适配器，就可以比较轻松地定义监听器类。即将监听器类设置为是 WindowAdapter 的子类，并在子类中覆盖那些感兴趣的成员方法，这样就可以大大地减少编写程序的工作量。下面是一个使用适配器构造监听器类的例子。

例 9.16 处理窗口事件（监听器类从适配器类继承来）

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class WindowAdapterDemo extends JFrame   //窗口类
{
    JLabel label;
    public WindowAdapterDemo()
    {
        super("WindowEvent示例2");
        label = new JLabel();
        getContentPane().add(label,BorderLayout.CENTER);
        this.addWindowListener(new WindowEventHandle2()); //注册监听器
        setSize(300,140);
        setVisible(true);
    }
}
class WindowEventHandle2 extends WindowAdapter  //窗口事件的监听器
{
    public void windowOpened(WindowEvent e) {   //打开窗口事件
        displayMessage("打开窗口",e);
    }
    public void windowClosing(WindowEvent e)       //处理关闭窗口事件
    {
        displayMessage("关闭窗口",e);
```

```
            ((WindowAdapterDemo)e.getSource()).dispose();
            System.exit(0);
        }
        public void windowDeactivated(WindowEvent e) {  //窗口失活事件
            displayMessage("窗口失去焦点",e);
        }
        public void windowActivated(WindowEvent e) {   //激活窗口事件
            displayMessage("窗口获得焦点",e);
        }
        //设置标签内容
        public void displayMessage(String prefix,WindowEvent e)
        {

    ((WindowAdapterDemo)e.getSource()).label.setText(prefix+"':"+e.getWindow()+
"\n");
        }
    }
    public class Example9_16 {
        public static void main(String args[])
        {
            WindowAdapterDemo frame = new WindowAdapterDemo();
        }
    }
```

由于使用了适配器，所以在监视器类中只需覆盖那些感兴趣的成员方法。因此，建议大家在定义监视器类时，尽可能地使用 API 提供的适配器。

9.5.3 键盘事件的处理

键盘事件属于低级事件，这种类型的事件常伴随语义事件的发生，例如当某个按钮组件具有焦点时，按下“回车”键，既发生了低级键盘事件，又发生了“单击按钮”的语义事件。在这个情况下，是按照低级事件处理，还是按照语义事件处理，需要根据用户的需求作出决策。但语义事件的优先级往往高于低级事件。下面首先介绍键盘事件按照低级事件的方式，然后介绍相关的语义事件的处理方式。

键盘事件用 java.awt.event.KeyEvent 类描述，其中提供了 getKeyCode()成员方法可以获得按下或释放的那个键所对应的键码；getKeyChar()成员方法用于获得按下的那个键所对应的字符；getKeyText(int Code)成员方法用于返回键码 Code 的描述信息。

键盘操作可以被分成 3 个类别，它们用不同的 ID 标识，即当发生键盘事件时，需要给出事件的标识 KEY_EVENT_MASK 及对应的事件 ID。表 9-11 列出了 3 个类别的事件 ID。

表 9-11　3 种键盘事件的 ID

事件 ID	描述
KEY_PRESSED	当按下键盘中的某个键时发生该事件
KEY_RELEASED	当释放按键时发生该事件
KEY_TYPED	当按下键盘中的字符键（非系统键）时发生该事件

处理键盘事件的监听器接口是 KeyListener 接口，在这个接口中，声明了对应上述 3 种事件的成员方法：

KeyPressed(KeyEvent) 处理 KEY_PRESSED 事件。

KeyReleased(KeyEvent) 处理 KEY_RELEASED 事件。

KeTyped(KeyEvent) 处理 KEY_TYPED 事件。

例 9.17 通过实现 KeyListener 接口定义键盘事件的监听器类

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class KeyListenerDemo extends JFrame
{
    JLabel label;
    public KeyListenerDemo()
    {
        super("KeyListenerDemo应用举例");
        label = new JLabel("没有按下键盘");
        getContentPane().add(label,BorderLayout.CENTER);
        addKeyListener(new KeyEventHandle());
        setSize(200,100);
        setVisible(true);
    }
}
class  KeyEventHandle implements KeyListener  //键盘事件监听器类
{
    public void keyPressed(KeyEvent e) {}
    public void keyReleased(KeyEvent e) {}
    public void keyTyped(KeyEvent e) {

((KeyListenerDemo)e.getSource()).label.setText("KEY_TYPED:"+e.getKeyChar())
;
    }
}
public class Example9_17 {
    public static void main(String args[]){
        KeyListenerDemo frame = new KeyListenerDemo();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

运行这个程序后，只要单击键盘上的字符键就会在屏幕上看到单击的字符。

同样，由于键盘事件含有 3 个具体的事件，对应的 KeyListener 接口也含有 3 个成员方法，因此 API 为之提供了一个适配器 KeyAdapter。下面是 KeyAdapter 的程序代码：

```
public abstract class KeyAdapter implements KeyListener
{
    public void keyTyped(KeyEvent e){}
    public void keyPressed(KeyEvent e){}
    public void keyReleased(KeyEvent e){}
}
```

利用适配器声明监听器类可以只覆盖需要处理的事件对应的成员方法。下面将上述 KeyEventHandle

类改写成继承适配器 KeyAdapter，类中只覆盖了成员方法 keyTyped(KeyEvent e)：

```
class KeyEventHandle extends KeyAdapter
{
    public void keyTyped(KeyEvent e){
        ((KeyListenerDemo)e.getSource()).label.setText("KEY_TYPED:"+e.getKey
    Char());
    }
}
```

9.5.4 鼠标事件的处理

鼠标事件由 MouseEvent 类描述。在这个类中，提供了下面几个成员方法可以获得鼠标信息的成员方法。

int getX()和 int getY()返回发生鼠标事件时光标所处的坐标位置。

Point getPoint() 以 Point 类型的形式返回发生鼠标事件时光标所处的位置。

int getClickCount() 返回单击鼠标的次数。

与窗口事件和键盘事件不同，鼠标事件被划分成两个类别：一类被称为鼠标事件，用 MOUSE_EVENT_MASK 标识；另一类被称为鼠标移动事件，用 MOUSE_MOTION_EVENT_MASK。它们分别对应 MouseListener 接口和 MouseMotionListener 接口。表 9-12 列出了这两个类别的鼠标事件所包含的具体事件 ID。

表 9-12 鼠标事件的 ID

事件 ID	描述
MOUSE_CLICKED	当单击鼠标时发生该事件
MOUSE_PRESSED	当按下鼠标时发生该事件
MOUSE_ENTERED	当鼠标进入组件显示区域时发生该事件
MOUSE_EXITED	当鼠标退出组件显示区域时发生该事件
MOUSE_RELEASED	当释放鼠标时发生该事件
MOUSE_MOVE	当移动鼠标时发生该事件
MOUSE_DRAGGED	当拖动鼠标时发生该事件

表 9-12 中的后两个 ID 属于 MOUSE_MOTION_EVENT_MASK 事件类别。

在 MouseListener 接口中声明了下面 5 个成员方法，对应处理属于 MOUSE_EVENT_MASK 事件类别的 5 个不同的事件。

mouseClicked(MouseEvent) 处理 MOUSE_CLICKED 事件。

mousePressed(MouseEvent) 处理 MOUSE_PRESSED 事件。

mouseReleased(MouseEvent) 处理 MOUSE_RELEASED 事件。

mouseEntered(MouseEvent) 处理 MOUSE_ENTERED 事件。

mouseExited(MouseEvent) 处理 MOUSE_EXITED 事件。

在 MouseMotionListener 接口中声明了处理 MOUSE_MOTION_EVENT_MASK 事件类别的以下两个不同事件的成员方法。它们是：

mouseDragged(MouseEvent) 处理 MOUSE_DRAGGED 事件。

mouseMove(MouseEvent) 处理 MOUSE_MOVE 事件。

例 9.18 通过实现 MouseListener 和 MouseMotionListener 接口定义处理鼠标事件的监听器类

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.Line2D;
import javax.swing.*;
```

```
class  DragDrawPanel extends JPanel
{
    int x1,x2,y1,y2;
    public DragDrawPanel()
    {
        addMouseMotionListener(new MouseMotionListener(){
            public void mouseMoved(MouseEvent e){}  //鼠标移动事件
            public void mouseDragged(MouseEvent e)  //鼠标拖动事件
            {
                Graphics2D g = (Graphics2D)getGraphics();
                x2 = e.getX();                    //获取鼠标坐标
                y2 = e.getY();
                g.draw(new Line2D.Double(x1,y1,x2,y2));  //绘制直线
                x1 = x2;
                y1 = y2;
                g.dispose();
            }
        });
        addMouseListener(new MouseAdapter(){          //鼠标按下事件
            public void mousePressed(MouseEvent e)
            {
                x1 = e.getX();               //获取第一个点的坐标
                y1= e.getY();
            }
        });
    }
}
class DragDrawDemo extends JFrame
{
    public DragDrawDemo()
    {
        setTitle("DragDraw");
        DragDrawPanel panel = new DragDrawPanel();  //创建绘图面板
        getContentPane().add(panel);
    }
}
public class Example9_18 {
    public static void main(String []args)
    {
        DragDrawDemo frame = new DragDrawDemo();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300,300);
        frame.setVisible(true);
    }
```

}

这个程序实现了徒手绘图的功能。当鼠标按下时调用 mouseePressed(MouseeEvent event)成员方法获得第一点的坐标；当鼠标拖动时不断地绘制直线，（x1,y1）是前一个鼠标点，作为直线的起点，（x2,y2）是当前鼠标点，作为直线的终点。

同样，MouseListener 接口和 MouseMotionListener 接口也有对应的适配器。MouseListener 接口的适配器是 MouseAdapter，MouseMotionListener 接口的适配器是 MouseMotionAdapter 适配器。但需要注意，在上述例子中，由于 DrawPane 同时实现了 MouseListener 接口和 MouseMotionListener 接口，所以它同时充当着两个类别的鼠标事件监听器，适配器无法达到这个效果，其原因是 Java 不支持多继承。

9.5.5 语义事件的处理

语义事件是与组件有关的一些事件。例如，单击某个按钮组件就会产生 ActionEvent 事件；当复选按钮或单选按钮被选择或取消选择时就会发生 ItemEvent 事件；当拖动滚动条时就发生 AdjustmentEvent 事件。这 3 个类别的事件分别用 ActionEvent.ItemEvent.和 AdjustmentEvent 类描述。

实际上，语义事件与低级事件的处理方法基本一样，都需要利用相应的接口定义监听器类，然后将监听器注册到相应的组件上。

上面 3 个类别的监听器接口分别 AcionListener 接口。ItemListener 接口和 AdjustmenttListener 接口。这 3 个接口有一个共同点是只包含一个抽象方法。表 9-13 列出了每个接口的成员方法。

表 9-13 语义事件监听器接口的成员方法

监听接口	方　法
ActionEvent	void actionPerformed(ActionEvent e)
ItemEvent	void iftemStateChanged(ItemEvent e)
AdjustmentEvent	void adjustmentValueChanged(AdjustmentEvent e)

由于这些语义事件的监听器接口都只声明了一个成员方法，所以没有必要定义适配器。

下面是一个处理按钮事件的例子。这是一个简单的计算器，其中使用了大量的按钮组件。

例 9.16 按钮事件处理

运行这个程序后应该在屏幕上看到如图 9-29 所示的结果。

这个程序实现了简单的计算器功能。其中定义了两个单击按钮的监听器类：一个是 InsertAction 内部类，在它声明的 actionPerformed(ActionEvent event)中实现了拼接数字的功能；另一个是 CommandAction 内部类，在它声明的 actionPerformed(ActionEvent event)中实现了计算。重新初始化数值的功能。不同的按钮注册不同的监听器将会产生不同的事件响应效果。

本章介绍了设计 Java 图形用户界面应用程序的基本方法，并阐述了 Java 提供的事件处理机制，其中运用了 Java API 提供的大量标准类。良好的重用性使得人们可以轻而易举地开发出操作友好的应用程序。通过本章列举的 18 个示例，不但可以了解 Java 图形用户应用程序的设计过程，还可以体会面向对象程序设计方法带来的好处。

小结

本章主要介绍 Swing 组件的使用。Swing 组件是在 AWT 组件基础上发展起来的轻量级 GUI 组件，它完全实现了 GUI 组件的平台无关性。一个 GUI 系统通常包括容器、原子控件等一些组件，布局管理器，事件处理模式以及一些控制整个界面观感的高级控件。Swing 所提供的 GUI 系统也包括这些部分。

在构建 GUI 时，主要要完成的事情是根据设计阶段对主画面的设计，创建各种组件，使用合适的布局管理器直接或间接添加到顶层容器的内容窗格。在创建组件时，可为组件注册合适的事件监听器，并定义类实现相应的事件监听器接口，提供合适的事件处理程序完成应用程序的功能。

Swing 提供了包括画框、对话框及 JApplet 在内的顶层容器，也提供了多种中间容器。Swing 提供的包括标签、按钮、检查框、组合框、列表、文本字段、文本区域等。本章介绍了其中一些组件的简单用法。

Swing 提供各种布局管理器管理组件的布局，利用布局管理器以及容器的嵌套放置可构建美观的图形用户界面。

Swing 的事件处理模式涉及事件源、事件及事件监听器。事件源通常是与用户进行交互的 GUI 组件，或者是组件所依赖的数据模型。事件是携带信息的对象，当 GUI 组件与用户进行交互时，或者组件所依赖的数据模型发生各种变化时都会激发合适类型的事件，这些事件由事件分发线程进行传播。事件监听器可通过事件对象获得有关事件源等信息，事件监听器的事件处理程序利用这些信息完成应用程序的功能。

习题

一、基本概念

1、阐述 Java 图形用户界面的设计过程。

2、阐述 Swing 组件的主要特点。

3、阐述容器和组件之间的关系。

4、阐述布局管理器的概念，以及各种布局管理器的布局策略。为什么 Java 要提出这个概念？

5、阐述 Java 事件处理的基本过程。

6、Java 采用“委托”事件处理机制有什么好处？举例说明。

二、编程实践

1、设计一个输入身份证信息的用户界面。当用户提交输入的信息后，弹出一个对话框，并将输入的内容显示在其中。

2、编写一个程序，用户可以在文本框时输入数字字符，按 Enter 键后将数字放入一个文本区。当输入的数字大于 2000 时，弹出一个对话框，提示用户数字已经大于 2000，问是否继续将该数字放入文本区。

3、请编写程序使用 GUI 让用户录入人员信息，人员信息包括：姓名、年龄、性别、籍贯（所属省份）、地址以及个人简介。请充分利用 Swing 提供的 GUI 组件，设计美观的界面，实现人员信息的录入。

4、设计一个排雷游戏的用户界面。

第 10 章 Java 的输入/输出

使用任何语言编写的程序都会涉及到输入/输出（I/O）操作。在 Java 语言中，输入/输出操作是使用流来实现的。流（Stream）是指数据在计算机各部件之间的流动，它包括输入流与输出流。输入流（Input Stream）表示从外部设备（键盘、鼠标、文件等）到计算机的数据流动；输出流(Output Stream)表示从计算机到外部设备(屏幕、打印机、文件等)的数据流动。Java 的输入/输出类库 java.io 包提供了若干输入流类和输出流类。利用输入流类可以建立输入流对象，利用输入流类提供的成员方法可以从输入设备上将数据读入到计算机中；利用输出流类可以建立输出流对象，利用输出流类提供的成员方法可以将程序中产生的数据写到输出设备上。

Java.lang 包中的 System 类有两个重要的类成员：in 和 out，分别是输入流和输出流类型的对象。in 的源是键盘，out 的目的地是命令行窗口。in 可以读取用户从键盘输入的数据，in 在读取数据时会引起阻塞，直到用户按 Enter 键。

基于文本的 I/O 都是一些人们能够阅读的字符（比如说程序的源代码），而基于数据的 I/O 是二进制（比如说表示图像的位图）。java.io 包将数据流分为两类：字符流（Character stream）和字节流（Byte stream）。

字节流用于读/写字节类型的数据(包括 ASCII 表中的字符)。字节流类可分为表示输入流的 InputStream 类及其子类，表示输出流的 OutputStream 类及其子类。

字符流用于读/写 Unicode 字符。它包括表示输入流的 Reader 类及其子类，表示输出流的 Writer 类及其子类。

10.1 File 类

File 类的对象主要用来获取文件本身的一些信息，例如文件所在的目录，文件的长度，文件读写权限等，不涉及对文件的读写操作。

创建一个 File 对象的构造方法有 3 个：

File(String filename);

File(String directoryPath, String filename);

File(File f, String filename);

其中，filename 是文件名字，directoryPath 是文件的路径，f 是指定成一个目录的文件。

使用 File(String filename)创建文件时，该文件被认为是与当前应用程序在同一目录中。

1. 文件的属性

经常使用 File 类的下列方法获取文件本身的一些信息：

- public String getName() 获取文件的名字。
- public boolean canRead() 判断文件是否是可读的。
- public boolean canWrite() 判断文件是否可被写入。
- public boolean exits() 判断文件是否存在。
- public long length() 获取文件的长度（单位是字节）。
- public String getAbsolutePath() 获取文件的绝对路径。
- public String getParent() 获取文件的父目录。
- public boolean isFile() 判断文件是否是一个正常文件，而不是目录。
- public boolean isDirectroy() 判断文件是否是一个目录。
- public boolean isHidden() 判断文件是否是隐藏文件。

例 10.1 演示使用上述的一些方法获取文件信息

```
import java.io.*;
public class Example10_1 {
public static void main(String[] args) throws IOException{
```

```
        File f1=new  File("E:\\chapter10","Example10_1.java");
        File f2=new File("E:\\chapter11");
        System.out.println(f1.getName()+"是可读的吗:"+f1.canRead());
        System.out.println(f1.getName()+"的长度:"+f1.length());
        System.out.println(f1.getName()+"的绝对路径:"+f1.getAbsolutePath());
        System.out.println(f2.getName()+"是目录吗？"+f2.isDirectory());
        }
}
```

2. 目录

（1）创建目录

File 对象调用方法 public boolean mkdir()创建一个目录，如果创建成功返回 true，否则返回 false（如果该目录已经存在将返回 false）。

（2）列出目录中文件的方法

如果 File 对象是一个目录，那么该对象可以调用下述方法列出该目录下的文件和子目录：

- public String[] list( )　用字符串形式返回目录下的全部文件。
- public File [] listFiles( )　用 File 对象形式返回目录下的全部文件。

有时需要列出目录下指定类型的文件，比如：.java、.txt 等扩展名的文件。可以使用 File 类的下述两个方法，列出指定类型的文件；

- public String[] list(FilenameFilter obj)　该方法用字符串数组形式返回目录下的指定类型的所有文件。
- public File[] listFiles(FilenameFilter obj)　该方法用 File 对象形式返回目录下的指定类型的所有文件。

FilenameFilter 是一个接口，该接口有一个方法：

```
public boolean accept(File dir,String name);
```

使用 list 方法时，须向该方法传递一个实现 FilenameFilter 接口的对象。List 方法执行时，参数不断回调接口方法 accept(File dir, String name)，accept 中的参数 name 被实例化为目录中的一个文件名，参数 dir 为调用 list 的当前目录对象，当接口方法返回 true 时，list 方法就将目录 dir 中的文件放到返回的数组中。

例 10.2 列出 e:\chaper10 目录下扩展名为.java 文件的名字

```
import java.io.*;
class FileAccept implements FilenameFilter
{  private String extendName;
   public void setExtendName(String s) {
      extendName="."+s;
   }
//实现FilenameFilter接口中的accept方法
public boolean accept(File dir,String name) {
      return name.endsWith(extendName);
   }
}
public class Example10_2 {
   public static void main(String args[]) {
      File dir=new File("E:\\chapter10");
      FileAccept fileAccept=new FileAccept();
```

```
        fileAccept.setExtendName("java");
        String fileName[]=dir.list(fileAccept);
        for(String name:fileName) {
            //遍历数组fileName中的每个分量并赋给name。
            System.out.println(name);
        }
    }
}
```

3. 文件的创建与删除

当使用 File 类创建一个文件对象后，例如：

File f=new File(“d:\\ chaper10”， “test.txt”);

如果 d:\ chaper10 目录中没有名字为 test.txt 文件，文件对象调用方法：

◆ public boolean createNewFile()

可以在 C:\myletter 目录中建立一个名字为 test.txt 的文件。

文件对象 f 调用方法：

◆ public Boolean delete()

可以删除当前文件，例如：

f.delete();

在下面的例子 10.2 中，列出 d:\chapter10 目录下扩展名是.java 文件的名字，并删除了 d:\chapter10 中的 Test.java 文件。

例 10.3 首先检测 e:\chapter10 目录下是否有 test.txt 文件，如果没有，则创建一个，然后读取 test.txt 文件的属性，最后删除该文件。

```
import java.io.*;
public class Example10_3
{
    public static void main(String[] args) throws IOException{
        File f = new File("E:\\chapter10\\test.txt");
        if(!f.exists()) f.createNewFile();
        System.out.println("");
        System.out.println(f);
        System.out.println("文件存在吗: " + f.exists());
        System.out.println("文件名:    " + f.getName());
        System.out.println("相对路径:   " + f.getPath());
        System.out.println("绝对路径:   " + f.getAbsolutePath());
        System.out.println("父目录:     " + f.getParent());
        System.out.println("是文件吗 :   " + f.isFile());
        System.out.println("是目录吗:    " + f.isDirectory());
        System.out.println("长度:       " + f.length());
        System.out.println("可读吗:     " + f.canRead());
        System.out.println("可写吗 :    " + f.canWrite());
        System.out.println("最后修改时间:" + f.lastModified());
        f.delete();
        System.out.println(f + " exist? " + f.exists());
    }
```

}

10.2 字节流

用字节文件输入输出流完成磁盘文件的读写时，首先要利用文件名字符串或 File 对象创建输入/输出流对象，其次是从文件输入/输出流中读写数据，最后要关闭流。

10.2.1 FileInputStream 类

该类是从 InputStream 中派生出来的简单输入类。FileInputStream 类的常用构造方法如下：

- FileInputStream(String name)
- FileInputStream(File file)

第一个构造器使用给定的文件名 name 创建一个 FileInputStream 对象。第二个构造器使用 File 对象创建 FileInputStream 对象。

从一个文件读取信息的操作分为以下 4 几个步骤：

1. 构造文件输入流

我们将要建立的许多程序都需要从文件中检索信息。文件输入流（输入流的子类）提供对文件的读取，使用文件输入流构造器来打开一个到达该文件的输入流（源就是这个文件，输入流指向这个文件）。

例如，为了读取一个名为 myfile.dat 的文件，建立一个文件输入流对象，代码如下：

```
FileInputStream istream = new FileInputStream("myfile.dat");
```

或

```
File f = new File("myfile.dat");
FileInputStream istream = new FileInputStream(f);
```

2．处理 I/O 异常

当使用文件输入流构造器建立通往文件的输入流时，可能会出现错误（也被称为异常）。例如，试图要打开的文件可能不存在时就会出现 I/O 错误，Java 使用一个 IOException 对象来表示这个出错信号。程序必须使用一个 try-catch 块检测并处理这个异常。例如，为了把一个文件输入流对象与一个文件关联起来，可以使用类似于下面所示的代码：

```
try{
        FileInputStream ins = new FileInputStream("myfile.dat");
}
catch (IOException e ){
        System.out.println("File read error:" +e );
}
```

3．从输入流中读取字节

输入流的唯一目的是提供通往数据的通道，程序可以通过这个通道读取数据。read 方法给程序提供一个从输入流中读取数据的功能。

read 方法的格式如下

- int read() 从输入流中读取单个字节数据，返回字节值（0~255 之间的一个整数）；
- int read(byte b[]) 从输入流中读取数据并存放在字节数组 b 中
- int read(byte b[], int off, int len) off 参数指定 read 方法把数据存放在字节数组 b 中的什么地方，len 参数指定该方法将读取的最大字节数

read 过程中，当到达输入流的末尾时，则返回-1。

FileInputStream 流顺序地读取文件，只要不关闭流，每次调用 read 方法就都顺序地从输入流中读取剩其余的内容，直到流的末尾或流被关闭为止。

4．关闭流

虽然 Java 在程序结束时自动关闭所有打开的流，但是当我们使用完流后，应该显式地关闭任何打开的流仍是一个良好的习惯。一个被打开的流可能会用尽系统资源，这取决于平台和实现。如果没有关闭

那些被打开的流，那么在这个或另一个程序试图打开另一个流时，这些资源可能得不到。关闭输出流的另一个原因是把该流缓冲区的内容冲洗掉（通常冲洗到磁盘文件上）。我们知道，在操作系统把程序所写到输出流上的那些字节保存到磁盘上之前，有时被存放在内存缓冲区中，通过调用 close 方法，可以保证操作系统把流缓冲区的内容写到它的目的地。

例 10.4 读取指定文件，并将所读内容显示在屏幕上

```
import java.io.*;
public class Example10_4 {
   public static void main(String args[]) {
      int b;
      byte tom[]=new byte[25];
      try{
           File f=new File("e:\\chapter10\\Example10_4.java");
           FileInputStream in=new FileInputStream(f);
           while((b=in.read(tom,0,25))!=-1) {
              String s=new String (tom,0,b);
              System.out.print(s);
           }
           in.close();
        }catch(IOException e) {
           System.out.println("File read Error"+e);
        }
   }
}
```

10.2.2 FileOutputStream 类

与 FileInPutStream 类相对应的是 FileOutputStream 类。FileOutputStream 类提供了基本的文件写入能力。除了从 OutputStream 类继承来的方法外，FileOutStream 类还有两个构造方法：

- FileOutputStream(String name）
- FileOutputStream(File file)

第一个构造器使用给定的文件名 name 创建一个 FileOutputStream 对象。第二个构造器使用 File 对象创建 FileOutputStream 对象。

可以使用 write 方法把字节发送给输出流。

write 的格式如下：

- public void write(byte b[])　　写 b.length 个字节写到输出流。
- public void.write(byte b[], int off, int len)　从给定字节数组 b 中起始于偏移量 off 处写 len 个字节到输出流，参数 b 是存放了数据的字节数组，off 是数据的起始偏移量，len 是要输出的字节数。

FileOutStream 流顺序地写文件，只要不关闭流，每次调用 writer 方法就顺序地向文件写入内容，直到流被关闭或写完数据。

例子 10.5 将一段文本写入到文件 test.txt 中

```
import java.io.*;
public class Example10_5 {
   public static void main(String args[]) {
      byte [] b="Hello world".getBytes();
      try{
          FileOutputStream out=new FileOutputStream("test.txt");
```

```
            out.write(b);      //将Hello world写入test.txt中
            out.write(b,6,5);  //将world写入test.txt中
            out.close();
        }
        catch(IOException e) {
            System.out.println("Error "+e);
        }
    }
}
```

可以使用记事本或其他文本编辑器查看 test.txt，该文件保存在和 Example10_5.class 的同一目录中。

10.3 字符流

与 FileInputStream、FileOutputStream 字节流相对应的是 FileReader、FileWriter 字符流，其分别是 Reader 和 Writer 的子类，构造方法分别如下：

- FileReader (String filename)
- FileWriter (String filename)

FileInputStream 按字节读取文件，字节流不能直接操作 Unicode 字符，所以 Java 提供了字符流。由于汉字在文件中占用 2 个字节，如果使用字节流，读取不当会出现乱码现象。采用字符流就可以避免这个现象，因为在 Unicode 中，一个汉字被看作一个字符。

FileReader 流顺序地读取文件，只要不关闭流，每次调用 read 方法就顺序的读取源中其余的内容，直到源的末尾或流被关闭。

例 10.6 复制文件

```
import java.io.*;
public class Example10_6
{
    public static void main(String[] args) throws IOException
    {
        // 构造输入流对象
        File file = new File("e:\\chapter10\\Example10_6.java");
        try{
            FileReader fin = new FileReader(file);
            FileWriter fout = new
FileWriter("e:\\chapter10\\copy-of-file.txt");
            System.out.println("Current charset is :" + fin.getEncoding());
            //分3次读完,每次读取n个字符
            int n = (int) (file.length() / 3);
            System.out.println("前" + n + " 个字符用read()读");
            // 使用read()和write()
            for (int i = 0; i < n; i++)
            {
                fout.write(fin.read());
            }
            System.out.println("接下来的 " + n + " 用 read(b[])读");
            // 使用read(char[]b )和write(char[] b);
            char b[] = new char[n];
```

```
            if (fin.read(b) != n)
            {
                System.err.println("读 " + n + " bytes时出错");
            }
            fout.write(b);
            System.out.println("用 read(b[],offset,len)读剩余字符");
            // 使用read(b,offset,len)和write（b,offset,len）
            int count = 0;
            while ((count = fin.read(b, 0, n)) != -1)
                fout.write(b, 0, count);
            fin.close();
            fout.flush();
            fout.close();
        }catch(IOException e)
        { System.out.println(e);}
    }
}
```

上面的例子实现了把 Example10_6.java 文件的内容读出，并写入名为 copy-of-file.txt 的文件中。为了充分说明读写方法，该例子将源文件内容分为三部分，每部分用不同的方法读出并写入，在实际复制文件中，选以上三种读写方法中的任意一种即可。

如果我们想每次必须读取一行，FileReader 类没有提供这种方法，但 Java 提供了名为 BufferedReader 的类能够实现按行读取，其构造方法如下：

◆ BufferedReader(Reader in)

方法 readLine()按行读取文本。

通过向 BufferedReader 传递一个 Reader 对象（如 FileReader 的实例），来创建一个 BufferedReader 对象，如

FileReader inOne=new FileReader("Student.txt")

BufferedReader inTwo= BufferedReader(inOne);

类似地，可以将 BufferedWriter 流和 FileWriter 流连接在一起，然后使用 BufferedWriter 流将数据写到目的地，例如：

FileWritertofile=new FileWriter("hello.txt");

BufferedWriter out= BufferedWriter(tofile);

然后 out 使用 BufferedReader 类的方法:

write(String s，int off，int len) 方法把字符串 s 写到文件 hello.txt 中，参数 off 是 s 开始处的偏移量，len 是写入的字符数量。

例 10.7 使用 BufferedReader 和 BufferedWriter 复制文件

```
import java.io.*;
public class Example10_7
{
    public static void main(String[] args) throws IOException
    {
        // 构造输入流对象
        String temp;
        File   sourceFile,targetFile;
```

```
        BufferedReader source;
        BufferedWriter target;
        try{
            sourceFile = new File("e:\\chapter10\\Example10_7.java");
            targetFile = new File("e:\\chapter10\\copy-of-file2.txt");
            source = new BufferedReader(new FileReader(sourceFile));
            target = new BufferedWriter(new FileWriter(targetFile));
            while((temp=source.readLine())!=null){
                target.write(temp);
                target.newLine();
                target.flush();
            }
            System.out.println("复制文件完成!!!");
            source.close();
            target.close();
        }catch(IOException e)
        { System.out.println(e);}
    }
}
```

10.4 RandomAccessFile 类

Java 还提供了专门用来处理文件输入输出操作、功能更完善的 RandomAccessFile 流。当用户需要严格地处理文件时，就可以使用 RandomAccessFile 类来创建一个对象，称为 RandomAccessFile 流。

RandomAccessFile 类创建的流与前面的输入、输出流不同， RandomAccessFile 类既不是输入流 InputStream 的子类，也不是输出流 OutputStream 的子类。RandomAccessFile 类创建的流既可以指向源也指向目的地，换句话说，当想对一个文件进行读写操作时，只需创建一个指向该文件的 RandomAccessFile 流即可，这样既可以从这个流中读取文件的数据，也可以通过这个流写入数据到文件。

RandomAccessFile 类的构造方法：

◆ RandomAccessFile(String name, string mode)

参数 name 用来确定一个文件名，既是流的源，同时也是流目的地。参数 mode 取 r（只读）或 rw（可读写），决定流对文件的访问权限。

需要注意的是，创建 RandomAccessFile 对象时，可能产生两种异常：当指定的文件不存在时，系统将抛出 FileNotFoundException 异常；若试图用读/写方式打开具有只读属性的文件或出现了其他输入/输出错误时，则会抛出 IOException 异常。

RandomAccessFile 类的常用成员方法如表 10-1 所示。

表 10-1 RandomAccessFile 类常用成员方法

成员方法	说明
native long getFilePointer()	取得文件的指针
native long length()	以字节为单位获取文件的大小
int read()	自输入流中读取一个字节
int read(byte b[])	将输入的数据存放在指定的字节数组中
int read(byte b[],int offset,int len)	自输入流的 offset 位置开始读取 len 个字节并存放在指定的数组中
void write(int b)	写一个字节
void write (byte b[])	写一个字节数组

void write (byte b[],int offset,int len)	将字节数组 b[]从 offset 位置开始,长度为 len 个字节数组的数据写到输出流中
native void close()	关闭数据流
native void seek(long pos)	将文件位置指针置于 pos 处，pos 以字节为单位

RandomAccessFile 类中有一个 seek(long a)方法，用来定位 RandomAccessFile 流的读写位置，其中参数 a 确定读写位置距离文件开头的字节个数。RandomAccessFile 流还可以调用 getFilePointer( )方法获取流的当前读写位置。RandomAccessFile 类对象的文件位置指针遵循以下规律：

（1）新建 RandomAccessFile 对象文件位置指针位于文件的开头处。

（2）每次读写操作之后，文件位置指针都相应后移读写的字节数。

（3）利用 seek()方法可以移动文件位置指针到一个新的位置。

（4）利用 getPointer()方法可获得本文件当前的文件位置指针。

（5）利用 length()方法可得到文件的字节长度。利用 getPointer()方法和 length()方法可以判断读取的文件是否到文件尾部。

例 10.8 从键盘输入五个整数写入文件 t1.txt 中，再从这个文件中随机读出其中的某个数（由键盘输入确定），将它显示在屏幕上，同时允许用户对这个数进行修改。

```
import java.io.*;
public class Example10_6
{
    public static void main(String args[])
    {
        int  num,a;
        long fp;
        try{
            InputStreamReader din=new InputStreamReader(System.in);
            BufferedReader in=new BufferedReader(din);
            RandomAccessFile rf=new RandomAccessFile(args[0],"rw");
            System.out.println("请输入五个整数");
            int b[]=new int[5];
            for(int i=0;i<5;i++){
                System.out.print("第" + (i+1)+"个数");
                b[i]=Integer.parseInt(in.readLine());
                rf.writeInt(b[i]);
            }
            while(true){
                fp=0;
                rf.seek(0);
                System.out.print("请输入要显示第几个数（1-5）: ");
                num=Integer.parseInt(in.readLine());
                num=num-1;
                fp=(num)*4;
                rf.seek(fp);
                a=rf.readInt();
                System.out.println("第"+(num+1)+"个数是: "+a);
                System.out.print("改写此数 ");
```

```
                b[num]=Integer.parseInt(in.readLine());
                fp=num*4; rf.seek(fp);
                rf.writeInt(b[num]);
                System.out.print("继续吗?(y/n) ");
                if((in.readLine()).equals("n")) break;
            }
            rf.close();
        }catch(Exception e)
         {System.out.println("I/O错误!"+e.getMessage());}
    }
}
```

运行结果示例如图10_1所示：

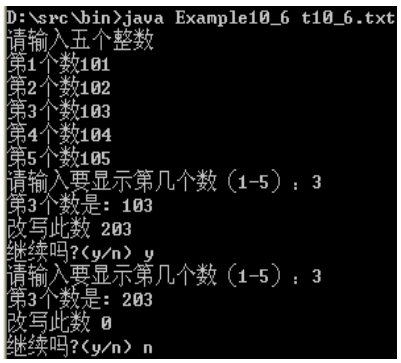

图 10_1 例 10.8 运行结果

10.5 数据流

DataInputStream 类和 DataOutputStream 类创建的对象被称为数据输入流和数据输出流。这两个流允许程序按与计算机无关的风格读写原始数据。也就是说，当读取一个数值时，不必再关心这个数值应当是多少个字节。

DataInputStream 类和 DataOutputStream 类的构造方法：

- DataInputStream(InputStream in) 将创建的数据输入流指向一个由参数 in 指定的输入流，以便从后者读取数据
- DataOutputStream(OutputStream out) 将创建的数据输出流指向一个由参数 out 指定的输出流，然后通过这个数据输出流把 java 数据类型的数据写到输出流 out。

1. DataInputStream 类的成员方法

DataInputStream 类是抽象类 InputStream 的子类，主要成员方法见表 10-2 所示。

表 10-2 DataInputStream 类的成员方法

成员方法	说明
int read(byte b[]) throws IOException	从输入流中将数据读取到数组 b 中
int read(byte b[], int offset,int len)throws IOException	从输入流中读取 len 字节的数据到数组 b 中，在数组中从 offset 位置开始存放
void readFully(byte b[])throws IOException	读取输入流中的所有数据到数组 b 中
void readFully(byte b[],int offset,int len)throws IOException	读取输入流中的所有数据到数组 b 中，在数组 b 中从 offset 位置开始存放 len 字节
int skipBytes(int n)throws IOException	读操作跳过 n 个字节，返回真正跳过的字节数
boolean readBoolean() throws IOException	读 1 个布尔值

byte readByte()throws IOException	读 1 个字节
int readUnsignedByte()throws IOException	读取一个 8 位无符号数
short readShort() throws IOException	读取 16 位短整型数
int readUnsignedShort() throws IOException	读取 16 位无符号短整型数
char readChar()throws IOException	读 1 个 16 位字符
int readInt( )throws IOException	读 1 个 32 位整数数据
long readLong( )throws IOException	读 1 个 64 位长整数数据
float readFloat()throws IOException	读 1 个 32 位浮点数
double readDouble() throws IOException	读 1 个 64 位双字长浮点数
String readLine()throws IOException	读 1 行字符
String readUTF() throws IOException	读 UTF(UnicodeTextFormat)文本格式的字符串，返回值即是读得的字符串内容。

2. DataOutputStream 类成员方法

由于 OutputStream 是抽象类，程序中创建的输出流对象隶属于 OutputStream 类的某个子类。表 10-3 列出了 OutputStream 的子类 DataOutputStream 类的成员方法。

表 10-3　DataOutputStream 主要成员方法

成员方法	说明
void write(int b)throws IOException	向输出流写一个字节
void write(byte b[], int off,int len) throws IOException	将字节数组 b[]从 off 位置开始的 len 个字节写到输出流
void writeBoolean(boolean v) throws IOException	将指定的布尔数据写到输出流
void writeByte(int final void writeByte(int v) throws IOException	将指定的 8 位字节写到输出流
void writeShort(int v) throws IOException	将指定的 16 位短整数写到输出流
void writeChar(int v) throws IOException	将指定的 16 位 Unicode 字符写到输出流
void writeInt(int v) throws IOException	将指定的 32 位整数写到输出流
void writeLong(long v)throws IOException	将指定的 64 位长整数写到输出流
void writeFloat(float v)throws IOException	将指定的 32 位实数写到输出流
void writeDouble(double v)throws IOException	将指定的 64 位双精度数写到输出流
void writeBytes(String s)throws IOException	将指定的字符串按字节数组写到输出流
void writeChars(String s)throws IOException	将指定的字符串作为字符数组写到输出流
void writeUTF(String str)throws IOException	将指定的字符串按 UTF 格式的字符数组写到输出流
int size()	返回所写的字节数
void flush ( ) throws IOException	将缓冲区的所有字节写到输出流

例 10.9 写几个 Java 类型的数据到一个文件中然后读出显示到屏幕上。

```
import java.io.*;
public class Example10_9
{
    public static void main(String[] args)
    {
        boolean b = false;
```

```
        int i = 20;
        char c = '长';
        double d = 1.1;
        String s = "长江师范学院";
        String filename = "e:\\chapter10\\example.dat";
        FileOutputStream fout = null;
        FileInputStream fin = null;
        DataInputStream din = null;
        DataOutputStream dout = null;
        try
        {
            fout = new FileOutputStream(filename);
            dout = new DataOutputStream(fout);
            // 把数据写入文件
            dout.writeBoolean(b);
            dout.writeInt(i);
            dout.writeChar(c);
            dout.writeDouble(d);
            dout.writeUTF(s);
            dout.flush();
            dout.close();
            fout.close();
            // 把数据从文件中读出来
            fin = new FileInputStream(filename);
            din = new DataInputStream(fin);
            System.out.println(din.readBoolean());
            System.out.println(din.readInt());
            System.out.println(din.readChar());
            System.out.println(din.readDouble());
            System.out.println(din.readUTF());
        } catch (FileNotFoundException ex)
        {
            System.out.println("文件不存在.");
            System.out.println(ex);
        } catch (IOException ex)
        {
            System.out.println("IO异常.");
            System.out.println(ex);
        }
    }
}
```

10.6 对象流与序列化

1. ObjectInputStream 类和 ObjectOutputStream 类

ObjectInputStream 类和 ObjectOutputStream 类分别是 InputStream 类和 OutputStream 类的子类。

ObjectInputStream 类和 ObjectOutputStream 类创建的对象被称为对象输入流和对象输出流。对象输出流使用 writeObject(Object obj)方法将一个对象 obj 写入输出流送往目的地，对象输入流使用 readObjct()方法从源中读取一个对象到程序中。

ObjectInputStream 类和 ObjectOutputStream 类的构造方法分别如下：

- ObjectInputStream(InputStream in)
- ObjectOutputStream(OutputStream out)

ObjectOutputStream 的指向应当是一个输出流对象，因此，当准备将一个对象写入到文件时，首先用 FileOutputStream 创建一个文件输出流，代码如下：

FileOutputStream file_out=new FileOutputStream(“tom.txt”);

ObjectOutputStream object_out=new ObjectOutputStream(file_out);

同样 ObjectInputStream 的指向应当是一个输入流对象，因此，当准备从文件中读入一个对象到程序中时，首先用 FileInputStream 创建一个文件输入流，代码如下：

FileInputStream file_in=new FileInputStream("tom.txt");

ObjectInputStream object_in=new ObjectInputStream(file_in);

当使用对象流写入或读入对象时，要保证对象是序列化的。所谓对象序列化就是将对象的状态转换成字节流，以后可以通过这些值再生成相同状态的对象。一个类如果实现了 Java.io.Serializable 接口，那么这个类创建的对象就是所谓序列化的对象。Java 对象序列化不仅保留一个对象的数据，而且递归保存对象引用的每个对象的数据，将整个对象层次写入字节流中，从而保证能把对象写入到文件或在网络上传递，并能再把对象正确读回到程序中。

Serializable 接口中的方法对程序是不可见的，因此，实现该接口的类并不需要实现额外的方法。当把一个序列化的对象写入到对象输出流时，JVM 就会实现 Serializable 接口中的方法，将一定格式的文本——对象的序列化信息，写入目的地。

需要注意的是：使用对象流把一个对象写入到文件时不仅要保证该对象是序列化的，而且该对象的成员对象也必须是序列化的。当 ObjectInputStream 对象流从文件或网络上读取对象时，就会读回对象的序列化信息，并根据对象的序列化信息创建一个对象

例 10.10

```
import java.io.*;
public class Example10_10
{
    public static void main(String args[]) throws Exception
    {
        Student stu = new Student(20081064, "张三丰", 20, "数学与计算机学院");
        FileOutputStream fout = new
FileOutputStream("e:\\chapter10\\data1.dat");
        ObjectOutputStream oout = new ObjectOutputStream(fout);
        // 将序列化对象写入文件
        oout.writeObject(stu);
        oout.close();
        stu = null;
        FileInputStream fin = new FileInputStream("e:\\chapter10\\data1.dat");
        ObjectInputStream oin = new ObjectInputStream(fin);
        // 从文件读入序列化对象
        stu = (Student) oin.readObject();
        oin.close();
```

```
            System.out.println("学号:      " + stu.id);
            System.out.println("姓名:      " + stu.name);
            System.out.println("年龄:      " + stu.age);
            System.out.println("院系:      " + stu.department);
        }
    }
    class Student implements Serializable
    {
        private static final long serialVersionUID = 1L;
        int id;
        String name;
        int age;
        String department;
        public Student(int id, String name, int age, String department)
        {
            this.id = id;
            this.name = name;
            this.age = age;
            this.department = department;
        }
    }
```

小结

程序的输入输出就是程序与外部的信息交互，在操作系统中使用文件来抽象外部设备与磁盘文件，而在 Java 程序中使用流来抽象程序与外部进行交互的通道。流使得 Java 程序可以统一地处理输入输出，不必关心流具体关联到什么输入源或输出目标。

Java API 提供了面向字节的流和面向字符的流，面向字节的流以字节作为输入输出的基本单位，可处理任意格式的输入输出数据，但需要程序自己负责解释输入输出数据的格式。

面向字符的流以字符作为输入输出的基本单位，能处理各种编码集的字符数据，从而能正确处理中文、日文等非英文的数据信息。

文件流是最基本的流，它用于文件的输入输出操作，通常面向字节的文件流用于处理二进制文件，而面向字符的文件流用于处理文本文件。文件流用于操纵文件的内容，而辅助类 File 用于操作文件本身或文件目录。文件流只能顺序读写文件，但类 RandomAccessFile 可对文件进行随机读写。

DataInputStream 类和 DataOutputStream 类创建的对象被称为数据输入流和数据输出流。这两个流允许程序按与计算机无关的风格读写原始数据。

对象流用于将对象串行化和反串行化，可支持轻量级的对象持久性。对象串行化实际上是将内存中的对象图映射到输入输出流的字节序列。声明实现 Serializable 接口的对象都可串行化，串行化的细节由 Java API 提供的类和方法进行维护，不过程序员也可定制对象的串行化行为。对象串行化是一件十分复杂的工作，程序员需要慎重对待。

习题

一、基本概念

1、文本文件与二进制文件有什么区别？

2、阐述 Java 是如何处理以字节为单位和以字符为单位的输入输出流的。

3、阐述如何利用 File 类对文件类进行处理？

4、操作文件时使用缓冲与不使用缓冲有什么区别？

5、阐述如何利用 DataInputStream 类和 DataOutputStream 将 Java 提供的基本数据类型的数值直接写入文件或从文件中读出？

6、什么是对象串行化？对象串行化有什么用处？如何让对象具有串行化能力？

二、编程实践

1、编写一个程序，从给定文件中读取一个整数序列，并将其进行排序，然后再写入另外一个文件中。

2、编写程序，统计给定文件中每个字母出现的频率。

3、创建一个文本文件，并编写程序统计其中包含的单词数目。

4、请设计一个图书类，并编写一个程序，将各本图书的信息写入给定文件中。

5、改写编程第 4 题中设计的图书类，使其具有串行化功能。

6、编写一个程序比较两个文件是否完全相同，列出它们不同之处的位置与各自的内容，源文件与目标文件的参数应由命令行给出。

7、编写三个类 A, B, C，让类 A 有一个域指向类 B 的对象，类 B 有一个域指向类 C 的对象，类 C 有一个域指向类 A 的对象。创建三个对象，让它们循环引用，将其中任意一个对象写入文件，再读出，考察 Java 语言的对象串行化机制是否能正确恢复对象引用关系。

8、设计一个雇员类 Employee，其属性包括工号、姓名、部门、年龄、工资等，请读入用户输入的一个雇员信息，将其存入一个二进制文件，然后再将其读出并显示在屏幕上。

第 11 章 网络通信

Java 语言能够风靡全球的重要原因之一就是它和网络的紧密结合。作为网络编程语言，Java 可以很方便地将 Applet 嵌入网络的主页中，也可以实现客户端和服务器端的通信，并且通信可以是多客户的。

Java 语言使用基于套接字的网络通信方式。这种套接字的网络通信方式分为流套接字和数据报套接字两种。流套接字网络通信方式使用的协议是传输控制协议(Transmission Control Protocol，TCP)，它提供一种面向连接的高可靠性的传输。利用它进行通信，首先需要建立连接，如同我们打电话，接通电话后才能说话。数据报套接字网络通信方式使用的协议是 UDP(User Datagram Protocol)，它是一种无连接，高效率，但不十分可靠的协议，利用它进行通信如同寄信，无需建立连接就可以进行。

Java 语言通过软件包 java.net 实现三种网络通信模式：URL 通信模式(在使用 URL 通信模式时，它的底层仍使用流套接字方式)；Socket 通信模式(也称为流套接字通信模式)及 Datagram 通信模式(也称为数据报套接字通信模式)。

11.1 URL 通信

11.1.1 URL 通信基础

URL(Uniform Resource Locator)是统一资源定位器的简称，它表示 Internet/Intranet 上的资源位置。这些资源可以是一个文件、一个目录或一个对象。当我们使用浏览器浏览网络上的资源时，首先需要键入 URL 地址，才可以访问相应的主页。例如：

http://www.sun.com/books/java_series.html

file//c:/ABC/xx.java

每个完整的 URL 由个以下几部分组成：

传输协议://主机地址:端口号/文件所在路径及名字

一般的通信协议都已经规定好了开始联络时默认的通信端口号，例如，HTTP 协议的缺省端口号是 80，FTP 协议的缺省端口号是 21 等。URL 使用协议的缺省端口号时，可以不写出缺省端口号。所以，一般的 URL 地址只包含传输协议、主机名和路径和文件名就足够了。

网络通信中，我们常常会碰到地址(Address)和端口(Port)的问题。两个程序之间只有在地址和端口方面都达成一致时，才能建立连接。这与我们寄信要有地址、打电话要有电话号码一样。两个远方程序建立连接时，首先需要知道对方的地址或主机名，其次是端口号。地址主要用来区分计算机网络中的各个计算机，而端口的定义可以理解为扩展的号码，具备一个地址的计算机可以通过不同的端口来与其他计算机进行通信。

TCP 协议中，端口被规定为一个在 0～65535 之间的 16 位的整数。其中，0～1023 被预先定义的服务通信占用(如 FTP 协议的端口号是 21，http 协议的端口号为 80 等)。除非我们需要访问这些特定服务，否则，我们就应该使用 1024～65535 这些端口中的某一个来进行通信，以免与预先定义的服务端口发生冲突。

11.1.2 URL 类

要使用 URL 进行网络编程，必须创建 URL 对象。创建 URL 对象要使用 java.net 软件包中提供的 java.net.URL 类的构造方法。

1．创建 URL 对象

URL 类提供了用于创建 URL 对象的构造方法有 4 个：

（1）public URL(String spec) throws MalformedURLException

这个构造方法使用 URL 的字符串 spec 来创建一个 URL 对象。若字符串 spec 中使用的协议是未知的，则抛出 MalformedURLException 异常，在创建 URL 对象时必须捕获这个异常。例如：

URL file=new URL(“http://www.chd.edu.cn/index.html")

这种以完整的 URL 创建的 URL 对象称为绝对 URL，该对象包含了访问该 URL 所需要的全部信息。

（2）public URL(String protocol,String host,String file)　throws MalformedURLException

这个构造方法用指定的 URL 的协议名、主机名和文件名创建 URL 对象。参数中的 protocol 为协议名；host 为主机名；file 为文件名；端口号使用缺省值。若使用的协议未知的则仍会抛出 MaiformedURLException 异常。例如：

URL url = new URL("http","www.yznu.cn ","index.html");

（3）public URL(String protocol,String host,int port,String file) throws MalformedURLException

这个构造方法与构造方法（2）相比，增加了 1 个指定端口号的参数。例如：

URL url = new URL("http","www.yznu.cn ",80,"index.html");

（4）public URL(URL context,String spec) throws MalformedURLException

这个构造方法基于一个已有的 URL 对象创建一个新的 URL 对象，多用于访问同一个主机上不同路径的文件，例如：

URL u=new URL(“http://java.sun.com:80/docs/books/”);

URL u1=new URL(u, ”tutorial.intro.html”);

URL u2=new URL(u, ”tutorial.super.html”);

2．URL 类的常用成员方法

创建 URL 对象后，可以使用 java.net.URL 成员方法对创建的对象进行处理。URL 类提供了多个方法获取 URL 对象的状态，从而可帮助程序员从一个字符串描述的 URL 地址中提取协议、主机、端口号、文件名等信息。常用的 java.net.URL 成员方法表示如表 11-1：

表 11-1 java.net.URL 类的常用成员方法

成员方法	说明
int getPort()	获取端口号。若端口号未设置，返回-1
String getProtocol()	获取协议名。若协议未设置，返回 null
String getHost()	获取主机名。若主机名未设置，返回 null
String getFile()	获取文件名。若文件名未设置，返回 null
boolean equals(Object obj)	与指定的 URL 对象 obj 进行比较，如果相同返回 true，否则返回 false
final InputStream openStream()	获取输入流。若获取失败，则抛出一个 java.io.Exception 异常。
string toString	将此 URL 对象转换成字符串形式

例 11.1 创建 URL 对象并获取 URL 相关信息。

```
import java.net.*;
public class Example11_1 {
    public static void main(String[] args) throws Exception {
        URL url = new URL("http://java.sun.com:80/docs/books/"
+"tutorial/index.html#DOWNLOADING");
        System.out.println("协议：" + url.getProtocol());
        System.out.println("主机：" + url.getHost());
        System.out.println("端口：" + url.getPort());
        System.out.println("文件名：" + url.getFile());
        System.out.println("引用：" + url.getRef());
        System.out.println("默认端口：" + url.getDefaultPort());
    }
}
```

运行结果是：

```
协议：http
```

主机：java.sun.com

端口：80

文件名：/docs/books/tutorial/index.html

引用：DOWNLOADING

默认端口：80

11.1.3 通过字节流读取 WWW 资源

URL 对象创建后，就可以通过它来访问指定的 WWW 资源。这时需要调用 URL 类的 openStream()方法，该方法与指定的 URL 建立连接并返回一个 InputStream 类的对象，这样访问网络资源的操作就变成了我们熟悉的 I/O 操作，接下来就可以用字节流的方式读取资源数据。

例 11.2 创建一个 URL 对象，并通过该对象读取并显示 URL 地址标识的资源的内容

```
import java.net.*; import java.io.*;
public class Example11_2 {
    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            System.out.println("按以下格式输入：Example11_2 <URL地址>");
            return;
        }
        URL url = new URL(args[0]);
        BufferedReader in = new BufferedReader(new
InputStreamReader(url.openStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
        in.close();
    }
}
```

在控制台输入“java Example11_2 http://www.163.com”将显示 163 首页的文档内容，并未考虑资源的数据格式，而是直接将资源以一种字符流的形式读出并显示在屏幕上，并没有理会资源本身是一个 HTTP 文档、一个 PHP 程序还是一个 GIF 图片。

11.1.4 使用 HttpURLConnection 实现双向通信

实际应用中，只读取数据是不够的，很多情况下，我们都需要将一些信息发送到服务器中去，这就要求我们能够实现同网络资源的双向通信，URLConnection 类就是用来解决这一问题的。

类 URLConnection 也是定义在包 java.net 里，它表示 Java 程序和 URL 在网络上的通信连接。当与一个 URL 建立连接时，首先要在一个 URL 对象上通过方法 openConnection()生成对应的 URLConnection 对象。对于一个使用 HTTP 协议的 URL 连接，调用 openConnection()方法返回的是其具体类 HttpURLConnection 的一个实例。HttpURLConnection 提供了对 http 协议的支持。

1. 创建 URLConnection 类的对象

创建 URLConnection 对象分两步完成：第一步是创建一个 URL 对象；接下来调用该对象的 openConnection()方法就可以返回一个对应其 URL 地址的 URLConnection 对象。例如：

URL url=new URL("http://www.163.com");

HttpURLConnection connection=(HttpURLConnection)url.openConnection();

但是要注意，如果 URL 地址不是一个 http 地址，那么就无法用类型转换获取 HttpURLConnection 的实例。

2. 建立输入 / 输出数据流

读取或写入远方的计算机节点的信息时，首先要建立输入或输出数据流，利用 URLConnection 类的成员方法 getlnputStream()和 getOutputStream()来获取它的输入输出数据流。例如，下面的两行用于建立输入数据流：

InputStreamReader ins = new InputStreamReader(con.getInputStream());

BufferedReader in = new BufferedReader(ins);

下面的语句行建立输出数据流：

PrintStream out = new PrintStream(con.getOutputStream());

3. 读取远方的计算机节点的信息或向其写入信息

读取远方计算机节点的信息时，调用 in.readLine()方法；而向远方计算机节点写入信息时，调用 out.println(参数)方法。

例 11.3 使用 HttpURLConnection 向 URL 连接读写数据

```
import java.io.*;
import java.net.*;
public class Example11_3{
    public static void main(String[] args) throws Exception {
        // 建立指向本地磁盘上cgi的URL对象
        URL url = new URL("http://www.yznu.cn/cgi-bin/test");
        HttpURLConnection connection =
(HttpURLConnection)url.openConnection();
        connection.setDoOutput(true);
        PrintStream ps = new PrintStream(connection.getOutputStream());
        ps.println("0123456789");
        ps.close(); // 向服务器输出数据
        DataInputStream dis = new DataInputStream(connection.
getInputStream());
        String inputLine;
        while ((inputLine = dis.readLine()) != null) {
            System.out.println(inputLine);
        }
        dis.close();// 从服务器读数据
    }
}
```

上例所执行的功能依赖于服务端的 CGI 脚本。

11.2 Socket 通信

尽管类 URL 和 URLConnection 面向应用层的 HTTP、FTP、FILE 等协议提供了一个相当高层的因特网资源访问机制，使应用程序可方便地访问各种资源，但有时程序员也需要较低层的通信方式。基于 Socket 的通信为开发因特网上的客户机／服务器应用提供了可靠的数据通信手段。

Socket 套接字是应用于网络通信中的重要机制。Socket 最初是加利福尼亚大学 Berkeley 分校为 UNIX 操作系统开发的网络通信接口。随着 UNIX 操作系统的广泛使用，套接字成为当前最流行的网络通信应用程序接口之一。Java 语言中采用的 Socket 通信是一种流式套接字通信，它采用 TCP 协议，通过提供面向连接的服务，实现客户/服务器之间双向、可靠的通信。java.net 包中的 Socket 类与 ServerSocket 类对流式套接字通信方式提供了充分的支持为流式套接字通信方式提供了充分的支持。

1. Socke 的概念

Socket 称为“套接字”，也有人称为“插座”。在两台计算机上运行的两个程序之间有一个双向通信

的链接点，而这个双向链路的每一端就称为一个 Socket。

建立连接的两个程序分别称为客户端(Client)和服务器端(Server)。客户端程序申请连接，而服务器端程序监听所有的端口，判断是否有客户程序的服务请求。当客户程序请求和某个端口连接时，服务器程序就将“套接字”连接到该端口上，此时，服务器与客户程序就建立了一个专用的虚拟连接。客户程序可以向套接字写入请求，服务器程序处理请求并把处理结果通过套接字送回。通信结束时，再将所建的虚拟连接拆除。

一个客户程序只能连接服务器的一个端口，而一个服务器可以有若干个端口，不同的端口使用不同的端口号，并提供不同的服务。

2．Socket 通信机制

利用 socket 进行网络通信由三部分组成：

（1）建立 socket 连接：在通信开始之前由通信双方确认身份，建立一条专用的虚拟连接通道。

（2）数据通信：利用虚拟连接通道传送数据信息进行通信。

（3）关闭：通信结束时，再将所建的虚拟连接拆除。

利用 java.net 包中提供的 Socket 类和 ServerSocket 类及其方法，可完成上述操作。图 11_1 表示了 Socket 通信机制。从图中可以看到，服务器端的程序首先选择一个端口(port)注册，然后调用 accept( )方法对此端口进行监听，等待其他程序的连接申请。如果客户端的程序申请和此端口连接，那么服务器端就利用 accept( )方法来取得这个连接的 Socket。客户端的程序建立 Socket 时必须指定服务器的地址(host)和通信的端口号(port#)，这个端口号必须与服务器端监听的端口号保持一致。

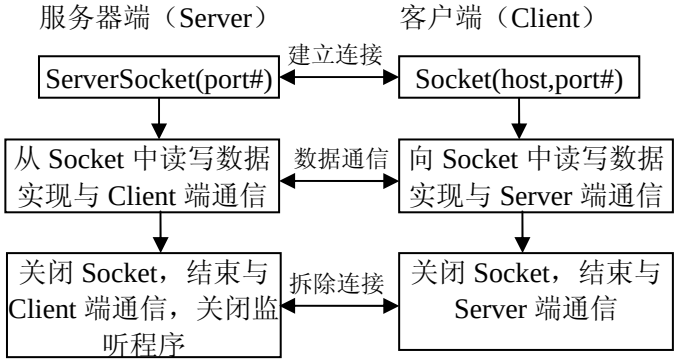

图 11_1 Socket 通信机制

11.2.1 InetAddress 类

在基于 Scoket 的通信中，应用程序需直接使用 IP 地址或域名指定运行在因特网上的某一台主机。java.net 包中定义的 InetAddress 类是一个 IP 地址或域名的抽象。创建 InetAddress 类的一个实例时既可使用字符串表示的域名，也可使用字节数组表示的 IP 地址。该类的主要内容如下所示：

```
////类方法
// 利用主机名创建一个实例
static InetAddress getByName(String host) throws UnknownHostException
// 利用 IP 地址创建一个实例
static InetAddress getByAddress(byte[] addr) throws UnknownHostException
// 利用主机名和 IP 地址创建一个实例
static InetAddress getByAddress(String host, byte[] addr) throws UnknownHostException
//根据主机名返回该主机所有 IP 地址的实例数组（例如多接口主机可绑定多个 IP 地址）
static InetAddress[] getAllByName(String host) throws UnknownHostException
// 返回本地主机的一个实例
static InetAddress getLocalHost() throws UnknownHostException
```

//// 属性访问方法（实例方法）
// 取出当前实例的主机名（参数 check 指定是否执行安全检查，默认值为 true）
String getHostName()
String getHostName(boolean check)
// 取出当前实例的完整的域名
String getCanonicalHostName()
// 取出当前实例的 IP 地址
byte[] getAddress()
// 取出当前实例的 IP 地址字符串
String getHostAddress()
//// 检测当前实例的 IP 地址所属的范围（实例方法）
// 判断是否一个多播地址（在 IPv4 中即 224.0.0.0 至 239.255.255.255 范围的所谓 D 类 IP 地址）
boolean isMulticastAddress()
// 判断是否一个通配地址（在 IPv4 中即 0.0.0.0）
boolean isAnyLocalAddress()
// 判断是否一个环回地址（在 IPv4 中即 localhost 的 IP 地址 127.*.*.*）
boolean isLoopbackAddress()
// 判断单播地址是否本地链路范围（在 IPv4 中即 169.254/16 为前缀的地址）
boolean isLinkLocalAddress()
// 判断单播地址是否本地站点范围（在 IPv4 中即 10/8、172.16/12 和 192.168/16 为前缀的地址）
boolean isSiteLocalAddress()
// 判断多播地址是否全局范围（在 IPv4 中即 224.0.1.0 至 238.255.255.255，MC 指 Multicast）
boolean isMCGlobal()
// 判断多播地址是否本地结点范围（仅 IPv6 使用）
boolean isMCNodeLocal()
// 判断多播地址是否本地链路范围（在 IPv4 中即 224.0.0/24 为前缀的地址）
boolean isMCLinkLocal()
// 判断多播地址是否本地站点范围（在 IPv4 中即 239.255/16 为前缀的地址）
boolean isMCSiteLocal()
// 判断多播地址是否本地机构范围（在 IPv4 中即 239.192/14 为前缀的地址）
boolean isMCOrgLocal()

例 11.4 InetAddress 对象的创建与使用

```
import java.net.*;
    public class Example11_4 {
        public static void main(String args[]) {
            try{
                InetAddress address1=InetAddress.getByName("www.163.com");
                System.out.println(address_1.toString());
                InetAddress address2=InetAddress.getByName("220.181.28.212");
                System.out.println(address2.getHostName());
            }catch(UnknownHostException e) {
                 System.out.println("无法找到 www.163.com");
            }
        }
```

}

当运行上面的程序时应保证已经连接到 Internet 上。该程序的运行结果是：

```
www.163.com/220.181.28.212
www.163.com
```

11.2.2 Socket 类与 Server Socket 类

Java.net 中提供了两个类：ServerSocket 和 Socket，它们分别用于服务器端和客户端的 socket 通信，进行网络通信的方法也都封装在这两个类中。

1. 创建 ServerSocket 对象与 Socket 对象的构造方法

Java 在软件包 java.net 中提供了两个类 ServerSocket 和 Socket 对应双向连接的服务器端和客户端。它们分别包含表 11-2 列出的几个构造方法。

表 11-2 ServerSocket 与 Socket 类构造方法

构造方法	功能
ServerSocket(int port)	在指定的端口创建一个服务器 Socket 对象
ServerSocket(int port，int count)	在指定的端口创建一个服务器 Socket 对象，并说明服务器端所能支持的最大连接数
Socket(InetAddress address，int port)	使用指定端口和本地 IP 地址，创建一个服务器 Socket 对象
Socket(InetAddress address，int port，boolean stream)	使用指定端口和本地 IP 地址，创建一个 Socket 对象，布尔参数为 true，则采用流式通信方式
Socket(String host，int port)	使用指定端口和主机，创建一个 Socket 对象
Socket(String host，int port，boolean stream)	使用指定端口和主机，创建一个 Socket 对象，布尔参数为 true，则采用流式通信方式

2．异常处理

在建立 socket 的同时要进行异常处理，以便程序出错时能够及时作出响应。

（1)服务器端：在建立 ServerSocket 和取得 socket 时都要进行异常处理，例如，下面语句中的 try-catch 语句。

```
ServerSocket server;
Socket socket;
try{
    server_socket=new ServerSocket(3561);
}catch(IOException e){ System.out.println(“Error occurred：”+e); }
try{
    socket=server_socket.accept();
}catch(Exception e){ System.out.println(“Error occurred：”+e); }
```

当服务器的 ServerSocket 对象 server_socket 建立后，就可以使用 accept()方法接收客户的套接字连接呼叫 server_socket.accept()，accept()方法会返回一个和客户端 Socket 对象相连接的的 Socket 对象。

（2）客户端：在建立 Socket 时进行异常处理。如下面的 try-catch 语句。

```
Socket socket;
try {
    socket=new Socket(“Server Name”,3561);
}catch(IOException e) { System.out.println(“Error occurred：”+e); }
```

3．获取输入/输出流

建立 Socket 连接后，就可以利用 Socket 类的两个方法 getOutputStream()和 getInputStream()分别获得向 Socket 读写数据的输入/输出流。此时同样也要进行异常处理，因此，通常将读写数据的输入/输出流

语句写在 try-catch 块中。例如：

```
try{
    InputStream    ins=socket. getInputStream();
    OutputStream    outs=socket. getOutputStream();
}catch(IOException e){ System.out.println(“Error occurred:”+e);}
```

4．读/写数据流

获取 Socket 的输入/输出流后，为了便于进行读/写，需要在这两个流对象的基础上建立易于操作的数据流如：InputStreamReader ,OutputStreamReader 或 PrintStream。可采用如下语句：

```
InputStreamReader in=new InputStreamReader(ins);
BufferedReader inn=new BufferedReader(in);
OutputStreamReader out=new InputStreamReader(outs);
```

或者

```
PrintStream out=new PrintStream(outs);
```

要读入一个字符串并将其长度写入输出流中，则可以用如下语句：

```
String    str=inn.readLine();
Out.println(str.length());
```

5．断开连接

无论是编写服务器程序还是客户端程序，通信结束时，必须要断开连接并释放所占用的资源。Java 提供 close()方法来断开连接，即：

（1）关闭 socket：例如，socket.close();

（2）关闭 server: 例如，server.close();

服务程序关闭连接的正确次序是：首先关闭所有输入流和输出流，然后关闭已建立连接的 socket，最后关闭用于监听的 socket。

例 11.5　客户端向服务器端发送大写字母 A—D，服务器端将其转换为小写字每 a—d 后返回给客户端。

（1）服务器端程序

```
import java.io.*;
import java.net.*;
public class Server_Socket {
  public static void main(String args[]) {
    ServerSocket server=null;
    Socket you=null;
    DataOutputStream out=null;
    DataInputStream  in=null;
    try {
     server=new ServerSocket(4331);
    }
    catch(IOException e1) {
         System.out.println(e1);
    }
    try{ System.out.println("等待客户端呼叫");
        you=server.accept();   //堵塞状态，除非有客户呼叫
        out=new DataOutputStream(you.getOutputStream());
        in=new DataInputStream(you.getInputStream());
```

```
            while(true) {
               char c=in.readChar(); // in从客户端读取信息，堵塞状态
               System.out.println("服务器收到:"+c);
               out.writeChar((char)(c+32)); //向客户端发送信息
               Thread.sleep(600);
            }
         }
         catch(Exception e) {
         System.out.println("客户已断开"+e);
         try{
             out.close();
             in.close();
             you.close();
             server.close();
         }catch(Exception ex) {
             System.out.println("关闭连接时异常"+ex);
         }

         }
       }
}
```

（2）客户端程序

```
import java.io.*;
import java.net.*;
public class Client_Socket {
   public static void main(String args[]) {
      Socket mysocket;
      DataInputStream in=null;
      DataOutputStream out=null;
      try{  mysocket=new Socket("127.0.0.1",4331);
            in=new DataInputStream(mysocket.getInputStream());
            out=new DataOutputStream(mysocket.getOutputStream());
            char c='A';
            while(c<'E') {
              out.writeChar(c);     //out向服务器发送信息
              char s=in.readChar(); //in读取服务器发来的信息
              System.out.println("客户收到:"+s);
              c++;
              Thread.sleep(500);
            }
            System.out.println("客户发送信息结束");
            out.close();
            in.close();
            mysocket.close();
```

```
        }
        catch(Exception e) {
            System.out.println("服务器已断开"+e);
        }
    }
}
```

运行结果如图 11_2 所示：

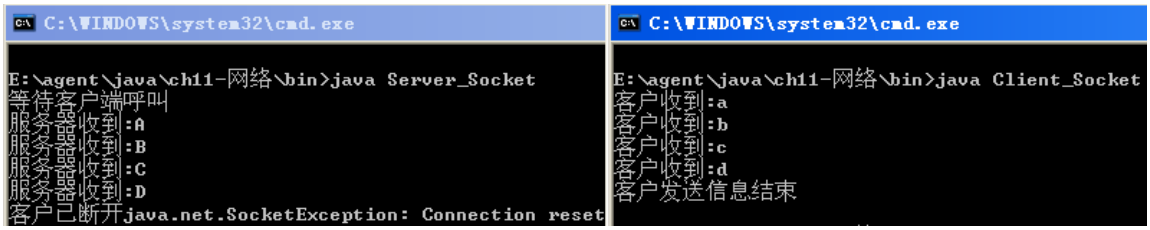

图 11_2　简单的客户 / 服务器端编程运行结果

11.2.3 多线程的客户/服务器程序

从上一小节的例子中可以发现，应用程序的服务端在同一时刻只能处理一个客户连接。一旦服务程序的 accept()方法被调用，服务程序主线程将持续执行客户程序发来的服务请求，再无其他线程监听服务程序对外发布的端口，导致后续的客户连接请求失败。

一种简单的改进途径是当服务程序处理完一个客户连接后再次循环执行 accept()。这种改进虽然可使服务程序在处理完一个客户程序的所有服务请求后，还可继续建立与下一客户程序的连接并处理其服务请求，但在一个客户程序提交的服务请求需占用服务程序较长时间的情况下，其他客户程序的连接请求将进入队列等待，甚至可能因队列溢出而丢失。

解决问题的正确途径是使用多线程编程，让服务程序的主线程执行监听客户程序连接请求的任务，而处理客户程序服务请求的任务则交由另一个新建的线程负责。例 11.6 演示了这种简单的多线程处理模型。

例 11.6 多线程服务程序

（1）服务器端程序

```
import java.io.*;
import java.net.*;
public class ServerSocketMult {
    public static void main(String[] args) throws IOException {
        ServerSocket server=null;
        ServerThread thread;
        Socket you=null;
        System.out.println("服务器端启动");
        while (true) {
            try{
                server = new ServerSocket(4331);
           }catch(IOException e1) {
                System.out.println("正在监听"); //ServerSocket对象不能重复创建
           }
          try{
```

```
                System.out.println("等待客户呼叫");
                you = server.accept();
                System.out.println("客户的地址:"+you.getInetAddress());
            }
            catch (IOException e) {
                System.out.println("正在等待客户");
            }
            if(you!=null) {
                try {
                    new ServerThread(you).start();//为每个客户启动一个专门的线程
                } catch (IOException e) {
                    you.close();
                }
             }
           }
        }
}
class ServerThread extends Thread {
    private Socket socket;
    private DataOutputStream out=null;
    private DataInputStream  in=null;
    private String s=null;
    public ServerThread(Socket s) throws IOException {
        socket = s;
        out=new DataOutputStream(socket.getOutputStream());
       in=new DataInputStream(socket.getInputStream());
    }
    public void run() {
        while(true) {
            try{
                int c=in.readInt();//接收客户发来的客户ID
                double r=in.readDouble();//堵塞状态，除非读取到信息
                double area=Math.PI*r*r;
                System.out.println("收到客户"+c+"请求:计算半径为"+r+"的圆的面积");
                out.writeUTF("半径是"+r+"的圆的面积是:"+area);
             }
             catch (IOException e) {
                  System.out.println("客户离开");
                  System.out.println("继续监听客户请求....");
                  return;
             }
         }
    }
}
```

（2）客户端程序

```
import java.io.*;
import java.net.*;
public class ClientSocketMult {
    static final int MAX_THREADS = 4;
    public static void main(String[] args) throws
IOException,InterruptedException {
        InetAddress addr = InetAddress.getByName(null);
        while (true) {
            if (ClientSocketMultThread.threadCount() < MAX_THREADS){
                new ClientSocketMultThread(addr); //模拟多个客户请求
                Thread.sleep(100);
            }else
            {
                System.out.println("所有客户请求完成,准备离开");
                break;
            }
        }
    }
}
class ClientSocketMultThread extends Thread {
    private Socket socket;
    private DataInputStream in=null;
    private DataOutputStream out=null;
    private static int threadcount = 1;
    public static int threadCount() {
        return threadcount;
    }
    public ClientSocketMultThread(InetAddress addr) {
        System.out.println("当前是第" + threadcount+"个客户");
        threadcount++;
        try {
            socket = new Socket(addr, 4331);
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
        try {
            in =new DataInputStream(socket.getInputStream());
           out = new DataOutputStream(socket.getOutputStream());
            start();
        } catch (IOException e) {
            try {
                socket.close();
            }catch (IOException e2) {}
```

```
        }
    }
    public void run() {
        String s=null;
        try{
            out.writeInt(threadcount-1);//向服务器发送客户ID;
            double r=Math.random()*10;//产生0-1000的双精度随机数
            out.writeDouble(r);  //向服务器发送要计算的圆的半径
            s=in.readUTF();     //接收服务器的计算结果
            System.out.println(s);
        }
        catch(IOException e) {
            System.out.println("与服务器已断开"+e.getMessage());
        }
    }
}
```

例 11.6 在客户端程序中用多线程模拟多个客户发出计算半径为 r 的圆的面积的请求，服务器端为每一个客户请求生成一个线程计算处理，并将计算结果返回给相应客户。该程序运行结果如图 11_3 所示。

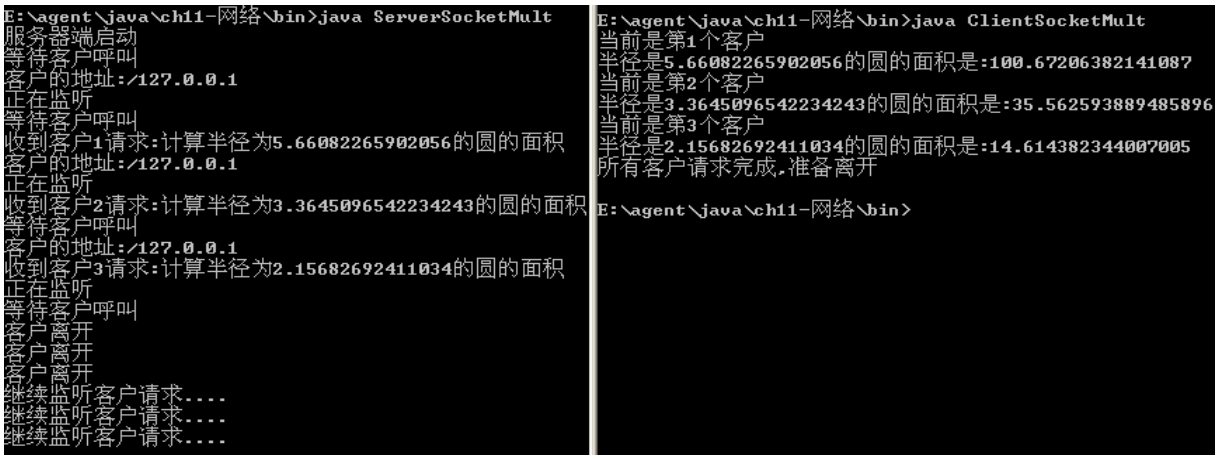

图 11_3 多线程服务程序运行结

11.2.4 URL 通信与 Socket 通信的区别

URL 通信与 Socket 通信都是面向连接的通信，它们的区别在于：Socket 通信方式为主动等待客户端的服务请求方式，而 URL 通信方式为被动等待客户端的服务请求方式。

利用 Socket 进行通信时，在服务器端运行了一个 Socket 通信程序，不停地监听客户端的连接请求，当接到客户端请求后，马上建立连接并进行通信。利用 URL 进行通信时，在服务器端常驻有一个 CGI 程序，但它一直处于睡眠状态，只有当客户端的连接请求到达时它才被唤醒，然后建立连接并进行通信。

在 Socket 通信方式中，服务器端的程序可以打开多个线程与多个客户端进行通信，并且还可以通过服务器使各个客户端之间进行通信，这种方式适合于一些较复杂的通信。而在 URL 通信方式中，服务器端的程序只能与一个客户进行通信，这种方式比较适合于 B/S 通信模式。

11.3 数据报通信

URL 和 Socket 通信是一种面向连接的流式套接字通信，采用的协议是 TCP 协议。在面向连接的通信中，通信的双方需要首先建立连接再进行通信，这需要占用资源与时间，具有较高的开销。但是在建

立连接之后，双方就可以准确、同步、可靠地进行通信了。流式套接字通信在建立连接之后，可以通过流来进行大量的数据交换。TCP 通信被广泛应用在文件传输、远程连接等需要可靠传输数据的领域，。

UDP 通信是一种无连接的数据报通信，采用的协议是数据报通信协议 UDP(User Datagram Protocol)。UDP 并不刻意追求数据包会完全发送出去，也不能担保它们抵达的顺序与它们发出时一样。我们认为这是一种“不可靠协议”（TCP 当然是“可靠协议”）。按照这个协议，两个程序进行通信时不用建立连接；数据以独立的包为单位发送，包的容量不能太大；每个数据报需要有完整的收/发地址，可以随时进行收/发数据报，但不保证传送顺序和内容准确；数据报可能会被丢失、延误等。因此，UDP 通信是不可靠的通信。由于 UDP 通信速度较快，因此常常被应用在某些要求实时交互，准确性要求不高，但传输速度要求较高的场合。

Java.net 软件包中的类 DatagramSocket 和类 DatagramPacket 为实现 DUP 通信提供了支持。

11.3.1 DatagramSocket 类

DatagramSocket 类用于收/发数据报。它有三个构造方法：

（1）DatagramSocket()；

（2）DatagramSocket(int port)；

（3）DatagramSocket(int port,InetAddress iaddr);

其中，第一个构造方法将 socket 连接到本机的任何一个可用的端口上；第二个将 socket 连接到本机的 port 端口上；第三个则将 socket 连接到指定地址的 port 端口上。

这里需要注意两点：一是规定端口时不要发生冲突；二是在调用构造方法时要进行异常处理。

receive()和 send()是 DatagramSocket 类中用来实现数据报传送和接收的两个重要方法，其格式如下：

public synchronized void receive(DatagramPacket packet)throws IOException

public void send(DatagramPacket packet)throws IOException

receive()方法将使程序中的线程一直处于阻塞状态，直到从当前 socket 中接收到信息时，将收到的信息存储在 receive()方法的对象参数 packet 的存储机构中。由于数据报是不可靠的通信，所以 receive()方法不一定能读到数据。为防止线程死掉，应该设置超时参数(timeout)。

send()方法将其参数 DatagramPacket 对象 packet 中包含的数据报文发送到所指定的 IP 地址主机的指定端口。这两个方法都可能产生输入／输出异常，所以都抛出 IOException 异常。

11.3.2 DatagramPacket 类

DatagramPacket 类用来实现数据报通信，它有两个构造方法，分别对应接收数据报和发送数据报：

public DatagramPacket(byte sBuf[],int sLength,InetAddress iaddr,int iport)

这个构造方法用来创建发送数据报对象。其中，sBuf 代表发送数据报的字节数组；sLength 代表发送数据报的长度；iaddr 代表发送数据报的目的地址，即接收者的 IP 地址；iport 代表发送数据报的端口号。

public DatagramPacket(byte rBuf[],int rLength)

这个构造方法用来创建接收数据报对象。其中，rBuf 代表接收数据报的字节数组；rLength 代表接收的数据报的长度，即读取的字节数。

11.3.3 UDP 通信的示例程序

下面通过建立一个简单的 UDP 服务器端和一个客户端的程序例子，讲述 UDP 的工作方式。在这个例子中，服务器端的程序只是不停地监听本机端口，一旦收到客户端发来的数据报，就回应一个简单的信息通知客户已经收到了数据报。客户端的程序向服务器发送一个包含一个字符串的数据报，同时告知服务器自己的地址及端口，以便服务器做出回应。

例 11.7

（1）服务器端程序

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

```
import java.util.*;
class  UDPServerThread extends Thread{ //启动服务器线程的主程序
    private DatagramPacket packet;
    private DatagramSocket socket;
    static final int sport=4331;
    UDPServerThread()
    {
        try{// 将socket连接到本机的一个可用的端口上
            socket=new DatagramSocket(sport);
            System.out.println("监听端口:"+socket.getLocalPort());
        }
        catch(Exception e){  System.out.println("Error: "+e); }
    }
    public void run()
    {  //线程的主要操作
        if(socket==null)return;
        while(true)
        {
            try{
                InetAddress address;
                int cport;
                byte[] buf1=new byte[1000],buf2=new byte[1000];
                String s="你的信息已经收到";
                packet=new DatagramPacket(buf1,buf1.length); //生成一个接收数据报
                socket.receive(packet);    //接收数据报
                String s1=new String(packet.getData());
                System.out.println("这是从客户端收到的信息: "+s1);//打印数据报的内容
                address=packet.getAddress();
                cport=packet.getPort();    //获得数据报的源地址与端口
                buf2=s.getBytes();
                //生成发送的数据报
                packet=new DatagramPacket(buf2,buf2.length,address,cport);
                socket.send(packet);     // 发送数据报给客户
            }
            catch(Exception e){ System.out.println("Error: "+e); }
        }
    }
    protected void finalize()
    {
        if(socket!=null)
        {
            socket.close(); //关闭socket
            System.out.println("Socket Closed.");
        }
```

```
    }
}
public class UDPServer
{
    public static void main(String[] args){
        UDPServerThread server=new UDPServerThread();
        server.start();
    }
}
```

（2）客户端程序

```
import java.io.*;
import java.net.*;
import java.util.*;
class UDPClient{
    public static void main(String[] args)
    {
        DatagramSocket socket;//用于发送接受UDP
        DatagramPacket packet;//用于保存UDP的内容
        InetAddress address;
        int port;
        byte[] buf1=new byte[1000], buf2=new byte[1000];
        String s="你好,我是客户端!",s2;
        if(args.length<3)
        {
            System.out.println("输入本地端口号,服务器名,服务器端口号 ");
            System.exit(0);
        }
        try {
            socket=new DatagramSocket(Integer.parseInt(args[0]));
            address=InetAddress.getByName(args[1]);
            port=Integer.parseInt(args[2]);
            buf1=s.getBytes();
            packet=new DatagramPacket(buf1,buf1.length,address,port);
            socket.send(packet);  //向服务器发送packet
            packet=new DatagramPacket(buf2,buf2.length); // 生成接收的packet
            socket.receive(packet);  //接收服务器传来的packet
            s2= new String(packet.getData());
            System.out.println("这是服务器端返回的信息: "+s2); // 打印packet内容
            socket.close();  // 关闭 socket
        }catch(Exception e){
            System.out.println("Error:"+e);
        }
    }
}
```

服务器端的运行结果如下：

监听端口:4331

这是从客户端收到的信息：你好,我是客户端!

客户端的 java UDPClient 8080 127.0.0.1 4331 的结果如下：

这是服务器端返回的信息：你的信息已经收到

小结

Java 程序员处理网络通信的主要优势在于完善的异常处理机制、内建的多线程机制、以及采用输入／输出流作为应用程序统一的 I/O 接口。

JDK 提供的包 java.net 为程序员编写基于 TCP 或 UDP 通信的应用程序提供了有力支持,并可帮助程序员直接、方便地使用 HTTP、FTP、FILE 等协议。该包的一个显著特征是其简单性。基于 TCP 的应用程序使用该包提供的 Socket 和 ServerSocket 等类；基于 UDP 的应用程序则使用该包提供的 DatagramPacket 、DatagramSocket 和 MulticastSocket 等类；基于 HTTP、FTP、FILE 的应用程序可使用该包提供的 URL 和 URLConnection 等类。

习题

一、基本概念

1、一个 URL 对象通常包含哪些信息？

2、怎样读取 URL 中的资源?

3、写入 URLConnection 的步骤是什么?

4、客户端的 Socket 对象和服务器的 Socket 对象是怎样通信的？

5、什么叫 Datagram? 基于 UDP 的通信和基于 TCP 的通信有什么不同？

二、编程实践

1、编写一个程序，获取指定主机的主机名、主机地址。

2、编写一个程序，读取人民网中的新闻，并设计一个算法，尝试从中过滤出新闻标题、时间、新闻内容，然后再用该程序读取其他新闻网站的新闻内容，检验其有效性。

2、结合 Java 网络编程与线程的知识，编写一个简单的聊天程序。

第 12 章 泛型程序设计与对象容器

泛型是 JDK5.0 增加的一种抽象级别更高的程序设计机制，它可将不同类型对象的同一种操作加以抽象、封装，使其享有更广泛的代码重用性，为有效地体现面向对象的程序设计思想提供了一个良好的途径。

对象容器是能够容纳其它对象的对象，容纳的对象称为对象容器的元素，例如数组就是一种最基本的对象容器。对象容器内的元素与元素之间具有一定的结构，这种结构通过对象容器提供的操作及约束体现。对象容器提供了一些有用的数据结构和算法，是程序组织和操纵批量数据的强有力工具，使用对象容器可提高软件开发效率。

12.1 泛型程序设计

在程序设计中经常遇到不同的数据类型可以实施同一种操作的情形。例如，在学生信息管理系统中需要按照学号排列学生名单，在图书信息管理系统中需要按照书号排列图书目录，在销售商品信息管理系统中需要按照销售量排列商品销售情况等。尽管它们属于不同类别，但排序过程应该是一样的。如果为每个类别重复编写具有相同处理过程的程序代码会增加编写程序的工作量。在 Java 早期版本中可以声明一个元素为 Object 的一维数组存放具有这种操作特性的数据。例如：

```
public class ArrayList
{
    private Object[] element;
    ……
    public Object get(int index){ return element[index];}
    public void add(Object obj){ …… }
    public void sort(){…… }
    ……
}
```

这样可以实现数组元素引用任何类对象的目的。但这种方法存在下面两个问题：

（1） 获取对象时必须进行强制类型转换。例如：

```
ArrayList list = new ArrayList();
list.add("Hello");
String str1 = (String) list.get(0);
```

这样看上去十分烦琐，如果忘记强制类型转换就会出现错误。

（2） 没有检查类型错误

由于可以向 list 中添加任何类对象，所以在后面的操作中可能会现强制类型转错误。例如：

```
Integer value = (Integer)list.get(0);
```

显示，list.get()应该返回 String 类对象的引用，所以将其强制转换为 Integer 类是错误的。为了更好地解决这类问题，Java 提出了泛型概念，并在 JDK5.0 版本中给予了支持。所谓泛型是指将数据类型作为参数实现相同操作过程的代码重用机制，它不但增强了代码重用率，还具有类型检查功能，确保了程序的安全性。

12.1.1 泛型类的定义与使用

下面通过一个例子说明定义泛型类的格式：

例 12.1

```
public class Pair<T> {
    private T first;
    private T second;
    public Pair(){
```

```
            first = null;
            second = null;
        }
        public Pair(T first, T second){
            this.first = first;
            this.second = second;
        }
        public T getFirst(){
            return first;
        }
        public T getSecond(){
            return second;
        }
        public void setFirst(T first){
            this.first = first;
        }
        public void setSecond(T second){
            this.second = second;
        }
        public static void main(String args[]){
            Pair<Integer> intPair = new Pair<Integer>(new Integer(2),new
Integer(1));
            Integer v1,v2;
            v1 = intPair.getFirst();
            v2 = intPair.getSecond();
            Pair<String> strPair = new Pair<String>("hello","world");
            String s1,s2;
            s1 = strPair.getFirst();
            s2 = strPair.getSecond();
        }
    }
```

在这个类声明中引入了类型参数 T，这个参数书写在类名之后并用一对尖括号括起来。 有了这个声明就可以按照 main()方法中的方式使用它。

在定义对象时类型参数 T 替换为 Integer 表示类中所有 T 的位置都为 Integer；将类型参数 T 替换为 String 表示类中所有 T 的位置都为 String，这样既能让各种类型重用同一段程序代码，以可以避免人为地强制类型转换带来的错误。注意，在实例化泛型类时不允许带入基本数据类型。例如 Pair<double>是错误的，必须写成 Pair<Double>。

上面定义的 Pair 类只包含一个类型参数。实际上，可以根据需求设置多个类型参数，并通它们带入不同的类型。例如，下面定义的 ThirdElement 类包含了 3 个类型参数。

例 12.2

```
    public class ThirdElement<U,V,W> {
        private U element1;
        private V element2;
        private W element3;
```

```
        public ThirdElement(){
            element1 = null;element2= null;element3=null;
        }
        public ThirdElement(U e1, V e2, W e3)
        {
            element1=e1;
            element2=e2;
            element3=e3;
        }
        public U getElement1(){ return element1; }
        public V getElement2(){ return element2; }
        public W getElement3(){ return element3; }

        public static void main(String args[])
        {
            ThirdElement<Integer,String,Double> e=
            new ThirdElement<Integer,String,Double>(new Integer(1),"Java",new
Double(3.0));
            Integer e1 = e.getElement1();
            String e2 = e.getElement2();
            Double e3 = e.getElement3();
        }
    }
```

有时需要对参数类型加以限定。例如，要求带入的类必须具有比较大小的功能，即这些类必须实现 Comparable 接口。例 12.3 是表示这种限定参数类型的泛型定义格式。

例 12.3

```
    public class Interval<T extends Comparable> {
        private T lower;
        private T upper;
        public Interval(T first, T second){
            if(first.compareTo(second)<=0){
                lower = first; upper = second;
            }else{
                lower = second; upper=first;
            }
        }
    }
```

上面的例子中，<T extends Comparable>的含义是 T 限定为 Comparable 的子类。注意，尽管 Comparable 是接口，但在这里用 extends 关键字，而不用 implements 关键字。

12.1.2 对象包装器

在 Java 语言中，每一种基本数据类型都有一个对应的类，这些类被称为包装器。表 12-1 列出了各种基本数据类型所对应的包装器名。

表 12-1 Java 基本数据类型的包装器

基本数据类型	包装器	基本数据类型	包装器

byte	Byte	char	Char
short	Short	boolean	Boolean
int	Integer	float	Float
long	Long	double	Double

表 12-1 中的包装器都是 final 类，即不能再定义这些类的子类，并且一旦创建了这些类的对象，则其中包含的值不允许更改。

注意，基本数据类型与包装器是两个不同的概念。例如：

int value = 10;

Integer obj = new Integer(10);

value 的值是一个整型数值，而 obj 是一个引用，它引用了一个 Integer 类对象。为了便于操作，Java 提供了自打包和自拆包功能。所谓自打包是指在应该提供包装器对象的地方如果给的是基本数据类型值时，Java 虚拟机则自动将其打包成包装器对象。例如：

Integer value = 200;

Java 虚拟机自动将 200 打包使其等价于：

Integer value = new Integer(200);

所谓自拆包是指在需要提供基本数据类型的地方如果给的是包装器对象，Java 虚拟机则自动将其拆包成基本数据类型值。例如：

int value = new Integer(200);

Java 虚拟机自动将 Integer(200)拆包使其等价于：

int value = 200;

在编写程序时，由于 Java 语法的限制，有些地方要求提供基本数据类型值，有些地方要求提供包装器对象，因此，Java 提供的自打包和自拆包功能极大地方便了程序的书写，特别是在应用泛型类时经常会遇到使用它的场合。

12.2 Java 对象容器

对象容器用于存放对象并将它们组织成一定结构，这种结构由对象容器提供的操作体现，反过来说，对象容器提供的操作也需要满足这种结构方面的约束。

主要有四个因素影响对象容器存放的对象所形成的结构：一是是否允许存放重复的对象，这里所谓存放重复的对象，是指容器中包含有多个使用深比较相等的对象；二是存放的对象之间是否有序，如果要求存放的对象有序，则所存放对象之间在某种意义下必须是可比较的；三是如何访问所存放的对象，这通常有两种方式，顺序访问和随机访问，随机访问就是指能用常数时间访问容器内任意位置的对象，而顺序访问只能从存放对象的起始或者末尾位置开始访问，如果只能从某一位置顺序访问下一个，是单向顺序访问，如果还能从该位置访问上一个就是双向顺序访问；四是存放对象的数目是否有限制，如果存放对象的数目没有限制，则容器总是可以自动扩展以存放更多的对象。

Java 的对象容器都定义在程序包 java.util 中，该程序包及其子程序包为 Java 程序提供了一系列有用的工具，除对象容器外，还包括操作日期时间的工具、压缩解压缩数据的工具等。

程序包 java.util 中有关对象容器的部分包括一些接口、类及辅助工具，其中接口是对对象容器所要提供的基本操作的抽象，类给出了对象容器的一些具体实现，辅助工具用于操纵、包装对象容器，并提供各种对象容器间的转换功能。

12.2.1 Java 对象容器接口

Java API 按对象容器能存放的对象将容器分为两种：一种称为类集(collection)容器，用接口 Collection 描述这种容器的基本操作，其中存放的基本单位是单个对象；一种称为映射(map)容器，用接口 Map 描述这种容器的基本操作，其中存放的基本单位是对象对(object pairs)，其中一个对象称为键(key)对象，另一个对象称为值(value)对象。

接口 Collection 定义了类集容器的一些基本操作，这些操作包括添加(add)、删除(remove)、查找是

否包含某对象(contains)及返回迭代器(iterator)用于遍历容器内对象等。

根据是否能存放重复的对象，类集容器又可进一步分为列表容器与集合容器，Java API 分别用接口 List 和 Set 描述它们的基本操作。列表容器允许存放重复的对象，且能够双向顺序访问和随机访问其中的对象。列表容器除可使用通用迭代器类 Iterator 的对象单向遍历存放的对象外，还可使用专用迭代器类 ListIterator 的对象双向遍历存放的对象。同时列表容器还可通过下标（即存放的位置）随机访问其中的对象。集合容器则不允许存放重复的对象，是数学中集合的一种实现。集合容器只能单向顺序遍历存放的对象，不能双向遍历，也不能随机访问。Java API 为这两个接口分别提供了抽象类 AbstractList 和 AbstractSet。

列表容器和集合容器中存放的对象之间一般来说是没有序的，但列表容器可将存放位置或下标看作对象之间的一种序（不过这种序不是通过比较对象得到的顺序）。普通集合容器存放的对象没有序，但 Java API 为接口 Set 派生了一个接口 SortedSet 描述有序集合容器，该容器中存放的对象应该可以互相比较，通过这种比较形成一种序。

Java 类库提供的部分类集容器的接口和类的关系可使用图 12_1 来描述：

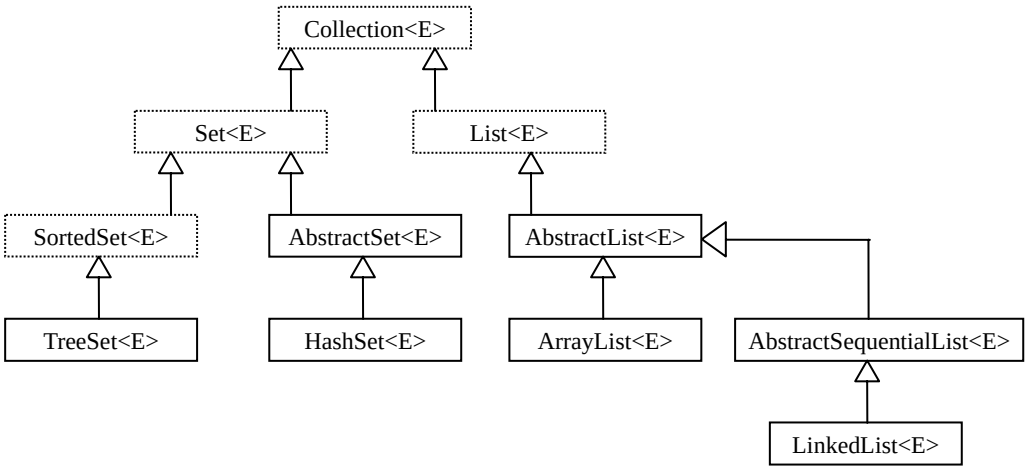

图 12_1 Java 类库提供的部分数据结构接口与类的关系

在图 12_1 中，虚线框表示接口，实线框表示类。可以看出，它们都是泛型类或接口，Collection 是集合类结构的顶层接口，其中声明了对集合和链表操作的 15 个成员方法，将集合和链表的操作统一地列在这里，这些方法需要在集合类或链表类中加以实现。

映射容器存放的对象对，一个是键对象，一个是值对象。接口 Map 内部定义了一个嵌套接口 Entry 表示映射容器中存放的对象对，该接口定义了取键对象和值对象的方法，也定义了修改值对象的方法，但没有定义修改键对象的方法。

接口 Map 定义了添加键和值对象(put)、根据键对象删除对象对(remove)、根据键对象和值对象查找对象对(contains)等操作。接口 Map 不能返回用于遍历映射容器的迭代器，但可根据键对象直接访问其对应的值对象，也可将整个映射容器转换为集合容器（其中存放的对象具有类型 Map.Entry），或者分别将映射容器中存放的所有键对象转换为一个集合容器，将所有值对象转换为一个（一般的）类集容器，这些容器可返回迭代器对它们进行遍历。这也说明了映射容器存放的所有键对象可看作构成了一个集合容器，而所有值对象则可看作构成了一个一般的类集容器。由此可知，键对象不允许存放重复对象，但值对象允许存放重复对象，也就是说，一个键对象只对应一个值对象，但不同的键对象可能具有（深比较）相等的值对象。

Java API 同样为接口 Map 提供了一个抽象实现类 AbstractMap。有关映射容器的接口和类可用图 12_2 表示：

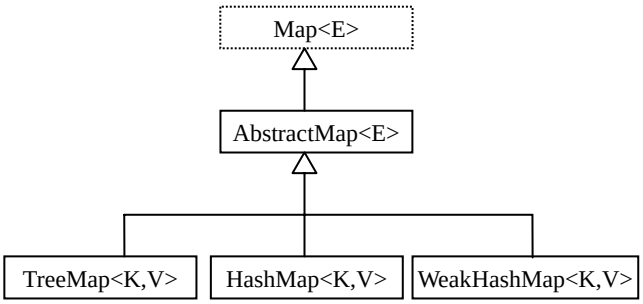

图 12_2 与 Map 接口有关的类关系图

12.2.2 Java 的对象容器类

上一节介绍了描述列表容器、集合容器和映射容器的基本操作的接口，这一节介绍 Java API 提供的实现这些容器接口的具体类，这些类是可使用的对象容器类。

Java API 提供的列表容器具体类主要包括类 LinkedList、ArrayList、Vector 和 Stack，这些类都实现了接口 List，都是 AbstractList 的后代类。类 LinkedList 内部使用双向链接表实现，创建的列表容器的特点是在添加对象时可以指定是添加在列表的起始位置还是添加在列表的结尾位置，并且可返回起始位置或者结尾位置的对象。类 ArrayList 创建的列表容器的特点是相当于是一个可变长的数组。类 Vector 和 Stack 是 JDK 1.2 版以前提供的对象容器类，在 JDK1.2 版以后的 Java 平台下编写的 Java 程序已不推荐使用这两个容器类。它们与 JDK1.2 版以后提供的对象容器类的最大区别是，这些类的方法都是同步实现的，所谓同步实现就是指在多线程情况下要保证只有一个线程在访问容器内的对象。类 Vector 是类 ArrayList 的同步实现版本，类 Stack 是 Vector 的子类，它提供操作允许将类 Stack 的对象用作堆栈，即按先进后出方式使用它存放的对象。

Java API 提供的集合容器具体类主要包括类 HashSet 和 TreeSet，其中类 HashSet 实现普通集合容器接口 Set，而类 TreeSet 实现有序集合容器接口 SortedSet。这两个类从外部结构特点的差别来说主要是存放的对象之间是否有序，从内部来看则主要是采用了不同的实现方式，类 HashSet 使用散列表，而类 TreeSet 使用红黑树。

类 HashSet 内部使用散列表存放元素，散列表是一种常见的数据结构，它使用散列函数计算要存放元素的散列码，根据散列码确定元素的存放位置。当有两个要存放元素的散列码相同时就发生了冲突(collision)，这时需要使用一种合适的策略解决冲突，例如最简单的策略可能是在每个位置再引出一个表，表中依次存放散列码相同的元素。不难看出，增加、删除和查找散列表中的元素通常都可以在常数级时间完成，除非散列函数设计得太糟糕使得存放元素时总是发生冲突。当然散列表的性能还取决于散列表的容量与装载因子，所谓容量是指散列表中位置（用散列表的术语是桶 bucket，因为每个位置实际上还可能引出一个表以解决冲突）的个数，装载因子用于度量散列表被充满的程度，等于已经存放的元素个数除以散列表的容量。通常散列表达到装载因子时就应自动扩展其容量以降低冲突，提高效率。

Java 语言的每一个类都从根类 Object 继承了方法 hashCode()，该方法虽然不能直接作为散列表中的散列函数，但可辅助散列函数计算要存放对象的散列码。Java API 提供的每一个类都提供了 hashCode()方法的合适的重定义版本，以保证使用深比较相等的对象具有相同的散列码。我们自己编写的类也应该遵循这种要求，一种简单的方法就是调用用于深比较的域的 hashCode()方法，将其返回的值进行某种数学运算，然后把最后的计算结果作为整个类的 hashCode()方法的返回值。

例如假定深比较两个学生对象是否相等，只要判断它们的学号相等即可，那么可将学号的 hashCode()方法（如果用基本数据类型表示学号，则直接将其转换为数值）的返回值作为学生类的 hashCode()方法的返回值。如果深比较两个学生对象相等需要判断他们的学号与姓名，那么可以将调用学号与姓名的 hashCode()方法得到的返回值相加或作其它什么运算得到的值作为学生类的 hashCode()方法的返回值。这样就能够保证使用深比较相等的对象具有相同的散列码。

类 TreeSet 内部采用红黑树存放元素。红黑树是一种二叉查找树，它每一个层次的节点都着以红黑

两种不同的颜色，使得在插入和删除元素时能尽量保证每个树叶的高度（即离根的节点数）都大致相等，因此是一种简单的平衡二叉树。平衡二叉树在插入、删除和查找元素时也具有较好的性能，当红黑树中存放 n 个元素时，这些操作的时间量级大致在 log(n)。与散列表相比，如果散列表的散列函数较好使得发生冲突的可能小，那么散列表的性能优于红黑树，如果散列函数使得发生冲突可能性大，那么红黑树的性能可能优于散列表。所以，在具体编程时是采用类 HashSet 还是类 TreeSet，主要看需要存放的对象之间是否有序。

Java API 提供的映射容器具体类主要包括类 HashMap、TreeMap 和 HashTable。类 HashTable 同样是 JDK 1.2 版以前提供的容器类，JDK1.2 版以后已经将它改为实现接口 Map。类 HashMap 和 HashTable 十分类似，内部都采用散列表存放对象对，主要区别在于类 HashTable 的方法是同步实现的，而类 HashMap 为提高效率放弃了同步实现。类 TreeMap 则实现接口 SortedMap，内部使用红黑树存放元素，创建的对象是有序映射容器。表 12-1 总结了 Java API 提供的对象容器类所实现的容器种类及其实现方面的特征。

表 12-1 Java API 提供的对象容器类

容器种类			实现的接口	容器类名	实现方面的特点	JDK版本
类集容器	列表容器		List	LinkedList	内部使用双向链表	>= 1.2
			List	ArrayList	相当于可变长数组	>=1.2
			List	Vector	同步实现的可变长数组	< 1.2
			List	Stack	类 Vector 的子类实现堆栈先进后出的特征	< 1.2
	集合容器	普通集合容器	Set	HashSet	内部使用散列表	>= 1.2
		有序集合容器	SortedList	TreeSet	内部使用红黑树	>= 1.2
映射容器	普通映射容器		Map	HashMap	内部使用散列表	>= 1.2
			Map	WeakHashMap	与 HashMap 十分类似但自动删除不用的元素	>= 1.2
			Map	HashTable	与 HashMap 十分类似相当于其同步实现的版本	< 1.2
	有序映射容器		SortedMap	TreeMap	内部使用红黑树	>= 1.2

12.2.3 支持对象容器的辅助工具

Java API 用于辅助对象容器的接口包括 Iterator、Comparable 和 Comparator。接口 Iterator 抽象了遍历类集对象容器内所有对象的行为，主要包括 hastNext()方法（判断有否下一元素）和 next()方法（返回下一元素）。通常类集对象容器类使用内部类实现接口 Iterator，并通过方法 iterator()方法返回迭代器对象，程序可利用该迭代器对象遍历容器内存放的所有对象。

接口 Comparable 和 Comparator 用于对象之间的比较，使得对象可以放入有序的对象容器。当容器要求存放的对象之间有序时，有两种方法满足这种要求：第一种方法是声明要存入容器的对象实现接口 Comparable，这时它必须实现方法 compareTo()，以比较两个具有相同类型的对象；第二种方法是在创建对象容器的同时提供一个实现接口 Comparator 的对象（这种对象称为比较子），该对象实现方法 compare()，它应能够比较要放入容器的任意两个对象。这两种方法到底采用哪种方法在创建容器时决定：如果创建容器时提供比较子则采用第二种方法，否则采用第一种方法。

通常容器内两个对象之间的比较应该是深比较。虽然 Java API 的文档规范没有要求这种比较（使用 compareTo()方法或 compare()方法）要与对象相等（使用 equals()方法）一致，为了使对象的行为在逻辑上合理，程序员在实现 compareTo()方法或 compare()方法时，应该使得它们与 equals()方法保持一致。对象比较与对象相等保持一致是指，将对象 A 与对象 B 比较时，若对象 A 小于等于对象 B 且对象 B 小于等于对象 A 时，则对象 A 与对象 B 应该相等。

Java API 用于辅助对象容器的类包括类 Arrays 和 Collections。类 Arrays 用于辅助操纵最基本的对象容器——数组，它提供了对各种基本数据类型及 Object 类型的数组进行二分查找(binary search)、排序(sort)、填充(fill)以及判断两个数组相等(equals)等一系列的公有静态方法。

类 Collections 用于辅助操纵除数组以外的对象容器，它提供的功能包括：对列表容器内的对象进行二分查找和排序、对类集容器内的对象进行统计（主要是求最大值和最小值）、将各种容器（包括类集容器和映射容器）包装成不可修改的容器（即不可对容器进行添加和删除等操作）、将各种容器包装成同步的容器（即保证在多线程情况下同一时刻只有一个线程访问该容器）以及其它一些辅助功能。

12.3 迭代器

在组织和操纵批量数据时，遍历所有的数据是其中一项重要的操作。迭代器(Iterator)模式是完成此操作的一种很好的设计方案。在介绍对象容器的典型用法前，这一节先介绍迭代器模式的基本思想和用法。

12.3.1 迭代器设计模式

迭代器是对象容器遍历行为的一种抽象。迭代器模式的设计目标是将遍历行为与对象容器的其它行为（如插入、删除元素等）分离开。迭代器模式的基本思想是定义一个迭代器接口 Iterator，支持遍历行为的对象容器（称为迭代器的基础容器）使用内部类实现该接口，对象容器的使用者可通过公有方法获得该容器的一个迭代器对象。

在 Java API 中，迭代器接口 Iterator 定义为：

```
public interface Iterator<E> {
    public boolean hasNext(); // 判断基础容器是否有下一元素
    public E Object next(); // 返回基础容器的下一元素
    public void remove(); // 删除基础容器的当前元素
}
```

其中，方法 hasNext()和 next()描述最基本的遍历行为，方法 remove()是一个可选行为，用于在遍历时删除某些满足或不满足某个条件的元素。

对象容器类使用一个私有的内部类实现迭代器接口，然后使用公有的方法返回该内部类的一个对象实例给使用者使用。由于内部类对象可以共享其外围类对象的数据，因此使用内部类实现使得迭代器通常是轻量级的对象，即创建迭代器所需的开销可以很少。例如，类 AbstractList 的实现中有如下程序片段：

```
public abstract class AbstractList<E> extends AbstractCollection implements
List {
    //...  其它语句
    // 定义私有类实现接口Iterator
    private class Itr implements Iterator<E> {
        //...  实现接口Iterator 的方法
        public boolean hasNext() {
            // ......
        }
        public E next() {
            // ......
        }
        public void remove() {
        }
    }
    // 使用公有方法返回类Itr 的一个实例作为使用者使用的迭代器
```

```
    public Iterator iterator() {
        return new Itr();
    }
    //... 其它语句
}
```

这是 Java 语言内部类的典型用法。虽然在使用对象容器时无需关心其内部实现迭代器的方式，但了解迭代器模式对程序员编写其它应用程序有很好的借鉴作用。

采用迭代器模式的优点是对象容器对使用者隐藏了遍历行为的内部实现，使用者可使用统一的接口方法遍历任意的对象容器。迭代器模式还可支持一个对象容器使用几种迭代器，以支持不同的遍历行为。例如 Java API 的列表容器就可有两种迭代器：一种是通用迭代器，支持单向遍历，一种是列表专用迭代器，支持双向遍历。进一步，迭代器模式还支持对象容器的并发遍历，即同一时刻可有多个线程使用迭代器对同一对象容器进行遍历等。

12.3.2 迭代器的基本用法

在 Java 程序中使用迭代器遍历对象容器存放的所有对象十分简单，下面的程序片段给出了迭代器的基本用法：

```
// container 是任意的类集容器
Iterator iterator = container.iterator();
while (iterator.hasNext()) {
    // 假定container 存放对象的类型是MyClass。
    // iterator.next()返回容器container 存放的当前元素的引用，并移向下一元素
    MyClass obj = (MyClass)iterator.next();
    ... // 利用对象引用obj 对该元素进行其它操作
    // 进一步可根据某种条件过滤（即删除）container 中的当前元素
    if (...) iterator.remove();
}
```

对于迭代器的使用要注意几点：

（1）迭代器方法 next()返回的对象引用是对其基础容器中的当前对象的引用，也就是返回的对象引用与基础容器的当前对象引用指向同一个对象实例，使用者使用返回的对象引用进行操作也会影响容器中的对象引用所指向的对象实例；

（2）迭代器方法 remove()是将容器内的当前对象引用删除，不会影响后续的遍历过程，删除的对象引用由于使用者已经得到了一个副本（由 next()方法返回的对象引用），使用者实际上还可对其进行操作（即调用该对象定义的一些方法等）；

（3）通用迭代器是一次性消耗品，当遍历完容器所有元素之后，通用迭代器对象就再也没有用了，因为通用迭代器对象没有提供返回到起始位置的方法；

（4）在获得迭代器后，只能通过迭代器的 remove()方法（列表专用的迭代器还有 add()方法）删除（或增加）基础容器内的元素，如果使用其它途径改变了基础容器内元素的个数（但改变某元素引用的对象实例的状态则无妨），则会使获得的迭代器抛出异常 ConcurrentModificationException 而无效，迭代器的这种性质被称为 fail-fast（遇并发修改则即时作废）。例如下述程序片段将抛出异常 C ConcurrentModificationException：

```
Iterator iterator = container.iterator();
container.add(new MyClass()); // 改变容器内的元素
while (iterator.hasNext()) {
    // 下述语句抛出异常ConcurrentModificationException
    MyClass obj = (MyClass)iterator.next();
```

```
}
```

12.3.3 列表专用迭代器

列表容器除了可以使用通用迭代器之外，还可使用列表专用迭代器，它支持对列表容器的双向遍历。Java API 定义接口 ListIterator 描述列表专用迭代器提供的方法：

```
public interface ListIterator<E> extends Iterator<E> {
    // 向前遍历的方法
    public boolean hasNext(); // 判断基础容器是否有下一元素
    public E next(); // 返回基础容器的下一元素
    // 返回基础容器下一元素的下标，准确地说是下一次调用next()将得到的元素的下标
    public int nextIndex();
    // 往回遍历的方法
    public boolean hasPrevious(); // 判断基础容器是否有上一元素
    public E previous(); // 返回基础容器的上一元素
    // 返回基础容器上一元素的下标，准确地说是下一次调用previous()将得到的元素的下标
    public int previousIndex();
    // 修改容器的方法
    public void remove(); // 删除基础容器的当前元素
    // 插入元素在当前位置，准确地说是下一次调用next()得到的元素之前的位置
    // 在下一次调用next()之后，再调用previous()即得到新插入的元素。
    public void add(E obj);
    // 替换当前元素，准确地说是替换刚才使用next()得到的元素
    public void set(E obj);
}
```

例12.4定义了一个用于演示的简单类SmallClass，这个类重定义了equals()方法和hashCode()方法，并实现了接口Comparable。

例 12.4 列表专用迭代器的用法

```
class SmallClass implements Comparable{
    private int x = 0;
    public SmallClass(int x) { this.x = x; }
    public int get() { return x; }
    public void set(int x) { this.x = x; }
    public String toString() { return "The SmallClass's x="+x; }
    // 重定义equals()方法，当两个SmallClass的对象中的域x相等时就认为这两个对象相等!
    public boolean equals(Object other) {
        if (other == null) return false;
        if (other == this) return true;
        if (! (other instanceof SmallClass)) return false;
        if (x == ((SmallClass)other).x) return true;
            return false;
    }
    // 重定义hashCode()方法，返回域x作为散列码，这满足当调用上述equals()方法相等的对象有相同的散列码
    public int hashCode() {return x;}
    public int compareTo(Object other) throws ClassCastException {
```

```
            if (! (other instanceof SmallClass)) throw new ClassCastException();
            //约定当前对象小于参数other时返回负数，等于时返回0,大于时返回正数
            return (x-((SmallClass)other).x);
        }
    }
    public class Example12_4 {
        public static void main(String[] args) {
            // 创建一个ArrayList容器，初始时添加三个元素
            ArrayList<SmallClass>  List list = new ArrayList<SmallClass> ();
            list.add(new SmallClass(10));
            list.add(new SmallClass(20));
            list.add(new SmallClass(30));
            // 获取列表专用迭代器
            ListIterator iterator = list.listIterator();
            System.out.println("向前遍历的结果：");
            while (iterator.hasNext()) {
                // 返回下一个元素的下标，第一次调用时nextIndex()时返回的值为0
                int index = iterator.nextIndex();
                // 返回下一个元素存放的对象引用，返回之后，我们将该引用称为当前对象引用
                SmallClass obj = (SmallClass)iterator.next();
                // 调用当前对象引用的get()方法
                System.out.println("Element[" + index + "] = " + obj.get());
                // 如果当前对象引用的对象是new SmallClass(20)则删除它，删除该元素之后，
                // 所有在它后面的元素下标将减1.
                // 如果当前对象引用的对象不是new SmallClass(20)，则将其修改为
                // 指向新的对象引用，也就是说使得在index处的元素指向新的对象实例
                if (obj.get() == 20) iterator.remove();
                else iterator.set(new SmallClass(obj.get()+1));
                //在当前对象引用的前面插入新的元素，该元素的下标将是上面返回的index。
                /*因为下一次调用nextIndex()返回的是当前对象引用的下一个元素，由于插入新的元
素之后，当前对象及其以后的对象的下标都会加1，所以下一次调用nextIndex()将返回index+2。*/
                iterator.add(new SmallClass(obj.get()*10));
            }
            System.out.println("往后遍历的结果：");
            while (iterator.hasPrevious()) {
                int index = iterator.previousIndex();
                SmallClass obj = (SmallClass)iterator.previous();
                System.out.println("Element[" + index + "] = " + obj.get());
            }
        }
    }
```

运行结果：

```
    向前遍历的结果：
    Element[0] = 10
```

```
Element[2] = 20
Element[3] = 30
```
往后遍历的结果：
```
Element[4] = 300
Element[3] = 31
Element[2] = 200
Element[1] = 100
Element[0] = 11
```
上述程序只是用于演示，实际编程时应尽量避免在遍历的同时增加、删除元素，最好将增加、删除和修改操作放在不同的遍历过程。显然由于通用迭代器只允许单向遍历且只允许删除当前元素，它的用法比列表专用迭代器的用法要简单得多。

12.4 类集容器

创建类集容器的类都必须实现接口Collection 或其后代接口，这一节先介绍接口Collection 提供的基本方法，然后举例说明列表容器和集合容器的用法。

12.4.1 类集容器接口

类集容器的特点是只能存放单个对象。类集容器接口Collection 描述了类集容器应该实现的插入、删除、查找、迭代等基本方法，其结构如下：

```
public interface Collection<E> extends Iterable<E> {
    /*基本操作*/
    int size();  //判断当前容器所存放对象的数目
    boolean isEmpty(); //判断当前容器是否为空
    //查找当前容器是否包含与参数指定的对象o相等的对象
    boolean contains(Object o);
    //向当前容器添加对象o 以确保容器内包含该对象
    boolean add(E element);
    //在当前容器中删除与参数指定的对象o相等的对象
    boolean remove(Object o);
    //返回一个可用于迭代（遍历）当前容器的迭代器对象
    Iterator<E> iterator();
    //判断当前容器对象本身与参数指定的对象o 是否相等
    boolean equals(Object o);
    /*成批操作*/
    //对于参数指定的容器c 中的每个对象，查找当前容器是否包含与之相等的对象
    boolean containsAll(Collection<?> c);
    //将参数指定的容器c 中的对象都添加到当前容器
    boolean addAll(Collection<? extends E> c);
    //在当前容器中删除那些与参数指定的容器c 内某个对象相等的对象
    boolean removeAll(Collection<?> c);
    //在当前容器中删除那些不与参数指定的容器c 内某个对象相等的对象
    boolean retainAll(Collection<?> c);
    //删除当前容器中所有的对象
    void clean();
    /*数组操作*/
    //将当前容器转换为对象数组
```

```
    Object[] toArray();
    //将当前容器转换为对象数组，与Object[] toArray()不同的是，这里可指定数组参数a
    <T> T[] toArray(T[] a);
}
```

在上述方法中，判断两个对象是否相等都是通过调用该对象的equals()方法完成，因此放入容器的对象必须重定义合适的equals()方法以确定两个对象是否具有逻辑（通常是深比较）意义上的相等。重定义的equals()方法必须使得对象相等是一个等价关系，且应检查进行比较的两个对象所属的类是否有正确的类型，是否都是空引用，是否自己与自己比较等。

重定义equals()方法的同时也应重定义hashCode()方法，以保证调用equals()方法比较相等的两个对象返回相同的散列码。hashCode()方法用于将对象放入使用散列表实现的对象容器时计算对象的散列码。

虽然从概念上说，对象容器存放的是对象，但对于Java 语言的对象模型而言，因为只能通过对象引用访问对象实例，因此对象容器存放的总是对象引用而非对象实例。对象容器添加、删除的都是对象引用，这些操作不会影响该引用指向的对象实例。只有通过迭代器获取容器内的对象引用后，将其向下转换为合适的类类型，并调用这个类的方法才可能改变对象实例的状态。

12.4.2 列表容器

列表容器是一种类集容器，它也只存放单个对象。列表容器的特点是能够访问对象存放的位置信息，该信息作为下标能随机访问列表容器内的对象。列表容器还允许存放重复的对象，并且除了有通用迭代器支持单向遍历外，还有列表专用迭代器支持双向遍历。

列表容器类以实现接口List为标志。接口List抽象描述了列表容器的基本操作，它是类集容器接口Collection 的子接口，继承了该接口的所有方法，除此之外，还定义了一些方法体现列表容器的特点，包括列表容器使用下标插入、删除、修改和访问对象的方法、返回对象下标的方法以及返回列表专用迭代器的方法等，该接口的结构如下：

```
public interface List<E> extends Collection<E>{
    //按位置访问
    E get(int index);           //取指定下标的元素
    E set(int index,E element); //设置指定下标的元素
    boolean add(E element); //向当前容器添加元素。添加成功返回true，否则返回false。
    void add(int index,E element); //在指事实上位置index处加入元素,原来的对象及后面的对象（如果有的话）都将后移
    E remove(int index); //删除指定下标的元素,返回被删除元素
    //搜索
    int indexOf(Object o);   //返回列表中第一次出现的指定元素的下标,或者-1(无)
    int lastIndexOf(Object o); //返回列表中最后出现的指定元素的下标,或者-1(无)
    //
    ListIterator<E> listIterator();  //从表头开始遍历
    ListIterator<E> listIterator(int index);//从指定位置开始遍历
    //显示一段范围的元素
    List<E> subList(int from,int to);
}
```

1、列表容器的实现

Java有两种通用目的的List实现：ArrayList和LinkedList,前者提供固定时间的按位置访问，插入、删除消耗线性时间，通常性能比较好；后者的插入、删除消耗固定时间，在经常需要插入删除操作时性能稍好。

（1）基本操作

创建List对象格式：

ArrayList<Type> list = new ArrayList<Type>();

LinkedList<Type list = new LinkedList<Type>();

再用add()加入列表元素：

list.add(Type element);

对已有的列表，可用get()方法等按位置访问：

list.get(int i);

例12.5 ArrayList实现示例。

```
import java.util.*;
public class Example12_5 {
    public static void main(String args[]){
        ArrayList<Integer> list = new ArrayList<Integer>();
        for(int i=0;i<10;i++)
            list.add(i);
        System.out.println(list);
        list.set(5,list.get(5)*2);
        System.out.println(list);
        list.remove(9);
        System.out.println(list);
        System.out.println("第3个元素是:"+list.get(3));
    }
}
```

运行结果：

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 10, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 10, 6, 7, 8]
第3个元素是:3
```

上面的例子，将ArrayList类型改为LinkedList类型，运行结果一样。

（2）搜索操作

List有两个搜索方法：找指定元素在列表中第一次出现的位置和最后出现的位置，没有则返回-1。

例12.6 List搜索示例

```
import java.util.*;
public class Example12_6 {
    public static void main(String[] args){
        Random r = new Random();
        LinkedList<Integer> list = new LinkedList<Integer>();
        for(int i=0;i<10;i++)
            list.add(r.nextInt(10));
        System.out.println(list);
        System.out.println(list.indexOf(5));
        System.out.println(list.lastIndexOf(6));
    }
}
```

运行结果：

```
[0, 7, 4, 1, 4, 9, 1, 8, 4, 5]
5在列表中第一次出现的位置是:9
6在列表中最后出现的位置是:-1
```

（3）列举操作

List使用的ListIterator扩充了Collection接口的iterator，可以从两个方向遍历列表，可在遍历期间个性列表，而且获得iterator的当前位置。ListIterator接口结构见12.3.3节，例子见12.4。

12.4.3 集合容器

集合容器也是一种类集容器，也只能存放单个对象。集合容器的特点是它是数学上集合的实现，不允许存放重复的对象，不能访问对象存放的位置信息，因此只能单向遍历，不能随机访问集合容器内存放的对象。普通集合容器存放的对象之间没有序，但有序集合容器存放的对象可以通过比较而确定一个序关系，遍历有序集合容器的对象时将得到排好序的对象序列。

集合容器类以实现接口Set为标志。接口Set 抽象描述了集合容器的基本操作，它是类集容器接口Collection 的子接口，继承了该接口的所有方法，而且实际上没有给出新的方法，但为了体现集合容器的特点，进一步明确了其中一些方法的语义，这包括：

（1）插入元素的方法add()和addAll()都不能插入已经在集合容器内的元素，进一步addAll(Collection c)方法相当于将当前集合与类集容器c（也将其看作集合）作集合的并运算；

（2）删除元素的removeAll(Collection c)方法相当于将当前集合与类集容器c作集合的差运算；

（3）删除元素的retainAll(Collection c)方法相当于将当前集合与类集容器c作集合的交运算；

（4）查找元素的containsAll(Collection c)方法相当于判断看作集合的类集容器c是否是当前集合的子集。

1、集合容器的实现

Java有三种通用目的的Set实现：HashSet、TreeSet和LinkedHashSet。

HashSet 的特点是内部使用散列表存放容器，当散列函数产生的冲突概率小时，它的性能比使用红黑树实现的集合容器类TreeSet 要好，但在最坏情况下（即散列函数总是产生冲突时），它的性能比集合容器类TreeSet 要差得多。HashSet 创建的是普通集合容器，在接口Set 的基础上它没有定义任何新的方法，但它的构造方法允许设置初始容量和装载因子。存入类HashSet 的容器的对象可能需要重定义hashCode()方法（特别是当它重定义了equals()方法的时候）以帮助类HashSet 的散列函数得到好的散列码。

例12.7 集合容器的集合运算

```
import java.util.*;
public class Example12_7 {
    public static void main(String[] args) {
        Set<String> uniques = new HashSet<String>();
        Set<String> duplicates = new HashSet<String>();
        String[]s = {"a","book","a","pen"};
        for (int i = 0; i < s.length; i++) {
            if (!uniques.add(s[i])) duplicates.add(s[i]);
        }
        uniques.removeAll(duplicates);
        System.out.println("uniques中的单词: " + uniques);
        System.out.println("duplicates中的单词: " + duplicates);
    }
}
```

运行结果：

```
uniques中的单词: [pen, book]
duplicates中的单词: [a]
```

上例完成以下功能：在字符串s中只出现一次的单词放在集合容器uniques中，多次出现的单词放在集合容器duplicates 中，这通过将重复出现（第二次添加到uniques失败）的单词添加到duplicates，然后使用removeAll()方法作这两个集合的差运算可得。

与集合容器类TreeSet相比，除性能方面的差别外，HashSet 与TreeSet 的最大差别在于存放的对象是否有序，ThreeSet用红-黑树存放元素，按元素值排序。类HashSet 的容器存放的对象无序，而类TreeSet的容器存放的对象有序。因此，存入类TreeSet 的容器的对象必须实现接口Comparable 或者存在比较子（实现接口Comparator 的类的对象）能比较这些对象。下面例12.8使用例12.4中定义的类SmallClass演示这两个集合容器在存放对象方面的差别。注意类SmallClass已经重定义了equals()和hashCode()方法，并实现了接口Comparable。

例 12.8 HashSet 与 TreeSet 的比较

```
import java.util.*;
public class Example12_8 {
    public static void main(String[] args) {
        Set<SmallClass> hashSet = new HashSet<SmallClass>();
        for (int i = 0; i < 5; i++) {
            int temp = (int)(Math.random() * 100);
            hashSet.add(new SmallClass(temp));
        }
        System.out.println("Hash Set: " + hashSet);
        Set<SmallClass> treeSet = new TreeSet<SmallClass>();
        for (int i = 0; i < 5; i++) {
            int temp = (int)(Math.random() * 100);
            treeSet.add(new SmallClass(temp));
        }
        System.out.println("Tree Set: " + treeSet);
    }
}
```

运行结果：

```
Hash Set: [The SmallClass's x=2, The SmallClass's x=4, The SmallClass's x=9,
The SmallClass's x=8, The SmallClass's x=7]
Tree Set: [The SmallClass's x=0, The SmallClass's x=2, The SmallClass's x=5,
The SmallClass's x=9]
```

上面的输出结果表明类 HashSet 的对象容器存放的对象无序，而类 TreeSet 的对象容器存放的对象有序。该输出结果中，类 TreeSet 的对象容器存放的对象只有 4 个，这是因为在这一次运行中随机生成了两个相同的整数，而集合容器不允许添加重复对象。

LinkedHashSet 是链表实现的哈希表，按元素插入集合的顺序排次序，次序不混乱，但代价高一些，速度介于 HashSet 和 TreeSet 之间。由于 HashSet 速度快，因此，除了有排序需求外，一般都用 HashSet。

12.5 映射容器

创建映射容器的类都必须实现接口 Map 或其后代接口，这一节先介绍接口 Map 提供的基本方法，然后举例说明映射容器的用法。

12.5.1 映射容器接口

映射容器的特点是存放键——值对象对。每一个键对象都惟一地对应一个值对象，但不同的键对象

可以对应相同的值对象。因此，映射容器内存放的键对象不允许重复，所有的键对象相当于构成了一个集合容器。映射容器内存放值对象则可以重复，所有的值对象相当于构成了一个类集容器，既可相当于集合容器，也可相当于列表容器。映射容器接口 Map 描述了映射容器应该实现的插入、删除、查找、转换等基本方法，其接口的结构如下：

```
public interface Map <K,V>{
    //基本操作
    V put(K key ,V value);  //将映射容器内的键对象key关联到值对象value。
    V get(K key);      //返回映射容器内关联到键对象key 的值对象
    //删除映射容器内由键对象key 指定的键——值对象对，返回键对象key 在容器内原先关联的值对象
    V remove(K key);
    // 判断映射容器内是否包含键对象key，如果包含则返回true，否则返回false
    boolean containsKey(Object key);
    //判断映射容器内是否包含值对象value，如果包含则返回true，否则返回false。
    boolean ContainValue(Object value);
    int size(); //判断当前容器所存放键——值对象对的数目。
    boolean isEmpty(); //判断当前容器是否为空，如果为空返回true，否则返回false。
    //成批操作
    //相当于对参数指定的映射容器m的每一个键——值对象对，调用当前映射容器的方法put(K key, V value)。
    void putAll(Map<?extends K,?extends V> m);
    void clear(); //将映射容器清空。
    //收集视图
    public Set<K> keySet(); //将映射容器存放的所有键对象转换为集合容器
    public Collection<V> values(); //将映射容器存放的所有值对象转换为类集容器
    //将映射容器转换为集合容器，该集合容器中存放类型为Map.Entry 的键——值对象对
    public Set<Map.Entry<K,V>> entrySet();
    //其他
    boolean equals(Object o) //判断当前容器对象本身与参数指定的对象o 是否相等
    int hashCode()//返回当前容器的散列码
    //entrySet元素接口
    public interface Entry<K,V>{
        K getKey();
        V getValue();
        V setValue();
    }
}
```

Map 有个嵌套的接口 Map.Entry，用来实现按键-值对列举 Map。

与使用类集容器类似，我们需要注意几点：1) 容器内对象比较总是调用对象的 equals()方法，因此放入容器的对象应适当地重定义该方法及 hashCode()方法；2) 容器内存放的总是对象引用而不可能是对象实例，容器提供的添加、删除操作不会影响对象实例的状态，只有通过向下强制类型转换后才能调用容器内存放的对象的方法；3) 容器内不应该存放类型完全不相关的对象，而应当存放某个类或其后代类的对象，一个好的习惯是使用注释说明容器内存放对象的动态类型，特别是在使用多个容器的时候；4) 在使用对象容器时，用尽可能抽象的接口类型作为容器的类型，尽量使用各容器共同的方法，而避免使用

某种容器类专用的方法。

12.5.2 映射容器类

常用的映射容器类包括类 HashMap、TreeMap 和 LinkedHashMape。它们的性能与 HashSet、TreeSet 和 LinkedHashSet 类似。HashMap 内部使用散列表存放元素，存放的元素之间没有序，实现的是普通映射容器。类 TreeMap 是有序映射容器，存放的元素之间有序，这个序通过存放的对象实现接口 Comparable 或在创建容器是指定比较子（实现接口 Comparator 的对象）而确定。

例 12.10 Map 对象操作示例

```
import java.util.*;
public class Example12_10 {
    public static void main(String[] args)
    {
        HashMap<String,Integer> m1 = new HashMap<String,Integer>();
        m1.put("Math", 98);
        m1.put("Physics",82);
        m1.put("English",91);
        m1.put("Chemistry", 88);
        System.out.println(m1);
        System.out.println("输出所有键:");
        for(String s:m1.keySet())
            System.out.print(s+"\t");
        System.out.println("\n输出所有值:");
        for(Integer s:m1.values())
            System.out.print(s+"\t");
        System.out.println("\n英语成绩为:"+m1.get("English"));
        //Map.Entry接口的使用,实现按键-值对列举Map
        for(Map.Entry<String, Integer> e:m1.entrySet())
            if(e.getValue()>90)
                System.out.println(e.getKey()+" is good");
    }
}
```

运行结果：

```
{English=90, Physics=82, Math=98, Chemistry=88}
输出所有键:
English Physics    Math    Chemistry
输出所有值:
90  82  98  88
英语成绩为:91
English is good
Math is good
```

小结

对象容器是能够容纳对象的对象，数组是一种最基本的对象容器，它能存放基本数据类型的元素。Java API 提供了更多的对象容器，这些对象容器只能存放引用类型的元素。对象容器是程序组织和操纵批量数据的有力工具，可提高程序员的开发效率。

遍历对象容器存放的所有对象是对象容器的一个重要操作。迭代器是对这种操作的抽象描述。迭代

器模式将对象容器的迭代行为和其它行为分离开，使得一个容器可支持多种不同的迭代器方式，也使得一个容器可被多个迭代器并发遍历。

Java API 提供的对象容器根据存放对象的形式可分为类集容器和映射容器。类集容器存放的是单个对象，而映射容器存放的是键——值对象对。

根据存放对象形成的结构特点，类集容器又可分为列表容器和集合容器。列表容器的特点是能够存放重复的对象，存放对象的位置信息作为下标使得使用者可随机访问列表容器中的对象。Java API 的列表容器类主要有 ArrayList 和 LinkedList，类 ArrayList 的容器内部使用数组存放对象，随机访问元素的效率较高，但插入和删除元素的效率较低。类 LinkedList 的容器内部使用双向链接表存放对象，插入和删除元素的效率较高，但随机访问元素的效率较低。

集合容器的特点是不允许存放重复的对象，而且存放对象的位置信息对于使用者没有意义。除普通集合容器外，Java API 还提供了有序集合容器，它的特点是存放的对象之间有序，这种序由存放的对象实现接口 Comparable 的 compareTo()方法确定，或者由创建容器时指定的比较子（实现接口 Comparator 的对象）确定。Java API 的主要集合容器类有 HashSet 和 TreeSet，其中类 HashSet 的对象容器是普通集合容器，内部使用散列表存放元素，而类 TreeSet 的对象容器是有序集合容器，内部使用红黑树存放元素。

映射容器存放的是键——值对象对，其中一个键对象惟一地决定一个值对象，但不同的键对象可关联相同的值对象。映射容器内存放的所有键——值对象可转换为一个集合容器，存放的键对象也可转换为一个集合容器，所有的值对象则可转换为一个类集容器。映射容器自己没有迭代器可供遍历，但它可使用转换得到的类集容器进行遍历。

习题

一、基本概念

1、什么是对象容器？数组是否是对象容器？对象容器有什么作用？

2、Java API 提供的类集容器有何特点？它又可分为什么种类的容器，各自有何特点？

3、Java API 提供的映射容器有何特点？它又可分为什么种类的容器，各自有何特点？

4、什么是迭代器？为什么要采用迭代器模式？Java API 设计的迭代器有何特点？

二、编程实践

1、请编写程序演示列表容器的插入操作，该程序使用类 ArrayList 创建列表容器，存放的元素为类 Student（自己设计）的对象。演示在列表容器末尾添加一个元素、在指定位置添加一个元素、在列表容器末尾添加另一个列表容器、在指定位置添加另一列表容器等操作。请使用通用迭代器打印出列表容器每一个位置的元素及其下标信息。

2、请编写程序演示列表容器的删除操作，该程序使用类 ArrayList 创建列表容器并存放一定数目的类 Student（自己设计）的对象。演示删除一个指定的对象，删除指定位置的一个对象、删除在另一列表容器内的元素、删除不在另一列表容器内的元素等操作。请使用列表专用迭代器按下标递减的顺序打印列表容器每一个位置的元素及其下标信息。

3、请编写程序使用类 LinkedList 创建列表容器，并将一定数目用随机值创建的类 Student（自己设计）对象添加到列表容器。然后查找该列表容器内是否有两个整数值相同的对象，如果有则删除这两个对象之间的所有元素。

4、请编写程序演示映射容器的用法。程序使用类 HashMap 创建映射容器，存放的元素是以类 PhoneNumber（自己设计）的对象为键对象，以类 Person（自己设计）的对象为值对象。演示添加对象对、指定键对象删除对象对、查找某键对象的值对象、判断是否包含某键对象、判断是否包含某值对象，按键对象打印出容器内所有元素等操作。

第 13 章 JDBC 与数据库应用

数据库是存储和处理数据的重要工具，是许多企业应用系统的基础结构。本章简单介绍如何使用 Java 语言访问和操纵关系数据库，为将来进一步学习数据库编程技术奠定基础。

JDBC(Java Database Connection)是一种中间件，实现 Java 应用程序与数据库之间的接口功能。JDBC 提供了一种抽象，这种抽象建立在 SQL 的基础上，应用程序可嵌入 SQL 以访问和操作数据库。这使得应用程序可独立于特定的关系数据库管理系统。

JDBC 为应用程序提供一些编程接口，这些编程接口定义在 Java API 的 java.sql 程序包以及 JDK 1.2 版以后扩展的 javax.sql 程序包中。JDBC 与数据库管理系统之间则通过安装不同的（可能与特定数据库管理系统相关的）驱动程序进行通信。驱动程序负责解释应用程序使用 JDBC 编程接口嵌入的 SQL 语句，从而访问特定数据库管理系统中的数据库，然后返回相应的结果。应用程序利用 JDBC 编程接口使用返回的结果，完成对数据库的访问。

ODBC 也是应用程序与数据库连接的中间件，但与 JDBC 提供 Java 语言编程接口不同，ODBC 提供的是 C 语言编程接口。JDBC-ODBC 桥接驱动器负责将 ODBC 提供的 C 语言编程接口转换为 Java 语言编程接口。

ODBC 是一种已经广泛使用的数据库中间件，许多数据库管理系统供应商已经为它编写了相应的数据库驱动程序。本章给出的数据库编程例子都将采用 JDBC-ODBC 桥接驱动器，因此接下来将介绍如何设置 ODBC 数据源，从而建立起通过 JDBC-ODBC 桥接驱动器使用 JDBC 进行数据库编程的运行环境。

13.1 建立使用JDBC 的运行环境

我们拟采用 JDBC-ODBC 桥接驱动器作为 JDBC 的驱动器，为此需要设置 ODBC 数据源使得 ODBC 能通过相应的数据库驱动程序访问数据库。我们以 Windows XP 为操作系统平台，以 Access 数据据库 Student 为数据源，说明设置 ODBC 数据源的方法。

首先配置一个 ODBC 数据源。打开 Windows 中的控制面板，对于 Windows XP，选择“性能和维护”-“管理工具”-“数据源（ODBC）”；对于 Windows2000，选择“管理工具”-“数据源（ODBC）”。

（1）打开“数据源（ODBC）”，将出现 ODBC 数据源管理器对话框，如图 13_1 所示，该对话框显示了已有数据源的名称。

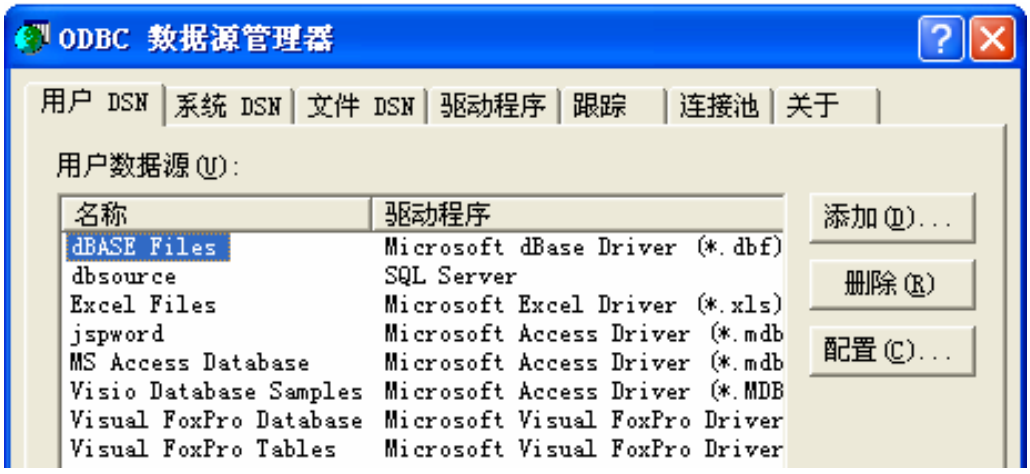

图 13_1 “ODBC 数据源管理器”对话框

（2）在 ODBC 数据源管理器对话框中选择“用户 DNS”，然后单击“添加”按钮，将出现安装数据源的驱动程序对话框，可以在选择列表中选择相应的驱动程序。由于我们将要创建的数据源是 Access 数据库，所以选择 Microsoft Access Driver(*.mdb)，如图 13_2 所示。

图 13_2 “选择驱动程序”对话框

（3）在“选择驱动程序”对话框中选择好驱动程序后，单击“完成”按钮，将出现“创建数据源”对话框，如图 13_3 所示。在该对话框中，需要为创建的数据源起一个名称，并为创建的数据源选择一个数据库。在这里新创建的数据源的名称为 student，选择的数据库是 E:\chapter13\student.mdb。如果要为数据源 student 设置一个 login 和 password 的话，就再单击“高级”按钮。最后单击创建数据源对话框上的“确定”按钮完成数据源的创建。

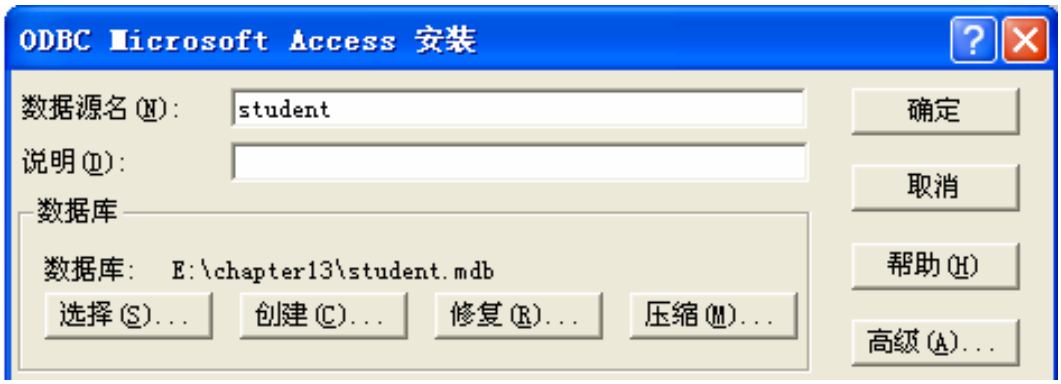

图 13_3“创建数据源连接到数据库 student.mdb 的数据源 student”对话框

13.2 使用JDBC-ODBC桥接器访问数据库

1．建立 JDBC— ODBC 桥接器

现在我们有了一个数据源，这个数据源就是一个数据库。

为了要连接到这个数据库，我们首先要建立一个 JDBC-ODBC 桥接器：

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

这里，Class 是包 java.lang 中的一个类，该类通过调用它的静态方法 forName 就可以建立 JDBC-ODBC 桥接器。

建立桥接器时可能发生异常，因此要捕获这个异常。所以建立桥接器的标准是

```
try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
}catch(ClassNotFoundException e) {}
```

2． 连接到数据库

首先使用包 java.sql 中的 Connection 类声明一个对象，然后再使用类 DriverManager 调用它的静态方法 getConnection 创建这个连接对象：

```
Connection con = DriverManager.getConnection("jdbc:odbc:数据源名字", " login name"," password ");
```

假如没有为数据源设置 login name 和 password，那么连接形式如下：

```
Connectioncon = DriverManager. getConnection("jdbc:odbc:数据源名字", "","");
```

为了能和数据源 sd 交换数据，建立 Connection 对象，建立连接时应捕获 SQLException 异常：

```
try{
    Connection con = DriverManager. getConnection("jdbc:odbc: student ", "gxy", "123");
}   catch(SQLException e){}
```

这样就建立了到数据库 student.mdb 的连接。

3．向数据库发送 SQL 语句

获得数据库连接后，可创建 SQL 语句对象以便发送和执行 SQL 语句，通常使用接口 Connection 的以下方法创建 SQL 语句对象：

```
Statement createStatement();
PreparedStatement PreparedStatement(String sql);
CallableStatement prepareCall(String sql);
```

上面创建了三种 SQL 语句对象：类型为接口 Statement、PreparedStatement 以及 CallableStatement 的对象。Statement 类型的对象可发送和执行没有参数的 SQL 语句，PreparedStatement 类型的对象可发送和执行带参数的 SQL 语句，这些参数可让 Java 应用程序动态地指定 SQL 语句某些位置应该取的值，例如指定 where 后面条件表达式中与字段名进行比较的值、SQL 修改语句、插入语句中所要设置的字段值等。CallableStatement 类型的对象则可发送和执行数据库的存储过程。

例如，使用 Statement 声明一个 SQL 语句对象，然后通过刚才创建的连接数据库的对象 con 调用方法 createStatment()创建这个 SQL 语句对象。

```
try {
    Statement sql=con.createStatement();
 }catch(SQLException e ){}
```

创建类型为接口 Statement 的对象后，则可使用该对象发送和执行 SQL 语句，这通过接口 Statement 的下述方法实现：

```
ResultSet executeQuery(String sql);
int executeUpdate(String sql);
```

应根据不同的 SQL 语句调用上面不同的方法，当执行的是 SQL 查询语句时调用方法 executeQuery()，而当执行的是 SQL 数据更新语句（包括 SQL 插入语句、修改语句和删除语句）时就应该调用方法 executeUpdate()。

创建类型为接口 PreparedStatement 和 CallableStatement 的对象时都应预先指定要执行的 SQL 语句，但这时的 SQL 语句含有参数。SQL 语句的参数使用 '?' 占位，在发送和执行 SQL 语句前，必须先使用接口 PreparedStatement 提供的各种 SetXxx()方法设置 SQL 语句参数应取的值，然后再调用以下方法发送和执行预先指定的 SQL 语句：

```
ResultSet executeQuery();
int executeUpdate();
```

例如，下述语句创建一个类型为 PreparedStatement 的对象：

```
String sql = "SELECT * FROM StudentTable WHERE name = ? AND birthday <= ?";
PreparedStatement stm = connection.prepareStatement(sql);
```

注意，在预先指定的 SQL 语句 sql 应使用'?'占据要设置值的参数的位置，然后使用以下方法设置它

们的值：

```
stm.setString(1, new String("张三"));
stm.setDate(2, new Date(1970, 2, 2));
```

设置 SQL 语句的参数值时，方法 SetXxx()的第一个参数指定要设置的 SQL 语句参数的位置，即第几个'?'所对应的参数，第二个参数要根据不同的 SQL 语句参数类型使用不同的值，SQL 语句参数的类型也决定了所调用方法 SetXxx()的名称。

例如在上面的例子中，表 StudentTable 的字段 name 应该与字符串比较，所以设置第一个 SQL 参数应调用方法 setString()，而字段 birthday 应该与日期值比较，所以设置第二个 SQL 参数应调用方法 setDate()，在后面我们将专门讨论数据库字段类型与 Java 语言的数据类型间的映射关系，这种映射关系可指导程序员使用合适的方法设置 SQL 语句参数的值。

接口 CallableStatement 用于执行数据库的存储过程，由于本书不打算讨论存储过程的用法，因此本书也不打算进一步讨论 CallableStatement 的用法，但它作为接口 PreparedStatement 的子接口，许多用法与接口 PreparedStatement 的用法类似。

4．处理查询结果

不管使用哪一种 SQL 语句对象发送和执行 SQL 语句，在执行 SQL 查询语句时都返回类型为接口 ResultSet 的对象，执行 SQL 更新语句则返回执行该语句后数据表受影响的数据记录数。

类型为 ResultSet 的对象存放了执行 SQL 查询语句所返回的结果。接口 ResultSet 支持迭代器访问模式，可使用如下方式访问 SQL 查询语句的执行结果：

```
ResultSet result = stm.executeQuery();
while (result.next()) {
    String id = result.getString("id");
    String name = result.getString("name");
    boolean sex = result.getBoolean("sex");
    Date birthday = result.getDate("birthday");
    System.out.println(id + "\t" + name + "\t" + sex + "\t" + birthday);
}
```

可将类型为 ResultSet 的对象存放的数据看作是由一条条数据记录组成，这些数据记录具有的字段由所执行的 SQL 查询语句的 SELECT 后的目标字段列表确定。第一次调用 ResultSet 的 next()方法将定位（也称将查询结果集的游标移动）到第一条数据记录，然后每调用一次 next()方法就定位到下一条数据记录。

ResultSet 对象可以使用位置索引，第一列使用 1，第二列使用 2 等等或使用列名称，以便使用 getXxx()方法获得字段值。接口 ResultSet 提供各种 getXxx()方法用于获取当前数据记录的某字段的值，字段的数据类型以及 SQL 数据类型与 Java 数据类型的映射关系决定使用哪个 getXxx()方法，例如取 id 字段的值应使用 getString()方法，取 birthday 字段的值则应使用 getDate()方法等等。表 13-1 给了出了 ResultSet 对象的若干方法。

表 13-1 ResultSet 类的若干方法

返回类型	方法名称
boolean	next()
byte	getByte(int columnIndex)
Date	getDate(int columnIndex)
double	getDouble(int columnIndex)
float	getFloat(int columnIndex)
int	getInt(int columnIndex)
long	getLong(int columnIndex)

String	getString(int columnIndex)
byte	getByte(String columnName)
Date	getDate(String columnName)
double	getDouble(String columnName)
float	getFloat(String columnName)
int	getInt(String columnName)
long	getLong(String columnName)
String	getString(String columnName)

5．释放占用的资源

每个 SQL 语句对象只能有一个查询结果集，这意味着在执行下一次查询之前，必须先处理好这一次的查询结果。PreparedStatement 类型的对象在发送和执行完 SQL 语句后，可使用该接口提供的 clearParameters()清除所设置的参数值，然后程序可重新设置参数，再使用新的参数发送和执行创建 PreparedStatement 类型的对象时所预先指定的 SQL 语句。

在使用完查询结果集，或执行完 SQL 更新语句后，就完成了对数据库的一次访问。接口 ResultSet、Statement、PreparedStatement、Connection 都提供了 close()方法分别释放查询结果集、SQL 语句对象、数据库连接所占用的资源并将它们关闭。在使用完后，将这些对象进行关闭是好的习惯，例如应先关闭查询结果集，而发送和执行完所有的 SQL 语句后，则应关闭 SQL 语句对象，当不再准备访问数据库后就应关闭数据库连接等等。

总的来说，使用 JDBC 进行数据库访问的步骤主要包括：

（1）向驱动器管理器注册数据库驱动程序；

（2）通过驱动器管理器类获取与要访问的数据源之间的数据库连接；

（3）创建发送和执行 SQL 语句的 SQL 语句对象；

（4）必要时设置 SQL 语句的参数值；

（5）发送和执行 SQL 语句；

（6）当执行的是 SQL 查询语句时，获得查询结果集，运用迭代器模式访问查询结果集；

（7）关闭与数据库访问有关的对象，释放其占用的资源。

图 13_4 给出了上面这些步骤所设计的类或接口，以及它们之间的关系，例 13.1 给出了一个完整例子说明这些步骤：

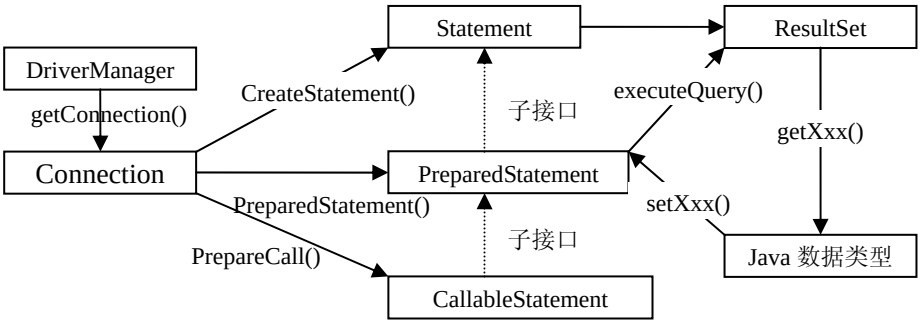

图 13_4 使用 JDBC 访问数据库涉及的类与接口

下面的例子针对 Access 数据库 student.mdb 中表名为 StudentTable 的表进行操作，该表中的记录如图 13_5 所示：

StudentTable : 表

学号	姓名	性别	出生	总分
20090001	张三	☑	1982-7-6	561
20090002	李四	☐	1981-8-13	533
20090003	王五	☐	1983-1-16	499
20090004	赵六	☑	1982-12-18	516

图 13_5 示例数据表 StudentTable 中的记录

例13.1

```
import java.sql.*;
public class Example13_1 {
    public static void main(String[] args) {
        String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
        String source = "jdbc:odbc:student";
        try {
            //查找用于JDBC驱动的类，这种查找会使得Java虚拟机装入该类，这个类的静态初始化
            //语句块对驱动程序进行初始化，从而下面可直接使用该驱动程序进行数据库连接，无需
            //再作其它额外的事情
            Class.forName(driver);
        } catch (ClassNotFoundException exc) {
            //当没有驱动程序时，应用程序无法继续运行，故退出程序
            System.out.println("没有发现驱动程序: " + driver);
            exc.printStackTrace(); System.exit(1);
        }
        try {
            //   建立与指定数据库的连接
            Connection connection = DriverManager.getConnection(source);
            //   如果连接成功则检测是否有警告信息
            SQLWarning warn = connection.getWarnings();
            while (warn != null) {
                System.out.println(warn.getMessage());
                warn = warn.getNextWarning();
            }
            //   创建一个用于执行预编译SQL的语句对象
            String sql = "SELECT * FROM StudentTable WHERE 学号 = ? AND 出生 <= ?";
            PreparedStatement pStm = connection.prepareStatement(sql);
            //   设置预编译SQL语句的参数值
            pStm.setString(1, new String("20090002"));
            pStm.setDate(2, new Date(1970, 2, 2));
            // 发送和执行预编译的SQL语句，获得查询结果集
            System.out.println("发送和执行预编译的SQL语句(有参数)，获得查询结果集");
            ResultSet result = pStm.executeQuery();
            //   使用迭代模式访问查询结果集
            while (result.next()) {
                String id = result.getString("学号");
```

```
                String name = result.getString("姓名");
                String sex = result.getBoolean("性别") ? "女" : "男";
                Date birthday = result.getDate("出生");
                int score = result.getInt("总分");
                System.out.println(id + "\t" + name + "\t" + sex + "\t" +
birthday+"\t"+score);
            }
            //关闭查询结果集
            result.close();
            //关闭SQL预编译语句
            pStm.close();
            //创建执行简单SQL语句的SQL语句对象
            Statement stm = connection.createStatement();
            //发送和执行简单SQL语句，获取查询结果集
            sql = "SELECT * FROM StudentTable";
            System.out.println("发送和执行简单SQL语句，获取查询结果集");
            result = stm.executeQuery(sql);
            //使用迭代模式访问查询结果集
            while (result.next()) {
                String id = result.getString("学号");
                String name = result.getString("姓名");
                String sex = result.getBoolean("性别") ? "女" : "男";
                Date birthday = result.getDate("出生");
                int score = result.getInt("总分");
                System.out.println(id + "\t" + name + "\t" + sex + "\t" +
birthday+"\t"+score);
            }
            //关闭查询结果集
            result.close();
            //关闭SQL语句
            stm.close();
            //关闭数据库连接
            connection.close();
        } catch (SQLException exc) {
            System.out.println("在执行数据库访问时发生了错误!");
            exc.printStackTrace();
        }
    }
}
```

运行结果：

```
发送和执行预编译的SQL语句(有参数)，获得查询结果集
20090002    李四男  1981-08-13 533
发送和执行简单SQL语句，获取查询结果集
20090001    张三女  1982-07-06 561
```

```
20090002    李四男   1981-08-13 533
20090003    王五男   1983-01-16 499
20090004    赵六女   1982-12-18 516
```

上例中，演示了创建数据库连接、创建及使用 Statement、PreparedStatement 返回结果集 ResultSet、并从 ResultSet 中读取数据的方法。所有与数据库访问有关的语句都可能抛出异常 SQLException，应用程序需要捕获或声明重新抛出该异常。接口 Connection、Statement 和 ResultSet 还提供了 getWarning()方法返回执行与数据库访问有关的操作时数据库驱动程序提供的警告信息，这些信息可供程序员调试程序时参考。

13.3 Java 与SQL 间的数据类型映射

Java 语言的数据类型与数据表的字段类型并不完全相同，而且不同的数据库管理系统也有可能支持不同的字段类型，例如在中文版 Microsoft Access 2000 支持的字段类型是文本、数字、日期、是 / 否、备注等类型。

为了在 Java 应用程序中正确访问数据表内容，必须在 Java 语言的数据类型与数据表的字段类型之间进行映射。这分两步完成，首先 Java 应用程序在 Java 语言的数据类型与 JDBC 支持的 SQL 数据类型之间进行映射，然后 JDBC 的数据库驱动程序进一步负责 SQL 数据类型与实际的数据表的字段类型之间进行映射。

在使用 PreparedStatement 类型的对象发送和执行 SQL 语句时，必须先使用该接口提供的方法 setXxx()设置参数的值，这时需要根据数据表的字段类型选择合适的 setXxx()方法。在使用类型为 ResultSet 的查询结果集时，需要使用该接口提供的 getXxx()方法获取数据记录的某个字段的值，这时也需要根据数据字段类型选择合适的 getXxx()方法，并使用合适的 Java 语言的数据类型变量存放得到的字段值。表 13-2 列出了 SQL 数据类型与 Java 语言数据类型之间的映射关系，并给出了应选用的 setXxx()方法和 getXxx()方法。

表 13-2 Java 语言的数据类型与 SQL 数据类型之间的映射

SQL数据类型	Java语言的数据类型	setXxx()方法的选择	getXxx()方法的选择
BIT	boolean	setBoolean()	getBoolean()
TINYINT	byte	setByte()	getByte()
SMALLINT	short	setShort()	getShort()
INTEGER	int	setInt()	getInt()
BIGINT	long	setLong()	getLong()
FLOAT	double	setDouble()	getDouble()
REAL	float	setFloat()	getFloat()
DOUBLE	double	setDouble()	getDouble()
NUMERIC	java.math.BigDecimal	setBigDecimal()	getBigDecimal()
DECIMAL	java.math.BigDecimal	setBigDecimal()	getBigDecimal()
CHAR	String	setString()	getString()
VARCHAR	String	setString()	getString()
LONGVARCHAR	String	setString()	getString()
DATE	java.sql.Date	setDate()	getDate()
TIME	java.sql.Time	setTime()	getTime()
TIMESTAMP	java.sql.Timestamp	setTimestamp()	getTimestamp()
BINARY	byte[]	setBytes()	getBytes()
VARBINARY	byte[]	setBytes()	getBytes()
LONGVARBINARY	byte[]	setBytes()	getBytes()

13.4 修改、添加、删除记录

在建立与数据源的连接后，不仅可以发送查询语句，而且可能使用 SQL 语句更新、添加以及删除记录。

在 Statement 接口中，主要提供了 2 种执行 SQL 语句的方法，分别是 executeQuery()、executeUpdate(), 具体使用哪一种方法由 SQL 语句所产生的内容决定。

executeQuery()方法一般用于执行一条简单的查询（select）语句，13.2 节已经介绍。

executeUpdate()方法用于执行对数据库进行更新操作，如修改、插入和删除记录以及创建、修改和删除表等，该方法将返回 0 或一个整数值，这个整数值是操作所影响的记录的行数，调用方法是：

```
public int executeUpdate(String sqlStatement);
```

例如：

```
Statement statement = connection.createStatement();
statement.executeUpdate("update StudentTable set 总分=560 where 学号='20090001'");
```

该语句将 studentTable 表中学号为 20090001 的学生总分改为 560 分。其他的更新操作与此类似。

PreparedStatement 接口也可调用 executeQuery、executeUpdate()方法来执行 SQL 语句，方法的返回值与 Statement 接口中的 executeQuery 和 executeUpdate()方法一样。与 statement 不同的是，由于在创建 PreparedStatement 对象时已经给出了要执行的 SQL 语句，并进行了预编译，因此，这些方法在调用时不需要任何参数，例如：

```
PreparedStatement ps = con.preparedStatement("update StudentTable set 总分=? where 学号=? " );
ps.setInt(1,560);
ps.setString(2,'20090001');
ps.executeUpdate();
```

下面的例子 13.2 分别用 Statement 和 PreparedStatement 接口实现了对 StudentTable 表中记录的更新, 添加和删除。

例 13.2

```
import java.sql.*;
public class Example13_2 {
    public static void main(String[] args) {
        String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
        String source = "jdbc:odbc:student";
        try {
            Class.forName(driver);
        } catch (ClassNotFoundException exc) {
            System.out.println("没有发现驱动程序: " + driver);
            exc.printStackTrace(); System.exit(1);
        }
        try {
            //建立与指定数据库的连接
            Connection connection = DriverManager.getConnection(source);
            //如果连接成功则检测是否有警告信息
            SQLWarning warn = connection.getWarnings();
            while (warn != null) {
                System.out.println(warn.getMessage());
                warn = warn.getNextWarning();
```

```
            }
            //创建执行SQL的Statement对象
            Statement stm = connection.createStatement();
            //执行更新语句
            String sql = "update StudentTable set 总分=560 where 学号
='20090001'";
            stm.executeUpdate(sql);
            //创建执行SQL的PreparedStatement对象
            String psql = "update StudentTable set 总分=566 where 学号
='20090002'";
            PreparedStatement pstm = connection.prepareStatement(psql);
            //执行更新语句
            pstm.executeUpdate();
            //查询更新后的结果
            String querysql="select * from StudentTable";
            ResultSet rs = stm.executeQuery(querysql);
            while (rs.next()) {
                String id = rs.getString("学号");
                String name = rs.getString("姓名");
                String sex = rs.getBoolean("性别") ? "女" : "男";
                Date birthday = rs.getDate("出生");
                int score = rs.getInt("总分");
                System.out.println(id + "\t" + name + "\t" + sex + "\t" +
birthday+"\t"+score);
            }
            //关闭查询结果集
            rs.close();
            //关闭SQL语句
            stm.close();
            pstm.close();
            //关闭数据库连接
            connection.close();
        } catch (SQLException exc) {
            System.out.println("在执行数据库访问时发生了错误!");
            exc.printStackTrace();
        }
    }
}
```

运行结果：

```
20090001   张三女  1982-07-06 560
20090002   李四男  1981-08-13 566
20090003   王五男  1983-01-16 499
20090004   赵六女  1982-12-18 516
```

13.5 使用数据表的元信息

JDBC 允许 Java 应用程序获取数据表的元信息，所谓数据表元信息是指与数据表的字段有关的信息，例如字段名称、字段数据类型、字段长度、字段小数位长度等。

在执行 SQL 查询语句之后，通过得到的查询结果集可获取数据表的元信息（更确切地说是查询结果集的元信息）。接口 ResultSet 的方法 getMetaData() 返回类型为接口 ResultSetMetaData 的对象，该对象保存了查询结果集的元信息。接口 ResultSetMetaData 提供了一系列方法返回有关查询结果集的字段信息：

```
// 返回查询结果集的字段数目
int getColumnCount();
// 返回第index 个字段的字段名称，注意index从1 开始计算，后面的方法也如此。
String getColumnName(int index);
// 返回字段的SQL 数据类型，返回值可用于setObject()和getObject()方法
int getColumnType(int index);
// 返回字段的SQL 数据类型的名称，例如"BIT"、"SMALLINT"等
String getColumnTypeName(int index);
/*返回字段类型对应的Java数据类型的类名，当字段对应的是基本数据类型时，返回基本数据类型的包装类，例如SQL 数据类型SMALLINT 返回类名"Integer"*/
String getColumnClassName(int index);
// 返回字段精度，通常就是在设计字段时所定义的字段长度
String getPrecision(int index);
// 返回字段的小数位数，通常就是在设计字段时所定义的字段长度
String getScale(int index);
/* 返回该字段是否允许空值，接口PreparedStatement 的方法setNull(int index,int sqlType)可为SQL 语句的参数设置空值。当字段允许空值时，使用ResultSet的方法getXxx()返回的值可能为null。*/
boolean isNullable(int index);
```

例 13.3 给出了如何使用数据表元信息的一个例子，它打印出数据表 StudentTable 的数据字段的最重要信息，包括字段名称、长度、类型及所属的 Java 类。

例13.3

```
import java.sql.*;
public class Example13_2 {
    public static void main(String[] args) {
        String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
        String source = "jdbc:odbc:student";
        try {
            Class.forName(driver);
        } catch (ClassNotFoundException exc) {
            System.out.println("没有发现驱动程序: " + driver);
            exc.printStackTrace(); System.exit(1);
        }
        try {
            //建立与指定数据库的连接
            Connection connection = DriverManager.getConnection(source);
            //如果连接成功则检测是否有警告信息
            SQLWarning warn = connection.getWarnings();
```

```
            while (warn != null) {
                System.out.println(warn.getMessage());
                warn = warn.getNextWarning();
            }
            //创建执行简单SQL语句的SQL语句对象
            Statement stm = connection.createStatement();
            //发送和执行简单SQL语句，获取查询结果集
            String sql = "SELECT * FROM StudentTable";
            ResultSet result = stm.executeQuery(sql);
            ResultSetMetaData metaData = result.getMetaData();
            int columnCount = metaData.getColumnCount();
            System.out.println("字段名称\t字段类型\t字段长度\t字段所属类");
            for (int i = 0; i < columnCount; i++) {
                //注意字段下标从1开始计算
                System.out.println(metaData.getColumnName(i+1) + "\t"
                        + metaData.getColumnTypeName(i+1) + "\t"
                        + metaData.getPrecision(i+1) + "\t"
                        + metaData.getColumnClassName(i+1));
            }
            //关闭查询结果集
            result.close();
            //关闭SQL语句
            stm.close();
            //关闭数据库连接
            connection.close();
        } catch (SQLException exc) {
            System.out.println("在执行数据库访问时发生了错误!");
            exc.printStackTrace();
        }
    }
}
```

运行结果：

```
字段名称 字段类型 字段长度 字段所属类
学号 VARCHAR    8   java.lang.String
姓名 VARCHAR    8   java.lang.String
性别 BIT        1   java.lang.Boolean
出生 DATETIME   19  java.sql.Timestamp
总分 SMALLINT   5   java.lang.Short
```

13.6 JDBC 编程实例

本节在数据库 studetn 中创建成绩表，并演示新增、修改、删除、查询记录。

例 13.4 创建学生成绩表 scores 表，此表有三个字段：学号，课程名，分数

```
package app;
import java.sql.*;
public class CreateTable
```

```
{
    public static void main(String[] args)
    {
        // 声明JDBC驱动程序类型
        String JDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
        // 定义JDBC的url对象
        String conURL = "jdbc:odbc:student";
        try
        {
            // 加载JDBC驱动程序
            Class.forName(JDriver);
        }
        catch (java.lang.ClassNotFoundException e)
        {
            System.out.println("无法加载JDBC驱动程序。" + e.getMessage());
        }
        Connection con = null;
        Statement s = null;
        try
        {
            // 连接数据库URL
            con = DriverManager.getConnection(conURL);
            // 建立Statement类对象
            s = con.createStatement();
            // 创建一个含有三个字段的顾客表conumser
            String query = "create table scores(" + "学号 char(8) not null,"
                    + "课程名 char(15)," + "分数 int"
                    + ")";
            s.executeUpdate(query); // 执行SQL语句
            System.out.println("创建表成功! ");
        }
        catch (SQLException e)
        {
            System.out.println("SQLException:" + e.getMessage());
        }
        finally
        {
            try
            {
                if (s != null)
                {
                    s.close();
                    s = null;
                }
```

```
                if (con != null)
                {
                    con.close(); // 关闭与数据库的连接
                    con = null;
                }
            }
            catch (SQLException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

例 13.5 在上例创建的数据表 scores 中插入三个学生的语文成绩记录

```
package app;
import java.sql.*;
public class InsertRecord
{
    public static void main(String[] args)
    {
        // 声明JDBC驱动程序类型
        String JDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
        // 定义JDBC的url对象
        String conURL = "jdbc:odbc:student";
        try
        {
            // 加载JDBC驱动程序
            Class.forName(JDriver);
        }
        catch (java.lang.ClassNotFoundException e)
        {
            System.out.println("无法加载JDBC驱动程序。" + e.getMessage());
        }
        Connection con = null;
        PreparedStatement ps = null;
        try
        {
            // 连接数据库URL
            con = DriverManager.getConnection(conURL);
            // 建立PreparedStatement类对象
            String sql = "insert into scores values(?,?,?)";
            ps = con.prepareStatement(sql);
            // 使用SQL命令insert插入三条顾客纪录到表中
            ps.setString(1, "20090001");
```

```
            ps.setString(2, "语文");
            ps.setInt(3, 118);
            ps.executeUpdate();
            ps.clearParameters();
            ps.setString(1, "20090002");
            ps.setString(2, "语文");
            ps.setInt(3, 116);
            ps.executeUpdate();
            ps.clearParameters();
            ps.setString(1, "20090003");
            ps.setString(2, "语文");
            ps.setInt(3, 109);
            ps.executeUpdate();
            ps.clearParameters();
            System.out.println("插入数据成功! ");
        }
        catch (SQLException e)
        {
            System.out.println("SQLException:" + e.getMessage());
        }
        finally
        {
            try
            {
                if (ps != null)
                {
                    ps.close();
                    ps = null;
                }
                if (con != null)
                {
                    con.close(); // 关闭与数据库的连接
                    con = null;
                }
            }
            catch (SQLException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

例 13.6 修改上例中的第二条和第三条记录的学生成绩，并把修改后数据表的内容输出到屏幕上

```
package app;
```

```
import java.sql.*;
public class UpdateRecord
{
    public static void main(String[] args)
    {
        //声明JDBC驱动程序类型
        String JDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
        // 定义JDBC的url对象
        String conURL = "jdbc:odbc:student";
        try
        {
            // 加载JDBC驱动程序
            Class.forName(JDriver);
        }
        catch (java.lang.ClassNotFoundException e)
        {
            System.out.println("无法加载JDBC驱动程序。" + e.getMessage());
        }
        Connection con = null;
        PreparedStatement ps = null;
        Statement s = null;
        try
        {
            // 连接数据库URL
            con = DriverManager.getConnection(conURL);
            // 修改数据库中数据表的内容
            String[] id = { "20090001", "20090003" };
            int[] scores = { 98, 99 };
            ps = con.prepareStatement("UPDATE scores set 分数=?  where 学号=?");
            int i = 0;
            do
            {
                ps.setInt(1, scores[i]);
                ps.setString(2, id[i]);
                ps.executeUpdate();
                i++;
            }
            while (i < id.length);
            // 查询数据库并把数据表的内容输出到屏幕上
            s = con.createStatement();
            ResultSet rs = s.executeQuery("select * from scores");
            System.out.println("学号 \t课程 \t分数");
            while (rs.next())
            {
```

```
                System.out.println(rs.getString("学号") + "\t"
                        + rs.getString("课程名") + "\t" + rs.getInt("分数"));
            }
        }
        catch (SQLException e)
        {
            System.out.println("SQLException:" + e.getMessage());
        }
        finally
        {
            try
            {
                if (ps != null)
                {
                    ps.close();
                    ps = null;
                }
                if (s != null)
                {
                    s.close();
                    s = null;
                }
                if (con != null)
                {
                    con.close(); // 关闭与数据库的连接
                    con = null;
                }
            }
            catch (SQLException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

例 13.7 在上例创建的数据表 scores 中删除第二条记录，然后把数据表的内容输出

```
package app;
import java.sql.*;
public class DeleteRecord
{
    public static void main(String[] args)
    {
        //声明JDBC驱动程序类型
        String JDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
```

```
// 定义JDBC的url对象
String conURL = "jdbc:odbc:student";
try
{
    // 加载JDBC驱动程序
    Class.forName(JDriver);
}
catch (java.lang.ClassNotFoundException e)
{
    System.out.println("无法加载JDBC驱动程序。" + e.getMessage());
}
Connection con = null;
PreparedStatement ps = null;
Statement s = null;
try
{
    // 连接数据库URL
    con = DriverManager.getConnection(conURL);

    // 删除第二条记录
    ps = con.prepareStatement("delete from scores where 学号=?");
    ps.setString(1, "20090001");
    ps.executeUpdate();
    // 查询数据库并把数据表的内容输出到屏幕上
    s = con.createStatement();
    ResultSet rs = s.executeQuery("select * from scores");
    System.out.println("学号 \t\t课程名 \t分数");
    while (rs.next())
    {
        System.out.println(rs.getString("学号") + "\t"
                + rs.getString("课程名") + "\t" + rs.getInt("分数"));
    }
}
catch (SQLException e)
{
    System.out.println("SQLException:" + e.getMessage());
}
finally
{
    try
    {
        if (ps != null)
        {
            ps.close();
```

```
                ps = null;
            }
            if (s != null)
            {
                s.close();
                s = null;
            }
            if (con != null)
            {
                con.close(); // 关闭与数据库的连接
                con = null;
            }
        }
        catch (SQLException e)
        {
            e.printStackTrace();
        }
    }
}
```

小结

Java 语言制定了 Java 数据库连接的规范——JDBC，JDBC 使用驱动器管理器管理各种数据库驱动程序，数据库驱动程序使用各种专用的协议与数据库进行连接，并解释和执行 Java 应用程序发送的 SQL 语句，以完成对数据库的访问。

Java 应用程序使用 JDBC 访问数据库需要首先根据所访问的数据库（数据源）向 JDBC 驱动器管理器注册数据库驱动驱动程序，从驱动程序获得对数据库的连接，使用该连接创建 SQL 语句对象，利用 SQL 语句对象发送和执行 SQL 语句。当执行的是 SQL 查询语句时，还可以获得查询结果集，应用程序可访问此查询结果集，从而完成对数据库的访问。数据库访问完毕之后应该关闭数据库连接，释放其占用的资源。JDBC 支持多种形式的 SQL 数据类型，并建立了 Java 语言的数据类型与 SQL 数据类型之间的映射关系，应用程序可利用这种映射关系读取或设置数据字段的值。JDBC 还支持应用程序获得查询结果集的元信息，从而可编写通用的数据库访问工具。

习题

一、基本概念

1、简述使用 Java 环境访问数据库的主要过程。

2、简述使用 JDBC API 访问数据库需要用到的主要类及接口。

3、简述使用 JDBC 进行数据库操作的完整过程。

4、比较 Statement、PreparedStatement 和 CallableStateme 在使用上的差异。

二、编程实践

1、编写一个程序，创建一个存储电话薄信息的数据库。

2、编写一个程序，对题 1 创建的电话薄数据库进行插入、删除和更新的操作。

3、编写一个程序，给定某个人的姓名，在题 1 中创建的电话薄数据库中查找相应的信息。

4、建立一个 Books 数据库表，字段包括书名、作者、出版社、出版时间和 ISBN，并编写一个应用程序，实现对该表中记录的插入、删除、更新。

5、假定数据库 StudentManager 有两个数据表：一个是 Classes，存放班级信息，包括班级编号、班级名

称、所属年级、所属专业三个字段，其中班级编号为主码；另一个是 Students，存放学生的信息，包括学号、姓名、年龄、所属班级四个字段，其中学号为主码，所属班级为外码，来源于数据表 Classes 的班级编号。请选择你最熟悉的数据库管理系统，创建数据库 StudentManager 及它的两个数据表 Classes 和 Students，并建立它们之间的关系，并录入一定数量的记录。

6、请基于题 5 中的数据库 StudentManager 的数据表 Classes 和 Students，写出实现以下查询的 SQL 语句：

(1). 查找所有的班级信息；

(2). 查找专业为计算机科学与技术的班级；

(3). 查找 2008 级至 2009 级之间（包括这两个年级）的所有班级；

(4). 查找所有姓王的学生；

(5). 查找 2008 级计算机科学与技术专业的所有学生；

(6). 查找姓名为王五的学生所在的班级信息；

7、请基于题 5 中数据库 StudentManager 的数据表 Classes 和 Students，写出对数据表作如下更新的 SQL 语句：

(1). 插入班级编号为 20090A、名称为 2009 级计算机专业本科 A 班、所属年级为 2009 级、所属专业为计算机科学与技术专业的班级记录；

(2). 插入学号为 09035020、姓名为张小强、年龄为 18 岁、所属班级编号为 20090A 的学生记录；

(3). 修改学号为 09035020 的学生的姓名为张大强、年龄为 19 岁；

(4). 删除学号为 09035020 的学生；