



第1章 Java简介

- 本章将要介绍Java的特点以及所应用的平台，然后带领读者从第一步做起，来完成一个Java的小程序，通过这个小程序的完成，可以让读者了解Java平台的搭建以及简单的开发步骤。通过本章的学习，读者应该能够达到如下几个目标。
- 熟练掌握如何搭建Java开发环境,包括下载、安装和配置JDK。
- 能够编写和编译Java程序，并能够运行生成文件。
- 这些是学习本章的目标，同时也是对读者的基本要求。学好本章是学习以后知识的基础，读者一定要熟练地掌握本章的知识。



1.1 Java的平台简介

Java语言在网络编程方面应用得很广，作为一个新的程序设计语言，它具有简单、多变、面向对象、不依赖操作系统的特点，具有很好的移植性和安全性，这些给网络编程带来了许多便利。Java的平台根据用途来区分，可以分为三个版本。

Java SE —— Java Standard Edition，这是Java的标准版，主要用于桌面级的应用和数据库的开发。

Java EE —— Java Enterprise Edition，这是Java的企业版，提供了企业级开发的各种技术，主要用于企业级开发，现在用的最多的也就是这个。

Java ME —— Java Micro Edition，这个版本的Java主要用于嵌入式的和移动式的开发，最常用的就是手机应用软件的开发。



1.2 安装工具包

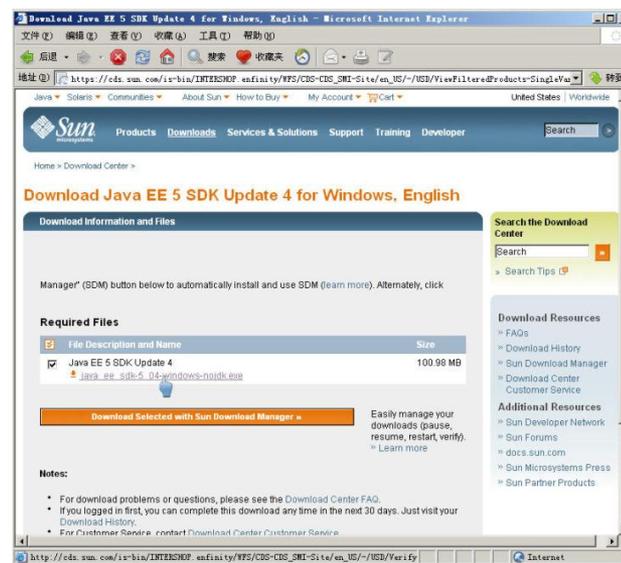
首先要进行Java JDK的安装，JDK就是提供Java服务的系统包。请根据操作系统来选择安装哪个版本的JDK。本节介绍如何安装和配置JDK的环境变量和一些常用命令。

1.2.1 下载JDK

Java的系统包为JDK，JDK的全称为Java Development Kit，可以根据不同的平台来下载不同的JDK，下面介绍在32位的Windows XP系统上，如何下载并完成配置的。



进入 Java 官方网站



单击链接进行下载

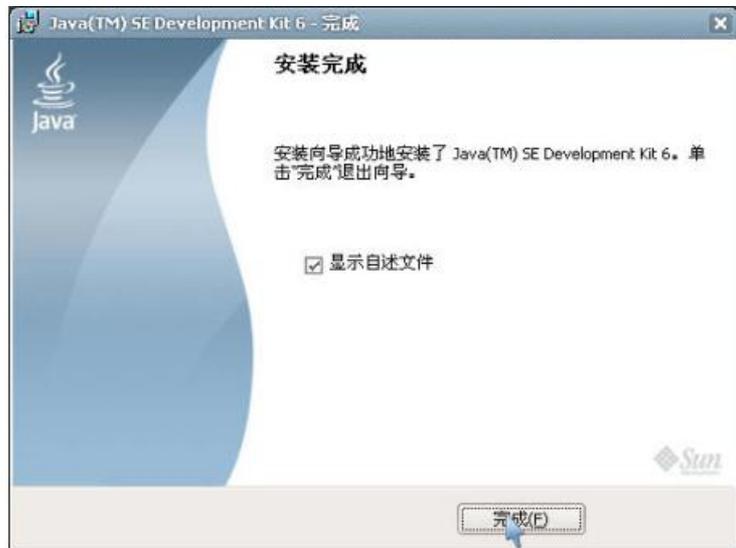


1.2.2 安装JDK

下载完成后，进行安装，下面介绍windows XP下的JDK安装步骤。



安装 JDK 的开始画面



JDK 安装完成，单击“完成”按钮结束安装

1.2.3 查看与设置环境变量

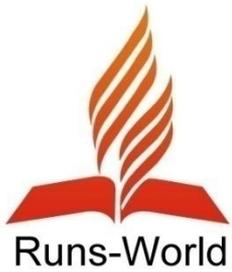
- 所谓环境变量是供系统内部使用的变量，是包含系统的当前系统用户的环境信息的字符串和软件的一个确定存放的路径，安装完JDK就必须配置环境变量，方法如下所述。



系统属性



填入新值



1.2.4 JDK常用命令

在图中显示了JDK中的一部分命令，在本节来对这些命令进行必要的讲解。

- 1. javac的常用命令
- g:: 生成调试信息。
- g:none: 生成无调试信息。
- g:{lines, vars, source}: 只生成部分调试信息。
- O: 优化, 可能增大类文件。
- nowarn: 无警告。
- verbose: 输出编译器信息。
- deprecation: 输出不鼓励使用的API的程序路径。
- classpath + 路径: 指定用户类文件的路径。
- sourcepath + 路径: 指定输入源文件的路径。
- bootclasspath + 路径: 覆盖自举类文件的路径。
- extdirs + 目录: 覆盖扩展类的路径。
- d + 目录: 指定输出类文件的路径。
- encoding + 编码: 指定源文件中的字符集编码。
- target + 版本: 生成虚拟机的类文件。



配置正确的环境变量信息。



1.2.4 JDK常用命令

- 2. JDK的常用命令
- native2ascii: 将中文unicode码转换成ascii码的, -reverse参数可以将ascii码转换回来。
- javap: 将class反编译成Java bytecodes。
- jdb: Java的debug工具。
- jps: 查看JVM进程信息用的。
- keytool: 生成keystore文件。
- jar: 可将多个文件合并为单个JAR文件, jar是个多用途的压缩工具, 它基于ZIP和ZLIB压缩格式的。
- javadoc: Javadoc解析Java源文件中的声明和文档注释, 并产生相应的HTML页面, 描述公有类、保护类、内部类、接口、构造函数、方法。在实现时, javadoc要求且依赖于Java编译器完成其工作。



1.2.5 Java各个目录含义

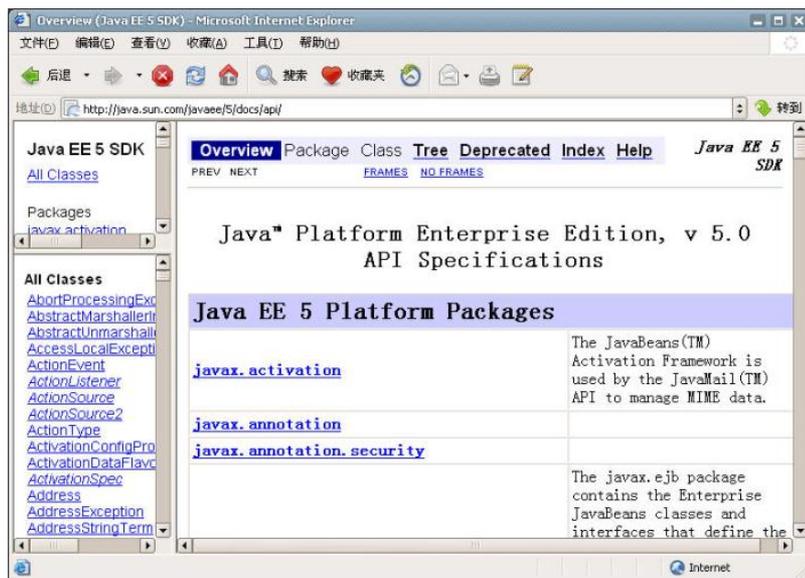
- JDK安装完成后，在安装目录下会安装很多目录和文件。这里再对这些目录进行简单的介绍。分类及说明如表所示。

各目录含义

<u>bin</u>	JDK的基本程序都在这里，如果javac、java、javadoc等。
<u>demo</u>	和字面意思一样，Java自带的一些例子程序。
<u>jre</u>	Java的运行时的环境都在这里。
<u>lib</u>	Java的类库。
<u>src</u>	Java类库的源代码

1.2.6 要善于使用JDK的帮助文件

- JDK的帮助文件有在线版本和离线版本两种，可以从Java的官方网站上下载 <http://java.sun.com>到最新的JDK <<http://java.sun.com>到最新的jdk/>帮助文件，帮助文件分为两种格式，有HTML格式和CHM格式。JDK的帮助文件使用很简单，只需要打开目录下的index.html即可。若想查找某个方法是怎么实现的，只需根据包的路径找到此方法就行了。Java EE5的帮助文件的路径和界面如图所示。



Java EE 5 的帮助文件的界面



1.3 程序开发过程

- 安装好JDK及配置好环境变量以后，就可以进行Java程序的开发。因为Java是一种编译性语言，所以在程序开发过程方面是和其他语言有所不同的。要开发Java程序，要经过以下3个步骤：
 - (1) 创建一个源文件。Java源文件就是Java代码文件，以Java语言编写。Java源文件是纯文本文件，扩展名为“.java”。可以使用任何文本编辑器来创建和编辑源文件，本书使用Windows系统自带的记事本做为Java源文件的编辑器。
 - (2) 将源文件编译为一个.class文件。使用JDK所带的编译器工具javac.exe，它会读取源文件并将其文本编译为Java虚拟机能理解的指令，保存在以后缀.class结尾的文件中。包含在CLASS文件中的指令就是众所周知的字节码（bytecodes），它是与平台无关的二进制文件，执行时由解释器java.exe解释成本地机器码，边解释边执行。
 - (3) 运行程序。使用java解释器（java.exe）来解释执行java应用程序的字节码文件（.class文件），通过使用Java虚拟机来运行Java应用程序。
- 对Java的程序开发过程有所了解后，在下一节中就来按照这个程序开发过程来开发一个最简单的HelloWorld程序。



1.4 编码规范

- 编写Java程序是要按照Java编码规范来进行编写的。一个程序不按照编码规范可能也是能够运行的，但是不按照编码规范编写的程序不是一个好程序，这种程序不易于程序的查看和维护。
- 编码规范包括很多内容，例如代码的编写规则，命名规则，代码注释等多项内容。命名规范和代码注释将在下一章中结合数据类型进行讲解。在本节中主要讲解一下代码的编写规则。
- 代码必须有缩进，缩进可以使用Tab键，或者四个空格。因为4个空格在eclipse中默认作为一个Tab缩进单位。
- 每行代码不要超过80个字符，这是由于很多编写工具不能对超过80个字符的内容进行很好的解释。
- 当代码在一行中放不下时，应进行换行。但是换行不能自动换行，而是按照级别来进行换行，并且同级别对齐。



1.5 HelloWorld: 第一个Java程序

- JDK安装完毕，环境变量也配置完毕后，下面开始编写第一个Java程序，以及讲解编译和运行程序的方法。



1.5.1 编写程序代码

- 打开文本文件编辑器，如Windows的记事本，也可使用更高级的编写工具。如Eclipse、JBuilder、NetBeans等，这些工具具有更加强大的功能，但现在不推荐使用，不利于初学者打下良好的基础。在记事本里添加如下代码。该代码可以直接复制到记事本中，当然如果自己输入是最好的。
- **【范例】**使用记事本编写的程序如下所示。

- 示例代码

- `//定义一个类名称为HelloWorld`
- `public class HelloWorld`
- `{`
- `//类的主入口函数`
- `public static void main(String args[])`
- `{`
- `//System.out.println为打印语句，用来显示结果`
- `System.out.println("欢迎使用Java来编写程序!");`
- `}`
- `}`



1.5.2 编译程序代码并运行

- 编写完Java程序的源代码就可以对该程序进行编译，编译Java程序的源代码的方法有如下几个步骤：

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\amer>d:

D:\>javac HelloWorld.java

D:\>_
```

编译 HelloWorld.java

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\amer>d:

D:\>javac HelloWorld.java

D:\>java HelloWorld
环境使用Java来编写程序!

D:\>_
```

程序 HelloWorld 的运行结果



1.5.3 注意事项

- 在编写、编译和运行Java程序时是有很多注意点的，这也是初学者需要注意的地方。
- 在运行时如果提示“`java.lang.NoClassDefFoundError`”的话，请查找环境变量是否设置正确。
- 在命令提示符下输入命令的时候要注意区分大小写，Java是区分大小写的。
- 用javac编译程序时是有扩展名的。
- 用java运行程序时是没有扩展名的。
- 源程序里要有main方法。
- 源程序里的类名要和文件名相同，包括大小写。



1.6 使用Eclipse集成开发工具开发

- Eclipse是目前最流行的Java开发工具，在Eclipse中集成了许多工具和插件，从而使Java的开发更容易。Eclipse是一个可以免费使用的软件，可以从Eclipse的官方网站<<http://www.eclipse.org/>>上下载。解压缩就可以使用，直接下载的Eclipse是英文版，可以下载中文语言包NLpack1-eclipse-SDK-3.2.1-win32.zip从而完成中文版Eclipse的安装。
- 下载和安装Eclipse后，就可以使用该集成工具了，双击eclipse.exe文件就可以运行Eclipse。由于篇幅原因，这里读者可以自己熟练一下Eclipse界面内容。这里主要来讲解如何在Eclipse中进行第一个HelloWorld程序开发。开发步骤如下所示。
- (1) 打开Eclipse，选择菜单栏“文件”，再选择级联菜单“新建”，最后选择子菜单“项目”，在弹出的对话框中选择Java项目，并单击按钮“下一步”。
- (2) 输入项目名称，例如：FirstProject；在“内容”选项卡中选择“从现有资源创建项目(X)”，然后在目录中找到前面HelloWorld.java的路径。单击“完成”按钮完成项目的创建。
- (3) 在“包资源管理器”中单击右键，弹出邮件菜单，选择“新建”->“包”菜单，在弹出对话框的“名称(N)”文本框中输入包名，这里输入FirstBao。
- (4) 打开Java编写界面，输入HelloWorld程序。单击运行按钮，就会在下面的控制台窗口中输出“HelloWorld”内容。这样一个Java程序就在Eclipse工具中编写、编译和运行完成。



1.7 综合练习

- 编写一个输出“我终于会Java了”的程序。
- **【提示】** 参考本章中编写的第一个Java程序来进行编写。
- ```
public class LianXi1
```
- ```
{
```
- ```
public static void main(String args[])
```
- ```
{
```
- ```
System.out.println("我终于会Java了");
```
- ```
}
```
- ```
}
```



## 1.8 小结

- 本章介绍了Java的各种平台，以及如何下载、安装JDK，设置环境变量等。这些是编写Java程序的基本，并通过一个小例子说明是如何编写Java程序的。在本章学习中，将重点学习如下几点。
- JDK的安装与配置。
- 如何编写、编译和运行Java程序。



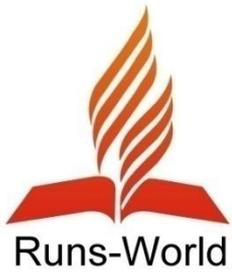
## 第2章 Java的基本数据类型

- 本章开始介绍Java的基本数据类型，如整型、浮点型等，以及它们之间的转换，最后介绍标识符的命名规则。通过本章的学习，读者应该能够完成如下目标。
- 了解Java有哪些基本数据类型。
- 掌握各种数据类型的基本含义。
- 学会如何进行数据类型转换。
- 了解标识符和保留字等基本概念。
- 了解如何在Java程序中进行注释。



## 2.1 数据类型

- 所谓数据类型，就是能真正表示数的类型，在Java里数据基本类型一共有8种，int表示整型，float表示浮点类型，下面将针对部分类型作详细地介绍。



## 2.1.1 整型

- 整型是Java数据类型中的最基本类型，使用int表示。所谓整型就好比日常生活中的十进制数，是没有小数点的。在Java里整型是有符号的，且有正负之分。如-10、20。
- Java里整型的数可以使用3种进制的数来表示，下面就对这三种进制来进行介绍。
- 1. 10进制:10进制数在日常生活中最常见，大家天天都在用。Java里定义一个10进制数如下。
  - //int为基本数据类型,是最常用的基本数据类型了
  - //正的10进制数
  - `int i = 11;`
  - //负的10进制数
  - `int j = -12;`
- 2. 8进制:8进制数的进制规则是满8进1，包含0-7的8个数字，在整数前面添加一个”0”就表示为8进制数。
- 3. 16进制数:16进制数的进制规则是满16进1，包含0-9，a-f的16个数字，在整数前面添加一个”0x”表示16进制数。



## 2.1.2 浮点型

- 浮点型同样也是Java数据类型中的基本类型，整型表示整数，浮点型则表示小数。所谓浮点类型就好比日常生活中的10进制数加上小数点。在Java里浮点类型是有符号且有正负之分的。
- 1. float: 单精度浮点数。:声明为float类型的浮点数时，要在结尾加F或f，浮点类型默认的类型是double。
- //正的浮点数
- `float i1 = 11.11F;`
- //负的浮点数
- `float j2 = -17.15f;`
- 2. double: 双精度浮点数:声明为double类型的浮点数时，要在结尾加D或d。声明为double类型时结尾的D和d可加可不加。这里建议是在double数据类型的数后面加上D或者d，以便更能够和单精度浮点数区分。



## 2.1.3 字符型(char)

- 字符型是一种表示字符的数据类型。char型表示一个字符，16位，占用2个字节。一般一个char型数值只用来表示一个字符的，用“ ’ ”单引号来表示。例如下面的例子。
- //表示一个字符
- char c1 = 'c';
- //表示一个unicode码
- char c2 = '\u005E';
- //表示一个整数
- char c3 = 56;
- Java中还有一种特殊的字符型数值，那就是转义字符。有一些特殊符号是不能通过一般字符来进行显示的，例如换行符和制表符。在表中列出了Java中比较常用的转义字符。

常用转义字符

| 转义字符            | 名称  | unicode码            |
|-----------------|-----|---------------------|
| <code>\n</code> | 换行  | <code>\u000a</code> |
| <code>\r</code> | 回车  | <code>\u000d</code> |
| <code>\b</code> | 退格  | <code>\u0008</code> |
| <code>\t</code> | 制表符 | <code>\u0009</code> |
| <code>\"</code> | 双引号 | <code>\u0022</code> |
| <code>'</code>  | 单引号 | <code>\u0027</code> |
| <code>\\</code> | 反斜杠 | <code>\u005C</code> |



## 2.1.4 布尔型 (boolean)

- 布尔型是一种起到判断作用的数据类型。boolean类型的取值非常简单，就好比日常生活中的真与假，在Java中用true与false，表示真与假。例如下面的例子。
- `boolean b1 = false;`
- `boolean b2 = true;`



## 2.2 数据类型间的转换

- 在日常生活中的斤和两，它们都是重量单位，一斤可以转换为十两。在Java中，整型、浮点型等都是基本的数据类型，它们是能够进行数据类型转换的。下面介绍数据类型之间的数据转换都有哪些转换方式。



## 2.2.1 自动转换

- 所谓自动转换就是不需要明确指出所要转换的类型是什么，是由Java虚拟机自动来转换的。转换的规则就是小数据类型变大数据类型，但大的数据类型的数据精度有的时候要被破坏。下面看一段代码。
- `//定义各种数据类型`
- `int i = 123;`
- `char c1 = 22;`
- `char c2 = 'c';`
- `byte b = 2;`
  
- `//自动转换的数据类型`
- `int n = b;`
- `long l = i;`



## 2.2.2 强制转换

- 所谓强制转换，是有一种强制性的，明明不能自动转换，而强制性地进行了转换。看下面的例子：
- `//定义数据类型`
- `int i = 22;`
- `long L= 33;`
- `//强制转换数据类型`
- `char c = (char) i;`
- `int n = (int) L;`
- 在其中*i*原来是一个int整型，但要将它强行转换成char字符型。同样*L*原来是一个long型，但要将它强行转换成int整型。通过前面的学习已经知道，long型的取值范围最大值可以为2的63次方减1，而int型的取值范围最大值只有2的31次方减1，所以如果*L*为一个大于2的31次方减1，在强制类型转换时就会丢失精度，使数值发生变化，这也是读者需要注意的地方。



## 2.2.3 隐含转换

- 所谓隐含转换和自动转换很相似，Java虚拟机根据数据类型的位数来判断此数据类型是否能装载此数据，如果能，Java就默认进行了转换。举例说明如下。
- //例子1
- `byte b = 111;`
- 
- //例子2
- `int i = 222;`
- `byte c = (byte) i;`
- 在这两条语句中有2个转换，一个是111转换成byte类型的数据库，因为byte类型的数据位数能装载下111，所以能进行转，这就是隐含转换。把222转换成byte类型的c就不能进行隐含转换，因为能进行隐含转换的只能是常量而不能是变量。



## 2.3 标识符的命名

- 在Java里方法名、类名、成员变量名都是标识符。所谓标识符，就好比日常生活中一个物品的名称一样，是一个代号，用来表示该物品。命名标识符的好处就是让外人看，一下就能了解这个标识符的用途。下面介绍怎样命名标识符。



## 2.3.1 标识符的命名规则

- 标识符要以英文字母开头，是由英文字母或数字组成的，其他的符号不能出现在标识符里。标识符具体说明如下所述。
- 英文字母是大写的A-Z，小写的是a-z，以及“\_”和“\$”。
- 数字包括0-9。
- 其他的符号是不能用在标识符里的。
- 不能用Java所保留的关键字。
- 在Java里标识符是大小写敏感的。
- 说明：符合标识符的命名规则并不是一种最好的命名方法。给一个标识符命名首先要符合命名规范，还要负责特点含义。



## 2.3.2 代码演示如何定义标识符

- 在本节中来演示什么是正确和错误的标识符。
- `int i = 22;`
- `int I = 33;`
- `char 2i = 23;`
- `float float = 3f;`
- 代码说明：
- 整型*i*和整型*I*在这里为两个不同的标识符，因为在Java里标识符是区分大小写的。
- `2i`标识符的第一个字母为数字，所以也不能为正确的标识符。
- `float`为Java保留的关键字，关键字不用在标识符里，而是另有用途的。



## 2.3.3 错误的标识符命名

- 一个良好的标识符是能体现此标识符所描述的方法、成员变量或类的含义的。下面看例子。
- **【范例】** 示例代码是一个错误标识符命名的程序。

示例代码

```
//定义了一个类 a
public class a
{
 //定义成员变量
 int i = 2;
 char c = 3;

 //定义方法 b
 public void b()
 {
 System.out.println("b");
 }

 //Java 程序的主入口方法
 public static void main(String args[])
 {
 System.out.println("main");
 }
}
```



## 2.3.4 正确的标识符命名

- 【范例】 示例代码是一个正确标识符命名的程序。

示例代码

```
//定义了一个类 bike,所描述的是一个自行车↵
public class bike ↵
{↵
 //定义成员变量 ↵
 String bike_color; //自行车的颜色↵
 String bike_size; //自行车的尺寸,即 26 或 28↵
 ↵
 //描述的是自行车的基本信息↵
 public void bike_info()↵
 {↵
 System.out.println("自行车的组成与出厂信息");↵
 }↵
 ↵
 //Java 程序的主入口方法↵
 public static void main(String args[])↵
 {↵
 System.out.println("main");↵
 }↵
}↵
```



## 2.4 关键字

- 所谓关键字，就好比日常生活中一个物品的标识，和人的名字很相似，具有特殊的含义。在Java里保留了很多关键字，这些关键字都有其各自的用途。因此标识符是不用这些关键字的。
- Java所保留的关键字在编码的时候是不能使用的，如果使用将提示编译错误。Java所保留的关键字如表所示。

表 Java所保留的关键字列表

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



## 2.5 代码注释

- 所谓注释，就好比在日常生活中听老师讲课所作的笔记，笔记的作用是解释知识点，帮助加强记忆。在Java中，在程序中通常给出一些解释，也可以提示某段代码的作用，这就是Java中的代码注释。注释的代码是不被编译的，所以不用担心执行效率的问题。



## 2.5.1 行注释

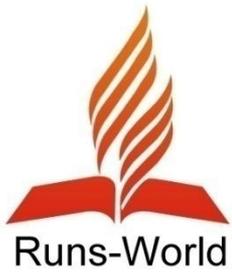
- 所谓行注释就是一整行的注释信息，单行注释也是最常用的，行注释的语法是“//”，在注释符号后面一整行都被作为注释信息。例如下面的小程序。
- **【范例】** 示例代码是一个进行单行注释的程序。
  - 示例代码

- `public class HelloWorld`
- `{`
- `//这是Java程序的入口方法`
- `public static void main(String args[])`
- `{`
- `System.out.println("环境使用Java来编写程序!");`
- `}`
- `}`



## 2.5.2 块注释

- 所谓块注释和行注释是一个意思，都是注释信息的意思，起到提示的作用。块注释的语法是“/\* \*/”，以“/\*”开始，以“\*/”结束，在这个区域内的文字都将作为注释信息。例如下面的小程序。
- /\*
- @param name
- @author amer
- \*/



## 2.5.3 文档注释用户自定义类型

- 所谓文档注释是描述类的，通过在类里定义的文档注释，可以帮助程序员了解此类具有哪些功能，以及此类的相关信息的一个注释。文档注释以“/\*\*”开头，以“\*/”结尾，把前面的例子加以修改。
- **【范例】**例如下面的小程序。

示例代码

```
/**
 作者 amer
 */

public class HelloWorld
{
 public String name; //成员变量

 /**
 Java 程序的主入口方法
 参数为 args
 */
 public static void main(String args[])
 {
 System.out.println("环境使用 Java 来编写程序!");
 }
}
```



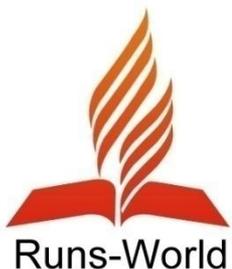
## 2.6 综合练习

- 1. 判断下面程序是否能够正常运行。
- ```
public class LianXi1
```
- ```
{
```
- ```
public static void main(String args[])
```
- ```
{
```
- ```
int For=1; //定义一个变量名称为For的变量
```
- ```
int Do=2; //定义一个变量名称为Do的变量
```
- ```
int t=For+Do;
```
- ```
System.out.println("变量和为"+t);
```
- ```
}
```
- ```
}
```



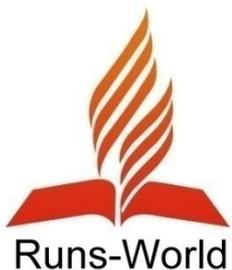
## 2.7 小结

- 通过对本章的学习可以让读者了解Java的基本数据类型的定义和它们之间的转换规则，以及它们之间的注意事项。掌握标识符的定义是学习本章的重点，也是写好程序的基本。如果还不完全了解标识符的问题，还可以参考电子工业出版社出版的《Java程序设计应用教程》一书来进行学习。在下一章中将继续讲解Java基本语法中的运算符。



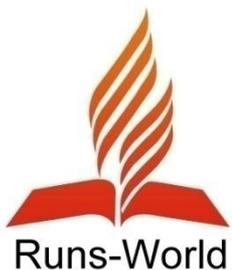
## 第3章 运算符

- 所谓运算符，就好比日常生活中的运算符号“+”、“-”、“\*”、“/”，这些符号几乎天天都要用到。在Java中，运算符就和日常生活中的运算符号一样，起到运算的作用，但是不再是这么简单的运算符。在本章中就介绍这些运算符，通过本章，读者应该完成下面的目标。
- 了解算术运算符的概念和熟练使用算术运算符。
- 了解关系运算符的概念和熟练使用关系运算符。
- 了解逻辑运算符的概念和熟练使用逻辑运算符。
- 了解三元运算符的概念和熟练使用三元运算符。
- 了解位运算符的概念和熟练使用位运算符。
- 了解位移运算符的概念和熟练使用位移运算符。
- 了解赋值运算符的概念和熟练使用赋值运算符。



## 3.1 算术运算符

- 算术运算符是大家最熟悉不过的了，比如“+”、“-”、“\*”、“/”。在本节中就先来介绍算术运算符如何应用和有哪些注意事项。
- +：加法运算符，也可做字符的连接用途。
- -：减法运算符。
- \*：乘法运算符。
- /：除法运算符。
- %：求余运算符。



## 3.1.1 “+”：加法运算符

- 加法运算符和日常生活中“+”是一样的，都是做两个数值的加法运算。下面举例在Java中的形式。
- //申明两个整数
- `int i = 33;`
- `int j = 44;`
- //将33和44做加法运算
- `int n = i + j;`

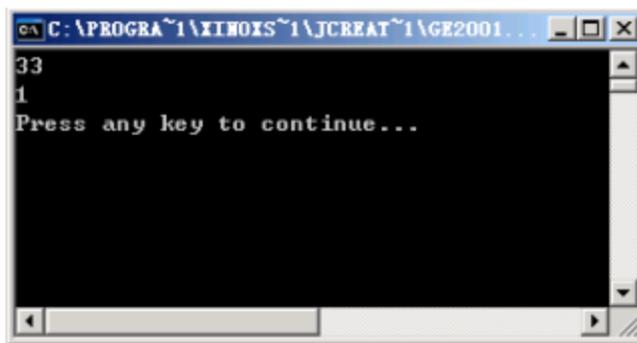
```
C:\PROGRA~1\XIN0IS~1\JCREAT~1\GE...
77
195
Press any key to continue...
```

加法运算符



## 3.1.2 “-”：减法运算符

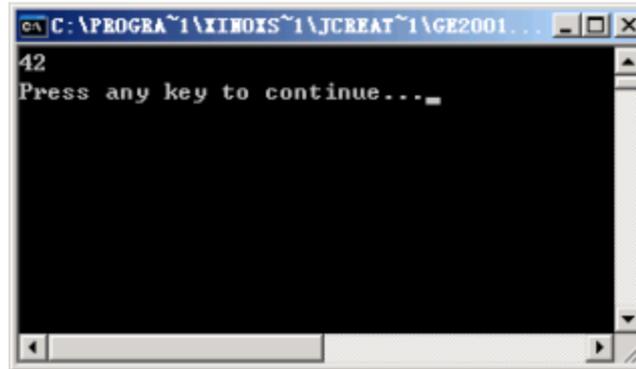
- 减法运算符和日常生活中“-”是一样的，都是做两个数值的减法运算。下面举例在Java中的形式。
- //申明两个整数
- `int i = 66;`
- `int j = 77;`
- //将66和77做减法运算
- `int n = i - j;`



减法运算符

### 3.1.3 “\*”：乘法运算符

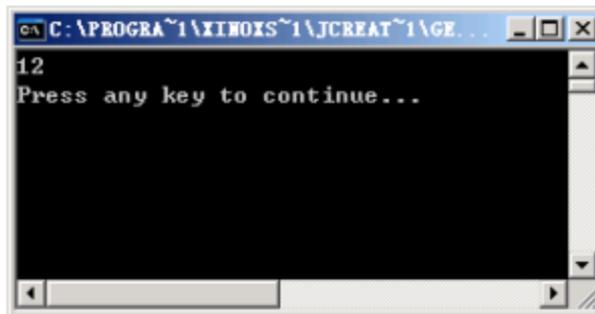
- 乘法运算符“\*”和日常生活中乘号类似，只是符号不一样而已，都是做两个数值的乘法运算。下面举例在Java中的形式。
- //申明两个整数
- `int i = 6;`
- `int j = 7;`
- //将6和7做乘法运算
- `int n = i * j;`



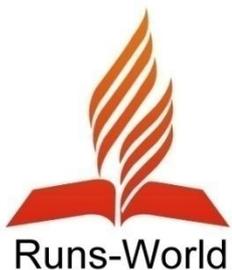
乘法运算符

## 3.1.4 “/”：除法运算符

- 除法运算符“/”和日常生活中的除号类似，只是符号不一样而已，都是做两个数值的除法运算。下面举例在Java中的形式。
- //申明两个整数
- `int i = 24;`
- `int j = 2;`
- //将24和2做乘法运算
- `int n = i / j;`

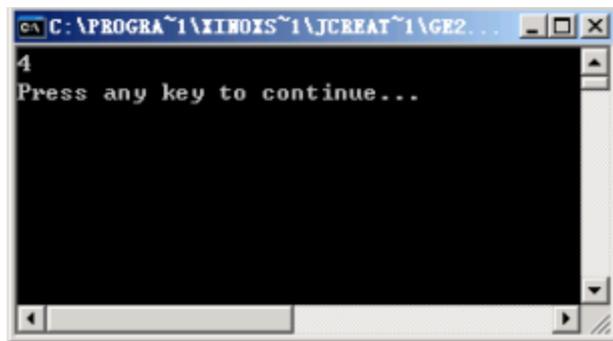


除法运算符



## 3.1.5 “%”：求余运算符

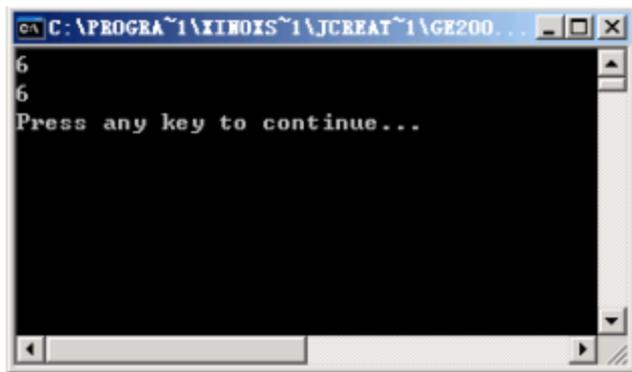
- 求余运算符“%”和日常生活中的除法求余类似。也是求两个数值的除法运算的余数。下面举例在Java中的形式。
- //申明两个整数
- `int i = 24;`
- `int j = 5;`
- //将24和5做求余运算
- `int n = i % j;`



求余运算符

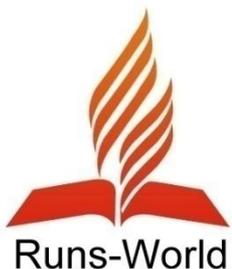
## 3.2 自增自减运算符

- 所谓自增减运算符，就是两个数做加减法运算将运算的结果赋值给做运算的变量。如下所示。
- `int i = 4;`
- `i++;`
- `int j = 4;`
- `j = j + 1;`



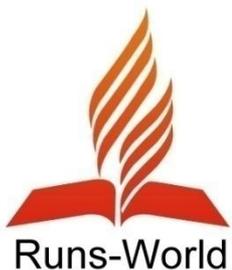
```
C:\PROGRA~1\XINDOIS~1\JCREAT~1\GE200...
6
6
Press any key to continue...
```

自增运算符



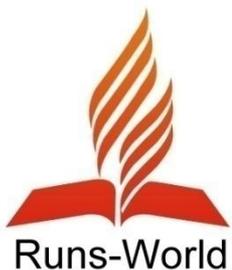
## 3.3 关系运算符

- 关系运算符描述的是一种关系，既然描述的是关系那结果就为对或不对。在Java里就表示为真或假。下面看关系运算符的分类。
- “==”：表示等于。
- “!=”：表示不等于。
- “>=”：表示大于等于。
- “<=”：表示小于等于。
- “>”：表示大于。
- “<”：表示小于。
- 关系运算符比较的是基本类型的话，就表示比较的是值是否相等。如果用“==”和“!=”比较的是对象的话就表示比较的是对象引用是否相等。



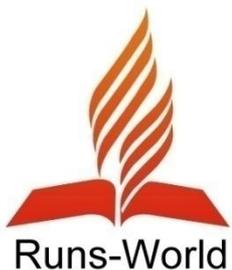
### 3.3.1 “==”、“!=”

- 等于和不等运算符比较的是运算数的等于和不等，结果为true和false。即真和假。例如下面的例子。
- //定义两个整型的变量
- `int i = 4;`
- `int j = 4;`
- 
- `boolean b1 = i == j;`
- `boolean b2 = i != j;`
- 
- //创建两个对象
- `String s1 = new String ();`
- `String s2 = new String ();`
- 
- `boolean b3 = b1 == b2;`
- `boolean b4 = s1 != s2;`



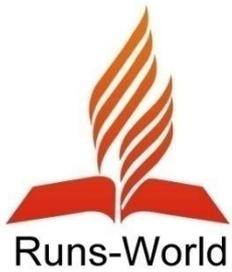
### 3.3.2 “>”、“<”、“>=”、“<=”

- 大于和小于运算符比较的是运算数的大于和小于，结果为ture和false。即真和假。例如下面是使用这些关系运算符的例子。
- //定义两个整型变量
- `int i = 5;`
- `int j = 4;`
  
- `boolean b1 = i > j;`
- `boolean b2 = i < j;`
- `boolean b3 = i >= j;`
- `boolean b4 = i <= j;`



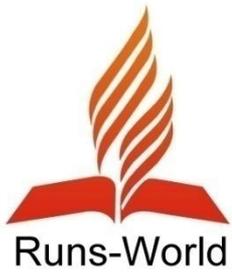
## 3.4 逻辑运算符

- 逻辑运算符，其实就是比较的二进制数的逻辑关系，运算结果为true、false。逻辑运算符包括如下。
- 与运算符：“&&”、“&”。
- 非运算符：“||”、“|”。



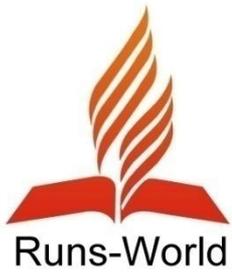
## 3.4.1 “&&” 与运算符

- “&&”运算符比较的是符号两边的表达式的真假。
- 【范例3-10】通过下面代码说明“&&”运算符。
- 示例代码3-10
- 01 //修改上节例子,如下形式
- 02 public class HelloWorld10
- 03 {
- 04 public static void main(String args[])
- 05 {
- 06 boolean n = (4 > 3) && (2 < 8);
- 07 System.out.println(n);
- 08 }
- 09 }



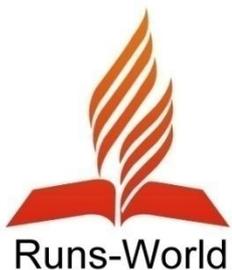
## 3.4.2 “||” 或运算符

- “||”运算符比较的是符号两边的表达式的真假，。
- 【范例3-11】通过下面代码说明“||”或运算符。
- 示例代码3-11
- 01 //修改上节例子,如下形式
- 02 public class HelloWorld11
- 03 {
- 04 public static void main(String args[])
- 05 {
- 06 boolean n = (4 > 3) || (2 > 8);
- 07 //打印并显示结果
- 08 System.out.println(n);
- 09 }
- 10 }



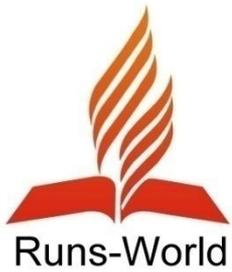
## 3.4.3 “!” 非运算符

- “!”非运算符是把符号右边的表达式的结果即true、false取反。如为true，取反为false；如为false，取反为true。
- 【范例3-12】通过下面代码说明“!”非运算符。
- 示例代码3-12
- 01 //修改上节例子,如下形式
- 02 public class HelloWorld12
- 03 {
- 04 public static void main(String args[])
- 05 {
- 06 boolean n = ! (2 > 8);
- 07 //打印并显示结果
- 08 System.out.println(n);
- 09 }
- 10 }



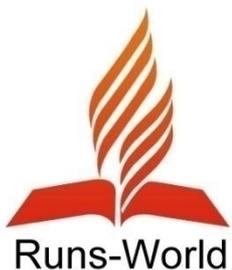
## 3.4.4 总结逻辑运算符

- “&&”运算符，符号的两边都为true时，结果为true。只要有一边不为true，结果即为false。
- “||”运算符，符号的两边只要有一边为true，结果就为true，如果都为false，结果即为false。



## 3.5 三元运算符

- 所谓三元运算符，是对三个表达式进行的集中比较，表达式1的结果为true时，就为第二个表达式，如果为false时，就为第三个表达式。语法是：
- 表达式1? 表达式2: 表达式3
- **【范例3-13】**通过下面代码说明三元运算符。
- 示例代码3-13
- 01 //修改上节例子, 如下形式
- 02 public class HelloWorld13
- 03 {
- 04 public static void main(String args[])
- 05 {
- 06 //4 < 3表达式的结果为true和false的其中一个。
- 07 boolean n = (4 < 3) ? true : false;
- 08 //打印并显示结果
- 09 System.out.println(n);
- 10 }
- 11 }



## 3.6 位运算符

- 所谓位运算符，就是将操作数转换成二进制，然后按位进行比较。
- 运算符包括：
  - “&”：按位与运算符。
  - “|”：按位或运算符。
  - “^”：按位异或运算符。

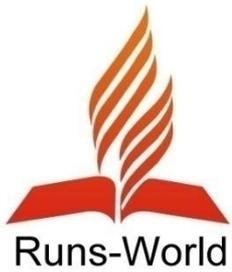
位运算符的比较规则

| $x$ | $y$ | $x \& y$ | $x   y$ | $x \wedge y$ |
|-----|-----|----------|---------|--------------|
| 0   | 0   | 0        | 0       | 0            |
| 0   | 1   | 0        | 1       | 1            |
| 1   | 0   | 0        | 1       | 1            |
| 1   | 1   | 1        | 1       | 0            |



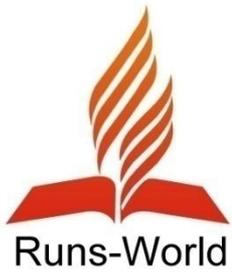
## 3.6.1 “&” 与运算符

- 按位与运算符，两个数同位都为1的时候即为1，有一边不是1的话就为0，即结果为false。
- **【范例3-14】**通过下面代码的演示来说明“&”与运算符。
- 示例代码3-14
- 01 //修改上节例子,如下形式
- 02 public class HelloWorld14
- 03 {
- 04 public static void main(String args[])
- 05 {
- 06 int n = 4 & 3;
- 07 //打印并显示结果
- 08 System.out.println(n);
- 09 }
- 10 }



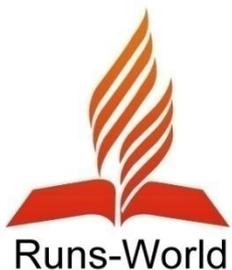
## 3.6.2 “!” 或运算符

- 按位或运算符，两个数同位有一个为1的时候即为1。
- 【范例】通过下面代码的演示来说明“!”或运算符。
- 示例代码
- 01 //修改上节例子,如下形式
- 02 public class HelloWorld15
- 03 {
- 04 public static void main(String args[])
- 05 {
- 06 int n = 4 | 3;
- 07 //打印并显示结果
- 08 System.out.println(n);
- 09 }
- 10 }



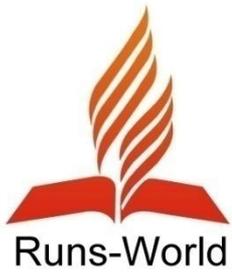
### 3.6.3 “^” 异或运算符

- 按位异或运算符，两个数同位都为1的时候即为0。有一个为1即为1。
- 【范例】通过下面代码的演示来说明“^”异或运算符。
- 示例代码
- 01 //修改上节例子, 如下形式
- 02 public class HelloWorld16
- 03 {
- 04 public static void main(String args[])
- 05 {
- 06 int n = 4 ^ 3;
- 07 //打印并显示结果
- 08 System.out.println(n);
- 09 }
- 10 }



## 3.7 位移运算符

- 所谓位移运算符，和逻辑运算符一样，都是对表达式进行比较的。位运算符是先把要比较的操作数转换成二进制数，然后向右向左移动相应的位数。位移运算符包括
- $\gg$ ：带符号右移。
- $\ll$ ：带符号左移。
- $\ggg$ ：无符号右移。



## 3.7.1 “>>” 右移运算符

- 右移运算符“>>”是把操作数转换成二进制数向右移动指定的位数。右移运算符是有符号的，如果为正数就补0，如果为负数就补1。
- **【范例】**通过下面代码的演示来说明“>>”右移运算符。
- 示例代码
- 01 //修改上节例子,如下形式
- 02 public class HelloWorld17
- 03 {
- 04 public static void main(String args[])
- 05 {
- 06 int n = 7 >> 2 ;
- 07 //打印并显示结果
- 08 System.out.println(n);
- 09 }
- 10 }



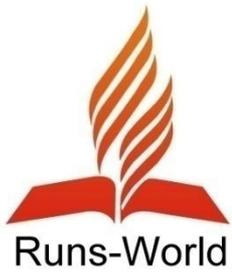
## 3.7.2 “<<” 左移运算符

- 左移运算符“<<”是把操作数转换成二进制数向左移动指定的位数。左移运算符是有符号的，如果为正数就补0，如果为负数就补1。
- **【范例】**通过下面代码的演示来说明“<<”左移运算符。
- 示例代码
- 01 //修改上节例子,如下形式
- 02 public class HelloWorld18
- 03 {
- 04 public static void main(String args[])
- 05 {
- 06 int n = 7 << 2 ;
- 07 //打印并显示结果
- 08 System.out.println(n);
- 09 }
- 10 }



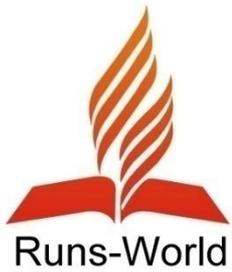
### 3.7.3 “>>>” 无符号右移运算符

- 无符号右移运算符“>>>”是把操作数转换成二进制数向右移动指定的位数。无符号右移运算符全在最高位上补0。
- **【范例】**通过下面代码的演示来说明“>>>”无符号右移运算符。
- 示例代码
- 01 //修改上节例子,如下形式
- 02 public class HelloWorld19
- 03 {
- 04 public static void main(String args[])
- 05 {
- 06 int n = 7 >>> 2 ;
- 07 //打印并显示结果
- 08 System.out.println(n);
- 09 }
- 10 }



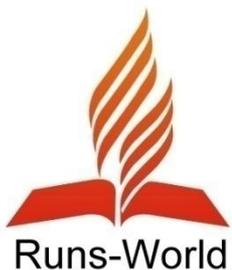
## 3.8 赋值运算符

- 所谓赋值运算符就好比在日常生活中的 $a=3$ , 即把3赋值给变量 $a$ 的意思是一样的, 以后就可以用 $a$ 表示3这个数值了。



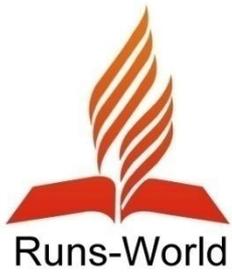
## 3.8.1 一般赋值运算符

- 一般运算符使用“=”，在编写代码里最常见，也是很容易理解的。如：
- `int n = 3;`
- 这一条代码的含义是把数值3赋值给整型的变量n。



## 3.8.2 运算赋值运算符

- 运算赋值运算符和一般赋值运算符很相似，也是赋值用的，但它具有运算的功能。
- **【范例】**通过下面代码的演示来说明运算赋值运算符。
- 示例代码
- 01 public class HelloWorld20
- 02 {
- 03 public static void main(String args[])
- 04 {
- 05 int n = 7;
- 06 int j = 0;
- 07 j += n;
- 08 //打印并显示结果
- 09 System.out.println(j);
- 10 }
- 11 }



## 3.9 运算符之间的优先级

- 运算符的运算优先级是有一定的顺序的。括号拥有最高的优先级，接下来是一元运算符，最后是二元运算符，如表所示。

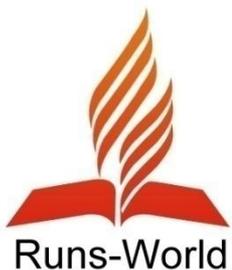
各个运算符的优先级

| 运算符从高到低                          | 含义       |
|----------------------------------|----------|
| . [] 0                           | 括号       |
| ++ -- ~ !                        | 自增和自减运算符 |
| * / %                            | 算术运算符    |
| + -                              | 算术运算符    |
| >> >>> <<                        | 位移运算符    |
| < <= > >=                        | 关系运算符    |
| = !=                             | 关系运算符    |
| &                                | 位与运算符    |
| ^                                | 位异或运算符   |
|                                  | 位或运算符    |
| &&                               | 逻辑与运算符   |
|                                  | 逻辑或运算符   |
| ? :                              | 三元选择运算符  |
| = += -= *= /= %= &= >>= >>>= <<= | 赋值运算符    |



## 3.10 综合练习

- 1. 区分前置自增减运算符和后置自增减运算符的不同。
- 【提示】通过程序来看这个问题。
- 01 public class LianXi1
- 02 {
- 03     public static void main(String args[])
- 04     {
- 05         int a=1;
- 06         int b=1;
- 07         System.out.println("使用后置运算符的结果为: "+(a++)); //显示后  
置结果
- 08         System.out.println("使用前置运算符的结果为: "+(++b)); //显示前  
置结果
- 09     }
- 10 }



## 3.10 综合练习

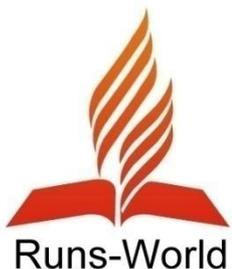
2. 三元运算符的应用有哪些?

【提示】同样还是通过程序来看这个问题。

```
01 public class LianXi2
02 {
03 public static void main(String args[])
04 {
05 int a=3;
06 int b=4;
07 System.out.println("使用条件运算符显示");
08 String s=(a<b)?"a小于b":"a大于b";
09 System.out.println(s);
10 System.out.println("使用if条件语句显示");
11 if(a<b)
12 {
13 System.out.println("a小于b");
14 }
15 else
16 {
17 System.out.println("a大于b");
18 }
19 }
20 }
```

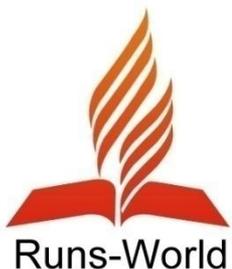
```
C:\PROGRA~1\XINNOIS~1\JCREAT~1\GE200...
使用条件运算符显示
a小于b
使用if条件语句显示
a小于b
Press any key to continue...
```

三元运算符



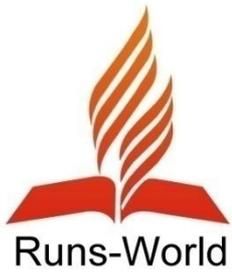
## 3.11 小结

- 通过对本章各种运算符的学习, 让读者对基本类型的运算有了新的认识, 虽然很简单, 但对以后学习有很大帮助。虽然运算符是比较简单的, 但其中也是有很多知识需要讲解的。在本章中最常用的运算符是自增自减运算符和赋值运算符, 这也是本章的重点。如果想了解更多的关于运算符的内容可以参考电子工业出版社 <<http://www.huachu.com.cn/itbook/publisher.asp?publisher=%B5%E7%D7%D3%B9%A4%D2%B5%B3%F6%B0%E6%C9%E7>>出版的《Java程序设计教程（第五版）（英文版）》 <<http://www.huachu.com.cn/itbook/itbookinfo.asp?lbbh=10062378>>一书来进一步学习。在下一章中将继续学习Java基本语法中的流程控制语句。



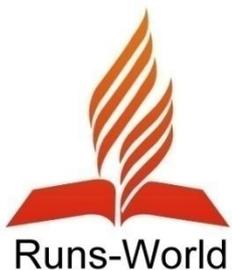
## 第4章 流程控制

- 在日常生活中，每个人早上起床后，通常要做洗脸、刷牙等事；如果有好看的电视节目，也会打开电视机进行收看；每一个人都有自己的安排。在Java中，洗脸、刷牙等事就好像代码程序，这些事是由流程控制语句来控制的。在流程控制语句中有一个叫做if的语句，它的作用就是根据条件来执行程序，就好像根据是否有好看的电视节目来决定一样。在Java里控制流程语句主要有条件语句、分支语句、循环语句。下面分别来介绍。通过本章的学习，读者应该能够完成下面的几点目标。
- 了解if条件语句和掌握各种if条件语句的使用。
- 了解switch分支语句和掌握switch分支语句的使用。
- 了解while循环语句和掌握while循环语句的使用。
- 了解do-while循环语句和掌握do-while循环语句的使用。
- 了解for循环语句和掌握for循环语句的使用。



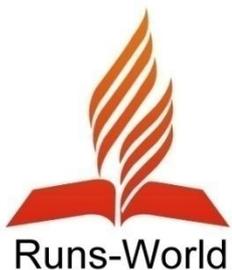
## 4.1 if 条件语句

- 在前面已经提到，如果有好看的电视节目时，就会打开电视进行收看。在Java中if条件语句就是实现这个功能，如果if条件中的条件语句是正确的，就会执行if语句中的程序语句。



## 4.1.1 if语句的语法

- if语句的基本语法为：
- `if(表达式) {方法体} else if(表达式) {方法体} else {方法体}`
- 下面用代码来演示。
- `if (a > 3)`
- 条件成功的方法体
- if语句的执行条件是，当表达式为true时，执行方法体的部分。
- 如果表达式不为false，执行else if的部分或else部分的方法体。



## 4.1.2 if语句用法举例

- if语句的用法有好几种，下面列举if语句的几种形式。
- 简写形式：if ...
- 一般形式：if ... else
- 完整形式：if ... else if ... Else
- 1. if语句的简写形式
- 2. if语句的一般形式
- 3. if语句的完整形式

```
C:\PROGRAMS\XINNOIS\JCREAT\AGE2...
变量n大于3
Press any key to continue...
```

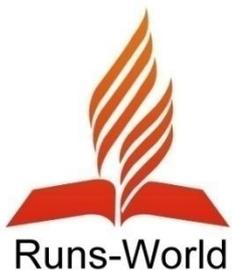
if 简写形式

```
C:\PROGRAMS\XINNOIS\JCREAT\1...
变量i小于5
Press any key to continue...
```

if 语句一般形式

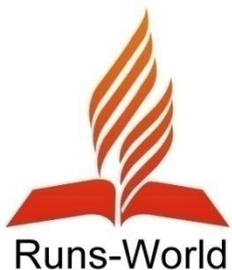
```
C:\PROGRAMS\XINNOIS\JCREAT\1\GE...
其他
Press any key to continue...
```

if 完整形式



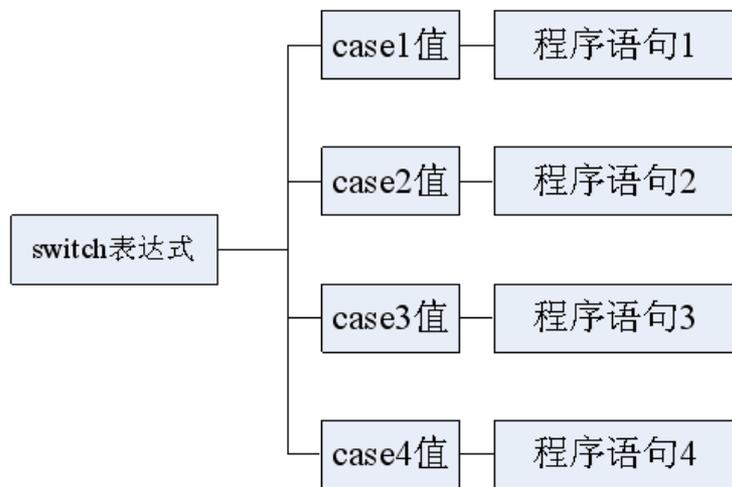
## 4.2 switch分支语句

- switch语句和if相似的，它是根据条件表达式的值来判断执行的程序语句。在日常生活中也经常有这样的情况，例如根据星期几来判断是否上课，如果等于星期一到星期五中的一天就上课，如果等于星期六或者星期日就不上课。switch分支语句要比if语句复杂的多。但当判断的条件很多时，switch分支语句要比if语句要方便很多。

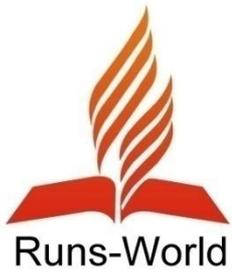


## 4.2.1 switch语句的语法

- switch分支语句和if语句一样都是通过表达式的成立与否，来选择执行哪条语句的。先来看一下switch语句的组成部分。
- switch (表达式)
- {
- case 表达式1:
- {
- 表达式的结果与表达式1相匹配时，所执行的方法体。
- break;
- }
- case 表达式2:
- {
- 表达式的结果与表达式2相匹配时，所执行的方法体。
- break;
- }
- case 表达式3:
- {
- 表达式的结果与表达式3相匹配时，所执行的方法体。
- break;
- }
- ...
- default:
- 表达式的结果与上述表达式的结果都不匹配时，所执行的方法体。
- }



switch 分支语句基本流程图



## 4.2.2 switch分支语句表达式的使用条件

- switch分支语句的表达式的使用有一定的条件，不是什么类型都能使用的。一般能使用的条件是具体的整型数值和一些有顺序的数列。下面先来对整型数值进行讲解。在Java中整数类型包括：byte、char、short、int型。

```
C:\PROGRA~1\XINNOIS~1\JCREAT~1\GE2...
所执行的分支是case 3分支
所执行的分支是case 4分支
Press any key to continue...
```

switch 分支语句

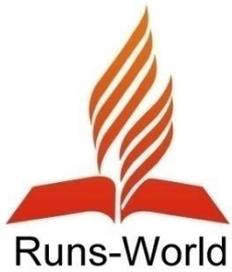


## 4.2.3 switch分支语句举例

- 在上一节中介绍了switch分支语句的各个组成部分的使用以及注意事项，下面用一个完整的例子来说明switch语句。

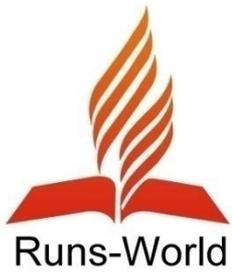
```
C:\PROGRA~1\XINHOIS~1\JCREAT~1\GE200...
所执行的分支是case 2 + 1分支
所执行的分支是case 3分支
所执行的分支是case two分支
Press any key to continue...
```

switch 分支语句



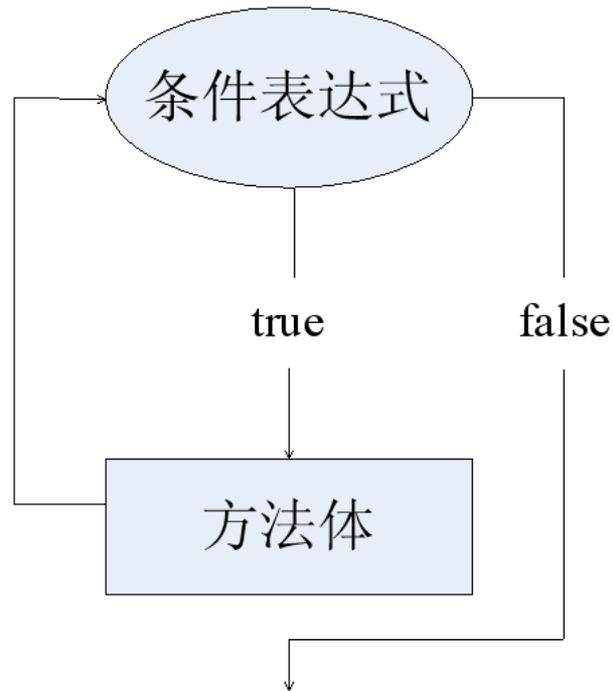
## 4.3 while 循环语句

- 所谓while循环语句，就是先进行判断再进行循环。通过判断表达式，来决定具体的循环次数。下面先介绍下while循环语句的语法并举例说明。

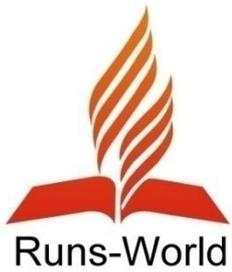


## 4.3.1 while语句的语法

- 通过判断表达式的成功与否，来决定循环的次数。先介绍基本语法：
- while(表达式)
- {
- 方法体
- }



while 语句流程图

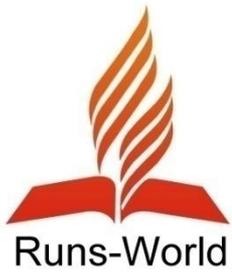


## 4.3.2 while循环语句举例

- 在上一节中介绍了while循环语句的具体语法后，下面用一个详细例子进行说明：【范例】在下面的程序中，来讲解如何显示乘法表。

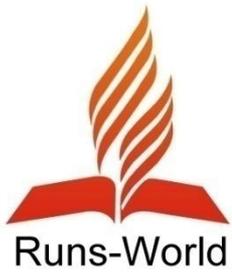
示例代码

```
//testWhile2 类,所描述的是一个 9*9 乘法表
public class testWhile2
{
 public static void main(String args[])
 {
 //定义整型变量
 int i = 9;
 int j = 9;
 //当变量 i 大于或等于 1 的时候执行循环
 while(i >= 1)
 {
 while((j <= i) && (j > 0))
 {
 System.out.print(i + " * " + j + " = " + j * j + " ");
 j--;
 }
 System.out.println(" ");
 i--;
 //把每次循环后的 j 值赋值给 i
 j = i;
 }
 }
}
```



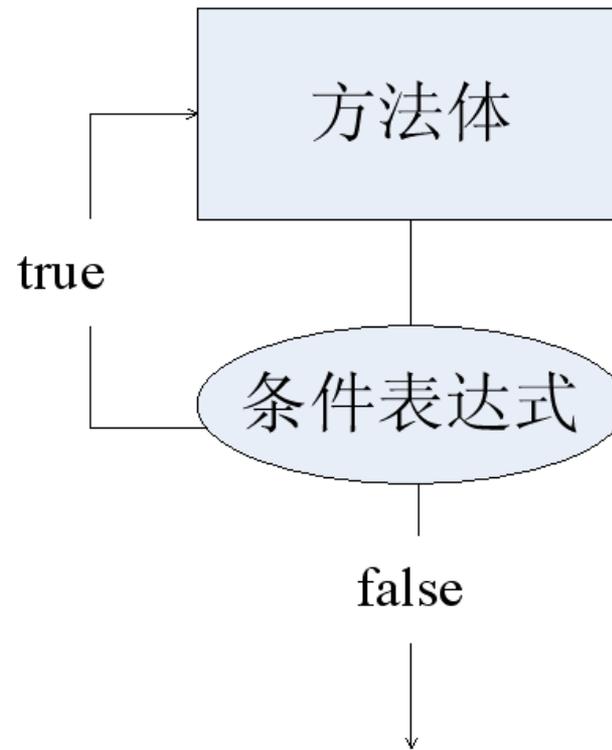
## 4.4 do...while循环语句

- 所谓do...while循环语句，就是先进行循环，再来进行表达式的判断，如果表达式不成立就退出循环。下面先介绍do...while循环语句的语法并举例说明。

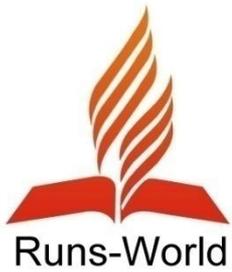


## 4.4.1 do...while语句的语法

- do...while循环语句是先进行循环，再进行判断。先介绍基本语法：
- do
- {
- 方法体
- }
- while(表达式);
- do ... while循环语句的流程图
- 如图所示。



do-while 语句流程图



## 4.4.2 do ... while循环语句举例

- 在上一节中介绍了do...while循环语句的具体语法后，下面用一个详细例子进行说明：
- **【范例】**下面是使用do-while循环语句完成乘法表功能的程序。

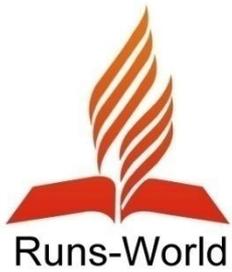
```
C:\PROGRAMS\XINNOIS\JCREAT\GE2001.exe
9*9=81 9*8=64 9*7=49 9*6=36 9*5=25 9*4=16 9*3=9 9*2=4 9*1=1
8*8=64 8*7=49 8*6=36 8*5=25 8*4=16 8*3=9 8*2=4 8*1=1
7*7=49 7*6=36 7*5=25 7*4=16 7*3=9 7*2=4 7*1=1
6*6=36 6*5=25 6*4=16 6*3=9 6*2=4 6*1=1
5*5=25 5*4=16 5*3=9 5*2=4 5*1=1
4*4=16 4*3=9 4*2=4 4*1=1
3*3=9 3*2=4 3*1=1
2*2=4 2*1=1
1*1=1
Press any key to continue...
```

do-while 循环语句



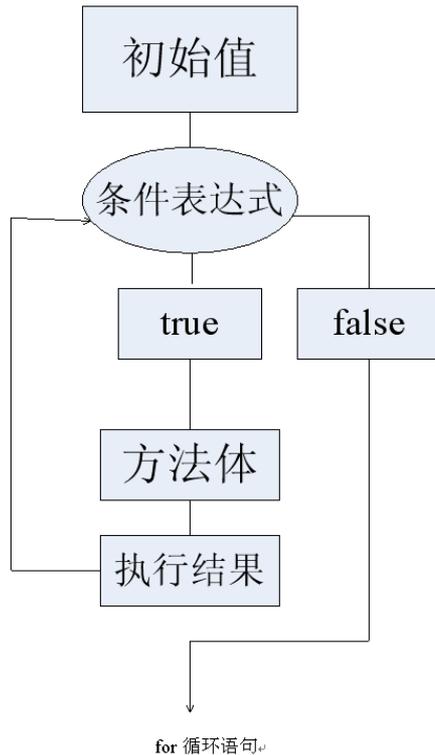
## 4.5 for循环语句

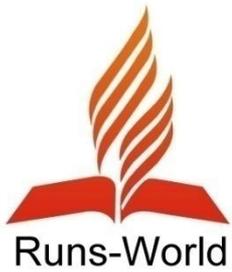
- 所谓for循环语句，就是明确了循环的次数，进行次数循环的。下面先介绍下for循环语句的语法并举例说明



## 4.5.1 for语句的语法

- 通过判断表达式的成立与否，来决定循环的次数。先介绍下基本语法：
- for(变量初始化, 表达式, 递增表达式)
- {
- 方法体
- }





## 4.5.2 用for循环来实现其他循环语句

- 【范例】修改上一节的9\*9乘法表代码。

示例代码

```
//testFor2 类,所描述的是一个 9*9 乘法表
public class testFor2
{
 public static void main(String args[])
 {
 for (int i = 1; i < 10; i++)
 {
 for (int j = 1; j < 10; j++)
 {
 if (j <= i)
 {
 System.out.print(i + "*" + j + "=" + (i * j) + " ");
 }
 }
 System.out.println(" ");
 }
 }
}
```

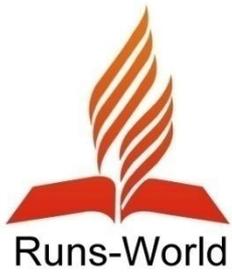


## 4.5.3 for循环语句的举例

- 下面介绍for循环语句的其他用法。并理解其含义。
- **【范例】**看下面生成正三角形的程序。

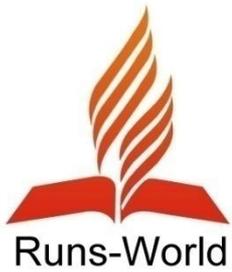
```
C:\PROGRA~1\XINNOIS~1\JCREAT~1\GE2001.exe
正三角:
 1
 212
 32123
 4321234
 543212345
 65432123456
7654321234567
876543212345678
98765432123456789
98765432123456789
Press any key to continue...
```

生成正三角



## 4.6 如何中断和继续语句的执行

- 在学校中，有时间会发生临时放假的情况，可能会临时放假一天，也可能一直放假。在Java循环语句中也有这种情况，可以使用break语句和continue语句来中断程序，就好比中断上课一样。不同的是break语句是一直放假，而continue语句是放假一天。语句的中断和继续就是指在语句的执行过程中，用代码中断语句的执行并退出此代码块。继续和中断类似，中断和继续在Java里用break和continue关键字来表示。



## 4.6.1 break : 中断语句执行

- break关键字在前面学习的switch分支语句中已经使用过了，下面直接用代码进行说明。
- **【范例】**下面是使用break来中断for循环的程序。

示例代码

```
public class testBreak
{
 public static void main(String args[])
 {
 for(int i = 0; i < 10; i++)
 {
 System.out.println(i);
 if (i == 5)
 {
 break;
 }
 }
 }
}
```

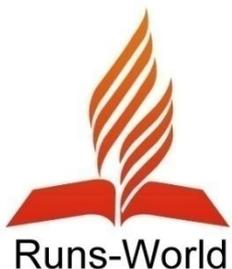


## 4.6.2 continue : 继续语句执行

- continue语句表示跳出本循环，继续执行下一次循环，同样还是采用程序来讲解continue语句的知识。
- 【范例】下面是使用continue语句的程序。

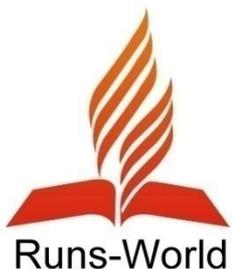
示例代码

```
public class testContinue
{
 public static void main(String args[])
 {
 for(int i = 0; i < 10; i++)
 {
 if (i == 5)
 {
 continue;
 }
 System.out.println(i);
 }
 }
}
```



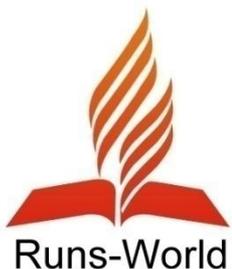
## 4.7 综合练习

- 1. 编写一个求从1到100数中所有的奇数和的程序。
- 【提示】使用for循环，将循环条件改为 $i+=2$ 。
- 01 public class LianXi1
- 02 {
- 03     public static void main(String args[])
- 04     {
- 05         int t=0;
- 06         //执行for循环，使循环间隔为2，从而之进行奇数操作
- 07         for(int i=1;i<=100;i+=2)
- 08         {
- 09             t+=i; //每次循环时，为表示总和的变量加上本次循环的变量值
- 10         }
- 11         System.out.println("从1到100的奇数为: "+t);
- 12     }
- 13 }



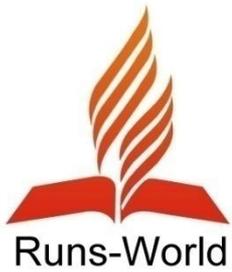
## 4.8 小结

- 通过本章的学习，可以让读者了解Java是如何控制程序的执行和中断，学好这些知识可以为以后编写代码打下了基础。其中对流程语句的讲解读者还可以参考电子工业出版社  
<<http://www.huachu.com.cn/itbook/publisher.asp?publisher=%B5%E7%D7%D3%B9%A4%D2%B5%B3>F6%B0%E6%C9%E7>> 《Java优化编程（第2版）》  
<<http://www.huachu.com.cn/itbook/itbookinfo.asp?lbbh=10059479>>一书来进行更详细的学习。本章的重点是for循环语句和if语句的使用。在下一章中将学习数组的创建和操作。



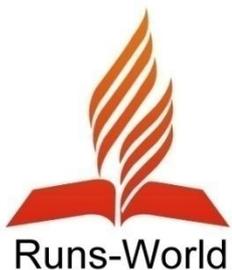
## 第5章 数组

- 在日常生活中，盒子的作用就是来放东西，但是是不可能把衣服和食品放在一起的，会有专门放衣服盒子，也会有专门放食品的盒子。在Java中，数组就好比日常生活中的盒子，用来存储数据。每一个数组也是有类型的，用来放相应类型的数据。数组是一种存储数据的数据结构。下面介绍数组的创建和使用，以及其注意事项。通过本章的学习，读者应该能够完成下面的目标。
- 知道如何创建数组，包括创建一维数组和多维数组。
- 能够对数组进行初始化操作。
- 熟练掌握在实际中，如何借助数组来解决问题。



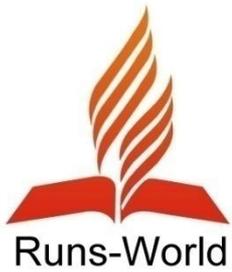
## 5.1 如何创建数组

- 假设现在有100个苹果，如果分散存放会很不好管理，如果集中存放的话，大家会想到放在一个盒子里。在Java里也是这样，100个苹果每个都放在一个变量里，会显示代码很凌乱，如果用数组存放会显示代码很整洁。所以说数组是存放数据的一种数据结构。下面讲解数组是如何创建的。
  -



## 5.1.1 创建数组

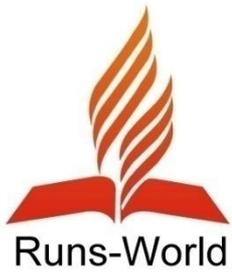
- 定义9 个int型的变量，分别存放1-9的数字。代码为：
- `int i1 = 1;`
- `int i2 = 2;`
- `int i3 = 3;`
- `int i4 = 4;`
- `int i5 = 5;`
- `int i6 = 6;`
- `int i7 = 7;`
- `int i8 = 8;`
- `int i9 = 9;`



## 5.1.2 创建多维数组

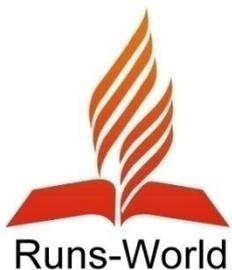
- 在一座楼中通常要有多个单元，一个单元中又有多个房间。在Java中多维数组就是这种设计，多维数组是一种嵌套的数组，一维数组的每个元素又是一个一维数组。多维数组的代码是这样的。
- **【范例】**下面是创建一个多维数组的程序。
  - 示例代码

```
• 01 public class ChuangJian2
• 02 {
• 03 public static void main(String args[])
• 04 {
• 05 //定义了一个多维数组
• 06 int i[][] = new int[4][4];
• 07 }
• 08 }
```



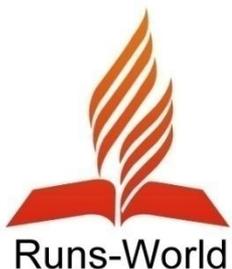
## 5.2 数组的初始化

- 初始化是给数组中的元素进行赋值，数组的赋值有创建赋值和动态赋值。这就好比确定一座楼有多少个单元，一个单元中又有多少个房间。下面介绍数组的赋值方法，并举例说明。



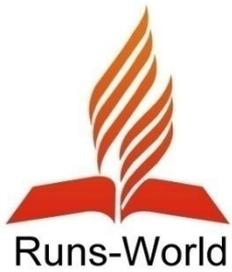
## 5.2.1 创建并初始数组元素

- 数组的创建初始化是数组创建完后系统对各个元素进行的默认赋值，系统对各个基本类型的默认初值如下：
  - `boolean` : `false`
  - `byte` : `0`
  - `char` : `' \u0000'`
  - `short` : `0`
  - `int` : `0`
  - `long` : `0L`
  - `float` : `0.0f`
  - `double` : `0.0`



## 5.2.2 循环初始化

- 数组除了可以在创建的同时进行初始化，也能在运行期间对数组各个元素进行赋值。对数组元素进行赋值通常使用for循环语句来进行。
- **【范例】**下面是一个使用for循环为数组进行赋值的程序。
- 示例代码
- 01 //testArray类, 所描述的是用for语句进行数组初始化
- 02 public class ChuShiHua3
- 03 {
- 04 public static void main(String args[])
- 05 {
- 06 //下面创建一个int型的数组, 数组的长度为10.
- 07 int a[] = new int[10];
- 08 for(int i = 0; i < a.length; i++)
- 09 {
- 10 a[i] = i + 1;
- 11 System.out.println("数组的各个元素的值为 : " + a[i]);
- 12 }
- 13 }
- 14 }

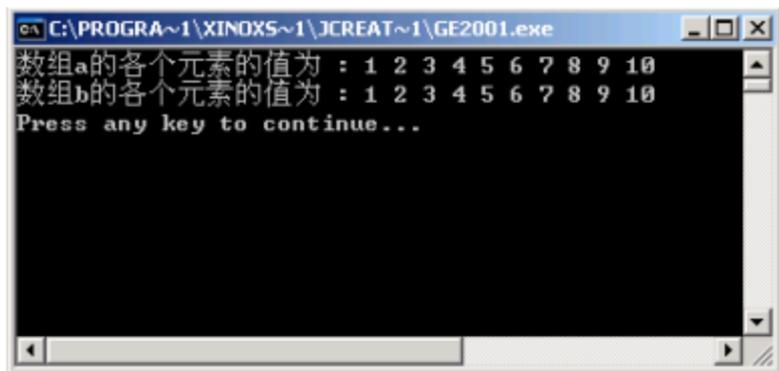


## 5.3 数组操作的举例

- 在前面介绍了创建一维数组，以及多维数组的方法，并演示了数组的初始化等操作。下面介绍操作数组的常用方法。

## 5.3.1 数组元素值的复制

- 数组里各个元素的值可以用数组的引用和使用循环对其值的赋值，但要注意两个不同长度的数组进行复制的时候下标越界的问题。下面通过代码来演示。
- 用for循环演示对数组各个元素的值的复制。

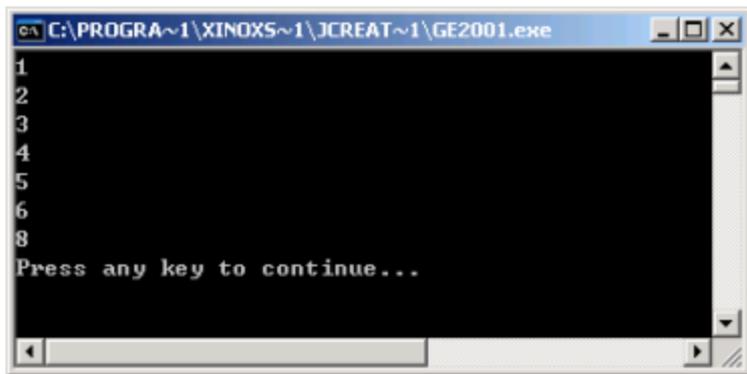


```
C:\PROGRA~1\XINOX5~1\JCREAT~1\GE2001.exe
数组a的各个元素的值为 : 1 2 3 4 5 6 7 8 9 10
数组b的各个元素的值为 : 1 2 3 4 5 6 7 8 9 10
Press any key to continue...
```

复制↵

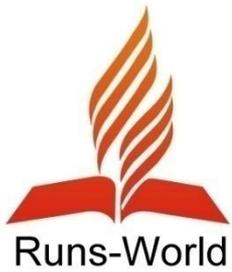
## 5.3.2 数组元素的排序

- 数组元素的排序是在数组的操作中是很常见的。下面分别介绍两种数组元素的排序方法。
- **【范例】**下面是使用冒泡排序法对数组中元素进行排序的程序。



```
C:\PROGRA~1\XINXS~1\JCREAT~1\GE2001.exe
1
2
3
4
5
6
8
Press any key to continue...
```

冒泡排序法

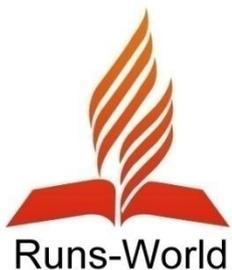


## 5.3.3 在数组里查找指定元素

- 在数组里查找指定的元素，下面来演示，用代码是怎么来实现的。
- **【范例】** 利用for循环语句来查找。



使用 for 循环查找

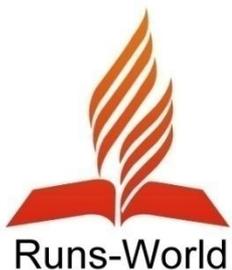


## 5.3.4 利用数组打印26个英文字母

- 在实际开发中，有时会让开发人员编写一个生成连续字母的字符串，该功能利用26个英文字母的ASCII码结合for循环来完成。
- 【范例】下面是一个显示生成连续字母的程序。

```
C:\PROGRA~1\XINOX5~1\JCREAT~1\GE2001...
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Press any key to continue...
```

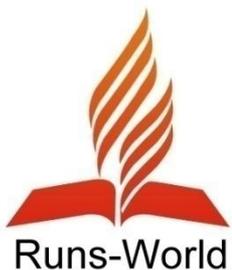
显示连续字母



## 5.4 综合练习

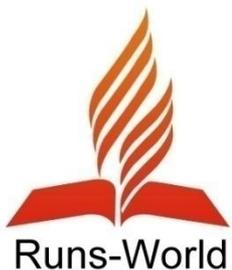
- 1. 利用数组打印连续小写字母。
- 【提示】可以使用示例代码5-14的形式作为参考来进行该程序的开发。

```
• 01 public class LianXi1
• 02 {
• 03 public static void main(String[] args)
• 04 {
• 05 //定义了一个char型的数组a
• 06 char[] a;
• 07 //实例对象数组，长度为25
• 08 a = new char[26];
• 09 for (int i = 0; i < 26; i++)
• 10 {
• 11 //通过把A的ascii码和循环变量进行相加来转换各个字码的ascii码
• 12 a[i] = (char)('a' + i);
• 13 System.out.print(a[i]);
• 14 if(a[i]=='z')
• 15 {
• 16 System.out.println("结束");
• 17 }
• 18 }
• 19 }
• 20 }
```



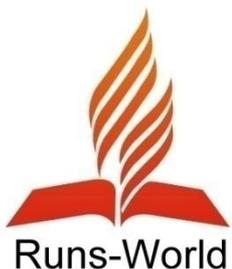
## 5.4 综合练习

- 2. 对已有数组元素按照一定规则进行排序。
- **【提示】**可以使用sort方法将数组中的所有数组元素按照从小到大的顺序进行排列。
- 01 //引入包Arrays, 因为Arrays.sort需要此包
- 02 import java.util.Arrays;
- 03 public class LianXi2
- 04 {
- 05 public static void main(String[] args)
- 06 {
- 07 //定义一个数组a, 并确定其元素
- 08 int[] a = {22, 33, 11, 56, 5};
- 09 System.out.println("数组排序前:");
- 10 paiXu(a);
- 11 //用sort方法对数组进行排序
- 12 Arrays.sort(a);
- 13 System.out.println("数组排序后:");
- 14 //调用方法进行数组排序的显示
- 15 paiXu(a);
- 16 }
- 17 private static void paiXu(int[] a)
- 18 {
- 19 for (int i = 0; i < a.length; i++)
- 20 {
- 21 System.out.println("a[" + i + "]=" + a[i] + " ");
- 22 }
- 23 }
- 24 }



## 5.5 小结

- 在本章通过对数组的创建以及数组的初始化的讲解，让读者对数组有了基本的了解。并演示了数组的常用方法等操作，且对前面的知识进行了巩固。本章的重点是对数组的创建和初始化。如果想了解更多的关于本章的内容，可以参考电子工业出版社出版的《JAVA面向对象编程》一书进行学习。



## 第6章 类与对象

- 在日常生活中，在盖房子之前要首先设计一个建筑图纸，然后根据图纸来盖房子。所谓类，好比在日常生活中描述一个物品的信息，如房子的建筑图纸。而对象就好比实实在在的房子。本章将要介绍类的定义、类的成员变量的定义和方法的定义、方法的参数等知识。通过本章的学习,读者应该能够完成如下几个目标。
- 了解什么是面向对象。
- 熟悉Java中的类并能够进行类的操作。
- 掌握成员变量和局部变量的区别。
- 掌握Java程序中的方法的创建和使用。



## 6.1 什么是面向对象

- 所谓面向对象，是指编写程序的时候要围绕着一个对象的功能进行编写的。本节将要介绍面向对象的特点以及与面向过程编程的区别。

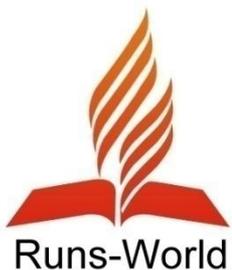


## 6.1.1 面向对象编程的特点

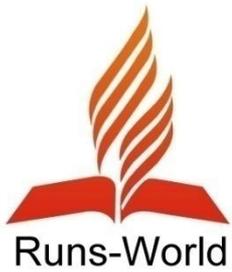
- 面向对象编程的缩写是OOP，全称为Object Oriented Programming。在进行面向对象的编程时，方法和成员变量都写在具体的对象里，并对其成员变量和方法有很好的隐藏性。对象之间的访问都是通过其接口进行的。下面列举面向对象编程的特点，分为如下几种。
- 首先要说的是继承。所谓继承，是发生在类与类之间的，是子类共享父类成员变量和方法的一种模式。通过扩展子类的方法可以使子类有比父类更加强大的功能。
- 说明：继承是面向对象编程的特点，同样也是Java的特点，这里和其他语言有很大不同。

- 示例代码

- 01 //bike类描述的是一个自行车
- 02 class bike
- 03 {
- 04 }
- 05
- 06 // racing\_cycle类描述的是一个公路赛车, 继承自bike
- 07 class racing\_cycle extends bike
- 08 {
- 09 }

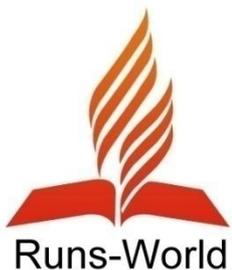


- 提示：继承是发生在类与类之间的。继承可以是单继承，也可以多层继承。
- 多态是指对象在运行期和编译期具有两种状态，多态的使用使代码具有了更多的灵活性和重用性。
- 抽象是指在定义类的时候，确定了该类的一些行为和动作。比如自行车可以移动，但怎么移动不进行说明。这种提前定义一些动作和行为的类为抽象的。
- 封装是指对一件物品的描述信息是这个物品所特有的，是不能让外界看到的一些成员变量和方法。在Java里成员变量和方法就被封装在类里，需要通过一些特有的方法访问它们。



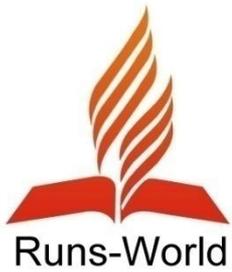
## 6.1.2 面向对象编程与面向过程编程的区别

- 面向过程是指在遇到问题的时候，怎么去解决这个问题，而分析问题的步骤，就是解决这个问题的方法，是通过方法一步一步来完成的。面向对象是指在遇到问题的时候，把问题分解成各自独立功能的类，而这个类是完成各自问题的。总结如下所述。
- 面向过程和面向对象最明显的区别就是，面向对象是按照要完成的功能来实现的，而面向过程是按照解决这个问题的步骤来实现的。
- 面向对象是按照程序中的功能进行划分的。
- 面向过程是按照问题的解决思路来划分的，是一步一步来解决问题的。
- 面向过程更看重的是完成问题的过程。
- 面向对象更看重的是功能，通过各种功能模块的组合来完成问题。



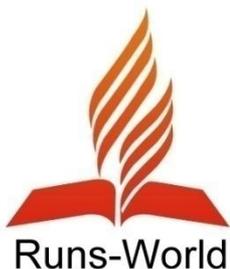
## 6.2 什么是类

- 所谓类是一种抽象的东西，描述的是一个物品的完整信息。比如房子和图纸的关系。在Java里，图纸就是类，定义了房子的各种信息，而房子是类的实体。



## 6.2.1 类的定义和对象的创建

- 定义一个类表示定义了一个功能模块。下面先介绍如何定义一个类，以及如何创建这个类的实例，即对象。类是通过关键字class来定义的，在class关键字后面加上类的名称，这样就创建了一个类。在类里面可以定义类的成员变量和方法。类的语法代码如下所示。
- class 类的名称
- {
- //类的成员变量
- //类的方法
- }
- 创建类的实例是通过new关键字来定义的，后面加上定义类时为类起的名称，需要注意的是在类名后还需要一个括号。创建类的实例的代码如下所示。
- new 类的名称();

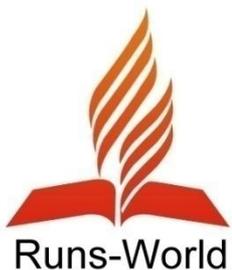


## 6.2.2 如何使用现有类

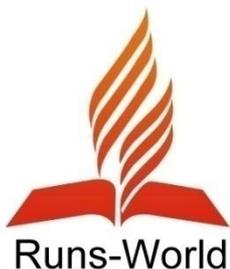
- 在定义一些类的时候，如何使用它们呢？这里需要分为多种情况。定义的类可以在一个包下面，也可以不在一个包下面，这在使用时是不同的。类又分为已有类和自定义类，它们之间的使用也是有区别的。下面就通过范例来讲解在不同情况下如何使用类。
- **【范例】**在同目录下使用类。首先是定义一个bike类，在该类中不存在任何成员变量和方法，这里只是演示如何在同一目录下使用类。

- 示例代码

- 01 //bike.java
- 02 class bike
- 03 {
- 04 }

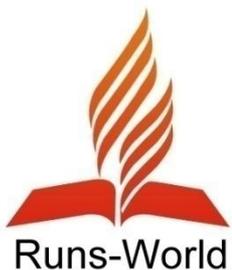


- 接下来定义一个使用**bike**类的类。
- 01 //testBike.java
- 02 //在testBike类里使用了**bike**类
- 03 class testBike
- 04 {
- 05   bike b = new bike();
- 06 }

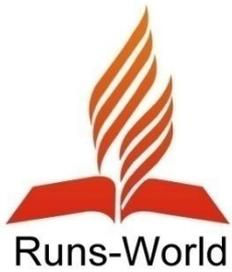


## 6.2.3 类设计的技巧

- 设计一个类要明确这个所要完成的功能，类里的成员变量和方法是描述类的功能的。如果定义了和这个类不相关的成员变量和方法将不是一个良好的设计。
- 【范例】示例代码是一个不太好的类设计。
  - 示例代码
- 01 public class bike
- 02 {
- 03 //这个成员变量描述的是自行车的颜色.
- 04 String color = "黄色";
- 05
- 06 //这个成员变量描述的是公路赛车的颜色, 所以在这里不太合适
- 07 String racing\_color = "绿色";
- 08 }
- 在本程序中定义了一个表示自行车颜色的color成员变量，又定义了一个表示赛车颜色的racing\_color成员变量；而该程序是定义的一个bike自行车类，所以定义表示赛车颜色的racing\_color成员变量是不太好的选择。

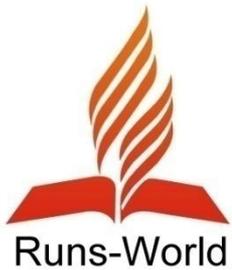


- 【范例6-8】 示例代码6-8是一个良好的类设计。
- 示例代码6-8
- 01 public class bike
- 02 {
- 03 //这个成员变量描述的是自行车的颜色
- 04 String color = "黄色";
- 05 }
- 01 public class racing
- 02 {
- 03 //这个成员变量描述的是公路赛车的颜色
- 04 String racing\_color = "绿色";
- 05 }
- 【代码解析】 在该范例中，定义了两个类。其中**bike**类中只定义了一个表示自行车颜色的**color**成员变量。同样在**racing**类中只定义了一个表示赛车颜色的**racing\_color**成员变量。这种设计相对上一个范例中的设计要好得多，这样使类和成员变量相对应，也使别人更容易读懂代码。



## 6.3 成员变量

- 所谓成员变量就是这个类里定义的一些私有的变量，这些变量是属于这个类的。就好比日常生活中的自行车的大小，即这个车子是26还是28的，这个尺寸就是自行车的成员变量，是描述这个自行车的。下面开始介绍成员变量。



## 6.3.1 成员变量的创建

- 成员变量描述的是这个类的一些属性或状态的，下面通过代码来演示怎么定义成员变量。语法为：变量的类型 变量的名称。
- 【范例】创建成员变量的一般形式。
  - 示例代码
- 01 //bike类描述的是一个自行车
- 02 public class bike
- 03 {
- 04 //这个成员变量描述的是自行车的颜色.
- 05 String color;
- 06
- 07 //这个成员变量描述的是自行车的大小, 即尺寸.
- 08 String size;
- 09 }
- 在该程序中，定义了一个叫做**bike**的类，在该类中定义了两个成员变量，一个是表示自行车颜色的**color**成员变量，一个是表示自行车型号的**size**成员变量。



下面看一个创建成员变量的完整形式。

- 代码讲解
- 通过**new**关键字来创建这个**bike**类的对象，用**bike**类的对象引用**b**来给其成员变量赋值。因为成员变量是在这个类实例化后才能访问到的。成员变量赋完值后，调用**println**语句来打印并显示结果。

## 6.3.2 成员变量的初始化

- 通过new关键字来创建一个对象后，会有一个系统默认的初始值。所以说不管有没有在创建成员变量的时候给变量一个值，系统都会有一个默认的值。
- 成员变量和对象的引用在声明的时候不对其赋初值，那么系统都会赋一个初值，具体的信息如表所示。

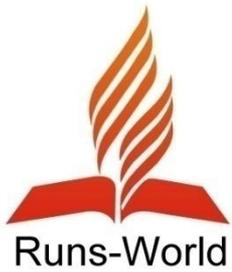
成员变量和对象引用的系统默认值

| 类型                   | 默认值                |
|----------------------|--------------------|
| <code>byte</code>    | 0                  |
| <code>char</code>    | '\u0000'           |
| <code>boolean</code> | <code>false</code> |
| <code>short</code>   | 0                  |
| <code>int</code>     | 0                  |
| <code>long</code>    | 0L                 |
| <code>float</code>   | 0F                 |
| <code>double</code>  | 0.0D               |
| 对象引用                 | <code>null</code>  |



## 6.4 局部变量

- 局部变量和成员变量很相似都是描述信息的。局部变量和成员变量的不同点就是局部变量是在方法体里创建的，在方法体外是访问不到这个变量的。

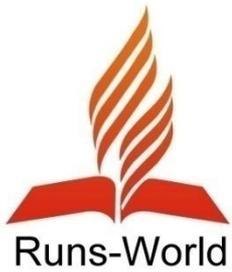


## 6.4.1 局部变量的创建和初始化

- 局部变量描述的是方法体的一些属性或状态的，下面通过代码来演示怎么定义局部变量。创建局部变量的基本语法为：变量的类型 变量的名称。
- **【范例】** 演示局部变量的例子。

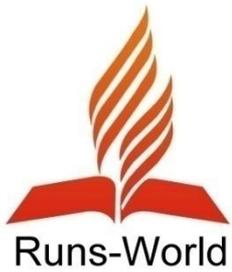
- 示例代码

```
• 01 //test类描述的是基本类型的初始化
• 02 public class test4
• 03 {
• 04 //程序的运行函数即主入口函数
• 05 public static void main(String args[])
• 06 {
• 07 //基本类型的局部变量
• 08 int size = 123;
• 09 boolean b = true;
• 10 //打印并显示局部变量
• 11 System.out.println(size);
• 12 System.out.println(b);
• 13 }
• 14 }
```



## 6.4.2 局部变量和成员变量的区别

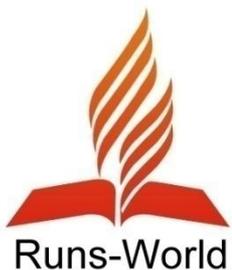
- 局部变量描述的是这个方法体内的属性的，而成员变量描述的是这个对象里的属性的，它们之间的区别，即访问区别如下：
- 成员变量可以被public、protected、default、private、static、final修饰符修饰。
- 局部变量可以被final修饰符修饰，但不能修饰为public、protected、default、private、static。
- 成员变量是在堆里进行创建的，而局部变量是在栈里进行创建的。
- 成员变量是系统默认值。
- 局部变量没有系统默认值，必须手动赋值。



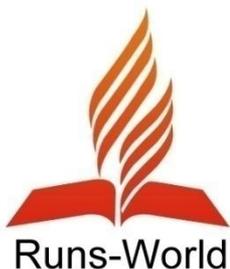
## 6.5 方法

- 每个人都有走、吃和睡等动作。在Java中，所谓方法就好比日常生活中的一个动作，是完成一系列操作的。在Java中也是如此，方法收到对象的信息，进行处理的操作。



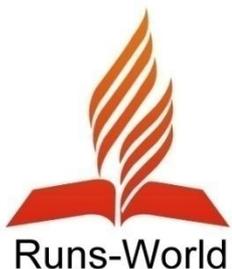


- 方法的返回类型有很多种，主要分为如下几类。
- 方法返回值为**void**类型时为无返回值。
- 方法返回值还可以为任意的类型，如**String**、**Boolean**、**int**。如果定义了方法的返回类型就必须在方法体内用**return**把返回值进行返回。
- 方法的返回值可以为**null**，但必须是对象类型。基本类型不能返回**null**。
- 在返回值为基本类型的时候，只要能够自动转换就可返回。
- 方法的参数也有多种形式，下面是对方法参数的讨论。
- 方法的参数可以为基本数据类型，也可以为对象引用类型。
- 每个参数都有完整的声明该变量的形式。
- 方法的参数可以有一个，也可有多个。
- **Java**程序的入口**main**就为一个方法，参数为**String[] args**，它是个特殊的方法。



## 6.5.2 方法参数的传递

- 参数的传递是传递的值还是引用呢。下面通过例子来分别说明，请仔细考虑。
- **【范例】**当传递类型为基本类型时，传递的是该类型的值。
- 01 //test类描述的是基本类型的传递
- 02 public class test
- 03 {
- 04 //方法add是把传入的参数进行+1，并显示其结果
- 05 public void add(int i)
- 06 {
- 07 i = i + 1;
- 08 System.out.println(i);
- 09 }
- 10

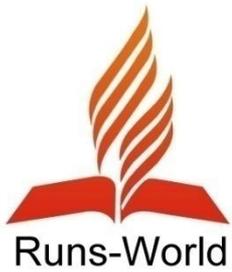


- 11 //程序的运行方法，即主入口方法
- 12 `public static void main(String args[])`
- 13 `{`
- 14 //基本类型的局部变量
- 15 `int size = 44;`
- 16
- 17 //创建bike类的对象实例，即bike类的对象引用b
- 18 `test t = new test();`
- 19
- 20 //打印原来的值
- 21 `System.out.println(size);`
- 22 //运行时的值
- 23 `t.add(size);`
- 24 //打印运行后的值
- 25 `System.out.println(size);`
- 26 `}`
- 27 `}`
- 在参数为基本类型进行传递的时候，是传递的这个值的备份，即第二份。不论在方法中怎么改变这个备份，都不是操作原来的数据，所以原来的值是不会改变的。

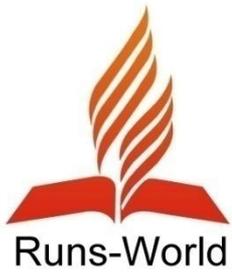


当传递的参数为对象引用类型时，也是利用的传值的方式进行的。

```
• 01 //test类描述的是方法的传递
• 02 public class test
• 03 {
• 04 public static void main(String[] args)
• 05 {
• 06 //创建一个对象类型
• 07 String s = new String("Hello ");
• 08
• 09 //打印其值
• 10 System.out.println("before : " + s);
• 11
• 12 //通过方法去改变其值
• 13 changeString(s);
• 14
• 15 //打印方法改变的值和原值
• 16 System.out.println("changeString : " + s);
• 17 System.out.println("after : " + s);
• 18 }
```

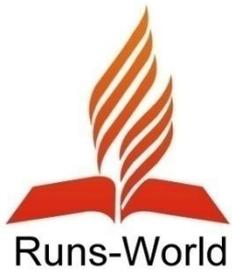


- 20 public static void changeString(String str)
- 21 {
- 22 str = new String("hi");
- 23 str = str + "china!";
- 24 }
- 25 }
- 当把对象引用s传递到一个方法后，这个方法可以改变这个对象的属性，并能返回相应的改变。但这个对象引用指向的这个字符串s是永远不会改变的。这里传递对象引用后，又通过这个引用去创建了一个新的String类型的字符串，这两个字符串在内存中当然不是同一个了。



## 6.6 对象引用的使用

- 所谓对象引用就是该引用名称指向内存中的一个对象，通过调用该引用即可完成对该对象的操作。本节将要讨论一些操作对象引用中将出现的一些常见问题。如不存在的对象、空引用、对象间的比较等问题，下面分别来说明。



## 6.6.1 调用不存在的对象或成员变量

- 如果调用的对象或成员变量没有创建，那么在编译的时候编译器将出现错误。下面用代码演示这个错误，并演示如何修正。
- **【范例】** 代码演示访问不存在的成员变量。

- 示例代码

- 01 //test类描述的是测试访问不存在的成员变量
- 02 public class test
- 03 {
- 04 //main方法为程序的入口函数
- 05 public static void main(String[] args)
- 06 {
- 07 //创建test类的对象实例
- 08 test t = new test();
- 09 //t.a访问的是一个不存在的成员变量，将提示不可识别的字段。
- 10 System.out.println(t.a);
- 11 }
- 12 }

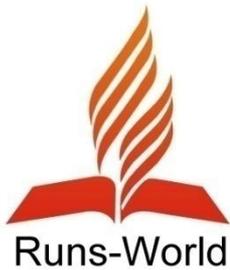


- 运行将会发生如下异常。
- Exception in thread "main" java.lang.Error: Unresolved compilation problem:
  - t.a cannot be resolved or is not a field
  - at test.main(test.java:7)
- **【代码解析】** 对象引用t要访问的是a这个成员变量，而a没有声明，在编译的时候将提示错误信息。在错误提示里，已经提示为main方法里的第7行，只需查看这里就能找到错误的所在。



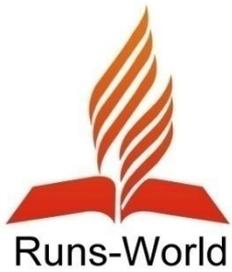
## 修改上述代码使程序运行通过。

- 01 //test类描述的是测试访问不存在的成员变量
- 02 public class test
- 03 {
- 04     //a为test类的成员变量
- 05     String a;
- 06
- 07     //main方法为程序的入口方法
- 08     public static void main(String[] args)
- 09     {
- 10         //创建test类的对象实例
- 11         test t = new test();
- 12         //t.a访问的是一个不存在的成员变量，将提示不可识别的字段
- 13         System.out.println(t.a);
- 14     }
- 15 }
- 根据上例中的错误提示在test类声明了一个名称为a的成员变量。因为String类型的a没有进行赋值，所以打印出来为null。

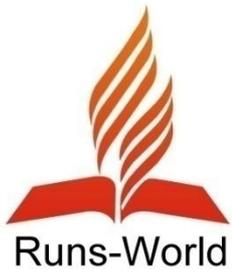


## 6.6.2 调用对象为null值的引用

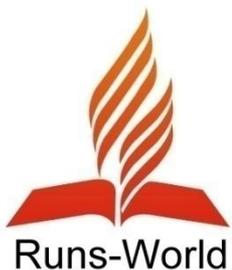
- 任何操作的对象的值为null的时候都将出现空指针错误，即“NullPointerException”错误，因为成员变量和方法是属于对象的，即属于用new关键字创建出来的对象的。下面用代码来演示这个错误，并演示如何进行修正。
- 01 //ArrayList类所需要的
- 02 import java.util.ArrayList;
- 03
- 04 //test类测试访问null值的对象
- 05 public class test
- 06 {
- 07 //声明一个成员变量a并进行初值
- 08 public String a = "test类的成员变量";



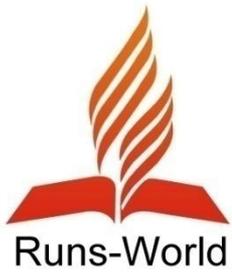
- 10 //Java程序的主入口方法
- 11 public static void main(String[] args)
- 12 {
- 13 //创建test类的对象实例
- 14 test t = new test();
- 15
- 16 //创建一个集合类，对象引用为一个null值
- 17 ArrayList al = null;
- 18
- 19 //向一个null的集合对象里添加数据
- 20 al.add(t.a);
- 21 }
- 22 }



- **ArrayList**类为一个集合类和数组很相似，都是用来存储数据用的。错误提示在**main**方法里的**20**行，提示为**NullPointerException**，即空指针错误。对象引用**al**声明为一个**null**值，表示这个对象并没有创建其对象的实例，只是一个引用而已。当操作任意一个为**null**的对象的时候都将提示空指针错误。



- 对本节的内容进行总结，可以概括成如下几点。
- 任何操作的对象的值为null，都将出现空指针错误，即“NullPointerException”。
- NullPointerException错误是运行期的错误，在编译的时候系统是不进行提示的。
- 在声明一个对象引用后尽量为其赋一个初值，来避免空指针的出现。

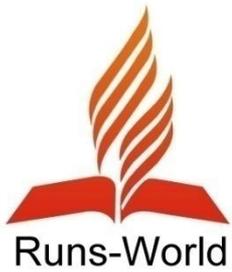


## 6.6.3 对象引用间的比较

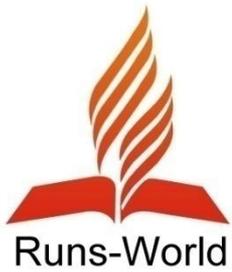
- 两个对象引用进行比较，比较的是这两个对象的引用，而引用是在内存中的一个地址。地址当然是不能相同的了。下面通过一个例子来演示引用间的比较。
- **【范例】** 演示两个对象引用的比较。

示例代码

```
01 //test 类测试访问 null 值的对象
02 public class test
03 {
04 //Java 程序的主入口函数
05 public static void main(String[] args)
06 {
07 //创建 test 类的对象实例
08 test t1 = new test();
09 test t2 = new test();
10
11 //这里的 equals 方法比较的是地址
12 if(t1.equals(t2))
13 {
14 System.out.println("t1 和 t2 相等");
15 }
16 else
17 {
18 System.out.println("t1 和 t2 不相等");
19 }
20 }
21 }
```

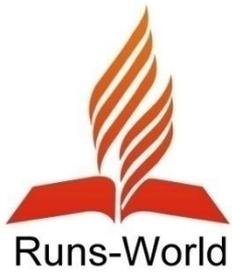


- `equals`方法在这里比较的是对象的引用，因为`equals`方法是`Object`类的方法，而任何类的父类都为`Object`，`equals`方法是继承过来的。继承将在后面的章节里做详细讲解。用`new`关键字创建的对象地址是重新分配的，它们进行比较，地址当然是不同的了。



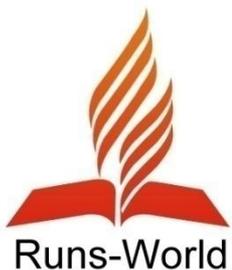
## 6.7 this

- **this**是Java保留的一个关键字，所谓**this**就好比日常生活中的“你我他”中的我，表示自己、本身的意思。在Java里也是如此，表示类的本身。



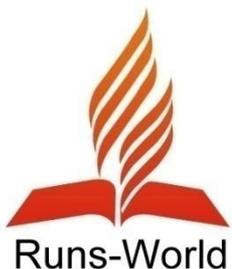
## 6.9 综合练习

- 1. 成员变量和局部变量相同情况下时如何访问？
- **【提示】** 成员变量和局部变量的访问方式是不同的。
- ```
public class LianXi1
```
- ```
{
```
- ```
    int i=5;    //定义一个成员变量i
```
- ```
 public static void main(String args[])
```
- ```
    {
```
- ```
 int i=6; //定义一个局部变量i
```
- ```
        System.out.println("局部变量的值为："+i);
```
- ```
 LianXi1 lx=new LianXi1();
```
- ```
        System.out.println("成员变量的值为："+lx.i);
```
- ```
 }
```
- ```
}
```



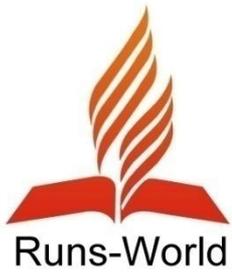
6.10 小结

- 通过学习本章，可以让读者了解面向对象的基本思想、类的创建和使用、成员变量和局部变量的区别，以及对象引用的一些注意事项等问题。学好本章可以为以后的学习打下基础。



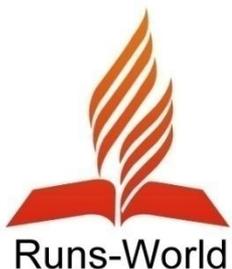
第7章 控制逻辑

- 在上一章中介绍了类和对象的概念，以及成员变量、局部变量和方法的概念。本章将介绍如何通过修饰符来控制变量的访问。首先介绍包的概念，后面将介绍各个控制权限的修饰符。通过本章的学习，读者应该能够完成如下几个目标。
- 了解包的概念和如何使用包。
- 知道类的访问级别有哪些，它们有什么区别。
- 重点掌握final修饰符和static修饰符。



7.1 包 (package)

- 所谓包，就好比日常生活中的箱子，是一个存放东西的空间。在Java中包的概念就好比Windows里的目录的概念，是一层一层的关系。按照功能的分类分别存放在各个包里。



7.1.1 创建一个包

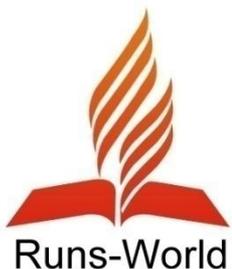
- 使用包是为了更好地将代码进行分别管理，首先介绍如何创建一个包，语法为：
- package 包名；
- 例如下面的程序语句。
- package a.b；
- 创建包时有几个注意事项。
- package为Java保留的关键字，不能使用别的符号进行代替。
- package语句必须在第一行。
- package语句只能有一个，不能有多个。
- 如果包有多层的话用句点”.”分隔



7.1.2 如何使用包

- 当创建了一个包时就要引入一个包，引入一个包的关键字为import，语法为：
- import 包名.*;
- import 包名.类名;
- **【范例】**下面通过代码来演示如何引入一个包。首先来看一个引入包的程序。
 - 示例代码

```
01 package a;
02
03 public class aaa
04 {
05     String emp = "包中的成员变量";
06
07     public void getMes()
08     {
09         System.out.println(emp);
10     }
11 }
```



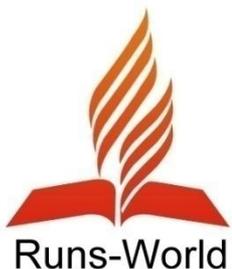
7.1.3 什么是静态引入

- 所谓静态引入就是引入包中的静态成员变量和静态方法。静态引入的关键字为static，静态的其他内容将将在后面的小结进行讲解。静态引入的语法为：

- `import static 包名.aaa.*;`
- `import static 包名.aaa.方法名称;`
- **【范例】**下面通过代码来演示如何静态引入。

- 示例代码

- `01 //静态引入System.out.println方法`
- `02 import static java.lang.System.out;`
- `03`
- `04 //test类测试包`
- `05 public class test`
- `06 {`
- `07 //Java程序的主入口函数`
- `08 public static void main(String[] args)`
- `09 {`
- `10 //打印并显示结果`
- `11 out.println("通过静态引入来打印数据");`
- `12 }`
- `13 }`



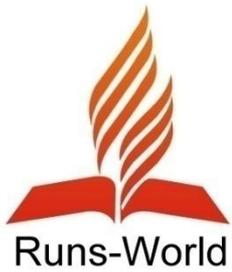
7.2 类的访问级别

- 类的访问级别，就好比日常生活中常见的大树，要想到达树顶，要从树底下慢慢地爬上去。是一层一层的进行访问的。树底下的树枝能看到旁边的树枝，但看不到树顶的树枝。而在Java中，类的访问也是有一种关系的。下面介绍类的访问级别和成员变量的访问级别。本节所提到的修饰符请读者先行了解，将在第八章对其含义做详细说明。



7.2.1 公开的访问级别

- 所谓公开的访问级别在Java中表示为public，即在类的名称前面带有public修饰符。用public修饰符修饰该类，表示该类在任何包中的任何类都能访问该类。但要注意不同包的问题。下面代码演示如何用public修饰符修饰一个类。
- //test类描述的是用修饰符修饰类
- public class test
- {
- ...//方法体
- }



7.2.2 默认访问级别

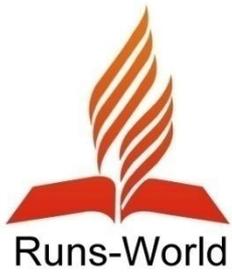
- 默认访问级别和公开的访问级别很相似，不同点就是默认访问级别不能访问不同包下的类。只能访问同包下的类。默认访问级别不需要在类前面加任何修饰符。下面通过代码来演示。下面代码演示如何定义一个默认访问级别的类。
- `//test`类描述的是默认访问级别的类
- `class test`
- `{`
- `...//方法体`
- `}`

7.3 什么是封装

- 所谓封装，就好比用一个盒子把一些东西装起来。在Java中就好比在一个类里定义了一些成员变量和方法，通过限制其成员变量和方法的可见性，使得外界不能访问它们。因此封装展现了接口，隐藏了细节。本节所提到的修饰符请读者先行了解，将在第八章对其含义进行详细的说明。
- **【范例】** 下面通过一个例子来演示如何进行封装。

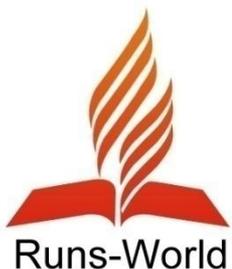


封装



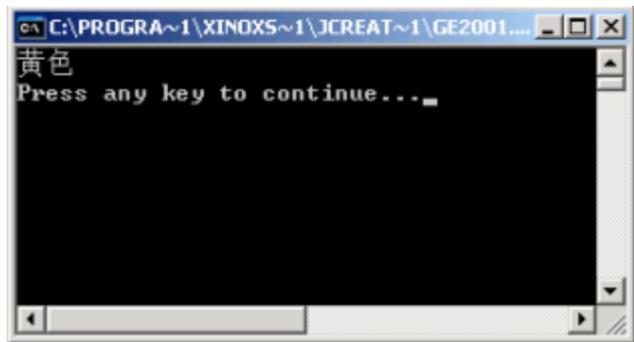
7.4 最终修饰符

- 所谓最终修饰符，在字面上可以说为最终的，不变的意思。修饰符final可以修饰很多类型的数据，其被修饰的数据所具有的含义也各有不同。下面将分别介绍修饰成员变量、局部变量、方法以及基本类型所具有的含义。



7.4.1 final修饰对象类型的成员变量

- final关键字修饰成员变量，其值是不能改变的。必须进行初始化。在一般情况下创建对象的时候，系统都对其成员变量进行默认初始化，被final关键字修饰的成员变量是不会被初始化的。
- **【范例】**下面用代码来演示final关键字修饰对象类型的成员变量没有初值的错误。



修改后运行结果



7.4.2 final修饰基本类型的成员变量

- 在本小节和上一小节里的对象类型的成员变量很相似。当final修饰基本类型的成员变量的时候，其值是不能改变的，也就是人们常说的常量。而对象类型的成员变量是指其引用不能改变。下面通过代码来介绍final修饰基本类型的成员变量有哪些特点。
- **【范例】**下面用代码来演示final关键字修饰基本类型的成员变量没有初值的错误。
 - 示例代码

```
01 //test类描述的是final修饰的成员变量
02 public class test
03 {
04     //把int变量申明为final类型
05     final int i;
06
07     //Java程序的主入口函数
08     public static void main(String[] args)
09     {
10         //创建test类的对象实例
11         test t = new test();
12
13         int n = t.i;
14
15         //打印并显示各个属性的值
16         System.out.println(n);
17     }
18 }
```



7.4.3 final修饰的局部变量

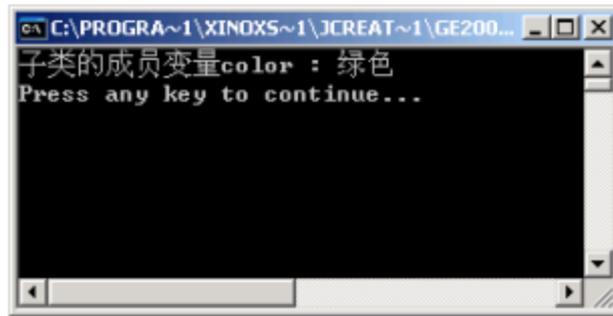
- final关键字修饰的局部变量和成员变量很相似，都是使其值不能被修改。但是被修饰的局部变量一旦被赋值后就不能进行修改了。如果在创建的时候没有对其赋值，那么在使用前还是可以对其赋值的。这就是成员变量和局部变量的不同了。下面用代码来演示。
- 【范例7-14】代码演示修饰局部变量可以不进行初始化赋值。

- 示例代码7-14

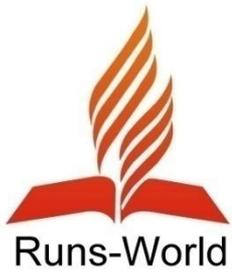
```
01 //test类描述的是final修饰的局部变量
02 public class test
03 {
04     //定义了一个方法
05     public void getMes()
06     {
07         System.out.println("程序顺利运行");
08     }
09
10     //Java程序的主入口函数
11     public static void main(String[] args)
12     {
13         //创建test类的对象实例
14         test t = new test();
15
16         //调用方法打印结果
17         t.getMes();
18     }
19 }
```

7.4.4 final修饰的方法

- 当用final关键字修饰方法时，和修饰成员变量、局部变量不太一样。被修饰的方法能被该类的子类所继承，但不能重写了。这样保护了父类某些特殊的数据。下面用代码来演示使用final关键字和不使用的区别。
- 【范例】不使用final关键字的代码例子。



不使用 final

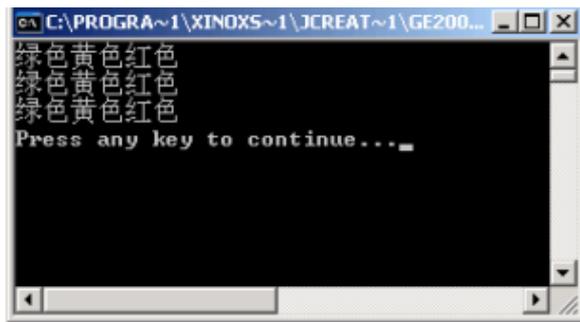


7.5 静态修饰符

- 静态修饰符static是Java保留的关键字，是静态的意思。所谓静态就是在内存中只能有一份。static能修饰变量、方法、语句块、内部类，下面分别对它们作介绍。

7.5.1 什么是静态变量

- 所谓静态变量就是只能存在一份，它属于类的，不随着对象的创建而建立副本。如果不想在创建对象的时候就需要知道一些相关信息，那么就声明为static类型的，被修饰为static类型的成员变量不属于对象，它是属于类的。下面通过代码来演示这一特性。用static关键字修饰成员变量的代码为
- static 成员变量类型 成员变量名称
- static String color = "绿色";



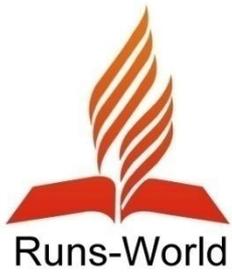
```
C:\PROGRA~1\XINOS~1\JCREAT~1\GE200...
绿色黄色红色
绿色黄色红色
绿色黄色红色
Press any key to continue...
```

静态变量



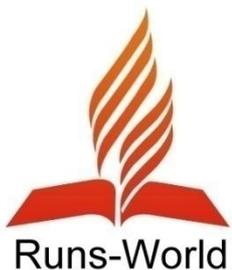
7.5.2 静态变量的访问

- 通过上一小节里的说明，让读者基本了解了什么是静态成员变量。下面介绍如何访问静态的成员变量。主要通过如下方式。
- 类名. 静态成员变量名称
- 静态成员变量名称
- **【范例】**下面通过代码来演示在静态的方法里使用非静态成员变量出现的错误。
 - 示例代码
- 01 //test类描述的是static修饰的成员变量
- 02 public class test
- 03 {
- 04 //申明一个static类型的String类型的变量color
- 05 String color = "绿色";
- 06
- 07 //Java程序的主入口函数
- 08 public static void main(String[] args)
- 09 {
- 10 //打印并显示
- 11 System.out.println(color);
- 12 }
- 13 }



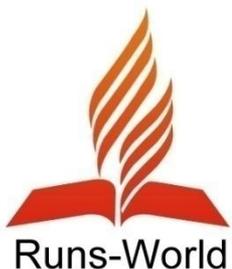
7.5.3 什么是静态常数

- 通过前面章节的学习让读者了解了static关键字的使用以及注意事项。下面介绍使用修饰符static的另一种形式——常量。所谓常量指的就是唯一的、不可变的、只存在一份的。在Java里用static final两种关键字来修饰成员变量。下面用代码来演示如何申明静态常量。
- //申明两个静态常量
- `public static final int i = 11;`
- `public static final float i = 11.0F;`
- `public static final double PI = 3.14;`
- static关键字修饰成员变量是属于类，随着类的创建而创建。
- final关键字修饰成员变量的值是不能改变的。
- static关键字和final关键字没有前后顺序之分。



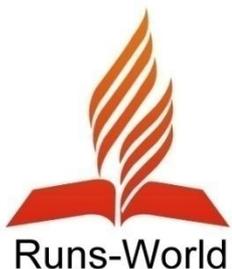
7.6 综合练习

- 访问静态变量有哪两种方法？
- **【提示】**访问静态变量可以直接访问，也可以使用类来访问。
- 01 public class LianXi1
- 02 {
- 03 static int i=5;
- 04 static int j=6;
- 05 public static void main(String args[])
- 06 {
- 07 System.out.println(i); //直接访问
- 08 LianXi1 lx=new LianXi1();
- 09 System.out.println(lx.j); //使用对象调用
- 10 }
- 11 }



7.7 小结

- 通过学习本章的包、类的访问级别以及final关键字、static关键字，读者了解了成员变量的访问是如何控制的。在后面的章节里将要学习继承这一重要概念，以及更多的关键字。如果想了解更多关于本章的知识可以参考电子工业出版社出版的《Java程序设计经典教程：融合上机操作实例》一书进行学习。



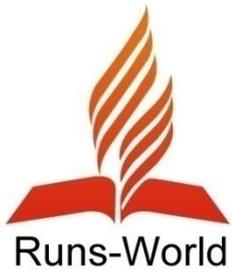
第8章 继承

- 经过了前面学习，对面向对象有了一定的认识，下面开始学习面向对象的一个重要的概念——继承。在此基础上讨论重写、重载、重写与重载之间的区别，以及多态和如何灵活运用final、abstract等概念，因此学好这些概念是灵活运用多态的基石。通过本章的学习，读者应该能够完成如下几个目标。
- 了解什么是继承和继承如何使用。
- 掌握声明成员变量的修饰符。
- 熟练掌握方法的重写和重载。
- 了解枚举、反射和泛型等热门技术。



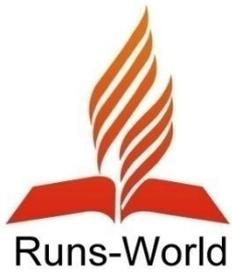
8.1 什么是继承

- 在日常生活中，经常遇到如下问题。有一辆自行车，自行车有颜色和型号大小之分，而公路赛车也有颜色和型号大小之分，公路赛车多了一项速度的优势。自行车有的东西公路赛车全都有，而公路赛车有的东西自行车不一定有，它们相同地方有很多。在Java中，对于这种情况下就采用继承来完成这个功能。【范例8-1】通过示例代码8-1来理解什么是继承。示例代码8-1
- 01 //这是一个类，表述的是一个自行车
- 02 public class bike
- 03 {
- 04 public String color; //自行车的颜色
- 05 public int size; //自行车的大小,即型号
- 06 }
- 07
- 08 //这是一个类，表述的是一个公路赛类
- 09 public class racing_cycle
- 10 {
- 11 public String color; //自行车的颜色
- 12 public int size; //自行车的大小,即型号
- 13 public String speed; //公路赛车的速度
- 14 }



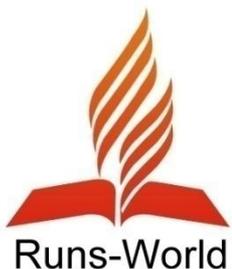
下面就来使用继承来简化上面的程序。

- 01 //这是一个类，表述的是一个自行车
- 02 public class bike
- 03 {
- 04 public String color; //自行车的颜色
- 05 public int size; //自行车的大小，即型号
- 06 }
- 07 //这是一个类，表述的是一个公路赛车，它继承于自行车
- 08 public class racing_cycle extends bike
- 09 {
- 10 public String speed; //公路赛车的速度
- 11 }
- 继承是为了让代码重复使用，提高效率，在此基础上衍生出更多的新类。继承是面向对象编程的特点，没有继承就不是面向对象编程，而是面向过程了。**Java**提供了单一继承，通过接口可以实现多重继承。本节要说明什么是继承，继承有那些特点。

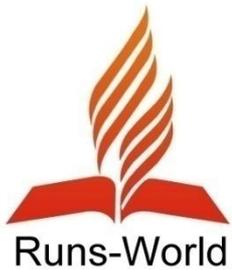


8.1.1 类的继承

- 在Java中，被继承的类叫超类（superclass），继承超类的类叫子类（subclass）。因此，子类是超类的一个功能上的扩展，它继承了超类定义的所有属性和方法，并且添加了特有功能方法。
- 首先举一个典型例子来说明继承有什么特点，然后再结合代码学习。
- 有一对爷俩，爸爸和儿子，爸爸的眼睛是单眼皮，个子很高，头发很好，皮肤很黑，而儿子同样有他爸爸的一些特征，但是儿子的皮肤很白，双眼皮，戴眼镜，在外人看来他们是爷俩。儿子具有爸爸的所有特征，但是儿子的皮肤很白和戴眼睛这些是儿子自己所特有的，也是和爸爸不一样的地方。这个小例子正是日常生活里常见的。
- 换到Java里，类与类之间的关系，可以看成倒置的金字塔，爸爸在上面，儿子在下面。爸爸可能有多个儿子，但是一个儿子只能有一个爸爸，这在日常生活里也是如此。



- 本节学习了继承的使用，下面对其内容做如下总结。
- 通过继承定义类，可以简化类的定义，让所需要的功能用相应的子类去定义和实现。
- **Java**是单继承的，子类可以有很多，父类只能有一个。上面的例子，如果加一个**Friend**类，**Son**只能继承自**Father**，要么继承**Friend**，不能同时继承**Father**和**Friend**。
- **Java**的继承是多层继承的，是一个类可以有很多子类，而子类下面又可以有很多子类。
- 父类里的属性和方法可以让子类所有，父类里的属性和方法可以使子类同样拥有，而子类的不能调用父类的方法和属性，子类的无参构造器默认是调用的父类的无参构造器。
- 父类没有定义一个无参的构造器，那么编译器就默认生成一个无参的构造器，也可以在子类构造器里显示使用**super**方法调用父类构造器，**super**方法里写几个参数就可以表示调用的是父类的哪一个构造器。
- 一般情况下，定义了一个有参的构造器，就应该定义一个无参的构造器。

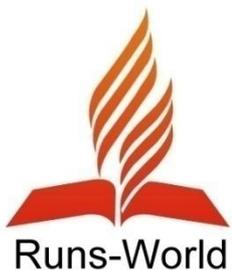


8.1.2 继承的语法

- 类的继承是通过Java保留的关键字extends来修饰的，通过extends的关键字表明前者具备后者的公共的成员变量和方法，在具备了所有的公共的成员变量和方法后，本身还能定义一些特有的成员变量和方法。基本语法如下所示。
- class 类名 extends 父类名称
- 【范例8-6】下面是使用继承的程序。

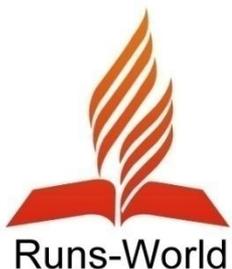
- 示例代码8-6

- 01 public class Father
- 02 {
- 03 public String name; //父亲的名字
- 04 public int age; //父亲的年龄
- 05 public String eye; //父亲眼睛的样子
- 06 public String height; //父亲的身高
- 07 public String cutis; //父亲的皮肤的颜色
- 08 }
- 09
- 10 public class Son extends Father //Son类继承与Father类
- 11 {
- 12 public String spectacle frame; //这个属性是儿子所特有的 表示儿子是否带眼镜
- 13 }



8.2 修饰符

- 修饰符是修饰的当前成员变量的访问限制和状态的。就好比一个眼镜，颜色是黑色的，这个黑色就修饰了这个眼镜，而眼镜的种类很多可以让不同的人群来使用，如近视眼镜就由有近视眼的人群来使用，别人来使用的话就不好了。
- `public String name;` //public 就是一个修饰符
- 成员变量的继承是指B继承与A后，B能使用A的属性和方法，是受成员变量的修饰符决定的。在上一节的例子中的成员变量都是使用的默认修饰符，本小节将详细介绍修饰符是如何限制成员变量的继承的。主要有4个修饰符：`public`、`private`、`default`、`protected`，对其详细介绍将分为小节来进行。



8.2.1 public: 声明成员变量为公共类型

- public表明被它修饰的成员变量为公共类型，那么这个成员变量在任何包里都能访问，包括子类也能访问到，下面用代码来说明。
 -
- 下面是使用public修饰符的程序。

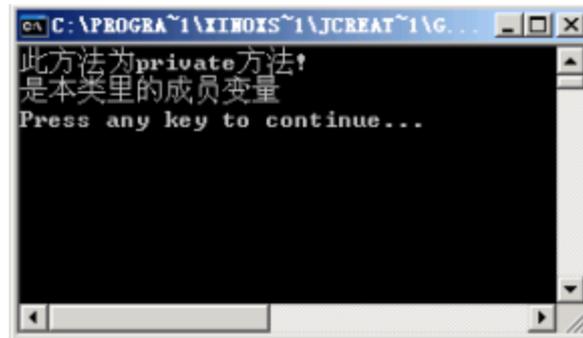
```
C:\PROGRA~1\ATMOIS~1\JCREAT~1\GE2...
门的颜色为黄色
此方法为public方法,为开门的动作
是本类里的成员变量
Press any key to continue...
```

public 修饰符↵



8.2.2 private: 声明成员变量为私有类型

- private表明被它修饰的成员变量为私有类型，表示除了本类外任何类都不能访问到这个成员变量，具有很好的保护性。下面用代码来说明。
- 【范例】下面是使用private修饰符的程序。

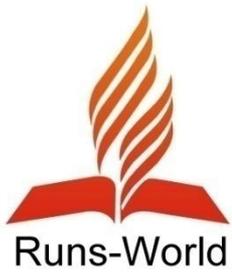


private 修饰符



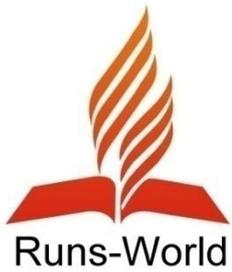
8.2.3 default: 声明成员变量为默认类型

- 如果不给成员变量添加任何修饰符，就表示这个成员变量被修饰为default类型，在一个同包里的类或子类是能够访问的，相当于public类型，但是在不同包里的类或子类没有继承该成员变量，是访问不到它的。



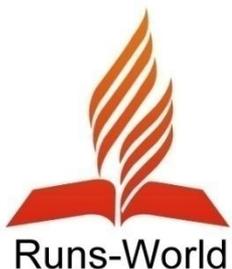
8.2.4 protected: 声明成员变量为保护类型

- protected表明被它修饰的成员变量为保护类型，在同一包里和public类型是一样的，也是能够访问到的，但是如果不同包里的protected类型的成员变量就只能通过子类来访问，这个修饰符是区别于其他的修饰符的。



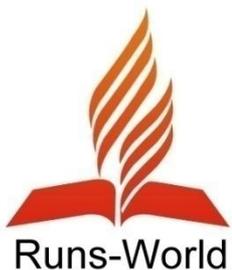
8.3 成员变量的覆盖

- 正如前面所举爸爸和儿子的例子，爸爸的眼睛是单眼皮，儿子的是双眼皮，不能说儿子没有继承爸爸的特性，只能说明儿子的特性把爸爸的覆盖了。成员变量的覆盖是子类里有和父类里相同的成员变量或方法，继承的关系，子类的成员变量将会使用，而父类的成员变量被保护起来。有时也因修饰符原因而变化，下面用代码来说明。



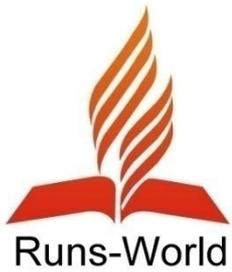
8.4 对象引用

- 对象引用就好比一个人的名字，是一个代号。也是为了方便和容易记忆所用的。比如去商店里买水果，进门就说我要买水果，而售货员也不知道要买的是什么。在Java里定义了一个类，这个类里有很多的成员变量和方法，再给这个类起一个名字，这个名字就是这个对象的引用。
- `bike b = new bike ();`
- 代码说明：
- `bike b`是创建Like类的一个对象应用，而这个**b**就相当于**bike**的名字。
- `new bike ()`相当于把**bike**这个类实例化了，真实存在与内存当中了。



8.5 方法的重写和重载

- 方法的重写和重载是体现继承特性的重要方面，理解了方法的重写和重载，可以为以后学习多态打下基础，本节重点学习重写和重载的用法和区别。

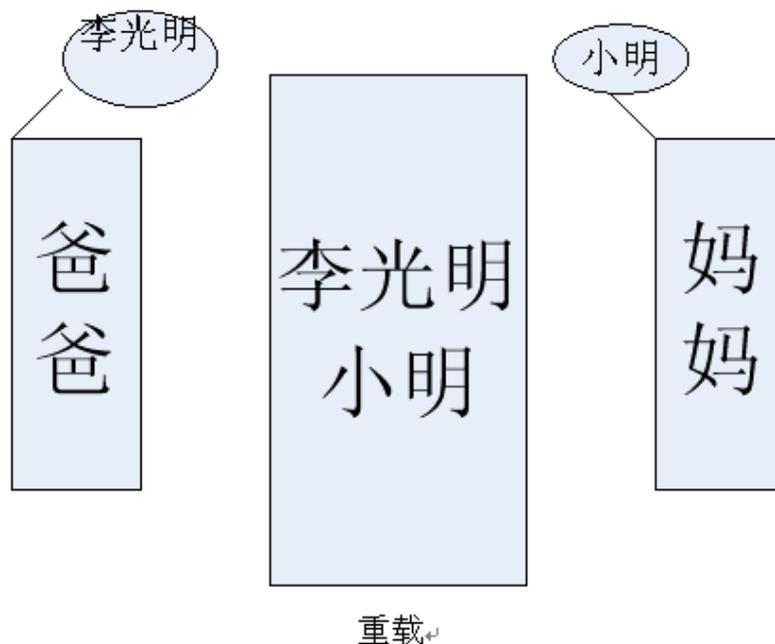


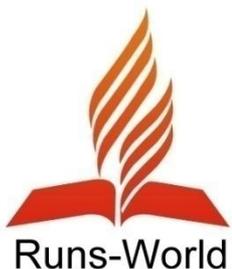
8.5.1 方法重写的特点

- 自行车的移动和公路赛车的移动都是靠外力来移动，二者是相同的。公路赛车继承了这一特点，但公路赛车的移动速度就不相同了，移动的快慢就是由它们各自移动特性不相同决定的，方法继承的特点和成员变量的覆盖很类似，但也有特殊情况，方法重写也可以叫方法的覆盖。关键字为override。
- **【范例】**下面用例子说明在日常生活中自行车和公路赛车的相同点和不同点。

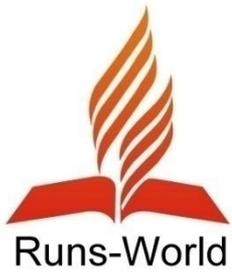
8.5.2 方法重载的特点

- 方法的重载就好比日常生活中人的名字，有大名也有小名，但这些名字都指的是这个人，不同点就是让这个人去做的事情可能不一样，如图所示。





- 这和Java里的重载很相似，关键字为**overload**。下面看一下重载的要求。
- 重载的方法名称相同，但方法的参数列表不相同。如参数个数和参数类型等。图8-9 重载
- 重载的方法的返回值可以相同也可以不相同。
- 代码演示：
 - `public String move(){};`
 - `public String move(String name){};`
 - `public void move(String name, int spend){};`
- 代码说明：
 - 虽然三种方法的名称是相同的，但这三种方法的参数列表，即个数和类型，是不相同的。
 - 判断方法是否是重载，看参数列表是很重要的。
 - 什么是传递基本类型，所谓基本类型就是用于数学计算的那些类型。当有两个名称一样的方法时，根据传递数值的类型来匹配哪两个方法的参数列表是相同的。



8.5.3 重写的返回类型

- 方法有它的返回类型，而重写的方法也有返回类型，并有一些相应的限制。方法被重写后，返回的类型为基本类型时，重写方法的返回要必须一样，否则会出现错误。
- **【范例】**下面是演示重写返回类型的代码。

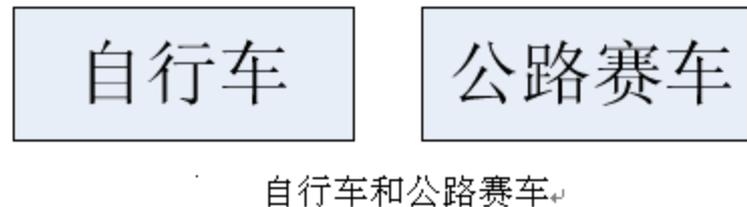
```
C:\PROGRA~1\XINNOIS~1\JCREAT...
21
Press any key to continue...
```

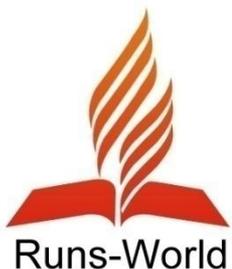
修改后↵



8.5.4 重写是基于继承的

- 重写和重载的产生是基于继承的，如果没有发生继承就不会产生重写和重载了。举个例子来说，自行车通过外力可以移动，而公路赛车通过外力也可以移动，公路赛车继承了自行车的特性。也可以说成，公路赛车是自行车的一个升级版本。相当于一个参照点吧，如图所示。





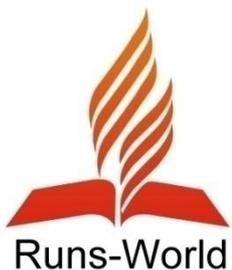
8.5.5 静态方法是不能重写的

- 静态方法就是被修饰为了static类型的方法，如果在类里声明具有唯一性，不是通过类的实例化而存在的，而是通过类的建立而存在，可以理解为用关键字new创建对象了，就是把这个对象实例化了。
- 对本节的内容进行如下总结。
- 父类的静态方法可以被子类的静态方法覆盖。
- 父类的非静态方法不能被子类的静态方法覆盖。
- 父类的静态方法不能被子类的非静态方法覆盖。
- 覆盖是用于父类和子类之间。
- 重写是用在同一个类中，有相同的方法名，但参数不一样。



8.5.6 三者之间的关系

- 重写的关键字是`override`，重载的关键字为`overload`，重写、重载、覆盖都是基于继承的关系。当继承的关系发生了，想用父类的方法就用`super`关键字来引用，如果想用新的方法了就重写下，来完成新的功能需要。对覆盖总结如下几点：
- 覆盖的方法的参数列表必须要和被覆盖的方法的参数列表完全相同，才能达到覆盖的效果。
- 覆盖的方法的返回值必须和被覆盖的方法的返回值一致。
- 覆盖的方法所抛出的异常必须和被覆盖方法的所抛出的异常一致，或者是其子类。
- 被覆盖的方法不能为`private`，否则在其子类中只是新定义了一个方法，并没有对其进行覆盖。

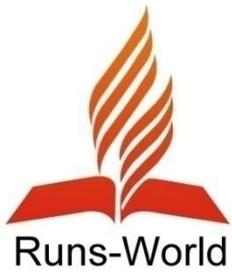


- 对重载总结如下：
 - 使用重载时只能定义不同的参数列表。
 - 不能通过重载的方法的返回类型、访问权限和抛出的异常来进行方法的重载。
- 对重写总结如下：
 - 重写的方法存在于父类中，也存在于子类中。
 - 重写的方法在运行期采用多态的形式。
 - 重写的方法不能比被重写的方法有更高的访问限制。
 - 重写的方法不能比被重写的方法有更多的异常。



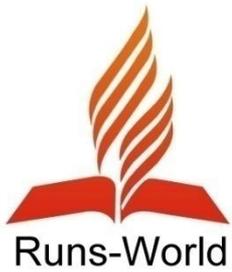
8.5.7 重写toString

- toString()方法是Java里Object类的方法，很多类都重写了该方法。该方法返回对象的状态信息。下面是这个方法的原型：
- `public String toString()`



8.5.8 重写equals

- 方法equals也是Object类的方法，很多类也进行了重写，一般重写equals方法是为了比较两个对象的内容是否相等。下面是这个方法的原型：
- `public boolean equals (Object obj)`
- `{`
- `return (this == obj) ; //这里比较的是两个对象的引用`
- `}`



8.6 final与继承的关系

- final关键字有最终、不变的意思，可以修饰成员变量，也可以修饰方法和类，通过final关键字的修饰可以改变其特性。
- final关键字修饰类时，说明其类不能有子类，也就是说该类不能被继承，该类的成员变量在这里将不起作用。
- final关键字修饰方法时，说明该方法不能被重写，因为类都不能继承了，方法就更不能重写了。
- 类里可以含有final关键字修饰的方法。
- final关键字修饰的成员变量的对象引用不能修改。
- final关键字修饰的类里的方法默认被修饰为final。

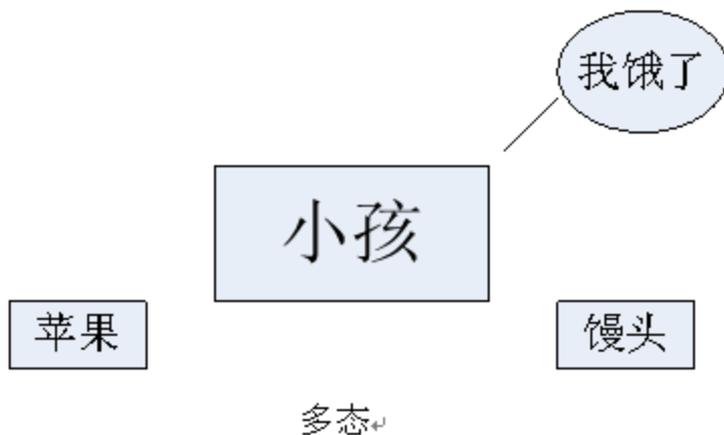


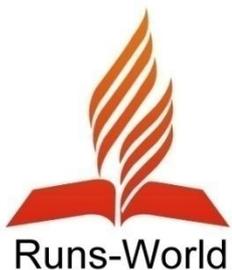
8.7 abstract与继承的关系

- abstract关键字表示抽象的意思。所谓抽象，就好比在日常生活中人们设计的图纸，而这个图纸就好比是一个抽象的房子似的，需要把房子盖起来来实现这个图纸。在Java里抽象类里最少要含有一个抽象方法，让它的子类去实现这个抽象方法，抽象类里也可以定义一些方法。

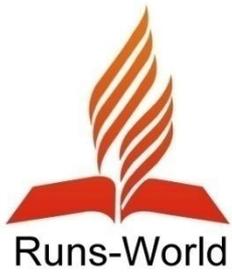
8.8 什么是多态

- 这里来拿苹果和馒头来打个比方，苹果属于水果的一种，馒头属于面食的一种，而苹果和馒头都可以属于食物，一种物品有两种状态表现，这就是多态，如图所示。



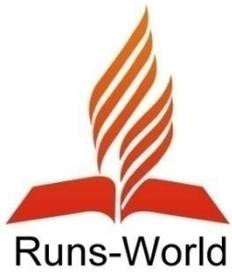


- 对本节学习的内容总结如下：
- **static**修饰的方法和**final**修饰的方法是在编译期绑定的；而其他的方法是在运行期绑定，动态地判断是什么类型。
- 多态是基于继承的，是类和接口相结合来实现的。



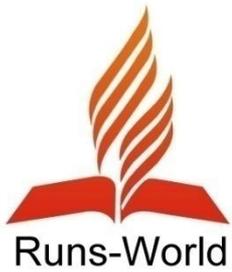
8.9 什么是枚举类

- 所谓枚举就好比日常生活中的星期一、星期二到星期天，是一组连续的数据。在Java里枚举类就是一组连续的基本类型的数值。下面举例如何创建枚举类型。
- `public enum Grade`
- `{`
- `A, B, C, D, E, F`
- `};`



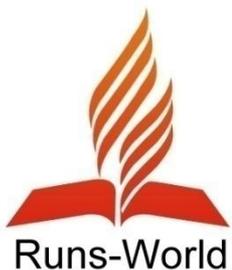
8.10 什么是反射机制

- 在日常生活中，通过放大镜可以看清楚物体的内部结构。在Java中，反射机制就是起到放大镜的效果，可以通过类名，加载这个类，显示出这个类的方法等信息。

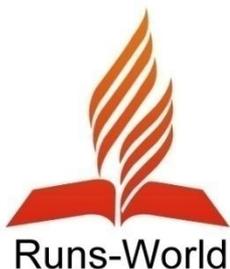


8.11 什么是泛型

- 在日常生活中，橡皮泥通过外力可以改变其形状，其形状是不固定的。在Java中，通过泛型可以给开发带来方便，通过参数的指定，可以改变其类型。下面通过代码演示。

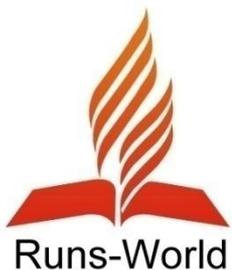


- 使用泛型给程序员的代码编写带来了好处，也带来的缺点，了解它的好处和缺点，能给程序编写带来很多好处和便利。对泛型的好处总结如下：
- 使用泛型，正如上面代码所示，能使代码看起来灵活；容易管理，不容易产生错误。
- 使用泛型能使代码量减少，能产生很多公共代码。
- 使用泛型在代码编译的时候能进行类型的检查并自动转换，使代码的运行效率得到提高。
- 使用泛型在编译进行自动转换的时候出现了错误，会进行错误提示。
- 使用泛型的时候参数只能是类的类型，不能是简单类型。
- 使用泛型的时候参数可以有多个。
- 使用泛型的时候参数也能继承别的类型。



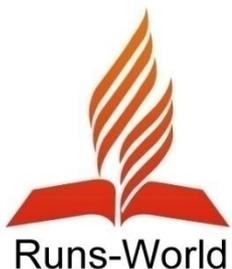
8.12 综合练习

- 1. 四种权限修饰符的不同点有哪些？
- **【提示】**从基本定义中进行分析。public修饰符表明被它修饰的成员变量为公共类型，这样这个成员变量在任何包里都能访问，包括子类也能访问到。private表明被它修饰的成员变量为私有类型，表示除了本类外任何类都不能访问到这个成员变量，具有很好的保护性。如果不给成员变量添加任何修饰符，就表示这个成员变量被修饰为default类型，在同一个包里的类或子类是能够访问的，就相当于public类型，但是在不同包里的类或子类没有继承该成员变量，是访问不到的。protected表明被它修饰的成员变量为保护类型，在同一包里和public类型是一样的，也是能够访问到的，但是如果不同包里的protected类型的成员变量就只能通过子类来访问，这个修饰符是区别于其他修饰符的。
- 2. 重写和重载的区别有哪些？
- **【提示】**重写是基于继承的，重写是重写父类中的方法，从而在子类中出现一个和该方法相同名称的方法。重载的方法名称相同，但方法的参数列表不相同。如：参数个数和参数类型等。重载的方法的返回值可以相同也可以不相同。



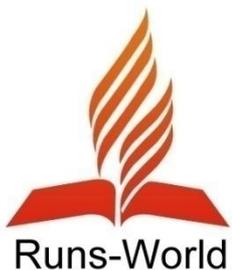
8.13 小结

- 通过本小节的学习，读者可以了解继承的相关概念、用法和注意事项等。对修饰符所修饰的成员变量和方法要多多理解，对以后的编码有帮助。本章重点学习，方法和成员变量的重写、重载、覆盖这些概念。如果想了解更多的关于本章的内容，可以参考电子工业出版社出版的《深入浅出JDK 6.0》一书进行学习。



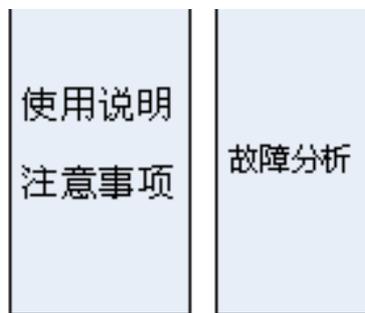
第9章 接口

- 本章将要介绍接口的各方面的知识，主要有抽象类，以及多态的特性等。通过学习这些知识可以让读者更深入地了解面向对象的思想，以及在平常的编码中的一些注意事项。通过本章的学习，读者应该完成如下几个目标。
- 会定义接口和访问接口中的变量。
- 熟练掌握接口的使用。
- 了解接口和抽象类的区别。
- 了解接口的多态问题。
- 熟练掌握使用instanceof判断类型。

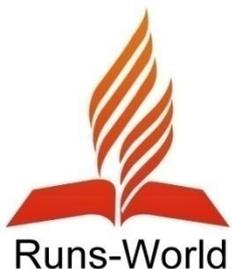


9.1 什么是接口

- 所谓接口就是一个完成某些特定功能的类。在日常生活中就好比一个产品的说明书，通过阅读说明书可以让消费者更多地了解产品的功能以及使用中的注意事项。在Java中也是如此，接口是一个功能的集合，如图所示。

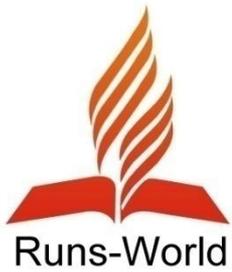


说明书



9.1.1 接口的定义

- 首先来举一个例子，汽车的移动就好比一个接口，在以后生产的汽车中都遵循这个接口进行制造。而接口中只定义了汽车移动的形式，没有具体的去定义是怎么进行移动的，所以接口就好比是一个规定。下面介绍下如何定义一个接口。
- 接口修饰符 interface 接口名称
- {
- //成员变量和方法的申明
- }
- 接口修饰符和类的修饰符是一样的。
- interface是定义接口的关键字。
- 接口里的成员变量默认为public static final类型的
- 接口不能声明为final的，因为final类型的必须要实现。



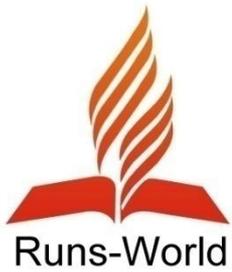
9.1.2 访问接口里的常量

- 在接口里定义的成员变量为常量，是因为接口为每个成员变量默认的修饰是`public static final`类型的，即便不显式的修饰也默认的加上了。下面通过代码来演示接口里的常量。
- **【范例】** 示例代码来是演示接口里的成员变量为常量。



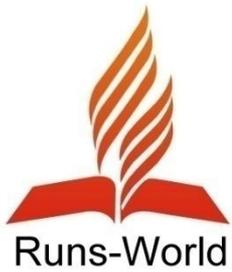
Runs-World

```
• 01 //创建一个包，名字为a
• 02 package a;
• 03
• 04 //创建一个接口名字为aaa
• 05 public interface aaa
• 06 {
• 07     int i = 2;
• 08 }
• 09
• 10 import a.aaa;
• 11
• 12 //test类描述的是访问接口的常量
• 13 public class test
• 14 {
• 15     //Java程序的主入口方法
• 16     public static void main(String[] args)
• 17     {
• 18         //取得接口里的值
• 19         int n = aaa.i + 1;
• 20
• 21         //打印并显示结果
• 22         System.out.println(n);
• 23     }
• 24 }
```



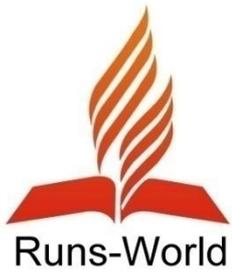
9.2 接口的使用

- 接口就是一个特殊的抽象类，抽象类里有抽象的方法和普通的方法，而接口里方法全为抽象的，需要在其子类进行具体的方法实现。类就是一个产品的详细功能说明，而接口就是这些功能的简要说明。在本节里详细说明接口是如何使用的，以及它们的注意事项。



9.2.1 接口里的方法如何创建

- 在接口里创建方法和一个类里的方法很相似，但不同点就是接口里的方法没有具体的方法体，而类里的方法必须去实现其方法体。下面用一段代码来演示在接口里如何定义方法。
- 01 //创建一个接口名字为aaa
- 02 public interface aaa
- 03 {
- 04 //创建一个接口方法getMax()
- 05 public int getMax();
- 06
- 07 //创建一个接口方法getMes()
- 08 String getMes();
- 09 }



- 总结一下接口的方法和类里的方法的区别。在定义接口里的方法和类里的方法都是有一定规则的，但是它们之间的定义是有一定区别，分别如下：
- 接口里的方法默认被修饰为public、abstract类型。
- 类里的方法如果修饰为abstract类型，将提示错误。
- 接口里的方法不能是static、final类型，只能为public、abstract类型。
- 类里的方法不能为final、abstract类型。



9.2.2 接口引用怎么使用

- 在前面小节里介绍了接口创建及其注意事项，创建接口就是为了使用。下面介绍如何使用接口，以及使用接口的注意事项。接口的语法为：类的修饰符 class 类名称 implements 接口名称。
- 通过上面的语法结构可以看出和类的继承很相似，下面通过一段代码来演示接口是如何实现的。
- 01 //创建一个包,名字为a
- 02 package a;
- 04 //创建一个接口名字为aaa
- 05 public interface aaa
- 06 {
- 07 //创建一个接口方法getMax
- 08 public int getMax();
- 09
- 10 //创建一个接口方法getMes
- 11 String getMes();
- 12 }



- 14 import a.aaa;
- 15
- 16 //test类描述的是实现接口的方法
- 17 public class test implements aaa
- 18 {
- 19 //实现接口里的方法
- 20 public int getMax()
- 21 {
- 22 //具体的方法体
- 23 return 0;
- 24 }
- 25
- 26 //实现接口里的方法
- 27 public String getMes()
- 28 {
- 29 //具体的方法体
- 30 return null;
- 31 }
- 32 }



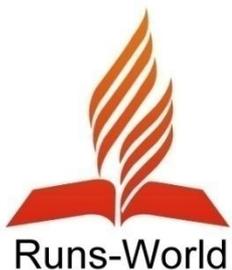
9.3 什么是抽象类

- 抽象类和接口是有些类似的，抽象类需要其他类继承来实现抽象类中的方法，以及给出更多的方法。在日常生活中的一个产品的简要介绍和详细介绍的结合，说明了产品具有什么功能，和这个功能都完成了什么。在Java中也是类似的，接口是抽象类的特殊版本。接口里必须都为抽象的，而抽象类里可以为抽象的也可以有其他形式的存在。

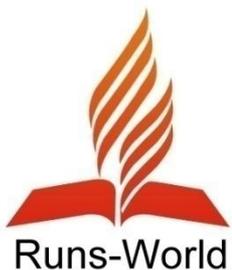


9.3.1 抽象类的使用和特点

- 抽象类和一般的类很相似，所谓抽象类，就是在类里存在一些没有方法体的方法，即抽象方法。下面通过一段代码来演示抽象类。
 -

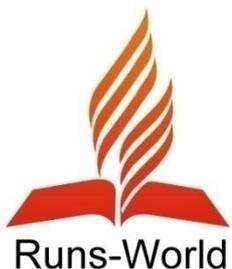


- 下面总结抽象类的特点，如下所示。
- 抽象类一般在父类中使用，而它的子类实现父类中的抽象方法。
- 如果父类有一个或多个抽象方法，那么父类必须为抽象类。
- 抽象类里的抽象方法没有任何方法体，子类要实现父类的所有抽象方法。如果没实现抽象方法，其子类即为抽象类。
- 抽象类是用来继承的，不能被实例化。
- 所有的抽象方法都必须声明在抽象类中。
- 抽象类里的抽象方法，只有在子类实现了才能使用。
- 抽象类里允许有抽象方法和普通方法。
- 抽象类里的普通方法可以被子类调用。



9.3.2 抽象类与接口区别

- 在前面的小节里学习了抽象类和接口，它们之间很相似但是也是有区别的。它们之间的区别如下：
- 抽象类中有一个抽象方法或多个抽象方法。
- 如果抽象类的子类里有一个没有实现的抽象方法，那么这个类也是抽象类。
- 实现抽象类里的方法可以实现部分方法，也可以实现所有方法。
- 抽象类里可以有成员变量。
- 抽象类里可以有私有的方法和私有的成员变量。
- 接口中的方法全部都被修饰为抽象方法。
- 接口里的方法都被默认修饰为public abstract类型的。
- 接口里的变量都被默认修饰为public static final类型的，即常量。
- 一个类可以实现一个接口，也可以实现多个接口。
- 接口里的方法必须要全部实现。
- 接口里没有成员变量。
- 接口里的方法全部都是public，即公共类型的。

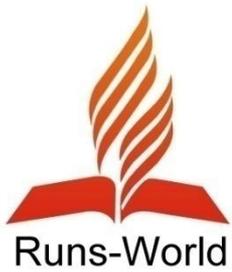


9.4 接口的多态

- 所谓多态，就好比日常生活中的橘子和羊肉都是食物的一种，而橘子又是水果的一种，羊肉是肉类的一种，橘子和羊肉是两种不相同的食物，但是食物可以同时指向它们这两种物品。这就是日常生活中多态的形式。在Java中也是如此。食物的对象可以指向橘子和羊肉对象。这样给编写代码带来了很大的灵活性。
- **【范例】**下面通过一个例子来演示接口是如何实现多态的。首先创建一个food接口。

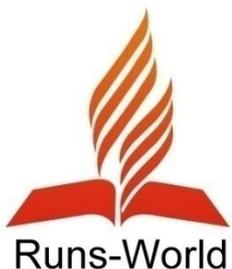
- 示例代码

- 01 //创建一个food接口
- 02 interface food
- 03 {
- 04 //得到食物的名称
- 05 public void getname();
- 06
- 07 //吃食物的方法
- 08 public void eat();
- 09 }



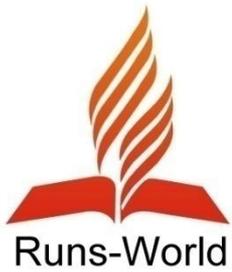
9.5 判断类型

- instanceof一般使用于多态的时候，在代码中判断对象的引用类型具体为哪一种类型。根据不同的对象类型来执行不同对象中的方法。本小节介绍什么是instanceof，以及使用它的注意事项。



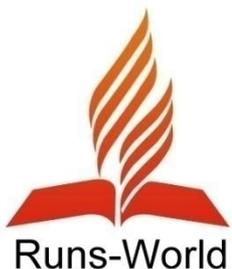
9.5.1 什么是instanceof

- 所谓instanceof在字面上可以理解为检查实例。instanceof在Java中是一个二元操作符，也是Java所保留的关键字。下面介绍instanceof的语法结构：
- 对象的引用 instanceof 类或接口
- instanceof语句的返回结果是boolean类型的。如果返回true，说明对象的引用是该对象所指的类或接口；如果返回false，说明对象的引用是该对象所指的类或接口。



9.5.2 使用instanceof的注意事项

- 在使用instanceof进行对象类型判断的时候也是有规则要遵循的，下面总结使用instanceof有哪些规则。
- instanceof关键字不能比较基本类型的数据。
- instanceof关键字可以对对象和接口使用。
- instanceof关键字的比较是基于多态的。
- 不推荐使用instanceof关键字，要多多应用多态。
- instanceof关键字右边比较的类型只能为类和接口。



9.6 综合练习

- 说明抽象类和接口的区别。
- **【提示】**抽象类中有一个抽象方法或多个抽象方法。接口中的方法全部都被修饰为抽象方法，并且接口里的方法都被默认修饰为 `public abstract` 类型的。
- 如果抽象类的子类里有一个没有实现的抽象方法，那么这个类也是抽象类。一个类可以实现一个接口，也可以实现多个接口。
- 实现抽象类里的方法可以实现部分方法，也可以实现所有方法。接口里的方法必须要全部实现。
- 抽象类里可以有成员变量。抽象类里可以有私有的方法和私有的成员变量。接口里的变量都被默认修饰为 `public static final` 类型，即常量。接口里没有成员变量。接口里的方法全部都是 `public`，即公共类型的。



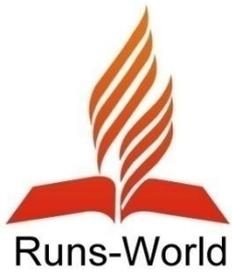
9.7 小结

- 本章学习了一个更重要的知识——接口，对象通过多态可以使其变得灵活，而接口可以使对象变得更加灵活。接口是基于继承的，通过在编译期和运行期来扮演不同的类型。学好本章知识能更好地了解面向对象的知识。如果想了解更多的关于本章的内容，可以参考电子工业出版社出版的《深入浅出JDK 6.0》一书进行学习。



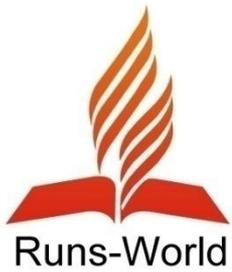
第10章 构造器

- 构造器在前面的章节里的代码，读者已经看到很多次了。通过用new关键字来调用构造器使其对象在内存中创建出来。下面将要详细的介绍构造器的一些知识。通过本章的学习，读者应该能够完成如下几个目标。
- 了解什么是构造器。
- 熟练掌握如何创建构造器。
- 熟练掌握构造器的使用，包括构造器如何调用等问题。
- 了解构造器的一些基本机制。



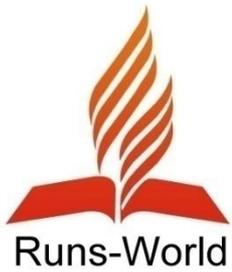
10.1 什么是构造器

- 在日常生活中，盖房子需要工具和工人，通过工人使用这些工具，来修建一个房子。在Java中，构造器就好比是工具，而new关键字就是工人，通过new关键字和构造器结合来创建对象。



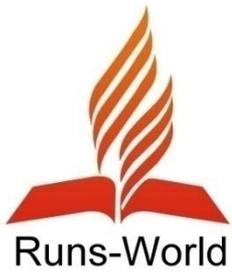
10.1.1 构造器的使用

- 要建立对象就要使用new关键字来建立对象，这是建立对象唯一的方法。下面介绍构造器的语法组成。
- 类的修饰符 类的名称(参数列表)
- {
- //方法体
- }
- 构造器可以使用的修饰符有public、protected、default、private，不写即为default类型的。
- 构造器的名称必须要和类的名称相同。
- 不能有返回值，void也不行。
- 构造器的参数可有可无。有一个也可有多个参数。



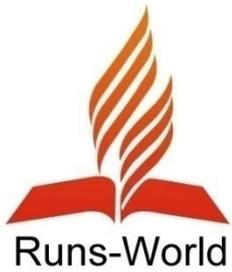
10.1.2 被修饰的构造器

- 构造器是可以被修饰符修饰的，不同的修饰符修饰构造器也具有不同的效果，本小节通过使用不同的修饰符来进行代码演示。



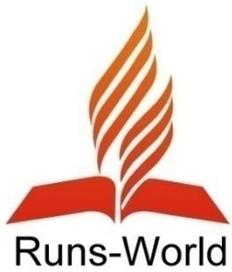
10.1.3 构造器方法与普通方法的区别

- 构造器方法和普通的方法是有一定区别，主要是功能上、修饰符上、返回值上和命名上有本质的区别。区别如下：
- 构造器是为了创建一个类的对象实例，也可以在创建对象的时候使用。
- 方法是为了执行相应的方法体。即Java代码。
- 构造器可以被修饰为public、protected、default、private类型，但不能修饰为abstract、final、native、static、synchronized
- 方法可以修饰为除了protected、native外的修饰符。
- 构造器没有返回值也没有void。
- 方法没有返回值或有任何类型的返回值。
- 构造器的名称要和类的名称相同。
- 方法的名称可以任意起，但要注意标识符的命名规则。使其更具有意义。



10.2 如何实例化一个对象

- 所谓实例化就是在内存中实实在在的创建一个对象，在日常生活中就好比创造了一个东西出来。而在Java中，实例化一个对象用new关键字来完成。下面先介绍new关键字的语法以及通过一个例子来演示。
- new 构造器的名称(参数列表)
- new为Java关键字要注意大小写。
- 构造器的名称要和类的名称相同。
- 通过调用构造器方法来对这个对象进行一些必要的初始化。
- 用new关键字实例化对象后返回该对象的引用。



10.3 构造器的使用

- 通过前面的介绍，读者已经对构造器有了基本的了解。但是使用构造器也是有一些注意事项的。在本节里介绍构造器在父子类中是如何使用的。



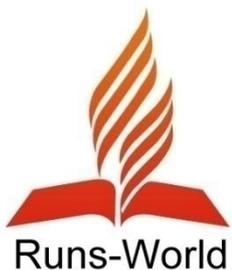
10.3.1 构造器的调用

- 构造器的调用一般有两种情况，一般是在本类里调用或同包下的另一个类调用，另一种情况是子类调用父类的构造器的。下面通过代码来分别演示。



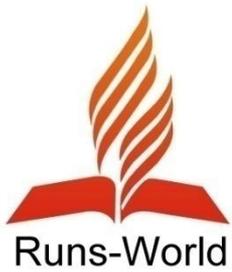
10.3.2 构造器的重载

- 所谓构造器的重载和方法的重载是一样的，重载就好比日常生活中人的名字，有大名有小名，但这些名字都指的是这个人，不同点就是让这个人去做的事情可能不一样。这个和Java里的方法的重载很相似。下面来看一下重载的要求。
- 构造器的重载的方法名称相同，但参数列表不相同。如：参数个数和参数类型等。
- 构造器的重载的方法是没有返回值的。
- 构造器不能被继承，这和方法有所区别。
- 构造器的修饰符只有public、private、protected这三个。



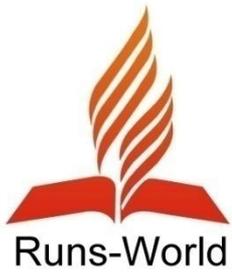
10.3.3 父子类间的构造器的调用流程

- 在前面学习过用new关键字来创建一个对象，但在继承关系发生时，父类与子类是如何创建对象的呢。它们的顺序又是怎样的呢。
- 详细的步骤如下所述。
- 在用new关键字创建对象aceing的时候。执行new aceing()会进入到aceing对象的构造器方法体内。
- 因为继承的关系，会默认调用方法super进入到父类bike对象的构造器方法体内。
- 对父类bike对象进行初始化。父类的构造器方法执行完毕后回到子类的构造器继续执行。
- 执行子类的构造器方法，并初始化数据。



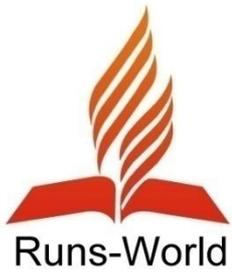
10.3.4 如何自定义构造器

- 自定义的构造器已在前面的代码中多次使用。所谓自定义构造器就是不显式的定义构造器，编译器就是自动地生成一个无参的构造器，但是一旦显式的定义了一个构造器的话，编译器就不会自动生成了。下面用代码演示如何自定义构造器。
- ```
public class test
```
- ```
{
```
- ```
//定义一个无参的构造器
```
- ```
public test()
```
- ```
{
```
- ```
//该构造器的方法体
```
- ```
}
```
- ```
//具有两个参数的构造器
```
- ```
public test(String i, int n)
```
- ```
{
```
- ```
//该构造器的方法体
```
- ```
}
```
- ```
}
```



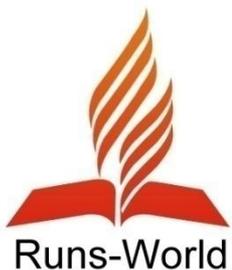
## 10.4 什么是单子模式

- 单子模式是Java模式工厂里的一种，所谓单子模式，就是在一个时间段内对象只存在一份。下面通过代码来演示什么是单子模式。单子模式就是把构造器修饰为private类型的，用一个public类型的方法返回该对象的引用。



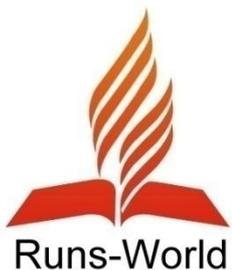
## 10.5 构造器在程序中是何时运行的

- 前面的章节里介绍了构造器是为了创建对象并对其对象的成员变量进行初始化等操作，那么在构造器运行前系统执行什么呢，之后又执行什么呢。下面总结一下。
- 加载要创建该对象的父类，以及成员变量和其他继承关系。
- 加载该类的静态块和静态成员变量，并对其进行初始化等操作。
- 静态块和静态成员变量加载完毕后创建对象并加载非静态成员变量，并对其进行初始化等操作。
- 执行构造器里的方法体，完成后该类的对象创建完毕。
- 父类的运行顺序和该类的顺序是一样的。



## 10.6 综合练习

- 1. 看下面的程序有什么错误。
- 01 public class LianXi1
- 02 {
- 03 public LianXi1()
- 04 {
- 05 System.out.println("调用无参构造器");
- 06 new LianXi1("hello");
- 07 }
- 08 public LianXi1(String s)
- 09 {
- 10 System.out.println("调用有参构造器");
- 11 new LianXi1();
- 12 }
- 13 public static void main(String args[])
- 14 {
- 15 new LianXi1();
- 16 }
- 17 }



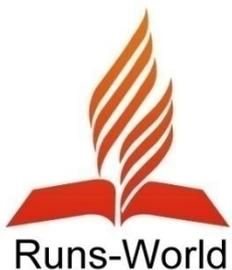
## 10.6 综合练习

- 2. 编写一个构造器重载的程序，在每一个构造器中显示一条语句。
- 【提示】可以采用构造器间调用。
- 01 public class LianXi2
- 02 {
- 03 public LianXi2()
- 04 {
- 05 new LianXi2("A"); //调用具有一个参数的构造器
- 06 }
- 07 public LianXi2(String s)
- 08 {
- 09 System.out.println("参数值为: "+s);
- 10 new LianXi2("B","C"); //调用具有两个参数的构造器
- 11 }
- 12 public LianXi2(String s,String ss)
- 13 {
- 14 System.out.println("第一个参数值为: "+s+", 第二个参数值为: "+ss);
- 15 new LianXi2("D","E","F"); //调用具有三个参数的构造器
- 16 }
- 17 public LianXi2(String s,String ss,String sss)
- 18 {
- 19 System.out.println("参数包括: "+s+", "+ss+", "+sss);
- 20 }
- 21 public static void main(String args[])
- 22 {
- 23 new LianXi2(); //调用无参构造器
- 24 }
- 25 }



## 10.7 小结

- 本章介绍了构造器的知识，以及它的使用和注意事项。了解构造器方法的执行顺序对了解程序的执行有很好的帮助。希望读者重点了解构造器的使用这个小节里的内容。如果想了解更多的关于本章的内容，可以参考电子工业出版社出版的《深入浅出JDK 6.0》一书进行学习。



## 第11章 异常处理

- 每个人都不能保证所写的程序没有错误，如果程序中可能发生错误就需要进行异常处理在学校中老师批改作业一样，通常要指出学生所犯的错误的。指出错误可能是准确的指出错误，也可能是给出一个错误范围，让学生在这个范围中自己查找。在Java中，异常处理也是这样的，通过异常处理来指出程序中的错误，可以给出一个具体异常，也可以给出一个异常范围。在本章中就来学习如何进行异常处理。通过本章的学习，读者应该完成如下几个目标。
- 了解什么是异常处理。
- 熟练掌握如何进行异常处理。
- 掌握异常的分类和区别不同的异常。
- 能够自定义异常和使用自定义异常。



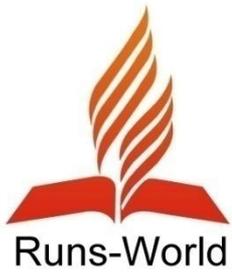
## 11.1 异常处理基本介绍

- 在本节中将对异常有一个大概的了解。异常发生的原因有很多，可能是软件的问题，也可以是硬件的问题。在Java程序中，对异常的处理都是一样的，一般情况下是通过try-catch语句来进行异常处理。该语句还可以存在finally语句。本节中就来对这些最简单的异常处理语句进行介绍。



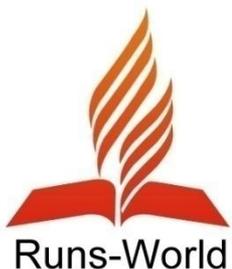
## 11.1.1 try和catch捕获异常

- 通常情况下，在Java程序中就是采用try-catch语句进行异常处理的。这种方法既好用，又容易让开发人员理解。try-catch语句的基本语法如下所示。
- try
- {
- //此处是可能出现异常的代码
- }
- catch(Exception e)
- {
- //此处是如果发生异常处理的代码
- }
- 在try语句中放可能出现异常的代码；在catch语句中需要给出一个异常的类型和该类型的引用，并在catch语句中放当出现该异常类型时需要执行的代码。

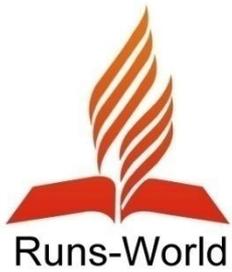


## 11.1.2 try-catch语句使用注意点

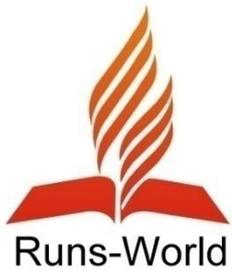
- 使用try-catch语句是有很多注意点和技巧的。在一开始学时就应该了解这些。有些初学者会认为使用了try-catch语句的程序就会发生异常，这是不对的。try-catch语句是对有可能发生异常的程序进行检查，如果没有发生异常，就不会执行catch语句中的内容。在程序中如果不使用try-catch语句，则当程序发生异常的时候，会自动退出程序的运行。而使用try-catch语句后，当程序发生异常的时候，会进行执行catch语句中的程序，从而使程序不自动退出。在前面的学习中经常会看到出现异常的情况，如果在其中使用try-catch语句就不会出现那种异常信息。



- 注意：**try-catch**语句是对有可能发生异常的程序进行查看，如果没有发生异常，就不会执行**catch**语句中的内容。在程序中如果不使用**try-catch**语句，则当程序发生异常的时候，会自动退出程序的运行。
- **try-catch**语句中的**catch**语句可以不只是一个，可以存在多个**catch**语句来定义可能发生的多个异常。当处理任何一个异常时，则将不再执行其他**catch**语句。

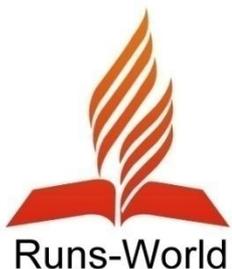


- 当对程序使用多个**catch**语句进行异常处理时，特别需要注意的是要将范围相对小的异常放在前面，将范围相对大的异常放在后面，这通过程序是很容易理解的。
- 注意：当对程序使用多个**catch**语句进行异常处理时，要将范围相对小的异常放在前面，将范围相对大的异常放在后面。

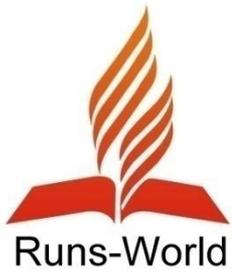


## 11.1.3 finally语句的使用

- 在try-catch语句中是还可以具有finally语句的。在实际开发中经常要使用到finally语句，尤其是将在后面学习到的数据库操作中。连接数据库是可以发生异常的，当然也是可能不发生异常的。但是有一点，不管是否发生异常，连接数据库所用到的资源都是需要关闭的，这些操作是必须执行的，这些执行语句就可以放在finally语句中。在finally语句中就是放肯定会被执行的语句。
- 提示：不管是否发生异常，连接数据库所用到的资源都是需要关闭的，这些操作是必须执行的。

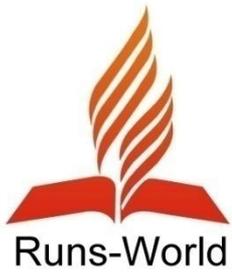


- **finally**语句的语法形式如下所示。
- **try**
- {
- //此处是可能出现异常的代码
- }
- **catch(Exception e)**
- {
- //此处是如果发生异常的处理代码
- }
- **finally**
- {
- //此处是肯定被执行的代码
- }
- **finally**语句虽然在程序中肯定执行，但是为了确保知识的严谨性，这里也给出了几个可能会中断**finally**语句执行的情况。首先是**finally**语句中本身就产生异常；再者就是执行**finally**语句的线程死亡，线程的问题会在后面的学习中学习到；还有一种情况，那就是程序执行到**finally**语句时停电了。



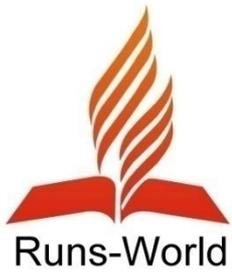
## 11.1.4 再谈异常处理注意点

- 学习finally语句后，又多出了很多在写try-catch-finally语句时需要注意的地方。这些在开发中是比较少见的，但是在考试中经常会出现。第一个注意点就是当不存在catch语句时，finally语句必须存在并且紧跟在try语句后面。读者可以自己写程序来验证这一点。
- 还有一个需要注意的格式是在try语句和catch语句间不能存在任何语句，同样在catch语句和finally语句中也不能存在任何语句，这地方的语句不包括注释语句。
- 注意：当不存在catch语句时，finally语句必须存在并且紧跟在try语句后面。在try语句和catch语句间不能存在任何语句，同样在catch语句和finally语句中也不能存在任何语句。



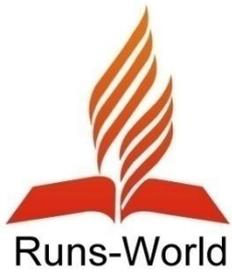
## 11.2 异常的分类

- 可以对异常进行分类的，从大的角度将异常分为捕获异常和未捕获异常两类。在Java类库中有一个叫做Throwable类，该类继承于Object类。所有的异常类都是继承Throwable类，Throwable类有两个直接子类，Error类和Exception类。在Exception类中又有一个RuntimeException类。在Exception类中的直接和间接子类中除去RuntimeException类的直接和间接子类，都是捕获异常。其他的都为未捕获异常。



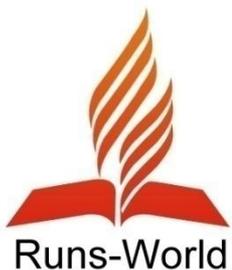
## 11.2.1 捕获异常

- 捕获异常是在翻译外文书时给起的名字，如果直接翻译的话是必须处理异常。捕获异常通常是由外部因素造成的，不是由程序造成的。例如连接网络等操作，这些是和很多因素有关系的，有可能并不是程序的错误。虽然这些错误并不是程序的错误，但是也是必须要进行处理的。
- 提示：捕获异常是在翻译外文书时给起的名字，如果直接翻译的话是必须处理异常。
- 在Java程序中对捕获异常的处理是必须进行异常处理，虽然有可能程序并不会出现异常。这种设计大大提高了程序的稳定性。下面通过程序代码来进行捕获异常的讲解。
- 注意：如果语句中不可能出现捕获异常，但是程序中仍然对该语句进行捕获异常处理，这样该程序运行时是会发生错误的。



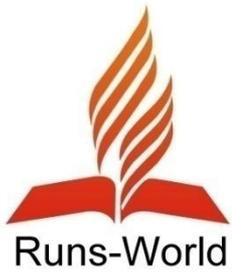
## 11.2.2 未捕获异常

- 在异常中，除了捕获异常以外的都是未捕获异常。未捕获异常包括Error类以及它的直接子类和间接子类和RuntimeException类以及它的直接子类和间接子类。Error类以及它的子类通常是由硬件运行错误所导致的错误。这些是很严重的错误，通常是不能通过程序来进行修改的。RuntimeException类以及它的子类通常是程序运行时引起的异常。
- 提示：Error类以及它的子类通常是由硬件运行错误所导致的错误。这些是很严重的错误，通常是不能通过程序来进行修改的。
- 前面使用最多的就是数组下标越界异常，该异常就是未捕获异常。未捕获异常是可以不进行异常处理的，如果不进行异常处理编译的时候是完全没有问题的，但是在运行时会发生错误。



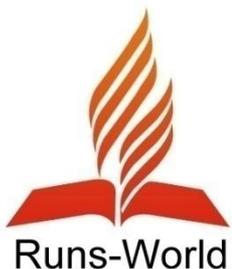
## 11.3 抛出异常

- 对异常的处理不是只有前面讲的使用try-catch语句来进行处理的。在有些情况下，异常是不需要立即进行处理的，但是也必须要进行异常处理，这时候就用到抛出异常的内容。



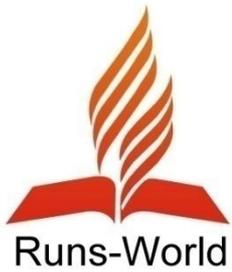
## 11.3.1 抛出异常的简单介绍

- 日常生活中，例如学校中有什么问题都会先去问老师，但是有一些问题例如转学是不能由老师来解决的，这时候老师就需要再去问校长，由校长来解决这个问题。可能校长还有不能解决的问题，就需要去问教育部。抛出异常也是这样的，当一个程序段发生异常时，如果自己不能够进行异常处理，就可以将抛出异常给上一层。如果上一层也不能解决就可以一直向上抛出异常，直到抛出给main方法。如果仍然不能解决，就会中断程序，将异常显示出来。
- 技巧：当一个程序段发生异常时，如果自己不能够进行异常处理，就可以将抛出异常给上一层。如果上一层也不能解决就可以一直向上抛出异常，直到抛出给main方法。



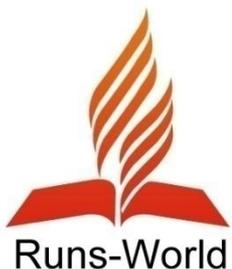
## 11.3.2 使用throws和throw语句抛出异常

- 在抛出异常的操作中，不但可以使用上一小节中的抛出异常的方法，还可以使用throws语句和throw语句进行抛出异常。throws语句是在方法的声明中使用来抛出异常，而throw语句是在方法体内使用抛出异常。
- 提示：throws语句是在方法的声明中使用来抛出异常，而throw语句是在方法体内使用抛出异常。
- **【范例11-12】** 示例代码11-12是一个使用throws和throw语句抛出异常的程序。



## 11.4 自定义异常

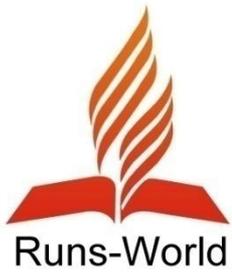
- 在Java中定义了非常多的异常类，几乎覆盖了可能出现的所有问题。但是再多的定义也不可能满足所有的情况，这时候就需要来进行自定义异常。在本节中就来学习如何创建自定义异常类，并让读者学习如何使用自定义异常。



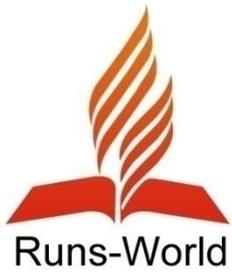
## 11.4.1 创建和使用自定义异常类

- 创建自定义异常类需要继承Exception类。在自定义的异常类中通过具有一个无参构造器和一个带有字符串参数的有参构造器。
- **【范例】** 示例代码是一个最简单的自定义异常类。
  - 示例代码class MyException extends Exception
- {
- public MyException()
- {
- }
- public MyException(String s)
- {
- super(s);
- }
- }
- **【代码解析】** 在该自定义异常类的程序中，让自定义的类继承Exception类，其中定义了一个无参构造器和一个需要字符串类型的有参构造器。这是一个最简单的自定义异常类。

北京源智天下科技有限公司

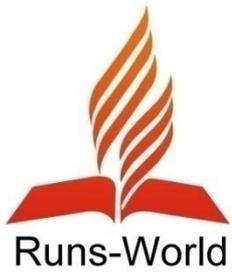


- 在Exception类中定义很多方法，这里讲解一些最常见的方法。使用printStackTrace方法可以显示异常调用栈的信息。使用toString方法可以得到异常对象的字符串表示。使用getMessage方法可以得到异常对象中携带的出错信息。在自定义的异常类中因为继承了Exception类，所以同时拥有这些方法。



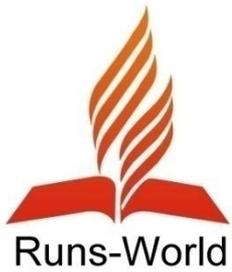
## 11.4.2 自定义异常的实际应用

- 在本节中来学习如何在实际应用中自定义异常。这里看一个非常简单的实际应用。在百分制的考试当中，0到100的得分都是可能发生的，但是超出这个范围，则肯定是发生了错误。这里就可以使用自定义异常，当使用不在0到100范围内的数的时候就会发生自定义异常。自定义异常类仍然还是采用上一小节中定义的自定义异常。
- 读者在学习前面示例代码时也许感觉不到使用自定义异常的优越性的，这是因为在实际应用中是不会将抛出异常的方法和调用方法写在一起的。



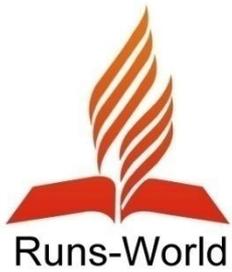
# 开发一个定义可能抛出异常方法的类。

```
• public class YiChang15
• {
• //定义一个可能发生自定义异常的方法
• public String deiFen(int fen)throws MyException
• {
• if(fen>=0&&fen<=100)
• {
• return "正常";
• }
• else
• {
• //当分数不在0到100的范围内时抛出自定义异常
• throw new MyException("错误输入");
• }
• }
• }
```



该方法定义后，其他人就可以进行使用了。先看一个不会发生自定义异常的程序。

```
• public class YiChang16
• {
• public static void main(String args[])
• {
• YiChang15 yc=new YiChang15();
• try
• {
• String s=yc.deiFen(68); //68在范围内，不会发生异常
• System.out.println(s);
• }
• catch(MyException e)
• {
• System.out.println("异常信息为: "+e.getMessage());
• }
• }
• }
```

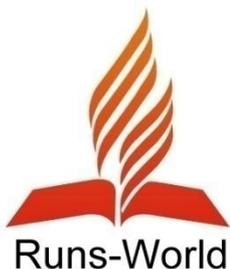


## 再看一个会发生自定义异常的程序。

```
• public class YiChang17
• {
• public static void main(String args[])
• {
• YiChang14 yc=new YiChang14();
• try
• {
• String ss=yc.deiFen(123); //123不在范围内，会发生异常
• System.out.println(ss);
• }
• catch(MyException e)
• {
• System.out.println("异常信息为: "+e.getMessage());
• }
• }
• }
```



- 上面的两个程序都调用了可能抛出自定义异常的类。该示例代码和上一个示例代码相比并没有更多的功能。但是读者会发现该程序更适合当有很多人同时使用自定义异常的时候。这种设计在实际的团队开发中经常使用，该团队为了完成自己需要的功能，定义了一个自定义异常类和抛出该异常类的方法。这些只需要一次编写，其他人就可以直接来使用该方法 and 自定义异常类。

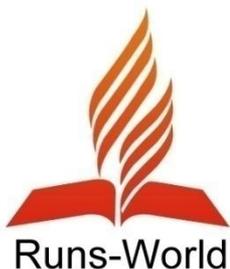


## 11.5 综合练习

- 1. 判断下面的程序是否能够正常运行。

```
• 01 import java.io.*;
• 02 class Fu
• 03 {
• 04 public void myVoid() throws IOException
• 05 {
• 06 System.out.println("父类");
• 07 }
• 08 }
• 09 public class LianXi1 extends Fu
• 10 {
• 11 public void myVoid() throws Exception
• 12 {
• 13 System.out.println("子类");
• 14 }
• 15 public static void main(String args[])
• 16 {
• 17 }
• 18 }
```

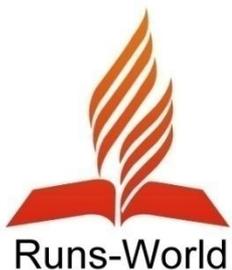
- **【提示】**编译该程序是会发生错误的，这是因为在子类中重写了父类中的myVoid方法，但是在父类中的方法使用throws抛出IOException异常，而子类的方法抛出Exception异常，这样违反了重写规则，所以是编译会发生错误的。



## 11.5 综合练习

- 2. 判断下面的程序是否能够正常运行。
- ```
01 import java.io.*;
02 class Fu
03 {
04     public void myVoid() throws IOException
05     {
06         System.out.println("父类");
07     }
08 }
09 public class LianXi2 extends Fu
10 {
11     public void myVoid(String s) throws Exception
12     {
13         System.out.println("子类");
14     }
15     public static void main(String args[])
16     {
17     }
18 }
```

• **【提示】**有些读者可能会认为该程序和上一个程序一样也会发生编译错误的，这里不是这样的，该程序能够正常编译。因为在该程序子类中的myVoid方法并不是重写的父类中的myVoid方法，而是重载的方法，从而不会发生编译错误。



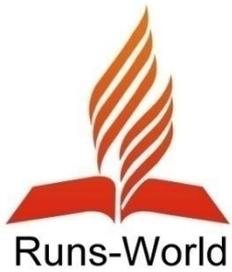
11.6 小结

- 在本章中主要对Java中的异常处理进行了详细的讲解。首先讲解了异常处理的基本结构，然后讲解了异常处理的分类，在本章的最后还对如何定义自己的异常进行了讲解。如果想了解更多的关于本章的内容，可以参考电子工业出版社出版的《深入浅出JDK 6.0》一书进行学习。



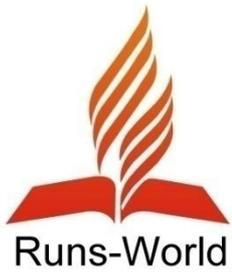
第12章 内部类

- 以前看到过这样一条新闻，说在一个鸡蛋中发现里面有一个小鸡蛋，这个小鸡蛋同样有蛋清和蛋黄，当时感觉很奇怪的。后来学习Java后，发现在Java中竟然也有这种奇怪的事，那就是内部类。内部类就好像刚提到的鸡蛋中的小鸡蛋一样，包含在另一个类中的。通过本章的学习，会发现内部类还有好多和该小鸡蛋相似的地方。读者通过本章的学习，应该完成如下几个目标。
- 了解什么是非静态内部类和如何进行非静态内部类和外部类之间的访问。
- 了解什么是局部内部类和如何进行局部内部类和外部类之间的访问。
- 了解什么是静态内部类和如何进行静态内部类和外部类之间的访问。
- 了解什么是匿名内部类和如何进行匿名内部类和外部类之间的访问。



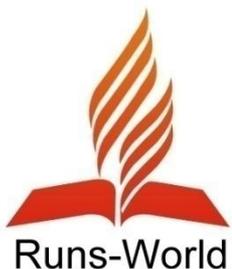
12.1 非静态内部类

- 当一个类作为另一个类的非静态成员，则这个类就是一个非静态内部类。在本节中就来学习如何创建和使用非静态内部类，同时也来讲解如何在内部类中访问外部类和在外类中如何访问内部类。



12.1.1 创建非静态内部类

- 创建非静态内部类是很容易的，只需要定义一个类让该类作为其他类的非静态成员。该非静态内部类和成员变量或者成员方法没有区别，同样可以在非静态内部类前面加可以修饰成员的修饰符。
 -
- 创建非静态内部类的基本语法如下所示。
- `class Wai`
- `{`
- `class Nei`
- `{`
- `//内部类成员`
- `}`
- `//外部类成员`
- `}`

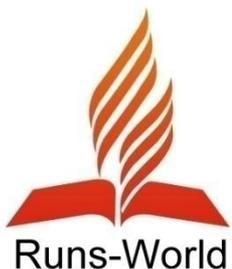


12.1.2 在外部类中访问内部类

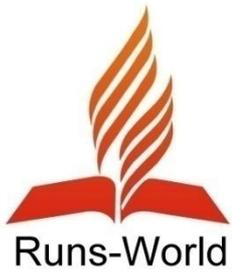
- 在内部类的程序中，经常会进行外部类和内部类之间访问的。在外部类中访问内部类是很容易的，只要把内部类看成一个类，然后创建该类的对象，使用对象来调用内部类中的成员就可以了。

- 【范例】示例代码是一个在外部类中访问内部类的程序。

```
01 class Wai
02 {
03     class Nei          //创建非静态内部类
04     {
05         int i=5;      //内部类成员
06     }
07     public void myVoid() //外部类成员
08     {
09         Nei n=new Nei(); //创建一个内部类对象
10         int ii=n.i;     //访问内部类成员
11         System.out.println("内部类的变量值为: "+ii);
12     }
13 }
14 public class NeiBuLei2
15 {
16     public static void main(String args[])
17     {
18         Wai w=new Wai();
19         w.myVoid();
20     }
21 }
```

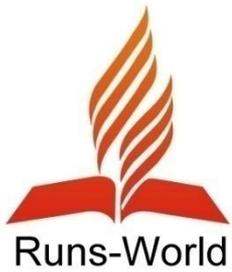


- 从程序的第**16**行主方法讲起，在**main**方法中，首先创建一个外部类对象，然后访问外部类的成员方法。在外部类的成员方法中，创建了一个内部类对象，然后使用内部类对象调用内部类的成员变量，从而得到结果。编译该程序将产生三个**class**文件，分别是主类、外部类和内部类。内部类产生的**class**文件的名称为**Wai\$Nei.class**，在该名称中可以区分该内部类到底是哪一个类的内部类。



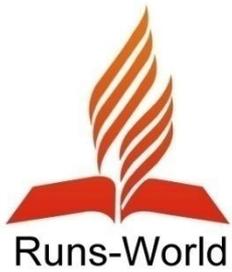
12.1.3 在外部类外访问内部类

- 不但可以在外部类中访问内部类，还可以在外部类外访问内部类。读者肯定会觉得非常难的，要想访问类成员中的成员怎么访问呢。其实在Java中，是很容易做到的。在外部类外访问内部类的基本语法如下所示。
- `Wai.Nei wn=new Wai().new Nei();`
- 使用该方法就能够创建一个内部类对象，使用该内部类对象就可以访问内部类的成员。该方法是不容易理解的，该方法也是可以分为两条语句的。
- `Wai w=new Wai();`
- `Wai.Nei wn=w.new Nei();`
- 这样就很容易理解了。首先是创建一个外部类的对象，然后让该外部类对象调用创建一个内部类对象。



一个在外部类外访问内部类的程序

```
• 01 class Wai
• 02 {
• 03     class Nei          //创建非静态内部类
• 04     {
• 05         int i=5;       //内部类成员
• 06         int ii=6;
• 07     }
• 08 }
• 09 public class NeiBuLei3
• 10 {
• 11     public static void main(String args[])
• 12     {
• 13         Wai.Nei wn1=new Wai().new Nei();
• 14         Wai w=new Wai();
• 15         Wai.Nei wn2=w.new Nei();
• 16         System.out.println("内部类中的变量i的值为: "+wn1.i);
• 17         System.out.println("内部类中的变量ii的值为: "+wn2.ii);图12-3 在外部类外访问内部类
• 18     }
• 19 }
```



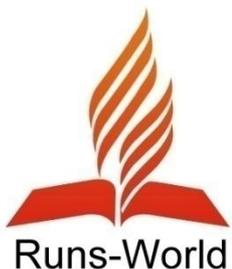
- 在示例代码中使用了两种方法来从外部类外访问内部类。在外部类外访问内部类时，是不能够直接创建内部类对象的，因为内部类只是外部类的一个成员。所以要想创建内部类对象，首先要创建外部类对象，然后以外部类对象为标识来创建内部类对象。



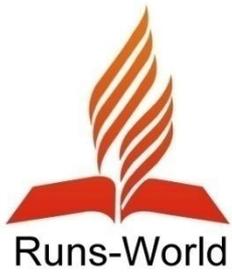
12.1.4 在内部类中访问外部类

- 在内部类中访问外部类，就像所有的同一个类中成员互相访问一样，这样是没有限制的，包括将成员声明为private私有的。
- 【范例】示例代码是一个在内部类中访问外部类的程序。
 - 示例代码

```
01 class Wai
02 {
03     int i=8;           //外部类成员变量
04     class Nei         //创建非静态内部类
05     {
06         public void myVoid() //内部类成员变量
07         {
08             System.out.println("外部类中的成员变量值为: "+i);
09         }
10     }
11 }
12 public class NeiBuLei5
13 {
14     public static void main(String args[])
15     {
16         Wai w=new Wai(); //创建外部类对象
17         Wai.Nei wn2=w.new Nei();//创建内部类对象
18         wn2.myVoid(); //调用内部类中成员
19     }
20 }
```

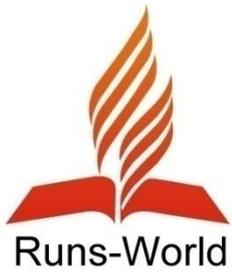


- 在示例代码中，在内部类中定义了一个myVoid来访问外部类中的成员变量i。可以看到从内部类中访问外部类是非常容易的，不需要添加任何内容，就像成员方法间调用一样。
- 有些读者学习完示例代码后，会有疑问，如果外部类中也有一个成员变量i怎么办呢？读者可以进行实验，从结果中可以看到得到的是内部类成员变量的值。下面通过示例代码解决这个问题。



一个在内部类和外部类中具有同名称变量访问的程序

```
• 01 class Wai
• 02 {
• 03     int i=8;           //外部类成员变量
• 04     class Nei       //创建非静态内部类
• 05     {
• 06         int i=9;
• 07         Wai ww=new Wai();
• 08         public void myVoid() //内部类成员变量
• 09         {
• 10             System.out.println("内部类中的成员变量值为: "+i);
• 11             System.out.println("外部类中的成员变量值为: "+ww.i);
• 12         }
• 13     }
• 14 }
• 15 public class NeiBuLei6
• 16 {
• 17     public static void main(String args[])
• 18     {
• 19         Wai w=new Wai(); //创建外部类对象
• 20         Wai.Nei wn2=w.new Nei(); //创建内部类对象
• 21         wn2.myVoid(); //调用内部类中成员
• 22     }
• 23 }
```

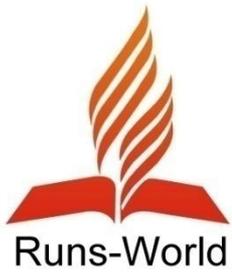


- 在本程序中的第**3**行定义了一个外部类的成员变量，第**6**行定义了一个内部类的成员变量，这两个成员变量的名称是相同的。而在内部直接访问时，将访问的是内部类的成员变量。要想访问外部类成员变量，就需要首先创建一个外部类对象，然后使用该对象调用外部类成员变量。



12.2 局部内部类

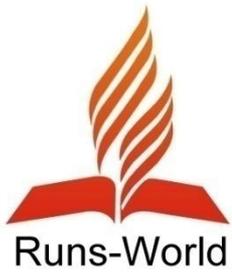
- 在上一节中介绍了非静态成员内部类，以及如何对非静态成员内部类进行操作。在本节中就来学习局部内部类的知识，通过非静态成员内部类的学习，是很容易来学习局部内部类的。从名称就可以看出局部内部类是作为一个类的局部变量来定义的。



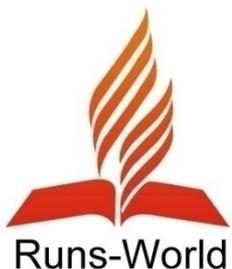
12.2.1 创建局部内部类

- 局部内部类的作用范围是和局部变量的作用范围相同的，只在局部中起作用，所以对局部内部类进行访问时，只能在该局部内部类的作用范围内。
- 【范例】示例代码是一个创建和访问局部内部类的程序。
 - 示例代码

```
01 class Wai
02 {
03     public void myVoid()
04     {
05         class Nei          //定义一个局部内部类
06         {
07             int i=5;        //局部内部类的成员变量
08         }
09         Nei n=new Nei();
10         System.out.println("局部内部类的成员变量为: "+n.i);
11     }
12 }
13 public class NeiBuLei8
14 {
15     public static void main(String args[])
16     {
17         Wai w=new Wai();    //创建外部类对象
18         w.myVoid();        //调用内部类中成员
19     }
20 }
```



- 在本程序中定义了一个局部内部类，并进行了对该局部内部类的访问。对该内部类进行访问必须在该内部类所在的方法中通过创建内部类对象来进行访问。这是因为这里的内部类是作为局部成员的形式出现的，只能在它所在的方法中进行调用。



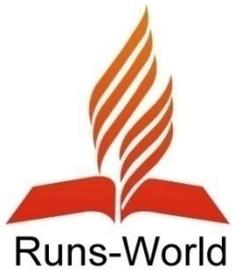
12.2.2 在局部内部类中访问外部类成员变量

- 在局部内部类中访问外部类成员变量是很容易实现的，并不需要进行过多操作。在局部内部类中可以直接调用外部类的成员变量。
- **【范例】** 示例代码是一个在局部内部类中访问外部类成员变量的程序。
 - 示例代码

```
01 class Wai
02 {
03     int i=9;           //定义一个外部类的成员变量
04     public void myVoid()
05     {
06         class Nei     //定义一个局部内部类
07         {
08             public void myNeiVoid()
09             {
10                 System.out.println("外部类的成员变量值为: "+i); //访问外部类的成员变量
11             }
12         }
13         Nei n=new Nei(); //创建内部类对象
14         n.myNeiVoid();   //调用内部类中的成员方法
15     }
16 }
17 public class NeiBuLei9
18 {
19     public static void main(String args[])
20     {
21         Wai w=new Wai(); //创建外部类对象
22         w.myVoid();      //调用内部类中成员
23     }
24 }
```



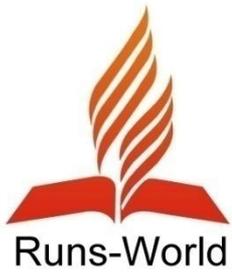
- 在示例代码中定义了一个局部内部类，在该局部内部类中定义了一个方法来访问外部类的成员变量。从运行结果中可以看出在内部类中可以成功访问外部类的成员变量。在该程序中同样需要注意的是，对内部类进行访问需要和内部类在同一方法中。



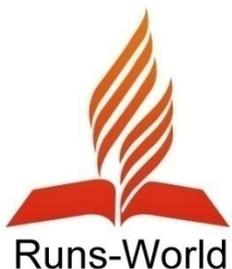
12.2.3 在局部内部类中访问外部类的局部变量

- 和访问外部类的成员变量不同，在局部内部类中访问外部类中和局部内部类在同一局部的局部变量是不能够直接访问的。
- **【范例】** 示例代码是一个错误的访问外部类局部变量的程序。
 - 示例代码

```
01 class Wai
02 {
03     public void myVoid()
04     {
05         int i=9;           //定义一个外部类的局部变量
06         class Nei         //定义一个局部内部类
07         {
08             public void myNeiVoid()
09             {
10                 System.out.println("外部类的局部变量值为: "+i); //访问外部类的成员变量
11             }
12         }
13         Nei n=new Nei();   //创建内部类对象
14         n.myNeiVoid();     //调用内部类中的成员方法
15     }
16 }
17 public class NeiBuLei10
18 {
19     public static void main(String args[])
20     {
21         Wai w=new Wai();   //创建外部类对象
22         w.myVoid();        //调用内部类中成员
23     }
24 }
```



- 运行该程序是会发生错误的，错误信息为“从内部类中访问局部变量*i*；需要被声明为最终类型”。在局部内部类中访问外部类的局部变量是不能够访问普通的局部变量的，必须将该局部变量声明为**final**。



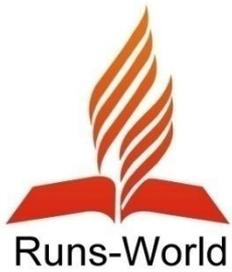
12.2.4 静态方法中的局部内部类

- 局部内部类定义在非静态方法和静态方法中是不同的，在前面的两小节中都是将局部内部类定义在非静态方法中，在本节中就学习静态方法中定义局部内部类的情况。在静态方法中定义的局部内部类要想访问外部类中的成员，该程序必须是静态成员。静态成员和非静态成员之间的访问是不变的。
- 注意：在静态方法中定义的局部内部类要想访问外部类中的成员，该程序必须是静态成员。静态成员和非静态成员之间的访问是不变的。

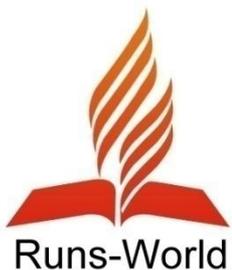


一个错误的访问成员的程序

```
• 01 class Wai
• 02 {
• 03     int i=3;
• 04     public static void myVoid()
• 05     {
• 06         class Nei          //定义一个局部内部类
• 07         {
• 08             public void myNeiVoid()
• 09             {
• 10                 System.out.println("外部类的局部变量值为: "+i);//访问外部类的成
                        //员变量
• 11             }
• 12         }
• 13         Nei n=new Nei();    //创建内部类对象
• 14         n.myNeiVoid();     //调用内部类中的成员方法
• 15     }
• 16 }
• 17 public class NeiBuLei12
• 18 {
• 19     public static void main(String args[])
• 20     {
• 21         Wai w=new Wai();    //创建外部类对象
• 22         w.myVoid();        //调用内部类中成员
• 23     }
• 24 }
```

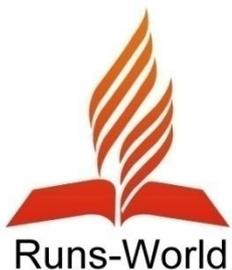


- 运行该程序是会发生错误的，错误信息为“无法从静态上下文中引用非静态变量i”。该程序主要错误原因是第三行定义的外部类变量是一个非静态成员变量。而本程序中定义的局部变量是定义在静态的方法中，所以是不能够正常访问的。如果想正常访问，就需要将程序修改成示例代码12-13的形式。



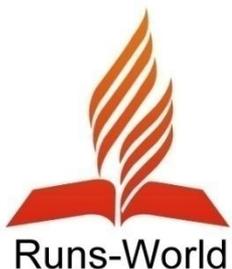
12.3 静态内部类

- 在第一节中已经讲解了非静态内部类，在本节中就来讲解什么是静态内部类。静态内部类就是在外部类中扮演一个静态成员的角色。在本节中就学习如何创建静态内部类和关于静态内部类访问的问题。



12.3.1 创建静态内部类

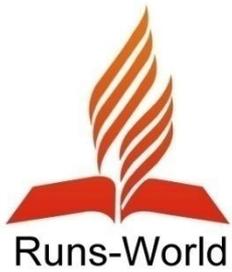
- 创建静态内部类的形式和创建非静态内部类的形式很相似的，只是需要将该内部类使用static修饰成静态的形式。使用static修饰类，这在正常类中是不可能的。定义静态内部类的语法如下所示。
- `class Wai`
- `{`
- `static class Nei`
- `{`
- `//内部类成员`
- `}`
- `//外部类成员`
- `}`



12.3.2 在外部类中访问静态内部类

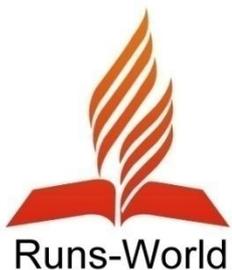
- 在外部类中访问静态内部类和在外类中访问非静态内部类一样的，只需要从成员间访问的角度就可以考虑到这一点。
- **【范例】** 示例代码是一个在外类中访问静态内部类的程序。
 - 示例代码

```
01 class Wai
02 {
03     static class Nei    //创建静态内部类
04     {
05         int i=5;        //内部类成员
06     }
07     public void myVoid() //外部类成员
08     {
09         Nei n=new Nei(); //创建一个内部类对象
10         int ii=n.i;      //访问内部类成员
11         System.out.println("静态内部类的变量值为: "+ii);
12     }
13 }
14 public class NeiBuLei15
15 {
16     public static void main(String args[])
17     {
18         Wai w=new Wai();
19         w.myVoid();
20     }
21 }
```



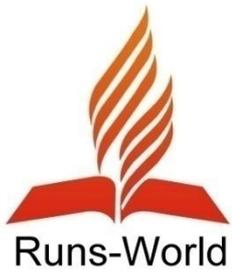
12.3.3 在外部类外访问静态内部类

- 通过上一小节的学习，知道在外部类中访问静态内部类和访问非静态内部类是相同的，但是在外部类中访问静态内部类和非静态内部类就不再相同。因为静态内部类是外部类的静态成员，静态成员是不需要外部类对象而存在的，所以在外部类外，对静态内部类进行访问时是不需要创建外部类对象的。
- 注意：因为静态内部类是外部类的静态成员，静态成员是不需要外部类对象而存在的，所以在外部类外，对静态内部类进行访问时是不需要创建外部类对象的。



12.4 匿名内部类

- 在所有的内部类中最难的就应该是匿名内部类。匿名内部类从名字上看就知道是没有类名的类。在本节中就来介绍如何创建匿名内部类和如何进行关于匿名内部类的访问问题。



12.4.1 创建匿名内部类

- 在创建匿名内部类中将使用到继承父类或者实现接口的知识，匿名内部类是没有名字的，所以在创建匿名内部类时同时创建匿名内部类的对象。创建匿名内部类的语法格式如下所示。
- `new NeiFather ()`
- `{`
- `//匿名内部类`
- `};`
- 在创建匿名内部类的语法中，`NeiFather`是匿名内部类继承的父类的类名，使用`new`同时创建了匿名内部类的对象。在匿名内部类中可以重写父类中的方法，也可以定义自己的方法。



12.4.2 匿名内部类的初始化

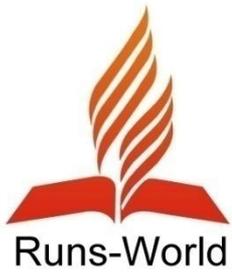
- 匿名内部类是没有名称的，所以匿名内部类也是不可能具有构造器的，这就出现一个问题。有时在匿名内部类中也是要定义成员变量的，但是该成员变量应该放在什么位置呢。这里的解决方法就是创建一个非静态语句块，将所有的初始化的成员变量都放在该非静态语句块中。这样在匿名内部类中的方法中就可以来调用这些成员变量。



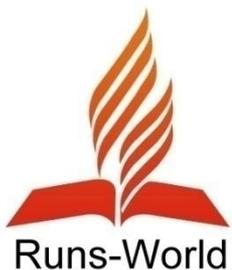
Runs-World

12.5 综合练习

- 在下面的程序中运行哪一条语句可以正常运行。
- 01 public class LianXi1
- 02 {
- 03 static int h=1;
- 04 private int i=2;
- 05 public void myVoid()
- 06 {
- 07 final int j=3;
- 08 int k=4;
- 09 class Nei
- 10 {
- 11 public void myNeiVoid()
- 12 {
- 13 //System.out.println(h);
- 14 //System.out.println(i);
- 15 //System.out.println(j);
- 16 //System.out.println(k);
- 17 }
- 18 }
- 19 Nei n=new Nei();
- 20 n.myNeiVoid();
- 21 }
- 22 public static void main(String args[])
- 23 {
- 24 LianXi1 lx=new LianXi1();
- 25 lx.myVoid();
- 26 }
- 27 }

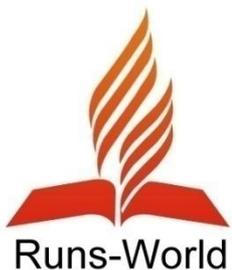


- 在该程序的第13行到第16行注释了4条访问程序中变量的语句，释放这些语句，可以发现访问h、i和j都是能够正常访问的，但是访问变量k就会发生编译异常。这是因为在局部内部类中，只能访问方法中的最终变量。



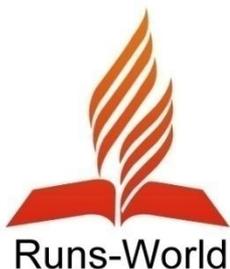
12.6 小结

- 在本章讲解了Java中的内部类，内部类包括非静态内部类、局部内部类、静态内部类和匿名内部类，并分别对这些内部类进行了讲解。在讲解每一种内部类时，首先讲解如何创建该内部类，并且讲解了如何对内部类进行访问。



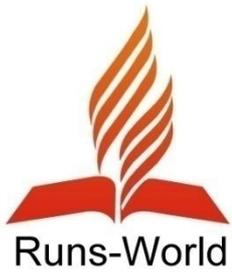
第13章 多线程

- 多线程是Java中的并发机制，表示能够在同一时间内同时执行多个操作。在日常生活中，边上网边听歌就是一个多线程。随着CPU进入双核，甚至多核时代，多线程的优势越来越明显。Java本身就是一门支持多线程的语言，在Java中使用多线程是很方便的，同样也是很高效的。通过本章的学习，读者应该能够完成如下几个目标。
- 了解什么是多线程。
- 熟练掌握如何定义和使用多线程。
- 了解多线程的生命周期。
- 掌握多线程的调用的几个情况。
- 了解多线程的同步问题。



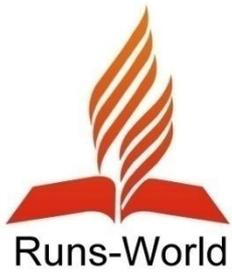
13.1 多线程简介

- 多线程就好像日常生活中同时做几件事一样，例如早上起床，要烧水洗脸，在烧水时就可以刷牙，还可以边刷牙边看早间新闻，这样就同时做着烧水、刷牙、看电视三件事。多线程也是一样的，在同一时刻有可能在执行多个线程，这样能够更好地提高办事效率。
- 在实际开发中也是在很多地方使用多线程的，例如在很多网站中，当用户注册后，系统一方面会通知用户已经注册成功，一方面向用户在注册时填写的Email中发送邮件。这里就需要使用多线程，如果使用的是单线程，系统就会向用户注册的Email中发送邮件后才显示用户注册成功，由于发送邮件可能需要很长的时间，从而影响整个注册进度。
- 在前面的学习中，虽然没有使用多线程，但是同样使用到了线程的知识。在每一个程序中的main方法就是一个线程，它一般被称为主线程。在主线程中可以启动多个子线程来执行。



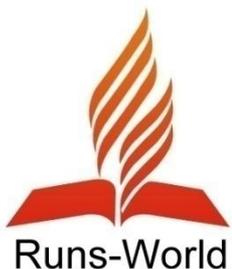
13.2 定义线程和创建线程对象

- 在上一小节中讲解了什么是多线程，在本节中就来讲解怎样来定义线程和如何创建线程对象。定义线程有两种方法，一种是继承Thread类，一种是实现Runnable接口，这两种方法是存在各自优缺点的。和定义线程对应的就是创建线程对象，也是有两种方法。在本节中就学习使用这两种方法来定义线程，以及相对应的创建线程对象。

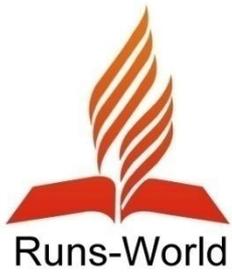


13.2.1 继承Thread类定义线程

- 定义一个线程可以通过继承Thread类来实现，这是一种相对简单的定义线程的方法。在Thread类中具有一个run方法，在定义的线程中需要重写这个方法。在重写的run方法中，可以定义该线程所要执行的语句。当线程启动时，run方法中的程序就成为一条独立的执行线程。
- **【范例】** 示例代码是一个通过继承Thread类定义线程的程序。
- 示例代码
- 01 public class XianCheng1 extends Thread
- 02 {
- 03 public void run()
- 04 {
- 05 System.out.println("通过继承Thread定义线程");
- 06 }
- 07 }



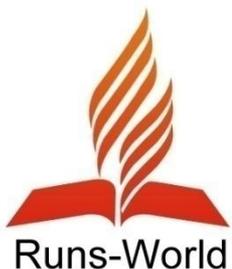
- 该程序是无法运行的，因为没有main方法，也就是没有启动线程的方法。在该程序中创建了一个线程类继承于Thread类，并且在该类中重写了run方法，在其中定义了该线程的功能是显示一条语句。
- 注意：重写的run方法也是可以作为一般的方法来调用的，但是这种调用并不是作为一个线程出现的，它只是主线程中的一部分。同样，run方法也是可以被重载的，但是重载后的run方法不作为一个线程，也是主线程的一部分。
- 讲解完定义线程后，就可以来学习如何创建线程对象。通过继承Thread类创建线程，是很容易创建线程对象的。在这种定义线程的方法中，创建线程对象和创建普通对象是一样的。下面是创建示例代码13-1中线程对象的代码。
- `XianCheng1 xc=new XianCheng1();`
- 从创建线程对象的程序可以看出，创建线程对象的方法和创建普通对象的方法是一样的。但是这只是对于使用继承Thread类创建线程的方法来说的。



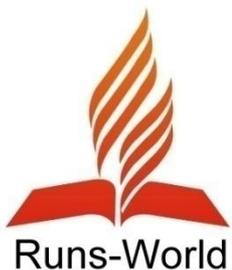
13.2.2 实现Runnable接口定义线程

- 定义线程除了通过继承Thread类来实现，还可以通过实现Runnable接口来实现。在Runnable接口中具有一个抽象的run方法，在实现Runnable接口时，需要实现该run方法。该run方法就会作为一个执行线程的方法。
- **【范例】** 示例代码是一个通过实现Runnable接口定义线程的程序。
 - 示例代码

```
01 public class XianCheng2 implements Runnable
02 {
03     public void run()
04     {
05         System.out.println("通过实现Runnable接口定义线程");
06     }
07 }
```

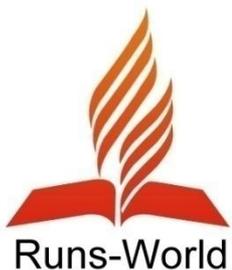


- 1是通过继承**Thread**类定义线程，2是通过实现**Runnable**接口来定义线程。这两种方法中都需要定义一个**run**方法，不管该方法是通过重写父类方法，还是实现接口方法。**run**方法是一个线程的入口，是线程必须具有的。
 -
- 在使用通过实现**Runnable**接口定义的线程中，要想创建线程对象就不是很容易做到的。因为直接创建类对象，创建的并不是一个线程对象。要想创建线程对象，必须要借助**Thread**类。
- **Thread**类具有4个构造器，最常用的就是具有一个参数，该参数是实现**Runnable**接口类对象的构造器。创建线程对象的程序如下所示。
- `XianCheng2 xc=new XianCheng2();`
- `Thread t=new Thread(xc);`
- 在该程序中，首先创建了一个实现**Runnable**接口的类对象，然后将该对象作为**Thread**类的参数，从而创建了一个线程对象。创建的类对象是可以作为多个**Thread**类构造器参数的，这样就创建了多个线程。这一点将在以后的学习中多次使用。



13.3 运行线程

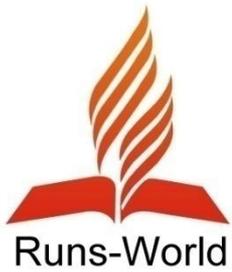
- 在上一节中学习了如何定义线程，并且知道了如何创建线程对象。对这些都了解后就需要来学习如何运行线程。在本节中分为两个小节来讲解，先来学习如何启动线程，然后讲解如何运行多个线程。



13.3.1 启动线程

- 有些读者会认为启动线程就是调用线程类中的run方法。例如示例代码13-3中所演示的。
- **【范例】** 示例代码是一个错误的启动线程的程序。
 - 示例代码

```
01 class MyRunnable implements Runnable
02 {
03     //定义一个run线程方法
04     public void run()
05     {
06         System.out.println("这是一个错误的启动线程的程序");
07     }
08 }
09 public class XianCheng3
10 {
11     public static void main(String args[])
12     {
13         MyRunnable mr=new MyRunnable();
14         mr.run(); //调用run方法
15     }
16 }
```

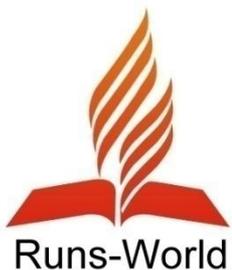


- 从该程序可以看出，**run**方法是可以通过方法调用来执行的，但是这并不代表创建了一个新线程。这是一个错误的启动线程的方法。
- 如果想正确地启动一个线程，需要调用线程对象的**start**方法，下面通过程序来演示如何正确的启动一个线程。



一个正确的启动线程的程序

```
• 01 class MyRunnable implements Runnable
• 02 {
• 03     //定义一个run线程方法
• 04     public void run()
• 05     {
• 06         System.out.println("这是一个正确的启动线程的程序");
• 07     }
• 08 }
• 09 public class XianCheng4
• 10 {
• 11     public static void main(String args[])
• 12     {
• 13         MyRunnable mr=new MyRunnable();
• 14         Thread t=new Thread(mr);
• 15         t.start (); //启动线程
• 16     }
• 17 }
```

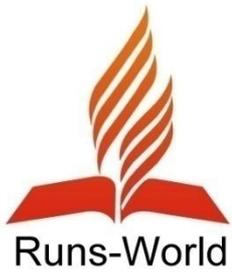


- 【代码解析】第一次看到该程序时，读者可能会感到有些奇怪，为什么调用的是**start**方法，而执行的是**run**方法，这就是Java对多线程的设计。在调用**start**方法后，就启动了线程，该线程是和**main**方法并列执行的线程。这样该程序就变为一个多线程程序。
- 注意：线程只能被启动一次，也就是只能调用一次**start**方法。当多次启动线程，也就是多次调用**start**方法时，就会发生异常。



13.3.2 同时运行多个线程

- 学习了如何启动线程，接下来就来学习如何同时运行多个线程。首先通过示例代码来看一下如何同时运行多个线程。
- 27 `public static void main(String args[])`
- 28 `{`
- 29 `MyRunnable1 mr1=new MyRunnable1();`
- 30 `MyRunnable2 mr2=new MyRunnable2();`
- 31 `Thread t1=new Thread(mr1);`
- 32 `Thread t2=new Thread(mr2);`
- 33 `t1.start(); //启动第一个线程`
- 34 `t2.start(); //启动第二个线程`
- 35 `}`
- 36 `}`

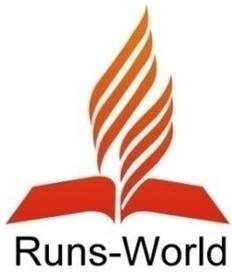


- **【代码解析】** 在示例代码13-6中首先定义了两个实现**Runnable**接口的类，在两个类中都定义了**run**方法，显示多个不同的符号。从运行结果中可以看出，不同的符号是交替显示的。
- 在同时运行多个线程时，运行结果不是唯一的，因为有很多不确定的因素。首先先执行哪一个线程就是不确定的，线程间交替也是不确定的。但是确定的是每一个线程都将启动，每一个线程都执行结束。



13.4 线程生命周期

- 线程是存在生命周期的。线程的生命周期分为五种不同的状态，分别是新建状态、准备状态、运行状态、等待/阻塞状态和死亡状态。在本节中就来对每一个状态进行讲解。



13.4.1 新建状态

- 当一个线程对象被创建后，线程就处于新建状态。在新建状态中的线程对象从严格意义上看还只是一个普通的对象，它还不是一个独立的线程。处于新建状态中的线程被调用start方法后就会进入准备状态。从新建状态中只能进入准备状态，并且不能从其他状态进行新建状态。新建状态是线程生命周期的第一个状态。



13.4.2 准备状态

- 处于新建状态中的线程被调用start方法就会进入准备状态。处于准备状态下的线程随时都可能被系统选择进入运行状态，从而执行线程。可能同时有多个线程处于准备状态，对于哪一个线程将进入运行状态是不确定的。线程从新建状态进入到准备状态后是不可能再进入新建状态的。在等待/阻塞状态中的线程被解除等待和阻塞后将不直接进入运行状态，而是首先进入准备状态，让系统来选择哪一个线程进入运行状态。



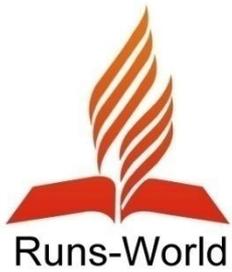
13.4.3 运行状态

- 处于准备状态中的线程一旦被系统选中，使线程获取了CPU时间，就会进入运行状态。在运行状态中将执行线程类run方法中的程序语句。线程进入运行状态后也不是一下执行结束的，线程在运行状态下随时都可能被调度程序调度回准备状态。在运行状态下还可以让线程进入到等待/阻塞状态。在通常的单核CPU中，在同一时刻只有一个线程处于运行状态的。在多核的CPU中，就可能两个线程或者更多的线程同时处于运行状态，这也是多核CPU运行速度快的原因。



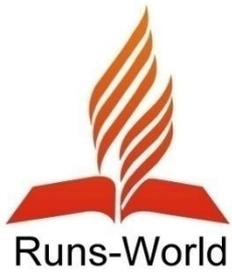
13.4.4 等待/阻塞状态

- 在Java中定义了许多线程调度的方法，包括睡眠、阻塞、挂起和等待，这些方法将在后面的调度章节中讲解。使用这些方法都会将处于运行状态的线程调度到等待/阻塞状态。处于等待/阻塞状态的线程被解除后，不会立即回到运行状态，而是首先进入准备状态，等待系统的调度。



13.4.5 死亡状态

- 当线程中的run方法执行结束后，或者程序发生异常终止运行后，线程会进入死亡状态。处于死亡状态的线程不能再使用start方法启动线程，这在前面的学习中已经学到了这一点。但是这不代表处于死亡状态的线程不能再被使用，它也是可以再被使用的，只是将被作为普通的类来使用。
- 注意：线程生命周期的问题，有些读者会觉得很容易的。线程生命周期的问题在后面的学习中会经常使用到，只有能充分了解线程的生命周期，才能更好地理解后面的内容。

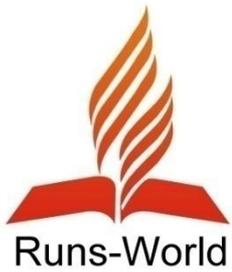


13.5 线程的调度

- 通过系统自动调度，线程的执行顺序是没有保障的。在Java中定义了一些线程调度的方法，使用这些方法在一定程序上对线程进行调度，使用这些方法只是给线程一个建议，具体是否能够成功，也是没有保障的。线程调度的方法有几个，包括睡眠方法、设置优先级、让步方法等，在本节中就来学习这些方法的使用。

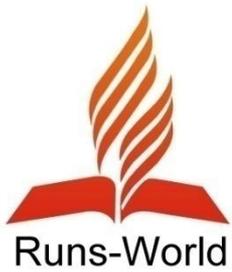


- 使用这两个**sleep**方法都能使线程进入睡眠状态，**mills**参数表示线程睡眠的毫秒数，**nanos**参数表示线程睡眠的纳秒数。**sleep**方法是一个静态的方法，所以**sleep**方法不是依赖于某一个对象的，它的位置是比较随意的。当在线程中执行**sleep**方法，则该线程就进入睡眠状态。要想让某一个线程进入睡眠状态，并不是让该线程调用**sleep**方法，而只是让该线程执行**sleep**方法。**sleep**方法是可能发生捕获异常的，所以在使用**sleep**方法时必须进行异常处理。
- 注意：**sleep**方法只是给线程一个调度的建议，是否调度成功是不能确定的。在该程序中只有两个线程，所以运行结果出现交替显示的结果。当程序中存在多个线程时，运行结果就可以发生变化，甚至出现意外的结果。



13.5.2 线程优先级

- 在大部分的系统中，对进程的调度都是采用优先级的方式来进行的。在Java中对线程进行调度时，也是可以采用优先级来调度的。不同的线程可以具有不同的优先级，优先级高的线程就会占用更多的CPU资源和被执行概率。
- Java中的优先级是采用从1到10来表示的，数字越大表示优先级越高。如果没有为线程设置优先级，则线程的优先级为5，这也是线程的默认值。但是对于子线程来说，它的优先级是和其父线程优先级相同的。
- 当需要对线程的优先级进行设置时，可以通过调用**setPriority**方法来设置。**setPriority**方法的语法格式如下所示。
- **public final void setPriority(int i);**
- 其中参数*i*表示的就是优先级的等级数，它可以从1到10。除了可以使用数字来表示优先级，Java还在**Thread**类中定义了三个表示优先级的常量。**MAX_PRIORITY**表示线程的最高优先级，**NORM_PRIORITY**表示线程的默认优先级，**MIN_PRIORITY**表示线程的最低优先级。



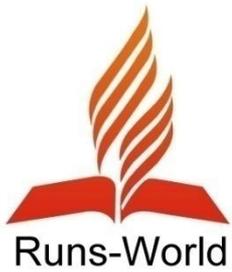
13.5.3 yield让步方法

- 在Java中具有两种线程让步方法，先来介绍第一种yield让步方法。yield让步方法是让线程让出当前CPU，而将CPU让给哪一个线程是不确定的，由系统来进行选择。使用yield让步方法的线程将从运行状态进入到准备状态。
- 注意：yield让步操作是可能不成功的。因为在线程中使用yield方法，使该线程进入准备状态。但是系统是有可能再次选择该线程，使该线程进入运行状态的。
- yield让步方法的基本语法格式如下所示。
- `public static void yield();`
- 可以看出**yield**让步方法是一个静态方法，所以该方法也是和对象无关的。当在正在运行的线程中运行该方法时，该线程将回到准备状态。



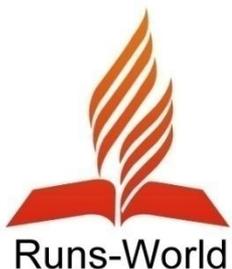
13.5.4 join让步方法

- 使用join让步方法，可以将当前线程的CPU资源让步给指定的线程。join让步方法的语法格式如下所示。
- `public final void join() throws InterruptedException;`
- `public final void join(long mills) throws InterruptedException;`
- `public final void join(long millis, int nanos) throws InterruptedException;`
- join让步方法是具有三种形式的，没有参数表示指定的线程执行完成后再执行其他线程，参数表示在参数的时间内执行让步给的执行线程。join让步方法也是可能发生捕获异常的，所以在使用时要进行异常处理。



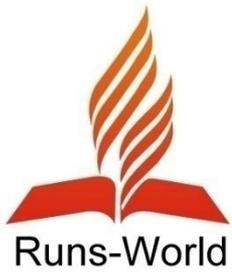
13.6 综合练习

- 1. 判断下面程序是否能够正常运行。
- 01 public class LianXi1 extends Thread
- 02 {
- 03 public void run()
- 04 {
- 05 for(int i=0;i<5;i++)
- 06 {
- 07 System.out.println(i);
- 08 }
- 09 }
- 10 public static void main(String args[])
- 11 {
- 12 LianXi1 lx=new LianXi1();
- 13 lx.run();
- 14 }
- 15 }



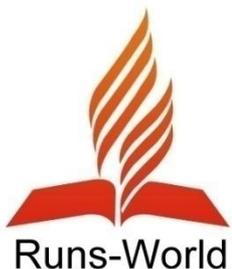
13.6 综合练习

- 2. 判断下面的程序是否能够正常运行。
- 01 public class LianXi2 extends Thread
- 02 {
- 03 public void run()
- 04 {
- 05 for(int i=0;i<5;i++)
- 06 {
- 07 Thread.sleep(100);
- 08 }
- 09 }
- 10 public static void main(String args[])
- 11 {
- 12 LianXi1 lx=new LianXi1();
- 13 lx.run();
- 14 }
- 15 }



13.7 小结

- 在本章中学习了Java中的多线程，首先对线程进行了简单的介绍，然后讲解如何定义、创建和运行多线程。接下来还讲解了线程的生命周期和对线程的调度。



第14章 Swing桌面程序开发

- Swing是一门开发桌面程序的技术。在本章中读者将学到如何开发界面程序，这要比前面学习的程序有意思的多。在本章中将对Swing的知识按从浅到深的顺序依次进行讲解。读者通过本章的学习，应该完成如下几个目标。
- 了解Swing开发的基本过程。
- 掌握如何创建窗口、面板、标签和按钮。
- 掌握和熟练使用Swing中的事件。

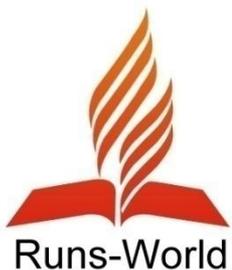


14.1 开发第一个Swing程序

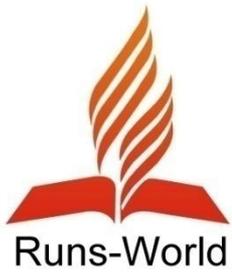
- 在本节中首先使用一个简单的程序，让读者知道什么是Swing程序，以及Swing程序的功能。在该程序中有很多知识是以前没有介绍过的，在后面的学习中将详细的分析各个地方。
- **【范例】** 示例代码是一个简单的Swing程序。

- 示例代码

```
01 import javax.swing.*;           //导入Swing包
02 //继承JFrame类
03 public class Swing1 extends JFrame
04 {
05     //定义构造器
06     public Swing1()
07     {
08         this.setLayout(null);      //设置布局管理器
09         JLabel jl=new JLabel();    //定义一个标签
10         jl.setText("第一个Swing程序"); //设置显示的文字
11         jl.setBounds(50, 50, 400, 50); //设置标签的大小和位置
12         this.add(jl);              //将标签放到窗口中
13         this.setBounds(300, 250, 500, 200); //设置窗口的大小和位置
14         this.setVisible(true);    //设置窗口是可见的
15     }
16     public static void main(String args[])
17     {
18         Swing1 s=new Swing1();
19     }
20 }
```



- 第一次看到该程序可能会觉得很复杂，其实其中都是很基础的内容，在以后的**Swing**程序中也会重复使用。在该程序中，首先要导入**Swing**包，然后继承该包中的**JFrame**类，使用该类才能使运行结果出现界面的形式。在程序中需要定义一个构造器，在构造器中首先要设置布局管理器，该程序没有使用布局管理器，布局管理器的知识会在后面用一章的内容来进行讲解。然后就是定义了一个用于显示文字的标签。在最后还需要设置窗口的大小和位置，以及可见性。
- 从第一个**Swing**程序可以看出，运行结果不再是以前在黑屏中显示信息，而是在界面中显示信息。这里的信息不只包括文字信息，也包括以后将要学到的一些组件信息。



14.2 JFrame窗口类

- 在Swing程序中，窗口是一个容器，在该容器中可以放其他一些组件。学习JFrame窗口类是学习其他组件的基础。在Swing程序中创建窗口可以使用继承JFrame类来完成。

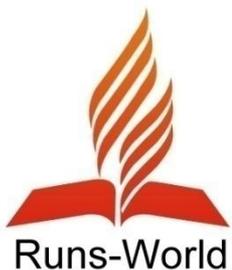


14.2.1 JFrame窗口类简介

- 在开发的Swing程序中，通常是通过继承JFrame窗口类来实现窗口的。在该类中具有很多很有用的方法，包括定义窗口标题、标框，以及窗口的大小和位置。在介绍这些方法之前，先来介绍一下JFrame窗口类的构造器。JFrame窗口类具有四种构造器。
- 最常用的JFrame窗口类的构造器是无参构造器，使用该构造器将创建一个初始不可见的新窗体。除此之外还有具有一个String类参数的构造器，使用该构造器能够在初始时就创建一个具有标题的新窗体。还有两种需要给出图形配置参数的构造器，这两种构造器在本书中不进行介绍。
- 创建新窗口后，就可以通过JFrame窗口类的方法来设置新窗口。首先使用无参构造器创建的是一个不可见的新窗体，所以要使用方法来将窗体设置为可见的形式。在JFrame窗口类中定义了一个setVisible方法来设置窗口的可见性，该方法具有一个布尔型参数，true表示可见，false表示不可见。将初始状态下的窗口设置为不可见是有原因的，因为有很多对窗口的操作需要在窗口不可见的状态下执行，从而setVisible方法通过在程序的最后执行。



- 在JFrame窗口类中有个setTitle方法，该方法需要一个字符型参数。使用setTitle方法可以设置窗口的名称；还有一个setBounds方法，该方法具有4个参数，前两个参数分别表示窗口位置的横坐标和纵坐标，后两个参数分别表示窗口大小的宽度和高度。JFrame窗口类最重要的方法就是add方法，使用该方法可以将组件添加到窗口中。这些都是比较常用的JFrame窗口类的方法。

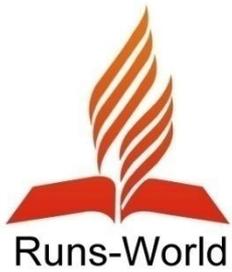


14.2.2 创建简单窗体

- 通过前面对JFrame窗口类的学习，可以来创建一个简单的窗体。创建窗体有两种方法，先来介绍第一种方法，可以直接使用JFrame窗口类的构造器来创建一个简单的窗体。
- **【范例】** 示例代码是一个直接使用JFrame窗口类创建简单窗体的程序。

- 示例代码

```
• 01 import javax.swing.*;           //导入Swing包
• 02 public class Swing2
• 03 {
• 04     public static void main(String args[])
• 05     {
• 06         JFrame jf=new JFrame();    //创建JFrame类构造器
• 07         jf.setTitle("直接使用JFrame窗口类");//设置窗口的名称
• 08         jf.setBounds(300,250,300,200); //设置窗口的大小和位置
• 09         jf.setVisible(true);       //设置窗口可见性
• 10     }
• 11 }
```



- 在本程序中直接使用**JFrame**窗口类来创建一个窗体。首先创建一个**JFrame**类对象，然后调用**JFrame**类中的方法。在本程序中使用**setTitle**方法来设置窗口的名称，使用**setBounds**方法来设置窗口的大小和位置，使用**setVisible**方法来设置窗口的可见性。这种方法只是创建一个简单窗口时需要的，如果创建一个复杂的窗口，使用这种方法就会使程序变的非常复杂难理解。

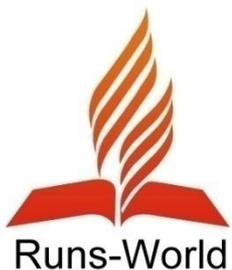


14.2.3 设置窗体

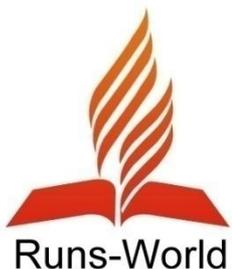
- 除了上一小节中学习的在创建窗体时必要的设置窗体的方法外，还有一些设置窗体的方法。例如setResizable方法，使用该方法可以设置创建的窗口是否可以调整大小。默认状态下窗体是能够调整大小的，也就是setResizable方法的默认值为true。
- 注意：默认状态下窗体是能够调整大小的，也就是setResizable方法的默认值为true。
- **【范例】** 示例代码是一个设置窗体不能被调整大小的程序。

- 示例代码

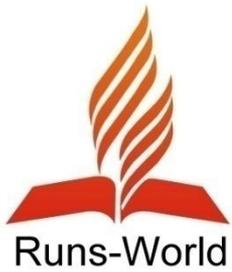
```
01 import javax.swing.*;           //导入Swing包
02 //继承JFrame类
03 public class Swing4 extends JFrame
04 {
05     //定义构造器
06     public Swing4()
07     {
08         this.setTitle("通过继承创建窗口");//设置窗口标题
09         this.setBounds(300,250,300,200); //设置窗口的大小和位置
10         this.setResizable(false);      //设置窗口不能被调整大小
11         this.setVisible(true);         //设置窗口是可见的
12     }
13     public static void main(String args[])
14     {
15         Swing4 s=new Swing4();
16     }
17 }
```



- 有些读者会认为该结果是和上面程序的运行结果图一样的，其实不然。该运行结果中的窗体是不能最大化的，这主要是由于在程序中使用**setResizable**方法的参数为**false**，使得窗口不能调整大小，从而也就使窗口不能最大化。如果试图通过鼠标来调整大小，也是不能成功的。
- 提示：在**JFrame**窗口类中还有一个**setUndecorated**方法，使用该方法可以将窗体的边框和标题栏去掉。

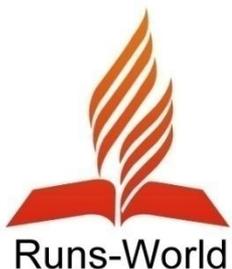


- 在JFrame窗口类中还有一个很重要的方法，那就是 `setDefaultCloseOperation` 方法。使用该方法可以设置当单击关闭按钮关闭窗口时所执行的动作。这里的动作包括4种情况，分别对应着4个常量。
- `DO_NOTHING_ON_CLOSE` 不执行任何动作
- `DISPOSE_ON_CLOSE` 释放窗体对象
- `HIDE_ON_CLOSE` 隐藏窗体
- `EXIT_ON_CLOSE` 退出JVM
- 提示：如果不使用 `setDefaultCloseOperation` 方法进行设置，默认值为 `HIDE_ON_CLOSE`，也就是在默认情况下单击关闭按钮将会使窗口隐藏。



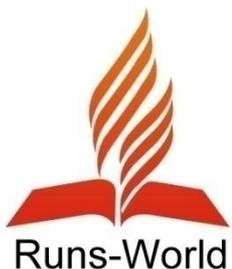
14.3 JPanel1面板类

- 在上一节中讲解的JFrame窗口类是一个容器类，从本小节开始来讲解一些控件。首先要讲解的就是JPanel面板类。面板可以说是控件，但它同样是一种容器，只不过它不是顶层容器。所以在本节中就要先了解一下什么是容器，然后再介绍JPanel面板类。

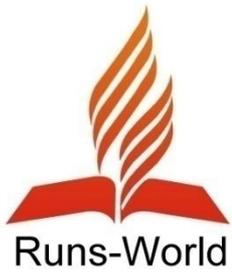


14.3.1 容器介绍

- Swing中的控件可以分为三类，顶层容器、非顶层容器和普通控件。在前面介绍的JFrame窗口类就是一个顶层容器。顶层容器是一种可以直接显示在系统桌面上的控件，其他控件必须直接或者间接的借助顶层容器进行显示。顶层容器除了包括JFrame窗口类外，还包括JWindow和JDialog等不常用的类。
- 在本节中将介绍的JPanel面板类是一个非顶级容器，非顶级容器是具有两面性的。非顶级容器是要放到顶级容器中使用的，对于顶级容器来说，非顶级容器是一般控件。在非顶级容器中还可以添加控件，对于这些控件来看，非顶级容器就是一个容器。
- 普通控件在控件中占大部分，使用这些控件可以实现特定的功能，但它们不具有容器的作用，它们只能放在容器中进行显示。普通控件包括按钮、文本框等很多控件。

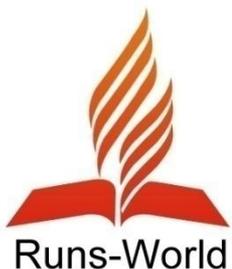


- 有些读者会认为将普通控件直接放到顶级容器中不也可以完成显示功能吗？这种说法在语法上是正确的，但是正确不一定是合理的。这种设计将会使程序变的非常复杂，而且难以维护。通常在设计界面时，都会先定义顶级容器，然后向顶级容器中添加非顶级容器，而将普通控件放在非顶级容器中。
- 这种设计的好处就是使程序开发变得简单，在开发时开发员在某一时间内只需要关注某一个非顶级容器界面的编写，最后将所有的非顶级容器添加到顶级容器中。这种设计还有一个好处就是，程序具有重用性，因为有可能在多个界面中使用同一个非顶级容器程序，这样就可以重复使用该程序。



14.3.2 JPanel面板类简介

- JPanel面板类是一个非顶级容器，使用JPanel面板类可以搭建一个子界面。JPanel面板类同样具有四种构造器，最常用的仍然是无参构造器。使用有参构造器可以在初始时设置面板采用什么布局管理器和是否使用双缓冲。
- JPanel面板类本身没有特殊功能，它的作用就是作为非顶级容器来添加普通控件，搭建子界面。所以JPanel面板类的方法也是很少很简单的。首先JPanel面板类具有一个添加控件的add方法，使用该方法能够将普通控件添加到面板中。getHeight方法和getWidth方法分别是返回当前面板的高度和宽度。
- 提示：JPanel面板类还有一个setToolTipText方法，该方法具有一个字符串参数，该方法的作用主要是当鼠标指针停留在面板上时显示文本，字符串内容就是要显示的内容。

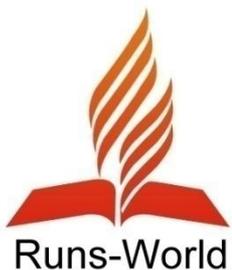


14.3.3 创建面板

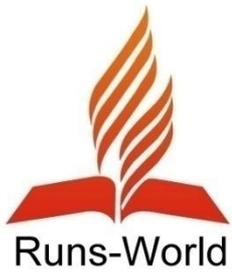
- 在前面的学习中已经知道，面板必须添加到窗口中，而面板中还可以添加普通的控件。在本节中就来学习如何创建面板，和如何进行添加操作。
- **【范例】** 示例代码是一个创建面板的程序。

- 示例代码

```
01 import javax.swing.*;           //导入Swing包
02 //继承JFrame类
03 public class Swing7 extends JFrame
04 {
05     JPanel jp=new JPanel();       //创建一个面板
06     JButton jb=new JButton("按钮"); //创建一个按钮
07     //定义构造器
08     public Swing7()
09     {
10         this.setTitle("创建面板"); //设置窗口名称
11         jp.add(jb);                //将按钮添加到面板中
12         this.add(jp);              //将面板添加到窗口中
13         this.setBounds(300,250,300,200); //设置窗口的大小和位置
14         this.setVisible(true);    //设置窗口是可见的
15     }
16     public static void main(String args[])
17     {
18         Swing7 s=new Swing7();
19     }
20 }
```



- 在本程序中是一个创建面板的程序。在示例代码14-7中的第5行创建了一个面板，在第12行是让窗体调用add方法将该面板添加到窗体中。在第6行是创建的一个按钮，在第11行将该按钮添加到面板中。有些读者可能会有疑问了，在运行结果中只有一个按钮，怎么没有看到面板。这是因为面板不是普通的控件，它是一个放置控件的容器，所以它是不显示的。



14.4 JLabel 标签类

- 标签是Swing中最基本的控件，它是一种非交互的控件，也就是不需要进行操作的控件。标签虽然通常只起到一个显示功能，但是它是界面编程中必不可少的。使用标签能够给用户提供更多的相关信息。



14.4.1 JLabel 标签类简介

- JLabel 标签类的知识点要比前面所学到的 JPanel 面板类的知识点多很多。首先 JLabel 标签类具有六个构造器来创建标签，在表 14-2 中列出了这六种构造器。
- 使用表中的标签类构造器都能够创建标签，其中最常见的是无参构造器。

标签类构造器

标签构造器	说明
<code>public void JLabel()</code>	创建没有图象和标题的标签
<code>public void JLabel(Icon image)</code>	创建具有图象的标签
<code>public void JLabel(Icon image,int horizontalAlignment)</code>	创建具有图象和指定对齐方式的标签
<code>public void JLabel(String text)</code>	创建指定文本的标签
<code>public void JLabel(String text,int horizontalAlignment)</code>	创建指定文本和对齐方式的标签
<code>public void JLabel(String text, Icon image, int horizontalAlignment)</code>	创建指定文本、图象和对齐方式的标签



- **JLabel**标签类的方法有很多，这些方法都是对应的形式，分别是获取和设置方法。这里给出一些比较常用的方法，其中**setText**方法已经在前面的学习中使用过，表示设置标签要显示的文本。同时和这个方法相对应的就是**getText**方法，使用该方法来获取标签显示的文本。除了这两个方法外，还有对图像、对齐方式等进行操作的方法，这些在以后的学习中使用时将进行讲解。

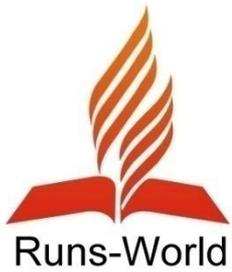


14.4.2 创建标签

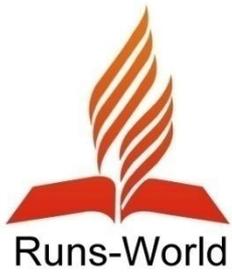
- 学习完了JLabel标签类后，创建标签就是很容易的问题。示例代码14-8就是一个创建简单标签的程序。
- **【范例】** 示例代码是一个创建标签的程序。

- 示例代码

```
01 import javax.swing.*;           //导入Swing包
02 //继承JFrame类
03 public class Swing8 extends JFrame
04 {
05     JLabel jl=new JLabel();       //创建一个标签
06     //定义构造器
07     public Swing8()
08     {
09         this.setTitle("创建标签"); //设置窗口名称
10         jl.setText("这是一个标签"); //设置标签显示的内容
11         jl.setVerticalAlignment(JLabel.CENTER); //设置标签垂直居中
12         jl.setHorizontalAlignment(JLabel.CENTER); //设置标签水平居中
13         this.add(jl);             //将标签添加到窗口中
14         this.setBounds(300, 250, 300, 200); //设置窗口的大小和位置
15         this.setVisible(true);    //设置窗口是可见的
16     }
17     public static void main(String args[])
18     {
19         Swing8 s=new Swing8();
20     }
21 }
```

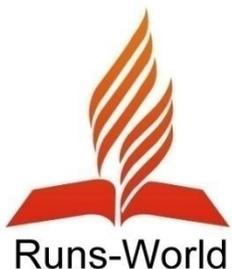


- 其中第5行使用JLabel标签类的无参构造器创建了一个标签。在第10行使用setText方法设置标签上要显示的内容。在第11行使用setVerticalAlignment方法设置标签在容器中垂直居中。在第12行使用setHorizontalAlignment方法设置标签在容器中水平居中。在第13行将该标签添加到窗口中。在该程序中为了使程序简单，就直接将标签放在窗体中，而没有再定义非顶级容器。



14.5 JButton按钮类

- 为了更好地学习下一章的布局管理器，在本章中也介绍一个Swing中最常见的控件，那就是按钮。按钮是进行交互操作使用最多的控件，同时按钮也是相对简单的控件。在下一章中学习布局管理器时，将使用按钮来进行举例说明，所以该节也是学习布局管理器的基础。



14.5.1 JButton按钮类简介

- 使用JButton按钮类可以创建最常用的按钮控件。JButton按钮类同样具有多个构造器，使用这些构造器都能够创建按钮控件。最常用的仍然是使用无参构造器来创建一个不带文本和图标的按钮。
- 在JButton按钮类中具有几个很常用的方法。其中setText方法是来设置按钮上显示的文本，和其对应的是用getText方法来获取按钮上显示的文本。在JButton按钮类中还有一个经常被使用，也是非常有意思的setMnemonic方法，使用该方法可以设置按钮的助记符。助记符就是使用键盘中的Alt加该助记符就能起到相应的功能。例如在Word中，使用Alt+F就能打开文件菜单。为按钮添加助记符后就可以使用Alt加该助记符来代替单击按钮的操作。
- 提示：助记符就是使用键盘中的Alt加该助记符就能起到相应的功能。



Runs-World

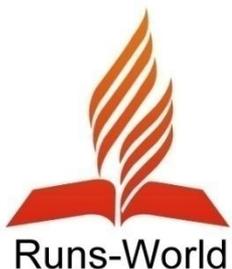
14.5.2 创建按钮

- 学习完了JButton按钮类后，创建按钮就是很容易的问题。示例代码14-9就是一个创建简单按钮的程序。

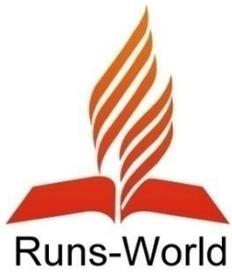
- **【范例】** 示例代码是一个创建按钮的程序。

- 示例代码

```
01 import javax.swing.*;           //导入Swing包
02 //继承JFrame类
03 public class Swing9 extends JFrame
04 {
05     JButton jb=new JButton();     //创建一个按钮
06     //定义构造器
07     public Swing9()
08     {
09         this.setTitle("创建按钮"); //设置窗口名称
10         jb.setText("这是一个按钮"); //设置按钮上显示的内容
11         jb.setMnemonic('a');      //设置按钮的助记符
12         this.add(jb);              //将按钮添加到窗口中
13         this.setBounds(300,250,300,200); //设置窗口的大小和位置
14         this.setVisible(true);    //设置窗口是可见的
15     }
16     public static void main(String args[])
17     {
18         Swing9 s=new Swing9();
19     }
20 }
```

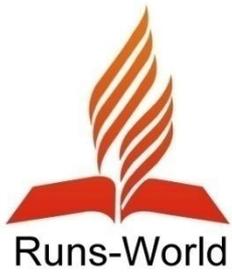


- 在示例代码14-9程序的第5行使用空构造器创建了一个按钮。在第10行使用**setText**方法设置了按钮上显示的内容。在第11行使用**setMnemonic**方法设置了按钮的助记符。在窗口中单击，可以看出是该界面中是一个按钮，同样使用**Alt+A**同样能起到单击按钮的作用。有些读者可能会感到奇怪，为什么整个窗口中就只有这一个按钮，而且占满整个窗口。这个问题在学完下一章就会明白是怎么回事。



14.5.3 按钮动作事件

- 按钮是具有动作事件的，单击按钮时触发动作事件，也就是ActionEvent事件。但是如果想让按钮在触发事件后执行程序，就需要为按钮添加动作事件监听器，并且需要为按钮注册动作事件监听器。编写动作事件监听器是通过实现ActionListener监听接口来完成的。
- 在ActionListener监听接口中只有一个actionPerformed方法，所以在动作事件监听器中只需要实现这一个方法。将触发事件后将执行的程序都写在actionPerformed方法中。定义完监听器后，还需要向按钮注册该监听器，注册监听器是通过addActionListener方法来实现的。



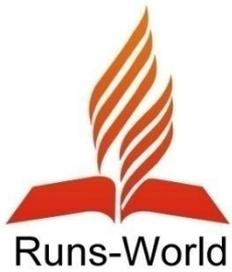
14.6 Swing中的事件

- 虽然在对按钮的讲解中已经对事件进行了使用，但是还是有必要对事件进行一个总体的讲解。对于一个界面程序来说，如果只能显示一些控件，这是完全不能满足功能要求的。通过事件的使用，就可以使界面具有更加丰富的功能。



14.6.1 事件简介

- 事件是一种很好的让界面和用户进行交互的手段。当用户和界面交互时，经常会进行一些操作，例如单击按钮，按下指定键盘键，都会触发事件。事件触发后会告诉程序发生的事件，程序会根据不同的事件来做出响应。在事件的发生和响应的过程中需要两个对象，事件源和事件监听器。
- 事件源就是触发事件的控件，这里包括按钮、文本框、窗体等很多种控件。但是不同的控件是存在不同的事件的，事件信息被封装在事件对象中。事件监听器是指实现专门的监听接口的类的对象。每一个事件都有对应的监听接口，同时在该接口中给出了处理事件的方法。在编写监听器时需要事件监听接口，同时实现其中的方法，在方法中编写触发事件后执行的程序。在编写程序时，还需要将监听器注册给事件源，这样才能执行事件。
- 提示：事件源和监听器之间是多对多的关系，一个事件源可以对应多个监听器，一个监听器可以为多个事件源服务，这在后面将会给出具体的程序进行讲解。



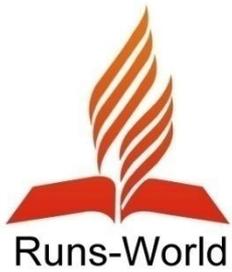
14.6.2 同一个事件源注册多个监听器

- 同一个事件源可以同时注册多个监听器，这种情况下触发事件，所有的监听器都将执行事件方法，对事件进行处理。
- 当为同一个事件源注册多个监听器时，监听器的执行顺序并不是先注册先执行的顺序，而是先注册后执行的顺序来执行的。读者可以写一个程序来演示这一点。



14.6.3 同一个监听器注册给多个事件源

- 同一个监听器注册给多个事件源的情况下，所有的事件源中的任意一个触发事件都会通知监听器，并执行监听器中的事件处理方法。
- 将一个监听器注册给多个事件源，从而不管是单击哪一个按钮，都将执行监听器方法。为了辨别是哪一个按钮被按下，在监听器方法中需要判断是哪一个按钮触发了事件，执行的效果是使另一个按钮的显示内容发生变化。



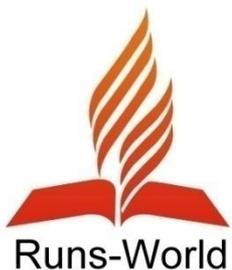
14.6.4 窗体获取和失去焦点事件

- 在Swing中，针对窗体的事件有很多，但是这些事件都是很容易理解的。窗体中的所有事件都是使用WindowEvent类来表示。在本节中就先来介绍窗体获取和失去焦点事件，该事件是通过实现WindowFocusListener监听接口实现的。



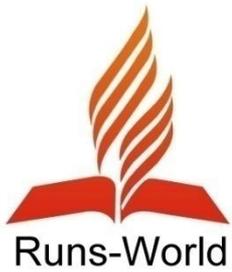
14.6.5 窗体打开、关闭和激活事件

- 要实现窗体打开、关闭和激活事件只需要实现WindowListener监听接口的监听器。WindowListener监听接口中同样具有很几种方法，这里还是通过程序来讲解这些方法。

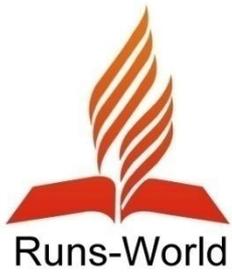


14.7 综合练习

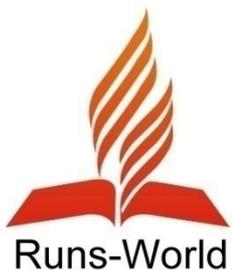
- 1. 开发一个同一个事件源注册多个监听器的程序。
- 06 JButton jb=new JButton(); //创建一个按钮
- 07 int i=0; //定义一个表示按下次数的变量
- 08 //定义构造器
- 09 public LianXi1()
- 10 {
- 11 this.setTitle("创建按钮"); //设置窗口名称
- 12 jb.setText("按钮按下了0次"); //设置按钮上显示的内容
- 13 jb.setMnemonic('a'); //设置按钮的助记符
- 14 this.add(jb); //将按钮添加到窗口中



- 15 //为按钮注册监听器
- 16 jb.addActionListener(new ActionListener())
- 17 {
- 18 //触发动作事件时，执行的方法
- 19 public void actionPerformed(ActionEvent e)
- 20 {
- 21 LianXi1.this.jb.setText("按钮按下了
- 22 "+(++i)+"次");
- 23 }
- 24);



```
• 25 //为按钮注册第二个监听器
• 26 jb.addActionListener(new ActionListener()
• 27     {
• 28         //触发动作事件时，执行的方法
• 29         public void actionPerformed(ActionEvent e)
• 30         {
• 31             LianXi1.this.jb.setText("按钮按下了"++i+"次");
• 32         }
• 33     }
• 34 );
• 35 this.setBounds(300,250,300,200); //设置窗口的大小和位置
• 36 this.setVisible(true); //设置窗口是可见的
• 37 }
• 38 public static void main(String args[])
• 39 {
• 40     LianXi1 s=new LianXi1();
• 41 }
• 42 }
```



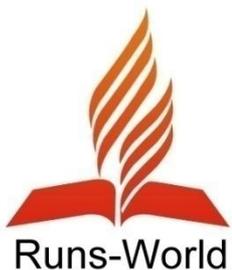
14.8 小结

- 本章是对Java中Swing程序入门的章，在本章中只是对界面开发作了一个简单的介绍。在本章中首先介绍了如何进行界面开发，然后分别介绍了窗口类、面板类、标签类和按钮类。在本章的最后对界面开发中非常重要的事件开发进行了讲解。



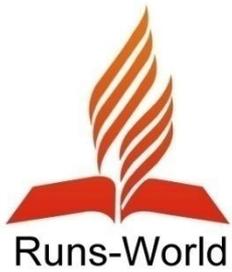
第15章 布局管理器

- 在日常生活中，超市已经变为一个必不可少的基础设施。在超市中，所有的商品都被超市管理人员有条理的分好类，摆在指定的位置，日常用品放在一起，食品放在一起。在Java Swing界面开发中，就用到了超市原理。其中窗体就好像一个超市，窗体中的控件就好像是商品，而布局管理器就是超市的管理人员。在Swing编程中使用布局管理器能够非常有效地对容器中的控件进行有条理并且美观的摆放。布局管理器也是有很多种的，包括流布局、网格布局、边框布局和空布局等，在本章中就来学习这些布局管理器。通过本章的学习，读者应该完成如下几个目标。
- 了解各种布局管理器的样式。
- 掌握每一种布局管理器的使用。



15.1 流布局

- 流布局是相对比较简单的一种布局管理器，也是最常用的布局管理器。在流布局中放置控件时，将按照控件的添加顺序，依次将控件从左到右进行摆放，并且在一行的最后会进行自动换行放置。在一行中，控件是默认居中放置的。

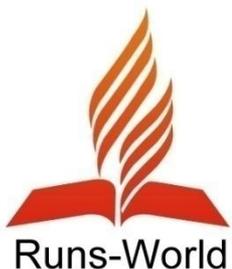


15.1.1 流布局介绍

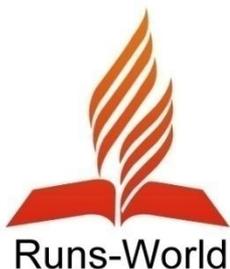
- 布局管理器也是通过构造器来创建的。流布局是通过FlowLayout类来创建，FlowLayout类具有三种构造器。首先是无参构造器，使用无参构造器能够创建一个默认的以居中对齐方式，控件间水平和垂直间距为5个像素的流布局。
- FlowLayout类还具有一个需要整型参数的构造器，使用该构造器能够创建一个指定对齐方式的流布局管理器，它的控件间水平和垂直间距仍然是默认的5个像素。流布局管理器的对齐方式如表所示。

流布局管理器对齐方式

对齐方式	说明
LEFT	控件左对齐
CENTER	控件居中，这也是默认值
RIGHT	控件右对齐
LEADING	控件与容器开始边对齐
TRAILING	控件与容器结束边对齐

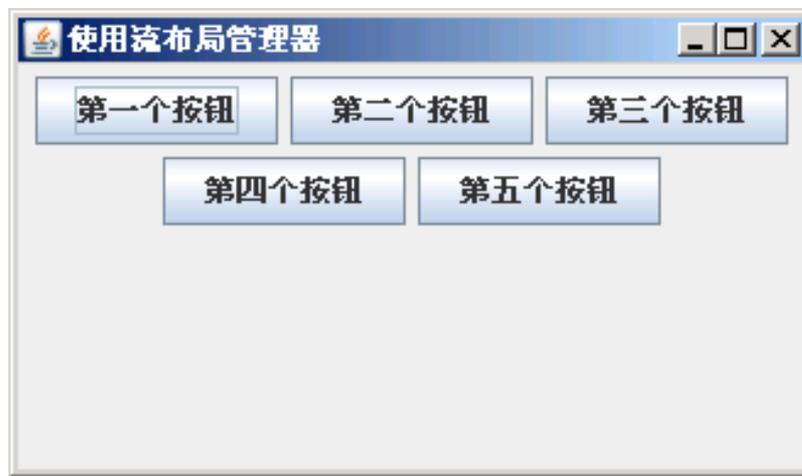


- 在创建流布局管理器时，就可以给出这些常量，来定义该流布局管理器的对齐方式。**FlowLayout**类还有一个具有三个参数的构造器，第一个参数表示流布局管理器的对齐方式，第二个参数表示流布局管理器中控件间水平间距，第三个参数表示流布局管理器中控件间垂直间距。
- **FlowLayout**类中还具有一些比较常用的方法，使用这些方法能够很有效地对流布局管理器进行操作。**getAlignment**方法和**setAlignment**方法分别获取和设置流布局管理器的对齐方式。**getHgap**方法和**setHgap**方法分别获取和设置流布局管理器中控件和控件之间的水平间距。**getVgap**方法和**setVgap**方法分别获取和设置流布局管理器中控件和控件之间的垂直间距。

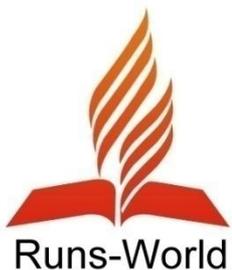


15.1.2 使用流布局

- 学习完如何创建流布局后，就可以自己动手来使用流布局。由于只学过按钮控件，所以这里只使用按钮来演示流布局管理器。
- 16 `this.setTitle("使用流布局管理器");` //设置窗口名称
- 17 `jp.setLayout(new FlowLayout());` //设置面板的布局为流布局
- 18 `jp.add(jb1);` //将按钮添加到面板中
- 19 `jp.add(jb2);`
- 20 `jp.add(jb3);`
- 21 `jp.add(jb4);`
- 22 `jp.add(jb5);`
- 23 `this.add(jp);` //将面板添加到窗口中



- 从运行结果中可以看出在流布局管理器中放置控件的方式，放置顺序是按照控件的先后顺序，从左到右依次摆放，当一行放不下时会进行自动换行。当控件不满一行时，会将该行中的控件居中显示。



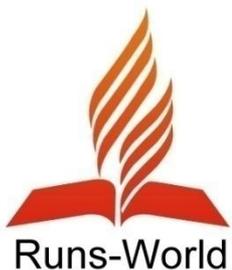
15.2 网格布局

- 网络布局也是一种比较常见的布局管理器。使用网格布局管理器后，会将所有的控件尽量按照给出的行数和列数来排列，同时网格布局管理器也会对控件进行尺寸的调整，使所有的控件具有相同的尺寸。在网格布局中，也会尽量使使用的空间成矩形的形式来显示。当窗体发生大小变化时，所有的空间也将自动改变大小来填充窗体。



15.2.1 网格布局介绍

- 网格布局是通过GridLayout类来创建的。GridLayout类具有三个构造器，使用无参构造器将创建具有默认行和默认列的网格布局。在创建网格布局管理器时最常用的就是具有两个整型参数的构造器，第一个参数表示网格布局管理器的行数，第二个参数表示网格布局管理器的列数。还有一个具有四个参数的构造器，除了可以定义行数和列数外，还可以定义控件间水平间距和垂直间距。
- GridLayout类中还定义了一些方法来对创建的网格布局进行操作。
 - getRows方法和setRows方法分别是获取和设置网格布局的行数。
 - getColumns方法和setColumns方法分别是获取和设置网格布局的列数。
 - getHgap方法和setHgap方法分别是获取和设置网格布局中控件间水平间距。
 - getVgap方法和setVgap方法分别是获取和设置网络布局中的控件间垂直间距。

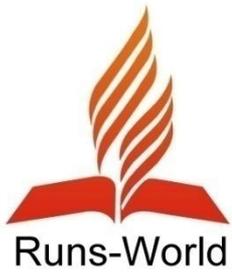


15.2.2 使用网格布局

- 学习完如何创建网格布局后，就可以自己动手来使用网格布局。这里由于只学过按钮控件，所以这里还是使用按钮来演示网格布局管理器。
- 16 `this.setTitle("使用网格布局管理器");` //设置窗口名称
- 17 `jp.setLayout(new GridLayout(3,2));` //设置面板的布局为网格布局
- 18 `jp.add(jb1);` //将按钮添加到面板中
- 19 `jp.add(jb2);`
- 20 `jp.add(jb3);`
- 21 `jp.add(jb4);`
- 22 `jp.add(jb5);`
- 23 `this.add(jp);` //将面板添加到窗口中
- 24 `this.setBounds(300,250,300,200);` //设置窗口的大小和位置

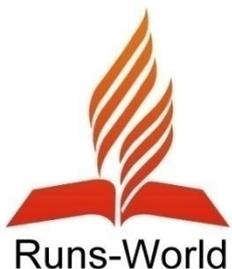


- 在示例代码**15-3**中将面板的布局管理器设置为网格布局，并将网格布局设置为三行两列，从而出现如图**15-4**的运行结果。如果在定义网格布局管理器时改变行数或列数，从而就会改变运行结果。



15.3 边框布局

- 前面学习的流布局和网格布局具有很多相似的地方，但是边框布局就和他们存在很大的不同。在使用边框布局时，通常都会由程序员来为控件指定在容器中的位置。边框布局将容器分为五个部分，包括东南西北中五部分。在每一个部分中只能放置一个控件，所以如果控件超过五个将不能完全显示。在使用边框布局时需要注意的是，当容器的大小发生变化时，四周的控件是不会发生变化的，只有中间的控件将发生变化。



15.3.1 边框布局介绍

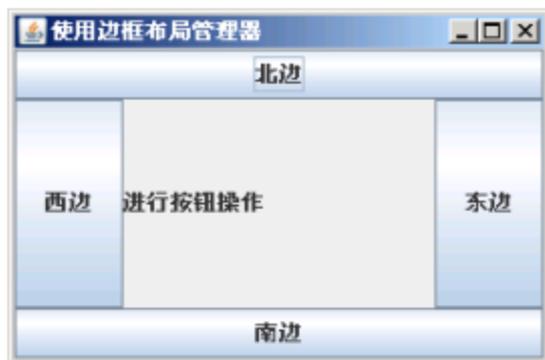
- 边框布局是通过BorderLayout类创建的。BorderLayout类具有两个构造器，一个是无参构造器，另一个是指定控件间间距的构造器，通常使用无参构造器来创建边框布局管理器。
- 在前面将控件添加到容器中都是通过add方法，将控件作为add方法的参数来进行添加的。但是在向边框布局容器中添加控件时，这样是不完全的。在向边框布局容器中添加控件是使用具有两个参数的add方法。其中第一个参数表示要添加的控件，第二个参数表示要添加到边框布局中的哪一个位置。边框布局的位置表示是通过常量来表示的，具体常量如表所示。

边框布局位置

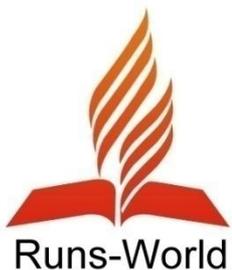
位置	说明
NORTH	容器的顶部
SOUTH	容器的底部
EAST	容器的左边
WEST	容器的右边
CENTER	容器的中间

15.3.2 使用边框布局

- 学习完如何创建边框布局后，就可以自己动手来使用边框布局。这里由于只学过按钮控件，所以还是使用按钮来演示边框布局管理器。
- **【范例】** 示例代码是一个使用边框布局的程序。



边框布局



15.4 空布局

- 空布局就是没有使用布局管理器，在空布局的情况下将根据控件的自身信息来为控件指定位置。在使用空布局时，就会使开发人员的工作量变的很多。开发人员需要为每一个控制指定位置和大小，因为这些都是开发人员指定的，所以当容器的大小发生变化时，控件的大小是不会发生变化的。虽然使用空布局加大了开发人员的工作量，但是这样使控件的摆放更加灵活，从而使界面的外观更加多样。



15.4.1 空布局介绍

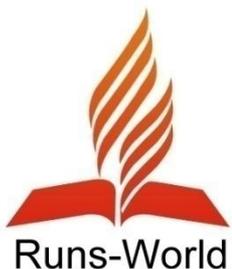
- 空布局是不需要使用类来创建的，只需要在程序指定布局管理器为null。将控件添加到空布局容器中时，仍然是使用add方法。因为这里使用的是空布局管理器，所以在添加控件之前，要对控件进行设置操作。设置操作是通过setBounds方法来完成的，setBounds方法的基本语法格式如下所示。
- `public void setBounds(int x,int y,int width,int height);`
- 其中x和y表示的是控件最左上侧的坐标，从而也固定了该控件的位置。width和height表示的是空间的宽度和高度，从而也指定了控件的大小。
- 使用空布局的优点是使用简单，对控件的摆放位置灵活。使用空布局的确定就是需要对每一个空间单独指定大小和位置，这在很多控件的情况下，工作量是可想而知的。

15.4.2 使用空布局

- 通过上一小节中对空布局的学习，已经对空布局有了基本了解。在使用空布局时要注意的是一必须为每一个控件设置位置和大小，这是读者特别需要注意的。
- **【范例】** 示例代码是一个使用空布局的程序。

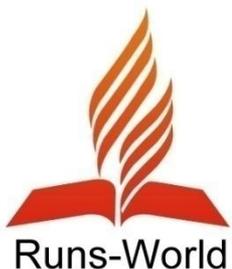


使用空布局



15.5 卡片布局

- 卡片布局是一种在Swing布局管理器中不很常用的布局管理器，但是使用卡片布局在特定情况下能够起到非常好的效果。在卡片布局的容器中可以添加任意多个控件，但是只能在容器中显示一个控件。添加在卡片布局容器中的控件的大小都和容器的大小相同，所有的控件也具有相同的大小。



15.5.1 卡片布局介绍

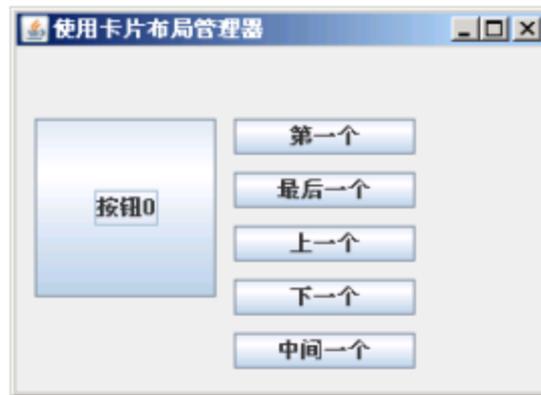
- 卡片布局是通过CardLayout类创建的。CardLayout类具有两个构造器，一个是无参构造器，另一个是需要指定控件和容器边界水平间距和垂直间距的构造器。这些已经在前面多次介绍，这里不再作过多介绍。
- 在卡片布局容器中一次只能显示一个控件，要想显示其他控件，就需要调用CardLayout类中的方法来执行。在CardLayout类的方法中，first方法和last方法分别是显示第一次添加的控件和显示最后一次添加的控件。next方法和previous方法分别是显示下一个添加控件和上一个添加控件。这四个方法都是具有一个参数的，该参数是指定对哪一个容器中的控件进行操作。



- **CardLayout**类中还有一个**show**方法，使用该方法可以显示指定的控件，该方法具有两个参数，第一个参数是指定对哪一个容器中的控件进行操作，第二个参数是指定要显示控件的名称。控件的名称是提前为控件设置的。
- 为控件起名称是在添加控件的时候设置的，向卡片布局容器中添加控件同样也是通过**add**方法来完成的。但是这里的**add**方法需要两个参数，第一个参数指定要添加的控件，第二个参数就是为该控件起的名称。

15.5.2 使用卡片布局

- 在使用卡片布局时，通常是一个非顶级容器来使用的，这是因为如果让顶级容器使用卡片布局，则整个窗体就显示一个控件，也无法进行操作显示其他控件，这样设计就没有意义。通常是将顶级容器设置为空布局管理器，在其中添加面板和控件。在面板中使用卡片布局，让控件对面板中的控件进行操作。
- **【范例】** 示例代码是一个使用卡片布局的程序。



卡片布局

15.6 综合练习

- 1. 编写一个让用户自由选择使用哪一种布局管理器进行控件显示的程序。
- 【提示】可以先只使用流布局和网格布局进行实验。在程序中首先要判断选择的是哪一种布局管理器，然后根据选择进行控件显示。

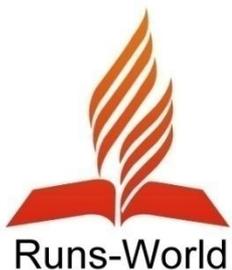


使用不同布局

- 2. 使用卡片布局编写一个摇奖程序。
- 【提示】摇奖程序肯定就是一个能够随机产生不同奖的程序。

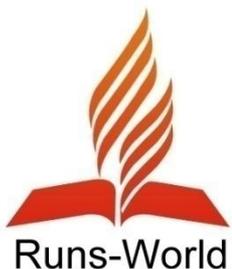


摇奖程序



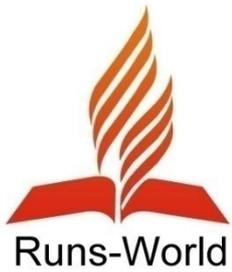
15.7 小结

- 通过本章的学习，已经对Swing开发中经常要使用的布局管理器进行了详细的讲解。Swing中不但可以使用本章中讲解的流布局、网格布局、边框布局、空布局和卡片布局，还可以使用箱式布局和弹簧布局。



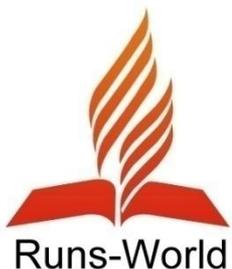
第16章 Swing常用控件

- 如果在超市中只卖一种商品，那就会非常单一的。在上一章学习布局管理器时就有这样的感觉，都是使用按钮控件来进行举例说明，这就好像超市中只卖一种商品。在本章中就来为超市中提供更多的商品，在Swing中也就是更多的控件。控件是使界面内容丰富的一个必不可少的一部分，在Swing中的控件除了按钮之外，还包括文本框、复选框、单选按钮和菜单等很多内容。通过本章的学习，读者应该完成如下几个目标。
- 了解如何创建文本框和文本框的实际应用。
- 了解如何创建复选框和复选框的实际应用。
- 了解如何创建单选按钮和单选按钮的实际应用。



16.1 文本框以及密码框和多行文本框

- 文本框和按钮一样，都是非常常用的控件，文本框提供了一个输入信息的控件。密码框和多行文本框是和文本框很相似的，密码框和文本框的外观是十分相似的，只是输入的内容显示为特殊符号，从而起到保护密码的作用。多行文本框从名称上就可以看出是一个具有多行文本的文本框，在多行文本框中输入内容时是可以进行换行操作的。

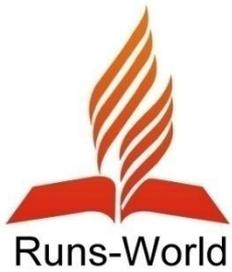


16.1.1 创建文本框

- 文本框是通过JTextField类来创建的，在创建的文本框中当文本超出文本框规定长度时，将自动滚动文本显示。文本框是通过JTextField类的构造器创建的，包括5种构造器，如表所示。

JTextField类构造器

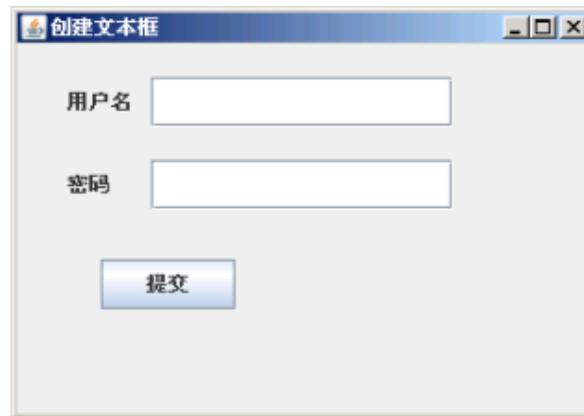
构造器	说明
<code>public JTextField()</code>	创建普通的文本框
<code>public JTextField(String text)</code>	创建具有默认值的文本框
<code>public JTextField(int columns)</code>	创建具有指定长度的文本框
<code>public JTextField(String text,int columns)</code>	创建具有默认值和指定长度的文本框
<code>public JTextField(Document doc,String text,int columns)</code>	创建具有默认值、长度与文档模型的文本框



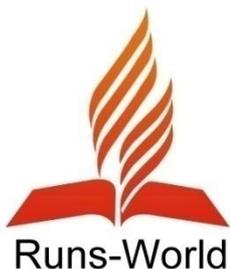
- 提示：文本框也是会触发事件的，它和按钮一样，都是触发 **ActionEvent** 事件。按钮是被单击时触发事件，而文本框是当用户按下回车键时触发事件。

16.1.2 创建密码框

- 密码框是文本框的改进的控件，是一种专门用于输入密码的文本框。在文本框中输入信息后，将不直接显示输入的信息，而是使用特定的特殊字符来进行显示。密码框是通过JPasswordField类来创建的，因为密码框和文本框的关系，所以JPasswordField类的构造器是和JTextField类的构造器相同的。



创建密码框



16.1.3 创建多行文本框

- 当希望进行多行输入时，文本框就不能满足其要求，这时候就需要创建多行文本框。多行文本框也是文本框的一种特殊形式，多行文本框是通过JTextArea类实现的。JTextArea类中提供了6种构造器来创建多行文本框，构造器如表所示。

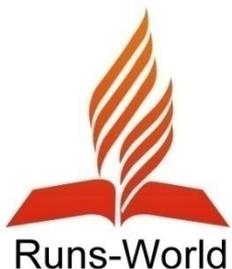
JTextArea类构造器

构造器	说明
<code>public JTextArea()</code>	创建一个没有内容的JTextArea，默认的行数和列数为0。
<code>public JTextArea(String text)</code>	创建一个具有默认内容的JTextArea，行数和列数为0。
<code>public JTextArea(int rows,int columns)</code>	创建一个具有指定的行和列的JTextArea，参数rows与columns分别表示指定的行与列。
<code>public JTextArea(String text,int rows,int columns)</code>	创建一个具有默认内容以及行数和列数的JTextArea，参数text为指定的文本内容，数rows与columns分别表示指定的行数与列数。
<code>public JTextArea(Document doc)</code>	创建一个具有指定文档模型的JTextArea，行数和列数为0。
<code>public JTextArea(Document doc,String text,int rows,int columns)</code>	创建一个具有指定文档模型、内容、行与列的JTextArea。



16.2 复选框和单选按钮

- 复选框和单选按钮具有很多相似的地方，它们在实际开发中也经常要使用到。例如需要用户来选择兴趣爱好时，一般都很少是只有一个爱好的，这里就可以创建复选框来实现其功能，来让用户进行多项选择操作。但是例如性别等信息，是不可能存在多个选择的，它只能在有限的几个选项中选择其中一个，这里就可以使用单选按钮。



16.2.1 创建单选按钮

- 单选按钮是一种只能在一组选项中选择其中一个选项的控件。单选按钮是通过使用JRadioButton类来创建的，在JRadioButton类中具有7种构造器形式，构造器如表所示。

JRadioButton类构造器

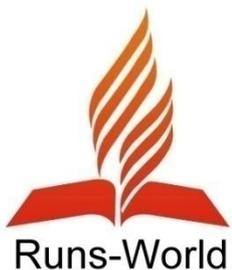
构造器	说明
<code>public JRadioButton()</code>	创建一个没有文本与图标并且未被选定的单选按钮。
<code>public JRadioButton(Icon icon)</code>	创建一个具有指定图标默认没有选中的单选按钮。
<code>public JRadioButton(Icon icon,boolean selected)</code>	创建一个具有指定图标的单选按钮。若selected为true，则该单选按钮处于选中状态，反之则为未选中状态。
<code>public JRadioButton(String text)</code>	创建一个具有指定文本默认没有选中的单选按钮，参数text为指定的文本。
<code>public JRadioButton(String text,boolean selected)</code>	创建一个具有指定文本的单选按钮，参数text为指定的文本。若selected为true，则该单选按钮处于选中状态，反之则为未选中状态。
<code>public JRadioButton(String text,Icon icon)</code>	创建一个具有指定文本和图标默认没有选中的单选按钮，参数text为指定的文本，参数icon为指定的图标。
<code>public JRadioButton(String text,Icon icon,boolean selected)</code>	创建一个具有指定文本和图标的单选按钮，数text为指定的文本，参数icon为指定的图标。

16.2.2 创建复选框

- 通过JRadioButton与ButtonGroup的配合使用，可以很方便地实现单项选择。若需要使用多项选择，则应该使用复选框——JCheckBox类。与JRadioButton的不同是，JCheckBox不需要编组使用，各个选项之间没有逻辑约束关系。
- 提示：通过JRadioButton与ButtonGroup的配合使用，可以很方便地实现单项选择。
- 该类提供了8个构造器，表中列出了其中7个比较常用的。

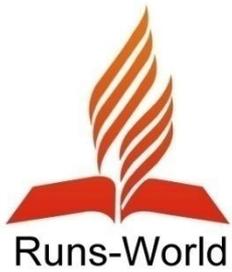
JCheckBox类的常用构造器

构造器签名	说明
<code>public JCheckBox ()</code>	创建一个没有文本与图标并且未被选定的复选框。
<code>public JCheckBox (Icon icon)</code>	创建一个具有指定图标默认未被选中的复选框，参数 <code>icon</code> 为指定的图标。
<code>public JCheckBox (Icon icon,boolean selected)</code>	创建一个具有指定图标的复选框，参数 <code>selected</code> 表示复选框的选中状态。若 <code>selected</code> 为 <code>true</code> 则处于选中状态，否则处于未选中状态。
<code>public JCheckBox (String text)</code>	创建一个具有指定文本默认未被选中的复选框，参数 <code>text</code> 为指定的文本。
<code>public JCheckBox (String text,boolean selected)</code>	创建一个具有指定文的复选框，参数 <code>selected</code> 表示复选框的选中状态。若 <code>selected</code> 为 <code>true</code> 则处于选中状态，否则处于未选中状态。
<code>public JCheckBox (String text,Icon icon)</code>	创建一个具有指定文本和图标默认未被选中的复选框，参数 <code>text</code> 为指定的文本，参数 <code>icon</code> 为指定的图标。
<code>public JCheckBox (String text,Icon icon,boolean selected)</code>	创建一个具有指定文本和图标的复选框，参数 <code>text</code> 为指定的文本，参数 <code>icon</code> 为指定的图标，参数 <code>selected</code> 表示复选框的选中状态。若 <code>selected</code> 为 <code>true</code> 则处于选中状态，否则处于未选中状态。



16.3 选项卡

- 选项卡也是开发GUI界面常用的控件之一，通过使用选项卡可以在同一个窗体中提供很多不同的界面，可以通过选项卡提供的标签在界面间方便地进行切换。本节将为读者详细介绍如何使用Swing中的选项卡，主要包括JTabbedPane类、ChangeEvent事件以及具体案例等内容。

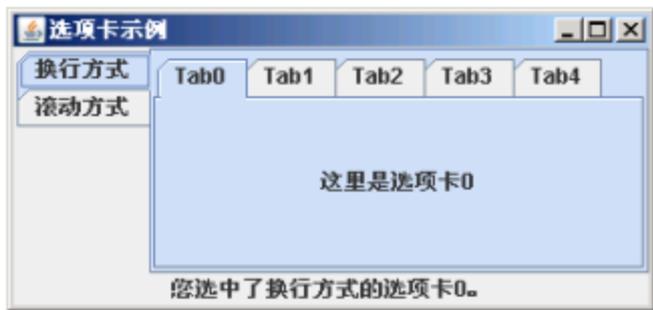


16.3.1 选项卡介绍

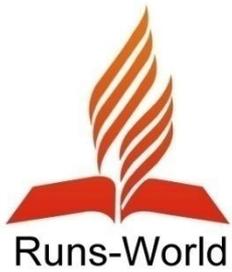
- 选项卡可以产生多个标签框架，每一个标签框架窗口自成一个系统，即包含多个页面，每个页面与一个标签对应。当选择某一个标签时，标签框架窗口会自动显示出此标签框架的内容，并触发一个ChangeEvent事件，这个事件由ChangeListener监听器监听并处理。
- 注意：每次只能选择标签组的一个标签。
- 选项卡具有以下构造方法及常用方法。
- `public JTabbedPane()` 方法：该方法创建一个TabbedPane对象，该对象具有默认的JTabbedPane.TOP选项卡布局。
- `public JTabbedPane(int tabPlacement)` 方法：该方法使用指定的参数tabPlacement创建一个TabbedPane对象，tabPlacement参数有常值，JTabbedPane.TOP、JTabbedPane.BOTTOM、JTabbedPane.LEFT或JTabbedPane.RIGHT。
- `public JTabbedPane(int tabPlacement, int tabLayoutPolicy)` 方法：该方法以指定的参数tabPlacement、tabLayoutPolicy创建一个TabbedPane对象，该对象具有指定的tabPlacement选项卡布局和tabLayoutPolicy选项卡布局策略。其中参数tabLayoutPolicy有常值，JTabbedPane.WRAP_TAB_LAYOUT、JTabbedPane.SCROLL_TAB_LAYOUT。

16.3.2 创建选项卡

- 学习了JTabbedPane类的构造器和相关方法就可以来创建选项卡。选项卡的创建和前面控件的创建都是很类似的。
- 【范例】示例代码是一个创建选项卡的程序。



创建选项卡



16.4 分割窗格

- 分割窗格（JSplitPane）也是Swing中常用的控件之一，其能够将单个空间分割成两个部分，并在两个部分中显示不同的内容，本节将为读者详细介绍JSplitPane类的相关知识及使用。



16.4.1 分割窗格介绍

- JSplitPane控件允许在单个空间中放置两个控件，开发人员可以自由决定按水平方向或垂直方向划分空间，还可以在程序运行期间使用鼠标自由调整空间的分割比例。通过JSplitPane控件的嵌套使用，可以将空间分割成更多的部分。JSplitPane类提供了5个构造器，如表所示。

JSplitPane类的构造器

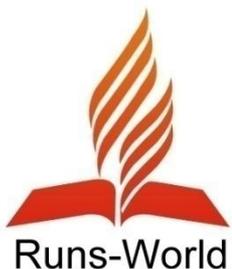
构造器签名	说明
<code>public JSplitPane()</code>	创建水平分割的分割窗格，并自动在左右两个空间中各添加一个按钮控件来填充。
<code>public JSplitPane(int newOrientation)</code>	创建按指定方向分割的分割窗格，参数newOrientation表示分割的方向。
<code>public JSplitPane(int newOrientation, boolean newContinuousLayout)</code>	创建按指定方向分割的分割窗格，参数newOrientation表示分割的方向，参数newContinuousLayout决定当分隔条改变位置时控件是否连续重绘。
<code>public JSplitPane(int newOrientation, Component newLeftComponent, Component newRightComponent)</code>	创建按指定方向分割的分割窗格，参数newOrientation表示分割的方向。参数newLeftComponent与newRightComponent分别为左（上）边与右（下）边的指定控件。
<code>public JSplitPane(int newOrientation, boolean newContinuousLayout, Component newLeftComponent, Component newRightComponent)</code>	创建按指定方向分割的分割窗格，参数newOrientation表示分割的方向，参数newContinuousLayout决定当分隔条改变位置时控件是否连续重绘。参数newLeftComponent与newRightComponent分别为左（上）边与右（下）边的指定控件。

16.4.2 创建分割窗格

- 通过上一小节介绍，对JSplitPane控件有了大体的了解，其能够将窗格分割成两个部分，且只能是两个部分，但是可以通过嵌套使用的方式将窗格分割成任意多份。本小节将给出一个嵌套使用JSplitPane控件的例子，进一步加深读者对该控件的理解。
- **【范例】** 示例代码是一个创建分割窗格的程序。

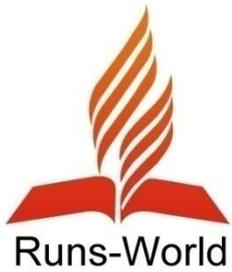


创建分割窗格



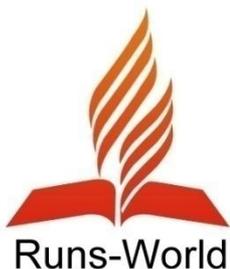
16.5 滑块和进度条

- 很多GUI应用程序中都通过滑块来让用户进行指定范围值的输入，这样很方便，而且用户也不再可能输入错误的数值，使界面变得很友好。通过进度条系统能够向用户即时反馈一些信息，避免用户不知道系统处于何种状态而焦急的等待。本节将向读者介绍如何使用Swing中提供的滑块与进度条，通过本节的学习，读者也可以方便地开发出使用滑块与进度条的应用。



16.5.1 创建滑块

- JSlider类是Swing包中提供的用于实现滑块的控件，通过JSlider控件可以让用户在限定的范围内方便地选择需要的值。JSlider类提供的滑块可以是水平方向的，也可以是垂直方向的，并且可以根据需要设置成为不同的外观风格。
- 提示：滑块组件是由可以拖动的滑块和一个范围组件组成的；用户可以通过拖动滑块在一个区间范围里进行选择。
- `JSlider ageSlider = new JSlider();`
- `ageSlider = new JSlider(SwingConstants.VERTICAL, 0, 120, 20);`

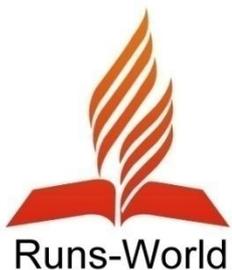


16.5.2 创建进度条

- JProgressBar是Swing中提供的用来实现进度条的控件，使用其可以非常方便地完成进度条的开发。在应用中恰当使用进度条可以即时通告用户系统的一些信息，避免用户因不知道系统运行情况而焦急地等待，从而使界面更加友好。
- 注意：Swing中提供的进度条不但可以像常见的进度条一样显示工作的进度，而且可以通过设置为模糊模式以动画形式来表示系统正在运行。
- JProgressBar类提供了5个构造器，其中有4个是比较常用的，表列出了这4个常用的构造器。

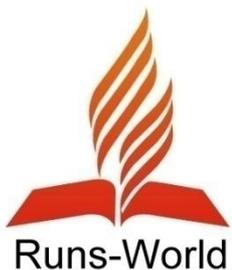
JProgressBar类的常用构造器

构造器签名	功能
<code>public JProgressBar()</code>	创建一个显示边框但不显示信息字符串的水平进度条，初始值和最小值都为0，最大值为100。
<code>public JProgressBar(int orient)</code>	创建一个显示边框但不显示信息字符串并且具有指定方向的进度条，初始值和最小值都为0，最大值为100，参数orient指定了进度条的方向。
<code>public JProgressBar(int min,int max)</code>	创建一个显示边框但不显示信息字符串并且具有指定最小值和最大值的水平进度条，进度条的初始值设置为指定的最小值，参数min与参数max分别为指定的最小值与最大值。
<code>public JProgressBar(int orient,int min,int max)</code>	创建一个显示边框但不显示信息字符串并且具有指定最小值和最大值以及指定方向的进度条，参数orient指定了进度条的方向，参数min与参数max分别为指定的最小值与最大值。



16.6 列表框

- 很多GUI应用程序中都需要让用户从一些选项中选择一项或多项，如果选项不多采用单选按钮或复选框是很方便的，但如果选项比较多就是采用列表框比较合适了。列表是图形用户界面程序中常用到的组件，列表允许用户从列表项中选择一个或多个选项，默认情况下，列表支持单选；选择状态由分隔符来区别。并且可以通过JList中提供的setSelectionMode方法使得列表支持多选。

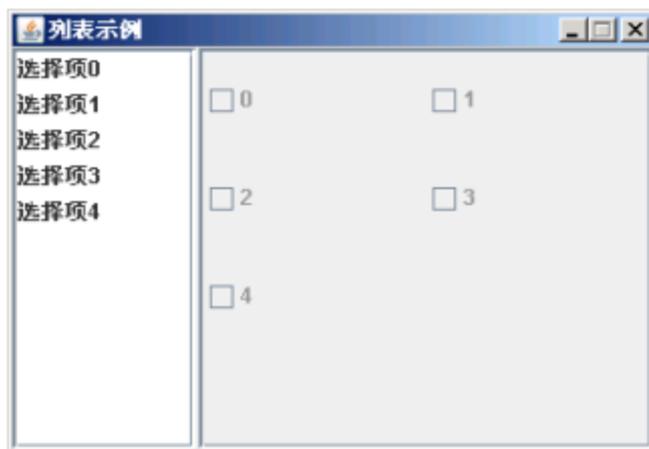


16.6.1 列表框介绍

- JList能够为用户提供一组可供选择的选项，这些选项可以以一列或多列的形式显示。默认的选择模式下，可以通过鼠标单击来选择单个选项，也可以在按住特定控制键的同时，单击鼠标来进行多项选择。
- 注意：JList类没有提供滚动功能，但是可以通过将其放置在JScrollPane中来实现滚动操作。
- JList的常用构造方法如下所示：
- `public JList()` 方法：该方法可以构造一个使用空模型的JList对象。
- `public JList(Object[] listData)` 方法：该方法以显示指定数组中的元素构造一个JList。

16.6.2 创建列表框

- 通过对JList类构造器的学习，就可以使用JList类构造器来创建列表框。在本节中就来使用JList类来创建一个列表框，同时为该列表框注册事件监听器。
- **【范例】** 示例代码是创建列表框的程序。



创建列表框



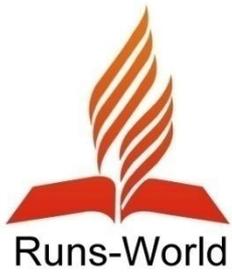
16.6.3 下拉列表框

- 下拉列表及组合框，下拉列表与列表不同的是，下拉列表只支持单个选项，只允许用户选择一个选项。优点是能节省空间，使界面更紧凑。并且只有用户单击下拉列表时，列表选项才会显示。
- 注意：在默认的情况下，下拉列表是不可以被用户编辑的，但是可以使用JComboBox提供的方法setEditable方法使其可以被编辑。
- JComboBox控件实际上组合了一个文本框与一个下拉列表，在默认情况下JComboBox控件提供的文本框是不可编辑的。在文本框旁边有一个包含向下箭头的小按钮，在按下这个按钮之后，会出现显示选项的弹出式列表，用户可以从其中选择需要的选项。



16.7 菜单

- 随着GUI开发的普及，菜单在开发中也变的越来越重要，几乎每个应用程序都会提供相应的菜单。因此，Swing为菜单的开发提供了良好的支持，通过Swing中提供的菜单系列控件，开发人员可以非常方便地开发出各种各样的菜单，本节将对Swing中菜单的开发进行详细的介绍。



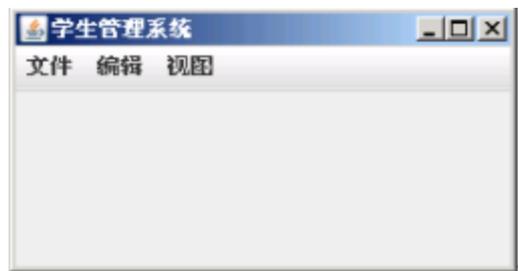
16.7.1 菜单介绍

- 菜单（JMenu）是标题栏下面的一行文字部分。菜单是应用程序中最常用的组件。菜单的组织方式为：一个菜单条JMenuBar包含多个菜单项（JMenuItem）。JMenuItem有两个子类，分别为JRadioButtonMenuItem及JCheckBoxMenuItem用于表示单选菜单项和复选菜单项。当用户选择某个菜单项后，就会触发一个ActionEvent时间，由ActionListener监听器处理。
- 菜单项有两种状态：启用状态和禁用状态，菜单项的状态可以使用setEnabled()方法设置。创建完整的菜单一般需要以下步骤：
 - 创建菜单栏
 - 创建菜单以及子菜单
 - 创建菜单项
 - 将菜单项加入到子菜单或菜单中，将子菜单加入到菜单中，将菜单加入到菜单栏中

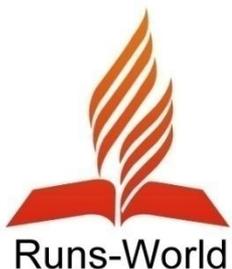


16.7.2 创建菜单

- 在上一小节对菜单进行了简单的介绍，在本节中就来先创建一个简单的菜单，然后再对菜单进行更详细的讲解。
- **【范例】** 示例代码是一个创建菜单的程序。



创建菜单

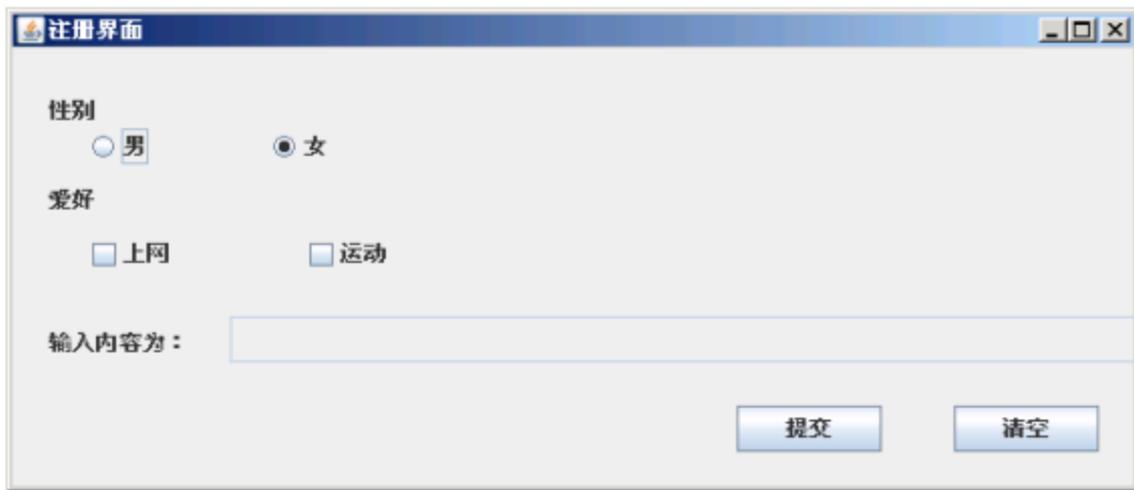


16.7.3 创建弹出式菜单

- 弹出式菜单有时也称为右键菜单，其一般在用户按下鼠标右键时在鼠标位置弹出，能够给用户的操作提供更大的方便。从某种程度上来说，右键菜单设计的好坏直接影响应用程序的易用性，本节将对Swing中弹出式菜单的开发进行详细的介绍。
- 注意：JPopupMenu类实现弹出菜单。JPopupMenu类不是继承JMenu类的而是从JComponent类继承过来。弹出式菜单的创建和菜单的创建基本相同，也需要新建一个弹出式菜单后再加入菜单项。
- 通过调用JPopupMenu类提供的show方法可以将弹出式菜单显示在指定控件的指定位置，下面的代码片段说明了如何显示弹出式菜单。
- ```
//测试鼠标事件是否应该触发弹出式菜单
```
- ```
if(jpm.isPopupTrigger(e))
```
- ```
{//显示弹出式菜单
```
- ```
jpm.show(this, e.getX(), e.getY());
```
- ```
}
```

## 16.8 综合练习

- 1. 使用本节所学的控制编写一个用户注册程序。
- **【提示】**可以先搭建一个最简单的界面程序，在向里面添加其他控件。例如下面给出的让用户选择性别和爱好的程序。



注册界面

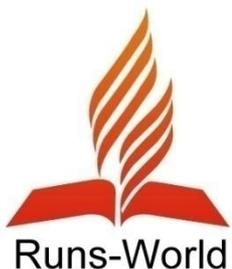
性别  
 男  女

爱好  
 上网  运动

输入内容为:

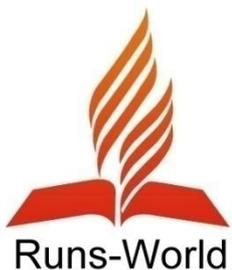
提交 清空

注册页面程序



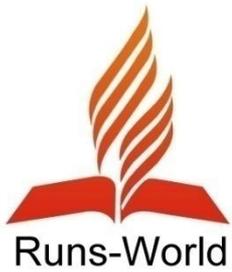
## 16.9 小结

- 在本章中对Swing中的常用控件进行了讲解，包括文本框、复选框、单选按钮、选项卡、分割窗格、滑块、滚动条、列表框和菜单。Swing中的控件除了这些外，还有很多控件，但是这些控件都是不常用的。



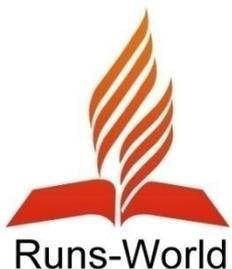
## 第17章 JDBC数据库编程

- 在超市中购买东西时，在很多的商品中是不容易找到自己想要的商品的，这时候通常就会找超市管理人员来帮忙解决。这就好像Java中的数据库编程JDBC。在Java程序中，如果希望对很多的数据进行操作时，通过使用数据库编程来解决。在本章中就来学习如何进行数据库编程。通过本章的学习，读者应该完成如下几个目标。
- 对数据库有基本了解。
- 熟练掌握JDBC的编程步骤。
- 掌握如何在Java中进行数据库操作。



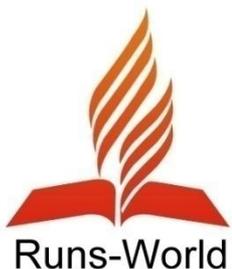
## 17.1 数据库基本介绍

- 数据库在应用程序中占有相当重要的地位，几乎所有的系统都必须要有数据。数据库发展到现在已经相当成熟了，已由原来的Sybase数据库，发展到现在的SQL（Structured Query Language）、Oracal等高级数据库。



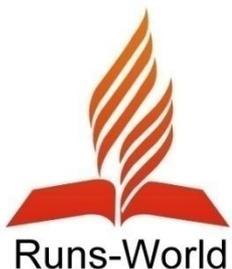
## 17.1.1 数据库介绍

- 首先从数据库的介绍上来看一下什么是数据库。数据库的基本结构分三个层次，反映了观察数据库的三种不同角度。物理数据层是数据库的最内层，是物理存贮设备上实际存储的数据的集合。这些数据是原始数据，同时也是加工的对象，由内部模式描述的指令操作处理的位串、字符和字组成。
- 概念数据层是数据库的中间一层，是数据库的整体逻辑表示，指出了每个数据的逻辑定义及数据间的逻辑联系，是保存记录的集合。它所涉及的是数据库所有对象的逻辑关系，不是它们的物理情况，而是数据库管理员概念下的数据库。
- 逻辑数据层是用户所看到和使用的数据库，表示了一个或一些特定用户使用的数据集合，即逻辑记录的集合。

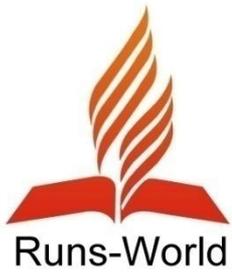


## 17.1.2 数据库应用架构

- 数据库应用架构包括两种不同形式的数据库应用程序架构模型，主要包括C/S两层结构的与三层（或多层）结构的两种。
- 两层结构数据库应用架构模型的特点是所有的用户输入、验证以及数据访问的功能都位于客户端中，一般来说客户端只适用于某一种特定的数据库。客户端与数据库服务器二者之间一般使用专用的协议进行联接，也有的情况是使用通用的数据库联接，如JDBC、ODBC等。
- 但是使用两层结构数据库也是存在很大缺点的。客户端与数据库服务器之间直接耦合，依赖度很高，无论哪边发生变化，都会直接影响到另一边。任何一种数据库服务器能够支持的联接数都是很有限制的，如果客户端很多，而又让每个客户端独自占用一个数据库联接不利于提高数据库的利用效率，也有可能造成其他用户不能正常使用数据库。

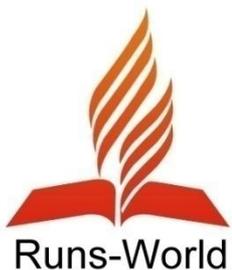


- 提示：在现在的开发中，已经很少使用两层结构的数据库应用模型了，而都是使用更加优越的三层结构数据库应用模型。三层结构的数据库应用模型的特点主要是，客户端与数据库之间不直接耦合，而是通过中间层应用服务器进行耦合，当客户端或数据库需要发生变化时可以通过中间层隔离变化，减小影响。
- 一般情况下在三层结构中，客户端软件都由通用的浏览器来担任，这样在对应用进行部署时就省去了为每台机器安装专用客户端的麻烦。同时，当开发了新的应用后，客户端机器也不需要做任何改变，打开浏览器浏览的自然就是新的功能了。根据需要，中间层的应用服务器可以同时连接几个同构或异构的数据库服务器，而这些在客户端的使用者是感觉不到也不用关心的。每个客户端不必独占一个数据库连接，可以大大提高数据库连接与数据库的利用效率。

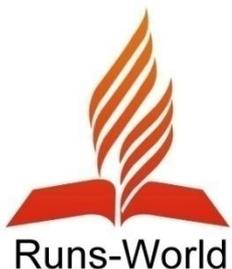


## 17.1.3 数据库模型

- 数据库又可以从基于不同的模型来分类，可以分为层次型数据库、网状型数据库、关系型数据库、面向对象型数据库。层次型数据库是一组通过链接而互相联系在一起的记录。树结构图是层次型数据库的模式。层次模型的特点是记录之间的联系是通过指针实现，表示的是对象的联系。其缺点是无法反映多对象的联系，并且由于层次顺序的严格和复杂，导致数据的查询和更新操作复杂，因此应用程序的编写也比较复杂。
- 网状数据库是基于网络模型建立的数据库。网络模型，是使用网络结构表示实体类型、实体间联系的数据模型。网状模型的特点是记录之间的联系通过指针实现，多对多的联系容易实现。缺点是编写应用程序比较复杂，程序员必须熟悉数据库的逻辑结构。

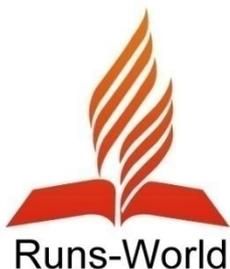


- 关系数据库是基于关系模型建立的数据库。关系模型由一系列表格组成，用表格来表达数据集，用外键（关系）来表达数据集之间的联系。现在应用最常见的就是关系数据库，在下一节也主要来介绍一下关系数据库。
- 提示：关系数据库是使用最广泛的数据库。
- 对象型数据库是建立在面向对象模型基础之上。面向对象模型中最基本的概念是对象和类。对象是现实世界中实体的模型化，共享同一属性集和方法集的所有对象构成一个类。类可以有嵌套结构。系统中的所有类组成一个有根、有向无环图，称为类层次。



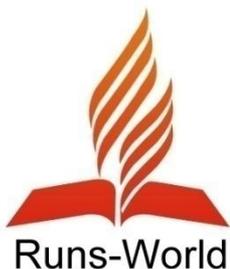
## 17.2 JDBC数据库编程介绍

- JDBC就是Java DataBase Connectivity, Java数据库连接。JDBC主要完成下面几个任务。与数据库建立一个连接。向数据库发送SQL语句。处理数据库返回的结果。实用Java程序语言和JDBC工具包开发程序, 是独立于平台和厂商的。JDBC就是将Java程序语言编写出来的程序, 与数据库相连接。接下来, 将详细讲述如何利用JDBC为程序连接数据库。

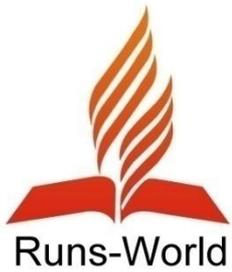


## 17.2.1 JDBC和ODBC的关系

- 在JDBC数据库编程中经常要使用ODBC。所以，在讲述JDBC的驱动程序分类之前，首先介绍什么是ODBC。ODBC是指Open DataBase Connectivity，即开放数据库互连，它建立了一组规范，并且提供了一组对数据库访问的标准API（应用程序编程接口），这些API利用SQL来完成其大部分任务。ODBC也提供了对SQL的支持。
- JDBC驱动程序由实施了这些接口的类组成，JDBC的总体结构有四个组件：应用程序、驱动程序管理器、驱动程序和数据源。将JDBC转换成ODBC驱动器，依靠ODBC驱动器和数据库通信。在这种方式下，ODBC驱动程序和桥代码必须出现在用户的每台机器中，这种类型的驱动程序最适合于企业网（这种网络上客户机的安装不是主要问题），或者是用Java编写的三层结构的应用程序服务器代码。

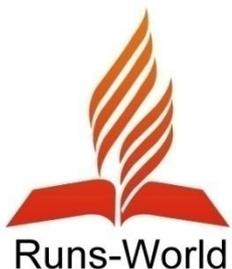


- 本地API一部分用Java来编写的驱动程序。这种类型的驱动程序把客户机API上的JDBC调用转换为Oracle、Sybase、Informix、DB2或其他DBMS的调用。像桥驱动程序一样，这种类型的驱动程序，要求将某些二进制代码加载到每台客户机上。
- JDBC网络纯Java驱动程序将JDBC转换为与DBMS无关的网络协议，这种协议又被某个服务器转换为一种DBMS协议。这种网络服务器中间件能够将它的纯Java客户机连接到多种不同的数据库上，所用的具体协议取决于提供者。通常，这是最为灵活的JDBC驱动程序。所有这种解决方案的提供者，都提供适合于Intranet用的产品。为了使这些产品支持Internet，它们必须处理Web所提出的安全性、通过防火墙的访问等额外要求，几家提供者正将JDBC驱动程序，加到他们现有的数据库中间件产品中。
- 本地协议纯Java驱动程序类型的驱动程序将JDBC调用直接转换为DBMS所使用的网络协议，这将允许从客户机机器上直接调用DBMS服务器，是Intranet访问的一个很实用的解决方法。由于许多这样的协议都是专用的，因此数据库提供者自己将是主要来源。

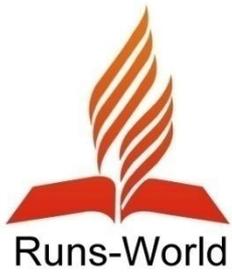


## 17.2.2 为什么使用JDBC数据库编程

- 目前市面上有很多种数据库，例如Oracle、Sybase、MS SQL Server和MS Access等数据库。有些读者就会认为这些多数据库，这里要学习数据库编程，是不是就要学习对应每一种数据库的编程方法呢。在JDBC之前是这样的，但是有了JDBC后，就变的非常容易。
- 使用JDBC在数据库编程中将起到非常重要的作用。首先程序员可以使用Java开发基于数据库的应用程序，在遵守Java语言规则的同时，可以使用标准的SQL语句访问任何数据库。如果数据库厂商提供较低层的驱动程序，程序员可以在自己的软件中，使用比较优化的驱动程序。



- 很多数据库系统带有JDBC驱动程序，Java程序就通过JDBC驱动程序与数据库相连，执行查询、提取数据等操作。Sun公司还开发了JDBC-ODBC bridge，用此技术，Java程序就可以访问带有ODBC驱动程序的数据库。目前，大多数数据库系统都带有ODBC驱动程序，所以，Java程序能访问诸如Oracle、Sybase、MS SQL Server和MS Access等数据库。



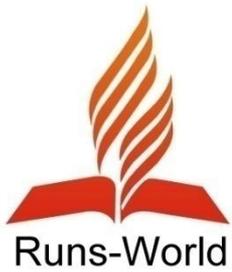
## 17.3 SQL数据库操作技术

- 在Java中进行数据库操作，除了需要JDBC编程外，还需要一个数据库的技术，那就是SQL技术。SQL技术是专门的，直接的对数据库操作的技术。



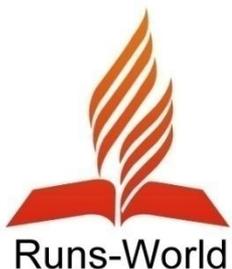
## 17.3.1 什么是SQL

- SQL是Structured Query Language的缩写，Structured Query Language翻译过来叫做结构化查询语言。SQL是一种专门用来与数据库通信的语言。与其他语言（如Java、Visual Basic这样的程序设计语言）不一样，SQL由很少的词构成，这是有意而为的。SQL能够很好地完成一项任务——提供一种从数据库中读写数据的简单有效的方法。
- SQL是存在很多优点的，从整体的角度来说SQL有如下的优点。首先SQL不是某个特定数据库供应商专有的语言，几乎所有重要的DBMS都支持SQL，所以，此语言几乎能与所有数据库打交道。然后就是SQL简单易学。它的语句全都是由具有很强描述性的英语单词组成，而且这些单词的数目不多，这个就是它简单易学的主要原因。最后SQL看上去尽管很简单，但实际上是一种强有力的语言，灵活使用其语言元素，可以进行非常复杂和高级的数据库操作。



## 17.3.2 如何进行SQL操作

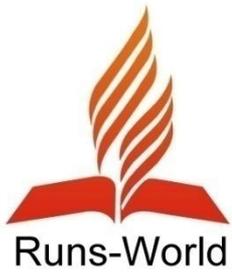
- 使用SQL能够完成数据库的创建、添加、删除、修改、查询等操作。在本节中就来简单的学习一下如何进行SQL操作。
- 查询操作是数据库操作中最常见的操作。在SQL中，使用Select语句可在需要的表单中检索数据，在进行检索之前，必须知道需要的数据存储在哪里，Select语句可由多个查询子句组成。查询操作的基本结构如下所示。
- `Select [all|distinct] [into new_table_name]`
- `From [表名|视图名]`
- `(where 搜索条件)`
- `Group by 把查到的按什么标准分组`
- `(having 搜索条件)`
- `(order by 按什么顺序排序) (升序|降序)`



- 在查询操作的基本结构中，**all**是指明查询结果中可以显示值相同的列，同时**all**是系统默认值。**distinct**是指明查询结果中如有值相同的列，只显示其中的一列，对**distinct**来说，**NULL**被认为相同的值。**into**子句用于把查询结果存放到一个新建表中。
- 注意：**select.....into**句式不能与**compute**子句一起使用。新表是由**select**子句指定的列构成。

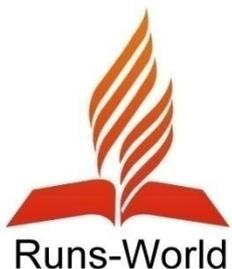


- **From**子句指定需要进行数据查询的表。**where**子句指定数据检索的条件，以限制返回的数据。**Group by**子句指定查询结果的分组条件。**having**子句指定分组搜索条件，它通常与**Group by**子句一起使用。**order by**子句指定查询结果的排序方式。
- 除了查询操作外，还有其他操作。数据插入语句如下：
- 数据插入：**insert into <表名>〔列名〕 value〔对应列的值〕**
- 数据修改语句如下：
- 数据修改：**update <表名>set<列名>=<表达式>〔where<条件>〕**
- 数据删除语句如下：
- 数据删除：**delete〔from〕{表名|视图名}〔where子句〕**



## 17.4 创建数据库

- 在前面的学习中已经对数据库编程的基本知识有了一些基本介绍。在学习在Java中进行数据库操作之前，首先来学习一下如何创建数据库。这里以Access数据库和SQLServer数据库为例。



## 17.4.1 创建Access数据库

- 学习如何使用JDBC进行数据库开发之前，首先需要建立一个数据库。本节使用的是非常简单易用的Access创建的数据库。例如创建一个记录学生信息的数据库，如表所示。

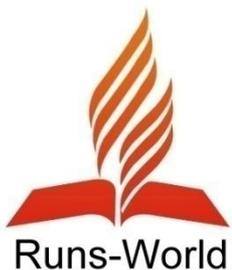
学生信息表

| 表名             | 字段名          | 字段类型  | 字段含义 | 备注       |
|----------------|--------------|-------|------|----------|
| <u>student</u> | <u>sid</u>   | 整数数值  | 学生编号 | sid字段为主键 |
|                | <u>sname</u> | 字符串文本 | 学生姓名 |          |
|                | <u>sage</u>  | 整数数值  | 学生年龄 |          |
|                | <u>stel</u>  | 字符串文本 | 联系电话 |          |

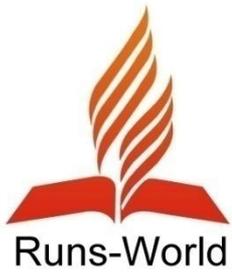


## 17.5 JDBC编程步骤

- 前面四节的介绍，都是进行JDBC数据库编程的基础。在本节就来对如何进行JDBC编程进行详细的讲解。JDBC编程是有严格的步骤的，在该节来就一步一步的学习如何进行JDBC数据库编程。

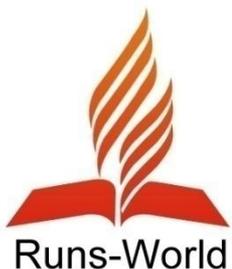


- 从前面的介绍中可以看出，在使用JDBC连接特定的数据库之前首先要加载相应数据库的JDBC驱动类，本小节将向读者介绍如何在开发中加载各种数据库的JDBC驱动类。
- 下面的代码片段说明了如何加载特定数据库的JDBC驱动类。
- 1     try
- 2     {
- 3         //通过Class类的forName方法加载指定的JDBC驱动类
- 4         Class.forName("驱动类全称类名");
- 5     }
- 6     catch(java.lang.ClassNotFoundException e)
- 7     {
- 8         e.printStackTrace();
- 9     }

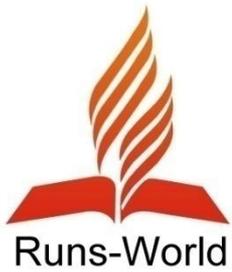


## 17.5.3 建立数据库连接

- 加载驱动程序后，就可以进行建立数据库连接。建立数据连接是通过调用 `java.sql.DriverManager` 类的 `getConnection` 方法来建立的，下面对该方法进行介绍。
- `public static Connection getConnection(String url, String user, String password) throws SQLException`
- 参数 `url` 为指定数据库的连接字符串，参数 `user` 为要连接数据库的用户名，参数 `password` 为用户名对应的密码。如果没有用户名与密码，可以用两个空字符串来代替。此方法有可能抛出捕获异常 `java.sql.SQLException`，因此在调用此方法时必须进行异常处理。
- 指定数据库的连接字符串由三部分组成，各部分之间用“:”分隔，如下所列。
- `jdbc:<子协议>:<子名称>`

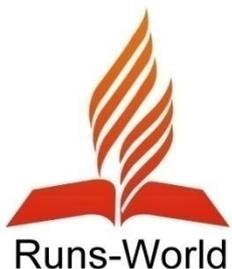


- 子协议指的是数据库的类型，例如可以是odbc，mysql或oracle等。子名称指的是数据源的名称或数据库的网络标识字符串。
- 例如，下面的代码片段以连接前面创建的ODBC数据源student为例，说明了如何获取与关闭数据库连接。
- 1 //声明连接引用
- 2 Connection con=null;
- 3 //创建数据库连接字符串
- 4 String url="jdbc:odbc:student";
- 5 try
- 6 {
- 7 //创建数据库连接
- 8 con=DriverManager.getConnection(url;""; "");
- 9 //连接以后操作数据库的代码
- 10 }
- 11 catch(java.sql.SQLException e)
- 12 {
- 13 e.printStackTrace();
- 14 }
- 上述代码中第8行通过DriverManager类的getConnection方法获取了数据库连接，在该程序url中给出的子协议为odbc，数据源为前面所建立的student数据源。

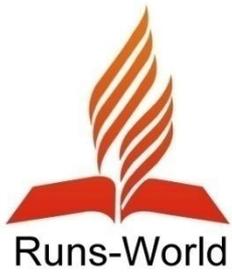


## 17.5.4 进行数据库操作

- 在上一节中讲解了如何建立数据库连接，当成功地创建了数据库连接后，就可以使用连接对象提供的createStatement方法来进行数据库操作，这里的数据库操作是通过SQL语句来完成的，下面给出了该方法的签名。
- `public Statement createStatement() throws SQLException`
- 创建了语句对象后就可以调用语句对象的executeUpdate方法来执行对数据库进行更新的语句了，下面给出了该方法的签名：
- `public int executeUpdate(String sql) throws SQLException`
- 参数sql为要执行的sql语句对应的文本字符串，如“`insert into student values('200801','Tom','23','123456789');`”。该方法返回值表示成功地操作了多少条数据库记录。同样该方法也是可以发生异常的，所以也需要进行异常处理。



- 例如，下面的代码片段说明了如何创建语句对象与执行更新数据库的SQL语句。
- ```
1  try
2  {
3      //获取Statement对象
4      Statement stat=con.createStatement();//con为一个指向Connection对象的引用
7      //插入一条记录
8      stat.executeUpdate("insert into student values('200801','Tom', '23','123456789');");
9      //关闭语句对象
10     stat.close();
11 }
12 catch(java.sql.SQLException e)
13 {
14     e.printStackTrace();
15 }
```
- 在该程序中首先是使用createStatement方法创建了一个Statement对象，然后使用创建后的Statement对象调用executeUpdate方法来对数据库进行操作。executeUpdate方法的参数为一个SQL参数，将按照该SQL进行数据库操作。

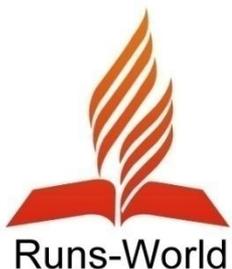


17.5.5 获取数据库中信息

- 在上一小节中学习了如何进行数据库操作，并进行了如何向数据库中插入记录。有时候经常需要来判断是否插入成功，这时候就需要来获取数据库中的信息。获取数据库中的信息是通过调用Statement对象的executeQuery方法来执行查询数据库的SQL语句。
 -
- `public ResultSet executeQuery(String sql) throws SQLException`
- 参数sql为要执行的SQL查询语句，例如“select * from student”，返回值为ResultSet类型的对象引用。ResultSet类型的对象中封装了查询的结果，可以调用其next方法移动指向结果集中记录的游标，下面给出了该方法的签名：
- `public boolean next() throws SQLException`



- 该方法是获取数据库信息的最重要的方法。同时，**ResultSet**中还提供了很多获取当前记录指定字段值的**getXxx**方法，如表17-2所列。
- 在所有的**getXxx**方法中都有两个重载的方法，其中一个接收**int**类型的参数，参数值为要获取值的字段对应的列索引，例如“**getString(1)**”将返回第一列的字符串内容。另一个接收**String**类型的参数，参数值为要获取值的字段对应的列名称，例如“**getString("sid")**”将获取**sid**列的字符串内容，并且所有的**getXxx**方法都有可能抛出**java.sql.SQLException**捕获异常，因此在调用时要进行异常处理。



17.6 事务处理

- 对数据库进行并发操作时，为了避免由于并发操作带来的问题，一般要将同一个任务中对数据库的增、删、改、查操作编写到一个事务中，同一个事务中的所有操作要么全部执行成功，要么都不执行。因此JDBC也提供了对事务开发的支持，本节将向读者介绍JDBC中有关事务开发的知识。

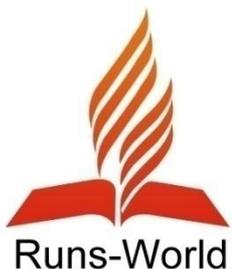


17.6.1 事务介绍

- 事物是SQL中的单个逻辑工作单元，一个事务内的所有语句被作为整体执行，遇到错误时，可以回滚事务，取消事务所作的所有改变，从而可以保证数据库的一致性和可恢复性。
- 一个事务逻辑工作单元必须具有以下四种属性，包括原子性、一致性、隔离性和永久性。原子性是指一个事务必须作为一个原子单位，它所作的数据库修改操作要不全部执行，要不全部取消。一致性是指当事务完成后，数据必须保证处于一致性的状态。隔离是指一个事务所作的修改必须能够跟其他事务所作的修改分离开来，以免在并发处理时，发生数据错误。永久性是指事务完成后，它对数据库所作的修改应该被永久保持。

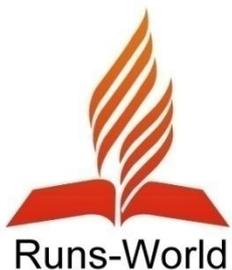


- 进行事务操作主要使用**Connection**对象中的三个方法。
setAutoCommit方法是指将此连接的自动提交模式设置为给定状态。如果参数**autoCommit**的值为**true**，则每执行一句**SQL**命令将自动被作为单个事务提交；否则，其将所有**SQL**语句聚集到一个事务中，直到调用**commit**方法或**rollback**方法为止，其默认值为**true**。
- **commit**方法是指提交当前的事务，并自动开始下一个事务。执行此方法后，会释放此**Connection**对象当前持有的所有数据库锁，同时要注意此方法只应该在自动提交模式被禁用的情况下使用。
- **rollback**方法是指回滚当前的事务，并自动开始下一个事务。执行此方法后，会释放此**Connection**对象当前持有的所有数据库锁，同时要注意此方法只应该在自动提交模式被禁用的情况下使用。



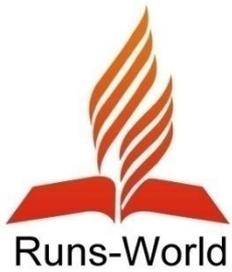
17.7 综合练习

- 1. 简要说明JDBC编程步骤。
- **【提示】**由于JDBC的编程步骤比较多，有些读者是很难全部记下来的，经常会由于没有做其中一步而造成程序发生错误。所以这里再来简单的说一下JDBC的编程步骤。
- (1) 创建数据库
- 因为很多数据库操作都是指定某一个数据库来进行操作的，所以在进行数据库编程前要首先建立一个数据库。如何创建库是因为数据库的不同而不同的。读者可以根据需求来相应的学习特定的一种数据库。
- (2) 创建数据源
- 创建数据源是数据库操作中最容易丢的一步，也是最难发现的一步。很多读者发现程序发生错误会，都会首先去找程序中有什么错误，从而很难发现没有创建数据源的错误。创建数据源中将为数据源起一个名字，读者在进行该步操作时一定要起一个容易记住的名字，因为在程序中还要用到该名称。
- (3) 编写程序
- JDBC数据库编程剩下的步骤都是在编写程序中的。在编写程序中也是不能丢掉必要步骤的。首先要加载驱动，根据数据库的不同而加载不同的驱动。然后就是建立数据库连接，接下来就是使用SQL语句进行数据库操作。最后就是获取信息。



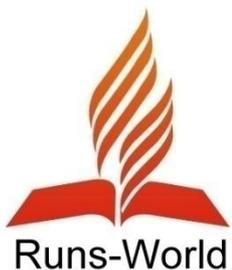
17.8 小结

- 在本章中首先对数据库、JDBC和SQL进行了简单的介绍，这些都是Java中进行数据库编程的基础。该书中对这些知识的讲解是很简单的。



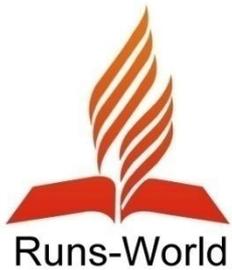
第18章 Java中输出输入流

- 在学习输出输入流之前，先来了解一下实际中的管道的问题。管道通常是可以双向流通的，Java中的流也是这样的，有输出流和输入流。管道是可以输送石油，也是可以输送水的，Java中的流也是可以输送不同的东西的，包括字节和字符等内容。Java中的输出输入流又称为IO流。通过本章的学习，读者应该能够完成如下几个目标。
- 了解什么是IO流。
- 掌握流的分类。
- 熟练掌握流如何进行文件操作。



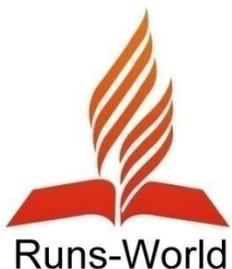
18.1 IO流简介

- 学习如何使用I/O流进行输入/输出操作之前，首先应当了解I/O流的基本原理与分类，这样才能恰当的运用各种I/O流进行不同的输入/输出操作。



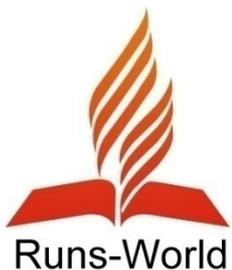
18.1.1 什么是IO流

- 数据流是形象的概念，可以理解为是一种“数据的管道”。管道中流动的东西可以是基于字节，也可以是基于字符的等。就好像管道里面可以流动水，也可以流动石油一样，当程序需要读取数据的时候，就会开启一个通向数据源的管道，这个数据源可以是存放在硬盘中的文件，也可以是内存中的数据，或是网络上的数据。
- Java中的数据流分为2种，一种是字节流，另一种是字符流。这两种流主要由4个抽象类来表示：InputStream, OutputStream, Reader, Writer，输入输出各两种。其中InputStream和OutputStream表示字节流，Reader和Writer表示字符流，其他各种各样的流均是继承这4个抽象类而来的。



18.1.2 节点流与处理流

- 根据流功能层次的不同可以将其分为两类：节点流（Node Streams）与处理流（Processing Streams），下面列出了这两种流的异同。
- 节点流一般用于直接从指定的位置进行读/写操作，例如磁盘文件、内存区域、网络连接等，其中一般只提供了一些基本的读写操作方法，功能比较单一。
- 处理流往往是用于对其他输入/输出流进行封装，对内容进行过滤处理，其中一般提供了一些功能比较强大的读写方法。



- 实际应用中，通常是将节点流与处理流二者结合起来使用。节点流直接与指定的源或目标相连，例如某个文件、某个网络连接等。而处理流则对节点流或其他处理流进一步进行封装，提供更丰富的输入/输出操作能力，例如缓冲、按字符串行读写等。

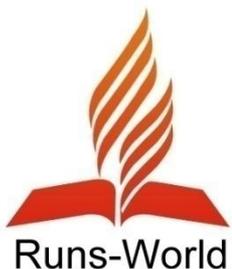


18.1.3 字节流与字符流

- 根据流处理数据类型的不同也可以将其分为两类：字节流与字符流，下面列出了这两种流的不同之处。
- 字节流以字节为基本单位来处理数据的输入/输出，一般都用于对二进制数据的读写，例如声音、图象等数据。
- 字符流以字符为基本单位来处理数据的输入和输出，一般都用于对文本类型数据的读写，例如文本文件、网络中发送的文本信息等。
- 表列出了Java I/O中字节流与字符流的四个抽象基类，Java I/O中的其他字节流与字符流都派生自这四个抽象基类。

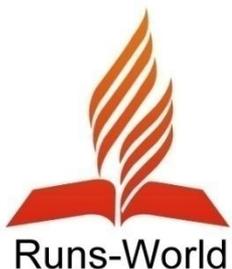
字节流与字符流的抽象基类

	字节流	字符流
输入流	InputStream	Reader
输出流	OutputStream	Writer



18.1.4 抽象基类

- 字节流和字符流的抽象基类包括InputStream类、OutputStream类、Reader类和Writer类组成，在后面将要讲到的流类都是继承这几个抽象基类的。
- 1. InputStream类:InputStream类是一个输入流，同样也是一个字节流。InputStream类是表示字节输入流的所有类的超类。其中定义了一些基本的读取字节数据流的方法，由其子类继承并扩展。

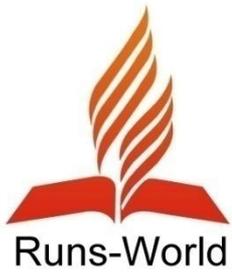


- `public int available() throws IOException` 获取可以从此输入流读取（或跳过）的字节数
- `public void close() throws IOException` 关闭输入流，同时释放系统资源
- `public void mark(int readlimit)` 在输入流中标记位置，参数 `readlimit` 为位置参数
- `public boolean markSupported()` 用于测试该输入流是否支持 `mark` 和 `reset` 方法
- `public abstract int read() throws IOException` 从输入流中读取下一个数据字节
- `public int read(byte[] b) throws IOException` 从输入流中读取字节数据，并存入字节数组 `b` 中
- `public int read(byte[] b, int off, int len) throws IOException` 从输入流中读取 `len` 个数据字节，并存入字节数组 `b` 中
- `public void reset() throws IOException` 将此流重新定位到最后一次调用 `mark` 方法时所处的位置
- `public long skip(long n) throws IOException` 跳过并且放弃输入流中的 `n` 个字节数据
- 使用 `InputStream` 类是可能发生 `IOException` 异常的，所以在使用时是要进行异常处理的。



OutputStream类

- **OutputStream**类是一个输出流，同样也是一个字节流。**OutputStream**类是表示输出字节流的所有类的超类。输出流接收输出字节并将这些字节发送到某个接收器。
- **public void close() throws IOException**关闭输出流，同时释放系统资源
- **public void flush() throws IOException**刷新输出流，同时输出所有缓冲的输出字节
- **public void write(byte[] b) throws IOException**将字节数组**b**中的**b.length**个字节数据输入到输出流
- **public void write(byte[] b,int off,int len) throws IOException**将字节数组**b**中的**len**个字节数据（从偏移量**off**开始）输入到输出流
- **public abstract void write(int b) throws IOException**将指定的字节输入到输出流
- 同样使用**InputStream**类是可能发生**IOException**异常的，所以在使用时是要进行异常处理的。



Reader类

- **Reader**类是一个输入流，同样也是一个字符流。**Reader**类是所有输入字符流的超类，也就是说所有的输入字符流都派生自**Reader**类，其中提供了很多关于字符流输入操作的方法，
- **public abstract int read()throws IOException**读取单个字符，返回值的低16比特存放读取字符的编码（0~65535），高16比特忽略，如果因为已经到达流末尾而没有可读的字符，则返回值-1。往往都会将此方法的返回值强制类型转换成**char**类型
- **public int read(char[] cbuf)throws IOException**从输入流中读取一定数量的字符，并将其存储在缓冲区字符数组**cbuf**中，以整数形式返回实际读取的字符数。若流中实际可读的字符数小于数组**cbuf**的长度，则返回值会小于数组**cbuf**的长度，否则返回值等于数组**cbuf**的长度



- `public abstract int read(char[] cbuf,int off,int len)throws IOException`将输入流中最多len个字符读入字符数组cbuf，将读取的第一个字符存储在元素cbuf[off]中，下一个存储在cbuf[off+1]中，依次类推，方法的返回值为实际读取的字符数。若流中实际可读的字符数小于len，则返回值会小于len，否则返回值等于len
- `public long skip(long n)throws IOException`跳过此输入流中的指定数量的字符，参数n为指定的数量，返回值为实际跳过的字符数
- `public boolean ready()throws IOException`判断此字符输入流是否准备好被读，若是则返回true，否则返回false
- `public void close()throws IOException`关闭此输入流并释放与该流关联的所有系统资源
- `public void mark(int readAheadLimit)throws IOException`在此输入流中标记当前的位置，参数readAheadLimit指出从此标记位置开始此流可以记忆的最大字符数，未来调用reset方法可以回到标记处重新读取字符。没有使用mark方法设置标记的流是不能回到某处再次读取的
- `public void reset() throws IOException`将此流重新定位到最后一次对此流调用mark方法时的位置
`public boolean markSupported()`测试此输入流是否支持mark和reset方法，支持则返回true，否则返回false



- **public void mark(int readAheadLimit) throws IOException**在此输入流中标记当前的位置，参数readAheadLimit指出从此标记位置开始此流可以记忆的最大字符数，未来调用**reset**方法可以回到标记处重新读取字符。没有使用**mark**方法设置标记的流是不能回到某处再次读取的
- **public void reset() throws IOException**将此流重新定位到最后一次对此流调用**mark**方法时的位置**public boolean markSupported()**测试此输入流是否支持**mark**和**reset**方法，支持则返回**true**，否则返回**false**
- 同样使用**Reader**类是可能发生**IOException**异常的，所以在使用时是要进行异常处理的。

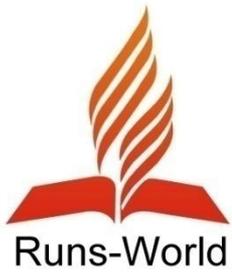


Writer类

- **Writer**类是一个输出流，同样也是一个字符流。**Writer**类是所有输出字符流的超类，也就是说所有的输出字符流都派生自**Writer**类，其中提供了很多关于字符流输出操作的方法，
- **public void write(int c) throws IOException**将指定的字符写入此输出流，参数**c**表示要写入的字符。要注意的是，**c**的低**16**个比特被作为一个字符写入流，而高**16**个比特被忽略
- **public void write(char[] cbuf) throws IOException**将指定字符数组**cbuf**的内容写入输入流
- **public abstract void write(char[] cbuf,int off,int len) throws IOException**将指定字符数组**cbuf**中从偏移量**off**开始的**len**个字符写入此输出流

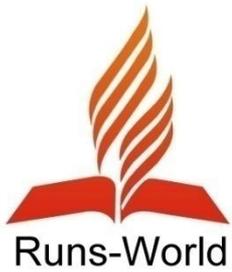


- `public void write(String str) throws IOException`向流中写入指定字符串`str`的各个字符
- `public void write(String str,int off,int len) throws IOException`将指定字符串`str`中从偏移量`off`开始的`len`个字符写入此输出流
- `public void flush() throws IOException`刷新此输出流并强制写出所有缓冲中的输出字符
- `public void close() throws IOException`关闭此输出流并释放与此流有关的所有系统资源
- 同样使用**Reader**类是可能发生**IOException**异常的，所以在使用时是要进行异常处理的。字节流与字符流提供的方法很相似的。它们的不同主要体现在操作的基本单位不同，一个是以字节为基本单位，另一个是以字符为基本单位。
- 注意：字节流和字符流的不同主要体现在操作的基本单位不同，一个是以字节为基本单位，另一个是以字符为基本单位。



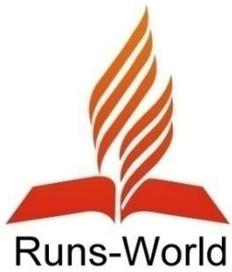
18.2 使用流进行文件操作

- 使用流来进行文件操作是流应用中最常见的应用之一。使用流能够读取文件内容，同样也能够写入文件内容。进行文件操作的类包括File、FileInputStream、FileOutputStream、FileReader、FileWriter等几个类。

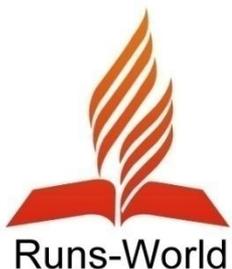


18.2.1 使用File类进行文件与目录操作

- Java中专门提供了一个表示目录与文件的类——`java.io.File`，通过其可以获取文件、目录的信息，对文件、目录进行管理。`File`类一共提供了四个构造器，表列出了其中常用的三个。
- 在**File**类中最常用的是第一个构造函数。使用构造函数**`public File(String pathname)`**创建一个文件对象。其中，如果**`pathname`**是实际存在的路径，则该**File**对象表示的是目录；如果**`pathname`**是存在的文件名，则该**File**对象表示的是文件。

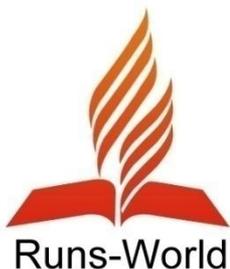


- **public File(String pathname)**通过指定的路径名字符串**pathname**创建一个**File**对象，如果给定字符串是空字符串，那么创建的**File**对象将不代表任何文件或目录
- **public File(String parent,String child)**根据指定的父路径名字符串**parent**以及子路径字符串**child**创建一个**File**对象。若**parent**为**null**，则与单字符串参数构造器效果一样，否则**parent**将用于表示目录，而**child**则表示该目录下的子目录或文件
- **public File(File parent,String child)**根据指定的父**File**对象**parent**及子路径字符串**child**创建一个**File**对象。若**parent**为**null**，则与单字符串参数构造器效果一样，否则**parent**将用于表示目录，而**child**则表示该目录下的子目录或文件

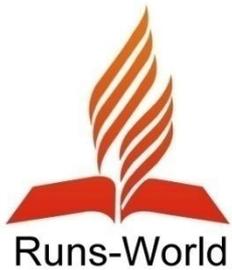


18.2.2 FileInputStream类与FileOutputStream类

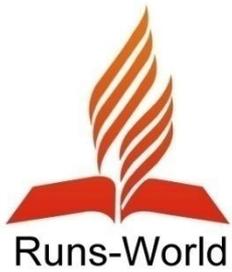
- FileInputStream类是InputStream的子类。FileInputStream类主要用于从文件系统中的某个文件中获取输入字节。FileOutputStream类是OutputStream的子类，FileOutputStream主要是用于将数据以字节流写入目标文件的输出流。表为FileInputStream中的方法声明及其使用描述。
- **FileInputStream**类从文件系统中的某个文件中获取输入字节。主要用于读取诸如图像数据之类的原始字节流。要读取字符流，可以考虑使用**FileReader**。



- `int available()` 返回可以从该文件输入流中读取的字节数
- `close()` 关闭文件输入流，同时释放系统资源
- `finalize()` 确认文件输入流不在被引用的状态，并可以调用 `close` 方法
- `getChannel()` 返回与该文件输入流有关的 `FileChannel` 对象
- `getFD()` 返回连接到文件系统（正被该 `FileInputStream` 使用）中实际文件的 `FileDescriptor` 对象
- `read()` 从该输入流中读取一个数据字节，并以 `int` 类型返回
`read(byte[] b)` 读取输入流中 `b.length` 个字节的数据，并存入字节数组 `b` 中
- `read(byte[] b, int off, int len)` 读取输入流中从偏移量 `off` 开始的 `len` 个字节的数据，并存入字节数组 `b` 中
- `skip(long n)` 跳过输入流中的数据，同时丢弃 `n` 个字节的数据

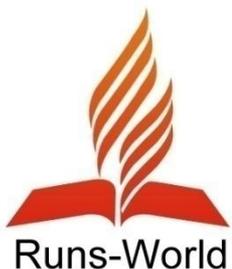


- 与FileInputStream类相对，FileOutputStream类用于将数据写入File或FileDescriptor的输出流。主要用于写入诸如图像数据之类的原始字节的流。要写入字符流，可以考虑使用FileWriter。
- finalize()断开到文件的连接，确认该文件输出流不处在被引用状态，并可以调用close方法
- ChannelgetChannel()返回与该文件输出流有关的FileChannel对象
- getFD()返回与该流有关的文件描述符FileDescriptor对象
- write(byte[] b)将b.length个字节的数据从指定字节数组写入到该文件输出流中
- write(byte[] b, int off, int len)将指定字节数组中的部分数据（从偏移量off开始的len个字节）写入到该文件输出流
- write(int b)将指定字节b写入到该文件输出流



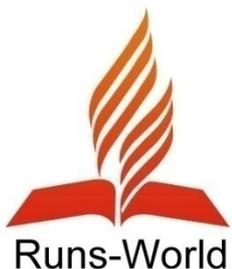
18.2.3 FileReader类与FileWriter类

- FileReader与FileWriter和FileInputStream与FileOutputStream类似，所不同的是它们是针对字符进行操作，而不是字节。这两个类的间接父类是字符流Reader和Writer。其中，FileWriter是用于写入字符文件的便捷类。FileReader是用于读取字符文件的便捷类。在FileReader类及FileWriter类中未自定义方法，继承了其父类及间接父类中的方法。



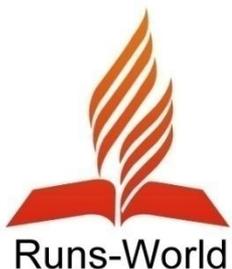
18.3 综合练习

- 字节流和字符流主要区别有哪些？
- **【提示】**字节流是最基本的，所有的“InputStream”和“OutputStream”的子类都是字节流，其主要用于处理二进制数据，并按字节来处理。
- 但是实际开发中很多的数据是文本，这就提出了字符流的概念，它按虚拟机的encode来处理，也就是要进行字符集的转化。这两者之间通过“InputStreamReader”和“OutputStreamWriter”来关联。实际上，通过“byte[]”和“String”来关联在实际开发中出现的汉字问题，这都是在字符流和字节流之间转化不统一而造成的。在从字节流转化为字符流时，实际上就是“byte[]”转化为“String”。
- `public String(byte bytes[], String charsetName)`
- 注意：有一个关键的参数字符集编码，通常可以省略，就是操作系统的lang。
- 字符流转化为字节流，实际上是String转化为byte[]。
- `byte[] String.getBytes(String charsetName)`
- 至于Java.io中还出现了许多其他的流，主要是为了提高性能和使用方便，如“BufferedInputStream”、“PipedInputStream”等。



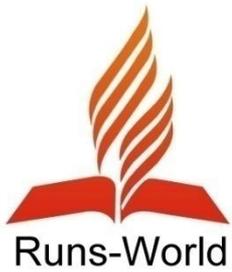
18.4 小结

- 在本章中主要对Java中的流进行了详细的讲解，同时也讲解了如何使用流进行文件操作。进行文件操作是流的最重要作用之一，这里对其进行详细的讲解。



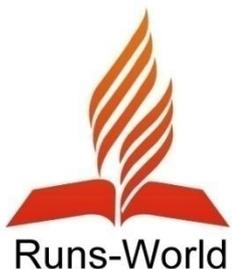
第19章 集合框架

- 在日常生活中，放衣服就是一门学问，是把衬衫和裤子等衣服都放在一个盒子中呢，还是每一件衣服放在一个盒子中呢，读者都知道这两种做法都是不好的，通常都是将一类衣服放在一起。同样，在Java中也提供了这样的功能，那就是集合框架。在前面已经学习了数组，集合框架也是和数组一样来保存一组数据。集合框架主要包括列表、集合和映射。通过本章的学习，读者应该完成如下几个目标。
- 了解什么是集合框架和集合框架包括哪些形式。
- 掌握什么是列表和列表中包括哪些类和接口。
- 掌握什么是集合和集合中包括哪些类和接口。
- 掌握什么是映射和映射中包括哪些类和接口。



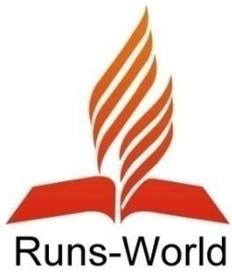
19.1 集合框架总论

- 集合是某一类对象的通称呼，这类对象代表以某种方式组合到一起的一组对象；它是将多个元素组合为一个单元的对象，用于存储，检索，操纵和传递数据。而对象的集合时，指的是对象引用的集合而不是对象的集合，在Java集合中只存储引用，对象在集合之外。集合框架提供用于管理对象集合的接口和类，它包括接口，实现和算法。

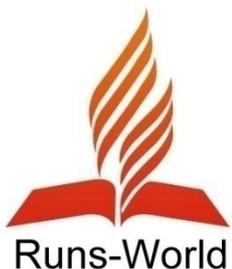


19.1.1 什么是集合框架

- 集合框架是Java提供的一些可以定义一个对象，该对象由其他的对象组成（如常见的向量（Vector）类）。集合框架是一个统一的可以代表及操作集合，并能对这些集合独立的进行操作的一些结构。集合框架的主要优点在于提高了编码效率、性能和复用性。
- Java集合框架提供了有效的数据结构及算法，因此程序员不需要自己去编写实现这些功能。Java集合框架提供了高性能的数据结构及算法的实现。因为对各个接口的实现是可以互换的，因此程序很容易可以换接口。提高了软件的复用性：软件可以提供标准的集合框架的接口对其进行操作。



- **Java**集合框架主要由一组用来操作对象的接口组成。不同接口描述一组不同数据类型。在这些接口中**Collection**接口是层次结构中的根接口。在**Collection**接口中具有开发中经常使用到的**set**接口、**list**接口和**map**接口。

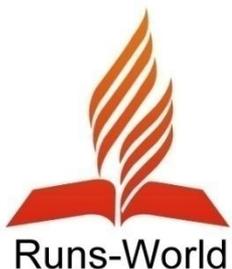


19.1.2 Collection接口

- 在上一小节中已经知道，Collection接口是集合继承树中最顶层的接口，该接口声明了集合中常用到的一些通用方法，在表中给出了这些方法。

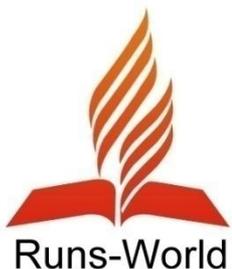
Collection类的方法名及说明

方法	说明
<code>boolean add(Object o)</code>	该方法添加参数指定的元素o至集合中。
<code>boolean addAll(Collection c)</code>	该方法将参数集合c中所有的元素都添加到该集合中。
<code>void clear()</code>	该方法将删除该集合中所有的元素。
<code>boolean contains(Object o)</code>	该方法用于判断元素o是否在该集合中，如果是则返回true，否则返回false。
<code>boolean containsAll(Collection c)</code>	该方法用于判断该集合中是否包含参数c集合中的所有元素，如果是则返回true，否则返回false。
<code>boolean equals(Object o)</code>	这个方法将比较该集合与参数指定的对象o。如果相等，返回true，否则返回false。
<code>int hashCode()</code>	该方法将返回集合的哈希码的整型表现形式的值。
<code>boolean isEmpty()</code>	判断该集合是否为空，如果为空则返回true，否则为false。
<code>Iterator iterator()</code>	返回该集合的迭代器，iterator，可以用户遍历集合中的元素。
<code>boolean remove(Object o)</code>	如果集合中存在参数指定的元素o，则将该元素从集合中删除。
<code>boolean removeAll(Collection c)</code>	将该集合中的某些元素删除，这些元素是参数c中存在的。
<code>boolean retainAll(Collection c)</code>	与removeAll(Collection c)方法相反，将集合中所有的非参数c中的元素删除，仅保留c中存在的元素。
<code>int size()</code>	返回该集合的大小，即其中的元素个数。
<code>Object[] toArray()</code>	将该集合中的元素，以数组的形式返回。数据的类型为Object。
<code>Object[] toArray(Object[] a)</code>	返回包含此collection中所有元素的数组；返回数组的运行时类型与指定数组的运行时类型相同。



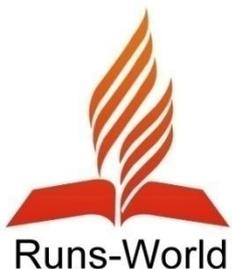
19.2 列表

- List列表作为集合中的一种，其主要特点在于其中的元素保持一定的顺序，并且元素是可以重复的。在本小节将具体讲解List的使用及其实现类（如ArrayList、LinkedList）的使用。List接口继承自Collection接口，代表列表的功能（角色），其中的元素可以按索引的顺序访问，所以也可以称之为有索引的Collection。实现该接口的类均属于Ordered类型，具有列表的功能，其元素顺序均是按添加（索引）的先后进行排列的。

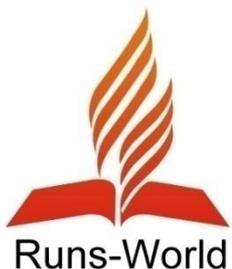


19.2.1 List列表接口

- 除了继承了Collection声明的方法外，List接口在iterator、add、remove、equals和hashCode方法的基础上加了一些其他约定，超过了Collection接口中指定的约定。同时，List比Collection多了10个方法，这些方法可以分为访问方法、迭代器方法、搜索方法和插入、删除方法。
- List接口声明了3种对列表元素进行定位（索引）访问方法：
- `Object get(int index)`：参数index表示将要需要得到元素的索引。该方法将返回此列表中指定index位置上的元素。
- `List subList(int fromIndex, int toIndex)`：参数fromIndex为指定的起始索引，参数toIndex为指定的结束索引，该方法将返回一个新的列表，这个新的列表将包含原来列表中从指定的起始索引到指定的结束索引并且不包含结束索引的元素。
- `Object[] toArray()`：该方法将列表转换成一个Object类型的对象数组，该数组用元素的顺序与列表中元素的顺序相同，并将该数组返回。



- **List**接口声明了特殊的迭代器，称为**ListIterator**，除了允许**Iterator**接口提供的正常操作外，该迭代器还允许元素插入和替换，以及双向访问。还提供了方法来获取从列表中指定位置开始的列表迭代器。**List**接口提供了对**ListIterator**的获取的两种方法，分别是**listIterator**方法和**listIterator(int index)**方法。
- **List**接口声明了两种搜索指定对象的方法。从性能的观点来看，应该小心使用这些方法。在很多实现中，这些方法将执行高开销的线性搜索。**List**接口对其声明如下：
- **int indexOf (Object o)**：参数**o**为指定查找的元素，该方法将遍历整个列表查找指定元素**o**，若列表中存在，则返回第一个找到的元素的索引，若列表中不存在，则返回负数。
- **int lastIndexOf (Object o)**：参数**o**为指定查找的元素，该方法将遍历整个列表查找指定元素**o**，若列表中存在，则返回最后一个找到的元素的索引，若列表中不存在，则返回负数。



- **List**接口声明了两种在列表的任意位置高效插入和删除元素的方法。
- **Object set (int index, Object element)**：参数**index**表示需要替换元素的索引。参数**o**表示将要替换为的元素。该方法操作成功后将返回替换掉的元素。
- **Object remove (int index)**：参数**index**为将要移除元素的索引。返回从列表中移除的元素。该方法操作后所有后续元素均向前移动，即列表中间不能有空位。
- **boolean remove (Object o)**方法：参数**o**为指定的需要移除的元素。若列表包含一个或多个与指定**o**相同的元素，则移除该元素，并返回 **true**，否则返回**false**。
- **boolean removeAll (Collection c)**方法：参数**c**为包含指定需要移除元素的 **Collection**，该方法将列表中有的并且**c**中有的元素从**set**中移除，若有元素被移除 **set**则返回**true**，否则返回**false**。
- **boolean retainAll (Collection c)**方法：参数**c**为包含指定需要保留元素的 **Collection**，该方法将列表中有的并且**c**中没有的元素从**set**中移除，若有元素被移除 **set**则返回**true**，否则返回**false**。
- **java.util.List**的几种实现中，有三种最为常用的实现类，这三个类分别是**Vector**类、**ArrayList**类和**LinkedList**类。接下来将逐个介绍。



19.2.2 Vector类

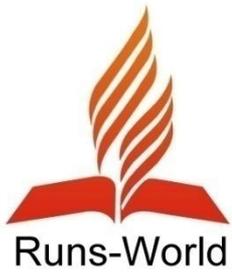
- Vector类也称为向量，从Java一诞生就有，后来被作为集合框架的一部分，其性能特点与ArrayList基本上是相同的。不同之处是该类的功能方法是同步的，同一时刻只能有一个线程访问，没有特殊需要，现在一般都使用ArrayList，ArrayList会在下一小节中讲解。

Vector类的构造器

构造器签名	功能
<code>public Vector()</code>	该构造器将构造一个空的Vector对象。该对象的初始容量为10。
<code>public Vector(int initialCapacity)</code>	参数initialCapacity表示指定的初始容量，该构造器将构造一个具有指定容量的空Vector对象。
<code>public Vector(int initialCapacity,int capacityIncrement)</code>	参数initialCapacity表示指定的初始容量，参数capacityIncrement表示当Vector满了时候，其容量的增量。该构造器将构造一个具有指定初始容量与指定增量的空Vector对象。
<code>public Vector(Collection c)</code>	参数c为包含指定元素的Collection。该构造器将构造一个以c中的元素为初始内容的Vector对象。

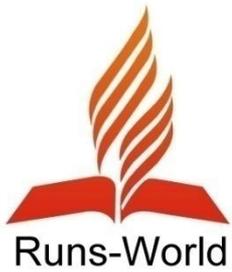


- **Vector**提供了用于增加元素的方法，方法如下所述。
- **public void addElement (Object obj)** 方法：该方法是将指定的组件添加到该向量的末尾，并将其大小增加1。如果向量的大小比容量大，则增大其容量。
- **public void addElement (int index, Object obj)** 方法：该方法在该向量的指定位置**index**插入指定的元素**obj**。并将当前位于该位置的元素及所有后续元素右移（即将元素索引加1）。如果索引超出范围（即**index < 0 || index > size()**），则程序抛出 **ArrayIndexOutOfBoundsException**异常。
- **public void insertElementAt (Object obj, int index)** 方法：该方法将指定对象作为此向量中的组件插入到指定的**index**处。



19.2.3 ArrayList类

- 本小节主要向读者介绍ArrayList类，其是List接口最常用的实现之一，可以向其中添加包括null值在内的所有对象引用型的元素，甚至该类对象引用自己也可以作为其中的元素，这样便可以方便的搭建一个树状结构的集合。
- ArrayList有三种构造方法，方法如下所示：
- `public ArrayList()` 方法：该构造器将构造一个空的ArrayList对象。该对象的初始容量为10。
- `public ArrayList(int initialCapacity)` 方法：参数 `initialCapacity` 表示指定的初始容量，该构造器将构造一个具有指定容量的空ArrayList对象。
- `public ArrayList(Collection c)` 方法：参数 `c` 为包含指定元素的Collection。该构造器将构造一个以 `c` 中的元素为初始内容的ArrayList对象。



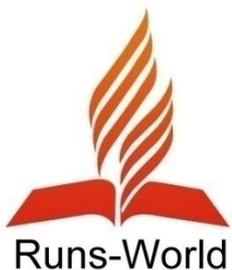
ArrayList类和**Vector**类一样，同样也具有很多的方法，这里不可能为每一种方法都给出程序

。

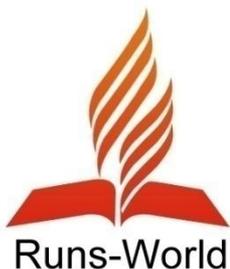
- `public boolean add(Object o)`该方法将指定的元素o追加到此列表的末尾
- `public void add(int index, Object o)`该方法将指定的元素o插入此列表中的指定index索引位置
- `public boolean addAll(Collection c)`该方法将对象c中的所有元素追加到此列表的末尾
- `public boolean addAll(int index, Collection c)`该方法将参数c中的所有元素从指定的index索引位置开始插入到此列表中
- `public void clear()`该方法删除列表对象中的所有元素
- `public Object clone()`该方法返回此ArrayList对象的复制对象，返回为Object对象
- `public boolean contains(Object elem)`该方法用于判断此列表中是否包含指定的元素elem，如果包含则返回true
- `public void ensureCapacity (int minCapacity)`该方法用于增加此ArrayList对象的容量，以确保它至少能够容纳最小容量参数所指定的元素数
- `public E get(int index)`该方法返回此列表对象中指定索引index位置上的元素



- `public int indexOf(Object elem)`该方法将返回给定参数`elem`元素第一次出现的位置
- `public boolean isEmpty()`该方法将判断此列表中是否没有元素
- `public int lastIndexOf(Object elem)`该方法返回指定的对象`elem`元素在列表中最后一次出现的位置索引
- `public Object remove(int index)`该方法删除此列表中指定位置`index`上的元素
- `public boolean remove(Object o)`该方法从此该列表中删除指定元素`o`
- `protected void removeRange(int fromIndex,int toIndex)`该方法删除列表中索引在`fromIndex`和`toIndex`之间的所有元素。包括`fromIndex`，不包括`toIndex`
- `public Object set(int index,Object o)`该方法用指定的元素替代此列表中指定位置上的元素
- `public int size()`该方法返回此列表中的元素数
- `public Object[] toArray()`该方法返回一个此列表中所有元素的数组
`public <T> T[] toArray(T[] a)`该方法返回一个此列表中所有元素的数组，返回的数组存在在参数`a`中
- `public void trimToSize()`该方法将此`ArrayList`实例的容量调整为列表的当前大小



- **ArrayList**类中提供了可以删除其中元素的方法，其方法声明及使用说明如下所示。
- **public E remove (int index)** 方法：该方法删除列表中参数指定位置**index**上的元素。向左移动所有后续元素，即将其索引减**1**。其中参数**index**为要删除元素的索引。
- **public boolean remove (Object o)** 方法：该方法删除列表指定元素**o**。如果列表中包含指定的元素，则返回**true**，否则为**false**。
- **public void clear()**方法：该方法删除列表中所有的元素。调用这个方法后，列表为空。



19.2.4 LinkedList类

- 本小节主要介绍LinkedList类，其功能与ArrayList、Vector相同，都是列表（List）的实现。其内部是依赖双链表来实现的，因此具有很好的插入删除性能，但随机访问元素的性能相对较差，适合用在插入、删除多，元素随机访问少的场合。表列出了LinkedList类的几个构造器。

LinkedList类的构造器

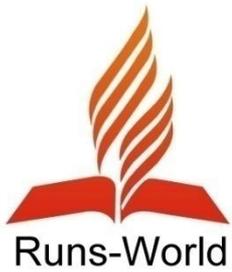
构造器签名	功能
<code>public LinkedList()</code>	该方法将构造一个空的列表。
<code>public LinkedList(Collection c)</code>	构造一个包含指定集合c中的所有元素的列表，这些元素按其集合的迭代器返回的顺序排列；其中参数c为要将其元素放到此集合中的列表；如果指定的集合为null，程序抛出NullPointerException异常。



- **LinkedList**类中同样具有很多方法。在前面学习**Vector**类和**ArrayList**类时，已经使用添加和删除方法进行了举例，这里就类使用**LinkedList**类中提供了获取或修改某个位置的元素方法来举例。获取和删除方法包括如下几个方法。
- **public E element()**方法：该方法找到但不删除此列表的头（即第一个元素），这个方法是实现了接口**Queue**中的同名方法。
- **public E get (int index)**方法：该方法返回此列表中指定位置**index**处的元素。如果指定的索引超出范围（**index<0||index>=size()**），则程序抛出**IndexOutOfBoundsException**异常。
- **public E getFirst()**方法：该方法返回此列表的第一个元素。如果此列表为空，则程序抛出**NoSuchElementException**异常。

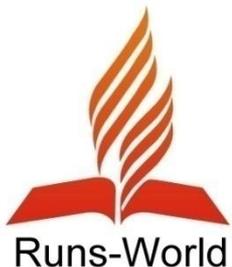


- **public E getLast():** 该方法返回此列表的最后一个元素。如果此列表为空，则程序抛出 **NoSuchElementException** 异常。
- **public int indexOf (Object o)** 方法：该方法返回此列表中元素 **o** 的索引，需要注意的是，如果包含多个 **o**，则返回为第一次出现的索引值。如果列表中不包含此元素，则返回 **-1**。这个方法是覆盖其父类 **AbstractList<E>** 中的 **indexOf** 方法。其中参数 **o** 为要搜索的元素。
- **public int lastIndexOf (Object o)** 方法：该方法返回此列表中最后出现的指定元素的索引，如果列表中不包含此元素，则返回 **-1**。参数 **o** 为要搜索的元素。
- **public E peek()** 方法：该方法找到但不删除此列表的头，即第一个元素。如果此队列为空，则返回 **null**。
- **public E poll()** 方法：该方法找到并删除此列表的头，即第一个元素。如果此队列为空，则返回 **null**。
- **public E set (int index, E element)** 方法：该方法将此列表中指定位置的元素替换为指定的元素。其中参数 **index** 为要替换的元素的索引，**element** 为要在指定位置存储的元素。



19.3 集合

- Set集合是一种不包含重复元素的Collection，即任意的两个元素e1和e2比较，结果都不相等。注意：Set的构造函数有一个约束条件，传入的Collection参数不能包含重复的元素。必须小心操作可变对象（Mutable Object）。如果一个Set中的可变元素改变了自身状态，`Object.equals(Object)=true`会发生一些问题。

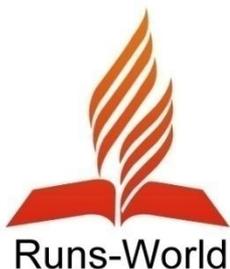


19.3.1 Set接口

- Set接口与List接口最大的区别在于：Set中没有重复的元素。Set是非常简单的集合，Set中的对象没有特定顺序。Sorted接口具有排序的功能，TreeSet类则是实现了该接口；HashSet类使用哈希算法存取集合中的元素，存取速度比较快。Set接口声明如下所示：
- public interface Set<E>extends Collection<E>
- 提示：Set接口与List接口最大的区别在于：Set中没有重复的元素。

Set接口的方法

方法修饰符	方法名	方法描述
boolean	add(E o)	将参数指定的元素o添加至集合中，如果该集合中已经存在该元素，则该集合不变化。
boolean	addAll(Collection<? extends E> c)	如果该集合中不存在所有的元素，将参数指定c集合中的所有元素添加至集合中。
void	clear()	该方法删除集合set中的所有元素。
boolean	contains(Object o)	该方法判断参数指定的元素o是否存在于集合中，如果存在则返回true，否则为false。
boolean	containsAll(Collection<?> c)	该方法判断参数c中所有的元素是否存在于集合中，如果存在则返回true，否则返回false。
boolean	equals(Object o)	该方法用于比较指定的对象o是否与集合相等，如果相等，则返回true，否则返回false。
int	hashCode()	该方法将返回该集合的哈希码的整数形式的值。
boolean	isEmpty()	该方法将判断该集合是否为空，如果为空则返回true，否则为false。
Iterator<E>	iterator()	该方法返回该集合的迭代器iterator，可以用于遍历集合中的元素。
boolean	remove(Object o)	该方法将删除集合中的参数指定的元素o，如果集合中不存在元素，则该集合不变化。
boolean	removeAll(Collection<?> c)	该方法删除集合中与参数集合c中元素相同的所有元素。
boolean	retainAll(Collection<?> c)	与方法retainAll(Collection<?> c)的作用正好相反，该方法仅保留参数指定c的元素。该集合中的其他元素都被删除。
int	size()	返回该集合的元素个数。即其容量。
Object[]	toArray()	将该集合中的元素以数组的形式返回，数组类型为Object。
<T>T[]	toArray(T[] a)	该方法返回一个包含set中所有元素的数组；返回数组的运行时类型是指定数组的类型。



19.3.2 SortedSet接口

- SortedSet继承自Set接口，所以该接口不但具有Set的所有功能，而且是一个Sorted类型的Set。也就是说，实现该接口的类将按元素的天然顺序自动排序，不管插入的顺序是什么，其总会按照元素的天然顺序进行遍历。表列出了该接口中的常用方法。

SortedSet接口中的常用方法

方法签名	功能
<code>Object first()</code>	该方法将返回SortedSet中第一个元素，也就是最小的元素。
<code>Object last()</code>	该方法将返回SortedSet中最后一个元素，也就是最大的元素。
<code>SortedSet headSet(Object toElement)</code>	参数toElement为一个指定的元素，该方法将返回一个包含所有小于指定元素并且不包含指定元素的SortedSet。
<code>SortedSet tailSet(Object fromElement)</code>	参数fromElement为一个指定的元素，该方法将返回一个包含所有大于指定元素并且包含指定元素的SortedSet。
<code>SortedSet subSet(Object fromElement, Object toElement)</code>	参数fromElement为指定的起始元素，参数toElement为指定的结束元素，该方法将返回一个包含从起始元素到结束元素中所有元素并且不包含结束元素的SortedSet。



19.3.3 TreeSet类

- TreeSet类是实现了接口SortedSet的类，已知SortedSet提供了集合元素的顺序存储，其中元素保持升序排列。此类保证排序后的set按照升序排列元素，根据使用的构造方法不同，可能会按照元素的自然顺序进行排序。
- TreeSet类提供了四种构造方法，这些方法声明及使用描述如下所示：
 - TreeSet()方法：该构造器将构造一个空的TreeSet对象。
 - TreeSet(Collection c)方法：参数c为包含指定元素的Collection。该构造器将构造一个以c中的元素为初始内容的TreeSet对象。
 - TreeSet(Comparator c)方法：参数c为指定的比较器，该构造器将构造具有指定比较器的空TreeSet对象。
 - TreeSet(SortedSet s)方法：参数s为包含指定元素的SortedSet。该构造器将构造一个以s中的元素为初始内容的TreeSet对象。

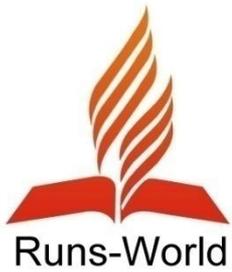


19.3.4 HashSet类

- HashSet类是Set接口最常用的实现之一，其既不是Ordered的也不是Sorted的，元素在其中存储不保证任何顺序。实际上，HashSet存储对象引用时是按照哈希策略来实现的。另外，可以向HashSet中添加null值，但只能添加一次。表列出了HashSet类的几个构造器。

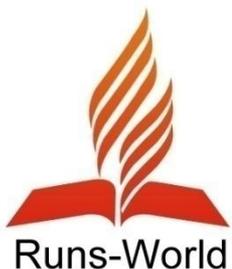
HashSet类的几个构造器

构造器签名	功能
<code>public HashSet ()</code>	该构造器将构造一个空的HashSet对象。该对象的初始容量为16。
<code>public HashSet (int initialCapacity)</code>	参数initialCapacity表示指定的初始容量，该构造器将构造一个具有指定容量的空HashSet对象。
<code>public HashSet (Collection c)</code>	参数c为包含指定元素的Collection。该构造器将构造一个以c中的元素为初始内容的HashSet对象。



19.4 映射

- Map（映射）是一个存储关键字和值的关联或者说是关键字/值对的集合。给定一个关键字，可以得到其相应的值。关键字和值都是对象。关键字必须是惟一的。但值是可以被复制的。而一个值对象可以是另一个Map，依次类推，这样就可形成一个多级映射。对于键对象来说，像Set一样，一个Map容器中的键对象不允许重复，这是为了保持查找结果的一致性。

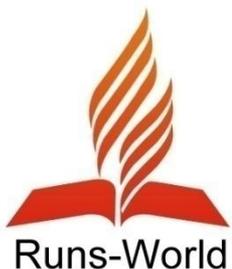


19.4.1 Map接口

- Map也可以称之为键/值集合，因为在实现了该接口的集合中，元素都是成对出现的，一个称之为键，另一个称之为值。键和值都是对象，键对象用来在Map中唯一的标识一个值对象。键对象在Map中不能重复出现，就像Set中的元素不能重复一样。
- 注意：与Collection系列的集合一样，系统并不真正把对象放到Map中，Map中存放的只是键和值对象的引用。

Map接口中的常用方法

方法签名	功能
<code>void clear()</code>	从此映射中删除所有映射关系。
<code>boolean containsKey(Object key)</code>	如果此映射包含指定键的映射关系，则返回true。
<code>boolean containsValue(Object value)</code>	如果此映射为指定值映射一个或多个键，则返回true。
<code>Object get(Object key)</code>	返回此映射中映射到指定键的值。
<code>boolean isEmpty()</code>	该方法将测试Map中是否还存在映射关系，若存在则返回false，否则返回true。
<code>Object put(Object key, Object value)</code>	将指定的键key与值value添加到Map中，并将其关联，如果之前有相同的key存在，则只将值value添加到Map中，并将其与之前相同的key关联。完成操作后将值返回。
<code>void putAll(Map t)</code>	参数t为包含需要添加键值对的Map，该方法将t中包含的元素添加进该方法所在的Map。
<code>Object remove(Object key)</code>	如果存在此键的映射关系，则将其从映射中删除。
<code>int size()</code>	该方法将返回Map中键值对的个数。

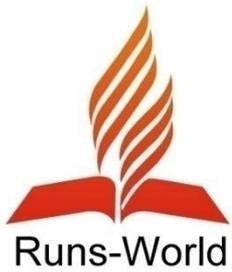


19.4.2 HashMap类

- HashMap类是基于哈希表的Map接口的实现。该类提供所有可选的映射操作，并允许使用null值和null键。但是此类不保证映射的顺序。
- HashMap的实例有两个参数影响其性能：初始容量和加载因子。容量是哈希表中桶的数量，初始容量只是哈希表在创建时的容量。加载因子是哈希表在其容量自动增加之前可以达到多满的一种尺度。当哈希表中的条目数超出了加载因子与当前容量的乘积时，通过调用rehash方法将容量翻倍。HashMap类声明如下所示：
- `public class HashMap<K, V> extends AbstractMap<K, V> implements Map<K, V>, Cloneable, Serializable`

HashMap类的构造器

构造器签名	功能
<code>public HashMap ()</code>	该构造器将构造一个空的HashMap对象，其初始容量为16。
<code>public HashMap (Map m)</code>	参数m为包含指定键值对的Map。该构造器将构造一个以m中的键值对为初始内容的HashMap对象。



- **HashMap**类可为基本操作（**get**和**put**）提供稳定的性能。迭代该集合视图所需的时间与**HashMap**实例的“容量”（桶的数量）及其大小（键-值映射关系数）之和成比例。所以，如果迭代性能很重要，则不要将初始容量设置得太高，或将加载因子设置得太低。

19.4.3 TreeMap类

- TreeMap类是SortedMap接口的基于红黑树的实现（红黑树是一种特定类型的二叉树，是一种自平衡二叉查找树，读者可以查阅相关资料）。此类保证了映射按照升序顺序排列关键字，根据使用的构造方法不同，可能会按照键的类的自然顺序进行排序（参见Comparable），或者按照创建时所提供的比较器进行排序。TreeMap类的声明如下所示：
- `public class TreeMap<K, V> extends AbstractMap<K, V> implements SortedMap<K, V>, Cloneable, Serializable`

TreeMap类的几个构造器

构造器签名	功能
<code>public TreeMap ()</code>	该构造器将构造一个空的TreeMap对象。
<code>public TreeMap (SortedMap s)</code>	参数s为包含指定键值对的SortedMap。该构造器将构造一个以s中的键值对为初始内容的TreeMap对象。
<code>public TreeMap (Map c)</code>	参数c为包含指定键值对的Map。该构造器将构造一个以c中的元素为初始内容的TreeMap对象。
<code>TreeMap(Comparator comparator)</code>	Comparator为指定的比较器，与TreeSet相同，如果想指定键的排序规则，则可以使用此构造器。

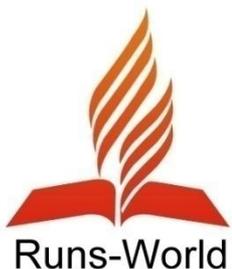


19.6 综合练习

- 1. 如何使用集合中的sort方法对集合中的元素进行排序?

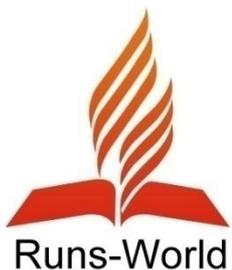
- **【提示】**

```
• 01 import java.util.*;
• 02 public class LianXil
• 03 {
• 04     public static void main(String[] args)
• 05     {
• 06         ArrayList al=new ArrayList();           //创建ArrayList对象
• 07         //随机创建10个整数, 并添加到集合中
• 08         for(int i=0;i<10;i++)
• 09         {
• 10             al.add(Integer.valueOf((int)(Math.random()*100)));
• 11         }
• 12         //打印初始化后ArrayList中的元素
• 13         System.out.println("排序前ArrayList中的元素");
• 14         System.out.println(al);
• 15         Collections.sort(al);                   //使用sort方法对元素进行排序
• 16         //打印排序后的ArrayList对象
• 17         System.out.println("排序后ArrayList中的元素");
• 18         System.out.println(al);
• 19     }
• 20 }
```



19.6 综合练习

- 2. 如何使用binarySearch方法搜索集合中的元素?
- **【提示】** 在使用binarySearch方法搜索元素之前, 首先要对元素进行排序。
- `import java.util.*;`
- `public class LianXi2`
- `{`
- `public static void main(String[] args)`
- `{`
- `ArrayList al=new ArrayList(); //创建ArrayList对象`
- `//随机创建10个整数, 并添加到集合中`
- `for(int i=0;i<100;i++)`
- `{`
- `al.add(Integer.valueOf((int) (Math.random()*100)));`
- `}`
- `Collections.sort(al); //使用sort方法对元素进行排序`
- `//使用binarySearch方法查找指定元素,`
- `int index=Collections.binarySearch(al, Integer.valueOf(15));`
- `if(index<0)`
- `{`
- `System.out.println("查找失败");`
- `}`
- `else`
- `{`
- `System.out.println("索引值为: "+index);`
- `}`
- `}`
- `}`



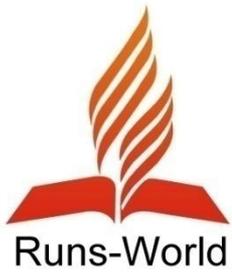
19.5 小结

- 在本章中主要对Java中的集合框架进行了详细的讲解。集合框架主要包括列表、集合和映射，同时在各个方面中还包括一些具体的接口和类。



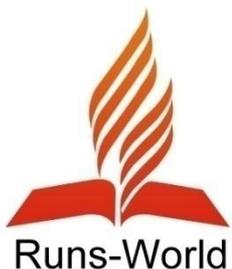
第20章 网络编程

- 随着Internet的逐渐普及，越来越多的软件项目应用涉及到网络。而Java似乎专门为网络设计的，它提供了很多方法供程序员使用，使大家能够很方便地开发网络应用软件。在本章中，将介绍Java网络编程的基础知识，以及网络编程的特点和方法。通过本章的学习，读者应该完成如下几个目标。
- 了解什么是协议，有哪些协议。
- 了解网络编程的模型。
- 熟练掌握使用Socket进行网络编程。



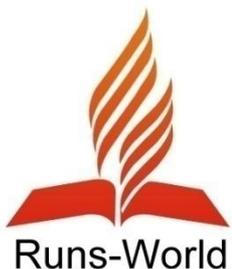
20.1 网络编程基础

- 简单来看，网络编程的目标就是计算机之间相互通讯数据。Java SDK提供了一系列API来完成这些工作，例如Socket。对于程序员来说，这些API被存放在java.net这个包里面，因此只要导入这个包就可以进行网络编程。

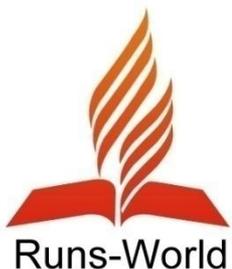


20.1.1 TCP/IP协议

- 现在的Internet或Intranet大部分都是使用TCP/IP协议进行网络通信的，实际上TCP/IP协议是一组以TCP与IP协议为基础的相关协议的集合。
- 注意：该协议并不完全符合OSI的七层参考模型，而是采用的4层结构。
- IP协议是TCP/IP协议族的核心，也是互连网络层中最重要的协议。其接收由更低层发来的数据包，并将该数据包发送到更高层，即TCP或UDP层；此外IP层也可以将从TCP或UDP层接收来的数据包传送到更低层。IP是面向无连接的数据报传送，所以IP协议将报文传送到目的主机后，无论传送正确与否都不进行检验、不会送确认以及不保证分组的正确顺序。

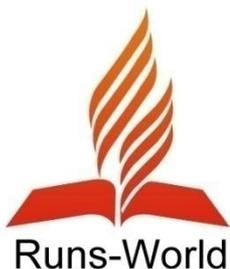


- **TCP**协议位于传输层，其提供面向有连接的数据包传送服务，保证数据包能够被正确传送与接收，包括内容的校验与包的顺序，损坏的包可以被重传。要注意的是，由于提供的是有保证的数据传送服务，因此传送效率要比没有保证的服务低，一般适合工作在广域网中，对网络状况非常好的局域网不是很合算。当然，是否采用**TCP**也取决于具体的应用需求。



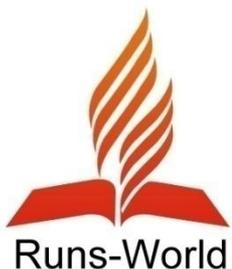
20.1.2 网络编程模型

- 对于网络编程来说，目前主要有两种编程模型，分别是C/S结构和B/S结构。C/S结构是指客户机/服务器结构。所谓客户机/服务器结构，指的是在客户端需要安装客户端软件，由客户端软件负责与服务器端的数据通讯，将任务合理分配到客户端和服务器端来实现，降低了网络的负载开销。
- B/S结构是指浏览器/服务器结构。客户端只需要安装有网页浏览器，不需要安装客户端软件，大部分逻辑事务处理在服务器端完成，客户端浏览器只完成少量的事务处理，减少了客户端的计算机负载，减轻了系统维护与升级的成本和工作量。在JAVA这样的跨平台语言出现之后，B/S架构管理软件更是方便、快捷、高效。

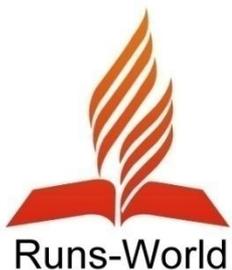


20.1.3 网络传输协议

- 在前面已经讲解了TCP协议的知识，网络传输协议除了TCP协议外，还有UDP协议。UDP (User Datagram Protocol) 是用户数据报协议的英文缩写，UDP是不可靠的无连接数据报服务，它不需要像TCP那样建立连接，它有点类似电报的形式，直接发送，而不管接收端是否收到了数据，所以效率高，发送时速度快，但是不可靠。
- UDP是一种无连接的协议，每个数据报都是一个独立的信息，包括完整的源地址或目的地址。UDP数据包在网络上传往目的地的路径是未知的，因此能否到达目的地，到达目的地的时间以及内容是否正确都是不能确定的。使用UDP不需要建立发送方和接收方的连接。而TCP协议，由于它是一个面向连接的协议，在socket之间进行数据传输之前首先要建立连接，所以在利用TCP协议传输的过程中增加了连接建立的时间。

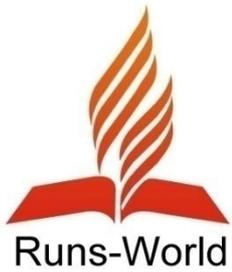


- 使用**UDP**传输数据时是有大小限制的，每个被传输的数据报必须限定在**64KB**之内。而**TCP**没有这方面的限制，一旦连接建立起来，双方的**Socket**就可以按统一的格式传输大量的数据。再次需要强调的是，**UDP**是一个不可靠的协议，发送方所发送的数据报并不一定以相同的次序到达接收方。而**TCP**是一个可靠的协议，它确保接收方完全正确地获取发送方所发送的全部数据。
- 提示：网络编程的目的在于通过网络协议与其他计算机进行通信。网络编程中主要有两个的问题：一是如何准确地定位网络上一台或多台主机，二是如何找到主机后可靠高效地进行数据传输



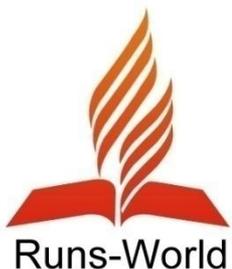
20.1.4 端口和套接字

- 端口被规定为一个在0~65535之间的整数。Http服务一般使用80端口，Ftp使用的是21端口，那么客户必须通过80端口才能连接到服务器的Http服务，而通过21端口，才能连接到服务器的Ftp服务器上。
- 在所有的端口中1~1023之间已经被系统所占用了，因此在定义自己的端口时，不能使用这一段的端口号，而应该使用1024~65535之间的任意端口号，以免发生端口冲突。
- 网络程序中的套接字用来将应用程序与端口连接起来，套接字是一个软件实现，也是一个假想的装置。在Java中，将套接字抽象化为类，所以程序只需创建Socket类的对象，就可以使用套接字。那么Java是如何实现数据传递的呢？答案是使用Socket的流对象进行数据传输，Socket类中有输入和输出流。使用Socket进行的通信都称为Socket通信。将编写的Socket类，用在Socket通信程序中，这就称为Socket网络程序设计。



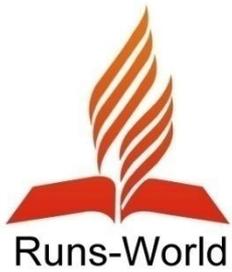
20.2 基于TCP/IP协议的网络编程

- 通过前面一节的介绍，读者已经对网络编程基础的知识有了一个简单的介绍。本节将继续向读者介绍如何利用Java进行基于TCP Socket（套接字）连接的网络应用的开发。TCP/IP是用于计算机通信的一组协议，它包括TCP、UDP等许多协议，这些协议一起称为TCP/IP协议。



20.2.1 Socket套接字

- Socket套接字是基于TCP/IP协议的编程接口，用于描述IP地址和端口，是一个通信链的句柄。应用程序通常通过Socket套接字向网络发出请求或者应答网络请求。
- Socket有两种主要的操作方式，包括面向连接的和无连接的。面向连接的socket操作就像人们打电话，必须先拨号码，建立一个连接，然后再对话。数据包在到达接收端时的顺序与它们出发时的顺序时一样，就像一个人在电话中对另一个人说话一样，每一个字到达另一端的时候与它出发时的顺序一样。
- 无连接的socket操作就像是邮递员送信，邮递员只负责把信送出，至于何时发出，最后能不能到达收信人手中不能保证，无连接的socket操作也一样，负责发出，但不保证数据包的传输质量，数据包到达目的地的顺序可能与出发时的顺序不一样。



20.2.2 ServerSocket类

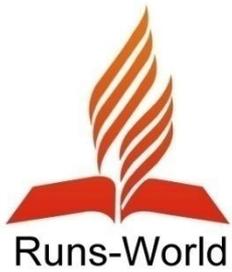
- Java中的网络编程是通过ServerSocket类和Socket类结合使用来完成的。这里首先来讲解一下ServerSocket类，ServerSocket是应用在服务器端的类。在服务器端，由ServerSocket类负责实现服务器套接字。ServerSocket类位于java.net包中。由ServerSocket对象监听指定的端口，端口可以任意指定，但是要注意1024以下的端口通常属于系统保留端口，因此不可以随便使用，应该使用大于1024的端口号，开始监听后，服务器就等待客户端连接请求，客户端连接后，会话开始；在完成会话后，关闭连接。

ServerSocket类的构造器

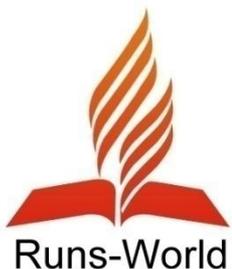
构造器签名	功能
<code>public ServerSocket() throws IOException</code>	创建一个不绑定到任何端口的空ServerSocket对象。
<code>public ServerSocket(int port) throws IOException</code>	创建一个绑定到特定端口port上的ServerSocket对象，若端口port的值为0，则表示使用任何空闲端口。
<code>public ServerSocket(int port, int backlog) throws IOException</code>	创建一个绑定到特定端口上的ServerSocket对象，参数port为指定的端口，参数backlog表示请求等待队列的最大长度。
<code>public ServerSocket(int port, int backlog, InetAddress bindAddr) throws IOException</code>	创建一个绑定到本机指定IP地址指定端口上的ServerSocket对象，参数port为指定的端口，参数backlog表示请求等待队列的最大长度，参数bindAddr表示要绑定到的本机IP地址。请读者注意，如果本机有几个不同的IP地址可以采用此构造器指定绑定到其中的哪一个。



- 注意：同一台主机上的同一端口号只能分配给一个特定的 **ServerSocket** 对象，不能两个 **ServerSocket** 对象监听同一个端口。端口号的理论范围为 0~65535，但前 1024 个中的大部分已经分配给了特定的应用协议，所以不能选用。



- 在**ServerSocket**类中有几个非常常用的方法。首先要讲的就是**accept**方法，使用该方法接收客户端的连接请求，并将与客户端的连接封装成一个**Socket**对象返回。
- 注意：此方法为阻塞方法，在没有接收到任何连接请求前调用此方法的线程将一直阻塞等待，直到接收到连接请求后此方法才返回，调用此方法的线程才继续运行。
- 除了**accept**方法外还有一个**close**方法来关闭**ServerSocket**对象。使用**getLocalPort**方法来获取设置的端口号。



20.2.3 Socket类

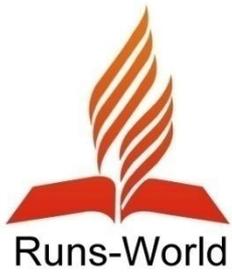
- Socket类表示套接字。使用Socket时，需要指定待连接服务器的IP地址及端口号。客户机创建了Socket对象后，将马上向指定的IP地址及端口发起请求且尝试连接。于是，服务器套接字就会创建新的套接字对象，使其与客户端套接字连接起来。一旦服务器套接字与客户端套接字成功连接后，就可以获取套接字的输入输出流，彼此进行数据交换。Socket类一共有9个构造器，表列出了其中常用的两个。

Socket类的常用构造器

构造器签名	功能
<code>public Socket(String host,int port) throws UnknownHostException, IOException</code>	创建一个连接到指定主机指定端口的Socket对象，参数host为指定的主机名或IP地址字符串，参数port为指定的端口。
<code>public Socket(InetAddress address, int port) throws IOException</code>	创建一个连接到指定主机指定端口的Socket对象，参数address给出要连接主机的IP地址，参数port为指定的端口。



- **Socket**类中同样具有一些方法。其中**getPort**方法和**getLocalPort**方法分别是获取连接的远程端口和使用的本地端口。**getInputStream**方法和**getOutputStream**方法分别是获取**Socket**对象的输入流和输出流，这两个方法是经常被使用到的。
- 注意：**close**方法虽然是很简单的，表示关闭**Socket**对象。但是该方法在程序中是必须有的，这是一个非常好的网络编程习惯。

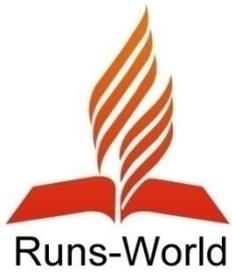


20.2.4 网络编程C/S架构实例

- C/S架构的网络编程程序是由服务器端和客户端所组成的，在开发时一定要先开发服务器端的程序，再来开发客户端的程序。
- 08 int count=0;//声明用来计数的int局部变量
- 09 try
- 10 {
- 11 //创建绑定到9876端口的ServerSocket对象
- 12 ServerSocket server=new ServerSocket(9876);
- 13 System.out.println("服务器对9876端口正在进行监听");
- 14 //服务器循环接收客户端的请求，为不同的客户端提供服务
- 15 while(true)
- 16 {
- 17 //接收客户端的连接请求，若有连接请求返回连接对应的Socket对象
- 18 Socket sc=server.accept();
- 19 //获取当前连接的输入流，并使用处理流进行封装
- 20 DataInputStream din=new DataInputStream(sc.getInputStream());



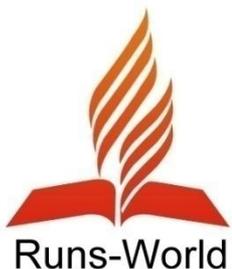
- 21 //获取当前连接的输出流，并使用处理流进行封装
- 22 `DataOutputStream dout=new`
- `DataOutputStream(sc.getOutput` `Stream());`
- 23 //打印客户端的信息
- 24 `System.out.println("这是第"++count+"个客户访问");`
- 25 `System.out.println("客户端IP地址: "+sc.getInetAddress());`
- 26 `System.out.println("本地端口号: "+sc.getLocalPort());`
- 27 `System.out.println("客户端信息: "+din.readUTF());`
- 28 //向客户端发送回应信息
- 29 `dout.writeUTF("服务器的时间为: "+(new Date())+"。");`
- 30 //关闭流
- 31 `din.close();`
- 32 `dout.close();`
- 33 //关闭此Socket连接
- 34 `sc.close();`



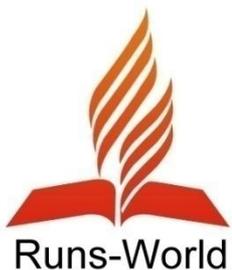
- 在该程序的第12行首先使用**ServerSocket**类创建了一个对象来对**9876**端口进行监听。当有客户端程序访问该程序时，就执行**while**循环，从而让服务器获取客户端的信息，并从服务器端向客户端发送当前时间信息。
- 开发完服务器端程序后，就需要继续来开发运行在客户端的程序。在客户端的程序需要使用**Socket**类来进行操作。



- 09 //创建连接到服务器的Socket对象
- 10 Socket sc=new Socket("192.168.1.119",9876);
- 11 //获取当前连接的输入流，并使用处理流进行封装
- 12 DataInputStream din=new DataInputStream(sc.getInputStream());
- 13 //获取当前连接的输出流，并使用处理流进行封装
- 14 DataOutputStream dout=new DataOutputStream(sc.getOutput
Stream());
- 15 //向服务器发送消息
- 16 dout.writeUTF("Hello");
- 17 //读取服务器的返回消息并打印
- 18 System.out.println(din.readUTF());
- 19 //关闭流
- 20 din.close();
- 21 dout.close();
- 22 //关闭此Socket连接
- 23 sc.close();

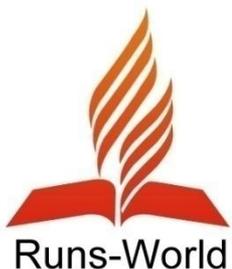


- 在该客户端程序中首先使用流来获取服务器端向客户端发送的信息，该信息是指服务器端的时间。从运行结果中也可以看到这一点。客户端访问服务器端程序后，服务器端运行结果也会发生变化，出现图20-3的运行结果。从运行结果中可以看出显示客户端的IP地址，本地端口号和客户端向服务器端发送的信息。
- 注意：在进行网络编程时，有一点是需要特别注意的。服务器端程序运行后，运行结果是不可以关闭的，这样客户端才会访问到该服务器端程序。
- 如果一个服务器端只能被一个客户端程序访问，这显然是不合理的。
- 当有多个客户方法该服务器程序时将同时显示所有用户的信息。这里采用的是使用一个客户端程序进行多次访问，从而出现两次信息相同的情况。



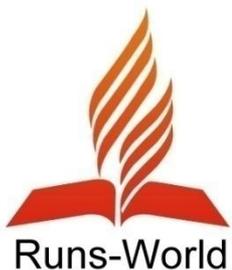
20.3 综合练习

- 1. 开发一个局域网聊天程序。
- **【提示】** 这里给出一个简单的程序，读者可以自己扩展。首先是开发服务器端程序。



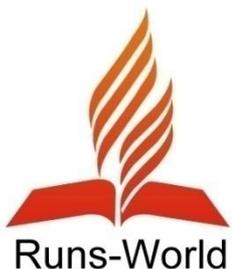
20.4 小结

- 在本章中对Java中如何进行网络编程进行了详细的讲解。首先讲解了一些网络编程的基础知识，包括TCP/IP协议、网络编程模型等。在后面又对基于TCP/IP协议的网络编程进行了详细的讲解，这里主要是使用Socket套接字来进行连接。



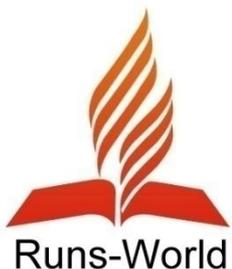
第21章 学生管理系统

- 学习完本书以后，读者可能会问学习Java有什么用。在本章中就使用一个综合案例来讲解如何使用Java进行实际开发。在本章中将开发一个读者最熟悉的学生管理系统，该系统在每一个学校中都有，该系统是非常容易开发的。学习完本章后读者应该完成如下几个目标。
- 掌握实际开发的步骤。
- 能够熟练开发和学生管理系统相类似的系统。
- 掌握Java中的界面开发。
- 掌握Java中如何连接数据库。



21.1 系统设计

- 首先确定学生管理系统的用户。学生管理系统的用户基本分为两类，分别是老师和学生。不管是哪种用户都是必须经过登录才能进入学生管理系统的，所以该系统必须有一个登录界面，并且在该界面中能够让用户选择用户是老师还是学生。该系统是不会对外开放的，所以也不存在注册界面。
- 因为用户分为两种，所以每一种用户进行操作的界面应该是不同的。首先是学生界面，在其中应该只有查询成绩和个人信息查询和插入。在本章中就来学习如何进行学生界面开发。
- 除了学生界面外，还要有一个老师界面。老师在老师界面中可以对学生的信息进行管理，包括查询、修改和删除。同样也可以对学生的程序进行管理，包括查询和插入，由于输入错误还要能够对学生的成绩进行修改，由于学生作弊还能够将学生的成绩进行删除。



21.2 数据库设计

- 在本节中，就来分析本系统中需要的数据库支持。首先数据库中应该有老师和学生这两个表，表中应该最少有用户名和密码两项，使用表中的这两项就可以进行登录。在学生表中还应该具有一些和学籍相关的信息，包括年龄、班级等内容，这样就可以在系统中对学生信息进行操作。
- 除此之外还需要一个成绩表，通过该表老师可以对学生的成绩进行查询、插入、修改和删除。学生也可以通过该表对自己的成绩进行查询。



21.3 登录界面开发

- 不管是老师和学生进入学生管理系统都是从登录界面进入的。在登录界面中应该是让用户选择自己身份的，然后系统将根据用户的选择来判断用户的身份并进行查询不同的数据库。这里为了让读者更容易理解登录界面，并没有连接数据库。读者可以将数据库的操作加到程序中。

21.3.1 界面设计

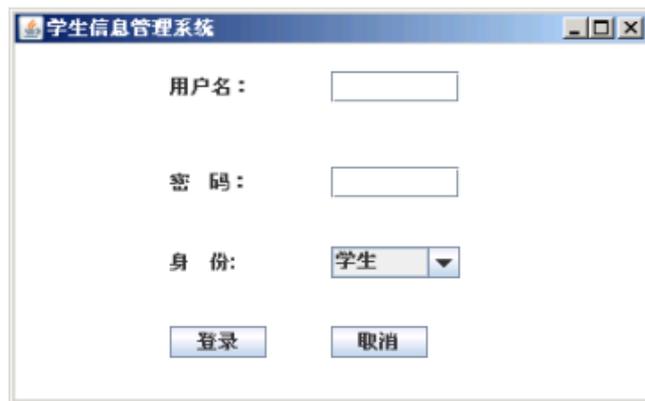
- 这里的登录界面是采用最简单的形式，只需要用户输入用户名和密码就可以登录。并且在界面中定义了一个下拉列表框让用户来选择自己的身份。该界面的基本形式如图所示。



登录界面

21.3.2 程序开发

- 对界面设计好基本形式后，就可以进行程序开发。首先要定义两个标签和两个文本框，分别来表示用户名和密码。并且还需要定义一个下拉列表让用户来进行身份选择，其中选项包括“学生”和“老师”。在程序的最后还定义了两个按钮，从而让用户输入用户名和密码后进行登录。
- **【范例】** 示例代码是用户登录界面程序。

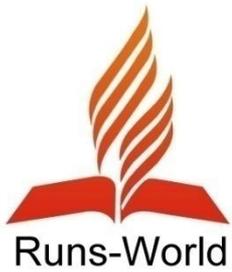


登录界面



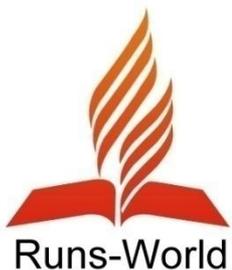
21.4 学生界面开发

- 在本节中将来开发学生界面，在学生界面中，学生可以对自己的信息进行查询，在第一次登录时还可以对自己的信息进行插入，并且学生能够查询自己的成绩。



21.4.1 界面设计

- 因为学生要完成对信息和成绩的操作，所以这里的设计是在界面中定义两个菜单，分别进行信息和成绩的操作。因为对信息的操作包括插入和查询，所以还需要在信息菜单下定义“插入”和“查询”两个子菜单。



21.4.2 程序开发

- 对界面进行设计后，就可以进行程序开发。同样首先是创建一个窗口，在窗口中要创建两个菜单，并且在信息菜单下还要创建“插入”和“查询”两个子菜单。
- 【范例】示例代码是一个学生界面程序。



学生界面

21.4.3 开发插入学生界面

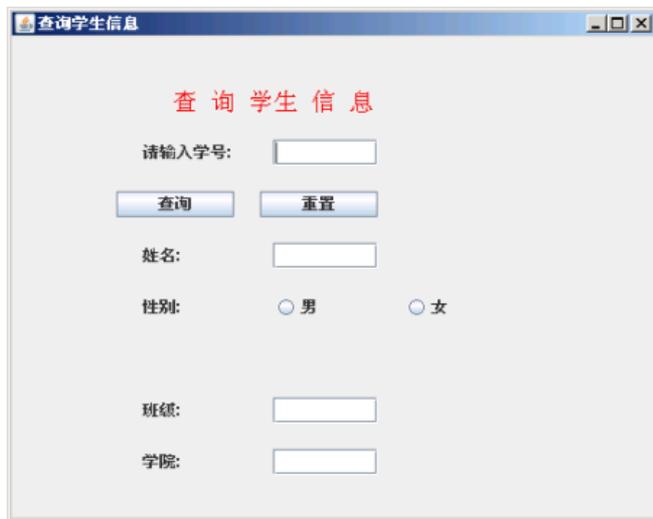
- 在学生界面中单击“信息”菜单下的“插入”子菜单，就会进入学生插入界面，在该界面中学生可以输入自己的信息。
- 【范例】示例代码是一个学生插入信息界面程序。



插入信息界面

21.4.4 查询学生信息界面

- 学生第一次插入信息后，老师是可以对学生的信息进行修改和删除的。除此之外，学生还可以查询自己被修改后的信息，在信息菜单下有一个查询子菜单，单击该菜单就触发事件，从而进入查询学生信息界面。
- 【范例】示例代码是查询学生信息界面。



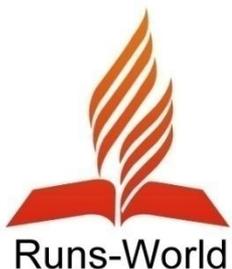
查询学生信息

21.4.5 查询成绩信息

- 在学生界面中还有一个“成绩”菜单，在学生的界面该菜单下只有一个“查询”子菜单。单击“查询”子菜单，将触发事件，进入到查询成绩界面。
- 【范例】示例代码是一个查询成绩界面程序。

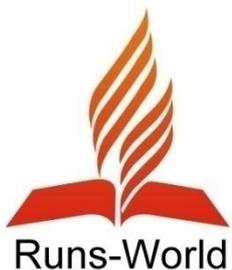


查询成绩



21.5 综合练习

- 1. 参考本章中学生界面，开发学生系统中的老师界面。
- **【提示】**老师界面和学生界面是非常类似的，但是也是存在很多不一样的地方的。首先在老师界面中能够对学生的信息除了插入和查询之外，还能够对学生的信息进行修改和删除，这样就需要在“信息”菜单下增加“修改”和“删除”两个子菜单，也从而需要开发修改学生信息界面和删除学生信息界面。
- 在学生界面中只能够进行对程序的查询，在老师界面中不但能够对学生的信息进行查询，还能够对学生的信息进行修改、插入和删除。同样也是需要增加相应的子菜单和对应的界面。



21.6 小结

- 在本章中以一个实际开发的综合案例来讲解了如何使用Java进行开发。这里以学生管理系统为例，在该程序中主要应用了Java中的Swing知识和数据库连接知识。