

# Chp1 Hello World!

本章是全书的第一章，本章会向你介绍开始学习 Java 之前的准备知识。

## 1 Java 语言介绍

### 1.1 Java 语言的历史

我们从 Java 语言的诞生说起。

1991 年，Sun 公司在一个叫做 James Gosling 的人的带领下，成立了一个项目组，名字叫做“Green”。当时这个项目组成立的时候，是计划开发一种能够运行在消费性电子设备上的编程语言。这种设备的特点是：1、运算能力和运算空间非常有限；2、不同的厂商设计产品时会使用完全不同的 CPU，因此会有完全不同的架构。为了解决这个问题，当时 Green 希望设计出一种具有非常优秀的“跨平台”特性的语言。

Green 项目组的程序员都有着很深厚的 Unix 下 C++ 编程的背景。事实上，从某种意义上说，Java 语言是脱胎于 C++，在很多基础语法方面，有很多跟 C++ 类似的地方。当然，C++ 语言中有很多让人觉得复杂并且难于掌握的特性，而 Sun 公司在设计 Java 语言的时候，把这些特性都摒弃了，而增加了很多优秀的新特性。这些内容随着我们学习的深入会慢慢为大家展开。

1992 年，Green 项目组最初的产品诞生。这个产品一开始被 James Gosling 称之为“Oak”，可能是因为他很喜欢他办公室外面的那颗橡树吧……但是，后来 Sun 公司的同事发现，Oak 已经是另外一种计算机语言的名字。于是，经过讨论，把 Oak 语言改名为 Java 语言。

然而，起初 Java 语言的发展并不是一帆风顺的。在 1994 年之前，Java 语言的优秀特性并没有一个充分发挥的舞台。然而随着时代的进步，一切都开始变得不同了：互联网开始兴起了。由于网络互联互通的需要，因此在多个平台上面运行同样的程序成了一个非常有挑战性但是又非常有意义和价值的东西。Gosling 察觉到了 Java 语言发展的机会，并把 Java 语言由一种在消费设备上运行的语言，修改成为了一种能够在互联网上运行的语言。由于 Java 语言天生具有跨平台特性，Gosling 认为这种特性能够非常好的与互联网结合。这个转折也拉开了 Java 语言发展的序幕。

1995 年 5 月 23 日，在 SunWorld 会议上，Sun 公司对外正式展示了 Java 语言。我们通常把这一天成为 Java 语言的诞生之日。

1996 年，Java 语言发布了第一个正式版本：1.0 版本。这时，Java 语言能够编写的程序称之为 Applet。这种 Applet 只能在集成了 Java 环境的浏览器中运行，当时主要为浏览器来增加各种动态效果，用来美化页面和增强用户与浏览器的交互。应当说，1.0 版本并不能算成功，这个版本非常不成熟。很快的，1.1 版本发布。这个版本修正了 1.0 版本中大量的 bug，并完善了 1.0 版本中的很多缺失的部分。然而，与 1.0 版本一样，1.1 版本同样具有非常大的局限性。

事情到 1998 年有了比较大的改变。1998 年 12 月，Java1.2 版本发布。Sun 公司对这个版本的 Java 做了非常大的结构方面的调整，原有的体系几乎推倒重来。为了表明这是一个非常重大的更新，Sun 公司把 Java1.2 版本，也称之为 Java 2 Platform，用来表示这是一个全新的 Java 平台。

与此同时，Sun 公司还把 Java 2 Platform 进行了细分。对 1.0 和 1.1 扩展之后得到的部分称之为“标准版”，也就是所谓的“Java 2 Standard Edition”，简称“J2SE”。标准版可以

用来写 Applet，也可以用来编写脱离浏览器，独立运行的程序（Application）。这些都是对于一个语言来说，比较基础和比较标准的模块。除了标准版之外，Java 2 还包括“企业版”（J2EE）和“微型版”（J2ME）。企业版主要提供了服务器端编程的功能，而微型版主要提供了在一些资源受限制的平台上（例如手机）运行 Java 的功能。

在本书中，我们不会涉及企业版和微型版。需要注意的是，J2SE 是学习其他两个版本的基础，是学习 Java 语言的第一门课程。

之后，Java 语言发布了 1.3 和 1.4 版本。这两个版本主要修正了 bug，完善了 Java 的类库，但是并没有对 Java 语言进行什么革命性的变化。因此，这两个版本依然是属于 Java 2 平台的范畴。此时，Java 语言进入了真正成熟期，逐渐成为了世界排名第一的语言。大量的企业级应用采用了 Java 语言来开发服务器端软件。

2004 年，Java 推出了一个新版本。这个版本是 1.1 之后，Java 第一次对语言的基础类库做出重大改进的版本。一开始，这个版本被称之为 1.5，后来，Sun 公司为了表明这个版本具有非常强大的功能，把这个版本称之为 5.0。这也就意味着，这个版本与原来的 Java 2 平台相比有着很多不同的地方，因此，现在 Java 2 Platform 已经逐渐成为了历史，而原来的三个平台：J2SE、J2EE、J2ME 也被改名成为了 Java SE、Java EE、Java ME。

2006 年底，Java 发布了版本 6。这个版本修正了很多 5.0 版本中的 bug，改善了性能并增强了类库。

那么，Java 语言究竟有什么魔力，能够在众多语言中脱颖而出，成为世界第一大语言呢？主要源于下面的一些语言特性：

- 纯面向对象

相对于另一种面向对象的语言 C++，Java 语言是一种非常纯粹的面向对象的语言。对于 C++ 而言，写程序除了可以使用面向对象的方式之外，还可以采用面向过程、面向模板等多种方式；而相对的，Java 语言只能采用面向对象的方式进行编程。

- 简单 VS 复杂

由于 Java 语言相对 C++ 来说是一种纯粹的面向对象的语言，因此 Java 语言在理解学习方面，都要比 C++ 语言更简单。Java 的简单性，指的是 Java 语言的这种特点：1、Java 语言本身的特点非常简单，没有复杂和晦涩的语法细节；2、Java 语言倾向于让程序员能够简洁清晰的完成任务。

而 Java 语言同时也是复杂的，体现在：1、虽然 Java 语言本身非常简单，但是它有大量强大而扎实的类库，这些类库极大的丰富了 Java 语言的特性；2、Java 语言最主要的阵地是企业级应用，这种应用本身，由于涉及到多线程、分布式、数据库、网络等各种各样的因此，需求非常复杂。为了应对这种复杂的需求，Java 语言也提供了各种对应的特性，因此从这个角度来看，Java 语言是复杂的。

或许，我们可以拿一句英语来总结 Java 语言的简单和复杂：“Simple thing should be simple, Complex thing should be possible”。

- 开放性

Java 语言是一种开放的语言。这种开放集中体现在 Sun 公司：Sun 公司已经对 Java 语言开源，任何人都可以读到 Java 语言的代码；Sun 公司接受任何人提交的 JSR，也就是说，任何程序员都可以对 Java 语言未来的发展提出自己的开发和建议；Sun 公司提供了 Java 语言的免费下载。

现在，Java 社区有大量开源、免费的东西可供下载和使用，这在一定程度上也帮助了 Java 语言的发展和推广。

- 跨平台性

这是溶入 Java 血液中的一个机制。这个机制决定了，运行 Java 可以在各种平台上面，包括常见的桌面 Windows 系统，也包括企业级应用需要的 Unix 系统。这意味着 Java 语言既能够轻快的在桌面上运行，也能够扎实稳定的在企业级操作系统中运行。这种在不同平台上运行的能力使 Java 语言在企业级应用中的地位有着深远的影响。

## 1.2 Java 语言的运行机制

上一节为读者介绍了 Java 语言的发展历史和一些基本特点，下面要介绍的就是 Java 语言的运行机制。

首先，我们简单介绍一下什么是计算机语言。对于计算机来说，真正能够直接执行的是所谓的“计算机指令”。这种计算机指令，一方面跟操作系统有关系，也就是说，Windows 系统和 Linux 系统下的指令不同；另一方面，也跟计算机的硬件有关系，不同的 CPU 具有不同的指令集。

直接操作计算机指令，使用的是机器语言以及汇编语言。然而，对于程序员来说，直接使用汇编语言来编写程序，开发起来非常的慢，也非常的辛苦。为了能让程序开发的速度提升，我们设计出了计算机高级语言。

所谓的计算机高级语言，实际上指的：人为的规定一些语法。然后，在遵循这些语法的前提下，写出一个文本文件，最后利用某种方式，把文本文件转化为机器指令进行执行。我们现在所谓的编程，往往指的就是编写文本文件的这个部分。这个文本文件一般我们称之为源文件

那么，应当如何把一个源文件转化为机器指令进行执行呢？在现代计算机语言中，主要有两种方式：一种是编译型，一种是解释型。

什么叫编译型语言呢？这指的是，通过一个编译器软件，把源文件转化为可执行文件。可执行文件的内容，就是一些机器指令，以及相关的一些数据。在 Windows 中，可执行文件往往以.exe 作为后缀名。在执行程序的时候，不需要源代码文件，只需要可执行文件即可。示意如下：

源文件 -> 编译器 -> 可执行文件 -> 运行可执行文件 -> 机器指令

与编译型语言相对的是解释型语言。解释型语言需要一个解释器软件，这个软件会读源文件，在读文件的过程中，同时完成将源文件内容翻译成机器指令以及执行的过程。换句话说，解释器将读取源文件、翻译成机器指令、执行指令这三步同时完成。示意如下：

文本文件 -> 解释器 -> 直接翻译成机器指令

由上可知，编译型语言在将源文件编译成可执行文件之后，运行程序只需要可执行文件，不再需要重复编译的过程。而解释型语言每次运行时必须重复翻译源文件，因此从运行效率上来说，解释型语言远远不如编译型语言。

当然，解释型语言也有自己的优势：跨平台性较好。由于编译型语言运行时只需要可执行文件，而可执行文件又与平台紧密相连，这也就意味着，对于不同的平台，必须要有不同的可执行文件才行。而相对而言，解释型语言就没有这么麻烦，对于不同的平台，只需要有不同的解释器就可以了，源代码几乎不用进行修改。

而 Java 语言，则兼具有编译型和解释型两种语言的特点：Java 语言运行时，采用的是先编译、后解释的方式运行。

首先，Java 源代码要写在后缀名为.java 的源文件中。然后，通过一个编译器，编译生成.class 文件，这个文件被称为“二进制字节码文件”。

而.class 文件并不能够直接在机器上执行。执行.class 文件，需要一个解释器，这个解释器会把.class 中的指令翻译成真正机器上的指令。也就是说，需要解释执行.class 文件。

示意如下：

java 源文件 -- 编译 --> .class 字节码文件 -- 解释执行 --> 真正的机器指令

字节码文件是平台中立的，也就是说，运行在不同平台上的，.class 文件内容相同，与所在平台无关。

那么.class 文件中保存的是什么内容呢？这个文件中保存的也是计算机指令，所不同的是，这些计算机指令不是真实计算机所拥有的指令，而是一些虚拟的指令。在解释执行.class 文件的指令时，为了能让这些虚拟的计算机指令能够转换成真正的计算机指令，我们需要一个 Java 虚拟机（Java Virtual Machine，简称 JVM）。

JVM 事实上是一个软件，这个软件为 Java 程序模拟出一个统一的运行环境。Java 程序只需要适应这个虚拟的环境，而与底层真正的硬件环境及操作系统环境无关。换句话说，JVM 的作用在于，它屏蔽了底层不同平台的差异。

Java 虚拟机接收.class 文件中的虚拟指令，这些指令很类似于真正的汇编语言指令，但这些指令与底层的操作系统平台和硬件平台无关，完全是另外设计出的一套独立体系。而不同平台下的 Java 虚拟机，在执行时，会把.class 文件中的虚拟机指令翻译成对应平台上真正的计算机指令。因此，我们可以修改上面的示意如下：

java 源文件 -- 编译 --> .class 字节码文件 - 在 JVM 中解释执行 --> 真正的机器指令

Java 语言这种“先编译，后解释”的运行机制，使得其同时拥有了编译型语言的高效性和解释型语言的跨平台性，Sun 公司给出了最好的注解：“Write once , run anywhere”

### 1.3 JRE 与 JDK

如果一个程序员要发布 Java 程序，一般来说，会发布.class 文件。而如果要运行 Java 程序，同样指的是运行.class 文件。因此，运行 Java 程序，只需要 Java 虚拟机和解释器就可以运行。即 JRE，也就是 Java Runtime Environment 的缩写，指的是 Java 的运行环境。包括 JVM 和 Java 解释器。

但是仅有 JRE，只能是完成从 class 文件到真正的机器指令这一步，而无法把一个源文件编译成一个.class 文件。在 Sun 公司的网站上，有一个术语叫做 JDK。所谓的 JDK，指的是 Java Development Kit，Java 开发工具包。从内容上说，

JDK = JRE + 工具（编译器、调试器、其他工具……）+ 类库

我们进行 Java 开发，至少应当有 JDK。可以到 Sun 公司的网站上进行 JDK 的下载（在撰写本书时，下载链接为：<http://java.sun.com/javase/downloads/index.jsp>）。JDK 的安装也可以参考 Sun 公司的网站（链接为：<http://java.sun.com/javase/6/webnotes/install/index.html>）。对于 Windows 系统而言，下载 JDK 之后，安装起来与其他的软件安装时并没有太大区别，在此不多赘述。

## 2 Java 开发环境配置

安装完了 JDK 之后，还不能马上进行 Java 开发，还需要进行一些环境变量的配置。配置完之后，才能真正进行 Java 程序的开发。

## 2.1 三个环境变量

Java 的环境配置，其实主要是对三个环境变量进行配置。这三个变量分别为：JAVA\_HOME、PATH 和 CLASSPATH。

**JAVA\_HOME** 环境变量，表示的是 Java 的安装目录。这个变量是用来告诉操作系统 Java 的安装路径的，当其他的程序需要 Java 进行支持的时候（例如一些 Java 的服务器、Java 的数据库客户端等），会通过 JAVA\_HOME 来寻找 Java 的安装路径。

**PATH** 环境变量，是在命令行上输入 Java 命令时，用来指示操作系统去哪个路径下找 Java 的相关程序。往往会被 PATH 变量配成 Java 的安装路径/bin 目录。

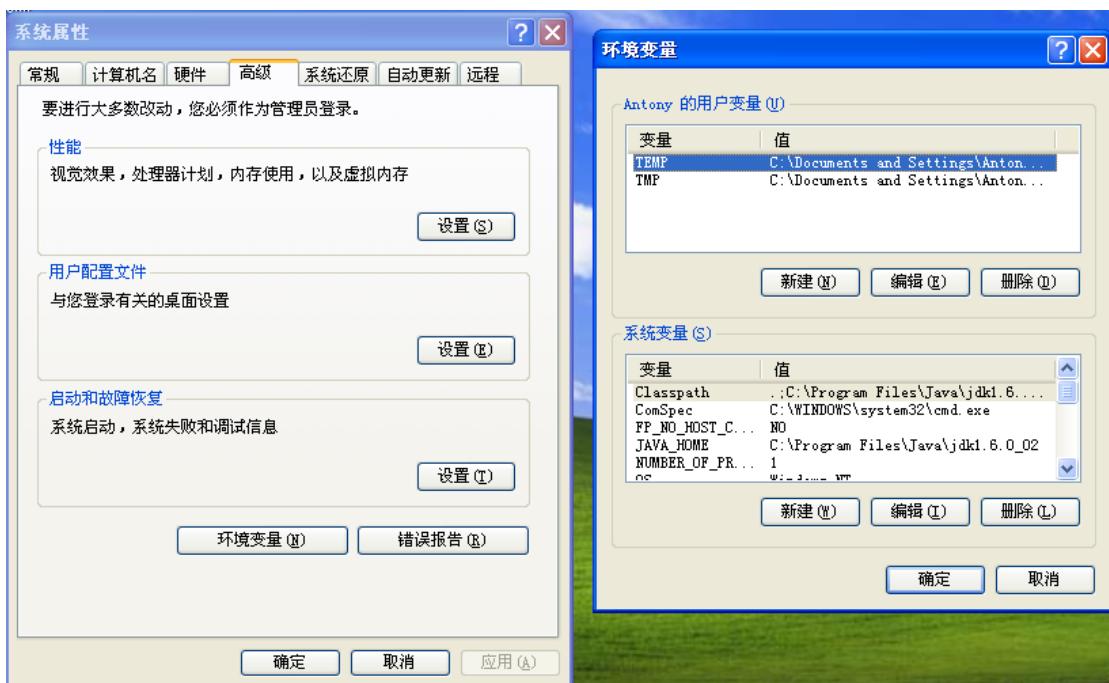
**CLASSPATH** 是用来指示编译器和 JVM 去哪个目录寻找.class 文件。当我们运行 Java 程序时，必然会需要获取.class 文件的信息，而且往往还需要不止一个.class 文件的信息。此时，我们就需要在硬盘中寻找到相应的.class 文件。而硬盘中的文件成千上万，JVM 如何寻找呢？为了能够让 JVM 有的放矢，我们需要设置 CLASSPATH 环境变量，指定一些目录，让 JVM 寻找.class 文件时，只需要寻找这些我们指定的目录即可。

我们分 Windows 和 Linux 两个平台，来具体的介绍如何配置环境变量。

## 2.2 环境变量的配置

### 2.2.1 Windows

右键点击“我的电脑” -- 选择“属性” -- 选择“高级” -- 选择“环境变量”

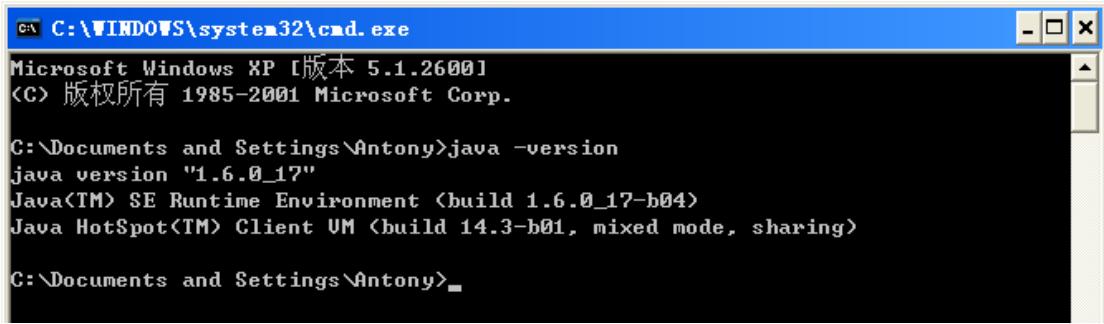


在系统变量中，点击“新建”，增加 JAVA\_HOME 系统变量，这个变量的值设为 Java 的安装目录。假设 Java 安装在 C:\Program Files\Java\jdk1.6.0\_02。

然后，在系统变量中，查找 PATH 变量（不区分大小写）。在 Path 变量的末尾，增加一句：“;C:\Program Files\Java\jdk1.6.0\_02\bin”。注意两个要点：1、在 Path 末尾增加，千万不要把原有的内容去掉；2、增加的值为 Java 安装目录下的 bin 目录。

最后，增加一个 CLASSPATH 变量。值为“.”，注意不带双引号。

当所有的环境变量都配置好了之后，打开 Windows 中的命令行（对于 WindowsXP 来说：开始菜单 -- 运行 -- cmd），执行 java -version。如果得到正确的版本信息，则说明配置正常。如下图：



The screenshot shows a Windows XP command prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window title bar also displays 'Microsoft Windows XP [版本 5.1.2600]' and '(C) 版权所有 1985-2001 Microsoft Corp.'. The command line shows the user running 'java -version'. The output of the command is:  
java version "1.6.0\_17"  
Java(TM) SE Runtime Environment (build 1.6.0\_17-b04)  
Java HotSpot(TM) Client VM (build 14.3-b01, mixed mode, sharing)  
The command prompt ends with 'C:\Documents and Settings\Antony>'.

## 2.2.2 Linux

Linux 下的配置我们简单的描述一下。假设在 Linux 中 Java 安装在 /opt/java/jdk6 目录下。对于使用 bash 的 Linux 来说，在用户主目录下，修改 .bash\_profile 文件，在这个文件的最末尾增加三行：

```
export JAVA_HOME=/opt/java/jdk6  
export PATH=$JAVA_HOME/bin:$PATH  
export CLASSPATH=.
```

要注意的是，Linux 中的环境变量区分大小写，因此要注意大小写不要写错。

## 3 第一个程序

完成了 Java 环境变量的配置，下面我们来看我们的第一个程序：HelloWorld 程序。

### 3.1 Hello World 程序

首先我们写出 HelloWorld 程序的代码。在硬盘上创建一个名为 test.java 的文件，并用记事本打开这个文件，书写如下内容。注意：区分大小写，并注意空格：

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

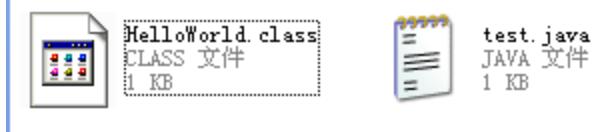
代码写完之后就可以进行保存。然后，在命令行模式下，进入这个.java 文件所在的文件夹，使用下面的命令来进行编译：

```
javac test.java
```

编译成功时如下图：

```
c:\ C:\WINDOWS\system32\cmd.exe  
D:\book>javac test.java  
D:\book>
```

在编译完成之后，会在目录下生成一个相应的 HelloWorld.class 文件，如下图：



之后，在命令行上，用下面的命令来执行：

**java HelloWorld**

执行结果如下：

```
c:\ C:\WINDOWS\system32\cmd.exe  
D:\book>java HelloWorld  
Hello World  
D:\book>
```

在屏幕上打印出一个“Hello World”的字符串。

请保证在你的机器上看到 Hello World 字符串再继续下一小节的学习。

## 3.2 深入 HelloWorld

### 3.2.1 Hello World！

我们首先来研究和分析一下 test.java 这个程序的代码：

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

首先，第一行： class HelloWorld，这句话定义了一个名字叫做 HelloWorld 的类。其中， class 是 Java 语言的关键字，而 HelloWorld 则是定义的类的名字。

至于什么是类，这个概念对于初学者来说比较复杂，在此不多做介绍。对于初学者来说，可以把类理解成：代码的容器。也就是说，在 Java 中绝大部分代码都要写在类的范围之内，要写代码就必须先定义一个类。

在定义完了 HelloWorld 之后，后面有一对花括号，花括号中的内容就表示是这个类中的内容。

在这个花括号里，下面的这行：

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

这一行定义了一个主方法（也叫主函数）。在 Java 中，主方法的定义比较长，但是非常有用，也许你现在还无法理解这些内容，随着学习的深入，你会理解这里面的每一个单词。但是现在，照着写并且记住主方法的写法就行了。

那么主方法有什么用呢？我们把主方法称之为：程序执行的入口。也就是说，Java 程序

在执行的时候，会执行类中的主方法，当主方法执行完毕之后，程序也就退出了。

在主方法内部，有下面这句：

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

这是一个 Java 的语句。注意，每一个 Java 语句都应当以分号结尾。

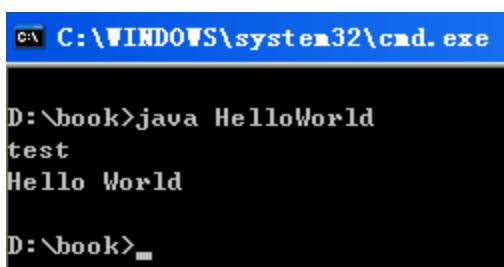
这个语句能够在屏幕上打印出一个字符串，这个字符串的内容，就是括号中的“Hello World”字符串。并且，这个语句在打印完字符串之后，还会打印一个换行符，进行换行。

与 System.out.println 对应的，还有 System.out.print 语句。这个语句同样完成打印，但是输出不换行。我们比较下面两个程序的区别：

代码：

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

输出结果：



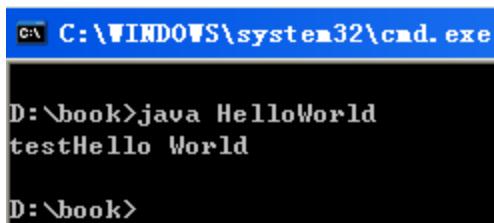
The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The command 'java HelloWorld' is entered, followed by two lines of output: 'test' and 'Hello World'. The prompt 'D:\book>' is visible at the bottom.

输出完 test 之后，换行，再输出 Hello World。

代码 2：

```
class HelloWorld{  
    public static void main(String args[]){  
        System.out.print("test");  
        System.out.println("Hello World");  
    }  
}
```

输出结果：



The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The command 'java HelloWorld' is entered, followed by a single line of output: 'testHello World'. The prompt 'D:\book>' is visible at the bottom.

输出 test 之后，不换行直接输出 Hello World。

### 3.2.2 类与.class 文件

我们注意到，编译 test.java 之后，会产生一个名为 HelloWorld 的.class 文件。这个.class

文件的文件名与我们在 test.java 中定义的类名一摸一样。换句话说，一个.java 文件中定义的每一个类，编译后都会对应的生成一个和类名完全一样的.class 文件。

这个.class 文件不是可执行文件，用文本编辑器也无法正常打开。我们也可以在 test.java 中定义多个类。例如下面的代码：

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

class Welcome{
    public static void main(String args[]) {
        System.out.println("Welcome to learn java");
    }
}
```

编译之后，会生成两个.class 文件，一个 HelloWorld.class，一个 Welcome.class。

在运行时，运行的是.class 文件。但是需要注意的是，使用 java + 类名运行，而不能有.class 这个后缀。

比如，我们要运行 Welcome 类，则使用的命令应当是：

java Welcome

而不是

java Welcome.class

要注意的是，我们使用 java Welcome 运行 Welcome 类时，JVM 需要在硬盘上找到相应的 Welcome.class 文件。此时，JVM 会通过 CLASSPATH 变量的指示，来寻找.class 文件。由于我们把 CLASSPATH 配置成了一个“.”，这表示当前目录，因此 JVM 就会在当前目录下寻找 Welcome.class 文件。

### 3.2.3 类与公开类

如果我们给 HelloWorld 类增加一个前缀：public，则此时，HelloWorld 就不是一个普通的类，而变成了一个公开类。代码如下：

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

公开类有自己的特殊性。此时再编译 test.java，则会产生一个编译时的错误，错误如下：

```
C:\WINDOWS\system32\cmd.exe

D:\book>javac test.java
test.java:1: 类 HelloWorld 是公共的，应在名为 HelloWorld.java 的文件中声明
public class HelloWorld<
^
1 错误
```

这个错误说明，如果要使用一个公开类，则有一个要求：**公开类的类名必须与.java 文件的文件名相同（包括大小写）。**

为了修正这个错误，我们必须把原来的 test.java 改名为 HelloWorld.java，再次编译才能编译通过。

由于一个.java 文件中只能有一个文件名，因此一个.java 文件中，最多只能有一个公开类。当然，如果不是公开类的话，一个.java 文件中可以有多个类。

## 4 初学者忠告

至此，我们对 Java 的基本介绍，以及 Java 环境的配置就全部介绍完了。在开始下一章的内容之前，下面是对初学者的一些忠告：

1、动手敲代码。不论是书本上的例子代码，还是练习中的代码，请每一个都自己敲一遍。有些题目很简单，但是在做这些简单练习的时候，或许你会犯一些低级错误，而提前犯这些错误能够避免你在解决难题时被这种低级错误纠缠。有些题目很难，你凭借自己的能力无法完成，那欢迎你看答案。但是看完答案之后，请记得，自己再敲一遍代码，巩固一下。

2、不要使用 IDE。Java 语言的流行，使得有大量的好用的开发工具，比如 eclipse，netbeans 以及 jbuilder 等。这些工具能够极大的提高程序员的开发效率，是程序员的好帮手，但是不适合初学者。初学者需要更多的锻炼和磨砺，才能够打下扎实的基础，才能够更好的掌握这些工具。如果一开始你就是用这些工具的话，很有可能你会一直被这些工具束缚住，对他们产生依赖，而影响你对 Java 的掌握。

建议你是用一个文本编辑器 + 命令行的方式，学习 Java 的开发。起码在学习 Java 的前半个月应该这么做。

3、选择一个有语法高亮的文本编辑器。如 notepad++ 和 notepad2，你也可以选择更加强大的 UltraEdit。有一个支持 java 语法的编辑器会让你愉快很多。

# Chp2 Java 语言基础

## 本章导读

在上一章里，我们介绍了 Java 语言的一些背景，并介绍了 Java 语言环境的配置，为后面进一步的学习打下良好的基础。

从本章开始，我们将逐步开始介绍 Java 语言的语言特性和语法规则。

## 1 注释

注释指的是一些描述代码的文字。

我们可以对代码的各方面描述，都写成代码的注释。注释中典型的内容包括：这段代码是如何工作的、这段代码使用了什么算法、这段代码执行的流程如何，等等。

注释不是 Java 代码的一部分，编译时，编译器会把 Java 代码翻译成字节码，而注释则会被编译器自动忽略。因此，代码中有没有注释，都不会影响到代码运行的结果。

但是，注释却是编程中必不可少的内容。有良好注释的代码，能够极大的增强代码的可读性。也就是说，加上注释的代码更容易让人读懂。

对于初学者而言，一开始可能并不会意识到注释的意义。然而我们要知道，Java 代码是给机器看的，同时也是给人看的，事实上，程序员看代码的时间远远超过 Java 编译器看代码的时间（想象一下你调试和写程序需要多久，而编译器编译程序需要多久）。因此，我们写程序应当注重“可读性”，也就是说，我们写出的代码应当尽可能让人容易读懂。随着工作量和代码量的增加，越来越多的代码和越来越复杂的逻辑会让程序员很难读懂、修改和维护代码。也许有一天，在看到自己写的代码的时候，你也会发出这样的感叹：“咦，这段代码真的是我写的么？我怎么一点都看不懂了？”（当然，发出这样的感叹说明你确实积累了不少代码了）。这时候，注释能给程序员带来最大的帮助。

Java 中的注释从语法上来说主要有三种：单行注释、多行注释和 javadoc 注释。

### 1.1 单行和多行注释

单行注释和多行注释，是很多编程语言中常见的注释语法。

在 Java 中，单行注释以“//”开头，直到遇到一个换行符为止。以下为合法的单行注释：

```
Eg 1: //This is a comment.  
Eg 2:  
System.out.println("Hello World"); // This is another comment  
Eg 3:  
if (XXX) { //This is comment  
} else { //This is else  
}
```

多行注释以“/\*”开头，以“\*/”结尾，在/\*和\*/之间的所有内容，均作为注释。

以下为多行注释的例子：

```
Eg 1 /* This is Comment */
```

多行注释可以跨行，例如：

```
Eg 2
```

```
/* This is a example  
Of multi-line comment */
```

特别要注意的是，多行注释不能嵌套。例如以下程序片段：

```
/* This is comment  
/*can not have inside comment*/  
*/
```

这个程序片段在多行注释中包含了另一个多行注释，会引发一个编译错误。

## 1.2 javadoc 注释

javadoc 注释，是 Java 中一种比较特殊的注释。这种注释用来生成 api 文档。

程序员在对外发布 Java 代码的时候，还应当对外公布这些代码的用法。这样，才能让其他的程序员能够更加方便的利用已有的代码。在 Java 中，我们往往使用 api 文档，来向其他程序员介绍代码的组成和用法。

例如，Sun 公司发布的 JDK，包括了一个庞大的类库，为了说明这些类的用法，Sun 公司还提供了相应的 api 文档。

Java1.6 的文档链接如下：

<http://java.sun.com/javase/6/docs/api/>

下图显示了 java.lang.Object 类的 api 文档：



对于程序员来说，如何来生成这样的 api 文档呢？很显然，让程序员直接手工编写，是一件非常麻烦的事情。为此，Java 提供了一种相对比较简单的机制。

首先，在代码中，我们可以使用 javadoc 注释。从语法上说，这种注释由 “`/**`” 开头，以 “`*/`” 结尾，在 “`/**`” 和 “`*/`” 之间可以有多行文本，并且与多行注释一样，javadoc 注释也不允许嵌套。这是一种特殊的多行注释，可以在代码中描述类、函数等等。

然后，在 JDK 中，有一个 javadoc 命令。这个命令能够从源文件中获得 javadoc 注释，并根据 javadoc 注释，自动生成 api 文档。

例如，我们写出如下代码：

```
/** 这是对类的注释 */
public class TestJavaDoc{
    /** 这是对主方法的注释
    并且有多行 */
    public static void main(String args[]){
        System.out.println("test java doc");
    }
}
```

上述代码，在类和方法前面，都加上了 javadoc 注释。

然后，进入命令行。在命令行上输入如下命令：

```
javadoc -d doc TestJavaDoc.java
```

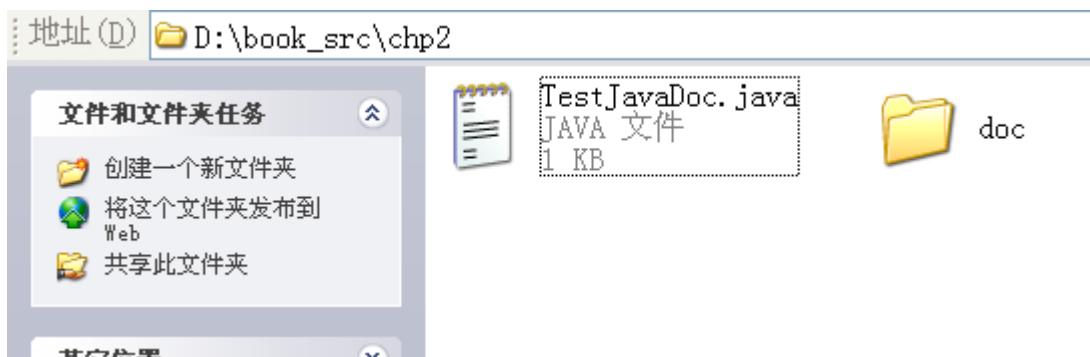
javadoc 命令能够根据代码中的 javadoc 注释生成文档。“-d 文件夹名” 是 javadoc 命令的一个选项，这个选项表示的是，生成的文档要放在后面指定的文件夹下。例如，上面 “-d doc”，就表示生成的 javadoc 文档要放在 doc 目录下。javadoc 命令最后一个参数，是一个 java 源文件的名字，表示要生成哪一个源文件的文档。

使用该命令之后，命令行上运行的结果如下：

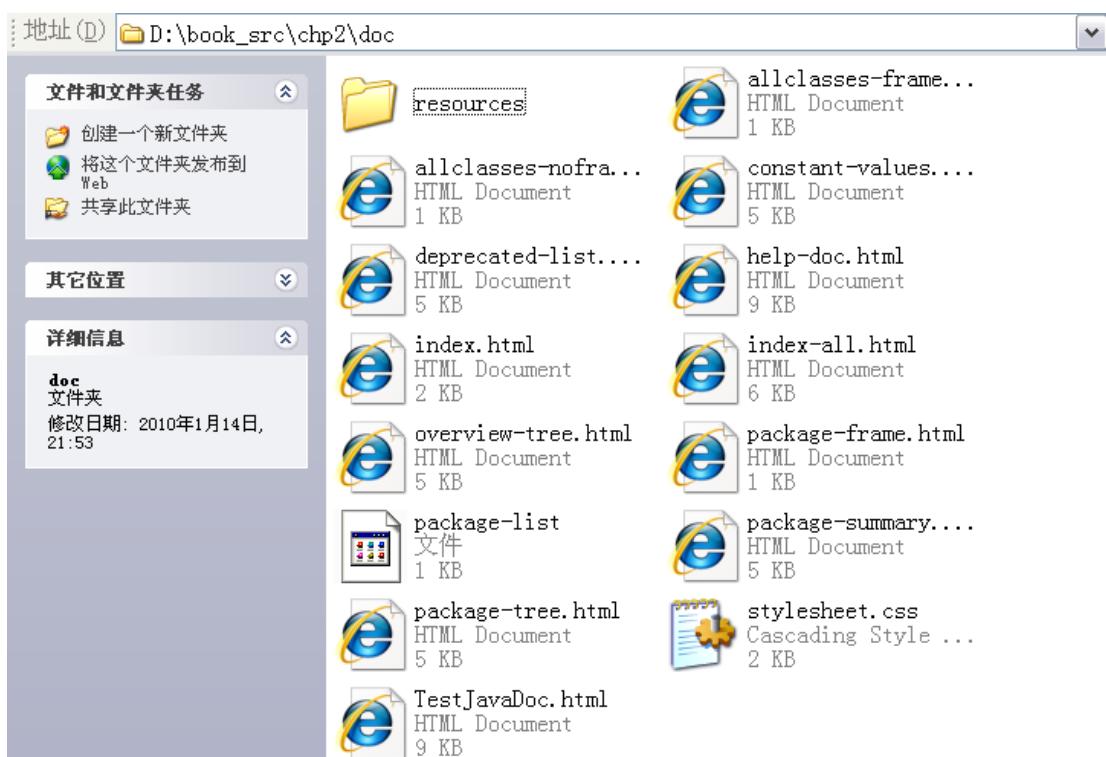
```
D:\book_src\chp2>javadoc -d doc TestJavaDoc.java
Creating destination directory: "doc\""
Loading source file TestJavaDoc.java...
Constructing Javadoc information...
Standard Doclet version 1.5.0
Building tree for all the packages and classes...
Generating doc\TestJavaDoc.html...
Generating doc\package-frame.html...
Generating doc\package-summary.html...
Generating doc\package-tree.html...
Generating doc\constant-values.html...
Building index for all the packages and classes...
Generating doc\overview-tree.html...
Generating doc\index-all.html...
Generating doc\deprecated-list.html...
Building index for all classes...
Generating doc\allclasses-frame.html...
Generating doc\allclasses-noframe.html...
Generating doc\index.html...
Generating doc\help-doc.html...
Generating doc\stylesheet.css...

D:\book_src\chp2>
```

然后，在 D:\book\_src\chp2 目录下，生成了一个 doc 目录，如下：



在 doc 目录下，生成了相当多的 html 网页以及一些其他的文件，如下图：



在生成的一系列文件中，可以用浏览器打开 index.html 网页，显示如下：

The screenshot displays a Java API documentation page. At the top, there are navigation links: All Classes, TestJavaDoc (highlighted), Package, Class Tree, Deprecated, Index, Help, and links for PREV CLASS, NEXT CLASS, SUMMARY, NESTED, FIELD, CONSTR, and METHOD. Below these are links for FRAMES, NO FRAMES, DETAIL, FIELD, CONSTR, and METHOD.

**Class TestJavaDoc**

java.lang.Object  
└ TestJavaDoc

---

public class TestJavaDoc  
extends java.lang.Object

这是对类的注释

---

**Constructor Summary**

[TestJavaDoc\(\)](#)

---

**Method Summary**

static void	<a href="#">main(java.lang.String[] args)</a>
-------------	---

这是对主方法的注释 并且有多行

---

**Methods inherited from class java.lang.Object**

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

javadoc 命令自动为我们生成了上图显示的文档。在文档中，“这是对类的注释”，以及“这是对主方法的注释 并且有多行”，这些内容都来源于我们在 TestJavaDoc.java 中写的 javadoc 注释。

上面，我们介绍了如何用 javadoc 注释自动生成 api 文档。关于这种注释的更详细的使用，可以参考 <http://java.sun.com/j2se/javadoc/>。

## 2 包

随着代码的大量增加，程序员写的.java 源文件以及编译产生的.class 字节码文件会大量的增加。如果任由这种情况发生的话，无论是查询还是管理都会非常的不方便。为了解决这方面的问题，Java 提供了“包”来帮助我们组织和管理类。

本节内容介绍包的使用，主要包括 package 语句和 import 语句。

### 2.1 package 语句

在操作系统中，如果有大量的文件，为了方便管理，我们往往会按照某种规则，创建结构合理的文件夹结构。例如，如果有大量的 mp3 音乐文件，用户可以把这些文件按照歌手、风格、专辑等，创建相应的文件夹，分门别类的进行管理。

类似的，在 Java 中，为了更好的管理大量的类，引入了“包”的概念。

使用 package 语句可以用来将一个特定的类放入包中。要注意的是，如果要使用 package 语句，则这个语句必须作为.java 文件的第一个语句，并写在任何类的外面。例如，对我们之前写的 HelloWorld 类放入 book.corejava.chp2 包，则代码如下：

```
package book.corejava.chp2;
public class HelloWorld{
```

```
public static void main(String args[]) {  
    System.out.println("Hello World");  
}  
}
```

上面的这段代码，就把 HelloWorld 类放在了 book.corejava.chp2 包下。要注意的是，在加包之后，使用 HelloWorld 类时必须加上包名作为前缀，因此完整的写法应当是：book.corejava.chp2.HelloWorld。这种在类名前面加上包名的写法称为类的全限定名。

假设上述代码存在一个 HelloWorld.java 文件中，而这个文件在硬盘的 C:\JavaFile 目录下，则可以使用编译命令：

```
javac HelloWorld.java
```

该命令在 C:\JavaFile 文件夹下生成 HelloWorld.class 文件。

到现在为止，除了加了一句代码之外，似乎没有别的事情发生。然而接下来，运行.class 文件时，会产生一个错误，结果如下：

```
C:\JavaFile>javac HelloWorld.java  
  
C:\JavaFile>java HelloWorld  
Exception in thread "main" java.lang.NoClassDefFoundError: HelloWorld (wrong name: book/corejava/chp2/HelloWorld)  
        at java.lang.ClassLoader.defineClass1(Native Method)  
        at java.lang.ClassLoader.defineClass(Unknown Source)  
        at java.security.SecureClassLoader.defineClass(Unknown Source)  
        at java.net.URLClassLoader.defineClass(Unknown Source)  
        at java.net.URLClassLoader.access$000(Unknown Source)  
        at java.net.URLClassLoader$1.run(Unknown Source)  
        at java.security.AccessController.doPrivileged(Native Method)  
        at java.net.URLClassLoader.findClass(Unknown Source)  
        at java.lang.ClassLoader.loadClass(Unknown Source)  
        at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)  
        at java.lang.ClassLoader.loadClass(Unknown Source)  
        at java.lang.ClassLoader.loadClassInternal(Unknown Source)  
Could not find the main class: HelloWorld. Program will exit.  
  
C:\JavaFile>
```

出现这个错误的原因，显然跟我们刚刚给 HelloWorld 加包有关。

在加包之后，使用 HelloWorld 类时，必须使用全限定名运行 HelloWorld 的 class 文件。即，命令必须是

```
java book.corejava.chp2.HelloWorld
```

但是，这样运行，结果依然出错。运行结果如下：

```
C:\JavaFile>java book.corejava.chp2.HelloWorld  
Exception in thread "main" java.lang.NoClassDefFoundError: book/corejava/chp2/HelloWorld  
Caused by: java.lang.ClassNotFoundException: book.corejava.chp2.HelloWorld  
        at java.net.URLClassLoader$1.run(Unknown Source)  
        at java.security.AccessController.doPrivileged(Native Method)  
        at java.net.URLClassLoader.findClass(Unknown Source)  
        at java.lang.ClassLoader.loadClass(Unknown Source)  
        at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)  
        at java.lang.ClassLoader.loadClass(Unknown Source)  
        at java.lang.ClassLoader.loadClassInternal(Unknown Source)  
Could not find the main class: book.corejava.chp2.HelloWorld. Program will exit  
.  
  
C:\JavaFile>
```

出现这个错误的原因是，在使用包时，**包结构必须和硬盘上的文件夹结构一致**。也就是说，在上述例子中，HelloWorld.class 文件，必须放在类路径下的 book/ corejava/chp2/目录下。

在 C:\JavaFile 文件夹下，建立如下目录结构，并移动.class 文件到特定文件夹下：

```
C:\JavaFile\  
|--HelloWorld.java  
|--book  
|   |--corejava  
|       |--chp2  
|           |--HelloWorld.class
```

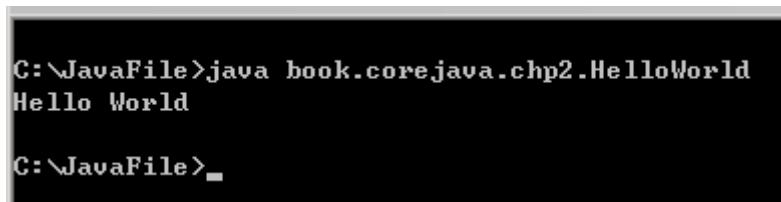
示意如下：



此时，使用 java 命令：

```
java book.corejava.chp2.HelloWorld
```

才能正确执行。运行结果如下：



特别提醒，**执行 java 命令时，当前路径必须在源代码包结构的上层目录（在本例中，是 C:\JavaFile 目录）下**，否则将会出现运行时错误。这是因为我们在配置环境变量的时候，将 CLASSPATH 这个变量的值设为了“.”。也就是说，JVM 会到当前目录下去寻找所需要的类文件，而在哪个目录下能够找到 book/corejava/chp2/HelloWorld.class 这个文件呢？当然是 book 目录的上层目录。因此，我们在运行时必须以这个目录作为当前目录。

也可以使用 javac 命令的-d 选项，要求编译器把编译生成的.class 文件按照包结构来存放。编译器会按照类的包结构，自动生成对应的目录结构。用法如下：

```
javac -d 目标目录 源文件名
```

其中的目标目录指的是：把生成的目录放在哪个目录下作为子目录。

例如，如果编译之前，硬盘文件结构如下：

```
C:\JavaFile\  
|--HelloWorld.java
```

我们把 C:\JavaFile 目录作为当前目录，使用命令

```
javac -d . HelloWorld.java
```

这就表示把编译成的字节码连同其目录放入当前目录下。于是，硬盘文件结构变为：

```
C:\JavaFile\  
|--HelloWorld.java  
|--book
```

```
|--corejava  
    |--chp2  
        |--HelloWorld.class
```

使用包的主要目的是为了避免类名冲突。例如，假设有两个程序员不约而同的都使用 HelloWorld 作为类名。如果这两个类都不使用包的话，一方面，两个类的类名相同，因此在使用上会产生歧义。另一方面，在把两个程序员写完的.class 字节码文件放在一起运行的时候，在同一个文件夹下会有两个同名的字节码文件，从而产生文件冲突。而如果这两个程序员使用不同的包 p1 和 p2，一方面生成的 HelloWorld.class 字节码文件一个放在 p1 目录下，另一个放在 p2 目录下，不会有文件名的冲突。另一方面，全限定名不同，在使用这两个类时，一个被称为 p1.HelloWorld，另一个被称为 p2.HelloWorld，从而避免了类名的冲突。

为了保持包名的绝对唯一性，Sun 公司建议将公司的 Internet 网址（必然是唯一的）的逆序作为包名，并在不同的项目中使用不同的子包。例如，假设公司的网址为 abc.com，则应当使用 com.abc 作为包名，而这个公司创建的 corejava 项目的包名则可以设定为 com.abc.corejava。

## 2.2 import 语句

把类加上包之后，除了运行时之外，在源代码中使用这个类时也必须使用类的全限定名。例如，假设我们要使用一个 HelloWorld 类型的变量，如果用下面的代码

```
public class Welcome{  
    public static void main(String args[]){  
        HelloWorld hello;  
    }  
}
```

则编译时会在第三行出错，如下：

```
C:\JavaFile>javac Welcome.java  
Welcome.java:3: cannot access HelloWorld  
bad class file: .\HelloWorld.java  
file does not contain class HelloWorld  
Please remove or make sure it appears in the correct subdirectory of the classpath.  
        HelloWorld hello;  
               ^  
1 error  
C:\JavaFile>_
```

因为在我们使用类的时候，由于 HelloWorld 类加了包，因此必须使用全限定名。

必须把第三行代码替换成

```
book.corejava.chp2.HelloWorld hello;
```

编译才能通过。

如果要多次使用 HelloWorld 类怎么办？例如要定义三个变量，hello1，hello2，hello3，则代码如下：

```
public class UseHelloWorld1{  
    public static void main(String args[]){  
        book.corejava.chp2.HelloWorld hello1;
```

```
        book.corejava.chp2.HelloWorld hello2;
        book.corejava.chp2.HelloWorld hello3;
    }
}
```

可以看到，冗长的包名出现了三次！这样，我们的代码有大量的冗余，清晰程度降低，可读性降低。

为了解决这个问题，Java 为我们提供了 import 语句。

**import 语句表示导入特定的类。在使用 import 语句导入一个类之后，使用时就能简单的使用类名。**例如：

```
import book.corejava.chp2.HelloWorld;
public class Welcome{
    public static void main(String args[]){
        HelloWorld hello;
    }
}
```

前面的 import 语句，就好比是一个声明。这个声明表明，在程序接下来的代码中，如果遇到 HelloWorld 类，则指的都是之前引入的 book.corejava.chp2.HelloWorld 类。这样在后面我们定义 hello 这个变量的时候，就省略了冗长的包名。

在使用 import 语句之后，如果我们需要三个 HelloWorld 类型的变量，则代码如下：

```
import book.corejava.chp2.HelloWorld;
public class UseHelloWorld2{
    public static void main(String args[]){
        HelloWorld hello1;
        HelloWorld hello2;
        HelloWorld hello3;
    }
}
```

可以看出，在使用了 import 语句之后，程序的冗余代码大大减少，变得相当的简洁和清晰。

**从语法上说，如果要使用 import 语句，则这个语句必须放在任何类的外部，仅能在 package 语句之后。**例如，以下均是 import 语句的正确用法：

Eg1

```
// 如果有 package 语句，则 import 语句应在 package 语句之后，
// class 定义之前
package book.corejava.chp2.hello;
import book.corejava.chp2.HelloWorld
public class ImportTest{
    .....
}
```

Eg 2

```
// 如果没有 package 语句，则 import 语句应为第一个语句
import book.corejava.chp2.HelloWorld
public class AnotherTest{
```

.....

}

此外，一个.java 文件中可以有多个 import 语句，用于导入多个类。

如果想要一次导入某个包下的所有类，则可以使用\*通配符。例如，下面这个语句

```
import java.util.*;
```

表示导入 java.util 包内的所有类。也就是说，当我们用到这个包里面任何一个类的时候，都可以省略包名。特别要提醒的是，上述语句并不能导入 util 包的子包，也就是说，java.util.jar、java.util.logging 等包下的类并未导入。

例如，我们要使用下面这几个类：

java.util.ArrayList, java.util.HashSet, java.util.logging.Logger，下面的代码：

```
import java.util.*;
public class TestImport{
    public static void main(String args[]){
        ArrayList list;
        HashSet set;
        Logger log;
    }
}
```

由于 import java.util.\*能够导入 java.util.ArrayList 和 java.util.HashSet 类，但不能导入 java.util 的子包中的类，因此 java.util.logging.Logger 类没有导入，编译器找不到这个类，因此编译出错，如下图所示。

```
C:\JavaFile>javac TestImport.java
TestImport.java:6: cannot find symbol
symbol  : class Logger
location: class TestImport
          Logger log;
                     ^
1 error

C:\JavaFile>_
```

另外，只能用\*导入一个包，而不能使用 java.\*或 java.\*.\*等语法，试图导入所有以 java 为前缀的包。

最后，java.lang 包为标准 Java 库中的最常用的包，这个包中的类会在所有 java 程序中自动被导入。也可以理解为，在所有.java 文件中，import java.lang.\*;这个语句都不用写，编译器会自动导入这个包下的类。

### 3 编码规范

这一小节主要介绍一些简单的编码规范。在实际工程中，一个团队一定会要求团队成员在写代码的时候，必须遵循统一的规范。只有这样，才能够让整个团队的代码风格统一，才能够让写出的代码有着比较好的可读性。

常见的编码规范有：

- 1、 良好的注释
- 2、 良好的标识符命名
- 3、 良好的缩进

作为一名初学者，一定要在一开始的时候就养成良好的编码习惯。首先介绍标识符的命名规范。

下面，我们对“标识符和命名”以及“缩进”做进一步的介绍。

## 3.1 标识符命名

标识符，指的是程序员为程序组件起的名字。起名字是一门艺术，这一点对标识符也一样。良好的标识符命名风格和习惯，能够大大增加代码的可读性。

我们首先来介绍一下 Java 中标识符的分类。标识符为程序组件的名字，而所有的 Java 程序组件基本分为 5 大类：包、类、变量、函数、常量。首先我们介绍一下 Java 标识符的命名规则。

在 Java 中，关于标识符命名有三条规则。如果我们定义的标识符违背了语法，编译时将不能通过。

标识符命名的规则如下：

一、Java 标识符由字母、数字、下划线( )、货币符号(\$)组成，其中数字不能开头。

要注意的是，所谓“字母”，从技术上说，是一个 unicode 字符，包括中文字符。换句话说，Java 标识符能够使用中文。例如，你可以写一个类叫做“学生”，写一个变量叫做“成绩”等等，Java 程序照样认得。但是在实际编程中，为了避免一些不必要的麻烦，所有 Java 程序员都会使用英文字母起名。对大部分程序员来说，代码里面出现中文总觉得怪怪的。

以下的标识符都是合法的：

abc \_score a\$b

但以下的标识符是非法的：

a#(出现了非法字符 #) 2a (不能以数字开头)

二、Java 标识符区分大小写。

也就是说，helloWorld，HelloWorld，HELLOWORLD 这三个标识符，对于 Java 来说是完全不同的三个名字。

三、不能与 Java 关键字重名。

首先我们罗列一下 Java 中的一些关键字：

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

有些关键字我们已经介绍过，例如 int, short 等，其余关键字随着我们的学习，会逐一进行详细讲解。

此外，Java 中有两个单词：goto / const，他们在 Java 中没有特殊的含义，但是由于这两个单词在其他语言中（例如 C 语言）有特殊含义，为了避免其他语言的程序员学习 Java 时产生混淆和误会，Java 语言不允许程序员使用这两个单词。也就是说，虽然在 Java 语言中。goto 和 const 这两个单词没有特殊含义，但是程序员在给程序组件起名字时，依然不能用这两个单词。

严格的说，“`true`”和“`false`”也不能称为 Java 语言的关键字。这两个单词是 `boolean` 类型的字面值，我们稍后再介绍。

除了语法规则之外，还有一些标识符命名方面的习惯。违反了这些习惯的标识符可能是符合语法，从而能够编译通过的。但使用了这样的标识符，会被认为是不良好，不规范的。在 Java 语言中，标识符命名应该注意两大习惯：

### 一、望文生义

这指的是说，标识符的名字应当起的有意义，最好能通过名字，让人一眼就能看出标识符的作用。例如，变量 `totalScore` 肯定是用来统计总分，函数 `addStudent` 肯定是用来增加一个学生。这样的名字是比较好的名字。它们很容易让人理解这个标识符的意义，从而提高程序的可读性。

### 二、大小写规范

相比上一条规范，这一条显得非常教条。Java 语言中，对于不同的程序组件，有着不同的大小写规范。罗列一下：

包名：全小写。例如 `book.corejava`；

类名：每个单词首字母大写。例如 `HelloWorld`

变量/函数名：首单词小写，后面每个单词首字母大写。例如 `helloWorld`

常量名：全大写，单词之间用下划线分隔。例如 `HELLO_WORLD`

我们可以参考 Java SE 类库中的命名方式，会发现所有的标识符都符合上述两个习惯。这无疑是 Java 世界中约定俗成的，所有程序员都恪守的准则。如果你是一个 Java 初学者，也请从一开始就养成良好的标识符命名习惯，千万不要轻视它们！

## 3.2 缩进

缩进，指的是在写代码的时候，在某些行的行首，会留出一些空白字符。良好、规范的缩进能够极大的提高代码的清晰程度，让可读性大大增加。

例如，下面的代码，有着混乱的换行和缩进。

```
import book.corejava.chp2.HelloWorld; class BadCode
    {public static void main(String
args[]) {System.out.println("welcome")
;HelloWorld
hw;}}
```

然而神奇的是，这一段代码能够编译通过。但是，这样的代码，可读性简直差到了极致！不仔细看几遍是很难读懂这段代码的。

而实际上，对上面的代码简单调整一下换行和缩进，马上就焕然一新：

```
import book.corejava.chp2.HelloWorld;
class GoodCode{
    public static void main(String args[]){
        System.out.println("welcome");
        HelloWorld hw;
    }
}
```

对于缩进的使用，应当注意下面两条：

- 第一， 对于每一个代码块，其内容都应当缩进。例如， class 顶格写，而 class 的内容，相对于 class 都应当有缩进。类似的，主方法有一级缩进；而主方法的所有内容，都应当在主方法的基础上，再缩进一级。
- 第二， 缩进应当统一。也就是说，每一级缩进的长度应当一致。比较推荐的缩进长度是四个空格或者一个制表符（也就是按键盘上的 tab 键所产生的空白）。

## 4 变量

### 4.1 变量的含义

变量是编程中最基本的概念之一。如果你已经熟悉了变量的定义，可以跳过这一节。如果不是，也不用担心。只要把这一小节的内容读完，然后简单的写两个程序，相信你很快就会掌握。

对于 Java 语言而言，每一个变量都代表着内存中的一小块区域，而这块区域能够用来存放某个数据。例如，在 Java 中，我们需要存放一个整数，那就可以定义一个变量，代码如下：

```
int a;
```

上面的这行代码定义了一个 int 类型的变量 a。首先简单介绍一下 int 类型，这种类型表示整数，一个 int 类型的变量占 4 个字节的空间。

定义一个 int 类型的变量，实际上完成了下面这样两个步骤：

- 1、在 JVM 中，分配 4 个字节的空间，用来存放一个整数。
- 2、为这 4 个字节的空间起了一个名字 a，供程序员使用。

这样，在程序中，我们就可以用“a”这个名字来指代这四个字节的空间，从而访问到这个空间中所存放的整数。下面是对变量操作的一些例子：

```
public class TestVariable{  
    public static void main(String args[]){  
        int a;  
        a = 10;  
        System.out.println(a);  
        a = 15;  
        System.out.println(a);  
        int b;  
        b = a;  
        System.out.println(b);  
    }  
}
```

我们仔细分析上面这段代码。

- 1) a = 10;

这被称之为变量的赋值。这句话的意思是，把 a 所表示的 4 个字节的空间，设置为数值 10。

2) System.out.println(a);

在这个语句中，打印 a，就是指打印 a 变量的值，也就是 a 所代表的 4 个字节的值。

此时，a 的值为 10，因此打印结果为 10。

3) a = 15;

再次为 a 变量赋值。这时，把 a 所表示的 4 个字节的空间，设置为 15。原来 a 的值 10 被新值 15 所替代。

4) System.out.println(a); 再次输出时，输出结果为新值 15

5) int b; 定义一个新变量 b。这个变量同样为 int 类型。

6) b = a;

这句话的含义是，把 a 变量的值赋值给 b 变量。a 变量的值是 15，通过赋值，把 b 变量的值也设置为 15。

7) 输出 b 变量，此时输出结果为 15

运行结果如下：

```
C:\JavaFile>javac TestVariable.java  
C:\JavaFile>java TestVariable  
10  
15  
15  
C:\JavaFile>
```

我们再来进一步研究一下 Java 中变量的一些基本操作。

```
01: public class TestVariable{  
02:     public static void main(String args[]){  
03:         int a;  
04:         a = 10;  
05:         int b = 20;  
06:         int c, d;  
07:         int e = 30, f = 40;  
08:         c = a + b;  
09:         d = 5;  
10:         d = d + c;  
             System.out.println(a);  
             System.out.println(b);  
             System.out.println(c);  
             System.out.println(d);  
             System.out.println(e);  
             System.out.println(f);  
     }  
}
```

第 3、4 行，定义了一个 a 变量，并赋值为 10；

第 5 行，在定义一个变量 b 的同时，为其赋值为 20。这说明可以在定义一个变量的同时赋值。这样的写法，和先定义变量之后再赋值，最终的结果是一样的。也就是说，第 5

行的代码，可以等价的写成：

```
int b;  
b = 20;
```

相对而言，在定义变量的同时赋值，能够让代码显得更加简洁。

第6行，在一个语句中定义了两个变量c和d。这说明在一个语句中可以定义多个变量，变量之间用逗号隔开。

第7行，在一个语句中定义了两个变量e和f，并同时对这两个变量赋值。

第8行，把a和b变量的值相加，并把结果保存在c变量中。由于a变量是10，b变量时20，因此最后c变量结果是30.

第9行，把d的值设为5。

第10行，把d和c的值相加，赋值给d。怎么来理解呢？在进行计算的时候，首先会计算“=”右边。此时，需要计算d+c的结果。由于此时d的结果为5，c的结果为30，因此d+c的值为35。

然后，把35赋值给d。此时，d的结果为35。

程序的输出结果如下：

```
C:\JavaFile>javac TestVariable.java  
  
C:\JavaFile>java TestVariable  
10  
20  
30  
35  
30  
40  
  
C:\JavaFile>■
```

此外，在Java中，变量是强类型的。所谓强类型，指的是每一个变量都有自身的类型。不同的变量类型，大小不同，所能存放的数据也不同。例如，int类型的变量只能存放整数数据，下面的代码是错误的：

```
int a = 123.45;
```

相比强类型来说，有些编程语言是弱类型的。用户只需要定义一个变量，就可以存入任意类型的数据。显然这样会引起混乱，甚至我们可能自己都很难搞清楚一个变量中究竟装的是哪种数据。很庆幸，Java不是这样的。

可以把变量想象成不同大小的盒子。不同类型的盒子，具有不同的空间大小，被用来存放不同类型的东西。饭盒和鞋盒，不仅大小是不同的，而且装的东西也不一样。我们绝不会把一双皮鞋放在饭盒里，是吧？

## 4.2 变量的类型

下面我们就来介绍一下Java语言中的变量类型。在Java中，所有变量类型分为两类，一类为基本类型，一类为对象类型。本章主要介绍的是**基本类型**。

基本类型，英语叫 primitive type，也有人翻译成“原始类型”、“简单类型”等等。这类变量属于编程语言中比较基础的组成部分，因此也被称之为“基本类型”。

基本类型总共分为8种，分别为byte、short、int、long、float、double、char、boolean。下面让我们来学习这八种基本类型。

### 4.2.1 整数类型

有四种类型都用来表示整数，他们是 byte、short、int、long，他们之间的区别在于他们所占的内存空间和表示范围。下表为这四种基本类型的参数。

类型名称	所占空间	表示范围
byte	1 个字节	-128 ~ 127
short	2 个字节	-32768 ~ 32767
int	4 个字节	-2147483648 ~ 2147483647
long	8 个字节	-9223372036854775808 ~ 9223372036854775807

可以把这四种基本类型想象成四个大小不同的饭盒，虽然空间不同，但是所装的数据基本上是一类的。要注意的是，表示范围小的类型可以直接赋值给表示范围大的类型，而反之不行。例如：

```
int a = 10;  
byte b = 10;  
a = b; //可以，表示范围小的类型赋值给范围更大的类型  
b = a; //编译错误！表示范围大的类型不能赋值给范围小的类型！
```

也不能给一个变量赋一个超过其表示范围的值。例如：

```
byte b1 = 100; //可以赋值  
byte b2 = 150; //编译错误！150 超过了 byte 类型的表示范围！
```

### 字面值

字面值，指的是某个类型的合法取值，或者说，可以为该类型的变量赋值的数据。例如，“int a = 5;”，在这个代码中，a 就是变量，5 就是字面值。

要注意的是，字面值同样有类型。对于 1、5、10、-99 等整数字面值来说，其类型都是 int 类型。

但是，下面的代码能够正常执行：

```
byte b = 100;
```

在上面的这一行代码中，虽然 100 是一个 int 类型的字面值，但是由于 100 在 byte 类型的表示范围内，因此程序能够自动把 100 这个 int 类型转成 byte 类型。

另外，如果需要 long 类型字面值，我们可以用在数值后面加 L 的方式（大小写均可）。例如，1000 是一个 int 类型的字面值，而 1000L 是一个 long 类型的字面值。例如，下面的代码都是正确的：

```
long l = 1000L;  
long l = 1000l;
```

当然，小写的“l”容易和数字“1”混淆，因此最好还是用大写的“L”。

## 4.2.2 浮点数类型

在计算机术语中，小数被称为浮点数。在 Java 语言中，浮点数有两种，分别为 float 和 double。两者相关参数如下：

类型名称	所占空间	表示范围
float	4 个字节	$3.4028235 \times 10^{38}$ $\sim 1.4 \times 10^{-45}$
double	8 个字节	$1.7976931348623157 \times 10^{308}$ $\sim 4.9 \times 10^{-324}$

浮点数类型的符号可以是正的，也可以是负的。

### 字面值

浮点数的字面值有两种。第一种是直接给出小数，例如 1.5, -0.38 等。需要注意的是，这样给出的字面值都是 double 类型，如果需要 float 类型的字面值，需要在数值后面写一个字母 f (大小写均可)。例如：1.6f, -10.39F。事实上，为了更明确的表示 double 类型的字面值，也可以在数值后面写一个字母 d (大小写均可)。例如：1.6d, -10.39D

第二种是用科学计数法表示。例如， $-1.5 \times 10^{23}$ ，就可以用-1.5e23 来表示。而  $3.8 \times 10^{-5}$ ，则可以用 3.8e-5 来表示。而 float 类型的字面值，则在数值后面再加一个 f。例如下面的代码

```
double d = 2.67e13;
float f = 1.57e-3f;
```

计算机表示小数，自然就涉及到表示精度的问题。由于计算机内部使用二进制表示小数，与我们通常的十进制表示法不同，因此小数在表示过程中有可能会有精度方面的损失。例如下面的程序：

```
public class TestFloat {
    public static void main(String[] args) {
        double a = 2.0 - 1.1;
        System.out.println(a);
    }
}
```

这段代码会输出：

0.8999999999999999

在上面的程序中，显示的结果并不是我们期望的“0.9”，这就是因为计算机内部用二进制表示这个数的时候，产生了精度上的问题。这就类似于，用十进制表示数，很难精确的表示 1/3 一样。

## 4.2.3 字符类型

Java 中的字符类型为 char 类型，其本质为一个无符号整数。相关参数如下：

类型名称	所占空间	表示范围
char	2 个字节	0 ~ 65536

在计算机中，一个字符是由一个正整数表示，这个整数被称为字符的编码。Java 中的

char 类型存放的就是字符的编码。例如，大写字母 ‘A’ 的编码为 65，用 16 进制表示为 0x41

给一个字符变量赋值总共有三种方式：字面值赋值，编码赋值，unicode 赋值。三种赋值方式的语法如下：

```
char ch1 = 'A'; //字面值赋值  
char ch2 = 65; //编码赋值  
char ch3 = '\u0041'; //unicode 赋值
```

**字面值赋值**：直接给出用单引号包围的单个字符。注意，Java 中单引号和双引号有不同的含义，**单引号用来包围字符，双引号用来包围字符串**。

**编码赋值**：直接给出字符的编码值，例如给出大写字母 ‘A’ 的编码 65。

**Unicode 赋值**：Unicode 是一种国际字符编码规范，能够处理全世界各国的语言。如果要使用这种赋值方式，则需要在一对单引号之下给出u 和 4 位 16 进制的 **unicode 编码**。在这个例子中，十六进制数 0041 表示成十进制为数字 65，因此 ch3 表示的也是大写字母 ‘A’。

由于 Java 中的 char 类型采用 Unicode 编码方式，前面介绍过，Unicode 能够处理世界各国的语言字符，因此，char 类型可以用来处理中文。例如：

```
char ch1 = '中';
```

另外，有一些字符在 java 中有特殊含义，在使用时要通过反斜杠 “\” 来转义。例如，想表示一个单引号字符 (')，则必须使用 \" 的语法。除了单引号 (\\') 之外，双引号 (\"")、反斜杠 (\\)、换行符 (\\n)，制表符 (\\t) 都是一些常用的转义字符。

#### 4.2.4 布尔类型

布尔类型来进行逻辑运算。Java 中的布尔类型为 boolean 类型，这种类型只有两种字面值：true 或者 false。要注意的是布尔类型无法跟其他类型进行运算，也无法自动转化为其他类型。例如：

```
boolean flag = true; // 正确  
boolean flag = 1; // 错误！1 不能转换为 boolean 类型！
```

### 4.3 变量的基本操作

介绍完八种基本类型之后，接下来我们要介绍的是对变量的一些基本操作。

#### 4.3.1 强制类型转换

我们在上一节中介绍，表示范围大的类型，不能给表示范围小的类型赋值。例如，下面的代码就会产生一个编译错误：

```
int a = 10;  
byte b;  
b = a; //!编译错误，int 类型不能直接给 byte 类型赋值
```

编译出错的原因，在于 int 类型中能够保存的数值，有可能远大于 byte 类型的表示范围。就好像如果要把一个大汤盆中的东西倒到一个小碗中，很有可能会出问题。这时候，编译器就会阻拦我们的这个操作。

但是，有些情况下，大汤盆中的东西只剩一点儿了，这个时候完全可以把大汤盆中的

东西倒到小碗中。对于上面的代码来说，虽然 int 的表示范围远大于 byte 类型，但是我们可以确认，int 变量 a 中的数据，完全可以让一个 byte 变量来保存。这种情况，我们不希望编译器阻拦我们。因此，可以使用强制类型转换。

强制类型转换的语法如下：

(类型) 变量

这个语法表示，把变量的值强制转换为某个特定的类型。例如上面的代码，我们要把 a 的值强制转换为 byte 类型，再给 b 赋值，则可以把代码写成：

```
b = (byte) a;
```

需要注意的是，上面的语法转换的是 a 变量的值。在执行过程中，会先分配一个 byte 类型的临时空间，然后把 a 的值转换成 byte 类型，然后放到这个临时空间去，最后把临时空间的值赋值给 b。在转换过程中，并没有改变 a 变量的类型和值。

那 a 变量有 4 个字节，怎么转换成为 1 个字节的 byte 类型呢？Java 会把 a 变量 4 个字节中，最后的一个字节取出来，然后把这个字节的值，放到临时空间去。而另外的 3 个字节的内容，则在强制类型转换的时候被舍弃。

因此，如果 a 中数值的范围超过了 b 所表示的范围，使用强制类型转换也能正常编译，但是转换的结果会有问题。例如

```
int a = 150;  
byte b = (byte) a; //编译通过，但是 b 的值为一个错误值  
System.out.println(b); //输出为-106
```

为什么一个超过表示范围的正数，转换之后会变成一个负数呢？这涉及到在计算机中如何用二进制表示正数。例如，下面的代码：

```
short s = 450;  
byte b = (byte) s;
```

首先我们来看 short 类型的变量 s。一个 short 类型占 2 个字节，一个字节是 8 个二进制位，因此一个 short 变量占 16 个二进制位。这 16 个二进制位中，有 15 个字节是表示数值，最高的一个二进制位表示的是数值的符号。因此，整数的最高位也被称之为符号位。如果一个整数是正数，则它的符号位为 0；相反，负数的符号位为 1。

450 是正数，因此其符号位为 0。然后，把十进制数 450，转换成二进制之后，得到结果 111000010。然后，除了符号和数值位之外，在其余位上补 0。最后，450 在内存中表示如下：

0	0	0	0	0	0	0	1	1	1	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

然后，把 s 强制转换成 byte 类型。此时，会开辟 1 个字节的空间，并把 short 类型最低 8 位取出来，得到结果如下：

1	1	0	0	0	0	1	0
---	---	---	---	---	---	---	---

可以看到，在转换过程中，舍弃了 short 变量的部分内容。

然后，11000010 这个数字，被当做 byte 类型时，首先要判断它的符号。由于这个数字的最高位为 1，因此 java 会把这个数当做是负数来处理。负数的表示方式与正数不同，相对比较复杂，在此不多介绍。需要明确的是，在强制类型转换时，如果对超过表示范围的数做强制类型转换，有可能产生一些意想不到的情况，例如正数变成负数，或者负数变成正数。因此，使用强制类型转换应当要注意，避免发生类似的错误。

### 4.3.2 自动类型提升

在正式介绍 Java 中的各种基本运算符之前，先给大家介绍 Java 中很特别的一个语法性质：自动类型提升。请看下面的一段代码：

```
byte a = 10, b = 20;  
byte c = a+b;
```

对于大部分语言而言，这都应该是一段合法的代码，而对于 Java 语言来说，这段代码会出一个编译时错误：

```
C:\JavaFile>javac TestVariable.java  
TestVariable.java:4: 可能损失精度  
找到: int  
需要: byte  
    byte c = a + b;  
                           ^  
1 错误
```

要理解产生这个错误的原因，就要了解 Java 的自动类型提升特性。我们首先来看下面这个表达式：

```
int a = 10, b = 20, c = 40;  
int d = a + b + c;
```

在运行上面的代码时，首先把 a、b、c 三个变量的值分别设为 10、20 和 40。然后，在计算 d 的值时，首先需要计算 “=” 右边表达式的值。由于这个表达式有两个 “+”，因此在计算的时候，需要进行两次加法操作：首先计算 a+b 的值，在计算出结果之后，再把结果和 c 的值相加。为此，Java 会首先获得 a 变量的值 10，然后获得 b 变量的值 20，之后进行第一次加法，计算出结果 30。因为这个结果在后面的运算中还要使用，所以，Java 会把这个数值保存在一个临时变量中。然后，Java 进行第二个加法运算时，首先取出临时变量的值 30，然后取出 c 变量的值 40，再计算出结果是 70。70 作为整个 “a+b+c” 表达式的值，也会存在一个临时变量里。最后，把这个临时变量的值赋值给 d 变量。

可以这么来理解上述的过程：就好比我们在做 a+b+c 这道数学题时，首先，会把 a+b 的值计算出来，计算出来之后，会把这个值暂时写在草稿纸上。再然后，计算这个写在草稿纸上的值和 c 相加的结果，并把所得到的最终结果也写在草稿纸上。最后，再把草稿纸上的值，重新写回到考卷上。

因此，当 Java 遇到例如 c = a + b 这样的式子时，会首先计算等号右边的 a+b 的值，然后赋值给等号左边的变量 c。在计算过程中，Java 会创建一个临时变量来保存 a+b 这个表达式的计算结果，然后把这个临时变量中的值赋值给 c。基本过程如下：

计算 a+b 的值 --> 保存到临时变量 --> 把临时变量的值赋给 c。

问题在于，尽管 a 和 b 两个变量都是 byte 类型，但是 Java 为临时变量选择类型时，会将这个类型“自动的提升”为 int 类型，大小为 4 个字节。于是，上述过程中的第三步，就成了把一个 int 类型赋值给 byte 类型的操作，从而产生一个编译时错误。这就是 Java 语言中的自动类型提升特性。

要避免这个问题，只需要对其结果进行强制类型转换即可。即把原代码改为：

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

需要注意的是，由于是对 a+b 的结果进行强制转换，因此要对 a+b 这个表达式加上括号。Java 自动类型提升的规则如下：

1. 如果运算数中存在 double，则自动类型提升为 double

2. 如果运算数中没有 double 但存在 float，则自动类型提升为 float
3. 如果运算数中没有浮点类型，但存在 long，则自动类型提升为 long
4. 其他所有情况，自动类型提升为 int。

换而言之，byte + byte, byte + short 之类的运算，都会被自动提升为 int 类型。需要说明的是，char 类型也能进行运算，并且 char 类型与其他类型运算时，也会进行相应的自动类型提升。

### 4.3.3 运算符

在本节中，我们将详细给大家介绍 Java 中的常用运算符。

在介绍运算符之前，首先介绍一个非常基本的概念：表达式。

所谓的表达式，指的是用运算符连接变量或字面值所形成的式子，例如：a+b, 2+c 等。需要强调的一点是，任何一个表达式都会有一个值。也可以理解为，所有表达式都会返回一个结果，例如 1+2 会返回 3 作为表达式的结果。

表达式的值也有不同类型。例如，布尔表达式，就说明这个表达式的值的类型为 boolean 类型。因此，在写程序的过程中，一定要明确表达式的值，以及这个值的类型。

理解了表达式的概念，我们来关注形成表达式的关键元素：运算符

#### ◆ 赋值号 (=)

赋值操作是编程中最常用的操作之一。Java 中的赋值号为一个等号 (=)。赋值号具有右结合性，也就是说，会先计算赋值号右边的内容，然后把计算结果赋值给左边的变量。

此外，`a = b` 构成一个赋值表达式，其作用就是将变量 b 中的值赋值给变量 a，这个表达式也有值，表达式的值为赋值号右边的计算结果。

由这个特性，在 Java 中可以进行连等操作，即：`a = b = c = 10` 这样的赋值操作。

#### ◆ 基本数学运算

基本数学运算包括加 (+) 减 (-) 乘 (\*) 除 (/) 以及取余 (%) 操作。这些操作和数学上的定义没有区别。

需要注意的是，类似于 `3/2` 这样的表达式。由于 3 和 2 都是整数类型，因此根据自动类型提升的规则，结果也一定是整数类型。因此，`3/2` 这个表达式的值是 1，如果希望得到数学上精确的结果 1.5，则需要使用 `3.0/2` 这样的表达式，来保证结果数据是 double 类型的。

此外，Java 中的数学运算法符合先乘除，后加减的规则。例如：`2+3*4` 这个表达式的值为 14。

#### ◆ +=, -=, \*=, /=

在实际编程中，经常会写出类似于 `a = a + 2` 之类的表达式，表示把 a 在原有值的基础上增加 2。这种写法有一种更方便的简写形式：`a += 2`。

`-=, *=, /=` 等运算符与 `+=` 类似。

#### ◆ ++与--

对于 `a+=1` 这样的表达式而言，还有一种更加简单的运算符：`++`，与之类似的，`a-=1` 可以用 `a--` 来代替。

要注意的是，使用 `++`（或 `--`）运算符有两种方式：前缀式或后缀式。例如：

`a++;` //后缀式

`++a;` //前缀式

要注意的是这两种方式的区别和联系。首先，对于 `a++` 和 `++a` 这两种方式而言，对 `a` 的操作是完全一样的，都会在表达式运算结束之后把 `a` 的值加 1。这两种方式所不同的地方在于表达式的值不同。

例如，假设 `a = 5`，则不管执行 `a++` 还是 `++a`，执行之后 `a` 的值均为 6。所不同的是，`a++` 这个表达式的值为 5（即 `a` 加 1 之前的值），而 `++a` 这个表达式的值为 6（即 `a` 加 1 之后的值）。

`a--` 和 `--a` 有类似的关系。

#### ◆ 位运算符

Java 语言中包含了四种位运算符：按位与 (`&`)，按位或 (`|`)，按位异或 (`^`)，取反 `~`。位运算符主要用于对数据的每个二进制位进行运算。

首先，从逻辑上说，与、或、异或的逻辑运算如下表所示

运算值 1	运算值 2	与	或	异或
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

可以来这么记忆：与运算只有在两者都为 1 时结果才为 1，或运算只要有一个为 1 结果就为 1，异或运算两者不同结果为 1。

要计算一个位运算的表达式结果，先将十进制的运算数转化为二进制，然后再进行按位操作。

例如，对于十进制数 10 和 6，转化为二进制之后为 1010 和 0110。对其分别进行与、或和异或的操作：

$$\begin{array}{ccc} 1010 & 1010 & 1010 \\ \& 0110 & | 0110 & ^ 0110 \\ \hline 0010 & 1110 & 1100 \end{array}$$

因此计算所得的结果分别为：2、14、12。

`~` 操作对整数按位求反。例如下面的代码：

```
int i = 10;
i = ~i;
```

这段代码中，首先定义了一个 `int` 类型的变量 `i`。这个变量占 32 位（4 个字节），在内存中表示如下：

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

然后，`~i` 表示按位求反，原数位为 1 的转为 0，0 转为 1。得到的结果如下：

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

这个数表示十进制的数 -11。

还要注意的是，这四种位操作一般只用来操作整数。而与、或这两种运算符，除了能进行整数操作之外，还能进行 `boolean` 类型的操作。关于这一点，我们会在讲述 `boolean` 类型时进行更详细的阐述。

#### ◆ 移位操作

Java 中的移位运算符有三种：算术右移 (`>>`)，逻辑右移 (`>>>`) 和左移 (`<<`)。

移位操作是指，把一个整数的二进制表示形式，向某个方向（左或者右）移动，并按照一定的规律，丢弃和补充相应位上的值。

例如，对于十进制数 6（0110），进行右移 1 位运算，结果为 3（0011），右移 2 位结果

为 1 (0001)。可以看到，右移 1 位于除 2 的结果一致，右移 n 位与除  $2^n$  结果一致。而左移与右移相反，对于 6 (0110)，左移 1 位结果为 12 (1100)，左移两位结果为 24 (11000)。因此可以看出，左移 n 位与乘以  $2^n$  结果一致。

逻辑右移与算术右移的区别很微妙。对于  $n>>>m$  (逻辑右移) 与  $n>>m$  (算术右移) 来说，当 n 为正数时，两种运算的结果是一样的。所不同的是，当 n 为负数时，算术右移的结果为一个负数，并且是基本符合算术规律的 (所以它叫算术右移)，而逻辑右移的结果为一个正数。例如下面的代码：

```
int n = 12;  
System.out.println(n>>2); //结果为 3  
System.out.println(n>>>2); //结果也为 3  
n = -12;  
System.out.println(n>>2); //结果为负  
System.out.println(n>>>2); //结果为正!!!
```

运行结果如下：

```
D:\Book\chp2>javac TestOperator.java  
  
D:\Book\chp2>java TestOperator  
3  
3  
-3  
1073741821  
  
D:\Book\chp2>
```

产生这种不同的原因，与计算机中表示整数的方式有关。之前介绍过，在计算机中，保存一个整数时，整数的最高位用来表示符号。其中，符号位为 0 表示正数，1 表示负数。在 使用算术右移时，移动数值时不会改变符号位的值，因此正数移动之后，符号位为 0，负数移动之后，符号位依然是 1。这样，对一个整数进行算术右移之后，正数依然是正数，负数依然是负数。

然而 使用逻辑右移时，最高位总会补上 0。对于正数来说，符号位没有改变；而对于负数来说，符号位由 1 变成了 0。因此，使用逻辑右移时，所得到的结果总是正数。

#### ◆ 布尔运算

我们首先把所有布尔运算符分为两大类。

第一类运算符为：> (大于), >= (大于等于), < (小于), <= (小于等于), == (相等), != (不相等)，这些运算符都接受两个参数，返回一个布尔值，表示判断结果。需要注意的是判断相等 (==) 是两个等号，一定要把这个布尔运算和赋值号 (=) 区分开来。

第二类运算符为：与 (&&)、或 (||)、非 (!)，这些运算符只能接受两个布尔类型的运算数。非运算符表示取反，即 !true 结果为 false，而 !false 结果为 true。对于与运算和或运算，运算规则见下表：

运算数 1	运算数 2	与	或
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

布尔运算与 (&&) 和位运算与 (&) 具有类似的运算结果，而布尔运算 (||) 和位运算 (|) 也具有类似的运算结果。所不同的有两点：

1、布尔运算（`&&`和`||`）只能接受布尔值作为运算数，而位运算了除了能进行布尔值的运算之外，还能进行整数运算；

## 2、布尔运算具有短路特性。

什么是短路特性呢？对于与运算而言，如果运算数 1 的值为 `false`，则无论运算数 2 的值是 `true` 还是 `false`，结果一定是 `false`。因此，如果计算机遇到第一个运算数为 `false` 的与（`&&`）操作，则不会去查看第二个运算数而直接返回。这就是所谓的短路特性。

“与运算”能被 `false` 短路，相对应的，“或运算”能被 `true` 短路（如果第一个操作数为 `true`，则不管第二个操作数如何，结果一定为 `true`）。在 Java 中，布尔运算（`&&`和`||`）具有短路特性，而位运算（`&`和`|`）不具有短路特性。

短路特性和`++`（`--`）运算符结合在一起会产生很多有趣的式子。例如下面的式子：

```
int a = 4, b = 5;
boolean flag = (a++>4) && (b++>3);
```

请读者思考一下，这段代码运行结束之后，`flag`、变量 `a`、变量 `b` 的值分别是什么？

答案：`flag` 为 `false`，`a` 的值为 5，`b` 的值也为 5。

说明：

在上述代码中，有两个括号，这两个括号的值会由左向右依次执行。

首先，计算 `a++>4`。在计算这个式子时，首先会执行 `a++`，执行完这个表达式之后，`a` 的值变为 5。但是，`a++`这个表达式的值是 4，而 `4>4` 为 `false`，因此第一个括号中，表达式的值为 `false`。

之后，由于后面遇到的是“`&&`”运算符，这个运算符能够被 `false` 短路，因此，第一个括号中的表达式计算出是 `false` 之后，第二个括号中的代码不会执行，因此 `b++`这个代码没有被执行，因此 `b` 的值没有被改变。

可是，如果表达式变为：

```
int a = 4, b = 5;
boolean flag = (a++>4) & (b++>3);
```

由于位运算符`&`不具有短路特性，因此，尽管第一个表达式已经被计算出是 `false`，第二个括号中的代码依然会执行。最终的结果，`b` 的值会变化为 6。

## ◆ 三元操作符：

Java 中只有唯一的一个三元运算符，其基本语法如下：

布尔表达式 ? 表达式 1 : 表达式 2

说这个运算符是三元运算符，指的是这个运算符在使用时，能够接受三个部分参与运算。

从语法上说，第一部分是一个布尔表达式，第二、第三个部分分别是一个表达式。第一部分和第二部分用“`?`”隔开，第二部分和第三部分用“`:`”隔开。这三个部分构成一个完整的三元表达式。

第一部分布布尔表达式有一个值，这个值要么是 `true`，要么是 `false`。而表达式 1 有一个值，表达式 2 也有一个值。这三个值参与运算，最后能够得出整个三元表达式的值。

整个表达式的值由下面的原则确定：

如果布尔表达式的值为 `true`，则整个三元表达式的值为表达式 1 的值；如果布尔表达式的值为 `false`，则整个三元表达式的值为表达式 2 的值。例如下面的代码：

```
c = a>b ? a-b : b-a;
```

上面的代码，`a>b` 是布尔表达式，`a-b` 是表达式 1，`b-a` 是表达式 2。而整个表达式的值要么是 `a-b` 的值，要么是 `b-a` 的值。得到结果之后，再把计算所得的值赋值给变量 `c`。

那么上面的代码完成什么功能呢？我们分情况讨论。如果 `a` 大于 `b`，则`(a>b)`这个表达式

的值为 true，这样，整个表达式的值就是  $a-b$ 。如果  $a$  小于  $b$ ，则( $a < b$ )这个表达式的值为 false。因此，整个三元表达式的值就是  $b-a$ 。

综上所述，无论  $a$  和  $b$  的值是多少，上述代码都能让  $a$  和  $b$  这两个变量中，较大的变量减去较小的变量，并把获得的差赋值给  $c$  变量。

### ◆ 运算符的优先级

在我们小学学习四则混合运算的时候，老师曾经反复的跟我们说，一定要注意，先乘除，后加减。对于下面的式子：

$2 + 3 * 2$

这个式子要先计算  $3*2$ ，再把所得到的结果与 2 相加。在这个运算的过程中，“先乘除，后加减”，体现的就是运算符优先级的思想：乘除法的优先级比较高，如果有乘除运算的话，应当先计算。

在 Java 中，同样有运算符优先级的概念。我们把本章提到的运算符的优先级罗列如下：

优先级	运算符	说明
1	()	最高优先级
2	!, ~, ++, --	除了括号外，一元操作符优先级最高
3	*, /, %	
4	+, -	Java 中同样满足先乘除，后加减
5	<<, >>, >>>	移位操作
6	<, <=, >, >=	比大小
7	==, !=	判断是否相等
8	&	按位与
9	^	异或
10		按位或
11	&&	与（逻辑操作）
12		或（逻辑操作）
13	:?	三元操作
14	=, +=, -=, *=, ...	所有的赋值操作

在 Java 中进行计算时，会先进行优先级高的运算，再进行优先级低的运算。但是，圆括号 “( )” 的优先级是最高的，如果有圆括号的话，就先计算圆括号中的值。

在写代码的时候，一般不用刻意去记忆操作符的优先级。如果不能确定运算符的优先级，可以使用圆括号()来指定运算的顺序。

## 4.4 String 类型初探

String 类型不是八种基本类型的一种，从分类上说，String 类型是一种对象类型。但是 String 类型和普通的对象类型又不尽相同，作为一个比较基本的数据类型，它有一些区别于其他对象类型的特点。

String 类是 Java 语言中比较重要的一个类型，随着学习的深入，我们会对 String 类型的了解越来越深入。在本章中，我们主要介绍 String 类型的三个特点。

### 一、String 类型有字面值。

关于这一点，我们之前已经有过接触：String 类型的字面值为双引号 ("") 中包含的内

容。例如：

```
String str = "Hello World";
```

要注意的是单引号和双引号的区别。单引号用在字符类型上，而双引号用在字符串上。

例如：

```
char ch1 = 'A'; //单引号用在字符类型字面值上
char ch2 = 'AB'; //错误! char 类型不能表示多个字符
char ch3 = "A"; //错误!
String str1 = "A"; //表示只有一个字符的字符串
String str2 = "ABC"; //表示有多个字符的字符串
String str3 = 'A'; //错误!
```

## 二、String 类型支持加法运算。

字符串类型是对象类型中唯一支持加法运算的类型。字符串加法表示字符串的连接。例如：

```
String str1 = "Hello World";
String str2 = " Welcome";
// 字符串加法, 输出结果为: Hello World Welcome
System.out.println(str1 + str2);
```

需要注意的是字符串加法和字符加法之间的区别。字符串加法表示连接，而字符的加法表示字符编码数值的相加，两者有本质的不同。例如：

```
System.out.println("a" + "b"); //字符串加法, 输出 ab
System.out.println('a' + 'b'); //字符加法, 将 a 和 b 的编码相加, 输出一个整数, 由于'a'的编码为 97, 'b'的编码为 98, 因此将输出 195。
```

## 三、任何类型与 String 类型相加，结果都为 String 类型。

这个特性可以当做是 String 类型的“自动类型提升”。例如：

```
int i = 10;
System.out.println("1234" + i);
```

输出 123410，整数 i 会被自动“提升”为字符串“10”，其结果就是“1234”和“10”这两个字符串的连接结果。

## 4.5 局部变量

局部变量是 Java 中很重要的一个概念，我们首先来看局部变量的定义。

局部变量是指：在方法内部定义的变量。在目前，我们能接触到的所有变量都被定义在主方法内部，因此目前我们接触到的变量都是局部变量。

关于局部变量，Java 有三条规则：

### 一、先赋值，后使用。

关于这条规则，可以看以下这个例子：

```
public static void main(String args[]) {
    int a;
    System.out.println(a); // 编译时错误! 因为 a 没有被赋值
}
```

在 Java 语言中，任何一个局部变量都必须先被赋值过之后，才能使用。这一点和很多其他语言不同。

**二、局部变量有作用范围，其作用范围从定义的位置开始，到包含它的代码块结束。**

我们看下面两段代码的例子：

```
public static void main(String args[]) {  
    System.out.println(a); //编译错误!  
    int a;  
}  
  
public static void main(String args[]) {  
    {  
        int a;  
    }  
    System.out.println(a); //编译错误!  
}
```

这两段代码都会产生一个编译错误。对于第一个例子而言，输出语句在定义 a 变量之前就是用了 a，显然不在 a 变量作用范围之内，因此是一个编译错误。对于第二个例子而言，a 变量作用范围为代码块内部，而输出语句在代码块外部，因此也会产生一个编译错误。

### **三、重合范围内，两个局部变量不能有命名冲突。**

请看下面这段代码的例子：

```
01: public static void main(String args[]) {  
02:     int a = 10;  
03:     {  
04:         int a = 20;  
05:         System.out.println(a); //编译错误!  
06:     }  
07:     System.out.println(a);  
08: }
```

这段代码会产生一个编译时错误。产生错误的原因，是由于在 main 方法中定义的第一个变量 a，其作用范围为 2~8 行，而第二个 a 变量作用范围为 4~6 行。因而在 4~6 行，有两个局部变量重名，则会产生一个编译时错误。

要修改也简单，可以把代码改成：

```
01: public static void main(String args[]) {  
02:     {  
03:         int a = 20;  
04:         System.out.println(a);  
05:     }  
06:     int a = 10;  
07:     System.out.println(a);  
08: }
```

这样，两个局部变量虽然重名，但是作用范围没有交集，也就不存在上述的问题。

## **本章小结**

在这一章中，我们给读者介绍了 Java 语言中的一些基本的语法知识。

注释是我们学习的第一个语法点，希望读者能够在编程过程中，能够从一开始就养成良

好注释的习惯。

包是我们学习的第二个语法点。实际工作中，几乎每一个程序都会处于特定的包下，从而能避免名字冲突，并且更加利于工程管理。从规范的角度而言，希望大家能够逐渐养成使用包的良好习惯。

还有一个与规范相关的知识点，是我们提到的标识符命名规范。只有使用了正确且规范的标识符命名，才能使我们的程序具有更高的可读性。为此，强烈要求读者严格的按照标识符命名规范的要求写程序。

除了一些基本的程序规范之外，本章的重点是 Java 中一些关于变量的基本知识。希望读者通过本章的学习，能掌握 8 种基本类型和 String 类型的一些基本操作，掌握自动类型提升的概念和原则，掌握运算符的用法以及局部变量的概念和使用的原则。

# Chp3 流程控制

## 本章导读

本章主要介绍 Java 中一些基本的流程控制语句，例如分支判断、循环、基本函数的洗发以及数组的知识。

在介绍流程控制的过程之前，我们首先先介绍一个预备知识，如何从命令行上读入一个数据。

## 0 读入数据

在本章的最后，简单为大家介绍一下 Java 中如何读入数据。Java1.5 中，有一个非常简单的用来读入数据的类：java.util.Scanner，使用这个类能够非常容易的读入数据。

使用时的代码如下：

```
//引入 Scanner 类
import java.util.Scanner;
public class TestScanner{
    public static void main(String args[]){
        //下面这行代码创建了一个 Scanner 对象
        //可以理解为，这行代码为读入数据做准备
        Scanner sc = new Scanner(System.in);

        System.out.print("请输入一个字符串: ");
        //读入一行字符串，可以使用 sc.nextLine() 语句
        String str = sc.nextLine();
        System.out.println(str + " 收到了!");

        System.out.print("请输入一个整数: ");
        //读入整数时，使用 sc.nextInt() 语句
        int n = sc.nextInt();

        System.out.print("请输入一个小数: ");
        //读入浮点数，可以使用 sc.nextDouble() 语句
        double d = sc.nextDouble();

        System.out.println(n * d);

    }
}
```

运行时，可以在根据提示，在控制台上进行输入。结果如下：

```
D:\Book\chp3>javac TestScanner.java

D:\Book\chp3>java TestScanner
请输入一个字符串: hello world
hello world 收到了!
请输入一个整数: 10
请输入一个小数: 2.6
26.0

D:\Book\chp3>
```

根据提示，可以在命令行上输入字符串、整数和小数。通过 `java.util.Scanner` 类，我们就实现了在命令行上读入数据的功能。

## 1 分支结构

### 1.1 if 语句

我们之前写的所有代码，都是从头到尾依次执行，没有任何判断。但是，光有顺序执行的功能是不行的，请看下面的代码：

```
import java.util.Scanner;
public class TestDivide{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int a = 10;
        int b = sc.nextInt();
        System.out.println(a/b);
    }
}
```

这段代码读取一个整数 `b` 的值，然后计算 `a/b`。在大部分情况下，代码执行都没有问题。例如，我们输入 2，结果如下：

```
D:\Book\chp3>javac TestDivide.java

D:\Book\chp3>java TestDivide
2
5

D:\Book\chp3>
```

然而当 `b` 的值为 0 时，会产生一个错误。如下图所示：

```
D:\Book\chp3>javac TestDivide.java

D:\Book\chp3>java TestDivide
0
Exception in thread "main" java.lang.ArithmaticException: / by zero
        at TestDivide.main(TestDivide.java:7)

D:\Book\chp3>
```

因此，仅仅顺序执行代码，已经无法满足要求了。我们需要对 `b` 的值进行判断：如果输

入的 b 的值不为 0，则可以执行 a/b；如果 b 的值为 0，则应当给出用户更加明确的提示，而不应该出现“Exception”之类的不太友好的错误信息。

### 1.1.1 if 语句的基本语法

为了能够完成上述的功能，我们引入了 if 语句。if 语句是最基本的分支结构之一，可以用来控制程序的执行。具体的说，if 语句表示能够对某些条件进行判断，根据是否满足特定的条件，让程序执行不同的代码。

最基础的 if 语句语法结构为：

```
if (布尔表达式) {  
    代码块 1  
} else {  
    代码块 2  
}
```

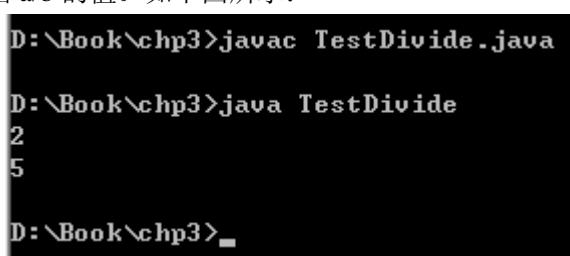
if 关键字后面跟一对圆括号，圆括号中是一个布尔表达式。所谓的布尔表达式，指的是值为 boolean 类型的表达式。例如， $a == 5$ 、 $b >= 3$ ，以及 $(a>4)&&(b<5)$ 等，都是布尔表达式。

if 语句会对布尔表达式的值进行判断。当布尔表达式值为 true，执行代码块 1；当布尔表达式为 false 时，执行代码块 2。

有了 if 语句，我们就可以对前面的 TestDivide.java 文件进行改写。修改之后的代码如下：

```
import java.util.Scanner;  
public class TestDivide{  
    public static void main(String args[]){  
        Scanner sc = new Scanner(System.in);  
        int a = 10;  
        int b = sc.nextInt();  
        if (b != 0){  
            System.out.println(a/b);  
        } else{  
            System.out.println("b 不能为 0");  
        }  
    }  
}
```

上面的代码，对 b 是否为 0 进行了判断。假设我们首先输入了 2，此时，b 的值不为 0，所以  $b \neq 0$  这个表达式的值为 true。根据 if 语句的特点，会执行第一个代码块中的内容，输出 a/b 的值。如下图所示：



```
D:\Book\chp3>javac TestDivide.java  
D:\Book\chp3>java TestDivide  
2  
5  
D:\Book\chp3>
```

当我们输入 0 时，此时 b 的值为 0， $b \neq 0$  这个表达式的值为 false。因此，会执行 if 语句第二个代码块中的内容，输出“b 不能为 0”。如下图所示：

```
D:\Book\chp3>java TestDivide
0
b不能为0

D:\Book\chp3>
```

这样，使用 if 语句，我们就能够对用户输入的值进行判断。根据判断结果的不同，执行不同的代码。

需要注意的是，在代码块 1 和代码块 2 中，都可能包含不止一个 Java 语句。例如，下面的代码：

```
import java.util.Scanner;
public class TestBiggerThanTen{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        if (n > 10){
            System.out.println("statement 1");
            System.out.println("statement 2");
        }else{
            System.out.println("statement 3");
            System.out.println("statement 4");
        }
    }
}
```

上面这段代码读入一个整数，如果这个整数大于 10，则输出 statement 1 和 statement 2，如下图所示：

```
D:\Book\chp3>javac TestBiggerThanTen.java

D:\Book\chp3>java TestBiggerThanTen
12
statement 1
statement 2

D:\Book\chp3>
```

当输入的整数小于或等于 10 时，则输出 statement3 和 statement4。如下图所示：

```
D:\Book\chp3>java TestBiggerThanTen
8
statement 3
statement 4

D:\Book\chp3>
```

上面我们介绍了 if 语句的基本形式。而除了基本形式之外，if 语句还有两种变化的形式。

第一，else 语句可以省略。当我们只需要当条件为真时执行某些操作，而条件为假时不需要任何操作时，就可以省略 else。例如，我们读入一个整数，然后取这个整数的绝对值。如果这个整数是正数或 0，则不需要对这个整数做任何的操作。只有在这个整数小于 0 时，才需要对其进行处理。

代码如下：

```
import java.util.Scanner;
public class TestAbs{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        if (n < 0){
            n = -n;
        }
        System.out.println("绝对值为" + n);
    }
}
```

上面的代码，当 `n` 为负数时，`n` 的绝对值为 `-n`。而当 `n` 为正数和 0 时，由于绝对值就是 `n` 本身，因此不需要执行任何代码。所以，上面的程序中，`else` 部分可以省略。

运行结果如下：

```
D:\Book\chp3>javac TestAbs.java
D:\Book\chp3>java TestAbs
100
绝对值为100

D:\Book\chp3>java TestAbs
-99
绝对值为99

D:\Book\chp3>
```

除了可以省略 `else` 部分之外，`if` 语句还有另外一种变化。这就是，当 `if` 或者 `else` 后面的代码块中只有一个语句时，花括号可以省略。例如我们第一个程序：

```
if (b != 0){
    System.out.println(a/b);
}else{
    System.out.println("b 不能为 0");
}
```

这个代码中，每个代码块中都只有一个语句，因此，花括号可以省略，写成下面的形式：

```
if (b != 0)
    System.out.println(a/b);
else
    System.out.println("b 不能为 0");
```

这两种形式的效果是一样的。但是，对于初学者来说，强烈建议大家无论在什么情况下，都保留花括号，这样能够避免初学者犯很多低级错误。

另外，需要注意的是，无论 `if` 和 `else` 的代码块中有多少个语句，一个 `if...else` 构成一个语句。也就是说，无论代码块的内容多么复杂，一个 `if...else` 只算是一句话。

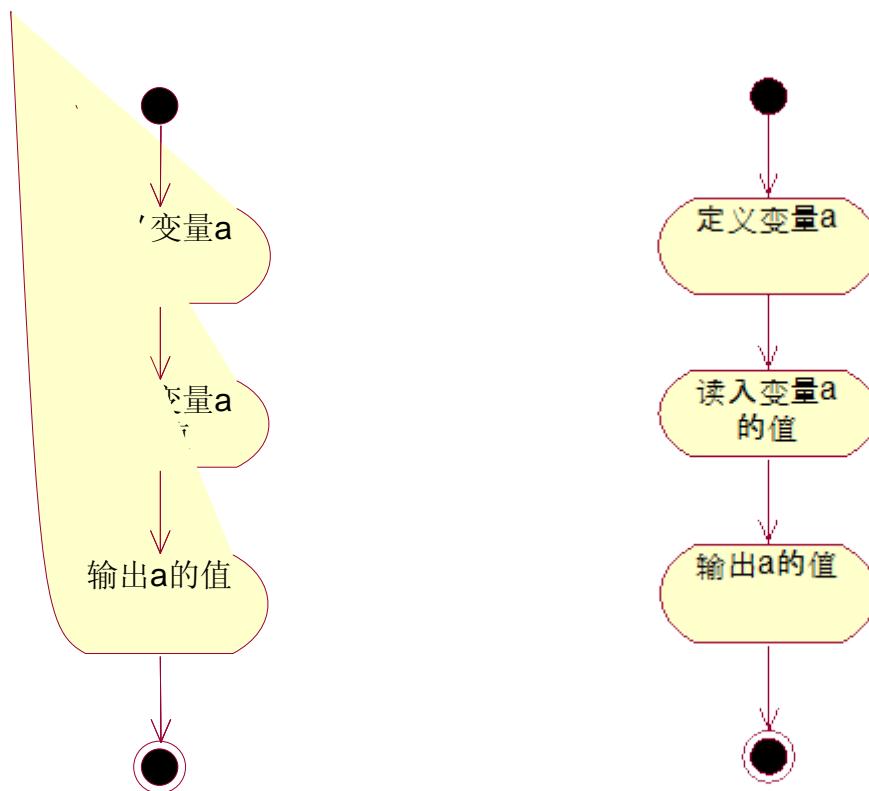
## 1.1.2 流程图

程序执行的流程有很多种。例如，有些代码是依次顺序执行的，有些代码要满足特定的条件才能执行，有些代码会循环的执行。为了能够直观而清晰的描述程序执行的流程，我们可以绘制流程图。

流程图可以用来表示程序执行的流程。首先，最基本的流程：顺序执行的流程。例如下面的代码：

```
int a;  
a = sc.nextInt();  
System.out.println(a);
```

这段代码顺序执行了这些操作：定义变量、读取变量的值、输出变量。可以用下面的流程图来表示这段代码的执行：



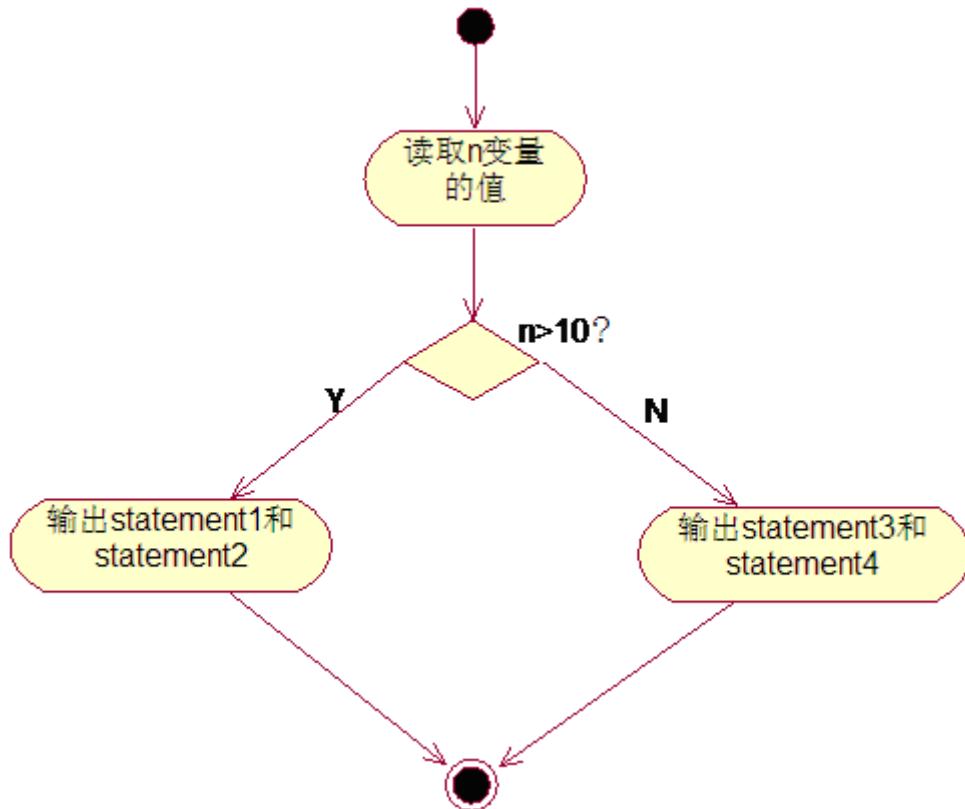
在上面的图形中，程序的开始用实心的黑色圆点表示，而程序的终止用黑色圆点加一个圆环来表示。另外，每一个框表示一个步骤，每个步骤包括一个或多个语句。通过箭头，表示从一个步骤跳到下一个步骤。

对于 if 语句来说，我们可以用一个菱形来表示判断。例如，下面的代码：

```
int n = sc.nextInt();  
if (n > 10){  
    System.out.println("statement 1");  
    System.out.println("statement 2");  
}else{
```

```
System.out.println("statement 3");
System.out.println("statement 4");
}
```

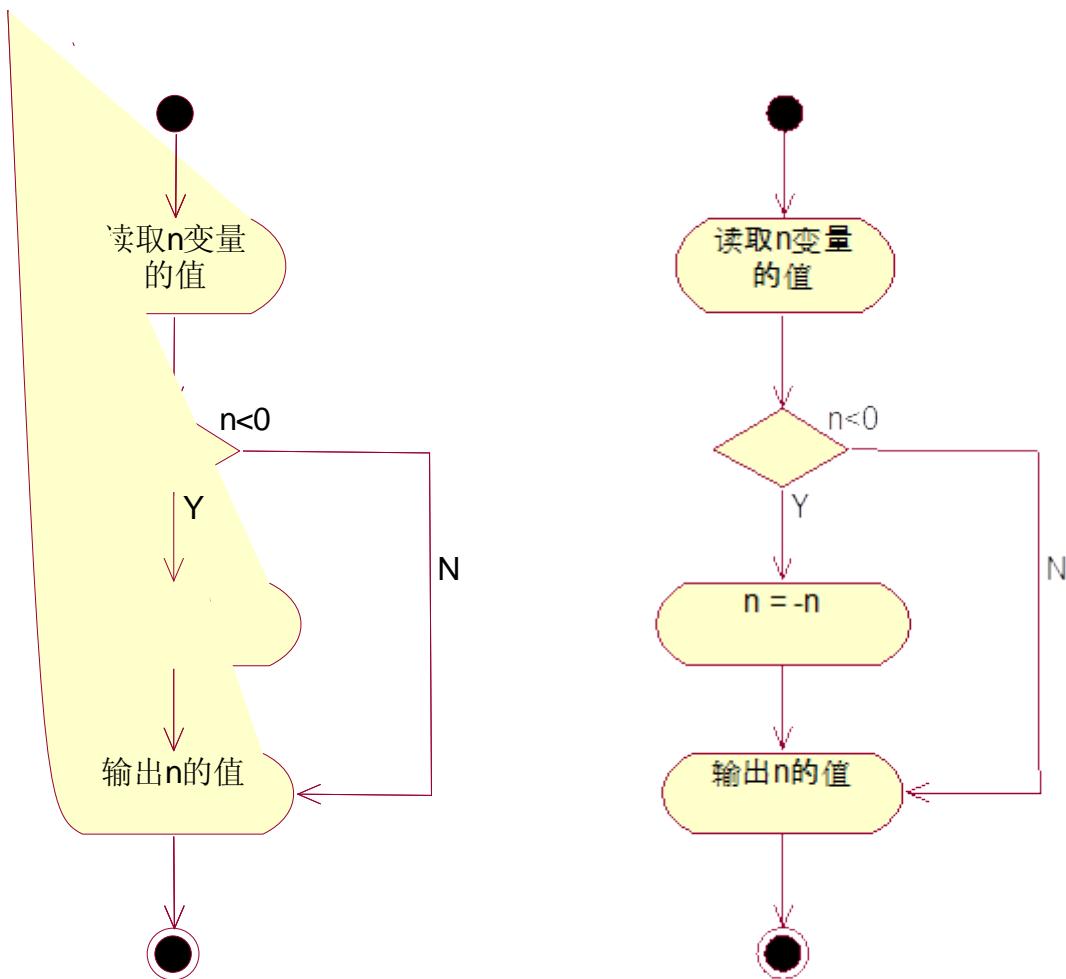
上面的代码，用流程图表示如下：



顺序执行的流程，在 if 语句的位置，被分成了两路：一路为 Y，表示的是当条件为真时的流程；另一路为 N，表示当条件为假时的流程。

if 语句通过对某些条件进行判断，能够把一条顺序执行的流程，分成两条支路。因此，if 语句也叫做分支结构。

而有些 if 语句会省略 else，例如前面提到的求绝对值的程序。用流程图表示如下：



### 1.1.3 用 if 语句做多重分支

通过上面介绍的内容，我们可以使用 if 语句对某些条件进行判断，根据判断的结果是 true 还是 false，来程序执行不同的代码。

然而，有些时候，判断的结果并不仅仅是 true 或者 false。例如，我们读入一个 0~100 之间的整数，表示学生的成绩。我们要写一个程序，对成绩进行评级：0~59 评为 E，60~69 评为 D，70~79 评为 C，80~89 评为 B，90~100 评为 A。我们要对表示成绩的这个整数区分出 5 种不同的情况。

这种类型的程序，我们需要进行多次判断。先定义一个 int 类型的变量 score，并从命令行上读入用户的输入，代码片段如下：

```
Scanner sc = new Scanner(System.in);
int n = sc.nextInt();
```

首先，应当判断这个整数是否是大于等于 0 并且小于 60。如果满足这个条件的话，则学生的成绩是 E 级，输出 E；

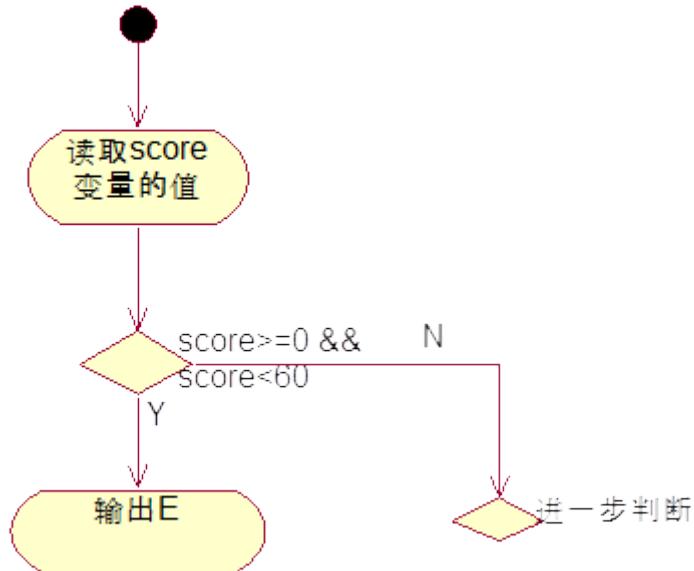
如果不满足这个条件的话，则说明学生的成绩大于 59。而学生成绩如果大于 59 分，则又有很多种不同的情况，需要继续判断。

代码片段如下：

```
if (score>=0 && score <60) { // 如果学生成绩大于等于 0 小于 60，则输出 E
```

```
    System.out.println("E");
} else{ //否则，需要进行进一步的判断
    ...
}
```

流程图如下：

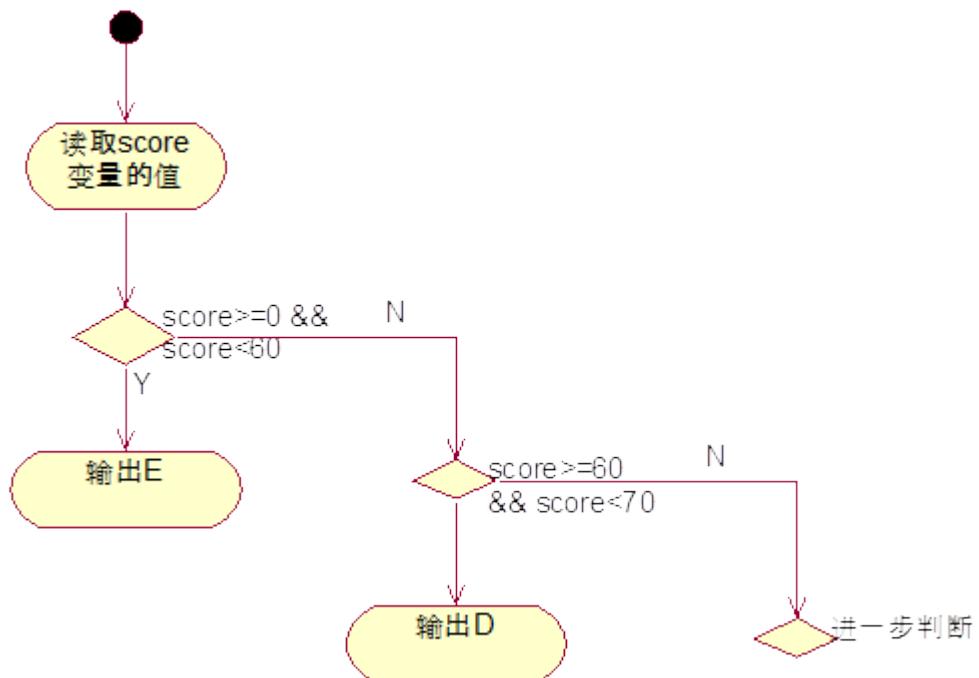


在 `else` 语句中，我们进行下一步的判断。接下来，我们判断 `score` 是否大于等于 60 小于 70。如果是，则可以确定学生成绩的等级是 D 级，可以利用 `System.out.println()` 方法输出 D；如果不是，则说明学生的成绩大于等于 70 分，还存在多种可能性，需进行更进一步的判断。

代码片段如下：

```
if (score >= 0 && score < 60) {
    System.out.println("E");
} else{
    if (score >= 60 && score < 70) {
        System.out.println("D");
    } else{
        ...
    }
}
```

流程图如下：



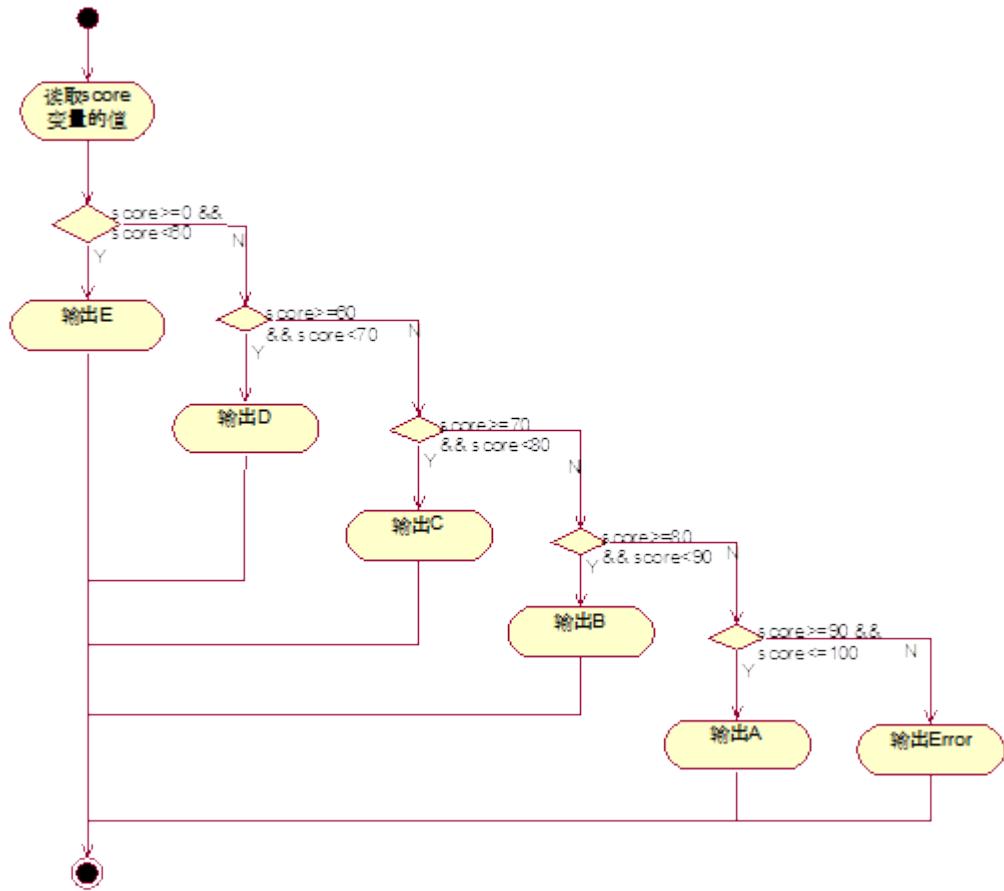
依此类推，完整的代码如下：

```

if (score>0 && score<60) {
    System.out.println("E");
} else{
    if (score>=60 && score<70) {
        System.out.println("D");
    } else{
        if (score>=70 && score<80) {
            System.out.println("C");
        } else{
            if (score>=80 && score<90) {
                System.out.println("B");
            } else{
                if (score>=90 && score <=100) {
                    System.out.println("A");
                } else{
                    System.out.println("Error");
                }
            }
        }
    }
}
}

```

完整的流程图如下：



注意，最后一个 else 语句，只有当前面所有判断都为 false 的时候才执行。也就是说，只有当 score 小于 0 或者大于 100 时才会执行这个 else 语句。

接下来，我们调整一下代码结构。有如下代码片段：

```

if (score >= 0 && score < 60) {
    System.out.println("E");
} else {
    if (score >= 60 && score < 70) {
        System.out.println("D");
    } else {
        ...
    }
}

```

在这段代码中，外层的 else 语句（字体加粗的部分），其代码块中，只有一个 if...else 语句。由于一个完整的 if...else 语句只能算一个语句，因此，我们可以把外层 else 代码块的花括号省略，并调整缩进，写成下面的形式：

```

if (score >= 0 && score < 60) {
    System.out.println("E");
} else if (score >= 60 && score < 70) {

```

```

        System.out.println("D");
}else{
    if (score>=70 && score<80) {
        System.out.println("C");
    }else{
        ...
    }
}

```

现在，字体加粗部分 else 语句，其代码块的花括号已经被省略。而在带下划线的 else 语句的代码块中，也只有一个 if...else 语句。我们可以省略掉这个 else 的花括号，并调整缩进：

```

if (score>=0 && score <60) {
    System.out.println("E");
}else if (score>=60 && score < 70) {
    System.out.println("D");
}else if (score>=70 && score<80) {
    System.out.println("C");
}else{
    ...
}

```

依次类推，最终代码的结构会被调整为：

```

if (score>0 && score<60) {
    System.out.println("E");
}else if (score>=60 && score<70) {
    System.out.println("D");
}else if (score>=70 && score<80) {
    System.out.println("C");
}else if (score>=80 && score<90) {
    System.out.println("B");
}else if (score>=90 && score <=100) {
    System.out.println("A");
}else{
    System.out.println("Error");
}

```

经过调整之后，代码看起来简洁和清晰了很多。这种 if...else if...else，是一种常用的结构。这种结构的语法如下：

```

if (条件 1) {
    条件 1 的代码块
}else if (条件 2) {
    条件 2 的代码块
}else if (条件 3) {
    条件 3 的代码块
}

```

```
...
else{
    所有条件都不满足时的代码块
}
```

如果有多个并列的条件，需要根据这些的条件执行不同的代码，这个时候，我们就可以使用 if ...else if 这个结构。

这个结构中，由一个 if 语句开头，判断第一个条件；中间有多个 else if 语句，每个 else if 语句中判断一个条件；最后有一个 else 语句，当所有条件都不满足时，执行这个 else 语句。

## 1.2 switch 语句

在生活中，我们可能都有过用手机开通业务的经验。在与服务台通话的过程中，可能会听到这样的部分：

“为手机充值请按 1，停开机请按 2，业务办理请按 3，人工服务请按 9，返回上一级菜单请按\*.....”

这样，根据我们按键的不同，手机服务台会进入不同的流程，并且根据值的不同，可能进入多个不同的流程。

这样，根据输入的值不同，进入多个不同的流程，这种程序结构称为多重分支。在 Java 中，我们可以用 switch 语句来完成多重分支。这种语句最基本的语法如下：

```
switch(表达式){
    case value1 :
        语句块 1;
    case value2 :
        语句块 2;
    case value3 :
        语句块 3;
    ...
    case valueN :
        语句块 N;
}
```

switch 语句会根据表达式值的不同，对每一个 case 所对应的 value 进行比较。如果有一个 value 与表达式的值相等，则流程跳转到该 case 语句的位置。

例如，我们从命令行上读入一个整数，这个整数的范围是 1~5，分别表示成绩的等级“A”~“E”。然后，根据这个整数的不同，输出该等级的成绩范围。例如，如果读入的值是 1，则表示“A”级，输出 90~100；如果读入的值时 3，则表示“C”级，输出 70~79。

利用 switch 语句，基本代码如下：

```
import java.util.Scanner;
public class TestSwitch{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        switch(n){
            case 1 :
                System.out.println("恭喜你！成绩不错！");
            case 2 :
                System.out.println("成绩一般般");
            case 3 :
                System.out.println("成绩及格");
            case 4 :
                System.out.println("成绩良好");
            case 5 :
                System.out.println("成绩优秀");
        }
    }
}
```

```

        System.out.println("90 ~ 100");
    case 2 :
        System.out.println("80 ~ 89");
    case 3 :
        System.out.println("70 ~ 79");
    case 4 :
        System.out.println("60 ~ 69");
    case 5 :
        System.out.println("0 ~ 59");
        System.out.println("不及格，要加油咯");
    }
}
}
}

```

在上面的代码中，switch语句会根据n的值的不同，跳转到不同的case的位置。例如，当n的值为1时，会跳转到case 1的位置，而当n的值为3时，会跳转到case 3的位置。另外，在每一个case后面跟的语句块中，都可以包含多个语句。例如，在上面的代码中，在case 1和case 5的语句块中，包含两个输出语句。

但是，上述的代码并不能完成我们的需求。例如，如果输入3，输出结果如下：

```

D:\Book\chp3>javac TestSwitch.java

D:\Book\chp3>java TestSwitch
3
70 ~ 79
60 ~ 69
0 ~ 59
不及格，要加油咯

D:\Book\chp3>

```

虽然，n的值为3时，switch语句会跳转到case 3的位置，输出70~79，符合我们的设想。但是，在输出完70~79之后，switch语句并没有结束，而是会从case 3的位置，继续往下执行，依次输出60~69，0~59，以及“不及格，要加油咯”。

很显然，这样的执行方式不满足我们的要求。我们希望当执行完一个case的语句块之后，就应该跳出switch语句，而不是继续执行。为了实现这种要求，我们需要在每一个case语句后面，增加一个语句：break。break语句能够跳出switch语句，不让程序继续向下执行。修改之后的switch代码如下：

```

switch(n){
    case 1 :
        System.out.println("恭喜你！成绩不错！");
        System.out.println("90 ~ 100");
        break;
    case 2 :
        System.out.println("80 ~ 89");
        break;
    case 3 :

```

```

        System.out.println("70 ~ 79");
        break;
    case 4 :
        System.out.println("60 ~ 69");
        break;
    case 5 :
        System.out.println("0 ~ 59");
        System.out.println("不及格, 要加油咯");
        break;
}

```

修改过的代码运行效果如下：

```

D:\Book\chp3>javac TestSwitch.java

D:\Book\chp3>java TestSwitch
3
70 ~ 79

D:\Book\chp3>_

```

现在，程序读入 3 之后，在 switch 语句中，会匹配到 case 3 的位置。然后，依次往下执行，执行 System.out.println("70 ~ 79"); 语句，输出 “70~79”。之后，执行 break 语句，跳出 switch 语句块，因此就不会继续输出 60~69, 0~59 等内容。

因此，从习惯上说，我们往往会在每一个 case 语句后面，都加上一个 break 语句，用来跳出 switch 代码块。此外，最后一个 case 语句的 break 语句可以省略，因为执行完最后一个 case 之后，后面没有其他的语句，程序会自动的跳出 switch 语句块。但是，为了让每个 case 的格式一致，不建议大家省略最后一个 break 语句。

如果输入的值超出 1~5 的范围，则没有一个 case 能够匹配这种情况。这样会直接跳出 switch 语句，没有任何的输出。例如，如果我们输入 10，则运行时结果如下：

```

D:\Book\chp3>java TestSwitch
10

D:\Book\chp3>_

```

为了在用户输入错误的情况下，给用户一个错误提示，我们需要在所有 case 都匹配不上时候，输出一个“输入错误”的提示。

在 switch 语句中，还有一个 default 的语法。语法如下：

```

switch(value) {
    case value1: XXX;
    case value2: XXX;
    ...
    default : 语句块;
}

```

有了 default 语句之后，在执行 switch 语句时，如果没有一个 case 的 value 值能够匹配上，就会执行 default 的语句块。

我们修改上面的程序，加入 default 语句。修改后的代码如下：

```
switch(n) {
```

```
case 1 :  
    System.out.println("恭喜你！成绩不错！");  
    System.out.println("90 ~ 100");  
    break;  
case 2 :  
    System.out.println("80 ~ 89");  
    break;  
case 3 :  
    System.out.println("70 ~ 79");  
    break;  
case 4 :  
    System.out.println("60 ~ 69");  
    break;  
case 5 :  
    System.out.println("0 ~ 59");  
    System.out.println("不及格，要加油咯");  
    break;  
default :  
    System.out.println("输入错误");  
    break;  
}
```

修改之后的代码执行时结果如下：

```
D:\Book\chp3>javac TestSwitch.java  
D:\Book\chp3>java TestSwitch  
10  
输入错误  
D:\Book\chp3>
```

要注意的是，`default` 语句的执行，与 `default` 的位置无关。例如，我们调整 `default` 语句的位置，把代码改成如下形式：

```
switch(n) {  
    case 1 :  
        System.out.println("恭喜你！成绩不错！");  
        System.out.println("90 ~ 100");  
        break;  
    case 2 :  
        System.out.println("80 ~ 89");  
        break;  
default :  
    System.out.println("输入错误");  
    break;  
    case 3 :  
        System.out.println("70 ~ 79");  
        break;
```

```

        case 4 :
            System.out.println("60 ~ 69");
            break;
        case 5 :
            System.out.println("0 ~ 59");
            System.out.println("不及格，要加油咯");
            break;
    }
}

```

我们把 default 语句放到 case 3 的前面，这个时候，如果输入 3，结果会是什么呢？编译运行结果如下：

```

D:\Book\chp3>javac TestSwitch.java
D:\Book\chp3>java TestSwitch
3
70 ~ 79
D:\Book\chp3>

```

可以看到，运行结果依然是输出“70~79”，并没有受到 default 语句的影响。因为 switch 语句在执行时，会首先比对所有的 case 语句，不管这些 case 语句在 default 语句之前还是在其之后。只有所有 case 都匹配不上时，才会执行 default 的代码块。例如上面的例子中，读入 3 之后，首先会比对 case 1、case 2，在代码遇到 default 之后，并不会马上执行，而是先比对 default 语句之后的 case 3。结果，由于读入的值为 3，能够跟 case 3 匹配上，因此输出“70~79”。

但是，从习惯上来说，我们往往会把 default 语句放在 switch 语句的末尾，这样能够提高代码的可读性。一般而言，switch 语句往往会写成如下形式：

```

switch(表达式) {
    case value1 :
        语句块 1;
        break;
    case value2 :
        语句块 2;
        break;
    case value3 :
        语句块 3;
        break;
    ...
    case valueN :
        语句块 N;
        break;
    default :
        default 语句块;
        break;
}

```

但是，switch 语句也有它的局限性。

首先，虽然 switch 语句能够做多重分支，但是做范围上的判断。例如，switch 可以表示读入的值为 1 时如何，为 2 时如何；但是却不能表示读入的值在 50~100 的范围之内应当如何。也就是说，我们用 case 1, case 2 这样的语句来表示匹配 1、2 这两个值，但是 switch 语句不支持如 case 50...100 这种语法，来表示范围。

如果想要对值的不同范围进行判断，则可以使用 if ... else if 的语法。我们之前写的，根据学生成绩判断学生成绩等级的例子，就是一个非常典型的对不同范围的值进行不同的操作。这种逻辑无法使用 switch 语句完成，只能用 if ... else if 这样的语法。

此外，switch 语句只能够判断四种类型的值：byte、short、int 或 char。也就是说，在 switch 语句的圆括号中，表达式的值只能是 byte、short、int 和 char 类型。这四种类型被称为与 int 兼容的类型，你可以这么来理解：根据自动类型提升，这四种类型相互运算时，会自动提升为 int 类型。

## 2 循环控制

循环是计算机程序当中最重要的一个特性之一，毕竟，当初人们发明计算机，就是想把一些重复而机械的劳动交给计算机来完成，而把一些比较需要创造的部分交给人来完成。

比如，我们在第一章中学会了怎样输出一个“HelloWorld”。那如果我们需要输出十个“HelloWorld”怎么办？这时候，也许还能够写 10 个输出语句。但是如果我们需要输出更多个 Hello World，比如 100 万个。显然，写 100 万个输出语句是不现实的。计算机更加擅长这种重复而机械的劳动，我们应当把这种反复的输出让计算机来完成。这就需要在编程时，掌握循环语句的使用。

首先，我们来对循环的概念进行分析。对于任何一个循环，我们都应该从四个方面来思考它：1、初始化；2、循环条件；3、循环体；4、迭代操作。

**初始化**，指的是在循环开始之前，所需要做的准备工作。比如，生活中，“包饺子”这件事情，就是一个重复的劳动，我们可以把“包饺子”当做是循环。但是，要真正开始包饺子，必须做好充分的准备工作，例如和面，擀皮，和馅……等等。

**循环条件**，往往是一个布尔表达式，当这个表达式为真时，循环继续执行；而当这个表达式为假时，循环退出。也就是说，循环条件指的是，控制循环是否能够继续的条件。例如，在包饺子这个循环中，循环条件就是：剩余的饺子皮>0 && 剩余的饺子馅>0。

**循环体**，指的是需要反复执行的重复性的操作。例如，包饺子的循环体，就是重复性的包饺子劳动，把饺子馅放在皮中间，蘸水，把饺子捏成元宝状。完成这些操作之后，一个饺子就包完了，然后开始包下一个饺子。

**迭代操作**，往往是体现每次执行循环体之间所产生的变化，也可以理解为每次执行循环体时所产生的不同。举例来说，在包饺子的过程中，每包一个饺子，会产生什么变化呢？产生的变化就是，剩余的饺子皮少了一张，剩余的饺子馅少了一块。

迭代操作往往是和循环条件联系在一起的，在迭代操作时，往往会修改循环条件中使用的变量值。例如，在我们的包饺子循环中，迭代操作就操作了剩余的饺子皮和饺子馅的数量，而循环条件正是对这两个量进行判断。

要注意的是，并不是每一种循环都具有循环的四个要素。有些循环里，循环条件和迭代操作是同一个语句；有些循环里循环体和迭代操作是一样的，等等。但是，在写循环的时候，我们应当对循环的四个要素进行通盘我们在这里罗列出循环的四个要素都进行考虑，避免遗漏。

介绍完循环的概念之后，下面要介绍的是 Java 中循环的语法。

在 Java 中有三种循环：for 循环、while 循环和 do...while 循环。下面我们依次来给大家阐述。

## 2.1 for 循环

for 循环包含循环的四个要素，语法如下：

```
for(初始化;循环条件;迭代操作){  
    循环体;  
}
```

举例来说，我们可以写一个 for 循环用来输出 10 个 HelloWorld。我们用一个变量 i，来统计总共循环了多少次，我们把这个 i 变量称为计数器。这样，循环的初始化，就应该把计数器清零。每当进行了一次循环之后，就把计数器+1，因此迭代操作，就是 i++。而循环条件，则是判断循环的次数：如果循环已经进行了 10 次，则结束循环；如果循环没有满 10 次，则循环继续。我们在判断循环的时候，只要把计数器 i 和 10 进行比较，就能判断是否执行循环。

由上面的分析，可以写出 for 循环的代码如下：

```
01: int i = 0;  
02: for(i = 0; i<10; i++){  
03:     System.out.println(i + " Hello World");  
04: }  
05: System.out.println(i);
```

首先，在 01 行，程序定义了一个变量 i，这个变量用作计数器。

之后，在 02 行开始进入 for 循环。

在 for 循环中，首先执行 i=0 的初始化操作，把计数器清零。要注意的是，初始化操作只执行一遍；

然后，进行循环条件的判断。此时 i 值为 0， i<10 判断为真，循环继续；

之后，执行循环体，输出 0 Hello World；

当循环体一次执行结束，执行迭代操作 i++，此时 i 值为 1，表示循环已经完成了一次；

然后，再进行循环条件的判断，此时 i 为 1， i<10 判断为真，循环继续；

执行循环体，输出 1 Hello World；

执行迭代操作 i++，此时，i 的值为 2；

进行循环条件的判断。由于此时 i 的值为 2， i<10 判断为真，循环继续

.....

依次类推，直到最后一次，当 i=9 时，输出 9 Hello World；

执行迭代操作， i++。此时，i 的值为 10，表示已经循环了 10 次。

再进行循环条件的判断。由于此时 i 的值为 10，因此 i<10 判断为假，循环退出。

退出循环之后，输出 i 的值：10。

至此，i 从 0 变化到 10，其中当 i 从 0 变化到 9 时各输出一个 HelloWorld，循环体总共执行了 10 次，而循环条件的判断则进行了 11 次。

完整的代码如下：

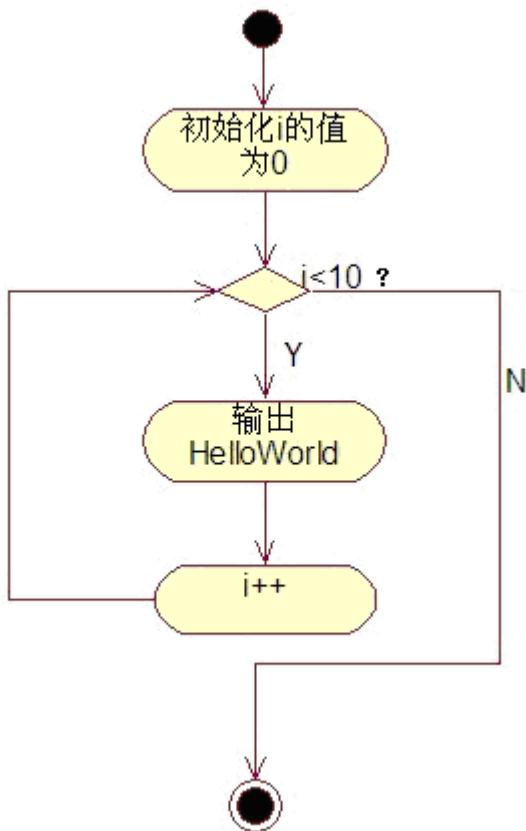
```
public class TestFor{  
    public static void main(String args[]){  
        for(int i = 0; i<10; i++){  
            System.out.println(i + " Hello World");  
    }
```

```
    }  
}  
}
```

运行结果如下：

```
D:\Book\chp3>javac TestFor.java  
  
D:\Book\chp3>java TestFor  
0 Hello World  
1 Hello World  
2 Hello World  
3 Hello World  
4 Hello World  
5 Hello World  
6 Hello World  
7 Hello World  
8 Hello World  
9 Hello World  
  
D:\Book\chp3>
```

for 循环的流程图如下：



在这个程序中，我们定义 *i* 变量的主要目的，就是用来控制循环的次数。用来控制循环的变量，被称为循环变量。

一般，程序员会把循环变量命名为 *i*、*j*、*k*，因此，这三个名字，建议大家除了作为循环变量之外，尽量不要乱用。当一个有经验的程序员看到这三个字母时，自然而然的会想到循环变量，这个时候如果这些变量被拿来做其他的用途，可能会对程序员造成误导。

此外，往往循环变量仅仅是用来控制循环，而一旦脱离循环之后就没有什么价值，因此，

我们能够在初始化的时候再定义这个变量。这样一来，这个变量的作用返回就被局限在循环的内部。例如：

```
//注意， i 变量在初始化被定义
for(int i = 0;i<10; i++) {
    System.out.println(i + " Hello World");
}
// ! System.out.println(i); 编译错误！找不到符号 i
```

在 for 循环的初始化部分定义变量，也是一种非常常见的写法。

在处理循环变量时，还要注意一点：在写 for 循环时一定要注意循环变量的范围。例如，请快速回答下面的问题：

1、int i=1; i<10; i++ 循环几次？

答案：i 的变化范围 1~9，循环 9 次

2、int i = 0; i<=10 循环几次？

答案：i 的变化范围 0~10，循环 11 次

3、int i = 1; i<=10; i+=2 循环几次？

答案，i 的变化范围 1~9，取值分别为 1、3、5、7、9，循环 5 次。

在写 for 循环的时候，一定要仔细分析循环变量的变化，明确循环总共执行了多少次。

另外，还要注意，**循环变量最好不要在循环体中进行赋值**。请看下面的这个程序：

```
//注意， i 变量在初始化被定义
for(int i = 0;i<10; i++) {
    System.out.println(i + " Hello World");
    i = 4;
}
```

这个程序输出几个 Hello World？

答案是：无数个！

当第一次循环时，输出 0 Hello World，之后 i 被赋值为 4，迭代操作之后 i 的值为 5；当第二次循环时，输出 5 Hello World，之后 i 又被赋值为 4！这样，i 的值永远在 4~5 之间变化，从而永远无法退出循环！

这个错误是典型的死循环。所谓的死循环，指的是永远无法退出的循环。往往程序中出现死循环，意味着程序逻辑上有严重的错误。

在上述代码中，产生死循环的原因，是因为在循环体中对 i 进行了赋值，从而使得循环条件永远为真，循环永远无法退出。我们应当尽量避免在 for 循环的循环体中对循环变量进行赋值操作。

虽然应该避免在 for 循环的循环体中对循环变量赋值，但是完全可以在循环体中读取循环变量的值。例如下面这个练习：

求出  $\text{sum} = 1 + 2 + 3 + \dots + 50$

对于这个练习，我们可以用下面的步骤来进行计算。

首先，让 sum 的值为 0；

其次，让 sum 的值为原有值+1，sum 的结果为 1；

然后，让 sum 的值在原有的基础上+2，sum 的结果为  $1+2=3$ ；  
再然后，让 sum 的值在原有的基础上+3，sum 的结果为  $3+3=6$   
.....  
最后，让 sum 的值在原有的基础上+50。

我们可以看到，在这个计算过程中，每一步都是让 sum 的值，在原有的基础上加上一个值。假设我们用一个变量 i 来表示这个值，则每一次进行的操作就是：

```
sum = sum + i;
```

其中，i 变量的变化范围是从 1~50。为此，我们可以设计一个 for 循环，在 for 循环中，让循环变量 i 从 1 循环到 50，表示这 50 个加数。循环体就是  $sum = sum + i$ 。

示例代码如下：

```
public class TestSum{  
    public static void main(String args[]){  
        int sum = 0;  
        for (int i = 1; i<=50; i++){  
            sum = sum + i;  
        }  
        System.out.println(sum);  
    }  
}
```

运行结果如下：

```
D:\Book\chp3>javac TestSum.java  
  
D:\Book\chp3>java TestSum  
1275  
  
D:\Book\chp3>
```

特别要注意一下，应当把 sum 变量的定义写在 for 循环外面。原因也很简单，for 循环是用来计算的，我们输出 sum 变量计算结果，应当在 for 循环结束之后。由于我们要在 for 循环之外继续使用 sum 变量，因此必须在 for 循环之外定义 sum 变量。

## 2.2 while 循环和 do...while 循环

While 循环和 do...while 循环比较类似，我们先来看这两种循环的语法。

While 循环：

```
while(循环条件){  
    循环体  
}
```

Do...while 循环：

```
do{  
    循环体  
}while(循环条件);
```

对于这两种循环而言，初始化部分并不是语法的一部分。但是作为一个良好的编程习惯，强烈建议读者在写循环的时候，在循环的上面加上初始化的代码。

另外。对于这两种循环而言，循环条件的含义是一样的：这是一个布尔表达式，当表达式为 true 时，循环继续；当表达式为 false 时，循环退出。

最后，对于两种循环而言，循环体的含义也类似，都表示反复执行的那部分操作。要注意的是，在 while 循环和 do...while 循环中，没有单独的地方写迭代操作。如果需要进行迭代操作的话，应当把迭代操作写在循环体中。

例如，我们分别使用 while 循环和 do...while 循环来完成输出 10 个 HelloWorld 的程序。

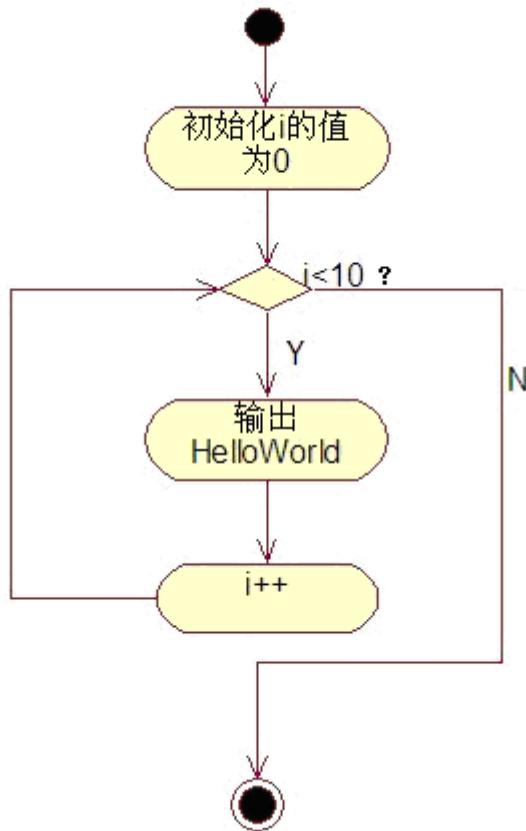
```
//使用 while 循环输出 10 个 Hello World
public class TestWhile{
    public static void main(String args[]){
        int i = 0;
        while(i <10){
            System.out.println(i + " Hello World");
            i++;
        }
    }
}

//使用 do...while 循环输出 10 个 Hello World
public class TestDoWhile{
    public static void main(String args[]){
        int i = 0;
        do{
            System.out.println(i + " Hello World");
            i++;
        }while(i <10);
    }
}
```

输出结果与 for 循环输出的结果类似，在此不再赘述。

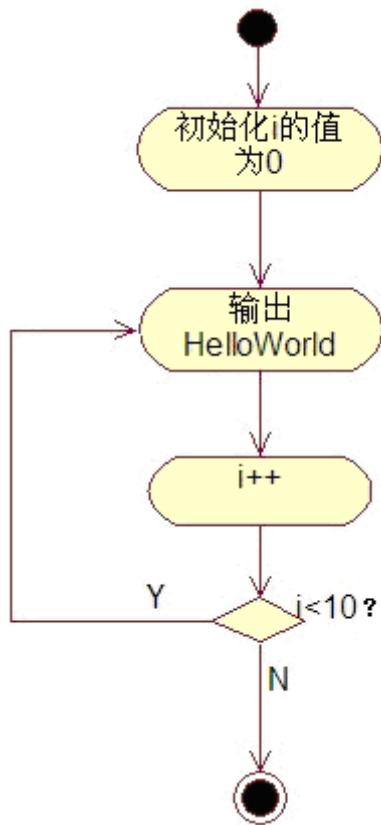
这两段代码的执行流程类似，只有一个细小的差别：对于 while 循环而言，是先进行判断，后执行循环体；而对于 do...while 循环而言，是先执行循环体，后执行判断。

while 循环的流程图如下：



可以看到，我们写的 while 循环程序的流程图，和 for 循环的流程图一样。

do...while 循环的流程图如下：



可以看出，与 while 循环不同，do...while 循环的判断，是在循环体之后执行的。这个

差别对于上面的例子而言，是循环条件判断次数的差别。

`while` 循环的循环条件，在 `i` 为 0~10 的过程中，共判断了 11 次；而 `do...while` 循环的循环条件，在 `i` 为 0 时没有判断。因为是先执行循环体，后执行判断，因此当 `i` 等于 0 时，会先执行循环体，在循环体中，执行了 `i++`。因此，`do...while` 循环执行第一次判断时，`i` 的值为 1。在整个循环过程中，循环条件的判断，当 `i` 的值为 1~10 的变化过程中，总共判断了 10 次。

由于 `while` 循环因为在第一次执行循环体之前，就要进行判断，因此循环体有可能一次都不执行；而 `do...while` 循环由于要执行一次迭代以后才进行判断，因此循环体至少会执行一次。例如，有如下代码：

```
public class TestLoop{  
    public static void main(String args[]){  
        int i = 100;  
        while (i < 10){  
            System.out.println(i + " Hello World");  
            i++;  
        }  
    }  
}
```

这段代码，在进入循环体之前，会先进行判断。此时，由于 `i` 的值为 100，循环条件 `i<10` 为假，因此，循环体一次都不执行，程序没有任何输出。运行结果如下：

```
D:\Book\chp3>javac TestLoop.java  
D:\Book\chp3>java TestLoop  
D:\Book\chp3>
```

而如果把上述代码改成 `do...while` 循环的写法，如下：

```
public class TestLoop2{  
    public static void main(String args[]){  
        int i = 100;  
        do {  
            System.out.println(i + " Hello World");  
            i++;  
        } while (i < 10);  
    }  
}
```

刚开始 `i=100` 时，没有进行判断，就进入了循环体。在循环体中，输出 100 Hello World，然后执行 `i++`。此时，`i` 的值为 101，循环条件 `i<10` 结果为假，于是退出循环。运行结果如下：

```
D:\Book\chp3>javac TestLoop2.java  
D:\Book\chp3>java TestLoop2  
100 Hello World  
D:\Book\chp3>
```

因此，我们可以看出，`while` 循环有可能一次都不执行，而与之对应的，`do...while` 循环

至少执行一次。

关于三种循环的基本语法，就介绍完了。我们可以看到，同样是输出 10 个 Hello World，用这三种循环都可以完成。既然三种方式都可以完成循环的操作，那在编程时，我们应当选择哪一种循环呢？以下是一些提示：

1) 能够确定次数的循环，应当用 for 循环。相反，如果循环的次数不能确定，则应当使用 while 循环或者 do...while 循环。例如，如果要输出 100 个 Hello World，这个循环我们能够确定其要循环 100 次，因此应当使用 for 循环的方式。而相反，假设我们要读取文件的所有内容，一次读取一个字节。由于我们不知道文件的长度究竟有多少，因此，不知道需要循环多少次。此时，就可以使用 while 循环，模拟代码如下：

```
while(文件中还有数据){  
    读取一个字节;  
}
```

2) 如果要对循环变量进行赋值操作，则应当使用 while 循环，而避免使用 for 循环。之前介绍过，应当尽量避免在 for 循环的循环体中对循环变量进行赋值操作。如果确实需要在循环体中修改循环变量的值，则应当使用 while 循环。

3) 如果循环体至少需要执行一次，则应当使用 do...while 循环。

## 2.3 break 和 continue

除了基本的循环之外，Java 还提供了循环中的 break 和 continue 语句。这两个语句能够帮助程序员对循环进行更加灵活的控制。

break 语句表示跳出当前的循环。例如：

```
01: public class TestBreak{  
02:     public static void main(String args[]){  
03:         for(int i = 0; i<=5; i++){  
04:             if (i == 3) break;  
05:             System.out.println("i="+i);  
06:         }  
07:     }  
08: }
```

程序的第 04 行，进行了一个判断，当 i 为 3 时执行 break 语句。在执行 for 循环过程中，i 的值为 0~2 时，判断为假，因此会执行循环体后面的输出语句，分别输出

```
i=0  
i=1  
i=2
```

而当 i 为 3 时，循环执行 break 语句。此时，会跳出 for 循环。由于 for 循环后面没有其他的代码，因此程序结束。程序运行结果如下：

```
D:\Book\chp3>javac TestBreak.java  
D:\Book\chp3>java TestBreak  
i=0  
i=1  
i=2  
D:\Book\chp3>
```

`continue` 语句表示跳出本“次”循环。所谓的本次循环，是指的，`continue` 语句会跳到循环体的末尾，然后执行迭代操作，之后，再进行循环条件的判断。也就是说，使用 `continue` 语句不会跳出整个循环，只是跳过这一轮的循环。例如下面的代码：

```
public class TestContinue{  
    public static void main(String args[]){  
        for(int i = 0; i<=5; i++){  
            if (i == 3) continue;  
            System.out.println("i="+i);  
        }  
    }  
}
```

前面，程序正常输出 `i=0, i=1, i=2`。当 `i` 为 3 时，执行 `continue` 语句。此时，代码会跳到 `for` 循环循环体的末尾，跳过输出语句。然后，执行迭代操作 `i++`，`i` 的值为 4，程序继续运行。因此，在最后的结果中，除了 `i=3` 被跳过之外，其他的部分都正常输出。

运行结果如下：

```
D:\Book\chp3>javac TestContinue.java  
D:\Book\chp3>java TestContinue  
i=0  
i=1  
i=2  
i=4  
i=5  
D:\Book\chp3>
```

## 2.4 多重循环

考虑下面这个练习：从命令行上读入一个正整数，根据这个正整数，输出下面的图形：例如，当 `n = 3` 时，输出：

```
*  
**  
***  
  
n = 4 时，输出  
*  
**  
***  
****
```

这个练习如何完成呢？思路如下：

对于任何一个正整数  $n$ , 都必须循环  $n$  次, 这样才能输出  $n$  行。因此, 必须写一个 for 循环, 在循环中定义一个变量  $i$ , 让  $i$  变量从 1 循环到  $n$ ; 而在循环中每一轮循环都输出第  $i$  行。这样, 经过  $n$  次循环, 最终会输出  $n$  行。循环如下:

```
for(int i = 1; i<=n; i++) {  
    //循环每次迭代输出第 i 行  
}
```

下面考虑循环体。我们可以发现规律: 对于第 1 行, 需要输出 1 个\*号, 然后换行; 第 2 行, 需要输出 2 个\*号, 然后换行……以此类推, 第  $i$  次执行循环体时, 需要输出  $i$  个\*号, 以及一个换行符。

为了输出  $i$  个\*号, 可以考虑写一个循环。这个循环的循环体每次输出一个\*号。这样, 通过控制循环的次数, 就可以控制输出的\*号的个数。

由于要输出  $i$  个\*号, 因此, 需要循环  $i$  次。我们可以定义一个变量  $j$ , 让  $j$  从 1 循环到  $i$ , 代码如下:

```
for(int j = 1; j<=i; j++) {  
    System.out.print("*");  
}
```

把两部分结合起来, 代码如下:

```
01:for(int i = 1; i<=n; i++) {  
02:    for(int j = 1; j<=i; j++) {  
03:        System.out.print("*");  
04:    }  
05:    System.out.println();  
06:}
```

上面的代码, 在外层循环的基础上, 又嵌套了一个内层循环。这种结构被称为循环的嵌套。如果嵌套只有两层, 则被称为二重循环。如果有多层次循环之间嵌套, 则被称为多重循环。

假设  $n$  的值为 3, 输出一个三行的三角形。在执行上面的代码过程中, 首先进入 01 行。此时,  $i$  的值为 1,  $i \leq n$  的判断为真, 执行循环。由此, 进入外层循环的循环体, 范围是 2~5 行。

当代码执行到 02 行的时候, 进入了内层循环。此时, 在内层循环中定义了一个变量  $j$ , 其作用范围是内层循环的内部, 因此作用范围是 2~4 行。此时,  $j$  的值为初始化时给出的值 1, 而  $i$  的值也为 1, 这样,  $j \leq i$  的值为真, 执行内层循环的循环体。

执行 03 行, 输出一个 "\*" 之后, 程序运行到 04 行。此时, 内层循环的循环体结束, 因此, 要执行内层循环的迭代操作:  $j++$ 。此时,  $j$  的值为 2。

然后, 再进行内层循环的循环条件判断。此时  $j$  的值为 2,  $i$  的值为 1,  $j \leq i$  的值为假, 因此内层循环退出。

内层循环退出之后, 程序从第 4 行继续往下执行, 执行到第 5 行并输出一个换行符。需要注意的是, 此时已经不再 2~4 行的范围之内, 已经在  $j$  变量的作用范围之外。也可以理解为, 这个时候,  $j$  变量不存在了。

然后, 程序进入第 6 行, 这意味着外层循环的循环体执行完了一遍。这个时候, 需要执行外层循环的迭代操作:  $i++$ 。此时,  $i$  的值为 2。

接下来, 进行外层循环的循环条件判断。此时  $i \leq n$  的值为真, 外层循环继续。然后, 循环进入第 2 行。

此时, 在第 2 行中, 再一次进入了内层循环。在进入这个内层循环的时候, 又重新定义了一个变量  $j$ 。要注意, 这个变量  $j$  与第一次进入内层循环时定义的变量, 不是同一个。再

次执行内层循环。此时，由于 i 的值是 2，因此内层循环执行两遍，输出两个“\*”号。

第三次进入内层循环的情况类似，在此不再赘述。

完整代码如下：

```
import java.util.Scanner;
public class TestStar{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        for(int i = 1; i<=n; i++){
            for(int j = 1; j<=i; j++){
                System.out.print("*");
            }
            System.out.println();
        }
    }
}
```

运行结果如下：

```
D:\Book\chp3>javac TestStar.java

D:\Book\chp3>java TestStar
3
*
**
***

D:\Book\chp3>java TestStar
4
*
**
***
****
```

## 2.4 多重循环下的 break 和 continue

有了多重循环之后，break 和 continue 语句就显得更加复杂。我们首先来看一个二重循环的代码：

```
public class TestBreakContinue{
    public static void main(String args[]){
        for(int i = 1; i<=3; i++){
            for(int j = 1; j<=4; j++){
                System.out.println("i=" + i + " j= " + j);
            }
        }
    }
}
```

```
}
```

这段代码的输出结果如下：

```
D:\Book\chp3>javac TestBreakContinue.java  
D:\Book\chp3>java TestBreakContinue  
i=1 j= 1  
i=1 j= 2  
i=1 j= 3  
i=1 j= 4  
i=2 j= 1  
i=2 j= 2  
i=2 j= 3  
i=2 j= 4  
i=3 j= 1  
i=3 j= 2  
i=3 j= 3  
i=3 j= 4
```

这是一个典型的多重循环。接下来，我们在内层的循环中，增加一个 break 语句。修改后的代码片段如下：

```
01: for(int i = 1; i<=3; i++) {  
02:     for(int j = 1; j<=4; j++) {  
03:         if (j == 3) break;  
04:         System.out.println("i=" + i + " j= " + j);  
05:     }  
06: }
```

我们来分析一下程序执行的过程。首先，进入 01 行之后，初始化 i 变量，并把其值设置为 1。然后，执行外层循环的循环体，进入 02 行，执行内层循环。此时的 i 变量值为 1。当 j 的值为 1~2 时，输出：

```
i=1 j=1  
i=1 j=2
```

然后，当 j 为 3 时，判断的结果为真，执行 break 语句，跳出循环。要注意的是，跳出循环时，跳出的是内层循环，因此，程序跳出内层循环的范围 2~5 行，跳转到第 6 行。

第 6 行是外层循环中，循环体的末尾。因此，此处会执行外层循环的迭代操作：i++，然后进行外层循环的条件判断。由于此时 i 的值为 2，因此 i<=3 的值为真，循环继续。从而，程序再次进入内层循环。

由上面的分析我们得知，**默认情况下，break 语句只能跳出一层循环。如果 break 语句在内层循环中，则只能跳出内层循环，而无法直接跳出外层循环。**

程序运行的结果如下：

```
D:\Book\chp3>javac TestBreakContinue.java  
D:\Book\chp3>java TestBreakContinue  
i=1 j= 1  
i=1 j= 2  
i=2 j= 1  
i=2 j= 2  
i=3 j= 1  
i=3 j= 2  
  
D:\Book\chp3>
```

与 break 语句类似，continue 语句在默认情况下，也只能对内层循环执行 continue。例如下面的例子：

```
public class TestBreakContinue{  
    public static void main(String args[]){  
        for(int i = 1; i<=3; i++){  
            for(int j = 1; j<=4; j++){  
                if (j==3) continue;  
                System.out.println("i=" + i + " j= " + j);  
            }  
        }  
    }  
}
```

在这段代码中，内层循环的 continue，只能对内层循环起作用。因此，当 i 为 1 时，程序会输出

```
i=1 j=1  
i=1 j=2  
i=1 j=4←
```

运行结果如下：

```
D:\Book\chp3>javac TestBreakContinue.java  
C:\Users\EWENHUO\Downloads\workspace\book\corejava\chp3>java TestBreakContinue  
i=1j=1  
i=1j=2  
i=1j=4  
i=2j=1  
i=2j=2  
i=2j=4  
i=3j=1  
i=3j=2  
i=3j=4  
  
D:\Book\chp3>
```

那有没有办法，能够让 break 语句一下跳出多层循环呢？能不能有办法让 continue 语句对外层循环起作用呢？

接下来，我们介绍一下带标签的 break 和 continue。首先以 break 语句为例，完成能够跳出外层循环的代码。

首先，为了在 break 语句中区分内层和外层循环，我们首先应该给这两层循环分别起个名字。这个名字，就是循环的标签。

我们可以在循环之前加一个标签，这个标签用来区分循环。代码如下：

```
outer:for(int i = 1; i<=3; i++) {
    inner:for(int j = 1; j<=4; j++) {
        if (j == 3) break;
        System.out.println("i=" + i + " j= " + j);
    }
}
```

通过上面的操作，我们就把外层循环加上标签 outer，把内层循环加上标签 inner。然后，在内存循环的 break 语句处，为了说明要跳出的是外层循环，可以为 break 语句明确指明要跳出的是 outer 循环。代码如下：

```
outer:for(int i = 1; i<=3; i++) {
    inner:for(int j = 1; j<=4; j++) {
        if (j == 3) break outer;
        System.out.println("i=" + i + " j= " + j);
    }
}
```

这样，我们就能够明确指明要跳出 outer 循环。执行的结果如下：

```
D:\Book\chp3>javac TestBreakContinue.java
D:\Book\chp3>java TestBreakContinue
i=1 j= 1
i=1 j= 2
```

我们可以看到，当 i 为 1，j 为 3 的时候，执行 break outer。这样，就跳出了外层循环。结果，就只输出了两行，程序就结束了。

同样的，continue 也有类似的使用方式。我们把上述的代码修改如下：

```
outer:for(int i = 1; i<=3; i++) {
    inner:for(int j = 1; j<=4; j++) {
        if (j == 3) continue outer;
        System.out.println("i=" + i + " j= " + j);
    }
}
```

这样，执行 continue 的时候，会让 continue 语句对 outer 标签起作用。因此，程序会跳转到外层循环的最末尾，然后执行外层循环的迭代操作。运行结果如下：

```
D:\Book\chp3>javac TestBreakContinue.java  
D:\Book\chp3>java TestBreakContinue  
i=1 j= 1  
i=1 j= 2  
i=2 j= 1  
i=2 j= 2  
i=3 j= 1  
i=3 j= 2  
D:\Book\chp3>
```

# Chp4 函数

函数是计算机编程中非常重要的部分，是编程中最基本的元素之一。函数表示的是一种通用的过程，这种过程能够对外界提供服务。例如，现实生活中，ATM 取款机上有不同的功能，我们可以理解为 ATM 机上具有不同的函数可以调用；我们在 ATM 机上取钱，就可以理解为我们在 ATM 机上调用了“取钱”函数。在这种关系中，我们是“取钱”函数的调用者，“取钱”函数为我们提供服务。

## 1 函数的基本使用

### 1.1 函数的三要素

对于函数而言，最重要的部分就是函数的三要素：返回值、函数名、参数表；这三个部分被称之为函数三要素。

**返回值**，这个概念表示调用函数之后，函数会返回什么数据给调用者；

**函数名**，顾名思义，这表示函数的名字；

**参数表**，表示调用函数时所给的参数是什么，也就是说，调用函数时需要给函数哪些“输入”。

以“取钱”函数为例，函数的返回值为“现金”，我们作为调用者调用“取钱”函数，目的就是获得这个函数的返回值“钱”；这个函数的参数表表示我们对调用“取钱”时应该给这个函数传递的参数，取钱时需要“银行卡、密码、取款金额”等一系列参数。

函数三要素表明了函数的基本特性，从这方面来说，函数三要素是设计、实现函数最重要的部分。

在 Java 中，函数（Function）也被称之为方法（Method）。Java 中并不区分这两个概念，因此，本书中“函数”和“方法”指的是同一个意思。

在 Java 中定义函数同样需要考虑函数的三要素，但是 Java 中对函数的定义远远超越三要素这么简单。后面我们将详细介绍 Java 中的函数。

### 1.2 函数的定义

首先我们介绍 Java 中函数的定义。

在 Java 中，函数定义的位置为：类的里面，其他函数的外面。例如下面的代码：

```
//1  
public class TestFunction{  
    //2  
    public static void main(String args[]){  
        //3  
    }  
    //4  
}
```

对于上面//1、//2、//3、//4 四个位置而言，只有//2 和//4 的位置能够定义函数。另外，定义了函数可以在主函数中调用。而不论这个函数是定义在主函数之前，还是定义在主函数之后，都能够进行调用。也就是说，一个函数定义在//2 的位置，或者定义在//4 的位置，在函数的定义和使用上，没有任何区别。对于函数来说，只要满足“类的里面，其他

函数的外面”这个要求，在定义的顺序方面是没有要求的。

在 Java 中定义一个函数时，首先可以先写两个单词：**public static**。这两个单词为 Java 中的修饰符。加上这两个修饰符是为了能在主函数中正常的调用。至于这两个修饰符在修饰函数时是什么含义，会在后面的课程中详细为大家阐述。

在 **public static** 之后，就是函数的三要素。假设我们要写一个 **add** 函数，该函数接受两个 **int** 类型作为参数，并且返回这两个参数的和。这样，可以定义 **add** 函数如下：

```
public static int add(int a, int b)
```

第一个 **int** 为返回值类型，表示 **add** 函数返回一个 **int** 值。**add** 是函数名，**add** 后面的圆括号是参数表。

参数表中，可以定义 0 个或多个参数。在函数参数表中定义的参数，被称为“形式参数”，简称形参。从语法上说，形参是特殊的局部变量。一方面，在参数表中定义形参，就好像定义局部变量一样，应当写出变量的类型和变量名。另一方面，在形参也有其作用返回，形参的作用范围就是函数的内部。例如，在上面的代码中，我们定义了 **a** 和 **b** 两个形参，这两个形参的作用范围就是 **add** 函数内部。

写参数表的时候还要注意，如果这个函数接受多个参数，则多个参数之间用逗号隔开。例如上面 **add** 函数的例子，参数表就写成：**int a, int b**。需要注意的是，虽然函数的这两个形参类型一致，但是不能写成 **int a,b**。在定义多个形参的时候，每个形参的类型和参数名都应当完整的列出来。

如果调用一个函数时不需要参数，则参数表为空，在圆括号中什么内容都不写即可。例如，如果我们要写一个函数 **time**，用来表示当前是几点。这个函数不需要任何的参数，因此可以写成：

```
public static int getCurrentHour()
```

这个函数没有任何参数，因此，其参数表为空。

定义完函数之后，需要在函数后面紧跟一个代码块，这个代码块称为函数的实现。

**函数的定义**，表明的是函数应该如何使用。而**函数的实现**，则表明的是函数中真正执行的内容。

目前这一阶段，我们接触的所有函数里，函数的定义和函数的实现都是无法分离的。在函数定义完之后，必须加上一对花括号，在花括号中写上函数要执行的内容。完整的 **add** 函数如下：

```
public static int add(int a, int b){  
    int c = a + b;  
    return c;  
}
```

注意，在 **add** 函数中，包括一个“**return c**”的语句。这个语句是 **return** 语句，表示函数的返回值是什么。在 **add** 函数中，**return c** 表示返回值为 **c** 变量的值。

**return** 语句除了表示函数返回值之外，同样可以表示函数的流程跳转。这一部分内容留在下一小节中阐述。

### 1.3 函数的调用

写完 **add** 函数之后，就能在主函数里对其进行调用。例如：

```
01: public static void main(String args[]){  
02:     int m = 10, n = 20;  
03:     int result = add(m, n);
```

```
04:     System.out.println(result);
05:     add(30, 40);
06: }
```

这样，就在主函数中，调用了 add 函数。要注意的是：

第 03 行。在这一行中，我们把 add 函数作为赋值语句的一部分，因此会先调用 add 函数，然后把 add 函数的返回值赋值给 result 变量。在调用函数时，需要给出函数名和参数表。在上面的代码中，首先，明确的写出函数名 add；其次，在函数名后面写一对圆括号，在括号中给出调用函数时需要的参数 m 和 n。

这里，m 和 n 是传递给 add 函数的参数，被成为实际参数，简称“实参”。实参指的是，在调用函数的时候，为函数指定的参数。在函数调用的过程中，会把实参的值传递给形参。例如上面的代码中，m、n 就是调用函数时的实参，而 a、b 就是定义函数时的形参。在函数调用的时候，会把 m 变量的值传递给形参 a，而把 n 变量的值传递给形参 b。这样，在 add 函数内部进行计算的时候，两个形参 a、b 的值，就是调用方法时两个实参 m、n 的值。

最后，函数返回时，将计算所得的值返回。这个返回值又被赋值给 result 变量。这样就完成了一次函数的调用。

第 05 行，在这一行中，我们又一次调用了 add 函数，并且传入了不同的参数。参数除了可以用变量之外，同样可以使用字面值。另外，需要注意的，调用函数之后，函数的返回值没有被赋值给任何变量，因此函数的返回值没有被保存下来。

完整代码如下：

```
public class TestFunction{
    public static void main(String args[]){
        int m = 10, n = 20;
        int result = add(m, n);
        System.out.println(result);
        add(30, 40);
    }

    public static int add(int a, int b){
        int c = a + b;
        return c;
    }
}
```

运行结果如下：

```
D:\Book\chp4>javac TestFunction.java
D:\Book\chp4>java TestFunction
30
D:\Book\chp4>
```

注意到，由于第二次调用 add 函数时，没有保存其返回值，也没有把它的值输出。

### 1.3.1 用 return 语句返回值

下面，我们更加详细的来介绍一下 return 语句。return 语句有两层含义，一个含义就是我们之前提到的，return 语句表示返回一个值。

在函数的定义中，如果给出了返回值类型，则必须要返回一个相应类型的值。例如，由于在函数的定义中，add 函数返回值类型为 int。因此，在 add 函数的中必须要返回一个 int 类型的值。

例如，有下面的函数定义：

```
public static int m()
```

这个函数的返回值为 int 类型，表明这个函数必须要返回一个整数值。如果在这个函数中没有 return 语句，则编译会出错。例如，假设代码如下：

```
public static int m(){  
    System.out.println("m()");  
}
```

则编译时的结果如下：

```
D:\Book\chp4>javac TestReturn.java  
TestReturn.java:4: missing return statement  
    ^  
    ^  
1 error  
  
D:\Book\chp4>
```

编译器提示，在代码中缺少返回语句。

而且，return 语句返回的值，如果与函数定义中的返回值类型不同，也有可能出错。例如，在 m 方法中如果返回一个 double 类型的值，代码如下：

```
public static int m(){  
    System.out.println("m()");  
    return 1.5;  
}
```

则编译时的结果如下：

```
D:\Book\chp4>javac TestReturn.java  
TestReturn.java:4: possible loss of precision  
found   : double  
required: int  
        return 1.5;  
               ^  
1 error  
  
D:\Book\chp4>
```

编译器提示，可能损失精度。

怎么来理解这个过程呢？我们可以结合 add 函数来理解。add 函数的代码片段如下：

```
public static void main(String args[]){  
    ...  
    int result = add(m, n);  
    ...  
}  
public static int add(int a, int b){
```

```
    int c = a + b;  
    return c;  
}
```

在 add 函数的内部，定义了一个变量 c。这个变量是一个局部变量，c 的作用范围是 add 函数的内部。然后，return 语句中，返回了 c 的值，并在主方法中把这个返回值赋值给 result。然而，主方法并不在 c 变量的作用范围之内，因此，不能直接在主方法中输出 c。

那 c 的值是怎么返回的呢？可以这么来理解：在调用 add 函数的时候，由于 add 函数的签名中，说明这个 add 函数会返回一个 int 类型的值，因此，Java 会为 add 函数准备一个临时变量，变量的类型是 int，用这个临时变量来保存 add 函数的返回值。当执行到 return c 的时候，会把 c 变量的值赋值给这个临时变量。然后，在主方法的复制语句中，int result = add(m,n)，这就意味着把 add 函数的返回值赋值给 result，也就是把那个临时空间的值赋值给 result。

而在刚刚的 m 方法中，我们看到

```
public static int m(){  
    System.out.println("m()");  
    return 1.5;  
}
```

在方法中返回一个 double 类型的值。m 方法的定义中，说明返回值是一个 int 类型，因此编译器为 m 函数分配一个 int 类型的临时变量，用来保存返回值。但是，在调用 return 1.5 的时候，程序会试图把一个 double 类型的 1.5 赋值给一个 int 类型的临时变量，这样就会产生一个错误。

我们继续修改一下 m 函数：

```
public static int m(int arg){  
    System.out.println("m()");  
    if (arg == 10) return 0;  
}
```

这段代码依然有问题，编译时会产生一个编译时错误，错误信息如下：

```
D:\Book\chp4>javac TestReturn.java  
TestReturn.java:5: missing return statement  
    ^  
1 error  
D:\Book\chp4>
```

为什么会产生这个问题呢？原因在于，当我们定义了一个函数并指明其返回值为 int 类型之后，就要保证，无论使用什么参数调用这个函数，函数都能够返回一个 int 值。而上面的代码中，我们对 arg 进行了判断，如果这个参数的值为 10，则返回 0。那如果这个参数的值不为 10 呢？在这个函数的实现中没有说明。由于调用这个函数有可能没有返回值，因此，编译时会产生一个编译时错误。

为了解决这个问题，可以为 if 语句增加一个 else 代码块，并进行 return。修改后的代码如下：

```
public static int m(int arg){  
    System.out.println("m()");  
    if (arg == 10) return 0;
```

```
    else return 1;
}
```

### 1.3.2 流程跳转以及 void

上一小节我们介绍了 `return` 语句返回值。`return` 语句除了能够返回值之外，还能够控制流程的跳转。具体的说，在执行 `return` 语句的时候，被调用的函数会终止执行，并返回到函数的调用点上。例如，看下面的例子，我们修改 `TestFunction` 的代码如下：

```
01: public class TestFunction{
02:     public static void main(String args[]){
03:         System.out.println("Line 3");
04:         int m = 10, n = 20;
05:         int result = add(m, n);
06:         System.out.println(result);
07:         System.out.println("Line 7");
08:         add(10, 20);
09:         System.out.println("Line 9");
10:     }
11:     public static int add(int a, int b){
12:         System.out.println("Line 12");
13:         int c = a + b;
14:         System.out.println("Line 14");
15:         return c;
16:     }
17: }
```

在这个程序中，作为程序执行的入口，从主函数开始执行。首先输出 Line3。

之后，在第 5 行，程序调用 `add` 函数，因此程序从第 5 行跳转到 11 行，然后继续往下执行。依次执行的结果，输出为 Line12、Line14。

在 15 行，程序返回。这里，`return` 语句包含两层不同的含义：一表示返回值为 `c` 变量的值，二表示程序流程返回。所谓的流程返回，指的是函数返回到调用点上，即程序从 15 行跳转回第 5 行。

要注意的是，在 `add` 函数中定义了 `c` 变量，这是一个局部变量。同时，`add` 函数的两个形参 `a` 和 `b` 也相当于 `add` 函数的局部变量。在程序跳转之后，程序就在 `a`、`b`、`c` 这三个局部变量的作用范围之外，无法使用 `a`、`b`、`c` 的值。

此外，因为 `return` 语句具有流程返回的特性，因此，如果在 15 行之后增加一句：

```
System.out.println("come here");
```

由于这句话写在 `return` 语句之后，而永远不会被执行，因此这个语句会引起一个编译错误。

另外，有一类函数不需要返回任何值。例如，我们写一个函数，这个函数接受一个参数 `n`，根据 `n` 的值不同，来打印 `n` 个 Hello World。

很明显，这个函数需要一个参数 `n`。但是，这个函数由于只是完成打印功能，我们在调用这个函数的时候，不需要这个函数返回任何值。因此，在定义这个函数的时候，我们

要告诉编译器，这个函数不需要返回值。为此，我们可以把这个函数的返回值类型写成“**void**”类型。

代码如下：

```
public static void printHelloWorld(int n){  
    for(int i = 0; i<n; i++){  
        System.out.println("Hello World");  
    }  
}
```

注意，上面的代码中，在定义函数时，把函数的返回值定义为 **void**。这意味着这个函数不返回任何值。

因为 **void** 函数不返回值，因此，在 **void** 函数中，可以不包含 **return** 语句。例如上面的例子，在 **printHelloWorld** 这个函数中，没有 **return** 语句出现。当函数中所有的代码都执行结束时，函数自然会结束，然后就会返回函数的调用点上。

当然，**void** 函数中，也能够出现 **return** 语句。但是，此时，**void** 函数的 **return** 语句不能返回一个值，只能够表示流程的返回。例如，我们可以在 **printHelloWorld** 中增加如下代码：

```
import java.util.Scanner;  
public class TestVoid{  
    public static void main(String args[]){  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Input a number:");  
        int a = sc.nextInt();  
        printHelloWorld(a);  
    }  
    public static void printHelloWorld(int n){  
        if (n > 10) return;  
        for(int i = 0; i<n; i++){  
            System.out.println("Hello World");  
        }  
    }  
}
```

在 **printHelloWorld** 函数中，在进入 **for** 循环之前，会进行一次判断，判断参数 **n** 是否大于 10。如果参数 **n** 大于 10 的话，则程序会执行 **return** 语句。这个 **return** 语句如果被执行的话，则 **printHelloWorld** 函数立刻返回，不执行后面的 **for** 循环而直接返回到了主函数中。执行结果如下，当我们输入 15 时，**printHelloWorld** 中的 **return** 语句被执行，因此 **for** 循环不被执行而程序直接返回，输出结果如下：

```
D:\Book\chp4>javac TestVoid.java  
  
D:\Book\chp4>java TestVoid  
Input a number:15  
  
D:\Book\chp4>
```

当我们输入 6 时，**return** 语句不被执行，因此 **for** 循环被执行，输出 6 个 **HelloWorld**。执行结果如下：

```
D:\Book\chp4>java TestVoid  
Input a number:6  
Hello World  
D:\Book\chp4>
```

### 3 实参与形参

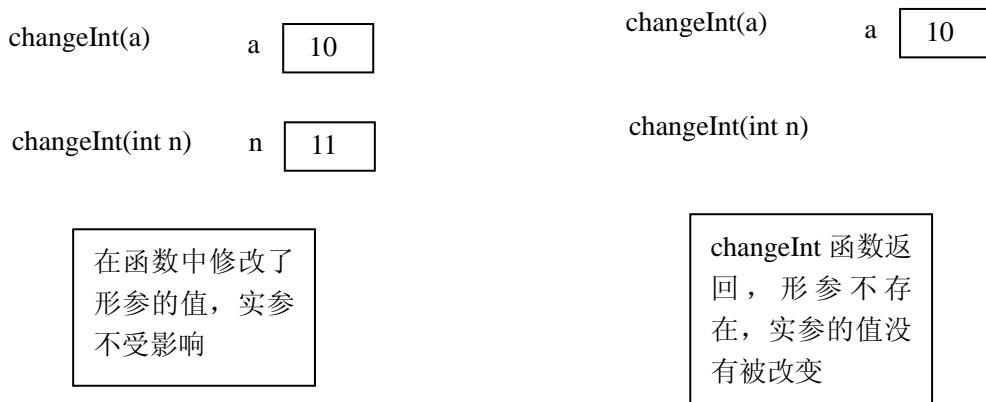
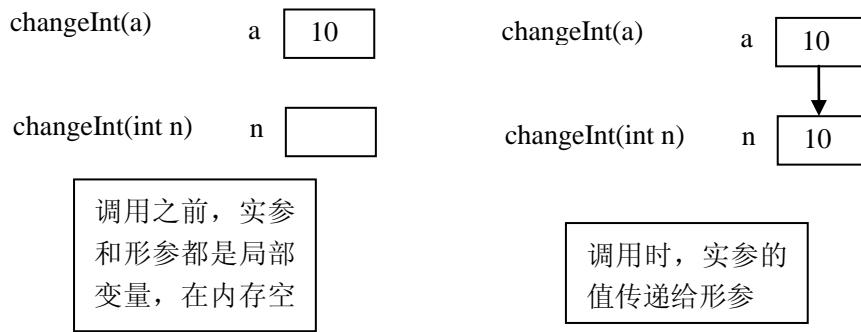
下面我们要为大家介绍的是实参和形参这两个概念。在上面的 TestFunction 程序中，实参就是 m, n，这两个参数是调用 add 函数时实际传递的参数。而在 add 函数的签名处，定义了两个参数(int a, int b)，这两个参数是所谓的形参。从本质上来说，形参相当于特殊的局部变量。例如，对于上面的 a、b 两个参数而言，这两个参数相当于 add 函数中定义的局部变量，它们的作用范围就是在 add 函数内部。在调用 add 函数的时候，会把实参的值传递给形参。

关于参数以及参数的传递，我们看下面这个例子：

```
public class TestParameter {  
    public static void main(String[] args) {  
        int a = 10;  
        changeInt(a);  
        System.out.println(a);  
    }  
  
    public static void changeInt(int n){  
        n++;  
    }  
}
```

编译运行这个程序，输出结果为 10。为什么会这样呢？明明在 changeInt() 函数中把参数加 1 了呀？

要注意的是，我们在调用函数的时候，实参传递给形参，是把实参的值传递给形参。如下图所示：



## 4 函数的嵌套调用

前面的内容, 我们为大家介绍了函数的一些基本使用。接下来, 要介绍的是函数中一些更加灵活的应用。看下面这个代码的例子:

```
public class TestNestedCall{
    public static void main(String args[]){
        System.out.println("main1");
        ma();
        System.out.println("main2");
    }

    public static void ma(){
        System.out.println("ma1");
        mb();
        System.out.println("ma2");
    }

    public static void mb(){
        System.out.println("mb1");
        System.out.println("mb2");
    }
}
```

```
    }  
}
```

我们来看一下上面这段代码的执行过程。在上面这段代码中，首先，是在 main 函数中输出“main1”，然后，在 main 函数中调用了 ma 方法，于是，流程从 main 中跳转到了 ma 函数中。在 ma 函数的第一个语句中，输出“ma1”。

然后，在 ma 方法中，又调用了 mb 方法。我们可以看到，在一个函数中，还可以调用另外一个函数，这种调用方式被称为函数的嵌套调用。在我们的这个例子中，主函数调用 ma 函数，ma 调用 mb 函数，示意图如下：

main → ma → mb

在 ma 调用 mb 函数之后，mb 函数输出“mb1”和“mb2”。然后，mb 函数中所有的代码都执行完了，mb 函数返回其调用点，也就是返回到 ma 方法中。然后，ma 方法继续执行，输出“ma2”。然后，ma 方法的代码也都全部完成，于是 ma 方法也返回调用点，也就是返回到主函数中。最后，主函数输出“main2”，主函数中所有代码都执行完毕，程序终止。

上述代码运行结果如下：

```
D:\Book\chp4>javac TestNestedCall.java  
  
D:\Book\chp4>java TestNestedCall  
main1  
ma1  
mb1  
mb2  
ma2  
main2
```

## 5 函数的递归调用

函数除了能够调用其他函数之外，还可以调用函数本身。这就是函数的递归调用。

我们来看一个例子。我们要写一个函数，这个函数接受一个整数 n 作为参数，然后输出这个整数的阶乘 n!。我们定义阶乘 n! 为：

$n! = n * (n-1) * (n-2) \dots * 1$

例如，5 的阶乘  $5! = 5 * 4 * 3 * 2 * 1 = 120$ 。

很显然，这个程序可以用循环来完成。在这儿，我们为大家介绍一种不用循环的方式，我们可以使用函数的递归来完成这个程序。

首先，由于我们要写一个计算阶乘的函数，因此，我们定义函数如下：

```
public static int factorial(int n)
```

这个函数接受一个参数，用来计算一个整数 n 的阶乘。那我们怎么实现呢？考虑到有下面这个数学规律：

$n! = n * [(n-1) * (n-2) * (n-3) \dots * 1] = n * (n-1)!$

也就是说，我们的函数可以写成这样的步骤：

```
public static int factorial(int n){ // 计算 n 的阶乘  
    //第一步，计算 (n-1)的阶乘  
    //第二步，把上一步计算所得的结果乘以 n 返回
```

```
}
```

注意到，第一步要完成的事情，就是计算 n-1 的阶乘。由于我们这个函数，就是用来计算 n 的阶乘。因此，第一步也可以理解为，以 n-1 为参数，调用 factorial。代码如下：

```
public static int factorial(int n){ // 计算 n 的阶乘  
    //第一步，计算 (n-1)的阶乘  
    factorial(n-1);  
    //第二步，把上一步计算所得的结果乘以 n 返回  
}
```

在这个函数中，出现了在 factorial 函数内部调用 factorial 函数的情况。这种函数自身调用自身的情况，我们就把它称为函数的递归调用。

我们完善上面的代码，把第二部也翻译成代码：

```
public static int factorial(int n){ // 计算 n 的阶乘  
    //第一步，计算 (n-1)的阶乘  
    int result = factorial(n-1);  
    //第二步，把上一步计算所得的结果乘以 n 返回  
    return result * n;  
}
```

这样，我们的函数已经快要写好了。但是，这里还有一个小问题。例如，如果当 n 为 3 时，在 factorial(3) 函数的内部，会调用 factorial(2)；在 factorial(2) 的内部，会调用 factorial(1)；在 factorial(1) 的内部，会调用 factorial(0)；在 factorial(0) 的内部，会调用 factorial(-1)……

因此上，如果不加以控制的话，递归调用会无限制的继续下去，直到计算机种所有资源耗尽为止。

为此，我们需要为递归调用规定一个结束的条件。具体到计算阶乘的这个程序中，当参数 n 的值为 1 时，表示计算 1 的阶乘。到这一步，就没有继续递归调用下去的必要了，因此，当 n 为 1 时，应当直接返回。

完整的代码如下：

```
import java.util.Scanner;  
public class TestFactorial{  
    public static void main(String args[]){  
        Scanner sc = new Scanner(System.in);  
        System.out.println("请输入一个整数 n: ");  
        int n = sc.nextInt();  
        int a = factorial(n);  
        System.out.println(n + "!=" + a);  
  
    }  
  
    public static int factorial(int n){  
        if (n == 1) return 1;  
        int result = factorial(n-1);  
        return n * result;  
    }  
}
```

```
}
```

运行结果如下：

```
D:\Book\chp4>javac TestFactorial.java  
D:\Book\chp4>java TestFactorial  
请输入一个整数n:  
3  
3!=6  
D:\Book\chp4>
```

在这个程序运行的过程中，这个程序首先会读入 n，然后，会把 n 的值传递给 factorial 函数。由于我们输入的是整数 3，因此，会调用 factorial(3)。

在 factorial(3) 函数的内部，由于  $3==1$  的值为假，因此会执行

```
int result = factorial(3-1);
```

为了计算这个 result 的值，会先计算 factorial(2) 函数的值。因此，factorial(3) 函数调用 factorial(2) 函数。示意如下：

$\text{factorial}(3) \rightarrow \text{factorial}(2)$

然后，在 factorial(2) 函数的内部，由于  $2==1$  的值为假，因此，会执行：

```
int result = factorial(1);
```

因此，会在 factorial(2) 函数的内部，调用 factorial(1) 函数，示意如下：

$\text{factorial}(3) \rightarrow \text{factorial}(2) \rightarrow \text{factorial}(1)$

然后，调用 factorial(1) 函数。由于此时，参数的值为 1，因此 factorial(1) 函数返回 1，返回给 factorial(2) 函数。示意如下：

$\text{factorial}(3) \rightarrow \text{factorial}(2) \leftarrow \text{返回 } 1 = \text{factorial}(1)$

返回到 factorial(2) 之后，此时，要执行 `return n * result` 语句，因此 factorial(2) 会返回  $1 * 2$ ，会把计算所得的值返回给 factorial(3) 函数。示意如下：

$\text{factorial}(3) \leftarrow \text{返回 } 2 = \text{factorial}(2) \leftarrow \text{返回 } 1 = \text{factorial}(1)$

最后，factorial(3) 执行到 `return n * result`。此时，n 的值为 3，result 的值为 factorial(2) 函数的返回值 2，因此 factorial(3) 会返回 6。至此，递归调用均完成。

如果需要写递归的话，需要注意以下几个问题：

1、要首先研究出一个推导公式，这个公式要与参数有关，最好这个公式能够用  $(n-1)$  调用的结果，来计算调用 n 的结果。例如，上面的  $n! = (n-1)! * n$  就是一个很好的例子。

2、要写出什么时候终止递归调用。例如，上面对 n 是否为 1 的判断，这部分就是在说明递归什么时候终止。

递归是一种非常强大的编程思想，我们可以使用递归的思想，把一个大的问题，分解成一个或多个小问题。例如，在上面这个阶乘的例子中，我们把一个与 n 相关的函数计算问题，变成了一个与  $n-1$  有关的计算问题。

**(递归的作用)**

## 6 函数的作用

那么，为什么我们要写函数呢？为什么不能把所有的步骤都写在主函数中，而要把一些步骤写成函数的形式呢？

我们来看下面的例子。

```
public class TestSeperator{
    public static void main(String args[]){
        System.out.println("Hello World");
        System.out.println("-----");
        System.out.println("你好，世界");
        System.out.println("-----");
        System.out.println("Bonjour tout le monde");
        System.out.println("-----");
        System.out.println("Hallo Welt");

    }
}
```

上面的代码，分别用英语、中文、法语、德语四种语言，输出“Hello World”这句话。在输出这四种语言的过程中，会输出三行“-----”这个分隔符，把不同的语言分隔开。运行结果如下：



```
D:\Book\chp4>javac TestSeperator.java
D:\Book\chp4>java TestSeperator
Hello World
-----
你好，世界
-----
Bonjour tout le monde
-----
Hallo Welt
D:\Book\chp4>
```

我们来分析一下上面的代码。由于要输出三个分隔符，因此，需要有三个一摸一样的打印语句。因此，我们可以写一个函数：printSeparator()，这个函数专门用来打印分隔符。修改后的代码如下：

```
public class TestSeperator{
    public static void main(String args[]){
        System.out.println("Hello World");
        printSeparator();
        System.out.println("你好，世界");
        printSeparator();
        System.out.println("Bonjour tout le monde");
        printSeparator();
        System.out.println("Hallo Welt");
    }
}
```

```
public static void printSeparator() {
    System.out.println("-----");
}

}
```

我们可以看到，在主函数中，没有出现输出分隔符的语句。取而代之的，是对 `printSeparator` 函数的调用。这样，我们把输出分隔符的逻辑统一写到一个函数中，就能减少在主方法中，重复和类似的冗余代码，使得主方法中的代码更加清晰，从而提高代码的可读性。这就是函数的第一个作用：**减少冗余代码**。

使用函数的第二个作用是，让代码更加便于维护。例如，如果现在需求产生了变化，在输出分隔符时，希望能够输出一行“+”而不是一行“-”，此时，就必须要修改代码。

如果不使用函数的话，则必须要改动主函数中所有的输出分隔符的语句。在这个具体的例子中，在主函数中的代码要修改三处。修改后的代码片段如下：

```
System.out.println("Hello World");
System.out.println("+++++++=");
System.out.println("你好，世界");
System.out.println("+++++++=");
System.out.println("Bonjour tout le monde");
System.out.println("+++++++=");
System.out.println("Hallo Welt");
```

而相对应的，如果使用函数的话，由于在主函数中仅仅是对同一个函数的多次调用，因此，让函数实现发生改变的时候，函数的调用不需要改变。我们只需要对函数的实现修改一次，就完成了我们的工作。修改后的代码如下：

```
public static void main(String args[]) {
    System.out.println("Hello World");
    printSeparator();
    System.out.println("你好，世界");
    printSeparator();
    System.out.println("Bonjour tout le monde");
    printSeparator();
    System.out.println("Hallo Welt");
}

public static void printSeparator() {
    System.out.println("+++++++=");
}
```

这就是函数的第二个作用：**提高代码的可维护性**。

另外，使用函数，利用函数参数的变化，能够让代码更加灵活。例如，假设现在对分隔符的长度有不同的要求，要求第一个分隔符长度为 20，第二个分隔符长度为 25，第三个

分割符长度为 30。

这样的需求，如果不适用函数的话，则必须要修改这三个输出的地方，并且每个地方的输出语句都比较复杂。而如果使用函数的话，则可以让函数增加一个参数，这个参数表示分隔符的长度。修改后的 printSeparator 函数如下：

```
public static void printSeparator(int n){  
    for (int i = 1; i<=n; i++){  
        System.out.print("+");  
    }  
    System.out.println();  
}
```

这样，在主函数中，只要对这个函数传入不同的参数进行三次调用即可。修改后完整的主函数如下：

```
public static void main(String args[]){  
    System.out.println("Hello World");  
    printSeparator(20);  
    System.out.println("你好，世界");  
    printSeparator(25);  
    System.out.println("Bonjour tout le monde");  
    printSeparator(30);  
    System.out.println("Hallo Welt");  
}
```

可以看到，这样修改的工作量是比较小的。而如果不使用函数呢？相对而言，工作量就比较大，在此不再具体阐述。

上面我们所说的，是使用函数的另一个好处：**能够让程序的更加灵活**。

另外，我们写出的 printSeparator() 函数，由于写好了打印分隔符的功能，因此，如果其他的程序员也要使用打印分隔符的功能的话，可以不用自己从头再完成，完全可以调用我们之前写好的 printSeparator() 函数。因此，使用函数还能够**提高代码的可重用性**。所谓的可重用性，指的是在完成类似的功能的时候，能够利用已有的代码，从而能够对代码重复利用，减少不必要的重复劳动。

在很多编程语言中，为了能够提高代码的重用性，往往会由语言的开发者以及一些优秀的程序员一起，提供大量的函数。这些函数能够完成各种各样的通用的功能，形成一个“函数库”。而程序员在编程的时候，就是利用函数库中提供的功能，在自己的程序中调用各个函数，最终完成自己需要的逻辑。

最后，使用函数，能够让我们在分析和解决问题的时候，把大的步骤拆分成小的步骤，从而提高代码的可读性。例如，我们完成一个简单的计算器程序，基本功能如下：

首先，让用户输入两个整数，表示参与运算的两个数；

其次，输出“1：+； 2：-； 3：\*； 4：/”，提示用户，选择某一种运算；

然后，读入用户的选择；

最后，进行运算。

如果我们把所有代码都写在主函数中，则代码如下：

```

import java.util.Scanner;
public class TestCaculator{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        System.out.print("请输入一个整数: ");
        int a = sc.nextInt();
        System.out.print("请输入一个整数: ");
        int b = sc.nextInt();

        System.out.println("1: +");
        System.out.println("2: -");
        System.out.println("3: *");
        System.out.println("4: /");
        System.out.println("请输入一个整数: ");
        int choice = sc.nextInt();
        switch(choice){
            case 1: System.out.println(a + b); break;
            case 2: System.out.println(a - b); break;
            case 3: System.out.println(a * b); break;
            case 4: System.out.println(a / b); break;
            //输出-1 表示出错
            default : System.out.println(-1); break;
        }
    }
}

```

上面的代码能够完成我们的需求，但是，由于所有的代码都写在主函数中，因此整个主函数看起来比较臃肿，可读性比较差。为此，我们对原有的代码进行修改：

```

import java.util.Scanner;
public class TestCaculator{
    public static void main(String args[]){
        int a = readInt();
        int b = readInt();

        printChoice();
        int choice = readInt();

        int result = calculate(a,b,choice);
        System.out.println(result);

    }

    public static int readInt(){
        Scanner sc = new Scanner(System.in);
        System.out.print("请输入一个整数: ");

```

```

        int n = sc.nextInt();
        return n;
    }

    public static void printChoice(){
        System.out.println("1: +");
        System.out.println("2: -");
        System.out.println("3: *");
        System.out.println("4: /");
    }

    public static int calculate(int a, int b, int choice){
        switch(choice){
            case 1: return a + b;
            case 2: return a - b;
            case 3: return a * b;
            case 4: return a / b;
            default : return -1;
        }
    }
}

```

我们把一些功能写成了函数的形式。这样，主方法看起来就非常清晰：首先读入两个整数（调用了两次 `readInt`），然后输出一个菜单（调用 `printChoice`），读入选项；然后根据读入的两个数和做出的选择进行计算（`calculate`），最后输出计算结果。运行结果如下：

```

D:\Book\chp4>javac TestCalculator.java
D:\Book\chp4>java TestCalculator
请输入一个整数: 10
请输入一个整数: 20
1: +
2: -
3: *
4: /
请输入一个整数: +
200
D:\Book\chp4>

```

通过使用函数，我们把主函数的代码进行了简化，使得整个程序的逻辑非常清楚，大大提高了代码的可读性。

## 7 自顶向下，逐步求精

我们介绍了函数的基本语法，也讲解了函数的一些作用，那么应当如何设计函数呢？在编程中，哪些部分应当写成函数呢？

下面，我们介绍一种编程的思路。使用这种思路，能够设计出比较好的程序结构，并提炼出比较合理的函数调用结构。这种思想，就是“自顶向下，逐步求精”的思想。简单的说，就是在写程序的时候，先列出程序中比较大的步骤，然后，再把每一个步骤细化，最终形成代码。

我们用一个实例来看看应当如何使用这种思想。

我们来看一个程序需求：

验证哥德巴赫猜想：任何一个大于 6 的偶数，都能分解成两个质数的和。质数，指的是除了 1 和本身之外，没有别的因子的数。

要求：输入一个整数，输出这个数能被分解成哪两个质数的和。

eg : 14

14=3+11

14=7+7

这是一个比较复杂的程序。拿到这个程序的需求之后，首先应该先设计出程序的大体思路。基本思路如下：

- 1、读入一个整数 n
- 2、把这个整数拆成两个数 a、b 的和
- 3、判断 a 是否是质数
- 4、判断 b 是否是质数
- 5、如果 3、4 两个判断都为真，则输出 a 和 b
- 6、如果这个整数还能拆分，则回到第 2 步。否则程序退出

很显然，2~6 步是一个循环，调整一下结构，如下：

```
1、读取整数 n  
循环（把整数 n 拆成两个不同的整数 a 和 b） {  
    判断 a 是否是质数  
    判断 b 是否是质数  
    如果 a、b 都是质数，则输出 a 和 b  
}
```

在上面的基本思路中，我们可以看到，“判断 a 是否是质数”和“判断 b 是否是质数”这两步操作基本一样。因此，很显然，这里我们应该写出一个函数，这个函数能够判断一个整数是否是质数。先定义这个函数如下：

```
public static boolean isPrime(int n){  
}
```

则，经过这几步的分析，我们的思路已经逐渐细化了。现在代码如下：

```
import java.util.Scanner;  
public class TestGoldBach{  
    public static void main(String args[]){  
        //读入整数  
        Scanner sc = new Scanner(System.in);  
        int n = sc.nextInt();
```

```

    循环{
        int a = 第一个整数
        int b = 第二个整数
        if (isPrime(a) && isPrime(b)) {
            System.out.println(n + "=" + a + "+" + b);
        }
    }
}

//判断一个整数是否是质数
public static boolean isPrime(int a){
}

```

继续细化。现在我们关注的重点是两个：1、如何把一个整数 n 拆成两个整数 a 和 b；  
2、如何判断一个整数是质数。

首先，我们来看拆数的逻辑。如果能够确定一个整数 a，则另外一个整数 b 也就确定了，可以通过  $b = n - a$  这个式子计算出 b 的值。

那么如果给出一个整数，如何确定 a 的值呢？我们可以先看一个例子。假设 n 为 14，则所有拆数的拆法是：

```

1 + 13
2 + 12
3 + 11
4 + 10
5 + 9
6 + 8
7 + 7

```

往下就是重复的拆法了。这样，我们把第一个数当做 a，则 a 从 1 变化到 7，也就是变化到  $14/2$ 。于是，我们拆数的循环就能够分析出来了：

```

for(int i = 1; i<=n/2; i++) {
    int a = i;
    int b = n-i;
    if (isPrime(a) && isPrime(b)) {
        System.out.println(n + "=" + a + "+" + b);
    }
}

```

至此，主函数全部完成。接下来，完成 isPrime 方法。对于如何判断一个整数是否是质数，我们依然使用自定向下，逐步求精的方式。由于质数，指的是除了 1 和本身之外，没有其他的因子，因此判断一个整数 a 是否是质数，只要看  $2 \sim a-1$  的范围内有没有 a 的因子就可以了。

因此，主要思路如下：

循环 i: 2~a-1{

    如果 i 是 a 的因子，则说明 a 不是质数

```
}
```

循环结束，则说明 2~a-1 都不是 a 的因子，因此 a 是质数

至此，相关代码也就呼之欲出了。完整的代码如下：

```
import java.util.Scanner;
public class TestGoldBach{
    public static void main(String args[]){
        //读入整数
        System.out.println("请输入一个整数");
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();

        for(int i = 1; i<=n/2; i++){
            int a = i;
            int b = n-i;
            if (isPrime(a) && isPrime(b)){
                System.out.println(n + "=" + a + "+" + b);
            }
        }
    }

    //判断一个整数是否是质数
    public static boolean isPrime(int a){
        for(int i=2; i<= a-1; i++){
            if (a % i == 0) return false;
        }
        return true;
    }
}
```

运行结果如下：

```
D:\Book\chp4>javac TestGoldBach.java
D:\Book\chp4>java TestGoldBach
请输入一个整数
10
10=3+7
10=5+5
D:\Book\chp4>
```

需要说明的是，上面给出的 isPrime 函数的代码并不是最好的。读者可以自己再想想，有没有办法能够优化 isPrime 函数的代码，让它的速度更快，更有效率。

上面我们介绍的，就是“自顶向下，逐步求精”的思想。可以看到，在写代码的过程

中，这种思想先把解决复杂问题时，先罗列出比较粗略的步骤。在这个过程中，可以从中找到重复、可重用的部分，应该把这一部分提炼成函数。然后，再把每一个步骤细化，最后把代码完成。

这种思想，也可以说是一种比较典型的“面向过程”的思想。这种思想知道我们，在解决计算机问题的时候，应当先考虑解决问题的步骤和过程，然后把每个步骤、过程进行计划，最终得到的代码，就是对应着每一个步骤。面向过程的思想是编程中最基本的一种思想之一，也是程序员的基本功之一。我们应当在自己的编程和练习中，锻炼自己面向过程的思想，提高自己解决计算机基本问题的能力。

# Chp5 数组（增加冒泡排序）

数组是一个语言中的基本要素，它能够用来保存和管理多个变量。例如，如果要统计三个学生的成绩，可以手动的定义三个变量 `a`、`b`、`c`，如果要输出这三个变量的值，也可以写三个输出语句。但是，如果要管理一个年级所有学生的成绩，此时，可能有上百个学生。如果为每个学生都手动定义一个变量的话，则程序中会有上百个变量。并且，如果要输出所有学生的成绩，就会有上百个输出语句。很显然，这样的代码是非常复杂和繁琐的。

因此，我们就需要有一个办法，能够比较方便的管理多个数据。在这种情况下，我们就应该用到数组。

## 1 数组的基本操作

数组是用来管理多个同类型的变量的。一个数组，是一段连续的内存空间，这段内存空间中，能够保存多个同类型的值。例如，一个长度为 3 的 `int` 数组，是说这个数组能够保存 3 个 `int` 类型的值。

下面，我们就来介绍一下数组的一些基本操作。首先是如何创建数组。

### 1.1 创建数组

如果要定义一个 `int` 类型的数组变量 `a`，可以写成如下的形式：

`int[] a` 或者 `int a[]`

上面的两种形式都可以用来定义 `int` 类型的数组变量 `a`。需要注意的是，定义完了数组变量之后，并没有分配连续的内存空间。怎么来理解这个问题呢？

数组是用来管理多个变量的，在这里，“多个”的概念并不明确。例如，在生活中，一本书、一支笔，这些都是比较精确的“一个”物品。而“一堆书”、“一捆笔”，这种描述多个物品的情况，则具体的数量就显得很不明确。数组变量也是如此，一个数组变量能够管理多个数据，但是，究竟这个数组的空间多大，还需要另外明确指定。

为了分配空间，也为了明确指定数组空间的大小，必须要使用关键字 `new`。用法如下：

`a = new int[3];`

上面的语句才真正分配了一个长度为 3 个 `int` 的内存空间。也可以在定义数组变量的时候立刻进行内存的分配：

`int[] a = new int[3];`

### 1.2 下标， `ArrayIndexOutOfBoundsException`

为数组分配完空间之后，就可以使用数组了。使用数组的时候，应当用下标来表示数组元素。例如，上面分配了长度为 3 个 `int` 的内存空间，这 3 个 `int` 分别可以用：`a[0]`、`a[1]`、`a[2]` 来表示。

需要注意的是，数组的下标从 0 开始，也就是说，如果分配了一个长度为 `n` 的数组，则数组的下标范围为 `0~n-1`。

有了下标之后，就可以操作数组元素。此时，数组的每个元素和普通的 `int` 变量没有区别。例如：

`a[0] = 10; //对数组元素进行赋值`

```
a[1] = 20;  
a[2] = 30;  
System.out.println(a[2]); //输出数组元素
```

而如果对数组进行下标操作时，超出了数组下标正常的范围，则 Java 会产生一个异常：`ArrayIndexOutOfBoundsException`。例如下面的代码：

```
public class TestArray {  
    public static void main(String[] args) {  
        int[] a = new int[3];  
        a[0] = 10;  
        a[3] = 20;  
    }  
}
```

上面的代码在编译时不会出错，因为从语法上说没有任何问题。但是，由于 `a` 数组长度为 3，因此起下标范围是 0~2，使用 `a[3]` 变量会在运行时出错。运行时得到的错误信息如下：

```
D:\Book\chp5>javac TestArray.java  
  
D:\Book\chp5>java TestArray  
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3  
at TestArray.main(TestArray.java:5)  
  
D:\Book\chp5>
```

我们分析一下这个错误信息。首先，这个错误的类型是：`java.lang.ArrayIndexOutOfBoundsException`。这种类型的错误，产生的原因一定是数组下标越界。在第一行的末尾，还有一个数字 3，这表明数组下标越界时，越界的数字为 3。在第二行，有一个数字 5，这表明数组下标越界的代码是在源文件（`TestArray.java`）的第 5 行。

### 1.3 遍历数组

所谓遍历数组，指的是把数组的所有元素都进行某一个操作，这是数组中最常用的操作之一。最典型的遍历数组，就是把数组中的所有元素都打印一遍。例如，我们写一个打印数组 `a` 的程序：

```
public class TestArray {  
    public static void main(String[] args) {  
        int[] a = new int[3];  
        a[0] = 10;  
        a[1] = 20;  
        a[2] = 30;  
        for(int i = 0; i<3; i++){  
            System.out.print(a[i] + "\t");  
        }  
        System.out.println();  
    }  
}
```

上面的程序就能完成打印 `a` 数组的功能，利用一个 `for` 循环，就能够完成数组的遍历。需要强调的是，`for` 循环中，循环条件是 `i < 3`，这表明 `i` 变量的变化是 0~2，正好在数组的

下标范围之内。可以看到，3正好是数组的长度。

由于打印数组这个功能非常常用，我们可以再提炼一下，把打印数组的代码写到一个函数中，这个函数叫做 `printArray`，接受一个 `int` 数组作为参数，签名可以写成如下形式：

```
public static void printArray(int[] n)
```

注意，这个函数接受数组类型作为函数形参，此处的写法与定义一个数组变量相同。

下面我们完成这个函数。从上面的代码中，我们可以看到，写遍历数组的循环，往往是这样的形式：

```
for(int i = 0; i<XXX; i++)
```

其中，`i<XXX` 这个循环条件，一定是 `i` 小于数组的长度。但是，由于 `printArray` 函数可以接受任何一个 `int` 数组作为参数，因此打印数组时，数组的长度是不固定的，必须根据传入的数组参数的不同，动态的获得数组的长度。那数组的长度我们应当怎么获得呢？

在 Java 中，我们可以使用“数组名.length”的方式获得数组的长度。举例来说，如果想要获得 `n` 数组的长度，则可以使用 `n.length` 这个变量来获得长度。

例如下面的代码：

```
int[] a = new int[3];
System.out.println(a.length);
```

上面的代码，会输出 `a` 数组的长度：3。

需要注意的是，`n.length` 这种写法只能读取数组的长度，而不能修改。也就是说，不能为 `n.length` 这种变量进行赋值，否则会产生一个编译时错误。

例如，下面的代码就会产生错误：

```
int[] a = new int[3];
System.out.println(a.length);
a.length = 5; //想要改变数组的长度，出错！
```

有了 `length` 这个变量，我们就能动态的获得数组的长度，因此 `printArray` 方法就可以完成了。代码如下：

```
public static void printArray(int[] a) {
    for(int i = 0; i<a.length; i++) {
        System.out.print(a[i] + "\t");
    }
    System.out.println();
}
```

## 1.4 数组的初始化

下面我们可以调用一下 `printArray()` 方法。定义一个数组 `a`：

```
int[] a = new int[3];
printArray(a);
```

注意，在分配完三个 `int` 变量之后，并没有为这三个变量赋值。也就是说，此时，这三个变量并没有被赋初始值，而直接被 `printArray` 方法打印。

运行结果如下：

```
D:\Book\chp5>javac TestArray.java  
D:\Book\chp5>java TestArray  
0 0 0  
D:\Book\chp5>
```

编译顺利通过，打印的结果是：三个 0。因此，我们可以得出结论：数组元素和局部变量不同，数组元素可以在没有赋初始值的情况下就使用。此时，这些数组元素也有特定的值，这就是元素的“默认值”。在为数组分配空间的时候，数组的元素会被 JVM 赋一个默认值。

那么元素的默认值都是什么呢？这需要根据不同类型来看。对于基本类型来说，**默认值**为各种各样的 0。怎么来理解呢？byte、short、int、long 这四种整数类型，默认值为 0；float 和 double 这两种小数类型，默认值为 0.0，boolean 默认值为 false，char 默认值也为 0。注意，char 类型的 0 不是指的字符 ‘0’，而是指的编码为 0。

对于对象类型来说，默认值为 null 值。

有了默认值这个概念，考虑下面的代码：

```
int[] a = new int[3]; a[0] = 10; a[1] = 20; a[2] = 30;  
printArray(a);
```

在这段代码中，每一个数组元素都被赋值了两次：一次是默认值，另一次是初始值 10、20、30。那有没有什么办法能够避免那一次不必要的默认值呢？

我们可以使用一个叫做“数组的显式初始化”的语法。这种语法有两种形式。第一种形式如下：

```
int[] a = {10, 20, 30};
```

这种语法的特点是，只能在定义数组变量的同时使用。如果代码如下：

```
int[] a;  
//! a = {10, 20, 30};
```

上面的代码在数组定义之后没有直接进行显式初始化，而是换了一个语句又一次赋值。这样会产生一个编译时错误。而且，这个错误的出错信息相当吓人：

```
D:\Book\chp5>javac TestArray.java
TestArray.java:4: illegal start of expression
    a = {1,2,3};
          ^
TestArray.java:4: not a statement
    a = {1,2,3};
          ^
TestArray.java:4: ';' expected
    a = {1,2,3};
          ^
TestArray.java:5: invalid method declaration; return type required
    printArray(a);
          ^
TestArray.java:5: <identifier> expected
    printArray(a);
          ^
TestArray.java:7: class, interface, or enum expected
    public static void printArray(int[] a){
          ^
TestArray.java:8: class, interface, or enum expected
        for<int i = 0; i<a.length; i++>{
          ^
TestArray.java:8: class, interface, or enum expected
        for<int i = 0; i<a.length; i++>{
          ^
TestArray.java:10: class, interface, or enum expected
    }
    ^
TestArray.java:12: class, interface, or enum expected
    }
    ^
10 errors
```

这些错误，其实都只是因为数组显示初始化的语法不对，才产生的。显示初始化的语法一旦产生错误，就会引起一系列的连锁反应。

第二种语法形式如下：

```
int[] a = new int[]{10, 20, 30};
```

注意，这种语法下，new 关键字后面的方括号中没有数字，也就是说，显式初始化时不能规定数组长度，数组长度由后面的元素个数决定。

要注意的是，这种语法形式能够进行赋值。也就是说，下面的代码可以编译通过：

```
int[] a;
a = new int[]{10, 20, 30};
```

## 2 数组在内存中的表示

下面我们分析一下，Java 数组在内存中的表示情况。看下面两行代码

```
int[] a;
a = new int[3];
```

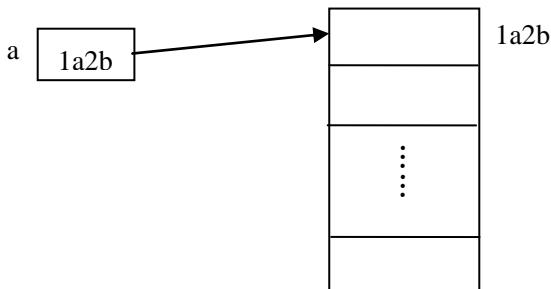
我们结合代码，分析一下数组在内存中的表示。

第一行，定义了一个数组变量 a，此时没有分配连续的内存空间。

第二行，首先执行了 new int[3]，这句代码分配了一个段连续的内存空间，总共能够放

入 3 个 int，因此是 12 个字节。这 12 个字节每个字节都有自己的一个内存地址，其中，12 个字节中的第一个字节，它的地址被称之为这块内存的“首地址”。假设首地址为 1a2b。

第三步，执行完了 new int[3]之后，进行了赋值。究竟是把什么赋值给了变量 a 呢？注意，**赋值赋的是内存的首地址**。也就是说，数组变量保存的是数组中的首地址 1a2b。如下图所示



## 3 二维数组和多维数组

### 3.1 二维数组的基本操作

上面我们介绍的都是一维数组，接下来要介绍的是 Java 中的二维数组以及多维数组。

在 Java 中，所谓的多维数组指的是：数组的数组。怎么来理解这一点呢？比如说，我们日常生活中的抽屉，我们可以认为抽屉就是用来存放多个物品的，因此抽屉就是一个物品的数组。而一个柜子中，可以存放多个抽屉，因此我们可以理解为，柜子就是抽屉组成的数组。因此，柜子就可以理解为是“数组的数组”，也就是：柜子的元素是抽屉，而抽屉本身又是一个数组。

那在代码方面怎么表示呢？我们首先来看二维数组的创建和遍历。

二维数组在创建时，也需要一个数组变量。定义数组变量，可以使用下面的代码：

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

可以看出，定义数组变量在 Java 中格式是非常自由的。需要注意的是，在定义二维数组变量的时候，同样没有分配数组空间。

如果要为二维数组分配内存空间，则可以使用下面的代码：

```
a = new int[3][4];
```

这表示，分配一个三行四列的二维数组。怎么来理解“行”和“列”呢？我们可以这么来看：我们分配的这个二维数组就相当于一个柜子，这个柜子有三层，每层放一个抽屉。这个抽屉里面分成了四个格子，每个格子又能放一个元素。由于二维数组是“数组的数组”，因此，二维数组的“行”，指的是这个二维数组中，包含几个元素。由于二维数组的元素是一维数组，因此，“行”也就是二维数组中包含几个一维数组。而列，则指的是，二维数组中的每一个一维数组，各自都包含几个元素。

如果要获得第 0 行第 2 列的元素，则可以使用双下标：a[0][2] 来获得。

下面我们来介绍遍历二维数组。遍历二维数组时，要获得行和列两个数值。首先，二维数组同样有 a.length 这样的变量，而使用 a.length 获得的长度，是二维数组元素的个数，也就是行的数目，也可以理解成：柜子里抽屉的个数。那如果要获得列呢？列，就相当于每一个一维数组的长度，也可以理解为，是每一个抽屉的大小。对于第 i 个抽屉，我

们可以使用 `a[i].length` 来获得它的长度。遍历二维数组的代码如下：

```
public static void printMultiArray(int[][] a) {
    for(int i = 0; i<a.length; i++) {
        for(int j = 0; j<a[i].length; j++) {
            System.out.print(a[i][j] + "\t");
        }
        System.out.println();
    }
    System.out.println();
}
```

完整的二维数组的代码如下：

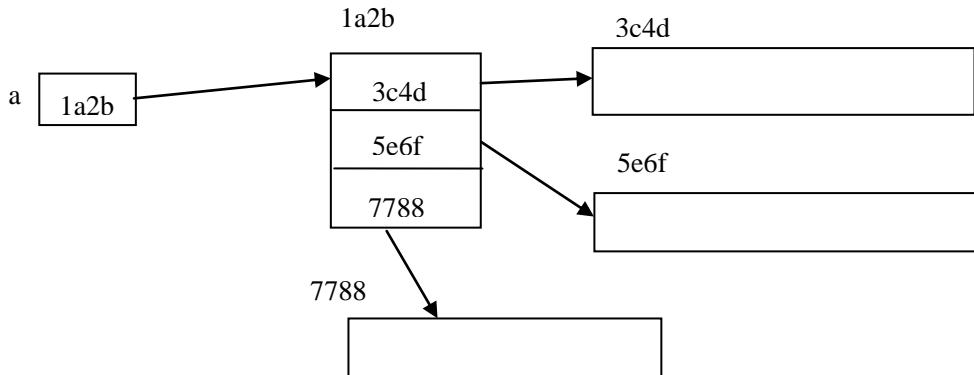
```
public class TestMultiArray{
    public static void main(String args[]){
        int[][] a = new int[3][4];
        printMultiArray(a);
    }
    public static void printMultiArray(int[][] a){
        for(int i = 0; i<a.length; i++) {
            for(int j = 0; j<a[i].length; j++) {
                System.out.print(a[i][j] + "\t");
            }
            System.out.println();
        }
        System.out.println();
    }
}
```

运行结果如下：

```
D:\Book\chp5>javac TestMultiArray.java
D:\Book\chp5>java TestMultiArray
0      0      0      0
0      0      0      0
0      0      0      0
```

## 3.2 二维数组的内存表示

二维数组在内存中的表示如下：



可以看出，在一维数组的情况下，第一个维度是一个长度为 3 的数组，这个数组保存的还是某一块内存的地址。我们可以把这个数组当做是 a 的 3 个元素，因此，非常明显，a 的这三个元素，是三个一维数组。

从这个例子我们可以推而广之到任意维度的数组。一个二维数组，就是数组的数组，一个 3 维数组就是数组的数组的数组，一个 4 维数组就是数组的数组的数组的数组，一个……

但是，对于平时的编程实践而言，使用比较多的还是一维数组和二维数组，更高维度的数组使用起来比较复杂，相对而言用的较少。

### 3.3 不规则数组

除了普通的二维数组之外，Java 还支持不规则数组。举例来说，如果一个柜子有三个抽屉，这个三个抽屉中并不一定每个抽屉都具有一样的大小，完全有可能第三个抽屉更大，元素更多，而第一个抽屉相对就比较小。在 Java 中，可以用下面的代码创建不规则数组：

```
int[][] a; //定义数组变量
a = new int[3][]; //先确定第一个维度，表明柜子里有三个抽屉
a[0] = new int[3]; //顶上的抽屉有三个元素
a[1] = new int[4]; //下一层有四个元素
a[2] = new int[5]; //最底层有五个元素
```

完整代码如下：

```
public class TestMultiArray{
    public static void main(String args[]){
        int[][] a; //定义数组变量
        a = new int[3][]; //先确定第一个维度，表明柜子里有三个抽屉
        a[0] = new int[3]; //顶上的抽屉有三个元素
        a[1] = new int[4]; //下一层有四个元素
        a[2] = new int[5]; //最底层有五个元素

        printMultiArray(a);
    }
    public static void printMultiArray(int[][] a){
        for(int i = 0; i<a.length; i++){

```

```

        for(int j = 0; j<a[i].length; j++){
            System.out.print(a[i][j] + "\t");
        }
        System.out.println();
    }

}

```

运行结果如下：

```

D:\Book\chp5>javac TestMultiArray.java

D:\Book\chp5>java TestMultiArray
0      0      0
0      0      0      0
0      0      0      0      0

D:\Book\chp5>

```

### 3.4 数组的显式初始化

与一维数组相似，二维数组也有显式初始化的语法。

在一维数组中，一对花括号，就表示一个数组。而二维数组，是数组的数组，所以，二维数组显式初始化的基本语法如下：

```

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

```

可以看到，这个二维数组 `a`，本身有三个元素。这三个元素都是数组，因此这三个元素都要加上花括号。元素和元素之间用逗号隔开，因此，三个花括号之间，用逗号隔开。

每一个花括号表示一个数组，在这三个元素中，分别表示三个数组：

```

第一个数组: {1, 2, 3}
第二个数组: {4, 5}
第三个数组: {6, 7, 8}

```

与一维数组类似，二维数组也有两种显式初始化的语法。除了我们上面介绍的初始化语法之外，二维数组还能用下面的方式显式初始化：

```

int[][] a;
a = new int[][]{
    {1,2,3},
    {4,5,6}
};

```

与一维数组一样，第一种显式初始化的语法只能用在初始化上；而第二种显式初始化的语法能够用来赋值。

## 4 数组的常见算法

接下来这一部分，我们将为大家介绍两个数组常见的算法。

### 4.1 数组的扩容

我们之前介绍过，数组有一个 `length` 变量，这个变量只能读，不能够修改。那么，如果我们需要让数组的长度增大怎么办？例如，原有数组的长度为 3，而我们希望能够得到长度更大的数组，应该怎么做呢？

首先，数组空间一旦分配完成之后，长度就不能改变了。因此，我们不能够直接在原有数组的空间后面增加新的内存空间。我们采用一个曲线救国的方式，来增加数组的长度：

- 1、分配一个新的数组，新数组的长度比原有数组要大（比如长度是原有数组的两倍）
- 2、把原有数组中的数据，拷贝到新数组中。

我们可以把数组的扩容，理解成：人想要住大房子。如果一个家庭，觉得目前的房子太小，面积不够了，应当怎么做呢？显然，不可能在原有的房子上增加新的面积，不可能在原来的墙外面再造一个新房间（这属于违章搭建）。如果想要住大房子的话，只能够重新找一间屋子，这件屋子比原来的面积要大。这就相当于分配了一块新的内存空间。

但是，分配完空间之后，一家子人不会马上就住进新房，还需要把老房子当中的东西，都搬到新房中。这样，原有的家具、电器等，都不会丢失。这就相当于在分配完空间之后，要把数组中原有的数据，拷贝到新数组中。

为此，我们写一个函数，用来完成数组扩容。这个函数接受一个 `int[]` 数组类型作为参数，把这个数组扩容一倍，然后再把扩容以后的数组返回。函数定义如下：

```
public static int[] expand(int[] a)
```

注意，如果返回一个 `int` 数组类型的话，返回值的写法就是 `int[]`。

这个方法的实现如下：

```
public static int[] expand(int[] a){  
    int[] newArray = new int[a.length * 2];  
    for(int i = 0; i < a.length; i++){  
        newArray[i] = a[i];  
    }  
    return newArray;  
}
```

完整代码如下：

```
public class ArrayExpand{  
    public static void main(String args[]){  
        int[] a = {10, 20, 30};  
        printArray(a);  
        a = expand(a);  
        printArray(a);  
    }  
}
```

```

public static int[] expand(int[] a){
    int[] newArray = new int[a.length * 2];
    for(int i = 0; i<a.length; i++){
        newArray[i] = a[i];
    }
    return newArray;
}
public static void printArray(int[] a){
    for(int i = 0; i<a.length; i++){
        System.out.print(a[i] + "\t");
    }
    System.out.println();
}
}

```

运行结果如下：

```

D:\Book\chp5>javac ArrayExpand.java
D:\Book\chp5>java ArrayExpand
10      20      30
10      20      30      0      0      0
D:\Book\chp5>

```

## 4.2 冒泡排序

排序，是数组操作中一个非常重要的部分。排序要完成的工作，就是把数组中的所有元素，进行位置上的调整，最终使得所有元素的排列都符合一定的顺序。例如，有如下数组：

```
int[] a = {5, 7, 1, 3, 2};
```

经过排序之后，希望 a 数组的值变为：

```
{1, 2, 3, 5, 7}
```

计算机科学领域中，排序的算法有很多种。在本章，我们为大家介绍最常用，也是最基本的一种排序方法：冒泡排序。

首先，冒泡排序的核心是这样的一个过程：

对数组相邻的元素两两进行比较，如果左边的元素值大于右边的元素值，则交换两个元素的位置。

我们演示一下这个过程，以上面的 a 数组为例：一开始，数组值为：

5	7	1	3	2
---	---	---	---	---

之后，把 a[0] 和 a[1] 进行比较。由于 5 比 7 小，因此，两个位置不交换。

然后，是 a[1] 和 a[2] 进行比较。由于 7 比 1 大，因此，交换两个元素的位置。得到下面的结果：

5	1	7	3	2
---	---	---	---	---

之后，进行 a[2] 和 a[3] 的比较。由于 7 比 3 大，因此，交换两个元素的位置，得到下

面的结果：

5	1	3	7	2
---	---	---	---	---

最后，进行  $a[3]$  和  $a[4]$  的比较。由于 7 比 2 大，因此，交换两个元素的位置，结果如下：

5	1	3	2	7
---	---	---	---	---

这样的一轮操作，把最大的数字 7 排到了整个数组的末尾。因此，这一轮操作就把最大的元素排好了。因此，第二轮操作的时候，就应该不用再考虑最后一个元素了。

下面我们开始第二轮操作。第二轮操作的过程不再用文字详细描述，过程如下：

5	1	3	2	7
1	5	3	2	7
1	3	5	2	7
1	3	2	5	7

可以看到，进过第二轮操作，把倒数第二个位置也排好了。第三轮，则只需要看前三个元素。第三轮执行的流程如下：

1	3	2	5	7
1	2	3	5	7

之后，第四轮还是会比较一下前两个元素。最后，排序完成。

我们可以看到，进行了四轮操作，第一轮排好了最大的元素，第二轮排好了第二大的元素，第三轮排好了第三大的元素……以此类推。在排序的过程中，相邻元素不停进行比较和交换。在交换的过程中，大的元素沉向数组的末尾，小的元素走向数组的开头；这就好像在水里面：重的东西往下沉，而轻的东西往上浮起来。正因为这种排序方式很像水里的气泡往上浮的过程，因此，这种排序方式被称为冒泡排序。

接下来，我们来写冒泡排序的代码。如果有五个元素，则需要进行 4 次循环，也就是说，如果数组的长度是  $a.length$  的话，则需要进行  $a.length-1$  次循环。因此，外层循环如下：

```
for(int i = 0; i<a.length-1; i++) {  
    ...  
}
```

内层循环稍有点复杂。我们让内层循环的循环变量为  $j$ ，则每次进行比较的时候，比较的都是  $a[j]$  和  $a[j+1]$  这两个元素。那么  $j$  的循环条件怎么写呢？

第 1 次循环， $i$  的值为 0，因为要排到最后一个，因此  $j+1$  最大值为  $a.length$ ， $j$  的最大值为  $a.length-1$ ；

第 2 次循环， $i$  的值为 1， $j+1$  的最大值为  $a.length-1$ ， $j$  的最大值为  $a.length-2$ ；

第 3 次循环， $i$  的值为 2， $j+1$  的最大值为  $a.length-2$ ， $j$  的最大值为  $a.length-3$ ；

.....

由上面，我们可知，每次循环  $j+1$  的最大值，都是  $a.length-i$ ；而  $j$  的最大值，就是  $a.length-i-1$ 。

因此，内层循环条件如下：

```
for(int i = 0; i<a.length-1; i++) {  
    for(int j = 0; j<a.length-i-1; j++) {  
        比较 a[j] 和 a[j+1],  
        如果 a[j] 比 a[j+1] 大，则交换两个元素的值  
    }  
}
```

```
}
```

```
}
```

进一步细化，代码为

```
for(int i = 0; i<a.length-1; i++) {
    for(int j = 0; j<a.length-i-1; j++) {
        if(a[j] > a[j+1]){
            则交换 a[j] 和 a[j+1] 的值
        }
    }
}
```

如何交换两个变量的值呢？假设有两个变量  $a = 5$ ;  $b=4$ ; 要交换两个  $a$  和  $b$  的值，应该怎么做呢？

如果直接执行  $a = b$  的话，则  $a$  的值 5 就会被  $b$  的值覆盖。这样， $a$  有了  $b$  的值，但是  $b$  却无法获得  $a$  变量原来的值了。因此，为了交换两个变量的值，需要第三个变量参与。

首先，定义一个新变量  $t$ ；

然后，把  $a$  的值赋值给  $t$ :  $t = a$ ;

接下来，把  $b$  的值赋值给  $a$ :  $a=b$ 。这样会覆盖  $a$  原有的值，但是  $a$  原有的值已经被保存在  $t$  变量中了。

再接下来，把在  $t$  变量中保存的原有的  $a$  变量的值，赋值给  $b$ 。

完整的代码如下：

```
int a = 5, b=4;
int t;
t = a;
a = b;
b = c;
```

因此，冒泡排序的交换部分，也可以完成了：

```
for(int i = 0; i<a.length-1; i++) {
    for(int j = 0; j<a.length-i-1; j++) {
        if(a[j] > a[j+1]){
            //交换 a[j] 和 a[j+1] 的值
            int t = a[j];
            a[j] = a[j+1];
            a[j+1] = t;
        }
    }
}
```

完整代码如下：

```
public class TestBubbleSort{
    public static void main(String args[]){
        int[] a = {7,5,1,3,2};
        System.out.println("排序前");
```

```
printArray(a);

    for(int i = 0; i<a.length-1; i++){
        for(int j=0; j<a.length-i-1; j++) {
            if (a[j] > a[j+1] ) {
                int t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
            }
        }
    }
    System.out.println("排序后");
    printArray(a);
}

public static void printArray(int[] a){
    for(int i = 0; i<a.length; i++){
        System.out.print(a[i] + "\t");
    }
    System.out.println();
}
```

运行结果如下：

```
D:\Book\chp5>javac TestBubbleSort.java
D:\Book\chp5>java TestBubbleSort
排序前
7      5      1      3      2
排序后
1      2      3      5      7
D:\Book\chp5>
```

# Ch6 对象和类

## 本章导读

本章是 Java 学习中最核心的内容之一。这一章中，我们会第一次接触“对象”和“类”这两个概念，第一次接触“面向对象”的编程思想。这部分的知识，可以说是 Java 中最核心的知识之一，Java 庞大的知识体系和结构，都是建筑在“面向对象”的思想之上的。因此，本章中既有大量的新的语法要掌握，也有很多新的概念要理解和消化。

### 1 面向对象基本概念

本小节将给读者阐述面向对象中的一些基本特点和概念。在这部分中，讲主要对面向对象的一些基本思想，做概念上的阐述和分析。

#### 1.1 编程思想

首先我们来介绍编程思想的概念。什么是编程思想呢？所谓的编程思想，简单的说，就是程序员的思考方式。程序员在编程的时候，需要按照一定的思考方式，把需求编程具体的代码，这种思考方式，就是编程思想。

例如，我们曾经介绍过“面向过程”的编程思想。利用这种思想进行编程时，程序员思考问题的方式，是“第一步干 AAA；第二步干 BBB”这种方式。这种方式，也就是考虑实现某个需求的步骤和过程。首先能够把一个需求，细化成很多步骤，然后把一些比较复杂的步骤，又细化成更小的步骤。这样，就能够把一个比较大的需求，逐步细化和求精，最终形成具体的代码。这也就是我们在介绍函数这一章时介绍的编程思想：自顶向下，逐步求精。

但是，面向过程的思想，在进行一些复杂问题的开发时，就会显得力不从心。例如，如果我们要创建一个电子商务网站，让网站的用户能够在网上开店和进行交易。如果用面向过程的思想，则首先要分析这个网站有哪些过程。在这个电子商务网站上，有用户登录的过程，有用户开店的过程，有用户买东西的过程，有用户进行搜索的过程，还有一些例如上传图片、修改价格、修改描述……等等。这么一个复杂的网站中，有非常多的步骤和过程。如果还是使用面向过程的编程方式的话，首先会有太多的过程需要操作；其次，会有很多过程相互之间有关系。例如，用户修改商品价格这个过程，就要保证用户是否是这个商品的拥有者，因此还要有一个验证的过程；而如果之前用户没有进行登录，这在验证之前还要有一个用户登录的过程，等等……

我们可以看到，如果使用面向过程的方式解决问题，当问题变复杂时，会让过程变得庞大而复杂，因此面向过程的编程思想，并不适合用来解决一些比较复杂的问题。

由于面向过程的思想无法解决一些复杂的问题，为此，人们需要寻找新的编程思想，希望用一种新的思维方式来指导程序员编程。很快的，面向对象的编程思想被提出。由于这种思想能够很好的应对复杂的需求，解决一些面向过程很难解决甚至根本无法解决的问题，因此，这种编程思想逐渐成为了计算机行业中的主流。

下面，我们就开始介绍面向对象的思想。

## 1.2 对象的基本概念

面向对象的思想，其根源是来源于现实世界。因此，我们首先介绍现实生活中，对象的概念是什么。

在现实生活中，凡是客观存在的事物都能称之为对象。例如汽车、台灯、电脑、书本、网站，这些都是现实生活中的对象。

根据现实生活经验，我们可以总结出，现实生活中所有对象都具有两个主要的要素：“对象有什么”以及“对象能干什么”。其中，**对象“有什么”**称之为对象的属性；而**对象“能干什么”**称之为对象的方法。例如，对于一个汽车对象，这个对象有颜色、品牌、价格、最高时速等属性，有启动、加速、刹车等方法。

此外，现实中的对象一定不是孤立存在的。对象和对象之间会通过某种方式产生联系。

**一种方式是方法调用的关系。**一个对象可以调用另一个对象的方法，从而在这两个对象中产生关联。例如，学生对象，可以调用老师对象的“讲课”方法，司机对象可以调用汽车对象的“行驶”方法，顾客对象可以调用厨师的“做饭”方法，等等。

**另一种方式是组合的方式：**把小对象组合成为大对象。例如，把主机、键盘、鼠标等小对象，组合成电脑这个大对象；把若干个学生对象，组合成班级对象；把若干个书本对象，组合成图书馆对象等等。这种方式也可以理解为，大对象的属性还是对象。例如，电脑的属性包括主机、显示器、键盘、鼠标等，这些属性同样也是对象。

通过这两种联系方式，我们可以把一些功能相对简单的对象组合在一起，形成复杂的系统。例如，现实生活中，企业这个对象，往往是一个非常复杂的系统，一些大规模、跨行业的企业，更是有着非常复杂的组织架构。然而，企业中每一个对象，每一个员工，要解决的问题总是相对简单的。例如，在一个软件企业中，程序员要解决的问题就是编程，会计解决的问题就是算账，销售解决的问题就是获得订单，等等。但是，虽然每个员工要解决的问题都相对简单，但是当大量功能简单的对象组合在一起之后，就能形成一个非常复杂的企业系统。

有上面的分析可知，我们生活的客观世界，就是由对象组成的世界。我们身边任何事物都是对象，而且这些对象之间，还通过各种方式产生联系，从而形成了我们这个复杂而多姿多彩的现实世界。

而我们在生活中，遇到一些问题时，也会采用面向对象而不是面向过程的方式来解决。例如，假设我们要从北京出发到上海。为了完成这个任务，首先要做的，是应当确定，用什么方式去上海，是坐飞机？坐火车？还是自己开车？等等。

这是一个选择交通工具的过程，也就是我们在选择合适的对象来解决问题的过程。在这个例子中，我们希望能够选择一个对象，这个对象能有一个“交通运输”的方法。很显然，汽车、火车、飞机，都存在这个“交通运输”方法。我们可以根据实际的情况，来选择不同的交通工具，并调用其“交通运输”方法，从而完成从北京到上海这个目标。

我们可以看到，由于现实世界是一个面向对象的世界，因此，在现实生活中我们解决问题的思路，往往就是面向对象的思路：首先找到符合要求的对象，然后调用这个对象的方法，达到我们的目的。

## 1.3 计算机中的对象

说完了现实世界，接下来我们来看看计算机世界。

计算机行业的问题，与现实世界中的问题总是相通的。这是因为，计算机就是一种人类发明的工具，这种工具的发明，就是为了给人提供帮助，帮助人更好的解决现实中的问题。

例如，在没有计算机的时候，要想买东西，就必须要去商场。为了解决很多人购物的需求，我们就利用网络和计算机，设计出了电子商务网站。这样，就通过网络部分解决了人们购物的需求。

设计电子商务网站，这是一个计算机领域的问题。但是，我们可以看到，这个问题的提出，是与现实世界紧密联系的。也就是说，计算机编程要解决的问题，往往也都是从现实生活中来的。例如，软件行业中有 word 软件，是为了解决人们处理文档的问题；有 excel 软件，是为了解决人们统计数据以及做报表的需求；有 QQ 软件，是为了解决人们交流和沟通的需求……等等。

既然计算机世界的问题都来源于现实世界，那最好的办法，就应当是让计算机来模拟现实世界。由于现实世界是一个面向对象的世界，因此，很自然的，我们就希望在计算机世界中，也引入面向对象的思想，这样，计算机就可以来更好的模拟现实世界，从而利用现实生活中的经验，更好的解决计算机的问题。

例如，如果要做一个电子商务系统，则可以采用现实生活中的经验，创建管理员对象、顾客对象、卖主对象等。每一个对象都有各自不同的属性和方法，例如，管理员对象就好像是现实生活中的工商管理人员，他们具有批准开店、查封等方法；顾客对象具有浏览商品、下订单和付款的方法；而卖家对象，则有上货、修改价格等方法。这些方法的设计和实现，完全可以参考现实生活中的交易流程来实现。

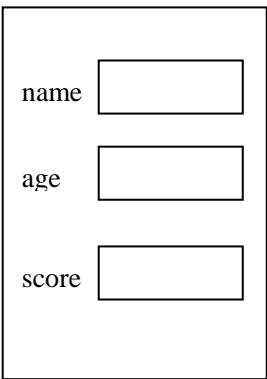
为了让计算机中也有面向对象的思想，首先要解决的问题，就是计算机中如何表示现实生活中的一个对象。

由于现实生活中的对象都具有属性和方法，因此，在计算机中，也应当让对象具有属性和方法。

然而，现实生活中对象的属性和方法非常丰富，在我们对现实世界的对象建模，创建对应的计算机中的对象时，需要对现实生活中对象的属性和方法有所取舍。例如，现实生活中，学生对象的属性非常丰富，学生有姓名、性别、身高、体重、视力、恋爱次数……等等属性，有学习、娱乐、唱歌、跳舞、吃饭、睡觉……等等方法。但是如果要设计一个学生信息管理系统，则需要对学生的属性和方法进行取舍，为学生对象保留姓名、性别、年龄等属性，保留学习、考试、练习等方法。这种取舍的过程称之为“抽象”。通过“抽象”，我们可以把现实生活中的对象用计算机模拟，可以把现实生活中活生生的对象用计算机中的一些数据表示。

通过抽象，我们就把现实中的对象，变成了计算机中的对象。那么，本质上，在计算机中的对象，是怎么来表示现实中的对象呢？

例如，我们对现实生活中的学生对象进行抽象，形成了计算机中的学生对象。抽象的时候，保留了学生对象的这样几个属性：姓名、年龄和学生成绩。因此，对于一个计算机中的学生对象而言，实际上就是在内存中的一块数据区域。这块区域中，有一个字符串，用来表示姓名，有一个整数用来表示年龄，有一个 double 类型的数，用来表示学生的成绩。在内存中的表示如下：



在内存中，我们分配出了一块数据区域，在这块区域中，包含了抽象出的学生对象的属性。这样，我们在计算机中，就模拟出了一个学生对象。因此，从本质上说，计算机中的对象，就是在内存中的一块数据区域。

需要注意的是，计算机中的对象，能够把多个相关的数据，放在同一个对象内进行操作。例如上面的例子，有了面向对象的思想之后，我们操作的是学生对象，而不是操作字符串、整数和 double 类型。我们通过对象的概念，把这些互相之间有关联的数据，放到一个区域中进行管理。

通过把现实生活中的对象抽象成计算机中的对象，我们就可以在计算机世界中模拟出跟现实世界中一样的面向对象的世界。这样，我们在进行编程的时候，就可以利用这种思想来解决问题。

例如，我们就可以使用面向对象的方法，来模拟从北京到上海的过程。在现实生活中，我们首先会寻找一个具有“交通运输”方法的对象。如果我们选择了“飞机”对象，那么就需要调用“机票代理人”对象的“卖机票”方法，来获得一个“机票”对象。再调用“出租车”对象的“开车”方法，到达机场。利用手中的机票，调用“飞机”对象的“登机”方法，“飞行”方法。从而到达目的地。示例代码如下：

```
//Ticket 表示机票
class Ticket{
}

//机票代理人对象
class Agent{
    //sellTicket 方法，表示机票代理人卖机票
    //返回值是一张机票
    public Ticket sellTicket(double money) {
        ...
    }
}

//出租车对象
class Taxi{
    //drive 方法，表示出租车开往某地
    public void drive(String location) {
        ...
    }
}

//飞机对象
```

```

class Plane{
    //登机方法，接受一个机票对象作为参数，返回值表示是否登机成功
    public boolean checkIn(Ticket t){
        ...
    }
    //表示飞往某地
    public void fly(String location){
        ...
    }
}
public class TestTravel{
    public static void main(String args[]){
        Agent a = new Agent(); //创建机票代理人对象
        Plane p = new Plane(); //创建飞机对象
        Taxi taxi = new Taxi(); //创建出租车对象

        //调用代理人的 sellTicket 方法，获得机票对象
        Ticket t = a.sellTicket(1000);
        taxi.drive("Airport"); //调用出租车的开车方法，开往机场
        if (p.checkIn()) { //调用飞机的登机方法，判断登机是否成功
            p.fly("Shang Hai"); //登机成功则飞往上海
        }
    }
}

```

可以看到，上面的代码和我们在现实生活中解决问题的思路是一样的：先获得相应的对象，再调用对象的方法。这与我们现实生活中先选择交通工具，再利用选定的交通工具出行，解决问题的思路是非常吻合的。

有了面向对象的思想之后，与面向过程不同，我们编程的方式发生了非常大的改变。当我们遇到一个复杂需求的时候，我们不应该把这个需求分解成一个一个的步骤，而是应该先设计在系统中有哪些对象。在编程的时候，我们应当先准备好系统中需要使用的对象，然后，通过组合对象以及让对象之间产生方法调用，从而把对象组合在一起。这样，我们通过对多个对象的操作，让对象之间产生联系，从而完成复杂的系统和功能，最终完成需求。

## 1.4 面向对象的特点

面向对象的思想之所以能够流行，原因在于，相对于面向过程，他有以下一些特点：

### 一、各司其职

这个特点可以说是面向对象最大的优点。面向过程之所以无法应对大型系统，就在于当问题比较复杂的时候，过程会变得繁琐而容易出错。例如，在进行电子商务的设计中，一个简单的修改价格的功能，在面向过程的实现中，需要考虑的因素非常复杂。可能在设计这个函数的时候，这个程序员既要考虑用户是否有资格进行修改，也要考虑修改的数值是否在合理的区间范围之内，还要考虑如果修改失败应当怎么处理错误……等

等一系列的问题。

而如果使用面向对象的话，我们在设计系统的时候，就可以专门设计一个对象，用来完成资格验证的功能；一个对象专门用来进行数据方面的验证，另一个对象负责处理错误……这样，我们就通过多个对象的协作，共同完成了一个功能。这样，每一个对象所要处理的问题都相对简单，从而把问题的复杂性降低了。

从现实生活中来说，各司其职也更符合现实生活中的情况。例如，就像我们之前提到的例子，现实生活中，公司就是一个复杂的系统。而在一个大公司中，完成一个过程，可能需要很多的手续和步骤。例如，如果公司要进行采购的话，首先要让老板进行审核和预算的批准，然后是财务入账，让出纳拿钱；再然后采购到的物品要进行公司资产的登记……这个步骤是非常繁琐而复杂的。但是，在这个过程中，涉及到了多个对象，而每个对象完成的功能又是相对简单的。例如，老板就负责签字，财务就负责走账，出纳就负责拿钱，等等。这样，每个对象的工作和功能都相对简单，但是通过配合，能够完成非常复杂的功能。

在我们的代码中，面向对象的思想把复杂的系统分解成相对简单的对象，使得每个对象处理的事情相对简单和集中。这样，让不同的对象能够“各司其职”，只完成一些相对简单的功能，然后通过各个对象之间的配合，完成整个系统的功能。

可以说，“各司其职”的思想，是整个面向对象思想的核心。

## 二、可重用性

所谓的可重用性，指的是对于类似的功能，不同的系统可以重复使用相同的代码。这样，有些通用的功能，程序员写了一遍之后，在遇到类似的功能之后，不需要从头开始开发，只需要对这些通用的功能进行使用就可以了。

可重用性并不是面向对象特有的概念。在面向过程的编码中，同样有重用的思想。面向过程的重用是函数的重用，我们可以把通用的过程写成函数，然后在程序中多次调用这个函数，这样就能够实现重用。但是，这种重用比较困难。一方面，对于不同的系统来说，虽然都有类似的功能，但是很难完全不变的直接使用原有的过程。例如，虽然在不同的公司中，都有采购、审批这样的过程，但是，每个公司都有自己过程的特殊性，每个公司的审批流程都是不一样的。因此，在现实生活中，很少有相同的过程，对于编程来说，也很难写出一个函数，能够满足所有的要求。

相对而言，面向对象重用起来很容易。例如，虽然每个公司采购的审批流程不同，但是都会有一个出纳，完成最后的工作。因此，一个出纳，既可以在 A 公司工作，适应 A 公司的流程，也可以在 B 公司工作，适应 B 公司的流程。这说明，按照面向对象的方式，可重用性是比较高的。对于类似的过程而言，只要把对象的组合方式和联系方式进行一些调整即可，对象本身的功能是不需要改变的。

因此，面向对象相对于面向过程而言，有着更好的可重用性。

## 三、可扩展性

可扩展性指的是，在不修改原有系统的前提下，能够进行扩展。在面向过程的编程方式里，如果要增加系统的功能，则大部分的过程就都必须要改变。例如，在我们的电子商务系统中，我们想要增加一个 VIP 会员的功能。这种会员，在开店和购物上面，会有更多的优惠以及便利的功能。如果我们按照面向过程的编程思路，增加了这种功能之后，就必须修改原有的开店和购物函数，在这两个函数中增加判断，如果是 VIP 会员的话，就执行一些新的代码。这样，增加了新的功能之后，势必会影响到原有功能的代码。造成的结果是，一旦系统的功能持续的增加的话，需要进行的修改和维护就变

得越来越复杂，最终导致的是增加一个新功能过于复杂，让系统无法维护，最终让系统无法增加新的功能，从而失去了可扩展性。

而如果是使用面向对象的编程方式呢？如果我们需要增加一个 VIP 会员功能，则只需要创建一个 VIP 会员对象，这个对象中包含了一个 VIP 会员应有的功能。这样，原有的普通会员的代码与 VIP 会员的代码互相独立，增加的 VIP 会员的代码不会影响到原有的功能。这样，新增加功能不会影响系统中原有的功能，这样，让系统增加功能就变得十分简单，从而让系统能够不断进步和扩展。这就是面向对象的可扩展性。

#### 四、弱耦合性

弱耦合性指的是，让对象和周围环境之间的联系尽可能的弱。那这样有什么优点呢？如果一个对象跟周围的联系比较弱，那么这个对象就可以很容易的被替换。

例如，台式机的显示器和整个系统之间，是通过与主机上的一个接口相连。而笔记本电脑的显示器和整个计算机系统之间，是通过内嵌在模具中联系的。这两种联系相比，台式机显示器的耦合较弱，而笔记本显示器的耦合较强。

弱耦合性能够提高可重用性和可扩展性。例如，如果想要换一个更大的显示器，扩展系统功能，对于台式机而言，相对简单；而对于笔记本电脑来说，这个需求几乎不可能完成。这就是由于台式机的显示器对象与系统之间是弱耦合的关系，而笔记本显示器与系统之间的耦合程度太强，因此不容易替换。

我们在写代码的时候，也会遇到这样的情况。我们希望我们的代码能够具有更好的弱耦合性，这样，如果有一个更好的模块的话，可以很方便的替换现有的模块，不需要对代码进行大量的修改。这就是弱耦合性为我们带来的好处。

### 1.5 类的概念

接下来要介绍的是“类”的概念。

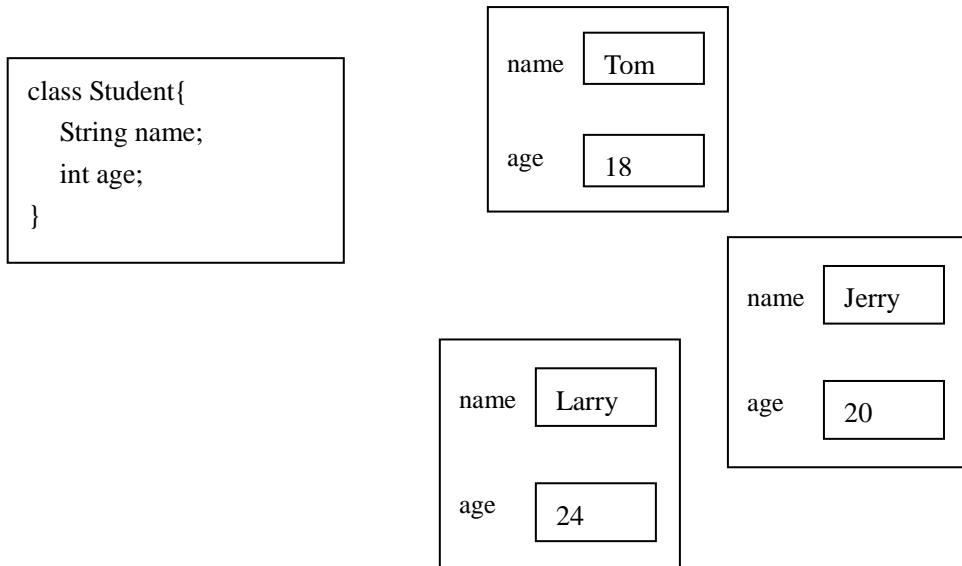
“对象”指的是客观存在的事物，因此每一个对象都意味着客观世界的一个事物。而“类”不同，它是对一类事物的总结，可以把类理解为：对大量对象共性的抽象。

例如，现实生活中，存在许许多多的狗“对象”（小白、大黄、旺财、来福……）。对于一个不认识“狗”的婴儿来说，当他见到越来越多的狗对象之后，慢慢的就能逐渐总结出，这一类对象的特点：四条腿，吃肉，摇尾巴……于是，慢慢的，他就认识了狗。也就是说，在他的脑子中，形成了“狗”这个概念，狗这个“类”就包含在他的脑子中了。

也就是说，当人面对大量的对象之后，慢慢的会把对象的共性进行抽象，从而形成了“类”。换句话说，类是对客观事物的总结和抽象。

与此同时，我们注意到，“类”是在人的脑子中形成的一个概念，也就是说，类是在人脑子里的，而不是一种客观事物。我们可以认为，类就是客观事物在人脑中的反映。

而在计算机世界中，类除了现实生活中的含义之外，还有另外一层含义：类能够作为创建对象的模板。也就是说，在计算机中，如果定义了一个类，则可以利用这个类创建多个对象。



## 1.6 小结

对象是客观事物。对象有两个主要元素：属性和方法，其中属性表示对象“有什么”，方法表示对象“能干什么”。

计算机中的对象本质上是一块数据，是对现实生活中对象的抽象。在编程时使用面向对象的思想，能够更好的模拟现实世界，更适合解决复杂的问题。

面向对象具有各司其职、可重用、可扩展、弱耦合等特性。

类是客观事物在人脑中的反映，是对对象共性的抽象，在计算机中，类是对象的模板。

## 2 类的定义

在上一节中我们主要讲述了面向对象的大量概念。在这一节中，我们将给大家讲述面向对象的一些基本语法。

### 2.1 编写一个类

我们讲过，类是对象的模板。如果我们希望创建一些对象，就必须把类写好。

定义类的语法很简单，相信我们也比较熟悉。

```

class 类名{
    类的内容
}

```

事实上，我们以前就写过很多类。简单回忆一下类的语法：定义类的关键字是“class”，后面跟一个代码块，代码块中定义了类的内容。

一个类在编译之后会生成一个名为“类名.class”的字节码文件。在一个 Java 源文件中，可以定义多个类，但是只能定义一个公开类。公开类的类名必须和文件名相同。

例如，我们来编写一个“学生”类，用来记录一个学生对象的属性和方法：

```

class Student{
}

```

## 2.2 定义类的属性

我们可以在类中定义类的属性。例如，学生类包含两个属性，姓名和年龄。则代码如下：

```
class Student{  
    int age;  
    String name;  
}
```

我们可以看到，定义属性实际上就是定义变量。在 Student 类中，我们定义了两个变量“age”和“name”。这两个变量和我们以前讲过的“局部变量”的概念是不同的。局部变量指的是在方法内部定义的变量，而这两个变量却是**定义在任何方法之外的，这种变量我们称之为：实例变量**。下面我们通过具体的代码，来介绍实例变量的用法：

考虑下面的代码：

```
public class TestStudent{  
    public static void main(String args[]){  
        Student stu = new Student(); // 创建 Student 类的对象 stu  
        System.out.println(stu.age); // 打印 stu 对象的 age 属性  
    }  
}
```

stu 对象的 age 属性在使用前没有被赋值，但是上述代码能够编译通过，输出结果为 0。由此可见，实例变量和局部变量不同，局部变量必须先赋值后使用，而对于**实例变量，系统会为其分配一个默认值。**

**实例变量的默认值规则与数组元素默认值一样，对于对象类型的属性，默认值为 null 值，对于数值类型的属性，默认值为 0，而对于 boolean 类型属性，默认值为 false。**

我们再来看下面的代码：

```
01: class MyClass{  
02:     public void print(){  
03:         System.out.println(value);  
04:     }  
05:     int value = 10;  
06: }
```

程序运行会打印出“10”。

尽管实例变量 value 的定义是在程序的第 5 行，但是我们依然可以在第 3 行来访问这个属性。因为 value 属性是定义在 MyClass 类中的，因此在 MyClass 类中任何位置，都可以访问这个属性。由此我们可以得出结论：**实例变量的访问范围是在整个类的内部。**

回忆一下局部变量，局部变量的访问范围是从定义的位置开始，到包含它的代码块结束。显然，实例变量的访问范围要大于局部变量。

我们再来考察一下局部变量和实例变量命名冲突的问题：

```
class MyClass{  
    public void print(){  
        int value = 20;  
        System.out.println(value);  
    }  
    int value = 10;
```

```
}
```

上述代码能够编译通过。在 `print` 方法内部，我们定义了局部变量 `value`，同时，由于 `print` 方法在实例变量 `value` 作用范围之内，因此在 `print` 方法内部，局部变量和实例变量存在命名冲突。很显然，程序编译通过，说明这种命名冲突是允许的。

如果我们来调用 `print` 方法，会发现，程序输出的结果是“20”。也就是说，**当实例变量和局部变量发生命名冲突时，以局部变量优先。**

## 2.3 定义类的方法

从概念上说，一个类的方法指的是这个类能干什么。从语法上说，定义一个方法需要定义“方法声明”和“方法实现”。事实上，方法的概念和前面我们介绍的“函数”概念是非常类似的。

### 2.3.1 方法声明和方法实现

**方法声明分为五个部分：**

**修饰符 返回值类型 方法名 参数表 抛出的异常**

细心的读者可以发现，“返回值类型 方法名 参数表”实际上就是我们以前讲过的“函数三要素”。

方法的修饰符比较灵活，有的方法可能没有修饰符，有的方法可能有很多修饰符，如果一个方法具有多个修饰符，那么修饰符的次序是无关紧要的。例如：我们熟知的主方法的声明如下：

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

这其中，“`public`”和“`static`”都是方法的修饰符，它们的次序并不重要，因此，下面的写法也是正确的：

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

抛出的异常部分设计到“异常处理”的知识点，本书将在后续章节中对此做专门介绍。

在方法声明之后，紧跟一个代码块，这个代码块称之为方法的实现。

方法声明描述了调用方法时的一些信息，例如调用方法应该传递哪些参数，调用方法有哪些限制，调用方法会返回什么类型的返回值，等等。我们可以这样理解：方法的声明代表着“对象能做什么？”。

而方法的实现，则定义了对象对于一个方法的实现细节，这往往代表了“对象怎么做？”例如，`Student` 类的 `study` 方法声明，定义了 `Student` 对象能够“学习”，而 `study` 方法的实现则表明了 `Student` 对象是如何“学习”的。

我们为 `Student` 类添加一个方法，代码如下：

```
class Student{  
    int age;  
    String name;  
    public void study(){  
        System.out.println("Student study for 8 hours");  
    }  
}
```

### 2.3.2 方法的重载

在一个类中，我们可以定义一系列方法，这些方法的方法名相同，参数表不同，这种语法被称为“方法的重载”。例如，我们可以为 Student 类定义两个 study 方法：

```
class Student{  
    public void study(){  
        System.out.println("study()");  
    }  
    public void study(int n){  
        System.out.println("study(int)");  
    }  
}
```

上面这段代码中，在 Student 类中定义了两个 study 方法，一个无参，另一个带一个字符串参数。程序运行时，根据不同的参数，会调用不同的方法。例如：

```
public class TestStudent{  
    public static void main(String args[]){  
        Student stu = new Student();  
        stu.study(); //调用无参的 study 方法，打印 study()  
        stu.study(10); //调用 int 参数的 study 方法，打印 study(int)  
    }  
}
```

需要注意的是，当程序被编译时，如果出现方法的重载，Java 编译器会根据实参的类型，来匹配一个合适的方法调用。因此，方法的重载又被称为“编译时多态”。请记住，这里格外强调“编译时”的概念，因为，哪一个重载的方法会被调用，这个问题在程序的编译时就已经决定了。

我们强调了，方法的重载的关键在于“方法名相同，参数表不同”。那究竟怎样的参数表算是“不同”呢？以下几种情况都可以认为是参数表不同：

- 参数个数不同。例如 void study() 和 void study(int n)
- 参数类型不同。例如 void study(int n) 和 void study(double d)
- 参数类型的排列不同。例如 void study(int n, double d) 和 void study(double d, int n)

特别要提示的是，如果两个方法仅仅是形参名不同，这不算重载！即下面的两个方法不构成重载：

void study(int a) 与 void study(int b)

此外，方法的重载对方法声明的其他部分没有要求。也就是说，对返回值类型、修饰符、抛出的异常这三个部分，方法重载并不要求它们相同或是不相同。

为什么要设计重载的语法呢？这是为了让方法的调用者，不用关心由于类型不同所造成的差异。举例而言，System.out 是一个对象，在这个对象中，有多个重载的 println 方法。例如，我们之前写过这样的代码：

```
int i = 10;  
System.out.println(i);  
System.out.println("Hello World");  
System.out.println();
```

在上面的代码中，我们调用了三次 println 方法。注意，这三次调用 println 方法的时候，

我们为这三个方法传递了不同的参数：第一次参数为 int 类型，第二次参数为字符串类型，第三次参数为空。

想这样，对于程序员而言，要打印一某个东西，不需要考虑参数不同的类型，而只需要交给 System.out.println 方法打印即可。因此，不论是打印整数和打印字符串，我们调用 System.out.println 方法的方式是一样的，这就是重载为我们带来的好处。

在生活中，也有重载的例子。例如，人是一个对象，而人具有“吃”这个方法。但是，人具有多个不同的吃方法。例如，吃药时，含药品拿水往下灌；吃西餐时，用刀叉；吃中餐时，用筷子；吃肯德基麦当劳时，用手……等等。对于人来说，吃不同的东西，具有不同的方法。

但是，我们在日常生活中，并不需要对不同的吃方法加以区分。例如，请朋友吃饭时，只要把东西端上来，自然客人会调用自己合适的吃()方法。这样，请客的主人，就好像是方法的调用者。主人的主要工作，就是为客人的吃()方法，准备合适的参数。例如，主人端上一盘牛排，只要对客人说：“请吃”，调用客人的吃方法就可以了。而客人呢，根据主人拿上来的牛排，就用刀叉来吃；这就相当于根据不同的参数，来选择相应的方法。

因此，我们可以看到，重载的作用在于：对方法的调用者，屏蔽由于不同参数带来的方法实现上的差异。例如，在 System.out.println 这种情况下，程序员就是方法的调用者，实现上的差异不需要程序员关心。

## 2.4 构造方法

构造方法（也被翻译成构造器）是面向对象特有的概念，是一种比较特殊的方法。这个方法与创建对象有关，也是面向对象当中非常重要的一个方法。

我们先看一个代码的例子，为 Student 类添加一个构造方法：

```
class Student{  
    public Student(){  
        System.out.println("Student()");  
    }  
}
```

### 2.4.1 构造方法的基本概念

构造方法是一个比较特殊的方法。相比一般方法，构造方法有如下特点：

- 构造方法的没有返回值类型。

参考上面的 Student 构造方法。要注意，没有返回值类型是说，省略返回值类型这个部分。要注意没有返回值类型和返回 void 之间的区别。类似 void Student()这种方法不是构造方法。

- 构造方法的方法名与类名相同。

例如，给定 Student 类，则其构造方法的名字必须是 Student。

要注意的是，构造方法对方法名有要求，对参数表没有要求。因此，一个类中可以利用方法重载，添加多个构造方法。例如：

```
class Student{  
    public Student(){  
    }
```

```

        System.out.println("Student()");
    }
    public Student(int n){
        System.out.println("Student(int)");
    }
}

```

上面的代码为 Student 定义了两个构造方法，这两个构造方法方法名相同参数表不同，从而构成重载。

- 构造方法不能手动调用，只能在对象创建时自动调用一次。

构造方法由 JVM 来自动调用。在使用 new 关键字创建对象时，会自动调用对象的构造方法。例如，有如下代码：

```

public class TestStudent{
    public static void main(String args[]){
        //创建一个 Student 对象，并在创建时自动调用无参构造方法
        Student stu1 = new Student();
        //创建一个 Student 对象，并在创建时调用有参构造方法
        Student stu2 = new Student(10);
    }
}

```

运行结果如下：

```

D:\Book\chp6>javac TestStudent.java
D:\Book\chp6>java TestStudent
Student()
Student<int>
D:\Book\chp6>

```

可以看出，上面的代码创建了两个对象，在创建这两个对象的时候会调用这两个对象的构造方法。要注意的是，要想给构造方法传递参数，参数应该写在“new 类名()”后的圆括号中。这样，编译器就会根据圆括号中的参数来决定，究竟应该调用哪一个构造方法。

#### 2.4.2 默认构造方法

考虑下面的代码：

```

class MyClass{
    int a;
    int b = 100;
}

public class TestMyClass{
    public static void main(String args[]){
        MyClass mc = new MyClass(); //编译正常
    }
}

```

```
    }  
}
```

在这段代码中，程序员没有写任何的构造方法。但是，在主方法中，由于创建对象时，使用了 new MyClass(); 在圆括号中没有加入任何的参数，因此，这就意味着在创建对象时，需要调用 MyClass 类的无参构造方法。然后，虽然我们没有在 MyClass 中定义无参构造方法，但是上述代码也能够编译通过。这是为什么呢？

这是 Java 编译器在编译程序时，为我们做的事情。**在 java 中，如果类中没有定义任何的构造方法，则编译器会自动生成一个公开的、无参的空构造方法。**例如，我们的 MyClass 类没有定义任何构造方法，于是，在使用 javac 进行编译的时候，此时，编译器会自动生成一个构造方法： public MyClass(){}。这个构造方法修饰符为 public，没有参数，方法的实现为空。这是编译器为我们生成的默认构造方法。

有了默认构造方法之后，在创建 MyClass() 对象时，就能够调用无参的构造方法，因此，下面的代码就能够编译通过。

```
class MyClass{  
    int a;  
    int b = 100;  
    //MyClass 中没有定义任何构造方法  
    //因此，编译器会自动加上一个构造方法：  
    //public MyClass() {}  
  
}  
  
public class TestMyClass{  
    public static void main(String args[]){  
        MyClass mc = new MyClass(); //MyClass 中存在无参构造方法  
    }  
}
```

需要注意的是，如果为 MyClass 类增加了任何一个构造方法，则编译器就不会自动生成默认构造方法。例如，把 MyClass 改成：

```
class MyClass{  
    int a;  
    int b = 100;  
    //增加一个有参构造方法，则编译器就不生成默认构造方法  
    public MyClass(int n){  
    }
```

则在创建 MyClass 对象时，不能调用无参构造方法。例如：

```
public class TestMyClass{  
    public static void main(String args[]){  
        MyClass mc = new MyClass(); //! 编译出错，找不到无参构造方法  
    }  
}
```

编译上述代码，出错信息如下：

```
D:\Book\chp6>javac TestMyClass.java
Test MyClass.java:10: cannot find symbol
symbol  : constructor MyClass()
location: class MyClass
        MyClass mc = new MyClass();
                           ^
1 error

D:\Book\chp6>
```

在这里，我们建议各位读者，作为一个良好的编程习惯，**我们应当尽量为每一个类，都提供一个无参的构造方法。**这样，无论编译器是否自动生成，我们都能够保证每一个类都有一个无参构造方法。这种保证能够给我们未来的 Java 编程带来很大的好处。

### 2.4.3 对象创建的过程

我们说，构造方法在对象创建的时候调用。那么，对象究竟是怎么创建的呢？分为几个步骤？在哪一个步骤调用构造方法呢？这一小节，我们就来为大家介绍对象创建的过程。

创建一个对象，在不涉及继承（这是下一章的内容）的情况下，基本步骤为 3 步：

#### 1、分配空间

我们曾经介绍过，计算机中的对象，本质上是内存中的一块数据。因此，在计算机中创建一个对象，就意味着必须在内存中为一个对象分配一块数据区域。

此外，在分配空间的同时，会把对象所有的属性设为默认值。这也是为什么实例变量

#### 2、初始化属性

如果直接对属性进行了赋值操作，则在这一步完成。

#### 3、调用构造方法

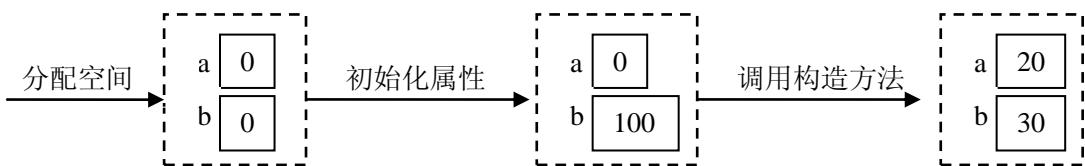
当分配空间和初始化属性完成之后，最后一步是对该对象调用构造方法。

例如：有如下代码

```
class MyClass{
    int a;
    int b = 100;
    public MyClass() {
        a = 20;
        b = 30;
    }
}
public class TestMyClass{
    public static void main(String args[]) {
        MyClass mc = new MyClass();
        System.out.println(mc.a);
        System.out.println(mc.b);
    }
}
```

```
    }  
}
```

分析上述 MyClass 对象创建的过程。首先，为对象分配空间。分配的空间中，有一块区域用来保存实例变量 a，有一块区域用来保存实例变量 b。这两块区域在分配完之后，获得默认值 0。如下图所示：



分配空间之后，属性被赋值为默认值。之后，执行初始化属性的步骤。由于 a 属性没有被直接赋值，因此初始化属性时仅需要为 b 赋值即可。之后，调用 MyClass 的无参构造方法。由于在构造方法内部为 a 属性和 b 属性赋值为 20 和 30，因此最后在程序中输出时，输出的是 20 和 30。完整的过程如上图所示。

在创建这个对象的过程中，a 属性被赋值了两次：一次是分配空间时的默认值，另一次是构造方法中的赋值。而 b 属性被赋值了三次：分配空间时获得了默认值 0；初始化属性时获得了初始值 100，调用构造方法时被赋值 30。

我们可以看到，构造方法，是创建对象的最后一个步骤。举例来说，如果说人是一个对象的话，那人的构造过程是什么呢？首先是十月怀胎，胎儿在母体内逐渐长大，并且各种器官逐渐形成，这就是在为新对象分配空间和初始化属性。然后，在分娩之后，医院的护士，会把胎儿和母亲之间的脐带剪断。这个“剪脐带”的动作，就可以认为是人的构造方法。首先，剪脐带这件事情，是生孩子的最后一个步骤。当剪断这根脐带之后，就意味着胎儿和母体相分离，从此，这个胎儿就成了一个独立的对象。其次，剪脐带这个动作，在人的一生中只会进行一次，当一个人出生之后，就再也不用进行第二次剪脐带的动作。第三，这个动作，是在孩子出生时，由父母、护士来完成的，当孩子长大之后，不会自己去调用自己的剪脐带方法。绝对不会有一个成年人，觉得自己肚脐眼的形状不好看，到医院要求重新剪一次的。

综上所述，对象的创建过程分为三步：分配空间、初始化属性、调用构造方法。其中，调用构造方法是对象创建中最后一个步骤。当构造方法完成之后，意味着对象的创建彻底完成了。

由于构造方法总是在最后一步调用，因此构造方法有一个常见的用法：用来为对象的属性赋值。这样，创建对象的时候就能够直接传递一些属性值，从而减少冗余的代码。典型的例子如下：

```
class Student{  
    String name;  
    int age;  
    public Student() { }  
  
    public Student(String stuName, int stuAge) {
```

```
    name = stuName;
    age = stuAge;
}
}
```

我们定义的 `Student` 类中，有一个无参的构造方法。这个构造方法什么都不做，因此创建对象时，如果调用无参构造方法，则对象的属性都是默认值。同时，`Student` 类还有一个带参数的构造方法，在这个构造方法中，为对象的属性进行了赋值。

## 3 对象的创建和使用

### 3.1 对象的创建

在介绍构造方法时，我们介绍了对象创建的过程。在这一小节中，我们来讲一下创建对象的语法。在介绍之前，我们首先来定义一个 `Student` 类：

```
class Student{
    String name;
    int age;
    public Student() { }
    public Student(String stuName, int stuAge) {
        name = stuName;
        age = stuAge;
    }
    public void study(){
        System.out.println("Student study for 8 hours");
    }
    public void study(int n){
        System.out.println("Student study for" + n + "hours");
    }
}
```

下面我们来介绍如何创建对象。例如下面的两行代码：

```
Student stu;
stu = new Student();
```

这两行代码中，第一行代码创建了一个 `stu` 变量。但是要注意的是，创建这个变量的时候，并没有真正创建一个对象。也就是说，在内存中没有为某一个对象分配一块空间。那么 `stu` 变量用来干什么呢？这个变量叫做引用，可以用来操作对象。具体内容在下一小节会有详细的介绍。

第二行代码，才真正创建了一个对象。在创建对象时，必须要使用到一个关键字：`new`。有了 `new` 关键字之后，后面写上要创建的对象的类型。例如，我们这里要创建一个 `Student` 类型的对象，因此，在 `new` 关键字后面写上 `Student`。

然后，在类名后面，跟上一对圆括号。圆括号中可以写上参数，用来表示调用构造方法时，为方法传递的参数。例如，上面的例子中，在圆括号中没有给出任何参数，所以创建对象时会调用无参的构造方法。

我们也可以在圆括号中给出参数。例如下面的代码：

```
Student stu2 = new Student("Tom", 18);
```

这样，我们定义了一个 stu2 变量的同时，还创建了一个新的学生对象。这个学生对象的 name 属性和 age 属性，我们在构造方法中就给出，让这个学生的姓名为“Tom”，年龄为 18 岁。

## 3.2 引用

### 3.2.1 引用的概念

引用是 java 中很重要的概念。这个概念是后面学习面向对象各种特性的基础。

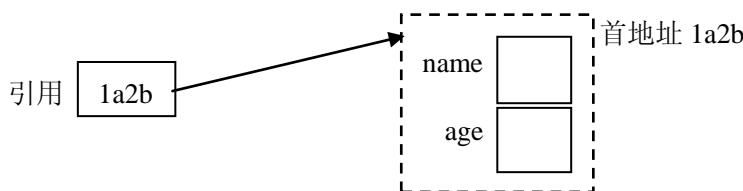
什么是引用呢？考虑下面的代码：

```
Student stu;  
stu = new Student();
```

第一行代码定义了一个对象类型的变量 stu，这个对象类型的变量就是一个引用。之前提过，定义了一个对象类型变量并没有创建一个真正的对象，事实上这只是声明了一个引用。

在计算机中，对象本质上是一块内存区域。当在 java 中创建一个对象时，本质上就是在内存中分配了一块数据区域。每一个内存中的数据区域，都会有它的内存首地址（参考数组一节中对内存首地址的描述）。因此，每一个对象，都会有个一个首地址。

在引用类型中，保存的就是内存中的首地址。下面的示意图就是引用和对象在内存中的关系。



可以看到，**引用就相当于一个指针，指向了内存中的对象。**

我们要明确的是，引用本身和对象之间，还是有区别的。引用保存的只是对象的地址，与真正的对象是有本质区别的。但是，通过引用，我们可以操作一个对象。我们可以通过引用来使用对象的属性或者调用对象的方法。

拿现实生活举例，如果把家用的空调当做对象的话，则空调的引用就是空调的遥控器。首先，遥控器和空调本身有本质的差别，遥控器不等于空调。如果家里面没有安装空调的话，即使有了空调的遥控器也没用。

其次，要想操作空调，必须通过遥控器才能操作。如果一个家用的挂壁式空调没有了遥控器的话，那就无法对这个空调进行任何操作。

例如下面的代码：

```
public class TestStudent {  
    public static void main(String[] args) {  
        Student stu1 = new Student();  
        stu1.age = 18;  
        System.out.println(stu1.age);  
    }  
}
```

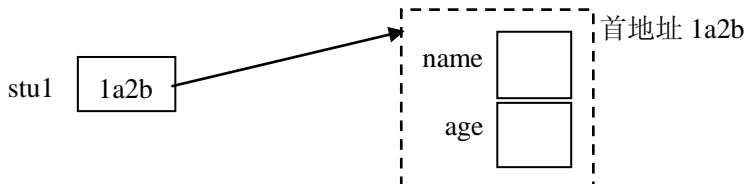
```

        stu1.study();
    }
}

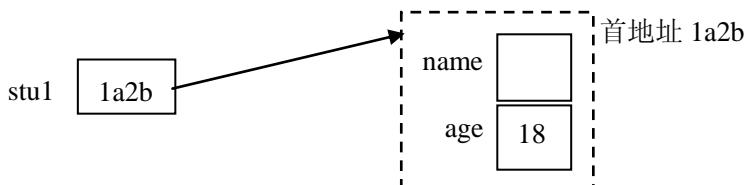
```

上面的代码演示了如何创建对象，以及如何利用引用来设置、获得对象的参数，以及调用对象的方法。

首先，在程序中有 `stu1 = new Student()` 这行代码。这行代码定义了一个 `stu1` 引用，并且创建了一个新的 `Student` 对象。在内存中的情况如下：



然后，`stu1.age = 18`。通过在引用后面加“.”，可以操作引用所指向对象的属性。这句代码的含义是，把 `stu1` 这个引用所指向对象的 `age` 属性，设为 18。执行完这句代码之后，内存中的情况如下：



然后，打印 `stu1.age`，这样获取 `age` 属性的值，输出结果为 18。

然后，调用 `stu1.study()`。这句代码的含义是：对 `stu1` 引用所指向的对象调用 `study()` 方法。程序运行结果如下：

```

D:\Book\chp6>javac TestStudent.java

D:\Book\chp6>java TestStudent
18
Student study for 8 hours

D:\Book\chp6>

```

### 3.2.2 多个引用指向同个对象

有了引用的基本概念之后，下面是对引用的进一步探讨。例如，有如下代码：

```

class MyValue{
    int value;
}

public class TestMyValue{
    public static void main(String args[]){
        MyValue mv1 = new MyValue();
        mv1.value = 100;
        MyValue mv2 = mv1;
    }
}

```

```

        mv2.value = 200;
        System.out.println(mv1.value);
    }
}

```

代码运行的结果如下：

```

D:\Book\chp6>javac TestMyValue.java

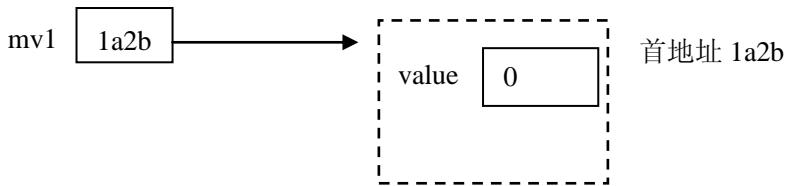
D:\Book\chp6>java TestMyValue
200

D:\Book\chp6>

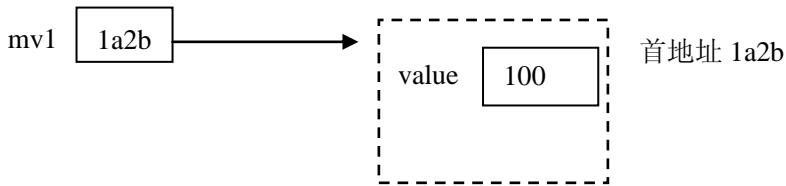
```

注意，输出 mv1 的 value 属性时，结果是 200。为什么修改了 mv2 的 value 属性之后，mv1 的 value 属性也跟着改变了呢？

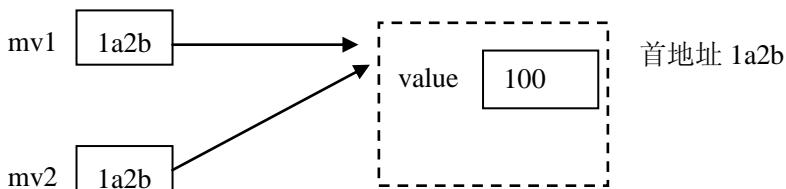
我们来分析一下代码。首先，我们创建了一个 MyValue 对象，并把这个对象的首地址赋值给 mv1 引用。内存中的图如下



注意到此时，对象的 value 属性是默认值 0。然后，执行 mv1.value = 100 这一行代码。在这行代码的意思是，把 mv1 引用所指向的对象的 value 属性设为 100，因此内存中的结果如下：

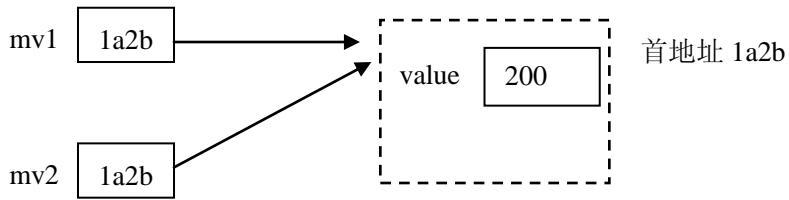


接下来，执行的是 MyValue mv2 = mv1 这一样。这一行把 mv1 的值赋值给 mv2。由于 mv1 是一个引用，而 mv1 的值，就是对象的首地址。因此。这行代码把引用 mv1 中保存的地址，赋值给另外一个引用 mv2。这样赋值之后，mv2 引用和 mv1 引用中保存的地址相同，换而言之，这两个引用指向同一个对象。在内存中的情况如下：



这样，mv1.value 指的是：“mv1 引用所指向的对象的 value 属性”，而 mv2.value 指的是“mv2 引用所指向的对象的 value 属性”。由于 mv1 和 mv2 所指向的对象相同，因此，实际上 mv1.value 和 mv2.value 指的是内存中的同一块数据。

因此，`mv2.value` 属性进行了修改成 200，就是指的把 `mv2` 引用所指向的对象的 `value` 属性设为 200。此时，在内存中的情况如下：



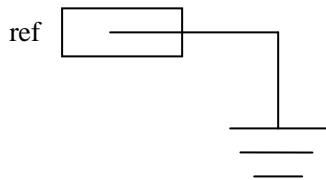
最后，打印 `mv1.value` 值时，打印的实际上是 `mv1` 引用所指向对象的 `value` 属性。由于这个属性在之前已经被修改成了 200，因此，此时打印的值就是 200。

上面的这个例子，给我们演示了两个引用指向同一个对象的例子。

### 3.2.3 对象类型的 null 值

我们在介绍默认值的时候，曾经介绍过，对于简单类型来说，默认值是各种各样的 0；而对于对象类型来说，默认值为“null”。而对象类型，也就是我们现在所说的引用。那么，引用是 `null` 值，表示的是什么含义呢？

所谓的 `null` 值，指的是：一个引用不指向任何对象。此时，这个引用中保存的地址为空，我们往往用下面的图形来表示：



我们往往用“接地”来表示引用的 `null` 值。

要注意的是，如果对一个值为 `null` 的引用调用方法或使用属性，则会产生一个错误。例如下面的代码：

```
MyValue mv1 = null;  
System.out.println(mv1.value);
```

上面的代码，把 `mv1` 这个引用的值设为 `null` 值。而在打印语句中，要输出 `mv1.value`。这表示，要输出 `mv1` 这个引用所指向的对象的 `value` 属性。由于 `mv1` 引用没有指向任何对象，因此，通过 `mv1` 引用只能找到一个 `null` 而无法找到 `value` 属性。这样，就产生了一个错误。错误如下：

```
D:\Book\chp6>javac TestMyValue.java  
  
D:\Book\chp6>java TestMyValue  
Exception in thread "main" java.lang.NullPointerException  
        at TestMyValue.main(TestMyValue.java:7)  
  
D:\Book\chp6>
```

可以看到，在编译时，程序能够正常编译通过。但是运行时，由于使用了一个 null 引用的属性，所以产生了 `NullPointerException`，这个错误叫做空指针异常，这是 java 中非常常见的一一个错误。产生这个错误的原因很简单，一定是对 null 引用调用了方法或者使用了属性。

## 2.5.2 方法参数传递

引用类型保存的是对象的地址，这个特点，在方法参数传递上表现的最为明显。本小节要介绍的就是 Java 中的方法参数传递规则。

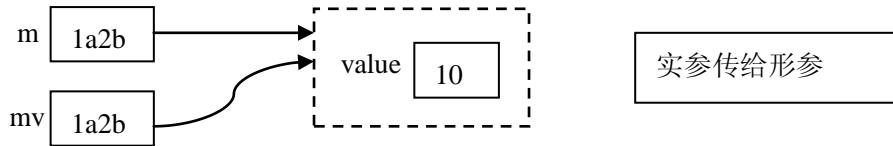
看如下代码：

```
class MyValue{  
    int value;  
}  
public class TestMyValue{  
    public static void main(String args[]){  
        int a = 10;  
        changeInt(a);  
        System.out.println(a);  
  
        MyValue m = new MyValue();  
        m.value = 10;  
        changeObject(m);  
        System.out.println(m.value);  
    }  
    public static void changeInt(int n){  
        n++;  
    }  
    public static void changeObject(MyValue mv){  
        mv.value++;  
    }  
}
```

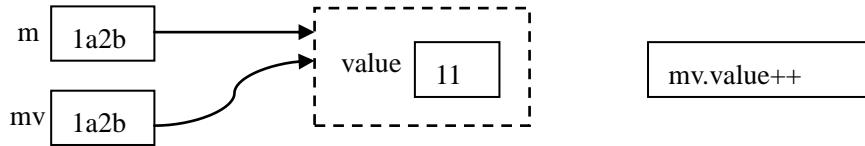
上面的程序展示了 Java 方法参数传递的问题。首先，在主方法中定义了一个 int 变量 a，并赋初值为 10，调用 `changeInt` 之后，输出 a 的结果，依然是 10。（这部分内容是上一章“函数”中的内容）。

之后，创建了一个 `MyValue` 对象，并把首地址赋值给 m。之后，为 `m.value` 赋值为 10，并调用 `changeObject` 函数。调用时，实参传给形参，传递的是对象的地址。函数调用过程如下。

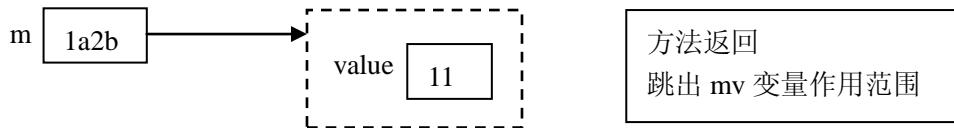
首先，在主方法中，调用了 `changeObject()` 方法，并且把 m 作为实参，传递给形参 `mv`。要注意的是，由于 m 是一个引用，保存的是一个对象的地址，因此进行传递时，传递给 `mv` 的值就是一个对象的地址。这样，实参 `m` 引用和形参 `mv` 引用中保存的内存地址相同，也就是说，这两个引用指向同一个对象。示意图如下：



然后，在 `changeObject` 方法内部，执行了 `mv.value++` 的操作。这个操作的含义是：对 `mv` 引用所指向的对象的 `value` 属性，进行+1 操作。修改之后，内存中的情况如下：



然后，`changeObject` 返回，这样 `mv` 这个形参就失去了作用范围，因此 `mv` 引用就不存在了。此时，内存中的情况如下：



此时，输出 `m.value` 的值，指的是 `m` 所指向对象的 `value` 值。这是的 `value` 值，就是修改之后的 11。

注意，对象类型当做方法的参数时，与基本类型不同。**在实参传给形参时，基本类型传递的值，就是本身的数值；而对象类型传递的值，是实参的地址。**因此，在使用上，这两者有很大的不同。因此，请记住这个结论：

在 Java 中的方法参数传递中，**基本类型传值，对象类型传引用**。

## 4 this

`this` 是一个很特殊的关键字。这个关键字在不同的用法下有不同的含义。

### 4.1 this 表示引用

首先我们来看下面这段代码：

```
class MyClass{
    int value = 10;
    public void print(){
        int value = 20;
        System.out.println(value);
    }
}
public class TestMyClass{
    public static void main(String args[]){
        MyClass mc = new MyClass();
        mc.print();
    }
}
```

```
}
```

在这段代码中，调用 print 方法时，打印的是 20。原因也很简单。我们在 MyClass 类中定义了一个实例变量 value，这个变量的值 10。但是，我们在 print 方法中，同样定义了一个局部变量 value，这个 value 的值是 20。根据实例变量的特点，当实例变量和局部变量发生命名冲突时，以局部变量优先。因此，此时输出的 value 值，就是实例变量的值 20。

那有没有办法，在局部变量和实例变量冲突的时候，明确的指明实例变量呢？这个时候，我们就可以用 this 关键字。修改后的代码如下：

```
class MyClass{
    int value = 10;
    public void print(){
        int value = 20;
        System.out.println(value);
        System.out.println(this.value);
    }
}
public class TestMyClass{
    public static void main(String args[]){
        MyClass mc = new MyClass();
        mc.print();
    }
}
```

我们在程序中加入了一行新代码，输出 “this.value”。这样，通过 “this.”，来明确的指明实例变量。运行结果如下：

```
D:\Book\chp6>javac TestMyClass.java
D:\Book\chp6>java TestMyClass
20
10
```

这样，我们就输出了实例变量的值：10。

那么 “this.” 究竟是什么含义呢？从概念上说，在这种使用 this 的用法中，this 关键字表示的是“当前对象的引用”。那么什么是“当前”对象呢？所谓当前对象，指的是：对哪个对象调用的方法，哪个对象就是当前对象。例如下面的例子：

```
class MyClass{
    int value;
    public MyClass() {}
    public void m(){
        System.out.println(this.value);
    }
}
public class TestMyClass{
    public static void main(String args[]){
        MyClass mc1 = new MyClass();
        mc1.value = 100;
```

```

        MyClass mc2 = new MyClass();
        mc2.value = 200;
        mc1.m();
        mc2.m();
    }
}

```

对 m1 和 m2 对象分别调用 m 方法。对 m1 调用 m 方法时，m 方法中的 this 指的就是 m1 对象，因此输出的值，就是 m1 对象的 value 值 100。而对 m2 调用 m 方法时，this 指的就是 m2 对象，此时输出的值就是 200。运行结果如下：

```

D:\Book\chp6>javac TestMyClass.java

D:\Book\chp6>java TestMyClass
100
200

D:\Book\chp6>

```

当然，在上面的代码中，由于 value 没有命名冲突，所以如果不加 this，直接输出 value 值，同样输出的是当前对象的实例变量。下面的代码运行结果相同：

```

class MyClass{
    int value;
    public MyClass() {}
    public void m(){
        System.out.println(value);
    }
}

```

因此，当局部变量和实例变量没有命名冲突时，表示实例变量时，可以不用 this。

另外，“this.” 这种用法，在构造方法中也非常常见。例如下面的代码：

```

class MyClass{
    int value;
    public MyClass() {}
    public MyClass(int value){
        this.value = value;
    }
}

```

在第二个有参构造方法中，参数名为 value。由于参数是特殊的局部变量，这个参数就跟实例变量 value 产生了命名冲突。于是，在构造方法中，我们就是用了 this 关键字，这句话：

this.value = value;

“=” 左边的 this.value 表示的是实例变量，而 “=” 右边的 value 指的是局部变量，也就是构造方法的参数。这句话的含义是，把构造方法的参数赋值给实例变量。

这种赋值的方法，在构造方法中非常常见。我们常常可看到这样的代码：

```

class Student{
    String name;
    int age;
}

```

```

public Student() { }
public Student(String name, int age) {
    this.name = name;
    this.age = age;
}
}

```

在这个 `Student` 类中，我们定义了带两个参数的构造方法。在这个构造方法内部，把构造方法的两个参数，赋值给了对象的属性。于是在创建对象时，就可以写出这样的代码：

```

public class TestStudent {
    public static void main(String[] args) {
        Student stu = new Student("Tom", 18);
        System.out.println(stu.name + " " + stu.age);
    }
}

```

这段代码在创建学生对象时，调用了其有参构造方法，并把学生对象的 `name` 属性设为 `Tom`，`age` 属性设为 `18`。程序运行结果如下：

```

D:\Book\chp6>javac TestStudent.java

D:\Book\chp6>java TestStudent
Tom 18

D:\Book\chp6>

```

## 4.2 this 用在构造方法中

除了把 `this` 当做当前对象的引用之外，`this` 还有一种其他的用法。

假设一个对象的构造比较复杂，例如：

```

class MyClass{
    int value;
    public MyClass(){
        //非常复杂的构造过程...
    }
    public MyClass(int value){
        //复杂的构造过程...
        ...
        this.value = value;
    }
}

```

假设有参数的构造方法 `MyClass(int value)` 相对 `MyClass()` 构造方法，只是多出一个语句：`this.value = value;`。除此之外的构造过程完全相同。既然这部分代码完全相同，那么最合理的想法，应当是重用 `MyClass()` 这个无参构造函数。为此，我们希望能在 `MyClass(int value)` 这个有参构造方法中，调用无参的构造方法。

**如果在一个类的构造方法中，要调用本类的其他构造方法，就必须使用 `this` 关键字。语法是 `this` 加上一个圆括号，圆括号中加上调用构造方法时传递的参数。例如，在这个例子中，**

我们要调用 MyClass 类的无参构造方法，于是，就应当写成：

```
this();
```

所以，上面的 MyClass(int value)，就可以写成：

```
public MyClass(int value) {  
    this();  
    this.value = value;  
}
```

这样，就实现了我们所说的，在 MyClass(int value)这个构造方法中，调用 MyClass()这个无参构造方法。

在使用 this()这种语法时，要注意三个要点：

1、只能在构造方法中使用，并且只能调用本类的其他构造方法。

例如，下面的代码会编译出错：

```
class MyClass{  
    int value;  
    public MyClass(){  
        System.out.println("MyClass()");  
    }  
  
    public void m(){  
        this();  
        System.out.println("m()");  
    }  
}
```

在这个程序中，我们在一个普通的 m()方法里面，使用 this()调用无参构造方法。这回产生一个编译时错误，出错信息如下：

```
D:\Book\chp6>javac TestMyClass.java  
TestMyClass.java:8: call to this must be first statement in constructor  
        this();  
               ^  
1 error  
  
D:\Book\chp6>
```

2、在使用时，this()必须作为构造方法的第一个语句，否则编译出错。

例如下面的代码：

```
class MyClass{  
    int value;  
    public MyClass(){  
        System.out.println("MyClass()");  
    }  
  
    public MyClass(int value){  
        System.out.println("MyClass(int)");  
        this();  
    }  
}
```

```
    }
}
```

在这段代码中，在 MyClass(int value)这个有参构造方法中，利用 this()调用了无参构造方法。但是，由于 this()不是构造方法的第一个语句，因此编译出错，出错信息如下：

```
D:\Book\chp6>javac TestMyClass.java
TestMyClass.java:9: call to this must be first statement in constructor
        this();
               ^
1 error

D:\Book\chp6>
```

需要强调的是，对 this()的调用必须是构造方法中的第一个语句，这个限制，指的是对“this + ()”的调用。而“this + .”的语法，不受这个限制。例如，下面的代码能够顺利的编译通过。

```
class MyClass{
    int value;
    public MyClass(){
        System.out.println("MyClass()");
    }

    public MyClass(int value){
        this(); //对 this() 的调用必须是构造方法的第一个语句
        this.value = value; //对 this. 的调用没有限制
    }
}
```

### 3、this()不能够递归调用。

在介绍函数时，我们曾经给大家介绍过递归的概念：如果在一个函数的内部，调用这个函数本身，则这种情况称之为递归。而 this()不允许递归调用，也就是说，构造方法不能调用自身。例如下面的代码：

```
class MyClass{
    int value;
    public MyClass(){
        this();
        System.out.println("MyClass()");
    }
}
```

这个类，在 MyClass()这个构造方法中，利用 this()调用了本身，从而形成了递归调用。这样会产生一个编译时错误：

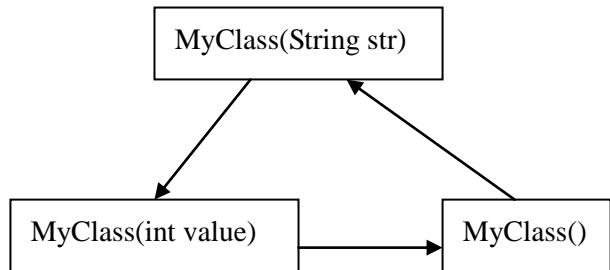
```
D:\Book\chp6>javac TestMyClass.java
TestMyClass.java:3: recursive constructor invocation
    public MyClass(){^
1 error

D:\Book\chp6>
```

这种构造方法内部，直接调用自身，被称为直接递归。除了这种方式之外，还有间接递归。例如：

```
class MyClass{  
    public MyClass(){  
        this("hello");  
    }  
    public MyClass(int value){  
        this();  
    }  
    public MyClass(String str){  
        this(10);  
    }  
}
```

在这个程序中，`MyClass(String str)`调用了`MyClass(int value)`，而`MyClass(int value)`调用了`MyClass()`，最后`MyClass()`又调用了`MyClass(String str)`。示意图如下：



这样，三个方法之间的调用形成一个循环，这样也构成递归调用。因此，这也会产生一个编译错误。错误信息与直接递归的错误信息相同，在此不再重复。

# Chp7 面向对象三大特性

## 本章导读

面向对象三大特性指的是：封装、继承、多态。这三大特性支撑了整个面向对象的知识和理论体系，是面向对象的核心。本章要介绍 Java 中这三大特性是如何体现的，以及与这三大特性相关的 Java 语法。

### 1 封装

有如下代码：

```
class CreditCard{  
    String password = "123456";  
}  
public class TestCreditCard{  
    public static void main(String args[]){  
        CreditCard card = new CreditCard();  
        System.out.println(card.password);  
        card.password = "000000";  
        System.out.println(card.password);  
    }  
}
```

上述代码，创建了一个信用卡对象，并且读取、修改了这个对象的 password 属性。

从 Java 基本语法上说，这并没有问题。但是对于生活来说，这就是一个大问题！对于信用卡对象而言，它的密码属性是不应该被随便访问和修改的。

面向对象中解决这个问题，可以采用封装的特性。封装指的是，任何对象都应该有一个明确的边界，这个边界对对象内部的属性和方法起到保护的作用。

#### 1.1 属性的封装

为上述的 CreditCard 的 password 属性增加 private 关键字，如下：

```
class CreditCard{  
    private String password = "123456";  
}
```

则原有代码中会出现编译错误：

```
public class TestCreditCard{  
    public static void main(String args[]){  
        CreditCard card = new CreditCard();  
        System.out.println(card.password); //编译错误  
        card.password = "000000"; //编译错误  
        System.out.println(card.password); //编译错误  
    }  
}
```

当为属性增加 `private` 之后，这个属性就成为了一个私有属性。所谓私有，指的是该属性只能在本类内部访问。例如，当我们把 `password` 属性设置为 `private`，对这个属性的访问就只能局限在 `CreditCard` 类的内部。现在我们试图在 `TestCreditCard` 类中访问这个属性，编译器就会报出编译错误。这就相当于，`card` 对象的边界对于 `password` 属性起到了保护的作用，任何试图越过边界，访问 `password` 属性的企图都会被阻止。

然而，对于用户而言，依然有可能要访问 `CreditCard` 的密码。例如，在生活中，如果忘了银行卡密码，我们可以凭借证件到银行去查询或重设密码。

对于这方面的需求，我们为 `CreditCard` 提供一对 `get/set` 方法。这两个方法的修饰符为“`public`”。用 `public` 修饰的属性和方法表示“公开的”，公开属性和方法不受对象边界的限制，在类的内部和外部都可以访问。代码如下：

```
class CreditCard{  
    private String password = "123456";  
    public void setPassword(String password){  
        this.password = password;  
    }  
    public String getPassword(){  
        return this.password;  
    }  
}
```

则 `TestCreditCard` 类可以改成：

```
public class TestCreditCard{  
    public static void main(String args[]){  
        CreditCard card = new CreditCard();  
        System.out.println(card.getPassword());  
        card.setPassword("000000");  
        System.out.println(card.getPassword());  
    }  
}
```

很显然，用户可以调用 `getPassword` 方法来获取 `password` 属性，调用 `setPassword` 方法来设置 `password` 属性。

下面是一个初学者常问的问题：既然提供了 `get/set` 方法就是为了访问属性，那又何必把属性作为私有？把属性做成公开的直接访问不行么？

把属性作为私有，并提供相应的 `get/set` 方法，最重要的概念在于：控制。由于不能直接访问属性，而必须通过 `get/set` 方法访问属性，因此可以在 `get/set` 方法上做手脚，来控制他人对对象属性的访问。

例如，希望 `password` 属性只能被获取，不能被改写。如果 `password` 属性用 `private` 修饰的话，可以只提供 `get` 方法而不提供 `set` 方法，这样 `password` 就成为了只读属性。而如果不把 `password` 做成私有，则无法达到“只读”的效果。

再例如，银行要求，信用卡的密码长度必须为 6，如果没有把 `password` 属性做成 `private` 的，那么下面的代码一定是正确的：

```
card.password="12345678";
```

我们就无法限制密码的长度了。而现在我们把 `password` 属性设置为 `private`，用户就只能通过 `setPassword` 方法来设置 `password` 属性：

```
card.setPassword("12345678");
```

我们就可以在 setPassword 方法中增加一个判断：密码长度为 6 才允许设置。

修改原有代码如下：

```
public void setPassword(String password) {  
    if (password.length() != 6) return;  
    this.password = password;  
}
```

这样，如果密码长度不为 6 的话，设置密码就不会成功。

从上面一个例子中，我们可以看到，把属性设为 private，并提供相应的 get/set 方法，程序员才能够对属性的访问增加控制。因此，从现在开始，应当养成良好的习惯，把类的属性都做成 private 的，然后再提供相应的 get/set 方法。

## 1.2 方法的封装

除了属性之外，我们也可以将方法设置为 private。

在前面的章节中，读者往往能看到，一个方法会包含修饰符“public”，这表示该方法是公开的，不受对象边界的限制。而我们同样可以把方法修饰为“private”，与属性类似，一个被修饰为“private”的私有方法只能在类的内部访问。

我们在设计一个类的时候，会为这个类设计很多方法。有些方法应该做成 public 方法，以供其他对象来调用，而有些方法只供自身调用，不作为对象对外暴露的功能，就应该做成 private 方法。例如，一个老师对象，拥有一个“讲课”方法，这个方法必须暴露出来，供学生对象来调用（老师从来不会讲课给自己听），因此这个方法应该是公开的。同时，老师作为一个人，还拥有“消化食物”方法，这个方法只供老师自己来调用，对别人是无益的，因此，应该是一个私有方法。

```
class MyClass{  
    public void method1(){ }  
    private void method2(){ }  
}  
  
public class TestPrivateMethod {  
    public static void main(String[] args) {  
        MyClass mc = new MyClass();  
        mc.method1(); // 正确，method1方法为公开的，可以在类外面访问  
        mc.method2(); // 编译失败，不能访问mc对象的私有方法  
    }  
}
```

如上述代码，MyClas类中具有两个方法，method1方法为public的，因此可以在类外面调用；method2方法为private的，不能在类的外部调用，否则会引发一个编译错误。

## 2 继承

### 2.1 继承的基本概念

继承是面向对象中另一个非常重要的概念。那继承的思想是怎么来的呢？我们首先从一个例子看起。

例如，我们要设计两个类：一个 Dog 类，一个 Cat 类，分别表示狗和猫。首先，我们来设计 Dog 类。首先考虑狗类的属性，也就是狗“有什么”。狗有年龄，有性别，因此狗就

有一个 age 属性和一个 sex 属性。其次，考虑狗类的方法，也就是狗“能干什么”。我们说，狗能够吃东西，狗也能够睡觉，狗还能够看家护院。因此，我们为狗设计三个方法：eat()、sleep()和 lookAfterHouse()。代码如下：

```
class Dog{
    int age;
    boolean sex; //true 表示雄性, false 表示雌性

    public void eat(){
        System.out.println("eat()");
    }

    public void sleep(){
        System.out.println("sleep()");
    }

    public void lookAfterHouse(){
        System.out.println("Dog can look after house");
    }
}
```

接下来，我们来设计 Cat 类。猫类有哪些属性呢？猫类有年龄，有性别，因此猫类有 age 和 sex 属性。那么猫有哪些方法呢？猫能够吃东西，猫能睡觉，猫还能抓老鼠。因此，我们为猫类设计三个方法：eat()、sleep()和 catchMouse()。代码如下：

```
class Cat{
    int age;
    boolean sex;

    public void eat(){
        System.out.println("eat()");
    }

    public void sleep(){
        System.out.println("sleep()");
    }

    public void catchMouse(){
        System.out.println("Cat can catch mouse");
    }
}
```

这样，我们就完成了 Dog 类和 Cat 类的设计。

我们仔细分析一下 Dog 类和 Cat 类的代码，会发现这两个类中有大量相同的代码。例如，在这两个类中，都有 age 和 sex 属性；同样的，都有 eat()和 sleep()方法。为什么这两个类有这么多类似的代码呢？这是由这两个类的特点决定的。在生活中，狗和猫这两类事物有着很多的共性，例如能“吃”，能“睡”。而这些正是“动物”这个类的特点。也就是说，狗

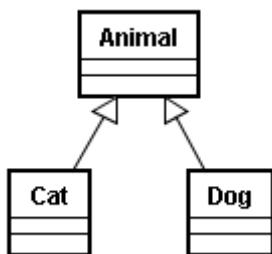
和猫，都是特殊的动物。为了能够更好的表达这个概念，我们把 Cat 和 Dog 的共性抽象出来，写在另外的一个类 Animal 中。这个类包含了 Cat 类和 Dog 类中的共性。

代码如下：

```
class Animal{  
    int age;  
    boolean sex;  
    public void eat(){  
        System.out.println("Animal Eat");  
    }  
  
    public void sleep(){  
        System.out.println("sleep 8 hours");  
    }  
}
```

我们让 Animal 类的 eat 方法输出 “Animal Eat”，让 Animal 的 sleep()方法输出：“sleep for 8 hours”（我们假设所有动物都是睡 8 个小时）。

然后，我们让 Cat 类和 Dog 类继承自 Animal 这个类。这样，Animal 类与 Cat 和 Dog 类之间，就形成了继承关系。而被继承的类 Animal，被称为“父类”；而 Cat 与 Dog 这两个类继承自 Animal，被称为“子类”。用图来表示如下：



我们如何来理解父类和子类呢？

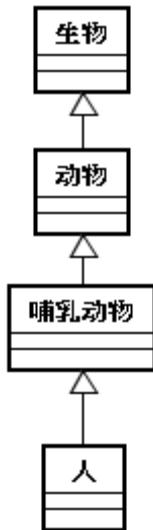
在生活中，首先，我们见到了很多的猫对象、狗对象、猴子对象、大象对象等，从而形成了猫类、狗类、猴子类、大象类等这些类。这些都是一些具体的类，而当我们遇到大量的具体的类时，会经过总结和归纳，抽象出他们的共性，形成一个新的概念：动物。这样，我们在脑子中，就有了“动物”这个类的概念。这个类的出现并不是因为我们见到了“动物”对象，而是我们从大量具体的类中总结，归纳出来的。

上面我们讲述的过程，就是父类产生的过程：父类，是对子类共性的抽象。

我们也可以从另一角度来理解。从上面的代码中，我们可以看到，由于 Animal 中已经定义了 age、sex 属性以及 eat()、sleep()方法，因此在两个子类 Cat 和 Dog 中，这些共性不需要重复定义，只需要写出 Cat 和 Dog 类中的特性就可以了。因此，在父类中，往往可以定义一些比较一般的属性和方法，而在子类中，定义子类特有的属性或者方法。也就是说，父类和子类的关系，是由一般到特殊的关系。例如，Animal 是一般的类，而 Dog 和 Cat 是特殊的类，Dog 和 Cat 可以当做是特殊的 Animal。

由于父类是子类的共性的抽象，是一个一般的类，因此，我们在进行继承关系设计的时候，应当尽量把子类的共性放在父类，特性放在子类。

例如，有下面的继承关系：



这是一个非常典型的继承关系。生物，是一个很宽泛的概念，包括植物、动物等，都可以看做是生物。而动物，就是一种特殊的生物，因此，动物类是生物类的子类。同样的，哺乳动物是特殊的动物，而人又是特殊的哺乳动物。

现在，我们要设计一系列的方法：繁殖()，喂奶()，制造工具()。这三个方法应当分别写在哪一个类中呢？

繁殖是所有生物的共性，因此，繁殖()方法应当作为生物类的方法；喂奶是哺乳动物的共性，因此这个方法应当写在哺乳动物类中；而制造工具()是人类特有的方法，因此这个方法应当写在人这个类中。

上面的部分，我们简单介绍了一下继承的基本概念：父类是对子类共性的抽象，父类和子类的关系，是由一般到特殊的关系。在设计类的继承关系时，应当把共性放在父类，特性放在子类。

## 2.2 继承的基本语法

从语法上说，**继承使用关键字：extends**。在定义子类的时候，可以用 extends 关键字说明这个类的父类是哪一个类。代码如下：

```

class Animal{
    int age;
    boolean sex;
    public void eat(){
        System.out.println("Animal eat");
    }

    public void sleep(){
        System.out.println("sleep 8 hours");
    }
}

class Dog extends Animal{
    public void lookAfterHouse(){
        System.out.println("look after house");
}
  
```

```

    }
}

class Cat extends Animal{
    public void catchMouse(){
        System.out.println("catch mouse");
    }
}
public class TestDog {
    public static void main(String args[]){
        Dog d = new Dog();
        d.sex = true;
        d.age = 3;
        d.eat();
        d.lookAfterHouse();
    }
}

```

我们可以看到，在代码中，使用 `extends` 关键字，来表明 `Cat` 类和 `Dog` 类继承自 `Animal`。由于共性在父类 `Animal` 中，因此，在 `Cat` 和 `Dog` 中，`Animal` 类中已经有的代码，没有必要进行重复。这样，程序中就少了很多重复和冗余的代码。与此同时，在主方法中，我们创建了一个 `Dog` 对象，并且，修改了这个 `Dog` 对象的 `sex` 属性和 `age` 属性，并调用了这个对象的 `eat()` 方法和 `lookAfterHouse()` 方法。

需要注意的是，在 `Dog` 类中，我们没有写代码来定义 `sex` 和 `age` 属性，也没有 `eat()` 方法的代码。这两个属性和一个方法，是 `Dog` 类从 `Animal` 类中继承而来的。也就是说，父类中的属性和方法，被子类继承之后，相当于子类中也有了相应的属性和方法。

这样，在父类中定义了属性和方法之后，子类中就能够直接继承，这样，就让父类中的代码得到了重用，从而提高了代码的可重用性。同时，子类也能够写一些子类的特性，这样，就在父类的基础上增加一些功能，体现了面向对象的可扩展性。

### 2.3 什么能被继承？

在上一节的例子中，我们在 `Animal` 类中定义了 `age` 和 `sex` 属性，这两个属性能够被子类 `Dog` 类和 `Cat` 类继承，并且我们在创建 `Dog` 对象之后，能够直接使用 `age` 和 `sex` 属性，也能够调用 `Animal` 类中定义的方法。

那是不是父类中所有的属性和方法，都能够被子类继承呢？我们看下面的例子：

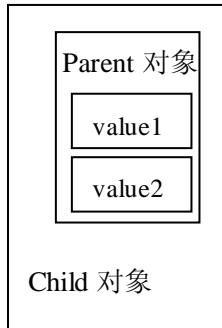
```

class Parent{
    int value1;
    private int value2;
}
class Child extends Parent{
    public void method(){
        value1 = 100; // 编译正确 Child 类从父类中继承到 value1 属性
        value2 = 200; // 编译错误，Child 没有继承到 value2 属性
    }
}

```

上面的例子中，Parent 类中的 value1 属性能够被 Child 类继承，所以在 Child 类的内部可以访问 value1 属性。但是，value2 属性被 private 关键字修饰，这就意味着，value2 只能在 Parent 类内部访问。对于 Parent 类来说，Child 类属于是 Parent 类的外部，因此不能够直接访问 value2 属性。

那么 Child 类中有没有 value2 属性呢？从空间上来说，在创建 Child 对象的时候，会在 Child 对象的内部，包含着一个 Parent 对象。在内存中的示意图如下：



可以看到，Child 对象的内部，包含了一个 Parent 对象。因此，在这个 Child 对象中，有两块数据区域，用来保存 parent 对象的 value1 属性和 value2 属性。但是，由于 value2 只能在 Parent 类内部访问，因此在 Child 对象中，无法访问 Parent 类中的 value2 属性。

也就是说，对于 value2 来说，在 Child 对象中有这个属性的空间，但是却无法访问。这种情况，我们就说 Parent 类中的 value2 属性无法被 Child 类继承。也就是说，父类中，无法被子类访问的属性和方法，不能被继承。

怎么来理解这一点呢？例如，一个百万富翁，死后留下了一大笔财产。有一部分财产，放在家里的抽屉中，他的儿子能够打开这个抽屉，取出这些钱。因此，这部分财产，他的儿子能够访问，因此上，可以认为他的儿子能够继承。

但是，如果还有一大笔钱，富翁存在了一个保险箱里面，这个保险箱的密码只有富翁本人知道。那么富翁死了之后，他的儿子算不算有这笔钱呢？从理论上说，他的儿子家里有一块空间用来存放这笔钱，他儿子也知道有这么一笔钱。但是，钱存在保险箱里面取不出来，那么这笔钱他儿子根本没法花，从实际效果来看，跟他儿子没有这笔钱一样。也就是说，这部分财产他的儿子无法访问，因此我们可以认为，他的儿子没有继承到这部分财产。

也就是说，只有子类能够访问的属性和方法，才能够被子类继承。

## 2.4 访问权限修饰符

上一节我们说到，只有能被子类访问的属性和方法，才能被子类继承。那么，怎么判断哪些属性和方法能够被访问，哪些不能被访问呢？这需要看属性或者方法的访问权限修饰符。

在 Java 中，总共有四种访问修饰符。除了我们之前介绍的 `private` 和 `public` 两个修饰符之外，还有两个跟访问权限相关的修饰符：`default` 以及 `protected`。要注意的是，`default` 修饰符指的是：如果在属性或方法前面，不加任何的访问修饰符（即不加 `private`、`public`、`protected`），则访问权限是 `default` 权限。例如：

```
class Student {  
    int age;  
}
```

在这个 `Student` 类中，其 `age` 属性没有加上任何访问修饰符，因此 `age` 属性的权限就是 `default` 权限。要注意的是，要把一个属性设为 `default` 时，千万不能写上“`default`”这个单

词。例如下面的代码就是错误的：

```
class Student{  
    default int age; //编译错误!  
}
```

`private` 和 `public` 这两个修饰符不再赘述。我们详细介绍下 `default` 和 `protected` 修饰符。

访问权限为 `default` 的属性或者方法，只能被本类或者同包的其他类访问。例如，有下面的代码：

```
package p1; //MyClassA 类处于 p1 包下  
public class MyClassA{  
    int value = 10; //value 属性的访问权限是 default  
    public void m(){  
        System.out.println(value); //value 属性能够在本类内部访问  
    }  
}  
  
//MyClassB 与 MyClassA 处于同一个包下  
package p1;  
public class MyClassB{  
    public void m2(){  
        MyClassA mca = new MyClassA();  
  
        //由于在同一个包下  
        //所以 MyClassB 中能够访问 MyClassA 对象的 value 属性  
        mca.value = 100;  
        System.out.println(mca.value);  
    }  
}  
  
package p2; //MyClassC 与 MyClassA 不在同一包下  
public class MyClassC{  
    public void m3(){  
        p1.MyClassA mca = new p1.MyClassA();  
  
        //由于不在同一个包下  
        //所以 MyClassB 中不能访问 MyClassA 对象的 value 属性  
        mca.value = 200; // 编译错误!  
    }  
}
```

对于 `default` 的属性和方法而言，只有同包的类才能够访问。**因此，只有那些和父类在相同包下的子类，才能够继承 `default` 修饰的属性和方法。**例如下面的代码：

```
package p1;  
public class Parent{  
    int value = 20;
```

```

}

package p1; //与 Parent 同一个包
public class Child1 extends Parent{
    public void m1(){
        System.out.println(value); //继承了 Parent 的 value 属性
    }
}

package p2; //与 Parent 不同包
public class Child2 extends p1.Parent{
    public void m2(){
        //编译错误！Child2 类和 Parent 不同包，没有继承 value 属性
        System.out.println(value);
    }
}

```

用 `protected` 修饰符修饰的属性和方法，能够被本类内部、同包的类以及非同包的子类访问。首先，`protected` 修饰符的访问权限，要比 `default` 的访问权限大。其次，由于 `protected` 的属性和方法，能够被同包的类访问，因此，必定能被同包的子类所继承。另一方面，这样的属性和方法也能被非同包的子类访问，因此，用 `protected` 修饰的属性和方法，能够被同包的子类和非同包的子类访问，也就是能被所有的子类继承。换句话说，用 `protected` 修饰的属性和方法一定能够被子类继承。

但是，如果是非同包的非子类，则无法访问 `protected` 属性或方法。例如下面的例子：

```

package p1;
public class Parent{
    protected int value;
}

package p1;
public class SamePackage{
    public void m1(){
        Parent p = new Parent();
        System.out.println(p.value); //同包的类可以访问
    }
}

package p2;
public class Child extends p1.Parent{
    public void m2(){
        //非同包的子类可以访问父类的 value 属性
        System.out.println(value);
    }
}

```

```

package p2;
public class Other {
    public void m2() {
        p1.Parent p = new p1.Parent();
        //编译错误！Other 类和 Parent 类并不同包，也没有继承关系，不能访问
        System.out.println(p.value);
    }
}

```

上面就是四种访问修饰符的介绍，我们对这四种访问权限修饰符总结如下：

修饰符	访问范围	是否能被子类继承
<b>private</b>	本类内部	不能被继承
(default)	本类内部+同包的其他类	能被同包的子类继承
<b>protected</b>	本类内部+同包的其他类+非同包的子类	能被继承
<b>public</b>	公开，能被所有类访问	能被继承

这张表所列的四种访问权限修饰符，按照“private→default→protected→public”的顺序，访问权限依次变宽。

## 2.5 方法覆盖

之前的代码中，我们为 Animal 写了一个 sleep()方法。由于 Dog 类继承自 Animal 类，因此 Dog 类中也有一个 sleep()方法。调用这个 sleep()方法时，实际上调用的是 Animal 类中的 sleep()方法。代码如下：

```

class Animal{
    public void eat(){
        System.out.println("Animal eat");
    }

    public void sleep(){
        System.out.println("sleep 8 hours");
    }
}

class Dog extends Animal{
    public void lookAfterHouse(){
        System.out.println("Dog can look after house");
    }
}

public class TestDog {
    public static void main(String args[]){
        Dog d = new Dog();
        d.sleep();
    }
}

```

```
    }
}
```

运行结果如下：

```
sleep 8 hours
```

这样，在调用 Dog 类的 sleep 方法时，实际上调用的是 Dog 类从 Animal 类中继承来的 sleep 方法。

现在，我们有了新的需求。假设说，Dog 类与 Animal 这个类相比，有自己的特点。狗与大多数动物不同，因为它要看家护院，所以它的睡眠时间，比一般动物要短。假设说，狗每天睡觉都是睡 6 个小时。

从概念上说，Animal 类中有 sleep 方法，而 Dog 类中也有 sleep 方法。在上一章我们提到过，方法分为两个部分：方法的声明和方法的实现。其中，方法声明表示一个类“能做什么”，而方法实现表示“怎么做”。Animal 和 Dog 类中都有 sleep 方法，就可以理解为，Animal 能睡觉，而 Dog 也能睡觉。也就是说，父类和子类的 sleep 方法，在方法声明上是一致的。

但是，Animal 类中的 sleep 方法，输出“sleep 8 hours”，而我们希望 Dog 类中的 sleep 方法输出“sleep 6 hours”。这样，在方法声明相同的基础上，我们希望 Dog 类有一个和父类不同的特殊实现。为此，我们可以在 Dog 类中，再次定义 sleep 方法，将父类继承下来的 sleep 方法重新实现。代码如下：

```
class Dog extends Animal{
    public void lookAfterHouse(){
        System.out.println("Dog can look after house");
    }
    public void sleep(){
        System.out.println("sleep 6 hours");
    }
}
public class TestDog {
    public static void main(String args[]){
        Dog d = new Dog();
        d.sleep();
    }
}
```

运行结果如下：

```
sleep 6 hours
```

在上面的代码中，我们在 Dog 类中，为 sleep 方法给出了一个新的实现，来替换从 Animal 类中继承的 sleep 方法的实现。像这样，**子类中用一个特殊实现，来替换从父类中继承到的一般实现，这种语法叫做“方法覆盖”。**

这样，运行代码时，程序输出：sleep 6 hours。说明程序运行时，对 Dog 对象调用 sleep 方法时，真正调用的是 Dog 类覆盖以后的方法。

**从语法上说，方法覆盖对方法声明的五个部分都有要求。具体来说，**

**1. 访问修饰符相同或更宽**

例如，父类的方法如果是 protected 方法，子类如果想要覆盖这个方法，则修饰符至少是 protected，也可以是 public，但是不能是 default 或者 private 的。

例如下面的例子：

```
class Super{
    protected void m() {}
}

class Child extends Super{
    void m() {} //编译错误!
}
```

由于子类中试图用一个 `default` 权限的 `m` 方法覆盖父类中 `protected` 权限的 `m` 方法，会让上面的代码会产生一个编译错误。错误如下：

```
D:\Book\chp7>javac TestOverride.java
TestOverride.java:5: m() in Child cannot override m() in Super; attempting to assign weaker access privileges; was protected
        void m(){}
                           ^
1 error

D:\Book\chp7>
```

### 2. 返回值类型相同。

如果返回值类型不同，则会产生一个编译错误。例如下面的代码

```
class Parent{
    public void m() {}
}

class Child extends Parent{
    public int m(){
        return 0;
    }
}
```

在这段代码中，子类的覆盖方法返回值为 `int` 类型，而由于父类方法的返回值为 `void` 类型，子类和父类方法的返回值不同，因此子类方法无法覆盖父类方法，所以会产生一个编译错误。错误如下。

```
D:\Book\chp7>javac ErrorOverride.java
ErrorOverride.java:6: m() in Child cannot override m() in Parent; attempting to use incompatible return type
found   : int
required: void
        public int m(){}
                           ^
1 error

D:\Book\chp7>
```

### 3. 方法名相同。

这是必然的，如果方法名不同，则谈不到覆盖。

### 4. 参数表相同。

要注意的，如果不满足参数表不同，编译不出错，但是不构成方法覆盖。例如

```
class Super{
    void m() {System.out.println("m in super");}
}
```

```
class Sub extends Super{  
    void m(int n){System.out.println("m(int) in sub");}  
}
```

这个代码能够编译通过。但是，虽然 `Sub` 中定义了一个 `m` 方法，但是这个方法并没有替换父类中的 `m` 方法的实现。相反，由于 `Sub` 类继承自 `Super` 类，因此 `Sub` 类中会继承到一个无参的 `m()` 方法；而此外，`Sub` 类中还定义了另一个带 `int` 参数的 `m(int n)` 方法。这样，`Sub` 类中就有了两个 `m` 方法，这两个方法方法名相同，参数表不同，因此，这样两个方法构成重载。所以，这样的代码编译不会出错，但是 `Sub` 类中的 `m` 方法并没有覆盖 `Super` 类中的 `m` 方法。

要注意的是，方法重载并不仅仅包含这些条件。方法重载还对包括 `static` 修饰符的要求、对抛出的异常的要求，等等。相关内容会在后续的课程中陆续进行介绍。

## 2.6 对象创建的过程

在有了继承关系之后，对象创建过程就变的相对复杂一些了。由于子类对象中包含一个父类的对象（虽然，父类对象中的有些属性无法访问，但是还是会创建一个完整的父类对象），因此创建子类对象时必然要先创建父类对象。

在有了继承关系之后，对象创建过程如下：

1. 分配空间。要注意的是，分配空间不光是指分配子类的空间，子类对象中包含的父类对象所需要的空间，一样在这一步统一分配。在分配空间的时候，会把所有的属性值都设为默认值。
2. 递归的构造父类对象。这一过程我们会在下面进一步介绍
3. 初始化本类属性。
4. 调用本类的构造方法。

我们下面结合一个具体的例子来介绍对象创建的过程。

假设有如下代码：

```
class A{  
    int valueA = 100;  
    public A(){ valueA=150; }  
}  
class B extends A{  
    int valueB = 200;  
    public B(){ valueB=250; }  
}  
class C extends B{  
    int valueC = 300;  
    public C(){ valueC=350; }  
}  
public class TestInherit{  
    public static void main(String args[]){  
        C c = new C();
```

```
    }  
}
```

我们在主方法中创建了一个 C 对象，则创建时的过程如下。

1. 分配空间。在分配空间时，会把 C、B、A 这三个对象的空间一次性都分配完毕，然后把这三个对象的属性都设为默认值。这样，value1, value2, value3 这三个属性都被设置为 0
2. 递归构造 C 对象的父类对象。在这里，要 C 对象的父类对象，就是 B 对象。因此，在这一步需要创建一个 B 对象。
3. 初始化 C 的属性，即把 valueC 赋值为 300
4. 调用 C 的构造方法。

其中，第 2 步，C 对象的父类为 B 对象，因此必须要先创建一个 B 对象。创建 B 对象不用重新分配空间，需要以下几步：

- 2.1 递归的构造 B 对象的父类对象
- 2.2 初始化 B 属性：把 valueB 赋值为 200
- 2.3 调用 B 的构造方法。

在 2.1 这个步骤中，递归的创建 B 对象的父类对象，也就是创建 A 对象。创建 A 对象不需要分配空间，因此，A 对象的创建有这样几步：

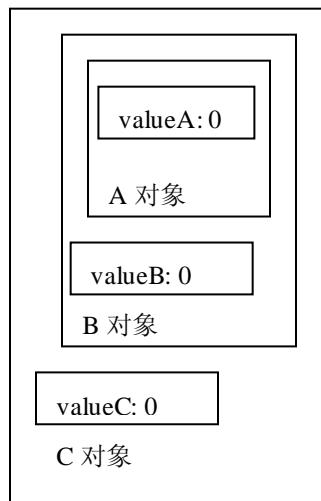
- 2.1.1 创建 A 对象的父类对象。这一步在运行时，没有任何的输出。
- 2.1.2 初始化 A 的属性，把 valueA 赋值为 100
- 2.1.3 调用 A 的构造方法。

总结一下：创建 C 对象的步骤一共有 7 步：

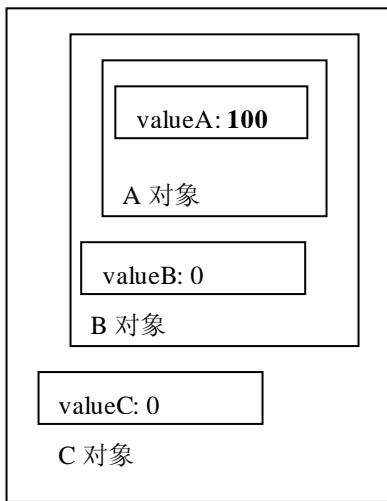
1. 分配空间
2. 初始化 A 类的属性
3. 调用 A 类的构造方法
4. 初始化 B 类的属性
5. 调用 B 类的构造方法
6. 初始化 C 类的属性
7. 调用 C 类的构造方法

每个步骤的内存示意图如下：

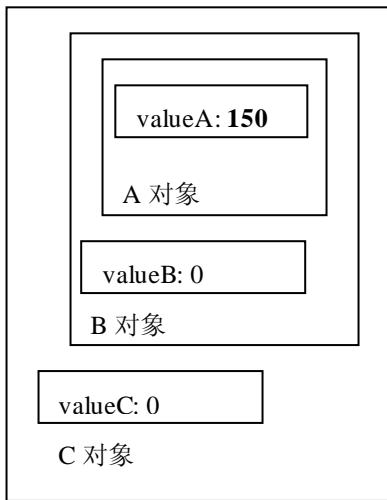
1. 分配空间，如之前所述，在一次分配空间时，会把整个继承关系中涉及到的类所需要的空间，都分配完毕，并把所有属性都设为默认值 0。如下图所示：



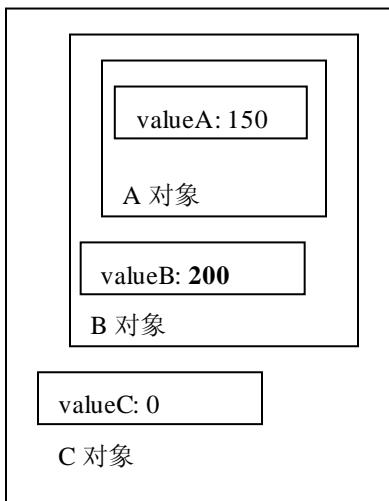
2. 初始化 A 的属性，把 valueA 赋值为 100。如下：



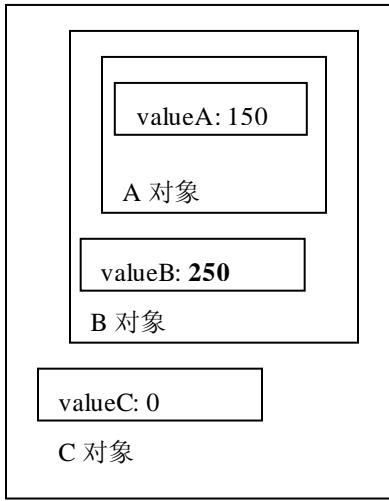
3. 调用 A 的构造方法，此时，会把 valueA 的值设为 150。如下图：



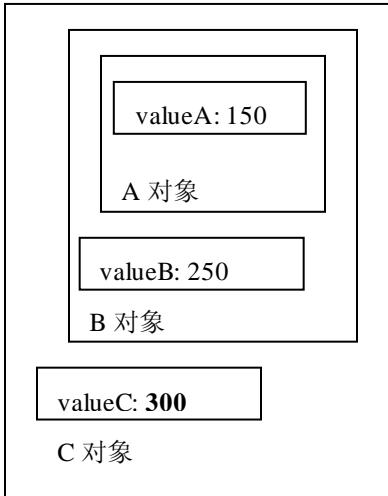
4. 初始化 B 属性：把 valueB 赋值为 200。如下图：



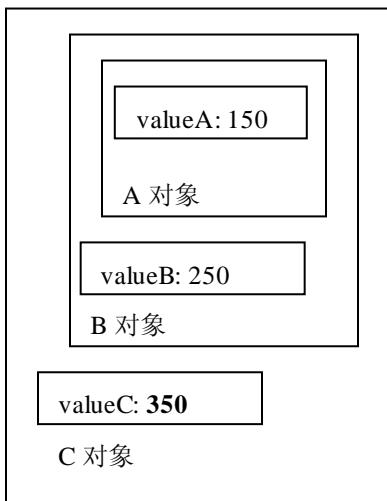
5. 调用 B 的构造方法，此时，会把 valueB 的值设为 250。如下图



6. 初始化 C 的属性，即把 valueC 赋值为 300。如下图：



7. 调用 C 的构造方法，此时，会把 valueC 的值设为 350。如下图



这就是一个完整的创建对象的过程。

## 2.7 super 关键字

在介绍对象创建的过程时，我们介绍了在创建对象时，会创建该对象的父类对象。而在创建父类对象的时候，很显然会调用父类的构造方法。但是，如果父类有多个构造方法，会调用哪一个呢？

例如下面的代码：

```

class Parent{
    public Parent(){
        System.out.println("Parent()");
    }

    public Parent(String str){
        System.out.println("Parent(String)");
    }
}

class Child extends Parent{
    public Child(String str){
        System.out.println("Child(String)");
    }
}

public class TestInheritConstructor{
    public static void main(String args[]){
        Child c = new Child("Hello");
    }
}

```

在上面的代码中，我们创建了一个 `Child` 类型的对象。在创建这个对象的时候，会首先创建一个 `Parent` 对象，并且调用 `Parent` 类中某一个构造方法。`Parent` 类中定义了一个无参的构造方法，也定义了一个带字符串参数的构造方法。那么，我们在调用父类的构造方法时，

会调用哪一个构造方法呢？

运行上面的代码，得到结果如下：

```
D:\Book\chp7>javac TestInheritConstructor.java  
D:\Book\chp7>java TestInheritConstructor  
Parent()  
Child<String>  
D:\Book\chp7>
```

可以看到，在创建 Child 对象时，首先会创建一个 Parent 对象，并且调用 Parent 对象的无参构造方法。也就是说，在默认情况下，创建子类对象时，都会调用父类的无参构造方法。

但是，在父类中我们定义好了带字符串参数的构造方法，能不能要求 Java 在创建 Parent 对象时，调用父类带字符串参数的构造方法呢？

为了解决这个问题，我们为大家介绍 super 关键字。super 关键字有两种不同的用法。

#### super关键字用法一：super 用在构造方法上

super 关键字的第一种用法，就是可以指定在递归构造父类对象的时候，调用父类的哪一个构造方法。

例如，我们要指定，在创建 Child 对象时，会自动创建 Child 的父类对象：Parent 对象，在这时，我们希望能够调用 Parent 类中带字符串参数的构造方法。这时，我们就可以在 Child 类的构造方法中，加上一个语句：super(str)。修改后的 Child 类代码如下：

```
class Child extends Parent{  
    public Child(String str){  
        super(str);  
        System.out.println("Child(String)");  
    }  
}
```

可以看到，在 Child 的构造方法中，我们用 super(str)，来指定要调用父类哪一个构造方法：我们在 super 后面的圆括号中传入了一个字符串参数，这就意味着我们希望调用父类中带字符串参数的构造方法。

修改后的代码运行结果如下：

```
D:\Book\chp7>javac TestInheritConstructor.java  
D:\Book\chp7>java TestInheritConstructor  
Parent<String>  
Child<String>  
D:\Book\chp7>
```

需要注意的是，虽然 super()是写在子类的构造方法中，但是这并不意味着能够在子类中调用父类的构造方法。我们在这里写 super()，是提示 Java，在创建父类对象时调用哪一个构造方法。编译器在进行编译的时候，就会记住我们的这个提示。这样在创建一个子类对象的时候，当执行到“调用父类构造方法”那一个步骤时，就会根据我们之前写的 super，来选择调用父类中某一个构造方法。

要格外注意的是，`super` 用在构造方法中时，只能作为构造方法的第一句。例如，上面的 Child 代码，如果不把 `super` 作为构造方法的第一个语句时，就会产生一个编译错误。代码如下：

```
class Child extends Parent{
    public Child(String str){
        System.out.println("Child(String)");
        super(str);
    }
}
```

编译时的出错信息如下：

```
D:\Book\chp7>javac TestInheritConstructor.java
TestInheritConstructor.java:14: call to super must be first statement in constructor
        super(str);
               ^
1 error

D:\Book\chp7>
```

然而，我们曾经介绍过，`this` 关键字可以在构造方法中，指明调用本类的其他构造方法。并且，对 `this()` 来说，这个语句也只能作为构造方法的第一个语句。

这样，在构造方法中，就不能够既使用 `this()`，又使用 `super()`。例如下面的代码：

```
class Child extends Parent{
    public Child(int n){
        System.out.println("Child(int)");
    }
    public Child(String str){
        this(10);
        super(str);
        System.out.println("Child(String)");
    }
}
```

这段代码编译的结果如下：

```
D:\Book\chp7>javac TestInheritConstructor.java
TestInheritConstructor.java:17: call to super must be first statement in constructor
        super(str);
               ^
1 error

D:\Book\chp7>_
```

编译器提示，对 `super` 的调用必须是构造方法中的第一个语句。那如果把 `super` 放到 `this()` 的前面呢？例如下面的代码：

```
public Child(String str){
    super(str);
    this(10);
    System.out.println("Child(String)");
}
```

此时编译代码，结果如下：

```
D:\Book\chp7>javac TestInheritConstructor.java
TestInheritConstructor.java:17: call to this must be first statement in constructor
        this<10>;
                  ^
1 error
D:\Book\chp7>
```

编译器提示，对 this 的调用必须是构造方法中的第一个语句。

通过上面的例子说明，this()和super()，在构造方法中不能同时使用。

这样，我们构造方法的第一个语句，就有了三种可能

1. super(参数) 指明调用父类哪个构造方法
2. this(参数) 指明调用本类哪个构造方法
3. 既不是 this(参数) 又不是 super(参数)。

在第3种情况下，编译器就会自动加上一句“super()”，即调用父类的无参构造方法。

例如，在一开始的例子中，Child类代码如下：

```
class Child extends Parent{
    public Child(String str){
        System.out.println("Child(String)");
    }
}
```

此时，在Child类的构造方法中，第一句既不是this(参数)，也不是super(参数)，因此，编译器会自动在这个构造方法中增加一个语句：“super();”，代码如下：

```
class Child extends Parent{
    public Child(String str){
        super(); //这句代码是系统默认添加的
        System.out.println("Child(String)");
    }
}
```

这也解释了，为什么在默认情况下，创建父类对象时会调用父类的无参构造方法。

掌握了在构造方法中如何使用super关键字之后，我们可以看下面这个例子。

有如下代码：

```
class Super{
}

class Sub extends Super{
    public Sub(){}
    public Sub(String str){
        super(str);
    }
}
```

上面的代码，应当如何修改才能编译通过？

首先，我们可以编译上述代码。出错信息如下：

```
D:\Book\chp7>javac TestSuper.java
TestSuper.java:7: cannot find symbol
symbol  : constructor Super<java.lang.String>
location: class Super
super(str);
^
1 error

D:\Book\chp7>
```

编译器提示，找不到一个 Super 类中，带字符串参数的构造方法。我们在 Sub 类中的构造方法里，使用 super(str)这个语句来调用了 Super 类中带字符串参数的构造方法。而 Super 类中显然缺少这个构造方法。

接下来，我们为 Super 类增加一个构造方法，代码如下：

```
class Super{
    public Super(String str){}
}

class Sub extends Super{
    public Sub(){}
    public Sub(String str){
        super(str);
    }
}
```

此时，再编译代码，依然出错。出错信息如下：

```
D:\Book\chp7>javac TestSuper.java
TestSuper.java:6: cannot find symbol
symbol  : constructor Super<>
location: class Super
public Sub<>()
^
1 error

D:\Book\chp7>
```

此时，编译器显示，找不到 Super 类的无参构造方法。

我们在什么地方调用了 Super 类的无参构造方法呢？注意，在 Sub 类的构造方法中：

```
public Sub(){}

```

这个构造方法中，没有写任何语句，因此，第一句既不是 this 又不是 super，此时编译器会自动加上一句：“super();”。因此，在 Sub 类的这个构造方法中，调用了 Super 类的无参构造方法。

既然在这儿调用了 Super 类的无参构造方法，那为什么第一次编译没有提示缺少无参构造方法呢？

要注意，当我们把 Super 类写成如下形式时：

```
class Super{}
```

此时，在 Super 类中没有定义任何构造方法，因此，编译器就会自动生成一个公开的、无参的空构造方法。因此，在 Sub 类中调用父类的无参构造方法时，能够找到这个无参的

构造方法，编译能够通过。

而当我们为 Super 类增加了一个字符串参数的构造方法后，编译器就不会自动生成这个无参构造方法，因此就会产生找不到 Super 类无参构造方法的错误。

因此，如果想要让代码编译通过，必须为 Super 类增加两个构造方法。代码如下：

```
class Super{  
    public Super(String str){}  
    public Super(){  
}  
  
class Sub extends Super{  
    public Sub(){  
    public Sub(String str){  
        super(str);  
    }  
}
```

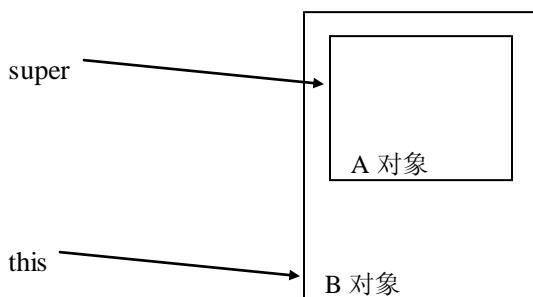
### super 关键字用法二：super 用作引用

这种用法，

super 关键字的第二种用法，就是把 super 当做一个引用，这个引用指向父类对象。例如，有如下代码：

```
class A{  
}  
  
class B extends A{  
}
```

则创建一个 B 对象时，B 对象内部，会包含一个父类对象：A 对象。同时，B 对象内部会有两个引用：this 和 super。其中 this 引用指向当前的 B 对象，而 super，则指向 B 对象内部的 A 对象。内存中示意图如下：



这样，super 引用就可以使用 A 父类对象中的属性和方法。最典型的用途是，使用 super 在子类中，调用父类被覆盖的方法。例如：

```
class Parent{  
    public void m(){  
        System.out.println("m in Parent");  
    }  
}  
class Child extends Parent{
```

```

public void m () {
    System.out.println("m in Child");
}
public void m1 (){
    this.m();
}
public void m2 (){
    super.m();
}
}

public class TestSuper{
    public static void main(String args[]){
        Child c = new Child();
        c.m1();
        c.m2();
    }
}

```

子类 Child 覆盖了父类 Parent 中的 m 方法。在子类的 m1 方法中，调用了 this.m()方法。由于 this 指向当前对象，因此这样，调用的是 Child 类中定义的 m 方法。当然，如果直接写 m()而省略 this 的话，调用的同样是当前对象的 m 方法。

而 Child 类中 m2 方法，则利用 super 关键字，调用了 Parent 类中，被覆盖之前的 m() 方法。运行结果如下：

```

D:\Book\chp7>java TestSuper
m in Child
m in Parent

D:\Book\chp7>_

```

super 可以调用父类被覆盖的方法，这种特性在实际编程中有着广泛的应用。例如，有一个 Server 类，这个类代表一个服务器。其中，有一个 startService()方法，这个方法的代码非常非常复杂，并且写的也相对比较成熟。遗憾的是，这个代码有一个缺点：无法记录日志文件。代码大概是这个样子：

```

class Server{
    public void startService(){
        //很长的代码
        //很复杂的逻辑
        //但是没有保存日志
    }
}

```

如果我们直接在 Server 类上进行修改的话，就会在 startService 这个复杂的方法中增加更多的复杂性，也降低了代码的可读性以及可重用性。

为此，我们可以创建一个新的类：MyServer，让这个类继承自 Server 类，并且在 MyServer 中，覆盖 Server 类的 startService()方法。

但是，如果让 MyServer 从头再把 startService 方法写一遍，则完全没必要。我们可以在

MyServer 的 startService 方法内部，利用 super 关键字，调用父类中的 startService()方法。然后，在根据需求，添加上保存日志的逻辑。代码示意如下：

```
class MyServer extends Server{
    public void startService(){
        super.startService();
        //保存日志的功能
    }
}
```

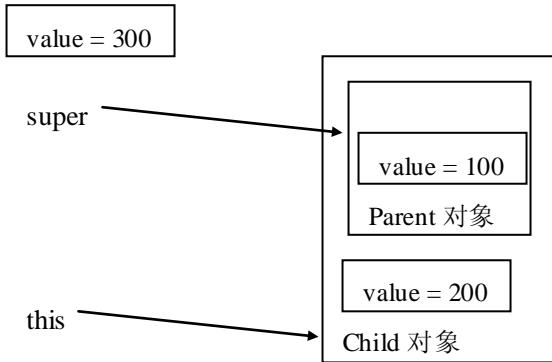
因此，在实际开发中，如果要增强一个类中某一个方法的功能，可以继承这个类，然后覆盖这个类的方法。在覆盖时，可以利用 super 关键字调用父类中的实现，然后在父类实现的基础上，增加子类特有的新功能。

super 关键字除了可以用来调用方法之外，也可以用来指向属性。例如下面的代码：

```
class Parent{
    int value = 100;
}

class Child extends Parent{
    int value = 200;
    public void print(){
        int value = 300;
        //...
    }
}
```

在这个代码中，我们定义了两个类：一个类是 Parent，一个类是 Child。这两个类中，都包含一个 value 属性，一个值为 100，另一个值为 200。此外，在 print 方法中，还定义了一个局部变量 value，值为 300。内存中的示意图如下：



此时，在内存中有三个 value 变量。如果在 print 方法中直接打印 value 变量，则由于局部变量优先，会打印 300 的值。如果想要打印 Child 中的 value 属性，输出 200，则应当使用 this.value。类似的，如果想要打印 Parent 中的 value 属性，则应当使用 super.value。

完整代码如下：

```
class Parent{
    int value = 100;
}

class Child extends Parent{
```

```
int value = 200;
public void print(){
    int value = 300;
    System.out.println(value);
    System.out.println(this.value);
    System.out.println(super.value);
}
}

public class TestParent{
    public static void main(String args[]){
        Child c = new Child();
        c.print();
    }
}
```

输出结果如下：

```
D:\Book\chp7>javac TestParent.java
D:\Book\chp7>java TestParent
300
200
100
D:\Book\chp7>
```

## 2.8 单继承

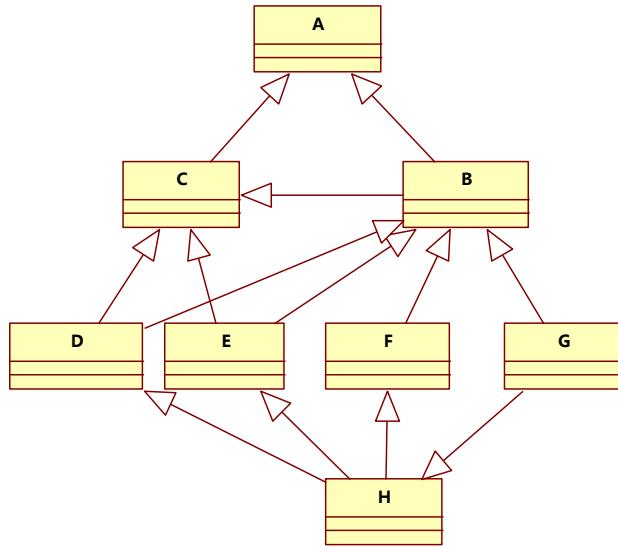
Java 语言规定，每一个类只能有一个直接父类。这就是 Java 语言的“单继承”规则。也就是说，我们不能试图让一个类去同时继承两个以上的类。如下面的代码是错误的：

```
class A{}
class B{}
class C extends A,B{} // 编译出错，一个类不能同时有两个直接父类
```

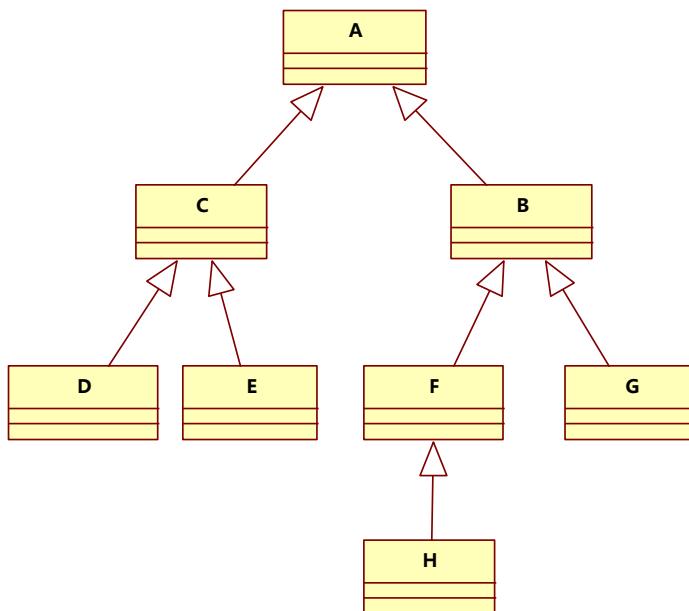
对于 C 类来说，它要么继承 A 类，要么继承 B 类，不能同时继承这两个类。这就是 Java 中的“单继承”。

我们知道，Java 语言的设计极大程度上借鉴了 C++ 语言。但是这两种语言在继承关系上有着重大的区别。C++ 语言是允许“多继承”的，也就是说，完全可以让一个类同时去继承多个类。

对于多继承来说，一个典型的类继承关系如下图：



而对于单继承来说，类继承关系如下图所示：



很显然，Java 中的类关系比较简单。这是因为，在多继承关系下，类之间会形成错综复杂的网状结构。而相比之下，由于单继承的原因，Java 中的类关系只会形成树状结构。树状结构自然会比网状结构要简单的多。

因此，Java 语言中单继承的特性，被认为是 Java 语言相对于 C++ 语言，“简单性”的一个重要体现。

### 3 多态

多态是面向对象三大特性中，最为重要也是最为灵活的一个特性。多态部分的讨论，是基于以下的代码：

```
class Animal{
```

```

public void eat(){
    System.out.println("Animal eat");
}
public void sleep(){
    System.out.println("sleep 8 hours");
}
}
class Dog extends Animal{
    public void sleep(){
        System.out.println("sleep 6 hours");
    }
    public void lookAfterHouse(){
        System.out.println("Dog can look after house");
    }
}
class Cat extends Animal{
    public void catchMouse(){
        System.out.println("Cat can catch mouse");
    }
}

```

这段代码在“继承”部分中曾经分析过，Animal 类代表一般的动物，具有“eat”和“sleep”两种行为，而 Dog 类代表狗，作为特殊的动物继承 Animal 类，除了添加自身特有的“lookAfterHouse”方法之外，狗类还覆盖了“sleep”方法。Cat 类代表猫，也是 Animal 类的子类。除了从 Animal 类中继承两个方法之外，猫类还添加了“catchMouse”方法。

### 3.1 引用类型和对象类型

我们来看下面的代码：

```

01 : Animal a ;
02: a = new Animal();

```

在这两行代码中，分别出现了“Animal”字样。但是含义却不相同：

第一行代码中，我们定义了一个引用 a，而约束这个引用的类型为 Animal。我们知道，Java 语言是一个强型的语言，在定义变量的时候，必须指定变量的类型。变量类型和变量中存放的数据类型必须匹配。这样，a 引用中只能存放 Animal 类型的对象。在这里，我们称“a 引用的引用类型为 Animal”。

第二行代码中，我们创建了一个 Animal 类型的对象，将这个对象的地址赋给 a 引用。每当我们创建对象时，总要指定这个对象的类型。对象的类型我们会写在“new”关键字的后面。在这里，我们称“a 引用所指向的对象类型为 Animal”。

也就是说，我们在定义引用的时候，为引用指定“引用类型”。而将对象放入引用的时候，引用中的对象还有一个“对象类型”。

以我们目前的知识程度，我们可以认为：引用类型和对象类型必须是一致的，这是 Java 语言强类型的约束。例如，下面的代码是正确的：

```
Dog d = new Dog();
```

d 引用的引用类型为 Dog，就必须存放对象类型为 Dog 类的对象。

而下面的代码是错误的：

```
Dog d = new Cat();
```

d 引用中只能存放 Dog 类的对象，而不能存放 Cat 类的对象。

而利用多态，下面的代码也是正确的：

```
Animal d = new Dog();
```

这是为什么呢？我们知道，Dog 类是 Animal 的子类。也就是说，Dog 类是特殊的 Animal 类。进而我们可以认为，一个 Dog 类的对象就是一个特殊的 Animal 类的对象。只要两个类之间存在继承关系，子类的对象就一定可以看作是特殊的父类对象。例如：汽车对象是特殊的交通工具对象；桌子对象是特殊的家具对象等等。

在上述代码中，d 引用是 Animal 类型的，只能存放 Animal 对象。而 Dog 对象也可以看作是特殊的 Animal 对象。那么一个 Dog 对象，当然就可以放入 Animal 类型的引用中。这并不违反 Java 语言强类型的限制。

因此我们可以得出结论：**子类的对象可以放入父类的引用中！**

也就是说，**一个引用的引用类型和对象类型未必完全一致，对象类型可以是引用类型的子类。**当我们看到一个引用时，一方面要考察这个引用的引用类型，另一方面还要考察这个引用中所存储的对象的类型，这两个类型可能是不同的。

## 3.2 多态的语法规特性

在上一章节中，我们介绍了，子类的对象可以存放在父类的引用中，这是多态语法的根本出发点。

我们设想这样一个场景：小强是一个 2 岁的小朋友，在他幼小的意识中，从来没有见过“狗”这种东西。但是，他非常明确什么是“动物”。也知道动物会“吃”，会“睡”。因此，当他第一次见到一只狗的时候，他奶声奶气的问妈妈“这是什么动物？”。也就是说，小强认为，出现在他眼前的是一个动物类的对象。

我们以前提到过，类是人对客观对象的认识。很显然，在小强的头脑中，已经建立起了“动物”类的概念，但是却没有建立起“狗”类的概念。因此，当他面对一个狗类的对象时，他把它当做是一个动物类的对象。这并没有错，因为狗对象确实可以看作是动物对象。

我们可以认为，小强的这种思维过程可以用下面的代码来描述：

```
Animal a = new Dog();
```

我们可以这样理解，引用类型代表着一种“主观类型”。把狗类的对象放入动物类型的引用中，就意味着，主观上，把一个狗对象看作是一个动物对象。而这事实上就是多态的典型应用。

我们结合这个例子来具体分析多态的语法。总结起来，多态的语法主要体现在以下几点：

### 1. 对象类型永远不变

一个对象在创建的时候，其对象类型就已经决定了。直到这个对象消亡，它的对象类型永远不会改变。在多态的语法中，一个对象可能存放在不同类型的引用中，但是请注意，对象自身的类型从来也不会发生变化。例如在小强的眼中，这个狗对象被当作了一个动物对象，但这个对象实际上还是一只狗，这是不会变化的。

这一点很好理解，一个对象，它是狗就是狗，是猫就是猫，如果你认为它是狗它就是狗，你认为它是猫它就是猫，这显然是不合理，不“唯物”的。

### 4 只能对一个引用调用其引用类型中定义的方法

在小强的眼中，这个对象是一个动物对象，而他知道，动物会“吃饭”，会“睡觉”。也

就是说，在动物类中定义了 eat 方法和 sleep 方法。那么小强自然可以对这个对象调用 eat 和 sleep 方法。

但是，出现在小强眼前的对象实际上是一个狗对象，狗类中不仅有 eat 方法和 sleep 方法，还有一个 lookAfterHouse 方法。也就是说，狗还会看家护院。那么小强能否调用这个方法呢？显然不行，尽管这个对象确实具有这个方法，但是小强并不清楚这一点。

在实际生活中，我们能够对一个对象调用什么方法，这并不取决于这个对象具有什么方法，而主要取决于，我们“知道”这个对象具有什么方法。例如，同样一个手机对象，在有些人看来，这只是一个“电话”对象，因此，这些人只会调用该对象的“打电话”，“接电话”方法。而对于另外一些人来说，可能还会调用手机的“拍照”，“玩游戏”，“上网”等方法。其实这些功能也是手机对象所具备的。

我们来看以下的代码：

```
public class TestPolymorphism {  
    public static void main(String[] args) {  
        Animal a = new Dog(); //将一个狗对象看作是一个动物对象  
        a.eat(); // 正确  
        a.sleep(); // 正确  
        a.lookAfterHouse(); // 编译错误  
    }  
}
```

当小强把 Dog 对象放入 Animal 类型的引用时，只能对这个引用调用 eat 方法，sleep 方法，而当他试图调用 lookAfterHouse 方法时，将会得到一个编译错误。因为他把狗对象看作是一个动物对象，他并不了解这个对象还具有 lookAfterHouse 方法。

也就是说，当我们对一个引用调用方法时，只能调用这个引用的引用类型中定义的方法。例如在上述代码中，a 引用的引用类型为 Animal，而 Animal 类中定义了 eat 方法和 sleep 方法，因此我们可以对 a 引用调用这两个方法。而由于 Animal 类中没有定义 lookAfterHouse 方法，我们就无法对 a 引用调用这个方法。

## 5 运行时，根据对象类型调用子类覆盖之后的方法

通过代码，我们看到：Animal 类中定义了 sleep 方法，打印“sleep 8 hours”。而 Dog 类作为 Animal 的子类，在 sleep 方法的实现方式上有自己独特的实现，因此，Dog 类覆盖了 Animal 类中的 sleep 方法，打印“sleep 6 hours”。

我们再来思考那个场景。小强是知道眼前的对象具有“睡觉”方法的，因此，他可以调用这个对象的“睡觉”方法。下面的代码是能够编译通过的：

```
public class TestPolymorphism {  
    public static void main(String[] args) {  
        Animal a = new Dog(); //将一个狗对象看作是一个动物对象  
        a.sleep(); // 对这个对象发出“睡觉”的指令  
    }  
}
```

而运行时，这段代码会打印什么呢？

我们可以这样来理解这个问题，一般的动物一天睡 8 个小时，而狗作为特殊的动物，一天睡 6 个小时。那么，当小强对眼前这个狗对象发出“睡觉”指令的时候，这个对象会睡几个小时呢？当然是 6 个小时！这与小强把这个对象看作狗还是看作动物是无关的，这个对象实际上就是一只狗，它当然会按照自己的方式去睡。

代码运行结果如下：

```
sleep 6 hours
```

从中我们可以得到结论：引用类型和对象类型之间如果存在方法的覆盖，那么在程序运行的时候，JVM 会根据引用中所存储的对象类型，去调用对象类型中覆盖之后的方法。例如：Animal 类型的 a 引用中存放的是 Dog 对象，而 Dog 类覆盖了 Animal 类中的 sleep 方法，则 JVM 会调用 Dog 类中覆盖之后的 sleep 方法，而不是 Animal 类中的 sleep 方法。

我们可以通过下面这个简单的代码来总结一下多态的基本语法特性：

```
class A{  
    public void method(){  
        System.out.println("method in A");  
    }  
}  
  
class B extends A{  
    public void method(){  
        System.out.println("method in B");  
    }  
}
```

我们可以把子类对象放入父类引用中。如：A a = new B();

由于 A 类中定义了 method 方法，因此我们可以对 a 引用调用：a.method();

而 a 引用中实际存放的是 B 类对象，因此运行时调用 B 类覆盖之后的 method 方法，将打印“method in B”。

### 3.3 强制类型转换和 instanceof

我们前面分析过，对一个引用，只能调用引用类型中定义的方法。因此，下面的代码是错误的：

```
Animal a = new Dog();  
a.lookAfterHouse();
```

那么如果我们希望调用 lookAfterHouse 这个方法，应该怎么做呢？很显然，由于这个方法定义在 Dog 类中，我们必须采用 Dog 类型的引用来调用这个方法。如：

```
Animal a = new Dog();  
Dog d = a;  
d.lookAfterHouse();
```

在这里，a 引用和 d 引用共同指向了一个 Dog 对象。不一样的是，d 引用的引用类型为 Dog，因此，使用 d 引用可以调用 lookAfterHouse 方法。

但是很遗憾，代码“Dog d = a”是错误的。

d 引用的类型为 Dog，因此 d 引用中必须存放一个 Dog 类的对象（或者是 Dog 类的子类的对象），但是 a 引用的类型为 Animal，a 引用中可能存放 Dog 类的对象，也可能存放 Animal 类其他子类的对象（比如 Cat）。如果 a 引用中存放了 Cat 类的对象，我们将 a 引用赋值给 d 引用，就错误的把一个 Cat 对象装在了 Dog 类型的引用中，这显然违反了 Java 语言强类型的约束。因此，代码“Dog d = a”在编译时，编译器会报错。

为了解决这个问题，我们必须把这句代码改写为：

```
Dog d = (Dog) a;
```

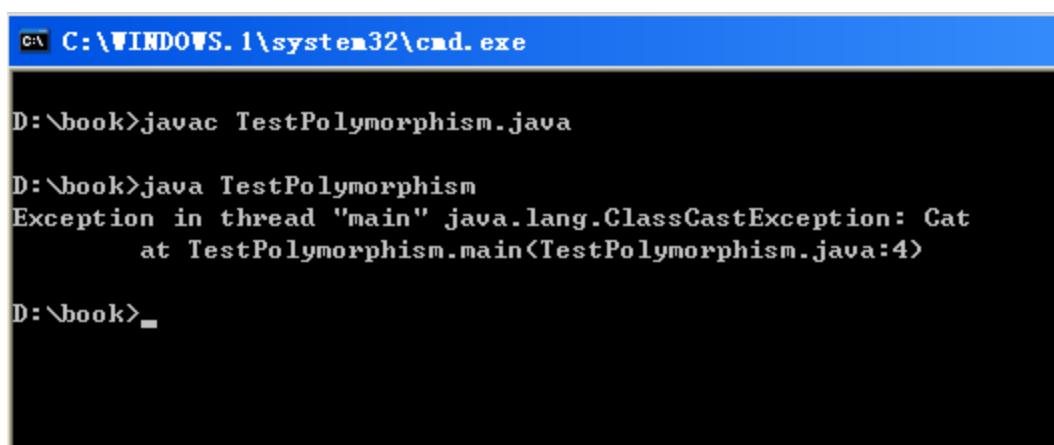
这个语法我们似曾相识，在介绍基本类型的时候，我们用过类似的语法：强制类型转换。

在这里，强制类型转换的含义是：我们认为，`a` 引用中存放的就是一个 `Dog` 类的对象，我们“强制性的要求”系统把 `a` 引用中的对象，作为 `Dog` 对象存放在 `d` 引用中。

由于我们表明了这样一种“强硬”的态度，编译器在编译的时候，就不会对这句话报出任何错误了。但是请注意，这并不意味着这句话一定是对的。因为 `a` 引用中依然可能存放 `Cat` 对象，一旦这种情况发生，虽然编译通过，但是运行时，JVM 会抛出一个 `ClassCastException`，类型转换异常。代码如下：

```
public class TestPolymorphism {  
    public static void main(String[] args) {  
        Animal a = new Cat();  
        Dog d = (Dog) a;  
        d.lookAfterHouse();  
    }  
}
```

该代码运行时，结果如下：



```
C:\WINDOWS.1\system32\cmd.exe  
  
D:\book>javac TestPolymorphism.java  
  
D:\book>java TestPolymorphism  
Exception in thread "main" java.lang.ClassCastException: Cat  
        at TestPolymorphism.main<(TestPolymorphism.java:4>  
  
D:\book>
```

我们可以看到，尽管编译正确，但是运行时系统抛出了 `ClassCastException`。

因此我们可以得出结论：

1. 子类的引用可以直接赋值给父类引用。（这是多态的基本用法）
2. 父类的引用赋值给子类引用，必须强制类型转换，并有可能在运行时得到一个类型转换异常。

那么，如何避免类型转换异常的发生呢？下面我们介绍 Java 中的一个关键字：`instanceof`。

`instanceof` 的基本语法如下：

引用 `instanceof` 类名

`instanceof` 是一个二元运算符，用来组成一个布尔表达式。用来判断某个引用所指向的对象是否和某个类型兼容。例如，我们假定：`a` 引用中存放的是 `Cat` 类型的对象。即：

```
Animal a = new Cat();
```

那么表达式“`a instanceof Cat`”的值为“true”，因为 `a` 引用所指向的对象确实为 `Cat` 类型的对象。显然，表达式“`a instanceof Dog`”的值为“false”。而表达式“`a instanceof Animal`”也会为“true”。因为 `a` 引用所指向的对象也可以认为是一个 `Animal` 类的对象。

完整的代码如下：

```
public class TestInstanceof {
```

```
public static void main(String[] args) {  
    Animal a = new Cat();  
    System.out.println(a instanceof Cat);  
    System.out.println(a instanceof Dog);  
    System.out.println(a instanceof Animal);  
}  
}
```

运行结果为：

```
true  
false  
true
```

我们可以把“instanceof”理解为“是不是”三个字，这样可以帮助我们方便的记忆这个关键字的用法。例如：“a instanceof Dog”就可以理解为“a引用所指向的对象是不是狗类的对象”。

由于 instanceof 关键字可以用来判断一个引用中的对象类型，因此，我们在对一个引用进行强制类型转换之前，就可以用这个关键字先进行判断，从而避免类型转换异常的发生。

例如以下代码：

```
public class TestPolymorphism {  
    public static void main(String[] args) {  
        Animal a = new Cat();  
        if (a instanceof Dog) {  
            Dog d = (Dog) a;  
            d.lookAfterHouse();  
        }  
    }  
}
```

由于我们在对 a 引用进行强制类型转换之前，用 instanceof 进行了判断，如果 a 引用中存放的不是 Dog 类型的对象，则不会运行强制类型转换的代码，因此，就避免了类型转换异常的发生。

### 3.4 多态的作用

多态最主要的作用在于：我们可以将若干不同子类的对象都当做统一的父类对象来使用，这样就会提高程序的通用性，屏蔽不同子类之间的差异。

我们先来看第一种情形：我们希望开发一个 Feeder 类，表示一个饲养员，饲养员应该具有一个“喂养动物”的方法。动物园中的所有动物都由这个饲养员来喂养。代码如下：

```
class TestPolymorphism{  
    public static void main(String[] args){  
        Feeder f = new Feeder();  
  
        Dog d = new Dog();  
        f.feed(d);  
  
        Cat c = new Cat();  
        f.feed(c);  
    }  
}
```

```

    }
}

class Feeder{
    public void feed(Dog d) {
        d.eat();
    }
    public void feed(Cat c) {
        c.eat();
    }
}

```

我们看到，在 Feeder 类中有两个重载的 feed 方法，分别以 Dog 和 Cat 作为参数类型，表示喂养狗和喂养猫。那么如果 Animal 类不止 2 个子类呢？假设动物园中有 1000 种动物，Animal 类有 1000 个子类，是不是意味着，我们要为 Feeder 类重载 1000 个 feed 方法呢？

其实没有必要这样，因为无论是狗还是猫，对于饲养员来说都是一样的，都是“动物”，因此，我们在设计 feed 方法参数类型的时候，完全可以采用父类 Animal 类作为形参类型。代码如下：

```

class TestPolymorphism{
    public static void main(String[] args) {
        Feeder f = new Feeder();

        Dog d = new Dog();
        f.feed(d);

        Cat c = new Cat();
        f.feed(c);
    }
}

class Feeder{
    public void feed(Animal a) {
        a.eat();
    }
}

```

我们将 feed 方法的参数类型设计为“Animal”，这就意味着，该方法接受一个 Animal 对象作为实参。而 Animal 的任何一个子类对象都可以看作是一个 Animal 对象。因此，无论以 Dog 对象作为参数，还是以 Cat 对象作为参数，都可以调用同一个 feed 方法。这样的一个 feed 方法，就可以为成千上万种动物“服务”了。

我们把这种用法称为“**把多态用在方法的参数类型上**”。我们在定义方法时，可以把形参定义为父类类型的引用，而调用方法时，完全可以把子类类型的对象作为实参。很显然，形参为父类类型的方法更加通用，能够接受更多种不同子类类型的实参对象。

作为初学者，我们应该牢牢记住，形如：

`public void m (A a )`

这样的方法，完全可以用 A 类对象或是 A 类的某个子类对象作为参数来调用。

我们再来看第二种情形：我们希望写一个方法，这个方法能够返回一个 Dog 对象。代

码如下：

```
public Dog getAnimal(){  
    return new Dog();  
}
```

这个方法非常简单。但是，如果我们的需求发生了改变，希望这个方法接受一个 int 类型的参数，当这个参数为偶数时返回 Dog 对象，为奇数时返回 Cat 对象。我们会遇到一个困难：

```
public ??? getAnimal(int i){  
    if (i % 2 == 0 ) {  
        return new Dog();  
    }  
    else {  
        return new Cat();  
    }  
}
```

看到了？我们无法定义这个方法的返回值类型。因为这个方法可能返回 Dog 对象，也可能返回 Cat 对象。

聪明的读者一定想到了，代码完全可以写成这个样子：

```
public Animal getAnimal(int i){  
    if (i % 2 == 0 ) {  
        return new Dog();  
    }  
    else {  
        return new Cat();  
    }  
}
```

这个方法可能返回 Dog 对象，也可能返回 Cat 对象，但返回的一定是个 Animal 对象。因此，我们可以用 Animal 类型作为这个方法的返回值类型。也就是说，我们在方法声明中“承诺”方法会返回一个 Animal 对象，在方法的实现中，完全可以将 Animal 的某个子类对象作为返回值返回。这并不违反我们在声明中的“承诺”。

我们把这种用法称为“把多态用在方法的返回值类型上”，很显然，我们把返回值类型定义为父类，会使得方法更加的通用。在方法的实现中，我们可以返回任何一个子类的对象。

我们应该牢牢记住，形如：

```
public A m ()
```

这样的方法，可能返回 A 类的对象，也完全可能返回 A 类的某个子类的对象。

我们来看一个完整的代码：

```
class TestPolymorphism{  
    public static void main(String[] args) {  
        B b = new B();  
        m1(b);  
  
        A a = m2(5);  
        a.method();  
    }  
}
```

```

    }
    static void m1(A a){
        a.method();
    }
    static A m2(int i){
        if (i % 2 == 0){
            return new B();
        }
        else {
            return new C();
        }
    }
}

class A{
    public void method(){
        System.out.println("method in A");
    }
}

class B extends A{
    public void method(){
        System.out.println("method in B");
    }
}

class C extends A{
    public void method(){
        System.out.println("method in C");
    }
}

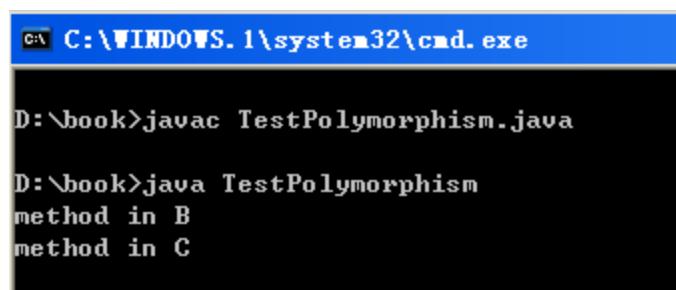
```

这段代码运行结果是什么？

对于“m1(b);”，我们以一个 B 对象作为参数调用 m1 方法，当实参传给形参时，B 对象就会被赋值给 m1 方法的形参 a。也就是说，a 引用中存放的是 B 类的对象。那么对 a 引用调用 method 方法，自然会调用 B 类覆盖之后的 method 方法，因此打印“method in B”。

对于“A a = m2(5);”，我们用 5 作为参数调用 m2 方法，根据代码，m2 方法会返回一个 C 对象。也就是说，a 引用中存放的是 C 类的对象。此时对 a 调用 method 方法，会调用 C 类覆盖之后的 method 方法，因此打印出“method in C”。

代码运行结果如下：



```

C:\WINDOWS.1\system32\cmd.exe
D:\book>javac TestPolymorphism.java
D:\book>java TestPolymorphism
method in B
method in C

```

在面向对象的三大特性中，多态最为灵活，不易理解和掌握。请读者勤加练习，多加揣摩。在编程实践中逐渐领悟多态的语法和作用。

# Chp6 三个修饰符

## 本章导读

本章将向读者介绍 Java 中的三个非常重要的修饰符：static、final 和 abstract。

学习修饰符，应该始终明确：该修饰符能够修饰什么程序组件，而修饰某个组件的时候，又表示了什么含义。

### 1 static

static 修饰符也被称为静态修饰符。这个修饰符能够修饰三种程序组件：属性、方法、初始化代码块。static 在修饰这三个不同的组件的时候，分别表示不同的含义。

需要注意的是，static 不能修饰局部变量和类。

下面根据修饰的组件不同，我们分别阐述 static 修饰属性、方法以及初始化代码块所代表的含义。

#### 1.1 静态属性

static 修饰属性，则该属性就成为静态属性。**静态属性是全类公有的属性。**例如，有如下代码：

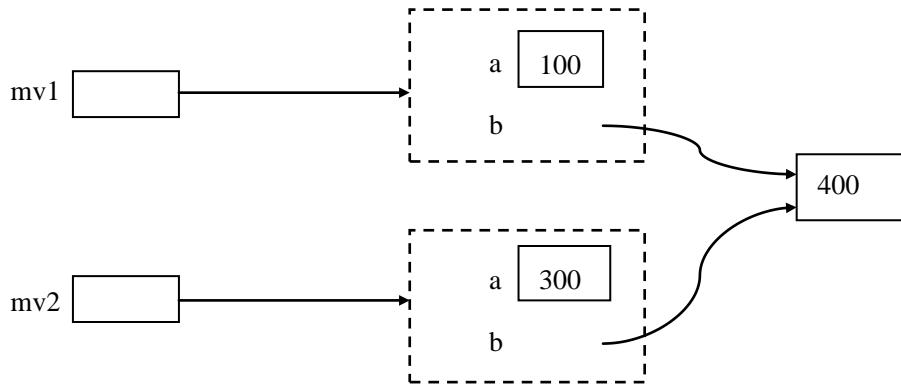
```
class MyValue{  
    int a;  
    static int b;  
}  
public class TestStatic{  
    public static void main(String args[]){  
        MyValue mv1 = new MyValue();  
        MyValue mv2 = new MyValue();  
        mv1.a = 100;  
        mv1.b = 200;  
        mv2.a = 300;  
        mv2.b = 400;  
        System.out.println(mv1.a);  
        System.out.println(mv1.b);  
        System.out.println(mv2.a);  
        System.out.println(mv2.b);  
    }  
}
```

编译运行之后，输出结果为：

```
100  
400  
300  
400
```

要注意，400 这个数字出现了两次，200 没有出现。

原因在于：b 属性是一个静态属性。**MyValue** 类的所有对象，都公用一个 b 属性，每一个对象的 b 属性，都指向同一块内存。如下图所示：



可以看到，mv1 对象的 a 属性和 mv2 对象的 a 属性，彼此之间是相互独立的。而 mv1 和 mv2 两个对象的 b 属性，指向的是内存中的同一块区域。因此，当代码执行到 mv1.b = 200；时，把这块区域设置为 200；执行到 mv2.b = 400 时，把内存中的同一块区域设置为 400。这时，无论通过 mv1.b，还是 mv2.b，读取到的都是同一块内存区域的值，因此输出语句输出的都是 400。

**每个对象的静态属性都指向同一块内存区域，事实上，这个属性不属于任何一个特定对象，而属于“类”。因此，可以使用类名直接调用静态属性。例如，可以把上面的 TestStatic 程序改写成下面的样子：**

```
class MyValue{
    int a;
    static int b;
}

public class TestStatic{
    public static void main(String args[]) {
        MyValue mv1 = new MyValue();
        MyValue mv2 = new MyValue();
        mv1.a = 100;
    MyValue.b = 200;
        mv2.a = 300;
    MyValue.b = 400;
        System.out.println(mv1.a);
    System.out.println(MyValue.b);
        System.out.println(mv2.a);
    System.out.println(MyValue.b);
    }
}
```

这段代码中，对 b 属性的使用都是用类名直接调用。

**在实际编程过程中，为了避免错误和误会，对静态属性的使用，应当尽量用“类名直接调用”的写法。用这种写法能够把静态属性和实例变量加以区分，从而提高程序的可读性。**

要注意的是，由于静态属性不属于任何一个对象，因此，在一个对象都没有创建的情况下，照样可以使用静态属性。这个时候，就只能用类名直接访问静态属性。例如下面的代

码：

```
class MyValue{  
    int a;  
    static int b;  
}  
  
public class TestStatic{  
    public static void main(String args[]){  
        //没有创建任何 MyValue 对象就直接使用属性  
        MyValue.b = 200;  
        System.out.println(MyValue.b);  
    }  
}
```

从概念上怎么来理解静态属性呢？举一个生活中的例子，例如：在一个班级中，每个学生对象都有一个属性：“Java 老师”。对于同一个班级的同学来说，Java 老师这个属性总是一样的，即使上课过程中更换老师，班级中所有学生对象的“Java 老师”属性都会进行同样的改变。因此，如果将学生看做是一个类，“Java 老师”这个属性，就属于这个类的“公有”属性，也就是静态属性。

最后，介绍一下几个名词。在之前的课程中，我们接触过的“属性”就是指的实例变量。现在，我们接触到了静态属性这个概念，再提“属性”这个概念，就分为静态属性和非静态属性两种。其中，静态属性也可以叫“类变量”，而非静态属性就是我们所说的“实例变量”。这几个名词的关系如下：



## 1.2 静态方法

用 static 修饰的方法称之为静态方法。首先我们看一个代码的例子，这个例子中反映了静态方法与非静态方法分别能访问什么样的属性和方法。

```
class TestStatic{  
    int a = 10;           //非静态属性  
    static int b = 20;     //静态属性  
    public void ma(){}   //非静态方法  
    public static void mb(){} //静态方法  
  
    public void fa(){    //fa 是一个非静态方法  
        System.out.println(a); //非静态方法能够访问非静态属性  
        System.out.println(b); //非静态方法能够访问静态属性  
        ma();   //非静态方法中, 能够调用非静态方法  
        mb();   //非静态方法中, 能够调用静态方法  
    }  
    public static void fb(){ //fb 是一个静态方法  
        System.out.println(a); //编译错误 静态方法中不能访问非静态属性  
        System.out.println(b); //静态方法中可以访问静态属性  
    }  
}
```

```
    ma(); //编译错误，静态方法中调用非静态方法
    mb(); //静态方法中可以调用静态方法
}
}
```

从上面这个例子中，我们可以总结出如下规律：

在非静态方法中，无论方法或属性是否是静态的，都能够访问；

而在静态方法中，只能访问静态属性和方法。

静态方法和静态属性统称为静态成员。以上的规律可以记成：静态方法中只能访问静态成员。需要注明的是，在静态方法中不能使用 this 关键字。

除此之外，静态方法与静态属性一样，也能够用类名直接调用。例如下面的代码：

```
public class TestStaticMethod{
    public static void main(String args[]) {
        TestStatic.fb();
    }
}
```

静态方法是属于全类公有的方法。从概念上来理解，调用静态方法时，并不针对某个特定的对象，这个方法是全类共同的方法。例如，对于班级来说，有一个“办联欢会”的方法。想要办好一个班级联欢会，不能依靠某个特定同学（也就是某个对象）的努力，而应该是全班同学共同配合完成。因此，这个方法不属于某个特定对象，而是“全类公有”的方法。

除了上面所说的特点之外，静态方法还有一个非常重要的特性，这个特性跟方法覆盖有关。看下面的代码：

```
class Super{
    public void m1() {
        System.out.println("m1 in Super");
    }
    public static void m2() {
        System.out.println("m2 in Super");
    }
}

class Sub1 extends Super{
    public void m1(){ //非静态方法覆盖非静态方法，编译通过
        System.out.println("m1 in Sub");
    }
    public static void m2(){ //静态方法覆盖静态方法，编译通过
        System.out.println("m2 in Sub");
    }
}

class Sub2 extends Super{
    public static void m1(){}
    public void m2(){}
}
```

根据上面的例子，可以发现：静态方法只能被静态方法覆盖，非静态方法只能被非静态

方法覆盖。

除此之外，我们再写一个程序来测试一个 Super 类和 Sub1 类。

```
public class TestStaticOverride{  
    public static void main(String args[]) {  
        Super sup = new Sub1();  
        sup.m1();  
        sup.m2();  
        Sub1 sub = (Sub) sup;  
        sub.m1();  
        sub.m2();  
    }  
}
```

编译运行，结果如下：

```
m1 in Sub  
m2 in Super  
m1 in Sub  
m2 in Sub
```

注意到，对于 m1 这个非静态方法来说。无论引用类型是 Super 还是 Sub1，调用 m1 方法，结果都是 m1 in Sub。这是上一章我们所讲的多态特性：运行时会根据对象的实际类型调用子类覆盖以后的方法。

然后，如果是 m2 这个静态方法，情况则大大不同。在引用类型为 Super 时，调用的是 m2 in Super；当引用类型为 Sub1 时，调用的是 m2 in Sub。这是静态方法很重要的一个特性：静态方法没有多态。当我们对一个引用调用静态方法的时候，等同于对这个引用的引用类型调用静态方法；所以代码

```
super.m2(); 相当于 Super.m2();  
sub.m2(); 相当于 Sub.m2();
```

总结一下静态方法的几个性质：

1. 静态方法可以用类名直接调用。
2. 静态方法中只能访问类的静态成员。
3. 静态方法只能被静态方法覆盖，并且没有多态。

### 1.3 静态初始化代码块

static 修饰符修饰初始化代码块，就称之为静态初始化代码块。

首先，简单介绍一下初始化代码块。看如下的代码例子

```
public class MyClass{  
    {  
        //此处为初始化代码块  
    }  
    int a;  
    public MyClass(){  
        System.out.println("MyClass()");  
    }  
}
```

在类的里面，所有方法的外面定义的代码块称之为初始化代码块（如上面的例子所示）。

能够用 static 修饰初始化代码块，使之成为静态初始化代码块。例如：

```
public class MyClass{  
    static {  
        //此处为静态初始化代码块  
        System.out.println("In MyClass Static");  
    }  
    int a;  
    public MyClass() {  
        System.out.println("MyClass()");  
    }  
}
```

这样就定义好了静态初始化代码块。

那么这个代码块什么时候执行呢？静态初始化代码块执行的时机非常特别，下面我们就来介绍这个问题。

### 1.3.1 类加载

首先我们来简单考虑一下下面的代码。

```
//TestStudent.java  
01: class Student{  
02:     static {  
03:         System.out.println("in Student static");  
04:     }  
05:     public Student() {  
06:         System.out.println("Student()");  
07:     }  
08: }  
09: public class TestStudent{  
10:     public static void main(String args[]) {  
11:         Student stu1 = new Student();  
12:         Student stu2 = new Student();  
13:     }  
14: }
```

上面的 TestStudent.java 文件，编译生成两个.class 文件：一个 Student.class，一个 TestStudent.class。

生成这两个.class 文件之后，执行

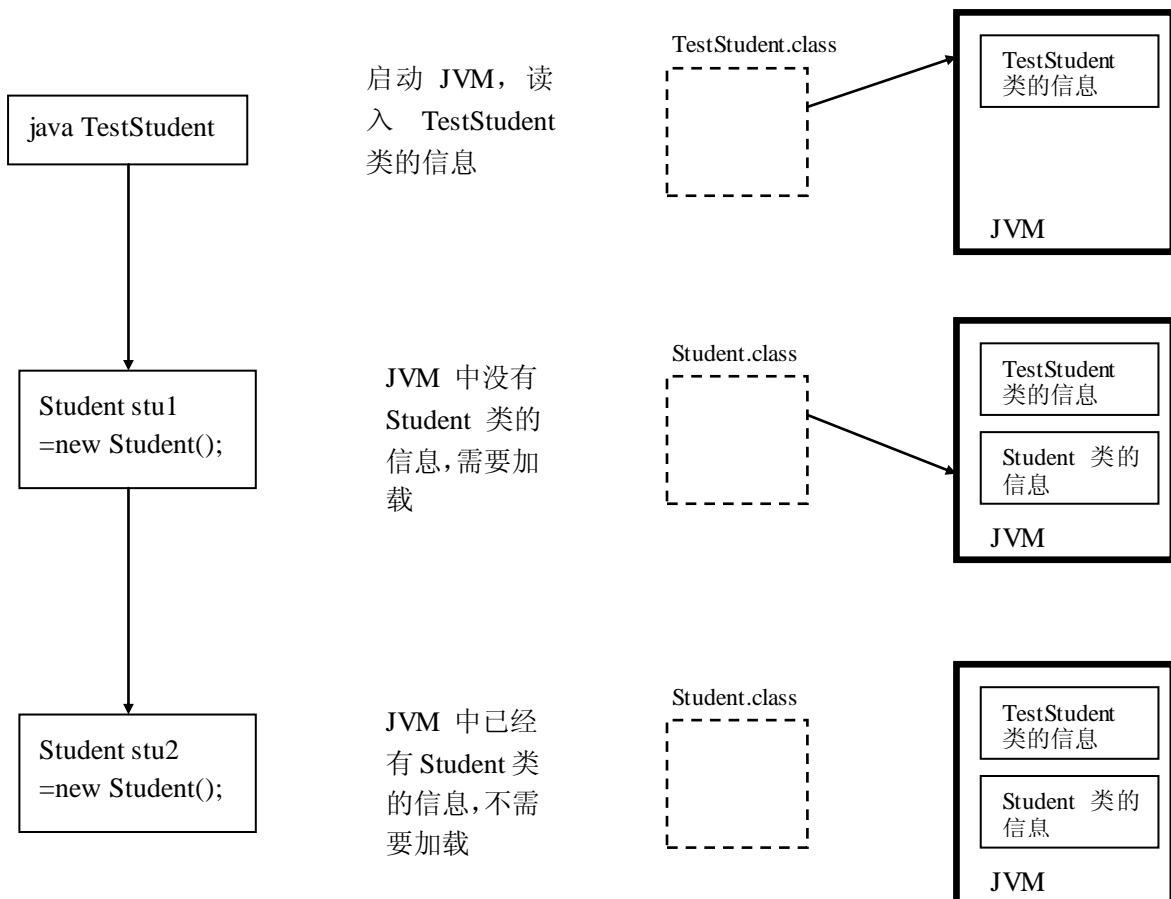
```
java TestStudent
```

则，首先启动 JVM，然后在硬盘上找到 TestStudent.class 文件，读入这个文件，并开始解释执行。此时，JVM 中只有 TestStudent.class 这个类的信息。

然后，执行主方法。在第 11 行时，遇到 Student stu1 = new Student();这句话。这个语句要求创建一个 Student 类的对象，但是此时，在 JVM 中，只有 TestStudent 类的信息，而没有 Student 类的信息！

这个时候，JVM 会自动的通过 CLASSPATH 环境变量，去硬盘上寻找相应的 Student.class 文件。当 JVM 找到这个文件之后，会把这个文件中所保存的 Student 类的信息读入到 JVM 中，并保存起来。此时，JVM 中保存有 Student 类和 TestStudent 类两个类的信息。示意图如

下：



当创建了第一个 Student 对象之后，创建第二个 Student 对象时，由于 JVM 中已经存在 Student 类的信息，因此就不需要重新读入 Student.class 文件。

类加载就是把.class 文件读入 JVM 的过程。也就是说，当 JVM 第一次遇到某个类时，会通过 CLASSPATH 找到相应的.class 文件，读入这个文件并把类的信息保存起来，这个过程叫做类加载。

而静态初始化代码块会在类加载的时候执行。

因此，执行 TestStudent，会输出：

```
in Student Static  
Student()  
Student()
```

当创建第一个 Student 对象时，由于是第一次遇到 Student 类，因此 JVM 会对 Student 类进行类加载，在类加载的时候，Student 类的静态初始代码块将会运行。此时会输出：in Student Static。

类加载完毕之后，创建对象时，会调用对象的构造方法，因此输出：Student()。调用完构造方法之后，第一个对象创建完毕。

创建第二个对象时，由于 JVM 中已经有 Student 类的信息，此时不需要类加载。创建对象时，调用对象的构造方法，于是输出 Student()。

另外，类加载除了读取类的信息，执行静态初始化代码块之外，还会为类的静态属性分配空间，并初始化其值为默认值。

## 2 final

final 属性能够修饰变量、方法和类。注意，所谓变量，既包括属性，又包括局部变量。也就是说，final 能够修饰属性、局部变量、方法参数（注意方法参数也是特殊的局部变量）。

### 2.1 常量

用 final 修饰的变量则称为常量。怎么来定义常量呢？可以把常量理解为：一旦赋值，其值不能改变的变量。

例如下面这个程序：

```
public class TestFinalVar{  
    public static void main(String args[]) {  
        final int N; // 定义一个常量，注意常量名要求全大写  
        N = 100; // 为常量第一次赋值  
        N++; // 编译出错，为常量赋值之后，常量值不可以改变  
        System.out.println(N); // 获得常量的值  
    }  
}
```

对于基本类型的变量而言，所谓的不能改变，指的是不能改变变量的值。下面看另一个例子。

```
01: class MyValue{  
02:     int value;  
03: }  
04: public class TestMyValue{  
05:     public static void main(String args[]) {  
06:         final MyValue MV; // 定义一个 MyValue 类型的常量  
07:         MV = new MyValue(); // 创建一个对象  
08:         MV.value = 10; // 为 MV 的 value 属性赋值  
09:         MV.value = 100; // 为 MV 的属性再次复制！编译通过  
10:         MV = new MyValue(); // 编译错误 MV 不能指向另一个对象！  
11:     }  
12: }
```

如上面的例子，MV 为一个对象类型的变量。对象类型的变量中，保存的是对象的地址，因此所谓“一旦赋值，不能改变”，是指的对象的地址不能改变。也就是说，对于对象类型的常量而言，一旦指向某个对象以后，就不能指向其他的对象。在 07 行，创建了一个对象，并让 MV 常量指向这个新对象。因此，在 10 行，不能创建新对象，让 MV 指向这个新对象。

但是，让 MV 指向一个对象之后，完全可以改变这个对象的属性，如第 08、09 行显示。这并不违反常量“一旦赋值，不能改变”的要求。

了解了 final 修饰不同类型变量的情况，下面我们讨论一下 final 修饰属性时的情况。例如，有如下代码：

```
class MyValue{  
    final int value;  
}
```

上面演示了 final 修饰实例变量的情况。要注意的是，上面这段代码无法编译通过！

考虑一下对象创建的过程。对于一般的实例变量而言，创建时第一步是分配空间，在分

配空间的同时，还会为实例变量赋一个默认值。然而，对于 final 类型的实例变量而言，如果在分配空间的同时赋默认值的话，那根据“一旦赋值，不能改变”的定义，final 实例变量的值在成为默认值之后将不能改变。这样的话，final 的实例变量将只能有默认值，不能再赋值为其他的值。在上述例子中，value 属性的值将被赋值为 0，且永远为 0。这显然并不是程序员所希望的。

正因为如此，对于 final 属性而言，虚拟机在分配完空间之后，将不会为其赋默认值，从而把为 final 属性赋值的那一次机会留给程序员。

但是，Java 语言规定，一旦对象创建完成，则对象的 final 类型的属性也就不能赋值了。因此，对于 final 类型的实例变量，能够赋值的时机，就是在分配空间之后，对象创建完成之前。实际上，在这段时间内，有两次赋值的机会：1. 初始化属性；2. 调用构造方法。

下面的两个例子演示了利用初始化属性以及调用构造方法为 final 属性赋值时的情况。

```
//使用属性初始化为 final 属性赋值
class MyValue1{
    final int value = 100; //初始化属性时赋值
}

//使用构造方法为 final 属性赋值
class MyValue2{
    final int value;
    public MyValue2(int value){
        this.value = value;
    }
}
```

由上面这两个代码例子，我们可以学习到怎样为 final 属性赋值。然而，有三个要注意的细节。

首先，为 final 属性赋值有两个时机。对于程序员来说，必须抓住两个时机中的一次，但是不能试图两次机会都抓住。例如下面的代码例子：

```
//注意！下面的代码会编译出错！
class MyValue3{
    final int value = 100;
    public MyValue3(int value){
        this.value = value;
    }
}
```

上面这段代码，由于既利用了初始化属性赋值，又使用了构造方法赋值，因此会编译出错！

其次，**对于使用构造方法赋值的情况。如果有多个构造方法，则多个构造方法都必须对 final 属性赋值。**

例如下面的例子：

```
class MyValue4{
    final int value;

    public MyValue4(){
```

```

    }

    public MyValue4(int value) {
        this.value = value;
    }
}

```

这段代码将编译失败，因为在 MyValue4 类的无参构造方法中，没有为 value 属性赋值。如果用户利用这个构造方法来创建对象，那么这个对象的 value 属性将没有赋值，这显然是错误的。

因此，在一个类中所有的构造方法里，都要对 final 属性赋值。这样就保证了，当一个对象创建的时候，无论调用的是哪一个构造方法，final 属性都被正确的赋值了。

上面是 final 属性的介绍。接下来考虑一个初始化 final 属性的问题。假设程序员选择了在定义属性的时候直接对 final 属性进行赋值，如下面代码：

```

class MyValue6{
    final int value = 200;
}

public class TestMyValue6{
    public static void main(String args[]) {
        MyValue6 mv1 = new MyValue6();
        MyValue6 mv2 = new MyValue6();
    }
}

```

考虑上面的代码。在这段代码中，创建了两个 MyValue6 类型的对象。这两个对象的空间中，分别保留着一块内存空间，用来保存 value 属性。

然而，由于 value 属性在初始化的时候被直接赋值为 200，因此对于所有对象来说，value 属性的值都为 200。并且，由于 final 属性的含义，所有对象的 value 属性都不能改变。因此，造成了这样的局面：每个对象中都保留了一块空间用来存放 value 属性，这个 value 属性的值都一样，而且修改不了。

这样，实际上造成了内存空间的浪费。由于这种属性全类所有对象都一致，因此可以把这个属性写成 static 的。即把 MyValue6 改为：

```

class MyValue6{
    final static int value = 200;
}

```

要注意的是，由于 static 属性是在类加载的时候分配空间的，因此静态的 final 属性不能在构造方法中赋值。我们可以选择在定义这个属性的时候赋值，或是在静态初始代码块中为这个属性赋值。

## 2.2 final 方法

相对 final 属性，final 方法的含义相对简单。final 修饰方法，表示该方法不能被子类覆盖。例如以下代码：

```

class Super{
    public void m1() {}
    public final void m2() {}
}

```

```
class Sub extends Super {  
    public void m1() {} // 能够覆盖父类方法  
    public void m2() {} // 编译出错！无法覆盖父类方法！  
}
```

如上面的代码例子所示，Super 类中有两个方法，m1 方法没有被 final 修饰，因此子类方法能够覆盖父类方法。m2 方法被 final 修饰，因此子类方法不能够覆盖父类方法。

## 2.3 final 类

final 修饰符也可以用来修饰类。**final** 类表示这个类不能被继承。例如：

```
final class MyClass{  
}  
class Sub extends MyClass{} // 编译出错，无法继承一个 final 类  
如上面的代码所示，final 类不能被继承。
```

要注意区分 final 修饰方法和 final 修饰类的区别。一个类中有 final 方法，这个类能够被继承，但是无法覆盖 final 方法；而一个类如果本身就是 final 的，则这个类无法被继承，这样的话，它所有方法都无法被覆盖。

## 3 abstract

abstract 可以用来修饰类和方法。abstract 单词本身表示“抽象”，是 java 中一个很重要的修饰符。

### 3.1 抽象类

abstract 修饰类，则这个类就成为一个抽象类。抽象类的特点是：抽象类只能用来声明引用，不能用来创建对象。例如下面的例子：

```
abstract class MyAbstract{  
    public void m() {}  
}  
public class TestAbstract1{  
    public static void main(String args[]){  
        MyAbstract ma; //可以声明抽象类的引用类型  
        ma = new MyAbstract(); // 编译错误 不能创建抽象类的对象  
    }  
}
```

如上面的代码所示，**抽象类可以声明引用，但是不能用来创建对象。**

这样的类有什么用呢？虽然抽象类**不能**创建对象，但是**抽象类可以被继承**，从而创建子类的对象。例如下面的例子：

```
abstract class MyAbstract{  
    public void m() {}  
}  
class MySubClass extends MyAbstract{  
}
```

```
public class TestAbstract1{
    public static void main(String args[]) {
        MyAbstract ma; //可以声明抽象类的引用类型
        // ma = new MyAbstract(); 错误的代码，不能创建抽象类的对象
        ma = new MySubClass(); //但是可以创建子类对象
    }
}
```

虽然抽象类不能创建对象，但是抽象类能够声明引用，并让这个引用指向子类对象。从某种意义上说，写抽象类的目的就是为了能够让子类继承。

更多的情况是把抽象类和抽象方法结合在一起使用，下面我们为大家介绍抽象方法。

## 3.2 抽象方法

用 `abstract` 修饰的方法称为抽象方法。抽象方法是 Java 中一个很重要的概念，在大家今后的编程实践中，会在很多地方用到抽象方法的概念。

我们在前面讲到“方法”这个概念的时候强调过，定义一个方法分两个部分：方法的声明，方法的实现。

**抽象方法指的是：一个只有声明，没有实现的方法。对于抽象方法来说，方法的实现部分用一个分号来代替。例如：**

```
class MyAbstract{
    public abstract void m1();
    public void m2(){}
}
```

其中的 `m1` 方法就是一个抽象方法。在这个方法的声明后面，没有代码块，只有一个分号，也就是说没有方法的实现。与之对应的是 `m2` 方法。`m2` 方法提供了一个空的方法实现，在实现之后没有分号。`m2` 方法不是一个抽象方法。

抽象方法是一种“只有声明，没有实现”的方法。方法的实现留给子类来完成。因此，如果一个类中有抽象方法，我们就可以认为这个类是一个“半成品”（因为这个类中的抽象方法缺少实现，实现的工作要由子类来继续完成）。一个“半成品”的类是不能用来创建对象的。

因此，**如果一个类中有抽象方法，这个类就必须是抽象类**。因此，上面的代码应该改写为：

```
abstract class MyAbstract{
    public abstract void m1();
    public void m2(){}
}
```

**注意：有抽象方法的类必须是抽象类，但是反过来，抽象类中未必有抽象方法。**

前面提到过，抽象方法是留给子类来实现的，因此，我们可以写一个子类来继承 `MyAbstract` 类：

```
class MySubClass extends MyAbstract{}
```

这个类将无法编译通过，因为 `MySubClass` 类将从父类中继承到抽象的 `m1` 方法，也就相当于，在 `MySubClass` 类中出现了一个抽象方法，因此 `MySubClass` 类也必须是抽象类。

如果我们不希望 `MySubClass` 类也成为抽象类，就必须想方设法的“去除掉”这个类中的抽象方法。

我们可以用一个有实现的，完整的 m1 方法，覆盖掉父类继承下来的抽象的 m1 方法。如下代码：

```
class MySubClass extends MyAbstract{  
    public void m1(){  
        System.out.println("In SubClass");  
    }  
}
```

这样，MySubClass 类中的抽象方法就被“覆盖”掉了。在前面的章节中，我们给出的“覆盖”的定义为：子类用特殊的方法实现替换掉父类的一般的实现。而这里的情况略有不同：子类覆盖父类的抽象方法时，并不是用一个特殊的实现替换一个一般的实现，而是在父类没有方法实现的情况下，子类给出一个方法的实现。因此，像这样，用一个有方法体的方法覆盖一个没有方法体的方法，也可以称之为“实现”了该方法。例如，我们可以说，MySubClass 类实现了父类中的 m1 方法。

因此我们得出结论：**子类继承一个抽象类，如果我们不希望子类也成为抽象类，就必须让子类实现父类中定义的所有抽象方法。**

我们再看主方法：

```
01: public class TestAbstract{  
02:     public static void main(String args){  
03:         MyAbstract ma = new MySubClass();  
04:         ma.m1();  
05:     }  
06: }
```

对于第 3 行代码，MyAbstract 类是抽象类，可以用来声明引用，而子类 MySubClass 实现了所有的抽象方法，不是抽象类，因此可以用来创建对象。这句代码是正确的。

对于第 4 行代码，能否编译通过呢？我们来回忆一下**多态的几条原则**：

- 只能对一个引用调用引用类型中定义的方法。
- 运行时会根据对象类型调用子类覆盖之后的方法。

在 MyAbstract 类中，我们确实定义了 m1 方法（尽管没有实现），因此对 ma 引用，我们完全可以调用 m1 方法，这句代码是能够编译通过的。进而，在运行时，将调用子类（也就是 MySubClass 类）中实现之后的 m1 方法，因此，运行时屏幕上将打印出：

```
In SubClass
```

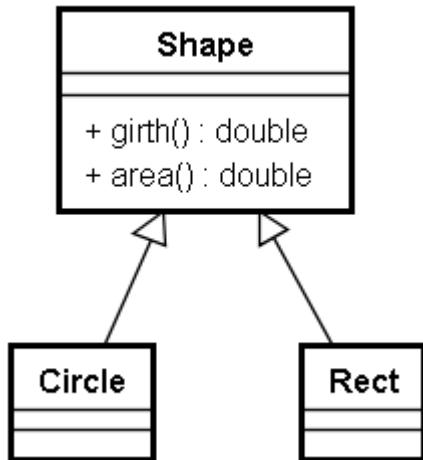
由此看出，抽象的语法和多态是紧密配合的。

总结一下：抽象方法的语法特征有如下几条：

1. 抽象方法只有声明，没有实现。实现的部分用分号表示。
2. 一个拥有抽象方法的类必须是抽象类。
3. 子类继承抽象类，要么也成为抽象类，要么就必须实现抽象类中的所有抽象方法。

### 3.3 抽象的作用

首先来看下面这个例子：写一个 Shape 类，表示一个形状，并为这个类提供两个方法：一个用来计算周长，一个用来计算面积。为这个类提供两个子类，一个是 Circle 类，表示一个圆形；一个是 Rect 类，表示一个矩形。下面是类的继承关系图：



其中，`girth`方法表示求周长，`area`方法表示求面积。

下面是代码的实现：

```

01: class Shape{
02:     public double girth(){
03:         return 0;
04:     }
05:     public double area(){
06:         return 0;
07:     }
08: }
09: class Circle extends Shape{
10:     private double r;
11:     private static final double PI = 3.1415926;
12:     public Circle(double r){
13:         this.r = r;
14:     }
15:     public double girth(){
16:         return 2*PI*r;
17:     }
18:     public double area(){
19:         return PI*r*r;
20:     }
21: }
22: class Rect extends Shape{
23:     private double a;
24:     private double b;
25:     public Rect(double a, double b){
26:         this.a = a;
27:         this.b = b;
28:     }
29:     public double girth(){

```

```
30:         return 2 * (a+b);
31:     }
32:     public double area() {
33:         return a*b;
34:     }
35: }
```

在这个继承关系中，Shape 中定义了 girth 和 area 方法，因为所有图形都能够计算周长和面积，这是所有图形的共性，应当放在父类。但是，对于不同的 Shape 中，对 girth 和 area 方法的实现都不相同，因此，这两个方法肯定需要被子类覆盖。然而，这两个方法虽然需要被覆盖，但是，在 Shape 类中，却不能不写“return 0”这个实现。因为如果不写，这两个方法就没有按照声明的要求，返回 double 值。这样会导致编译失败。

因此“return 0”这句代码就比较尴尬了，不写是不行的，写了又永远不会执行（因为根据多态，被调用的永远是子类覆盖之后的方法）。那么如何解决这个问题呢？

上面描述的 girth 与 area 方法，具有这样的特点：**方法的声明是共性，方法的实现是特性。**也就是说，所有形状都能够求周长和求面积，这是共性；而形状类不同的子类，求周长和求面积的方式不同，这是特性。遇到这种情况，我们就可以利用抽象方法来描述这种关系。

我们改写一下 Shape 类，结果为：

```
abstract class Shape{
    abstract public double girth();
    abstract public double area();
}
```

我们以前曾经分析过，方法的声明代表“对象能做什么”，而方法的实现代表“对象怎么做”，那么对于 Shape 类型来说，我们只能确定，任何一个形状对象都有“求周长”和“求面积”这两个功能，但是无法确定形状对象“怎么求周长”“怎么求面积”，具体周长和面积的算法要留给不同的子类，给出不同的实现。

因此，使用抽象方法，就可以让我们把方法的声明放在父类中，把方法的实现留在子类中。这样一方面，能够很好的体现出“共性放在父类”这个基本的原则，另一方面，用父类类型的引用又可以调用这些方法，并不影响多态语法的正常使用。

在 Shape 类中出现抽象方法的同时，这个类也成为了抽象类。这也是合理的，因为“形状”这个概念本身就是一个抽象的概念，在实际中，只会有矩形对象，圆形对象，菱形对象等等，不可能出现一个孤立的“形状”对象。因此，形状这个类就应该是一个抽象类。

我们换一个例子。在生活中，我们可以见到狗对象，猫对象，猴子对象等等，因此我们有了狗类，猫类，猴子类。从这些类中，我们归纳出了“动物类”。这些类都是动物类的子类。但是，谁又见过一个既非狗又非猫，什么具体动物都不是的“动物对象”呢？动物类是从那些具体的类中抽象出来的，而这个类本身又不会有任何对象，因此，动物类只能是个抽象类。

在动物类中，我们可以定义“吃”这个方法，因为我们确认，所有动物都会吃。但是我们又无法实现这个“吃”方法，因为不同的动物在“吃”方面有不同的行为方式，只能在不同的动物的子类中去分别实现“吃”方法。因此，“吃”方法在动物类中，就只能是个抽象方法。

由此可见，抽象的语法和其他的面向对象语法一样，都是从我们的实际生活中归纳总结出来的。

# Chp9 接口

## 本章导读

接口是 Java 语言中的核心概念之一。这个语法特性与“多态”具有非常紧密的联系。在学习接口之前，请读者首先复习一下多态的相关知识和练习，确认已经对多态有比较牢固的掌握之后，再进行下一步接口的学习。

### 1 接口的语法

#### 1.1 接口是特殊的抽象类

从语法特性上说，接口很类似于抽象类。

如果有一个抽象类，其所有属性都是公开静态常量，所有方法都是公开抽象方法，例如下面代码所示。

```
abstract class MyAbstractClass{
    public static final int VALUE1 = 100; //属性是公开静态常量
    public static final int VALUE2 = 200; //第二个属性
    public abstract void m1(); //方法是公开抽象方法
    public abstract void m2(int n); //第二个方法
}
```

上面的 `MyAbstractClass` 类，具有的两个属性 `VALUE1` 和 `VALUE2` 都是公开静态常量，具有的两个方法 `m1` 和 `m2` 都是公开抽象方法。

由于 `MyAbstractClass` 是抽象类，因此无法创建对象，只能声明引用。如果要创建对象的话，必须要写一个类继承 `MyAbstractClass` 类，并且实现这个类中的 `m1` 和 `m2` 方法。例如下面的代码：

```
class MySubClass extends MyAbstractClass{
    public void m1() {}
    public void m2(int n) {}
}
```

要注意的是，方法覆盖要求“子类的访问修饰符相同或更宽”，由于 `MyAbstractClass` 类中的 `m1` 方法和 `m2` 方法都是 `public` 的，`MySubClass` 中的 `m1` 和 `m2` 方法的访问修饰符也必须是 `public` 的。

对于 `MyAbstractClass` 这种特殊的抽象类，我们可以把其改写成接口。接口的特点和之前我们提到的 `MyAbstractClass` 的特点相同：

- 1、所有属性都是公开静态常量
- 2、所有方法都是公开抽象方法

使用关键字 `interface` 来定义接口。把 `MyAbstractClass` 改写成接口，结果如下：

```
interface MyInterface{
    public static final int VALUE1 = 100;
    public static final int VALUE2 = 200;
```

```
    public abstract void m1();
    public abstract void m2(int n);
}
```

注意, interface 替代了 abstract class 这两个关键字。interface 关键字和 class 关键字类似, 一个接口编译后会生成一个.class 文件; 一个.java 文件中可以有多个接口, 但是最多只能有一个公开的接口, 且公开接口的接口名与文件名相同。

既然接口中所有属性都是公开静态常量, 则接口中的属性, 可以省略 public static final 关键字; 同样的, 由于接口中所有方法都是公开抽象方法, 因此可以省略 public abstract 关键字。因此, 上面的 MyInterface 可以改写如下:

```
interface MyInterface{
    int VALUE1 = 100;
    int VALUE2 = 200;
    void m1();
    void m2(int n);
}
```

上面的这个接口中, 虽然没有写 public static final, 但是其属性都是公开静态常量; 虽然没有写 public abstract, 但是其方法都是公开抽象方法。

与抽象类类似, 接口可以声明引用, 但是不能创建对象。接口与抽象类不同的在于, 抽象类中可以定义构造方法, 以供子类的构造方法调用, 而接口中不能定义任何构造方法, 系统也不会提供默认无参的构造方法。

类似于子类继承抽象类, 接口也可以被子类“继承”。只不过, 接口具有自己的关键字: implements。使用这个关键字表示“实现”, 类似于抽象类中子类继承父类的关系。例如, 下面的 MyImpl 类就实现了 MyInterface 接口。

```
class MyImpl implements MyInterface{
    public void m1(){}
    public void m2(int n){}
}
```

要注意的是:

1. 一个类实现接口, 如果不希望这个类作为抽象类, 则应该实现接口中定义的所有方法。
2. 接口中所有的方法都是公开方法。因此, 在实现接口中的方法时, 实现类的方法也必须写成公开的! 由于类中的方法, 默认访问修饰符是“default”, 因此, 在实现接口中的方法时, 修饰符“public”不能省略。

接口最基本的使用就介绍到这里, 从这一节的内容可以看出, 从语法上说, 接口很类似特殊的抽象类, 只不过在接口语法中增加了两个关键字: interface 和 implements 而已。

除此之外, 抽象类之间可以继承, 同样的, 接口之间也可以继承。接口之间继承时, 使用的关键字同样为 extends。例如:

```
interface IA{
    void ma();
}

interface IB extends IA{
    void mb();
}
```

上面这个例子中，`IB` 接口继承自 `IA` 接口。因此，`IB` 中存在两个方法：`ma` 方法是从 `IA` 接口中继承来的，`mb` 方法是 `IB` 接口自身定义的。如果有一个类要实现 `IB` 接口，则必须实现 `ma` 和 `mb` 两个方法，例如：

```
class IAIBImpl implements IB{
    public void ma() {}
    public void mb() {}
}
```

## 1.2 多继承

当然，接口和抽象类除了关键字不同外，还有一些非常重要的不同之处。

首先，接口之间可以多继承。不同于 Java 中对于类之间的单继承的要求，接口之间没有这个限制。一个接口可以继承多个接口，例如下面这个例子：

```
interface IA{
    void ma();
}

interface IB{
    void mb();
}

interface IC extends IA, IB{ //IC 同时继承 IA 和 IB 两个接口
    void mc();
}
```

上面这段代码中，`IC` 接口继承自 `IA` 和 `IB` 接口。注意，继承多个接口时，多个接口之间用逗号隔开。`IC` 同时继承这两个接口，因此 `IC` 中同时包含有这两个接口中定义的方法，并且包含自身定义的 `mc` 方法。因此，如果有一个类要实现 `IC` 接口的话，则需要实现三个方法：`ma`、`mb` 以及 `mc`。

除此之外，一个类在继承另外一个类的同时，还可以实现多个接口，例如下面的例子：

```
interface IA{
    void ma();
}

interface IB{
    void mb();
}

//IC 继承自 IA, IB 接口。这里是接口的多继承
interface IC extends IA, IB{
    void mc();
}

interface ID{
    void md();
}

abstract class ClassE{
    public abstract void me();
}
```

```

}

/*
    一个类可以继承自一个类，并实现多个接口
    MyImpl 继承自 ClassE 类，实现了 IC 和 ID 接口
    注意，先写继承自哪个类，再写实现了哪些接口
*/
class MyImpl extends ClassE implements IC, ID{
    //IC 接口中包含 ma、mb、mc 方法
    public void ma() {}
    public void mb() {}
    public void mc() {}

    //ID 接口中包含 md 方法
    public void md() {}

    //ClassE 中包含 me 方法
    public void me() {}
}

```

以上就是接口和抽象类不同的地方：接口和接口之间可以多继承；一个类在继承一个父类的同时，还能够实现多个接口。

### 1.3 接口与多态

有了接口的多继承特性，加上一个类能够实现多个接口，这样，接口结合多态，语法和概念都变得非常的灵活。例如，在之前 MyImpl 类的基础上，写以下代码：

```

public class TestMyImpl{
    public static void main(String args[]){
        IA ia = new MyImpl();
        System.out.println(ia instanceof IA);
        System.out.println(ia instanceof IB);
        System.out.println(ia instanceof IC);
        System.out.println(ia instanceof ID);
        System.out.println(ia instanceof ClassE);
        System.out.println(ia instanceof MyImpl);
    }
}

```

上面的程序会输出 6 个 true。

我们以前讲过，`instanceof` 关键字用来判断对象和某个类型是否兼容。

在这段代码中，`ia` 引用指向了一个 `MyImpl` 类的对象。我们知道，`MyImpl` 类实现了 `ID` 接口，如果把 `ID` 接口看作是一个特殊的抽象类，那么 `MyImpl` 类就可以看作是这个抽象类的子类，因此代码

```

System.out.println(ia instanceof ID);

```

会输出“true”。

同理，`MyImpl` 类还实现了 `IC` 接口，而 `IC` 接口是 `IA`, `IB` 两个接口的子接口。因此，`MyImpl` 类也可看作是 `IA`, `IB` 两个接口的实现类，因此代码

```

System.out.println(ia instanceof IA);
System.out.println(ia instanceof IB);

```

```
System.out.println(ia instanceof IC);
```

也会输出“true”。

当然，`MyImpl`类本身又继承了`ClassE`类，因此代码：

```
System.out.println(ia instanceof ClassE);
```

```
System.out.println(ia instanceof MyImpl);
```

也会输出“true”。

所以，上述代码会输出6个“true”。因此，根据多态的语法，一个`MyImpl`类的对象可以放入：`IA, IB, IC, ID, ClassE, MyImpl`六种不同类型的引用中。

实际上，对于初学者，接口很多比较难掌握和理解的东西，都跟接口的多态特性有关。只要掌握好了多态，就能相当程度上把握住接口的应用。

例如下面这个例子：

首先定义一个`Teacher`接口：

```
interface Teacher{  
    void teach();  
}
```

然后，为接口提供两个实现类：

```
class CoreJavaTeacher implements Teacher{  
    public void teach(){  
        System.out.println("teach corejava");  
    }  
}  
  
class JavaWebTeacher implements Teacher{  
    public void teach(){  
        System.out.println("teach java web");  
    }  
}
```

之后，提供一个`TestTeacher`类如下：

```
public class TestTeacher{  
    public static void main(String args[]){  
        Teacher t = getTeacher(0);  
        beginClass(t);  
    }  
  
    public static Teacher getTeacher(int type){  
        if(type == 0) return new CoreJavaTeacher();  
        else return new JavaWebTeacher();  
    }  
  
    public static void beginClass(Teacher t){  
        t.teach();  
    }  
}
```

注意在上面的代码中的两个函数。首先，`getTeacher`方法的返回值为一个`Teacher`对

象。由于 Teacher 类型是一个接口类型，因此不会返回一个真正的接口对象，而返回的对象一定是接口的某一个实现类的对象。这是把多态用在方法的返回值类型上。

而 beginClass 方法能够接受一个 Teacher 类型的参数。由于 Teacher 是接口类型，因此接受的参数一定是 Teacher 接口实现类的对象。这是把多态用在方法的参数类型上。

以上的代码与我们在多态章节中见到的代码有很多类似之处，不同的是，将“父类”的概念换成了“接口”的概念。在含义上，这二者是相同的。

## 2 接口的作用

上一部分我们介绍了接口的语法。事实上，接口的语法并不困难，甚至于应该说相当的简单。与接口的语法相比，更难以理解和掌握的，是接口的作用，以及如何更好的利用接口。

在 Java 中，接口主要用来实现两大功能：一是用接口实现多继承；二是用接口来进行解耦合。其中，后者是接口最重要的作用。

### 2.1 接口与多继承

在介绍 Java 的历史和背景时，我们曾经介绍过，开发 Java 语言的工程师，都有多年 Unix 下使用 C++ 语言进行开发的经验。而且，Java 中有很多的语法以及关键字类似于 C++，可以说，Java 语言是一种脱胎于 C++ 的语言。

但是，虽然有些语法类似，但是 Java 语言与 C++ 语言有着本质的区别。Java 语言和 C++ 最大的区别之一，就是多继承：C++ 支持多继承，而 Java 语言只允许单继承。

例如，假设我们设计了一个类：Spider，这个类表示蜘蛛。又设计了一个类 Man，这个类表示人。现在，我们要设计一个类：SpiderMan，表示“蜘蛛侠”。

很显然，“蜘蛛侠”这个类，既有 Man 这个类的特点（会说话，会走路，会恋爱……），也有 Spider 这个类的特点（能射出蛛丝）。也就是说，我们可以把蜘蛛侠当做特殊的蜘蛛，也可以把蜘蛛侠当做特殊的人。从这个意义上说，SpiderMan 应当既是 Spider 的子类，又是 Man 的子类。

C++ 语言支持多继承，因此在 C++ 语言中，我们可以让 SpiderMan 这个类直接继承自两个父类。

但是，在 Java 中只允许单继承。为此，我们只能为 SpiderMan 选择唯一的一个父类，并让其实现别的接口。例如，我们可以让 SpiderMan 继承自 Man 类，然后，把 Spider 作为接口的形式，让 SpiderMan 实现这个接口。

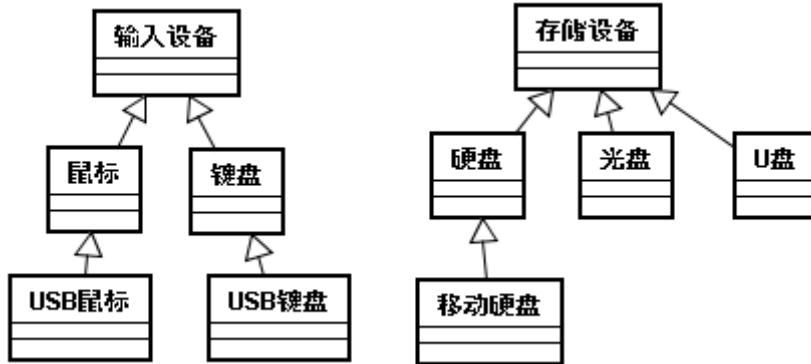
```
class SpiderMan extends Man implements Spider
```

虽然 Java 中只允许单继承，但是对于实现接口，Java 没有做数量上的限制。例如，SpiderMan 这个类只有一个父类 Man。但是，我们可以把接口当做是特殊的抽象类。而一个类实现一个接口，可以当做是特殊的继承。因此，从这个意义上来说，SpiderMan 这个类实现 Spider 接口，就是一种特殊的继承。因此，SpiderMan 继承了一个类 Man，又用一种特殊的方式继承了一个特殊的父类 Spider。这样，SpiderMan 就相当于继承自两个类。

上面的例子说明，由于实现一个接口，相当于继承自一个特殊的父类。因此，在 Java 语言中，我们可以使用接口，来实现了概念上的多继承。

既然 C++ 语言能够直接实现多继承，为什么 Java 语言要摒弃多继承这个特性，而要用接口这种语法来间接的实现多继承呢？

首先，使用接口实现多继承，能够区分主要类型和次要类型。考虑下面这个继承树：



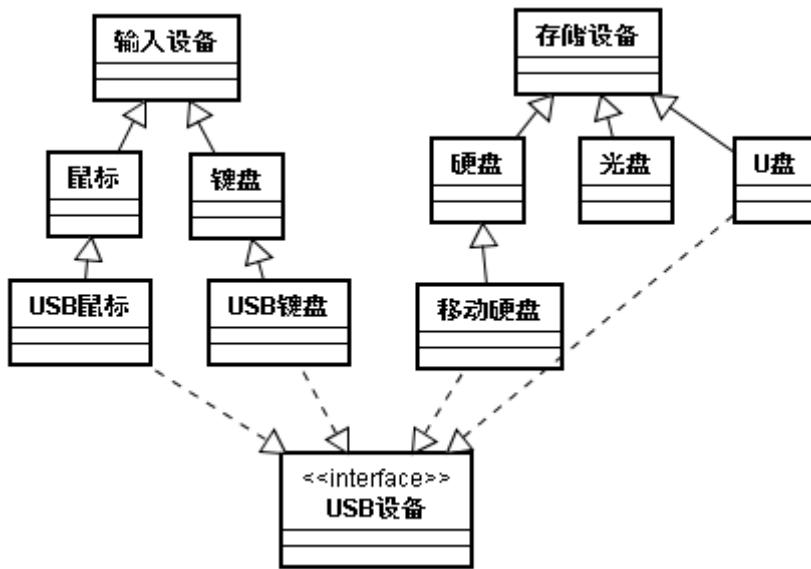
在上面这个继承树中，父类体现了设备的共性，而子类体现了特性。例如，USB 鼠标是特殊的鼠标，USB 键盘是特殊的键盘。而对键盘和鼠标这两个类提炼出共性，则得到父类“输入设备”。也就是说，键盘和鼠标都是特殊的输入设备。

再例如，移动硬盘是特殊的硬盘，而硬盘、光盘以及 U 盘，这些具体的子类抽象出共性，成为“存储设备”这个类。

我们可以看到，在上面的这几个类中，我们对一些具体的子类提炼出共性，从而形成了上面的继承树。

然而，这些类除了继承树中表现出的共性之外，还有其他的共性。

例如，USB 鼠标、USB 键盘、移动硬盘、U 盘，这些设备有一个共性：这些设备都能够通过 USB 端口与电脑相连，因此，他们除了各自的主要作用之外，有个额外的共性：他们都是 USB 设备。为了表示这个关系，我们设计一个接口：USB 设备，让上述四个设备实现这个接口。如下图：



这样，就利用接口实现了特殊的多继承。但是，虽然我们可以把实现接口当做特殊的继承，但是事实上，实现接口与继承父类相比，是处于相对“次要”的地位。这样，就能够区分“主要类型”和“次要类型”。

怎么来理解“主要类型”和“次要类型”呢？例如，对于“移动硬盘”来说。我们购买移动硬盘的主要目的，是为了存储数据。因此，“存储设备”是其主要类型。当然，为了让设备与电脑主机之间能够更加方便的交互，让移动硬盘实现“USB 设备”这个接口也是非常必要的。然而，相对于存储数据，用 USB 连接是一个次要的功能，因此，相对于“存储设

备”，“USB 设备”这是一个次要类型。我们把“USB 设备”定义为一个接口，就可以区分主要类型和次要类型。

在 Java 中，能够很容易的区分一个类的主要类型和次要类型。我们可以让一个类继承自其主要类型，而次要类型，可以作为接口，让这个类来实现。

例如，对于 SpiderMan 这个类来说，产生这个类的原因是一个 Man 类的对象受到了一些外界的影响（蜘蛛侠具有蜘蛛能力的原因，是因为小时候被蜘蛛咬了一口……），产生了变化，从而实现了 Spider 接口。在这个过程中，Man 是主要类型，而 Spider 是次要类型，是接口。

相对应的，对于“忍者神龟”这个类来说，产生这个类的原因是，四个乌龟对象受到影响之后，实现了“忍者”这个接口（四只动物园的海龟，被化学药品影响而成为了“忍者神龟”）。因此，对于忍者神龟来说，乌龟是主要类型，其次实现了“忍者”接口。

另外，通过“存储设备/输入设备”的例子，我们还可以看出，从概念上说，接口是怎么设计出来的。首先，我们在介绍继承关系的时候曾经说过，“父类”，从设计上说，是对多个不同子类的共性的抽象。在上面这个例子中，我们对“硬盘”、“光盘”、“U 盘”等子类进行了共性的抽象，抽象出“存储设备”这个父类来。然而，我们还可以对“移动硬盘”、“U 盘”、“USB 鼠标”等设备再一次进行共性的抽象，从而抽象出“USB 设备”这个接口。也就是说，父类可以认为是对子类主要共性的抽象，而接口可以认为是对子类次要共性的“再抽象”。

另外，单继承相对多继承的好处，就在于单继承具有简单性。使用单继承，类与类之间能够形成简单的树状结构。而对于多继承，类和类之间的关系相对要复杂的多，很有可能会形成复杂的网状结构。但是，用接口实现的多继承，则不会破坏类之间树状结构的简单性。这是因为这棵树是由类之间形成的，是事物主要类型所组成的关系。一个类实现再多的接口，有再多的次要类型，也不会改变其主要类型之间的树状结构。

例如，在生活中，每个家族的家谱都能够形成简单的树状结构，原因在于，在记录家谱的时候，只考虑一个人的亲生父亲。当然，人也可以认干爹。但是，干爹再多，也不会写到家谱中。因为干爹毕竟不是亲爹，在记录家谱的时候，亲爹是主要类型，而干爹只能是次要的，忽略不计的。

因此，在 Java 中，可以通过实现接口的方式来实现多继承。这种语法设计相对于 C++ 来说先进的多，因为，用接口实现多继承不会破坏类之间树状关系的简单性。

## 2.2 接口与解耦合

除了实现多继承之外，接口最重要的作用就是解耦合。什么叫解耦合呢？我们看下面这段代码的例子。

定义若干灯泡类。代码如下：

```
class RedBulb {  
    public void shine(){  
        System.out.println("Shine in Red");  
    }  
}  
  
class YellowBulb{  
    public void shine(){  
        System.out.println("Shine in Yellow");  
    }  
}
```

```

class GreenBulb{
    public void shine(){
        System.out.println("Shine in Green");
    }
}

```

然后创建一个台灯类，首先装上红灯泡，代码如下：

```

class Lamp{
    private RedBulb bulb;
    public void setBulb(RedBulb bulb) {
        this.bulb = bulb;
    }
    public void on(){
        bulb.shine();
    }
}
public class TestLamp{
    public static void main(String args[]){
        Lamp lamp = new Lamp();
        RedBulb rb = new RedBulb();
        lamp.setBulb(rb);
        lamp.on();
    }
}

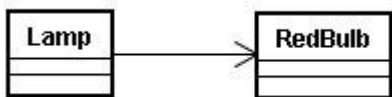
```

在上面的代码中，`lamp` 的 `on` 方法，调用了 `bulb` 的 `shine` 方法。也就是说，当我们调用台灯对象的“开”方法时，台灯对象会去调用灯泡对象的发光方法。

但是，问题来了：现在的这个台灯，装的是红灯泡。如果现在想要把红灯泡换成绿灯泡，应当如何操作呢？

由于 `Lamp` 类中的 `bulb` 属性是 `RedBulb` 类型，因此，一旦要修改，则必须要把 `Lamp` 类中的属性类型进行修改，并且修改 `setBulb` 方法的相应参数和实现。也就是说，当我们希望把 `bulb` 属性由 `RedBulb` 替换为 `GreenBulb` 的时候，必须修改 `Lamp` 类的代码。

也就是说，如果要想更换不同种类的灯泡，就要修改台灯的内部结构！这无疑是跟现实生活不相符合的。之所以产生这样的矛盾，原因在于 `Lamp` 类与 `RedBulb` 类型紧密联系在一起，形成了强耦合的关系，如下图：



下面我们使用接口来解决这样的问题。首先，定义 `Bulb` 接口：

```

interface Bulb{
    void shine();
}

```

然后，修改三种灯泡的代码，让他们都实现 `Bulb` 接口。

```

class RedBulb implements Bulb{

```

```

    public void shine(){
        System.out.println("Shine in Red");
    }
}

class YellowBulb implements Bulb{
    public void shine(){
        System.out.println("Shine in Yellow");
    }
}

class GreenBulb implements Bulb{
    public void shine(){
        System.out.println("Shine in Green");
    }
}

```

之后，修改 Lamp 类，并给出 TestLamp 类的代码。

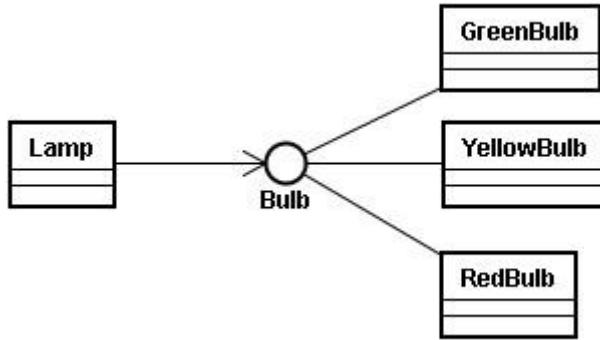
```

class Lamp{
    private Bulb bulb;
    public void setBulb(Bulb bulb){
        this.bulb = bulb;
    }
    public void on(){
        bulb.shine();
    }
}

public class TestLamp{
    public static void main(String args[]){
        Lamp lamp = new Lamp();
        Bulb b1 = new RedBulb();
        lamp.setBulb(b1);
        lamp.on();
        Bulb b2 = new GreenBulb();
        lamp.setBulb(b2);
        lamp.on();
    }
}

```

Lamp 类的 bulb 属性被改为接口类型 Bulb，从而，Lamp 类与具体的实现类之间用 Bulb 接口分开了。当 Lamp 类希望把 bulb 属性由 RedBulb 对象变更为 GreenBulb 对象时，不需要修改任何自身代码。只需要调用 setBulb 方法，接受不同的 Bulb 接口的实现类就可以了，从而，Lamp 类通过 Bulb 接口，实现了与 Bulb 实现类的弱耦合。如下图所示：



这样，我们就利用接口，把原来强耦合的关系，变为了弱耦合的关系。这就是接口最重要的作用：解耦合。

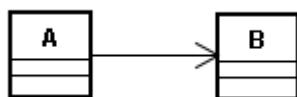
由于接口中所有的方法都是抽象方法，因此，定义一个接口，可以看作是定义了一个标准。它只定义了，一个对象应该具有哪些方法，而丝毫没有定义对象如何实现这些方法。方法的实现统统交给接口的实现类来完成。这样，接口的出现，就阻隔了接口使用者和接口实现者之间的耦合关系。当接口实现者变化的时候，对接口使用者不产生任何影响。

在生活中，对象之间的弱耦合关系也是通过标准来实现的。例如，当电脑的硬盘出现故障的时候，我们可以很容易的为电脑更换一块其他品牌的新硬盘，而对电脑的主板 CPU 等元件不产生丝毫影响。这显然是因为，不同的硬盘厂商在生产自己的硬盘产品的时候，都会遵循统一的标准，如电气接口的规格，硬盘产品的尺寸等等。试想一下，如果没有了这些标准，各个厂商各自为战，生产出规格各异的硬盘产品，那么我们在更换硬盘的时候，是不是就没有了那么多的选择了呢？

### 3 接口回调

有了接口和多态之后，对我们的开发模式和开发思路都有着很深刻的影响。在企业级开发应用中，肯定不会把所有的功能都写在一个模块里，也肯定不会把所有的功能都让一个程序员来完成。因此，就有了程序员之间的分工和合作。

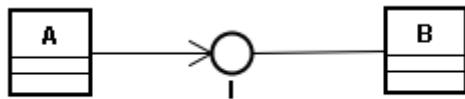
假设，现在有两个类需要完成，这两个类一个是 A 类，一个是 B 类，而 A 类需要调用 B 类提供的方法。示意图如下：



而现在有两个程序员，一个张三，一个李四，在项目经理的分配之下，两个人分别负责者两个类的编码。其中，张三负责 A 类，李四负责 B 类。

由于 A 类要调用 B 类提供的功能，因此，在李四没有完成 B 类的代码之前，A 类根本无法使用 B 类，必须要等到李四开发 B 类完成，张三才能够开始开发 A 类。也就是说，A 类作为功能的使用者，必须等到功能的实现者 B 类完成之后，才能进行开发。

但是，有了接口之后，我们可以用一种新的开发方式来解决这个问题。我们可以在开发之前，先设计一个接口 I，定义 B 类中应该具有哪些方法，让 B 类来实现接口 I。而利用 I，把 A 类和 B 类之间的紧耦合关系转为弱耦合关系，示意图如下：



有了 **I** 接口之后，张三在开发 **A** 类的时候，并不需要等待李四的 **B** 类开发完成。只要在 **A** 类中使用 **I** 接口类型，等 **B** 类开发完成之后，可以用多态调用 **B** 类中的实现。在这种情况下，**A** 类作为 **I** 接口功能的使用者，而 **B** 类作为 **I** 接口功能的实现者，在开发过程中，并不需要强调谁先谁后。**A** 类的开发者，完全可以在 **B** 类没有完成的情况下，就开始使用 **B** 类的方法（因为这些方法已经在 **I** 接口中定义了）。甚至于，有可能在没有开发 **B** 类的情况下，**A** 类就已经完成开发了。

这个改变，对程序开发来说，有着非常重大的影响。例如，在介绍“函数”时，我们提到，函数是面向过程中一个很重要部分。对于一个成熟的面向过程的语言来说，应当提供大量的函数库，让程序员使用。事实上，面向过程的程序员，都是去调用函数库中的函数，来完成自己所需要的功能。示意图如下：



也就是说，程序员在开发过程中，一直是扮演着“调用者”的角色。原因也很简单，面向过程的编程方式中，必须先把功能实现了，然后程序员才能去使用功能。

那能不能反过来呢？让程序员提供一些代码，而让系统提供的函数或者类，来调用程序员写的代码？也就是说，让程序员成为功能的提供者，而让系统成为功能的使用者？

上面的想法有什么意义呢？我们看下面这个例子：

例如，在 Java 中，可以利用 `java.util.Arrays.sort` 方法，来对数组进行排序。我们在数组部分的学习中，曾经为大家介绍过冒泡排序算法。冒泡排序，是所有的排序方法中最简单的排序方法，也是执行效率最低的方法。在计算机科学领域，有很多相对已经比较成熟的排序算法。这些算法都比冒泡排序要高效的多，但是也要比冒泡排序算法要复杂的多。幸运的是，Sun 公司提供了一个 `java.util.Arrays.sort` 函数，这个函数能够对数组进行排序，排序时使用的算法，是一种经过调优的快速排序算法。这个函数的使用如下：

```

public class TestArraySort{
    public static void main(String args[]){
        int[] a = {1, 7, 2, 5, 3};
        printArray(a); //输出 1 7 2 5 3

        java.util.Arrays.sort(a);
        printArray(a); //输出 1 2 3 5 7
    }

    public static void printArray(int[] a){
        for(int i = 0 ; i < a.length ; i++){

```

```

        System.out.print( a[i] + " ");
    }
    System.out.println();
}
}

```

我们可以看到，这个 `java.util.Arrays.sort()` 能够对 `int` 类型的数组进行排序。同样的，这个函数也能够对其他的一些基本类型的数组（例如 `byte[]`, `double[]` 等）进行排序。

那么除了基本类型之外，`Arrays.sort()` 能不能对对象类型的数组进行排序呢？

首先，如果要进行排序的话，除了排序算法之外，有一个最基本的要素：排序规则。简单的来说，因为排序是把一个数组中的元素从小到大依次排列，因此，必须要有一个途径，能够比较两个元素的大小。只有区分出元素的大小之后，才能够把小的元素放到前面，大的元素放在后面，从而最终让数组元素从小到大排列。

两个元素比较大小的方式，叫做排序规则。对于基本类型来说，排序规则就是数学上的比较方式。例如，如果要让 `int` 变量 `a` 和 `int` 变量 `b` 进行比较，只要直接使用表达式(`a>b`);如果这个表达式为 `true`，则 `a` 比 `b` 大；如果这个表达式为 `false`，则说明 `a` 小于等于 `b`。

但是比较对象时，就没有那么简单。首先，两个对象之间，不能直接使用`>`, `<`, `>=`, `<=` 等布尔运算符来比较大小。因此，程序员必须要自己指定两个对象如何比较大小。例如，当程序员创建了一个 `Student` 类之后，必须由程序员自己来指定排序规则，说明两个 `Student` 对象如何比较大小。假设有个两个 `Student` 对象，一个对象代表 18 岁的李四，另一个对象代表 20 岁的张三，这两个对象谁大谁小？程序员必须自己指定规则。

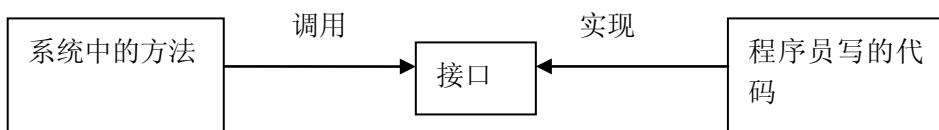
我们简单回顾一下刚刚说的内容。`java.util.Arrays.sort` 方法是一个高效的排序算法。但是，如果要对对象类型进行排序的话，则在 `sort` 方法中，必然会用到对象类型的排序规则。而这个规则，不是由 Sun 公司来定，是由程序员来定的。我们可以理解为，程序员提供排序规则，而在 Sun 公司提供的 `sort` 方法中，使用了这个排序规则。示意图如下：



可以看到，这跟传统的面向过程的模式有着很大的区别。在传统的开发模式中，系统提供函数库，让程序员调用；而在这种开发模式中，由程序员提供功能，让系统中的某些函数调用。

在 Java 中，我们可以利用接口实现这一点。

系统中事先定义好了一个接口，然后，在系统的方法中，调用这个接口中的方法。这样，系统中的方法就作为这个接口的使用者。而程序员实现这个接口，程序员就是接口的实现者。像这样，由程序员实现接口，由系统其他的类来通过多态进行调用，这种编程的方式，叫做接口回调。示意图如下：



例如，为了定义排序规则，Sun 公司定义了一个接口：java.lang.Comparable。这个接口用来表示一个对象的排序规则。我们编写的类应该实现这个接口。

Comparable 接口中只有一个方法：compareTo 方法，实现这个方法，就能规定两个对象如何比较大小。假设程序员要创建一个 Student 类，希望用 java.util.Arrays.sort() 对一些 Student 对象进行排序，则要求 Student 类实现 Comparable 接口。示意图如下：



那么如何实现 Comparable 接口呢？我们给出 `Student` 类的示例代码：

```
class Student implements Comparable<Student>{  
    int age;  
    String name;  
    public int compareTo(Student stu) {  
        //...  
    }  
}
```

这段代码有三个要注意的地方。

第一，在 `Student` 类实现 `Comparable` 接口时，后面有一个尾巴：“`<Student>`”。这部分是 Java5.0 提供的新特性，称之为“泛型”。这个特性在后面的还有详细的论述，此处不做过多的解释。

第二，在 `Student` 类中，必须要实现 `compareTo` 方法。这个方法接受一个 `Student` 对象作为参数。`compareTo` 方法用来比较两个对象，这两个对象一个是“当前对象”，另一个则是 `compareTo` 方法的参数。例如，假设有两个 `Student` 对象 `stu1` 和 `stu2`，则如果调用 `stu1.compareTo(stu2)`，就表示把 `stu1` 和 `stu2` 进行比较。其中“当前对象”就是指的 `stu1` 对象；而把 `stu2` 作为 `compareTo` 方法的参数，因此“参数”指的就是 `stu2`。

第三。`compareTo` 方法返回一个整数。这个整数的数值就表示比较的结果：如果返回值小于 0，则表明当前对象比参数对象小；如果返回值等于 0，则说明两个对象一样大；如果返回值大于 0，则说明当前对象比参数对象大。

例如，我们规定，对学生的年龄进行排序，年龄较小的学生排前面，年龄较大的学生排后面，则可以实现 `compareTo` 方法如下：

```
public int compareTo(Student stu) {  
    if (this.age > stu.age) {  
        return 1;  
    } else if (this.age < stu.age) {  
        return -1;  
    } else {  
        return 0;  
    }  
}
```

定义完 `compareTo` 方法之后，就可以使用 `Arrays.sort` 方法进行排序了。例如下面的例子：

```
public class TestSort {
```

```

public static void main(String args[]){
    Student[] ss = new Student[3];
    ss[0] = new Student("Tom", 18);
    ss[1] = new Student("Jim", 17);
    ss[2] = new Student("Jerry", 20);
    java.util.Arrays.sort(ss);
    for(int i = 0; i<ss.length; i++){
        System.out.println(ss[i].name + " " + ss[i].age);
    }
}
}

```

在调用 `Arrays.sort` 方法时，根据我们规定的排序规则，把年龄小的学生排在前面，把年龄大的学生排在后面。

完整代码如下：

```

class Student implements Comparable<Student> {
    int age;
    String name;

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public int compareTo(Student stu) {
        if (this.age > stu.age) {
            return 1;
        } else if (this.age < stu.age) {
            return -1;
        } else {
            return 0;
        }
    }
}

public class TestSort {
    public static void main(String args[]) {
        Student[] ss = new Student[3];
        ss[0] = new Student("Tom", 18);
        ss[1] = new Student("Jim", 17);
        ss[2] = new Student("Jerry", 20);
        java.util.Arrays.sort(ss);
        for (int i = 0; i < ss.length; i++) {
            System.out.println(ss[i].name + " " + ss[i].age);
        }
    }
}

```

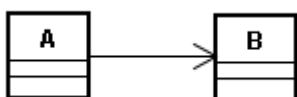
```
    }  
}  
}
```

输出结果如下：

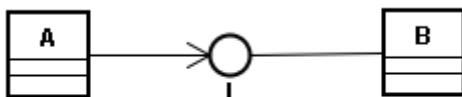
```
Jim 17  
Tom 18  
Jerry 20
```

可以看到，输出的结果，对学生对象的年龄进行了排序。在这个程序中，很显然先有 Sun 公司为我们提供的 Arrays.sort 方法，然后才有 Student 类作为接口的实现者。这是一个非常典型的接口回调：程序员提供 Comparable 接口的实现，供 JDK 中 Arrays.sort 方法来调用。

简单的说，我们习惯于这样的编程方式：



由系统为我们提供 B 类，而我们负责编写 A 类来使用 B 类。  
而接口回调为我们提供了新的方式：



由系统为我们提供 A 类和 I 接口，我们负责编写 B 类来实现 I 接口。A 类通过对 I 接口中方法的调用，利用多态，来调用我们所写的 B 类的方法。这不能不说这是编程方式的一次伟大的突破。

# Chp10 Object 类与常用类，内部类

## 本章导读

本章首先将为读者介绍 `Object` 类。这个类在 `Java` 中具有很特殊的地位，这是所有类的父类，是 `Java` 继承树的根。因此，这个类中的很多方法都很值得研究。

然后为大家介绍的是 `Java` 中的包装类和 `String` 类。这些都是 `Java` 开发中比较常用的类，也有很多常用的方法。

另外，本章还会介绍 `Java` 中内部类的一些语法。

## 1 Object 类

`Object` 类是 `Java` 中所有类的父类。例如下面的代码：

```
class ClassA{}  
class ClassB extends ClassA{}
```

在上面的代码中，`ClassB` 类明确的写明了，继承自 `ClassA` 类。那 `ClassA` 类呢？像这种没有明确写明 `extends` 的类，都继承自 `java.lang.Object`，也就是本章我们要介绍的 `Object` 类。正因为有这个特性，在 `Java` 中任何一个类，如果追根溯源的话，归根结底都是 `Object` 类的直接或者间接子类。

前面我们分析过，`Java` 中所有的类会组成一种树状关系，而 `Object` 类，就是这棵类继承关系树的树根。

既然 `Object` 类是所有类的父类，那我们就得好好研究一下这个类。

首先，`Object` 类既然是所有类型的父类，那么在 `Java` 中所有的对象，都能够赋值给 `Object` 类型的引用。这是因为子类对象可以直接赋值给父类引用，而所有 `Java` 中的类都是 `Object` 类的子类。

其次，由于子类中能够继承父类中的公开方法。因此，`Object` 类中所有的公开方法都能被子类继承。也就是说，`Object` 类中的公开方法，是 `Java` 中所有对象都拥有的方法。

接下来，我们就仔细来研究一下 `Object` 类的公开方法。

### 1.1 finalize

`finalize` 是一个 `protected` 的方法。虽然不是 `public` 的，但是这个方法也同样能够被所有子类继承（考虑一下，`protected` 能够被同包以及非同包的子类访问，也就是能够被所有子类访问）。

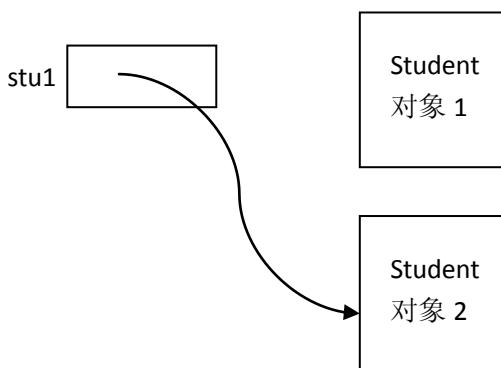
`finalize` 方法有什么特点呢？这个方法会在对象被垃圾回收时由垃圾回收器调用。

如何来理解垃圾回收呢？请看下面的代码：

```
class Student{  
    String name;  
    int age;  
}  
public class TestStudent{  
    public static void main(String args[]){  
        Student stu1 = new Student();  
    }  
}
```

```
stu1 = new Student();  
}  
}
```

在这段代码中，创建了两个不同的 `Student` 对象。内存中的结构如下图：



可以看出，一开始分配的那块内存（`Student` 对象 1），由于之后把 `stu1` 指向了 `Student` 对象 2（创建了一个新对象并把首地址赋值给 `stu1` 引用），因此再也没有引用指向 `Student` 对象 1。因此结果就是，这个对象占据着内存空间，但是没有引用指向这个对象，因此这个对象无法被使用。于是，这种没法使用但又占据内存空间的对象，就被称为垃圾对象。这些垃圾对象占用内存空间，如果一直不处理的话，会造成内存空间的浪费，严重的话会造成程序的崩溃。

那遇到垃圾对象怎么解决呢？在传统的编程语言中，程序员既要负责分配空间，又要负责回收内存资源，这样为程序员编程增加了很大的负担。而在 Java 中，程序员只需要负责分配空间（也就是 `new` 对象），而不需要操心处理垃圾对象的问题。`JVM` 有一个自动垃圾回收的机制，这个机制能够自动回收垃圾对象所占用的内存。这样，程序员就只需要负责分配空间、创建对象而不用担心内存的回收。

垃圾回收机制对于大部分程序员来说，都是天大的好事儿：在家里举办宴会之前，都愿意把家里布置的干干净净漂漂亮亮。但是，当宴会散场之后，收拾屋子、洗碗扫地这些工作，总是让人觉得很麻烦很折腾。而 Java 垃圾回收器就担当了义务的家政服务员。这就是 Java 垃圾回收给程序员带来的福利：程序员大可以自由自在的去使用内存空间，但完全不用关系任何事后收拾的任务。

当 `JVM` 进行一个对象的垃圾回收工作时，会自动调用这个对象的 `finalize` 方法。我们应该如何看待这个 `finalize` 方法呢？这要从垃圾回收的时机说起。

在 `JVM` 的规范中，只规定了 `JVM` 必须要有垃圾回收机制，但是什么时候回收却没有明确说明。也就是说，对象成为了垃圾对象之后，并不一定会马上就被垃圾回收。怎么来理解这个概念呢？

举一个生活中的例子：当顾客去餐厅吃饭，吃完饭之后，桌子上就留下了顾客吃剩的饭菜、汤水以及用过的餐具。这些餐具占用了餐厅的空间，但是却无法重复使用，因此，这些用过的餐具就可以被当做是垃圾对象。在餐厅，顾客不用管收拾桌子和餐具，而由服务员负责回收这些垃圾，并且清理空间，这就是“自动垃圾回收”。

但是服务员回收垃圾有不同的方式。在人潮拥挤的快餐店里，时时会有一个服务员到处在餐厅转悠，一旦有地方产生了垃圾，她会马上进行垃圾回收的工作。

然而，如果是八十年代的一些老饭馆，服务员的服务态度就比较差了。当顾客离开的时候，剩下的垃圾很有可能一直不去收拾，必须要等到新顾客来了以后没地方坐了，服务员才

会去收拾一下。

Sun 公司采用的垃圾回收的方式，是“最少回收”的方式：只有当内存不够的时候才会进行垃圾回收。形象的说，Sun 公司的 JVM，采用的是八十年代老饭馆的方式。这是因为回收垃圾必然需要占用 CPU，最少回收虽然可能会浪费一点空间，但是能够减少垃圾回收的次数，从而降低垃圾回收对 CPU 时间的占用，提高程序的执行效率。

但是“最少回收”的机制下，当对象成为垃圾对象之后，到 JVM 真正回收这个资源，可能之间会有很长的一段时间。

正因为如此，我们不应该在 `finalize` 中写上释放资源的代码。原因就像我们刚刚所说，当一个对象成为垃圾对象以后，可能并没有马上进行垃圾回收。如果把释放资源的代码写在 `finalize` 中，那么从对象成为垃圾对象，到对象真正被垃圾回收的这段时间，资源始终没有释放，这样就会造成资源的浪费。

例如，如果一个类要使用数据库资源，如果把数据库资源的释放写在 `finalize` 里，那么当这个对象成为垃圾，而没有被垃圾回收的时候，这段时间数据库资源始终被这个对象占用，可能就会造成其他对象访问不了数据库。因此，释放资源的代码不应该写在 `finalize` 里，而应该采用别的方式，一旦资源不使用了马上就释放，而不是依赖垃圾回收及 `finalize` 方法。

在 Java 中还有一个 `System.gc()` 方法。调用这个方法，就相当于通知 JVM，程序员希望能够进行垃圾回收。但是调用 `gc` 也不能保证马上就能进行垃圾回收，这一切都要看当时 JVM 的运行状态。举例来说，`System.gc()` 方法就相当于大厨在后灶吆喝：“前面的服务员，赶紧收拾一下桌子，厨房都没有干净盘子用了！”。如果当时服务员正闲着，有可能马上就帮大厨把垃圾回收了，但是如果当时服务员正忙，就有可能忽略大厨的要求，仍然按照自己的意愿，在高兴的时候再进行垃圾回收。

总结一下：`finalize` 方法在对象被垃圾回收的时候调用。但是，由于 Sun 公司的 JVM 采用的是“最少”回收的机制，因此不应当把释放资源的代码写在 `finalize` 方法中。

## 1.2 getClass

`getClass` 方法是 `Object` 类中的一个公开方法，这个方法的作用是：返回对象的实际类型。如何来理解对象的实际类型呢？

考虑如下继承关系：

```
class Animal{}  
class Dog extends Animal{}  
class Courser extends Dog{}
```

在这个继承关系中，`Dog` 类表示狗，继承自 `Animal` 类；`Courser` 类继承自 `Dog` 类，表示猎犬。

考虑下面的逻辑：写一个函数，接受一个 `Animal` 类型的参数，当这个 `Animal` 类型的引用指向一个 `Dog` 类型的对象时，返回 `true`，否则返回 `false`。

这个函数应该如何写呢？根据我们之前学习的多态的内容，可以利用 `instanceof` 操作符来进行判断。示例代码如下：

```
public static boolean isDog(Animal ani){  
    if (ani instanceof Dog){  
        return true;  
    }else return false;
```

```
}
```

这样的代码却有一个问题，当传入的 ani 参数所指向的对象是一个 Courser 对象时，这个函数同样返回 true。因为根据多态，Courser 是 Dog 类的子类，因此当 ani 指向一个 Courser 对象时，返回值也是 true。这与我们的要求并不一致，我们希望的是，仅仅当 ani 参数指向一个实际类型为 Dog 类的对象时才返回 true。

应该怎么解决这个问题呢？针对这个问题，我们可以使用 getClass 方法来解决。getClass 方法能够返回一个对象的实际类型，通过比较两个对象 getClass 的返回值，就能够判断着两个对象是否是同一个类型。例如有如下代码：

```
Dog d1 = new Dog();  
Dog d2 = new Dog();  
Dog d3 = new Courser();  
  
//d1 和 d2 实际类型相同，因此 getClass 方法返回值相同，输出 true  
System.out.println(d1.getClass() == d2.getClass());  
  
//d1 和 d3 实际类型不同，因此 getClass 方法返回值不同，输出 false  
System.out.println(d1.getClass() == d3.getClass());  
  
因此，利用 getClass 方法，就能够避免 instanceof 操作符的麻烦。改写 isDog 方法如下：  
public static boolean isDog(Animal ani){  
    Dog d = new Dog();  
    if (ani.getClass() == d.getClass()) {  
        return true;  
    }else return false;  
}
```

关于 getClass 方法的更多内容，我们将在反射章节中做更加详细的介绍。

### 1.3 equals

equals 方法是 Object 类中定义的方法，其方法签名为：

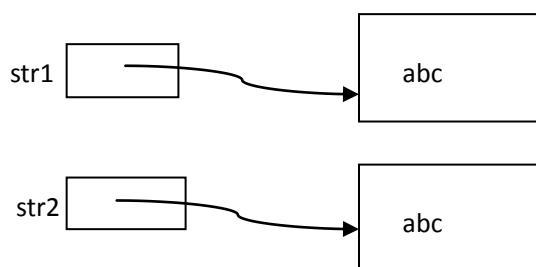
```
public boolean equals(Object obj)
```

由于 equals 是 Object 类中的公开方法，这意味着在 java 中所有对象都包含了 equals 方法。这个方法用来判断两个对象内容是否相等。要注意的是 equals 方法和双等号 “==” 之间的区别。例如，考虑下面的代码：

```
String str1 = new String("abc");  
String str2 = new String("abc");  
System.out.println(str1 == str2);
```

上面的代码，输出结果为 false。原因在于，使用双等号比较两个对象类型，比较的是两个引用中保存的地址，而不是对象的值。

在上面的三行代码执行之后，在内存中的示意图如下：



也就是说，str1 和 str2 这两个引用分别指向两个不同的对象。由于 str1 和 str2 指向不同

的对象，也就意味着 str1 中保存的内存地址和 str2 中保存的内存地址一定不相同，因此，那双等号比较这两个引用，返回值为 false。

但是，从另一个意义上说，有没有办法能够比较两个对象的内容是否相等呢？因为 str1 和 str2 这两个引用所指向的对象都包含字符串“abc”，因此，从对象的内容上来看，str1 和 str2 应该是相等的。

那如何比较两个对象的内容呢？我们可以调用 equals 方法来比较两个对象的内容是否相等。例如下面的代码：

```
String str1 = new String("abc");
String str2 = new String("abc");
//比较两个引用是否指向同一个对象，输出 false
System.out.println(str1 == str2);
//比较两个对象内容是否相等，输出为 true
System.out.println(str1.equals(str2));
```

从上面的例子可知，equals 方法是用来判断对象的内容是否相等。

注意 equals 方法的签名：这个方法接受一个 Object 类型的对象 obj 做为参数。equals 方法比较的就是当前对象（this）和 obj 这两个对象的内容。例如，对于 str1.equals(str2) 这个比较而言，当前对象就是 str1 对象，而 obj 对象就是 str2 对象。

要注意的是，在某些情况下需要程序员覆盖 equals 方法。例如下面的代码：

```
class Student {
    String name;
    int age;
    public Student() {}
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
public class TestStudent{
    public static void main(String args[]){
        Student stu1 = new Student("Tom", 18);
        Student stu2 = new Student("Tom", 18);
        System.out.println(stu1 == stu2);
        System.out.println(stu1.equals(stu2));
    }
}
```

上面这个程序，输出结果为两个 false。第一个 false 好理解，stu1 和 stu2 两个引用分别指向两个不同的对象，两个引用中存放的地址不同，因此返回值为 false。

但是第二个输出语句中，我们调用了 equals 方法来比较两个对象。在这两个学生对象中，两个学生对象的姓名相同，两个学生对象的年龄也相同，但是比较的结果依然是 false。这是怎么回事呢？

在我们的 Student 对象中并没有定义 equals 方法，因此在调用 stu1.equals 方法时，调用的实际上是 Student 类从 Object 类中继承的 equals 方法。那 Object 类中的 equals 方法进行判断时，判断的就是引用是否相等。以下代码来源于 Sun 公司 JDK6 的源码（扩展知识：在你的 JDK 安装目录下，有一个 src.zip 文件，这个压缩包中包含了 JDK 的源码）。下面的代码，

就来源于这个文件中的 `java/lang/Object.java` 文件)

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

可以看到，在 `Object` 类中的 `equals` 方法，使用的是双等号进行的引用的比较，因此，`Object` 类中的 `equals` 方法不能够帮助我们进行两个学生对象是否相等的比较。

为了替换掉 `Object` 类中对 `equals` 的实现，我们应当覆盖 `equals` 方法。也就是说，程序员应当自己来指定两个对象的比较准则。

要注意的是，实现的 `equals` 方法，应当满足以下几个条件：

- 自反性。自反性指的是，如果一个引用 `x!=null`，则 `x.equals(x)` 应当为 `true`。也就是说，无论什么情况，一个对象自己跟自己比较，必须为 `true`。

- 对称性。对称性指的是，如果有两个不为 `null` 的引用 `x` 和 `y`，如果 `x.equals(y)` 为 `true`，则 `y.equals(x)` 也为 `true`；而如果 `x.equals(y)` 为 `false`，则 `y.equals(x)` 也为 `false`。

这一条表明，在使用 `equals` 比较的时候，`x.equals(y)` 和 `y.equals(x)` 的结果是一样的。从理解上来说，可以把这一条当做交换律来理解。

- 传递性。这条准则指的是，如果有三个不为 `null` 的引用 `a`、`b`、`c`，如果 `a.equals(b)` 为 `true`，`b.equals(c)` 为 `true`，则 `a.equals(c)` 也必然为 `true`。

从逻辑上说，如果 `a` 和 `b` 相等，`b` 和 `c` 相等，则 `a` 和 `c` 必然相等。

举一个反例：假设，判断两个学生是否相等的时候，我们制定这样的比较规则：看这两个学生的成绩相差是否在 5 分以内。

这样判断的话，如果学生 A 考了 85 分，学生 B 考了 89 分，则 A 和 B 两个学生相等。如果学生 C 考虑 93 分，则 B 和 C 学生相等。但是， $93-85>5$ ，因此 A 和 C 不相等。这就违反了传递性原则，因此我们制定的比较规则是错误的。

- 一致性。这条准则比较简单，指的是如果有两个不为 `null` 的引用 `x` 和 `y`，在不改变 `x` 和 `y` 的属性的前提下，每次调用 `x.equals(y)` 返回的值都相同。也就是说，如果你不改变 `x` 和 `y` 的属性，则每次比较这两个对象，结果应该一致，不能因为时间的变化等因素而影响比较的结果。

- 如果 `x` 不为 `null`，则 `x.equals(null)` 应当返回 `false`。

根据这些条件，在开发 `equals` 方法时，有一套固定的模式。只要按照这个模式进行开发，就一定能写出满足 `equals` 方法的这几条准则，并且能符合程序员要求的代码。以上文的 `Student` 类为例，`equals` 方法的写法如下：

```
public boolean equals(Object obj) {  
    //判断 obj 是否和 this 相等，保证自反性  
    if (obj == this) return true;  
    //判断 obj 是否为 null，保证最后一条准则  
    if (obj == null) return false;  
    //判断两个对象的实际类型是否相等，  
    //如果不相等，则说明比较的是两个不同种类的对象，应当返回 false  
    if (obj.getClass() != this.getClass()) return false;  
    //强制类型转换  
    //由于之前已经使用 getClass 判断过实际类型，因此这里强转是安全的  
    Student stu = (Student) obj;  
    //判断每个属性是否相等  
    //对于基本类型的属性用 “==” 比较，对象类型的属性用 equals 比较
```

```
        if (this.age == stu.age && this.name.equals(stu.name) )
            return true;
        else return false;
    }
```

总结一下，`equals` 方法的五个步骤：

- 1、判断 `this == obj`
- 2、判断 `obj == null`
- 3、判断两个对象的实际类型（使用 `getClass()`方法）
- 4、强制类型转换
- 5、依次判断两个对象的属性是否相等

## 1.4 `toString`

`toString` 方法也是 `Object` 类中定义的方法，这意味着这个方法是 Java 中所有对象都有的方法。这个方法的签名如下：

```
public String toString()
```

这个方法没有参数，返回值类型是一个 `String` 类型。这个方法的返回值是某个对象的字符串表现形式。如何来理解“字符串表现形式”呢？请看这个例子：

```
class Student{
    String name;
    int age;
    public Student(){}
    public Student(String name, int age){
        this.name = name;
        this.age = age;
    }
}
public class TestToString{
    public static void main(String args[]){
        Student stu = new Student();
        System.out.println(stu.toString());
    }
}
```

在这个例子中，定义了 `Student` 类型，每一个 `Student` 对象都代表了一个学生。为了打印学生的信息，在打印语句中我们可以调用 `toString` 方法，这个方法返回了学生对象的字符串表现形式。也就是说，这个方法返回了学生对象的信息，这些信息组成了表示学生对象的字符串，也就是学生信息的字符串表现形式。

我们也可以这样使用打印语句：

```
System.out.println(stu.toString()); //手动调用 toString 方法
System.out.println(stu); //直接打印 stu 对象
```

上面的两行代码运行结果如下：

```
Student@c17164
Student@c17164
```

可以看出，无论是手动调用 `toString` 方法，还是直接打印 `stu` 对象，所得的结果都是一样的。这意味着，如果打印 `stu` 对象的话，就相当于打印 `stu` 的 `toString` 方法返回值。

另外一方面，打印出的信息是什么意思呢？其中，`Student` 表示的是对象的类名，而后面的`@XXXXX` 表示的是对象相应的内存地址。事实上，这种信息往往不是我们需要的信息，因为我们打印学生对象的信息时，不需要知道其地址，事实上也无法直接操作其地址。

我们调用学生对象的 `toString` 方法，目的是获知学生对象的相关信息，例如学生的姓名和年龄等。而由于在上面的代码中，我们没有为 `Student` 类写 `toString` 方法，这意味着学生对象中的 `toString` 方法是从 `Object` 类中继承来的，因此打印出的是类名以及相关地址。为了让 `toString` 方法能打印出我们想要的内容，我们可以覆盖这个方法如下：

```
public String toString() {
    return name + " " + age;
}

这样，打印学生对象时就会显示相应的学生信息。
完整代码如下：
class Student{
    String name;
    int age;
    public Student(){}
    public Student(String name, int age){
        this.name = name;
        this.age = age;
    }
    public String toString(){
        return name + " " + age;
    }

}
public class TestToString{
    public static void main(String args[]){
        Student stu = new Student("Tom", 18);
        System.out.println(stu.toString());
        System.out.println(stu);
    }
}
```

运行结果如下：

```
Tom 18
Tom 18
```

可以看出，无论用哪种方式，都打印的是 `Student` 类中 `toString()` 方法的返回值。

此外，我们在介绍 `String` 类型时曾经提过，`String` 类型与任何其他类型相加，结果都是 `String` 类型。这其中，“其他类型”既包括简单类型，又包括对象类型。例如下面的代码：

```
Student stu = new Student("Tom", 18);
String str = "my string " + stu;
System.out.println(str);
```

上面的例子中，我们使用一个“`my string`”字符串与一个学生对象相加，结果还是一个字符串。在生成这个字符串的时候，会把一个对象转换成一个字符串，转换的方式，就是调

用这个对象的 `toString` 方法并获取其返回值。上面的字符串加法的代码：

```
String str = "my string " + stu;
```

这句代码表示把“my string”字符串和 `stu` 的 `toString` 方法返回值相加，并把相加的结果赋值给 `str` 变量。

上述代码的运行结果如下：

```
my string Tom 18
```

## 2 包装类

由于 `Object` 类是所有对象的父类，因此 `Object` 类型的引用能够接受 Java 中所有类型的对象。但是，对于基本类型，`Object` 类就无能为力了，毕竟 `Object` 只能处理对象类型。

有没有什么办法，能够让 `Object` 类处理 Java 中所有的数据类型呢？在 Java 中解决这个问题的方法就是使用包装类。

### 2.1 包装类简介

包装类是为了把基本类型包装成对象类型，从而让其也成为对象类型，从而能被 `Object` 类型统一管理。那如何包装呢？下面我们以 `int` 类型的包装类为例，创建一个我们自己的 `int` 类型的包装类。

```
class MyInteger{  
    private int value;  
    public MyInteger(int value){  
        this.value = value;  
    }  
    public int intValue(){  
        return value;  
    }  
}
```

这样，我们定义了一个对象类型 `MyInteger`，这个类型封装了一个 `int` 类型的 `value`，从而把一个简单类型 `int` 类型封装成了一个对象类型。用这个 `MyInteger` 类的对象，同样可以表示一个整数。

事实上，在 Sun 公司的 JDK 中，已经为我们封装好了类似的对象，我们可以直接使用这些对象。简单类型和其对应的包装类类型如下表：

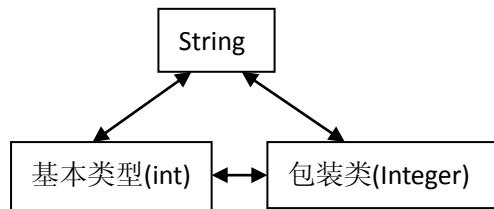
基本类型	包装类
<code>byte</code>	<code>Byte</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>
<code>boolean</code>	<code>Boolean</code>
<code>char</code>	<code>Character</code>

这就是 Sun 公司为我们提供的八种包装类。

## 2.2 六种转换

有了包装类之后，就要研究一下包装类和基本类型之间应该如何转换。除此之外，还有其他类型的转换：字符串和基本类型以及包装类之间的转换。由于在互联网上传输时，大部分情况下不会进行数值的传递，很多情况下传递的是字符串。例如，在一些购物网站上下订单进行消费时，会输入产品数量等信息。这些信息从浏览器传输到服务器时，并不是按照数值的方式进行传递，而是把数值转化成字符串进行网络传输。这样，就要求我们在服务器端完成字符串类型到数值类型的转换。

接下来，我们就为大家介绍三种类型之间的六种转换。所谓的三种类型，指的是字符串类型、基本类型、包装类类型。如下图示：



首先介绍一下包装类和基本类型之间的转换。以 Integer 和 int 为例，由 int 转换为 Integer 的方式，是利用 Integer 的构造方法，以 int 作为参数创建相应的 Integer 对象。如下面代码所示：

```
int i1 = 10;  
//int 类型转化为 Integer 类型  
Integer ii1 = new Integer(i);
```

那如何由 Integer 类型转换为 int 类型呢？我们可以直接调用 Integer 对象的 intValue 方法，用这个方法就可以把 Integer 类型转换为 int 类型。如下面代码所示：

```
int i2 = ii1.intValue();
```

下面是 String 类型与包装类之间的转换。字符串转为包装类类型非常简单，同样是利用包装类的构造方法，例如：

```
String str = "123";  
Integer ii2 = new Integer(str);
```

这样就可以把字符串类型转为相应的包装类类型。而包装类类型转为字符串类型也非常简单，直接调用包装类对象的 toString 方法，就能返回该对象的字符串表现形式，也就是把对象转化为字符串。如下面代码所示：

```
String str2 = ii2.toString();
```

那字符串类型和基本类型之间应该如何转化呢？首先是基本类型转换成字符串。这种转化有很多种不同的转换方式，下面为大家介绍两种不同的方式。

一种方式是使用 String 类中定义的一个静态方法：valueOf。因为这个方法是静态方法，因此可以用 String 类名来直接调用。在 String 类中定义了一系列重载的 valueOf 方法，分别表示把不同的基本类型转换为字符串类型。示例代码如下：

```
int i = 10;  
String str3 = String.valueOf(i);
```

第二种方式，我们可以利用字符串的特性，使用字符串加法来完成基本类型和字符串类型之间的转换。例如：

```
String str4 = "" + i;
```

由于字符串类型加上任何其他类型，结果都是字符串，因此我们可以用一个空字符串加上一个整数，从而把这个整数转化为一个字符串。

那字符串类型如何转化为基本类型呢？还是以 `int` 为例，如果要转换的话，可以调用 `Integer` 类的 `parseInt` 方法，把字符串转化为 `int`。示例代码如下：

```
int i = Integer.parseInt("123");
```

由此，我们就给大家介绍完了三种类型（字符串、基本类型、包装类）之间的相互转化。建议各位读者到这里先暂停一下，练习一下这些转化，试着把字符串、`Double` 和 `double` 类型之间进行六种转换。

## 2.3 JDK5.0 新特性：自动封箱拆箱

在掌握了三种类型、六种转换之后，我们介绍一个 `JDK5.0` 的新特性：自动封箱和拆箱。这个特性的核心就是一句话：在 `JDK5.0` 中，由系统自动完成基本类型和包装类之间的转换。例如，对于 `JDK5.0` 以前的代码，要进行 `int` 和 `Integer` 类型之间的转换，必须要把代码写成下面的形式：

```
//把 int 类型的 10 转换为 Integer 类型  
Integer myInteger = new Integer(10);  
//把 Integer 类型的 myInteger 转换为 int 类型  
int i = myInteger.intValue();
```

而对于 `JDK5.0` 以后的版本来说，下面的代码也能编译通过：

```
//把 int 类型直接赋值给 Integer 类型  
Integer myInteger = 10;  
//把 Integer 类型直接赋值给 int 类型  
int i = myInteger;
```

上面的这两行代码，可以把 `int` 类型的变量直接赋值给 `Integer` 类型，也可以把 `Integer` 类型的变量直接赋值给 `int` 类型。也就是说，可以对 `Integer` 类型和 `int` 类型自动进行转换。

需要注意的是，这个新特性只是语法方面的改变，而底层实现的原理，并没有发生改变。也就是说，从语法上说，

```
Integer myInteger = 10;
```

是把一个 `int` 类型的值直接赋值给一个 `Integer` 类型的值。但是，在 `JDK5.0` 的编译器编译这段代码时，会自动把上面的代码翻译成下面的形式：

```
Integer myInteger = new Integer(10);
```

也就是说，从底层的工作原理上说，与 `JDK5.0` 以前的版本是一样的。只不过 `JDK5.0` 的编译器提供了一种新的语法，能够帮助我们让代码变得更清晰。

同样的，从语法上说，

```
int i = myInteger;
```

是把一个 `Integer` 类型的变量直接赋值给 `int` 类型的变量。但是在底层实现中，`JDK5.0` 的编译器会自动把上面的代码翻译成如下形式：

```
int i = myInteger.intValue();
```

例如，下面的例子：

```
public class TestAutoBoxing {  
    public static void main(String[] args) {  
        Integer myInteger = null;  
        int i = myInteger;  
    }  
}
```

```
}
```

上面的代码在运行时，会产生一个 `NullPointerException` 的异常。为什么会产生这个异常呢？因为

```
int i = myInteger;
```

这行代码，编译器会自动翻译成：

```
int i = myInteger.intValue();
```

在这个过程中，由于调用 `intValue` 方法时，`myInteger` 的值为 `null`，相当于对一个 `null` 值调用了 `intValue()` 方法。因此，这行代码会抛出 `NullPointerException` 这个异常。

自动封箱和拆箱的语法和功能并不复杂，但是有时候，这个特性能够帮我们解决很大的问题。例如，如果我们有一个 `Integer` 类型的值：

```
Integer myInt = new Integer(10);
```

现在的需求是：让 `myInt` 的值加 1。如果不使用自动封箱和拆箱，则代码应当这么写：

```
Integer myInt = new Integer(10);
int t = myInt.intValue();
t++;
myInt = new Integer(t);
```

而如果使用自动封箱的话，只用一句：

```
myInt++;
```

就可以完成上述的功能。这个例子说明，自动封箱和拆箱能够很大程度上，帮助程序员简化代码，让程序的结构更清晰，可读性更强。

## 3 内部类

内部类是 Java 中很特殊的一个语法。一方面，内部类能够一定程度上的减少代码量，并且能够为程序员提供一些语法方面的比较方便的功能。另一方面来说，内部类有可能带来非常古怪的语法，这些语法会造成代码的可读性下降。因此，内部类是一把双刃剑。从规范上说，我们并不提倡程序员过多的使用内部类，不过依然应该了解一些内部类的基本语法和部分内部类的使用方法。

Java 中的内部类分为四种：成员内部类、静态内部类、局部内部类、匿名内部类，我们下面对这四种内部类分别进行介绍。

### 3.1 成员内部类

我们在定义类的时候，会定义成员变量。例如下面的程序：

```
class MyOuterClass{
    private int value;
}
```

其中，`value` 被成为 `MyOuterClass` 的属性，也被称为成员变量。在 `value` 的位置定义一个类，则这个类就被称为成员内部类。如下面代码所示：

```
class MyOuterClass{
    private int value;
    private class InnerClass{
```

```

        public void m(){
            System.out.println(value);
        }
    }
}

```

上面这段代码演示了如何定义一个成员内部类。首先，定义成员内部类的位置，在某一个类的里面，在任何一个函数的外面（也就是定义成员变量的位置）。

其次，由于成员内部类是属于外部类的内部，因此在成员内部类中可以访问外部类的私有成员。例如，上面的程序中，在 `InnerClass` 的 `m` 方法中，就访问了外部类 `MyOuterClass` 的私有属性 `value`。

第三，成员内部类可以作为 `private` 的。之前我们学习修饰符的时候曾经提过，`private` 不能修饰类。但是 `private` 可以修饰成员内部类，这表示只能在 `MyOuterClass` 的内部使用 `InnerClass` 类。

编译上面的程序，会在相应的目录中生成两个.class 文件：一个 `MyOuterClass.class` 文件和一个 `MyOuterClass$InnerClass.class` 文件。这说明，在编译时，内部类也会生成独立的.class 文件，换句话说，内部类也是独立的类。

下面我们来看一下，如何来使用成员内部类创建对象。第一种情况，在外部类中，可以直接创建内部类的对象。（在上面的例子中，指的是在 `MyOuterClass` 类中，可以直接创建内部类的对象）。例如，我们可以为 `MyOuterClass` 添加一个 `f` 方法，在 `f` 方法中创建 `InnerClass` 对象。示例代码如下：

```

class MyOuterClass{
    private int value = 100;
    private class InnerClass{
        public void m(){
            System.out.println(value);
        }
    }
    public void f(){
        InnerClass ic = new InnerClass();
        ic.m();
    }
}

public class TestInnerClass {
    public static void main(String[] args) {
        MyOuterClass moc = new MyOuterClass();
        moc.f();
    }
}

```

在上面的这段程序中，运行结果输出 `100`，也就是 `value` 的值。

第二种情况，在外部类的外部创建对象。以上面的例子来说，也就是在 `MyOuterClass` 类的外部如何创建内部类对象。我们来修改上面的例子，在 `TestInnerClass` 类中创建 `InnerClass` 类型的对象。

首先，为了能够让 `InnerClass` 在 `MyOuterClass` 类的外部创建，因此必须要把 `InnerClass`

的访问权限修改一下，去掉 `private` 修饰符。修改后的 `MyOuterClass` 以及 `InnerClass` 的代码如下：

```
class MyOuterClass{
    private int value = 100;
    //增加一个 value 属性的 get/set 方法
    public void setValue(int value){
        this.value = value;
    }
    public int getValue(){
        return value;
    }
    //去掉 private 修饰符
    class InnerClass{
        public void m(){
            System.out.println(value);
        }
    }
    public void f(){
        InnerClass ic = new InnerClass();
        ic.m();
    }
}
```

修改之后，接下来就要在 `TestInnerClass` 类中创建 `InnerClass` 类型的对象。

首先，内部类对象的类型，需要写成“外部类.内部类”。也就是说，在主方法中首先定义一个变量，代码如下：

```
MyOuterClass.Inner inner;
```

其次，创建一个内部类对象的时候，必须首先创建一个外部类的对象。代码如下：

```
MyOuterClass moc = new MyOuterClass();
```

下面到了最激动人心的时刻：我们要正式创建成员内部类对象了！创建的语法：

```
inner = moc.new InnerClass();
```

你没有看错，在 `new` 关键字前面，是一个 `MyOuterClass` 类型的引用名！这就是最最奇怪的地方。这说明，每一个成员内部类对象都要跟一个外部类对象相关联。

为什么成员内部类对象要跟外部类对象关联呢？考虑这样的问题：在 `InnerClass` 的 `m` 方法中，打印了外部类的 `value` 属性。这个属性是外部类的实例变量，也就是说，必定是某一个外部类对象的属性。假设创建了两个外部类对象：

```
MyOuterClass moc1 = new MyOuterClass();
moc1.setValue(10);
MyOuterClass moc2 = new MyOuterClass();
moc2.setValue(20);
```

这样，就创建了两个对象，这两个对象分别有自己的 `value` 属性。接下来需要创建成员内部类对象。由于内部类对象能够访问 `value` 属性，因此在创建成员内部类对象的时候，必须要说清楚，这个成员内部类对象究竟以那个对象为外部对象，究竟打印的是哪个 `value` 值。

例如，如果有下面的代码：

```
MyOuterClass.InnerClass in1 = moc1.new InnerClass();
```

上面的代码说明了 `in1` 这个对象的外部对象为 `moc1`，因此调用 `in1` 的 `m` 方法时，打印的是 `moc1` 对象的 `value` 值，因此打印的是 10。

完整的 `TestInnerClass` 对象的代码如下：

```
public class TestInnerClass {  
    public static void main(String[] args) {  
        MyOuterClass moc = new MyOuterClass();  
        MyOuterClass.InnerClass ic = moc.new InnerClass();  
        ic.m();  
    }  
}
```

最后要说明的一点，由于成员内部类必须与外部类某一个对象相关联，因此成员内部类中不能定义静态方法。

## 3.2 静态内部类

静态内部类与成员内部类相似，只有一点不同：在定义成员内部类之前，增加一个 `static` 关键字，则定义的内部类就成为一个静态内部类。示例代码如下：

```
class MyOuterClass{  
    //成员内部类  
    class MemberInner{}  
    //静态内部类  
    static class StaticInner{}  
}
```

需要说明的是，`static` 关键字不能够用来修饰类，但是能够用来修饰内部类。

接下来我们看一下静态内部类以及成员内部类对外部类成员的访问。

请看如下代码：

```
class MyOuterClass{  
    int value1 = 100;  
    public void m1() {}  
  
    static int value2 = 200;  
    public static void m2() {}  
  
    class InnerClass1{  
        public void f1(){  
            //成员内部类中能够访问外部类的静态成员以及非静态成员  
            System.out.println(value1);  
            System.out.println(value2);  
            m1();  
            m2();  
        }  
        //!public static void f2() {}  
        //编译出错，成员内部类不能定义静态方法  
    }  
}
```

```

static class InnerClass2{
    public void f1(){
        //静态内部类中只能访问外部类的静态成员
        //System.out.println(value1); 出错!
        System.out.println(value2);
        //! m1();
        m2();
    }
    //静态内部类中可以定义静态方法
    public static void f2(){}
}
}

```

由上面的代码，我们可以得出以下结论：

成员内部类中不能定义静态方法；成员内部类中能够访问外部类的所有静态以及非静态的成员；

静态内部类中可以定义静态方法，静态内部类中只能访问外部类的静态成员（即访问外部类的静态属性或者调用外部类的静态方法。）

至于如何创建静态内部类的对象，也分两种情况

第一种情况，在外部类中，可以直接创建内部类的对象。（在上面的例子中，指的是在 `MyOuterClass` 类中，可以直接创建内部类的对象）。这种情况与成员内部类的情况相同，在此不再赘述。

第二种情况，在外部类的外部创建对象。以上面的例子来说，也就是在 `MyOuterClass` 类的外部如何创建内部类对象。这种情况下，静态内部类比成员内部类要简单的多。直接使用“外部类.内部类”的方式就能够创建。示例代码如下：

```
MyOuterClass.InnerClass2 in2 = new MyOuterClass.InnerClass2();
```

完整的代码如下：

```

class MyOuterClass{
    private int value1 = 100;
    private static int value2 = 200;

    static class InnerClass2{
        public void m(){
            System.out.println(value2);
            m2();
        }
    }

    class InnerClass1{
        public void m(){
            System.out.println(value1);
            System.out.println(value2);
            m1();
        }
    }
}
```

```

        m2();

    }

}

public void m1(){}
public static void m2(){}

public void f1(){
    InnerClass1 in1 = new InnerClass1();
    InnerClass2 in2 = new InnerClass2();
}

}

public class TestInnerClass {

    public static void main(String[] args) {
        //创建成员内部类对象
        MyOuterClass moc = new MyOuterClass();
        MyOuterClass.InnerClass1 incl = moc.new InnerClass1();
        //创建静态内部类对象
        MyOuterClass.InnerClass2 inc2 =
            new MyOuterClass.InnerClass2();
    }
}

```

### 3.3 局部内部类

首先，我们回顾一下局部变量的概念：所谓局部变量，指的是定义在方法内部的变量。局部内部类与这个概念类似：所谓局部内部类，指的是定义在方法内部的类。示例代码如下：

```

class Outer{
    private int value = 100;
    private static int value2 = 200;
    public void m(){
        int localValue = 300;
        class Inner{
            public void f(){
                System.out.println(value);
                System.out.println(value2);
            }
        }
        Inner in = new Inner();
    }
}

```

```
    in.f();
}
}
```

在上面的代码中，在外部类 Outer 的方法 m 内部，我们定义了一个 Inner 类，这个类就是一个局部内部类。

与局部变量一样，局部内部类访问范围就是在方法内部。也就是说，定义的这个 Inner 类只有在 m 方法内部是用，而无法在 m 方法外部使用。

从上面的代码我们还可以看出，局部内部类能够访问外部类的私有属性、静态属性。

除此之外，在 m 方法内部，定义了一个 localValue 变量。这个变量是一个局部变量，其作用范围是在 m 方法内部。然而，Inner 类也在 m 方法内部，那在 Inner 类中能否访问外部类的局部变量呢？

试着修改一下 Inner 类的 f 方法如下：

```
public void f(){
    System.out.println(value);
    System.out.println(value2);
    System.out.println(localValue);
}
```

加上一句输出 localValue 的语句，结果会产生一个编译时错误！

为什么呢？请牢牢记住下面的规则：在局部内部类中能够访问外部类的局部变量，但是要求该变量必须是 final 的！

修改之后的代码如下：

```
class Outer{
    private int value = 100;
    private static int value2 = 200;
    public void m(){
        final int localValue = 300;
        class Inner{
            public void f(){
                System.out.println(value);
                System.out.println(value2);
                System.out.println(localValue);
            }
        }
        Inner in = new Inner();
        in.f();
    }
}
```

这就是局部内部类的语法。

那么局部内部类有什么作用呢？考虑下面的代码：

```
interface Teacher{
    void teach();
}

class Tom implements Teacher{
```

```

public void teach(){
    System.out.println("Tom teach");
}

class Jim implements Teacher{
    public void teach(){
        System.out.println("Jim teach");
    }
}

public class TestTeacher{
    public static void main(String args[]){
        Teacher t = getTeacher(10);
        t.teach();
    }

    public static Teacher getTeacher(int n){
        if (n == 10) return new Tom();
        else return new Jim();
    }
}

```

上面的代码中，我们创建了一个 `Teacher` 接口，这个类型用来表示一个老师。然后，`Tom` 和 `Jim` 可以认为是两个不同的老师，他们都实现了 `Teacher` 接口。在 `getTeacher` 方法内部，根据参数的不同，可以创建 `Tom` 和 `Jim` 对象并作为返回值返回。在主方法中，可以调用 `getTeacher` 方法，从而获得 `Teacher` 类型实现类的对象，之后就可以调用实现类的 `teach` 方法。

上面的代码，由于在主方法中操作的都是接口 `Teacher` 类型，而没有与实现类接触。因此，我们可以认为主方法与 `Teacher` 接口的实现类（也就是 `Tom` 和 `Jim` 这两个类）是弱耦合的关系。

换成生活中的例子，我们可以把主方法当做是求学的学生：学生调用学校的 `getTeacher` 方法，由学校选派一位老师，然后学生调用老师的 `teach` 方法。在这个过程中，学校应该尽量避免让学生直接指定具体的老师，而是通过 `Teacher` 接口，让学生和老师之间形成一个弱耦合的关系。

但是，上面的代码却依然存在着强耦合的可能性。考虑下面这种主方法的写法：

```

public static void main(String args[]){
    Teacher t = new Tom();
    t.teach();
}

```

在这段代码中，创建了一个 `Tom` 对象。明明有 `getTeacher` 方法可以实现弱耦合，但是如果程序员不使用这个方法来获得 `Teacher` 对象，而是自己直接创建一个 `Tom` 对象，代码依然是强耦合的。

现在我们把代码修改一下：我们把 `Tom` 类和 `Jim` 类都放入到 `getTeacher` 方法的内部，这样，就把这两个类变成了局部内部类。修改后的代码如下：

```

interface Teacher{
    void teach();
}

```

```

public class TestTeacher{
    public static void main(String args[]){
        Teacher t = getTeacher(10);
        t.teach();
    }
    public static Teacher getTeacher(int n){
        class Tom implements Teacher{
            public void teach(){
                System.out.println("Tom teach");
            }
        }
        class Jim implements Teacher{
            public void teach(){
                System.out.println("Jim teach");
            }
        }
        if (n == 10) return new Tom();
        else return new Jim();
    }
}

```

这样，在 `getTeacher` 方法的外部，无法访问到 `Tom` 类和 `Jim` 类，也就无法直接创建出 `Teacher` 接口的实现类。也就是说，程序员只能通过调用 `getTeacher` 方法来获得 `Teacher` 对象，而无法直接创建某个实现类的对象。这样，就能够实现“强制弱耦合”，即强制程序员必须要利用 `Teacher` 接口来写程序，从而实现弱耦合。

### 3.4 匿名内部类

匿名内部类是一种特殊的局部内部类。如果一个局部内部类满足这样两个特点：1、该内部类继承自某个类或者实现某个接口；2、该内部类在整个方法中只创建了一个对象。满足这两个特点的局部内部类就能够改写成匿名内部类。

考虑上面 `TestTeacher` 程序中的两个局部内部类 `Jim` 和 `Tom`。这两个类都实现了 `Teacher` 接口，并且在 `getTeacher` 方法中各自只创建了一个对象，因此这两个类能够改写为匿名内部类的形式。下面我们演示一下如何把 `Tom` 类改写成匿名内部类：

```

public static Teacher getTeacher(int n){
    //Jim 依然采用局部内部类的写法
    class Jim implements Teacher{
        public void teach(){
            System.out.println("Jim teach");
        }
    }
    //匿名内部类
    if (n == 10) return new Teacher(){

```

```
public void teach() {
    System.out.println("Tom teach");
}
};

else return new Jim();
}
```

上面这段代码，把 Tom 类修改成了一个匿名内部类。我们分析一下这段代码。首先，`return` 语句返回了一个 `Teacher` 类型的对象。由于 `Teacher` 是一个接口，不能创建对象，因此所谓的 `new Teacher()`，实际上是创建了一个实现 `Teacher` 接口的类的对象。

那创建的对象是什么类型的呢？这个实现类没有名字，这也就是为什么这种语法要叫做“匿名”内部类的原因。而且正因为这个类没有名字，因此没有办法通过 `new` 类名()`l` 的方式创建对象。也就是说，匿名内部类没有名字，并且只能在一次方法调用中创建一个匿名内部类的对象。

那这个匿名内部类是如何实现 `Teacher` 接口的呢？为了实现 `Teacher` 接口，要在一对圆括号`()`后增加一个代码块，在这个代码块中写上对接口的实现。在上面的这个例子中，就是实现了 `Teacher` 接口中的 `teach` 方法。

最后，在这一对花括号`{}`之后，还有一个分号。要注意的是，这个分号并不是匿名内部类的语法特点，这个分号是 `return` 语句的结尾。对比 `else` 后面的那个 `return` 语句：这个语句创建一个 `Jim` 对象，并且以分号结尾；而 `if` 的那个分支中创建了一个匿名内部类对象，最后那个分号也是 `return` 语句的结尾。

综上所述，使用匿名内部类，是一种简便的写法。它将实现接口（或者是创建类）的代码和创建类的对象的代码结合在了一起。形成了 Java 中比较有特色的语法：

```
new 接口名 () { 实现接口的代码 };
```

以上的代码首先用一对`{}`来实现了接口，然后用 `new` 关键字创建出了一个对象。请读者千万注意，这句话创建的绝不是接口的对象（接口是特殊的抽象类，无法创建对象），而创建的是一个实现了接口的，没有名字的内部类的对象！

既然匿名内部类是特殊的局部内部类，因此也具有局部内部类的特点。例如：匿名内部类不仅可以访问外部类的私有成员，还可以访问外部类的局部变量，但是要求这个局部变量被声明为“`final`”。

另外，由于匿名内部类没有明确的给出类的名字，因此，无法在匿名内部类中定义任何构造方法。因为我们知道，构造方法的名字必须和类名相同！

# Chp11 集合框架

## 本章导读

集合框架是 Java 中最重要的一部分内容之一。无论是最基本的 Java SE 应用程序开发，还是企业级的 Java EE 程序开发，集合都是开发过程中常用的部分。

在这一章中，我们会第一次大量查阅 JDK 的 API，会第一次大量的接触 JavaSE 平台的核心组件。从这一章之后，我们对面向对象的语法已经基本掌握，后面的学习主要都是针对 JavaSE 的学习。

集合框架掌握起来需要一定的时间和难度。在学习这部分知识时，需要控制节奏，并配合大量的练习。

### 1 集合的基本概念

首先，什么是集合呢？

集合是一种对象，只不过这种对象的功能，是储存和管理多个对象。例如，我们生活中的“抽屉”对象，抽屉就是用来放东西的，也就是说，“抽屉”这个对象的功能，就是用来储存和管理多个对象的。

那是不是除了集合之外，就没有别的管理多个对象的方式了呢？不是。我们之前学到的数组，就能够完成储存和管理多个对象的功能。

那是用数组管理和储存多个对象，有什么问题呢？

看下面这个需求：

1. 创建一个长度为 3 的字符串数组，在数组中放入 “zhang3”，“li4”，“wang5” 这三个字符串。
2. 在下标为 1 的位置插入 “zhao6” 字符串（这意味着需要进行数组扩容）
3. 删除 “li4” 这个字符串。

上面这段需求，用数组怎么实现呢？

参考实现代码如下：

```
public class TestArray {  
    public static void main(String args[]) {  
        String[] names = new String[3];  
        names[0] = "zhang3";  
        names[1] = "li4";  
        names[2] = "wang5";  
  
        //插入 zhao6 之前需要扩容  
        String[] newNames = new String[names.length * 2];  
        for(int i = 0; i<names.length; i++) {  
            newNames[i] = names[i];  
        }  
        names = newNames;
```

```

    //插入（不使用 for 循环，而直接赋值）
    names[3] = names[2];
    names[2] = names[1];
    names[1] = "zha06";

    //删除（不使用 for 循环）
    names[2] = names[3];
}
}

```

可以看出，用数组也可以实现相应的扩容、插入、删除等操作。但是，是用数组进行这些相关操作，却非常的不方便，需要撰写大量的基础代码。这些代码繁琐、重复，而且容易出错（很有可能产生数组下标越界异常等），有没有办法把程序员从这种繁重的劳动中解放出来呢？

我们可以把数组、以及对数组相关的操作封装在一个类中。例如，我们封装一个 `MyList` 类：

```

public class MyList {
    //data 用来保存数组数据
    private Object[] data;
    //index 保存有效元素的个数
    private int index;
    //初始数组长度是 5，有效元素个数为 0
    public MyList() {
        data = new Object[5];
        index = 0;
    }
    //把 value 元素放到末尾 如果 data 数组已满则自动扩充
    public void add(Object value) {
        ...
    }
    //把 value 元素插入在 pos 位置 如果 data 数组已满则自动扩充
    public void add(int pos, Object value) {
        ...
    }
    //删除 pos 位置的元素
    public void delete(int pos) {
        ...
    }

    //获得下标为 pos 的元素
    public Object get(int pos) {
        ...
    }
}

```

```

    }

    //获得有效元素的个数
    public int size(){
        ...
    }

    //判断数组中是否包含 obj 对象
    //如果存在则返回 true
    //否则返回 false
    public boolean contains(Object obj){
        ...
    }
}

```

注意几个要点。1、为了能够让 `MyList` 类更通用，能够保存 Java 中任何一种对象，我们把数据类型设为 `Object` 类型。由于 `Object` 类型是 Java 中所有类型的父类，因此可以把任何对象都赋值给 `Object` 类型的引用，也就是说，能够把任何一种对象都放入 `Object` 数组。如果遇到基本类型的数据，也能将数据转换为包装类对象，同样可以放入 `Object` 数组。

2、`MyList` 类有一个属性 `index`，这个属性用来保存有效元素的个数。例如，刚开始的时候，创建了一个长度为 5 的 `Object` 类型数组，但是这相当于有 5 个元素的空间。但是刚创建的时候，数组中并没有存入有价值的数据，因此有效元素的个数为 0 个，`index` 值也为 0。再举一个例子，假设在一个长度为 10 的数组中，存入了 10 个元素。之后，调用了一次 `delete` 方法，此时，由于删掉了一个元素，因此有效元素的个数变为 9 个。虽然有效元素的个数变少了，但是数组的长度并没有减小，数组的长度依然是 10。利用 `index` 属性，就可以判断数组是不是已经满了（如果 `index` 等于 `data.length`，则意味着数组满了）。在执行插入操作时，如果发现数组已满，则自动完成数组长度的扩充。

3、`MyList` 中，封装了数据 `data`，并且封装了跟数据相关的操作，例如对数组进行增加、删除和插入等操作。这样，我们就把数组这种数据，和对数组的基本操作封装在了一起，从而再遇到数组的一些插入和删除操作时，可以不用重新实现复杂的数组插入和删除操作，而直接利用 `MyList` 类中封装的函数。这样，通过封装 `MyList` 类，减少了程序员的工作量，也提高了代码的重用性。

例如，利用 `MyList` 改写之前那个数组的练习，代码下可以修改如下：

```

public class TestMyList {
    public static void main(String args[]){
        MyList list = new MyList();
        //初始化
        list.add("zhang3");
        list.add("li4");
        list.add("wang5");

        //插入 zhaο6
        list.add(1, "zhaο6");

        //删除 li4
    }
}

```

```
        list.delete(2);

    }
}
```

可以看到，利用了 list 的 add 和 delete 方法，可以让程序员省略自己实现数组插入删除的麻烦。也就是说，MyList 封装了数组的插入和删除操作，让程序员可以直接调用而不用自己重新实现。

完整的 MyList 代码如下：

```
public class MyList {
    //data 用来保存数组数据
    private Object[] data;
    //index 保存有效元素的个数
    private int index;

    //初始数组长度是 5，有效元素个数为 0
    public MyList() {
        data = new Object[5];
        index = 0;
    }

    //把 value 元素放到末尾
    public void add(Object value) {
        if(index == data.length) this.expand();
        data[index] = value;
        index++;
    }

    //把 value 元素插入在 index 位置
    public void add(int pos, Object value) {
        if(index == data.length) this.expand();
        for(int i = index; i>pos; i--) {
            data[i] = data[i-1];
        }
        data[pos] = value;
        index++;
    }

    //删除 index 位置的元素
    public void delete(int pos) {
        for(int i=pos; i<index-1; i++) {
            data[i] = data[i+1];
        }
        index--;
    }
}
```

```

//获得有效元素的个数
public int size(){
    return index;
}

//判断数组中是否包含 obj 对象
//如果存在则返回 true
//否则返回 false
public boolean contains(Object obj){
    for(int i = 0; i<index; i++) {
        if (data[i].equals(obj)) return true;
    }
    return false;
}

//获得下标为 pos 的元素
public Object get(int pos){
    return data[pos];
}

private void expand(){
    Object[] newArray = new Object[data.length * 2];
    for(int i = 0; i<data.length; i++){
        newArray[i] = data[i];
    }
    data = newArray;
}
}

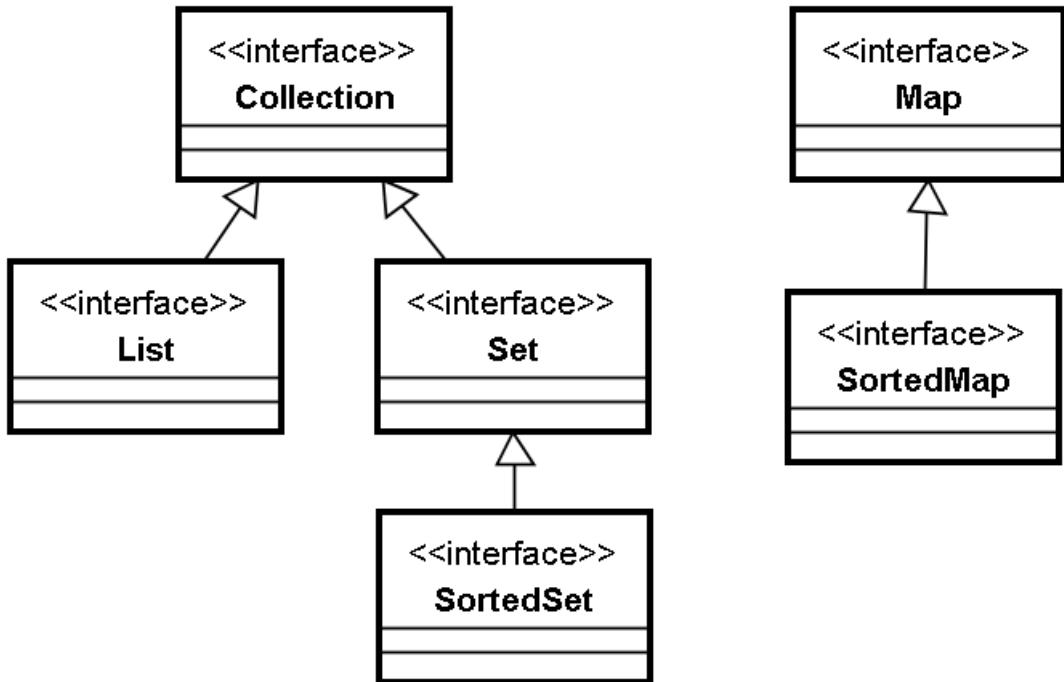
```

## 2 集合框架概览

类似于 `MyList` 这样的类，其实，Sun 公司已经为我们写好了，完全不需要我们自己实现。这就是 Sun 公司提供的集合框架。对于我们来说，重要的不是如何实现这些类，而是如何使用 Sun 公司提供给我们的集合类。

集合就是 Sun 公司为程序员写的很多类。这些类用来储存和管理多个对象。当然，对于管理多个对象来说，管理的方式和特点多种多样。对不同的管理方式会有不同的类来实现。将这些集合类提炼出共性，就能够提炼出很多不同的接口。这些包含着共性的接口，就是我们学习集合重点要掌握的内容。

下面就是 Java 集合框架中，几个主要的接口：



上图是 Java 集合框架中主要的接口。我们分别对每个接口进行描述。

1. Collection 接口。这个接口的特点是：元素是 Object。换而言之，Collection 接口所有的子接口，以及其实现类，所管理的元素单位都是对象。
2. Map 接口。与 Collection 接口对应，Map 接口所管理的元素不是对象，而是“键值对”。什么是键值对呢？“键”和“值”各是一个对象，这两个对象之间，存在着对应的关系，我们可以通过键对象，来找到对应的值对象。在 Map 中，键对象是唯一的，不可重复的，而键对象所对应的值对象是可以重复的。  
例如，每隔四年，都会举办一次世界杯，经过艰苦的捉对厮杀，最终会有一个世界杯冠军产生。这样，举办世界杯的年份，和世界杯冠军，组成了一个对应的关系。上面所说的对应的关系，就是“键值对”的关系。值可以重复，但是键却是唯一的。例如，世界杯举办年份和世界杯冠军的对应关系中，键是世界杯举办的年份，而值是世界杯冠军的获得者。世界杯举办年份这个键不可能重复。例如，2002 年世界杯冠军为巴西队，则“2002—巴西”形成一个键值对的关系。2002 这个键不能够重复，因为 2002 年只有一个世界杯冠军。而巴西队在 1994 年也获得过世界杯冠军，因此“1994—巴西”也形成一个键值对。由此可见，值对象可以重复。
3. Collection 有两个子接口，其中一个子接口为 List 接口。List 接口的特点，是 List 中元素有顺序，可以重复。所谓元素有顺序，指的是说，几个元素放入 List 的先后顺序，就是这几个元素在 List 中的排列顺序。通过集合中元素的顺序，我们可以区分出集合中第 1 个元素，第 2 个元素……
4. Collection 还有一个子接口 Set 接口。Set 接口的特点是元素不可以重复，无顺序。  
例如，在一家饭店中，有“蒸羊羔”、“蒸熊掌”、“蒸鹿尾”三道菜。对于厨师来说，他会做这三道菜，可以认为他会做的菜放在一个 Set 集合中，顾客可以从这个集合中挑选若干道菜。在这个集合中，没有元素重复（不会有厨师跟顾客说，我会做蒸羊羔，还有蒸熊掌，还有蒸羊羔、还有蒸熊掌……），并且元素的顺序也不重要，没有第 1 个第 2 个之分。
5. Set 接口有个子接口 SortedSet。这个接口具有 Set 的特点，其中的元素不能够重复。

但是这个接口与 `SortedSet` 不同的地方在于，这个接口中的元素会按照一定的排序规则，自动对集合中的元素排序。

6. `Map` 有个子接口 `SortedMap`。这个接口与 `Map` 一样，管理的元素是键值对，键不能重复，值可以重复。所不同的是，在这个接口中，键对象会按照一定的排序规则，自动排序。

下面我们就针对这几种接口，分别进行学习和讨论。

要掌握每种集合接口，就要重点掌握集合接口的这几个方面：

- 1、接口的特点
- 2、接口中定义的基本操作
- 3、该集合如何遍历
- 4、接口的不同实现类，以及实现类之间的区别

## 3 Collection

### 1、接口特点

`Collection` 接口的特点是元素是 `Object`。遇到基本类型数据，需要转换为包装类对象。

### 2、基本操作

`Collection` 接口中常用的基本操作罗列如下：

- `boolean add(Object o)`  
这个操作表示把元素加入到集合中。`add` 方法的返回值为 `boolean` 类型。如果元素加入集合成功，则返回 `true`，否则返回 `false`。
- `boolean contains(Object o)`  
这个方法判断集合中是否包含了 `o` 元素。
- `boolean isEmpty()`  
这个方法判断集合是否为空。
- `Iterator iterator()`  
这个方法很重要，可以用来完成集合的迭代遍历操作。具体的介绍会在介绍 `List` 接口如何遍历时再强调
- `boolean remove(Object o)`  
`remove` 方法表示从集合中删除 `o` 元素。返回值表示删除是否成功。
- `void clear()`  
`clear` 方法清空集合。
- `int size()`  
获得集合中元素的个数。

### 3、`Collection` 如何遍历\`Collection` 的实现类

`Collection` 没有直接的实现类。也就是说，某些实现类实现了 `Collection` 接口的子接口，例如 `List`、`Set`，这样能够间接的实现 `Collection` 接口。但是没有一个实现类直接实现了 `Collection` 接口却没有实现其子接口。

正因为如此，`Collection` 如何遍历，我们会在讲解其子接口时详细阐述。

## 4 List

### 4.1 List 特点和基本操作

List 接口的特点：元素是对象，并且元素有顺序，可以重复。

可以把 List 当做一个列表。例如，如果让我们列出历任美国总统，我们必然会按照顺序说出每一任的人名，对于连任的总统，在这个列表中就会出现多次。这样的结构就是一个典型的 List：元素有顺序，并且可以重复出现。

对于 List 而言，元素的所谓“顺序”，指的是每个元素都有下标。因此，List 的基本操作，除了从 Collection 接口中继承来的之外，还有很多跟下标相关的操作。

基本操作罗列如下：

- boolean add(Object o) / void add(int index, Object element)

在 List 接口中有两个重载的 add 方法。第一个 add 方法是从 Collection 接口中继承而来的，表示的是把 o 元素加入到 List 的末尾；第二个 add 方法是 List 接口特有的方法，表示的是把元素 element 插入到集合中 index 下标处。

- Object get(int index) / Object set(int index, Object element)

get 方法获得 List 中下标为 index 的元素，set 方法把 List 中下标为 index 的元素设置为 element。

利用这两个方法，可以对 List 根据相应下标进行读写。

- int indexOf(Object o)

这个方法表示在 List 中进行查找。如果 List 中存在 o 元素，则返回相应的下标。

如果 List 中不存在 o 元素，则返回-1。

这个方法可以用来查找某些元素的下标。

除此之外，List 接口中还有一些诸如 size、clear、isEmpty 等方法，这些方法与介绍 Collection 接口中的相应方法含义相同，在此不再赘述。

### 4.2 遍历

首先，为了能使用 List 接口，必须先简单介绍一个 List 接口的实现类：ArrayList。这个类使用数组作为底层数据结构，实现了 List 接口，我们使用这个类来演示应该如何对 List 进行遍历。

由于 List 接口具有下标，因此我们用类似对数组遍历的方式，采用 for 循环对 List 进行遍历。示例代码如下：

```
public class TestArrayList {
    public static void main(String args[]) {
        List list = new ArrayList();
        list.add("hello");
        list.add("world");
        list.add("java");
        list.add("study");

        for(int i = 0; i<list.size(); i++){
            System.out.println(list.get(i));
        }
    }
}
```

```
    }  
}
```

可以看出，利用 List 接口中的 size 方法，我们可以得出集合中元素的个数，那么集合中元素的下标范围就是 0 ~ size-1。继而我们可以通过 get 方法，根据下标获得相应的元素。利用这种方法，我们就可以遍历一个 List。

除此之外，List 接口还有另外一种遍历方式：迭代遍历。

#### 4.2.1 迭代遍历

迭代遍历是 Java 集合中的一个比较有特色的遍历方式。这种方法被用来遍历 Collection 接口，也就是说，既可以用来遍历 List，也可以用来遍历 Set。

使用迭代遍历时，需要调用集合的 iterator 方法。这个方法在 Collection 接口中定义，也就是说，List 接口也具有 iterator 方法。

这个方法的返回值类型是一个 Iterator 类型的对象。Iterator 是一个接口类型，接口类型没有对象，由此可知，调用 iterator 方法，返回的一定是 Iterator 接口的某个实现类的对象。这是非常典型的把多态、接口用在方法的返回值上面。

Iterator 接口表示的是“迭代器”类型。利用迭代器，我们可以对集合进行遍历，这种遍历方式即称之为迭代遍历。

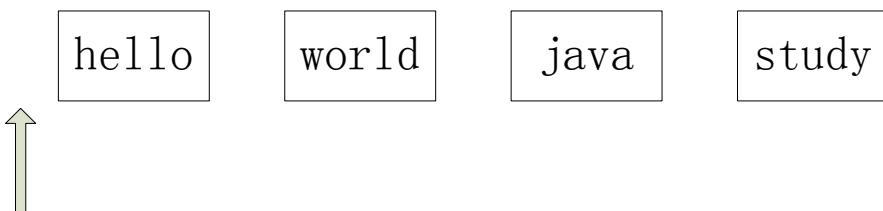
例如，有如下代码：

```
public class TestArrayList {  
    public static void main(String args[]){  
        List list = new ArrayList();  
        list.add("hello");  
        list.add("world");  
        list.add("java");  
        list.add("study");  
  
        Iterator iter = list.iterator();  
    }  
}
```

此时，在集合中有四个元素：hello、world、java、study。示意图如下：



在调用 list 的 iterator 方法之后，返回一个 Iterator 类型的对象。这个对象就好像一个指针，指向第一个元素之前。如下图：



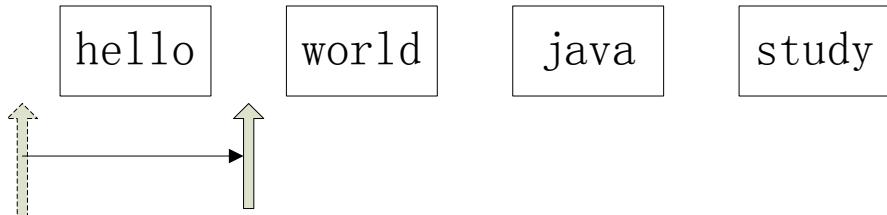
在迭代器中定义了两个方法：

- boolean hasNext()

这个方法返回一个 boolean 类型，表示判断迭代器右方还有没有元素。对于上面这种情况，对迭代器调用 `hasNext()` 方法，返回值为 true。

- `Object next()`

这个方法，会把迭代器向右移动一格。同时，这个方法返回一个对象，这个对象就是迭代器向右移动时，跳过的那个对象。如下图：



调用一次 `next` 方法之后，迭代器向右移动一位。在向右移动的同时，跳过了“hello”这个元素，于是这个元素就作为返回值返回。

我们可以再持续的调用 `next()` 方法，依次返回“world”，“java”，“study”对象，直至 `hasNext()` 方法返回 false，意味着迭代器已经指向了集合的末尾，遍历过程即结束。

利用 `Iterator` 接口以及 `List` 中的 `iterator` 方法，可以对整个 `List` 进行遍历。代码如下：

```
public class TestArrayList {  
    public static void main(String args[]) {  
        List list = new ArrayList();  
        list.add("hello");  
        list.add("world");  
        list.add("java");  
        list.add("study");  
  
        Iterator iter = list.iterator();  
  
        while(iter.hasNext()) {  
            Object value = iter.next();  
            System.out.println(value);  
        }  
    }  
}
```

迭代遍历往往是用 `while` 循环来实现的。利用 `hasNext` 方法返回值作为循环条件，判断 `List` 后面时候是否还有其他元素。在循环体中，调用 `next` 方法，一方面把迭代器向后移动，另外一方面一一返回 `List` 中元素的值。

### 4.3 实现类

`List` 接口有以下几个实现类。要注意的是，这几个类都实现了 `List` 接口，也就是说，如果我们针对 `List` 接口编程的话，使用不同的实现类，编程的方式是一样的。例如，`ArrayList` 和 `LinkedList` 都实现了 `List` 接口，如果我们要把上一小节的程序里的实现由 `ArrayList` 替换成 `LinkedList`，只需要把这一句代码

```
List list = new ArrayList();
```

改为

```
List list = new LinkedList();
```

其余代码由于都是针对 List 接口的，因此完全不需要修改。也就是说，不管实现类是什么样子，对 List 接口的操作是一样的。这也从一个侧面反映了接口的作用：解耦合。

下面针对不同的实现类，分别进行一下介绍。

#### 4.3.1 ArrayList 和 LinkedList

ArrayList 的特点是采用数组实现。很类似于之前我们写的 MyList 类，当然，实际的 ArrayList 类和我们写的 MyList 相比，还是复杂了很多。

用数组这样一种结构来实现 List 接口，具有以下特点：

用数组实现 List 接口，如果要查询 List 中给定下标的元素，只需要使用数组下标就可以直接查到，实现起来非常方便，而且由于数组中，元素的存储空间是连续的，因此通过下标很容易快速对元素进行定位，因此查询效率也很高。

但是，如果使用数组实现 List 接口，则必须要面对一个问题：数组的插入和删除的效率较低。例如，如果要进行数组的插入，有可能要大量移动数组的元素，有可能要进行数组的扩容从而进行大量的内存拷贝的工作。而数组的删除，同样可能意味着要移动大量的数组元素。因此，从这方面来说，数组的插入和删除操作效率比较低。

而 LinkedList 实现 List 接口时，采用的是链表的实现方式。下面简单介绍一下链表这种数据结构。

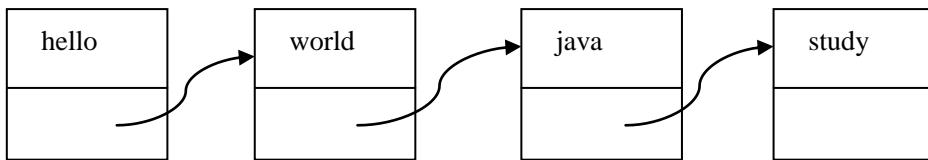
最基本的链表结构是这样的：链表由多个节点组成。每个节点分为两个部分：第一个部分用来储存链表的数据，另一个部分用来储存下一个节点的地址。如何来理解这个问题呢？

有些寻宝和侦探小说里，常常有这样的情节：要打开一个宝藏，需要分散在不同地方的 n 把钥匙。我们伟大的主人公刚出场的时候，往往手上握有一条线索。通过这条线索，能够找到一个装有钥匙的盒子，并且发现，在装着钥匙的装帧精美的盒子中，会发现寻找下一把钥匙需要的线索。就这样历经千辛万苦，最后终于获得宝藏。

在这种情形中，如果我们把钥匙当做数据，那么每个盒子就可以当做一个节点：在节点中，一部分放着数据，另一部分指向下一个节点。也就好像是，盒子中装着钥匙，并且装着发现下一个盒子的线索。我们可以用图来表示这种情况。例如，假设有如下代码：

```
List list = new LinkedList();
list.add("hello");
list.add("world");
list.add("java");
list.add("study");
```

上面的代码创建了一个 LinkedList，在内存中的图像如下：



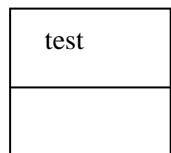
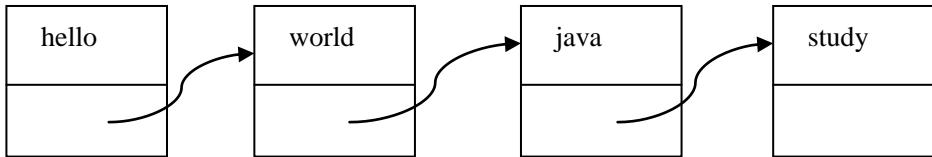
此时，如果调用 get(3)方法，则会由 hello 节点开始，先从 hello 节点找到 world 节点，再从 world 节点找到 java 节点，再从 java 节点找到 study 节点。因此在查询方面，相对于数组直接使用下标，链表实现的 LinkedList，在查询方面效率较低。

而如果要进行插入操作，LinkedList 就会有比较明显的优势：因为 LinkedList 不需要进行数据内容的复制。例如，假设运行了如下代码：

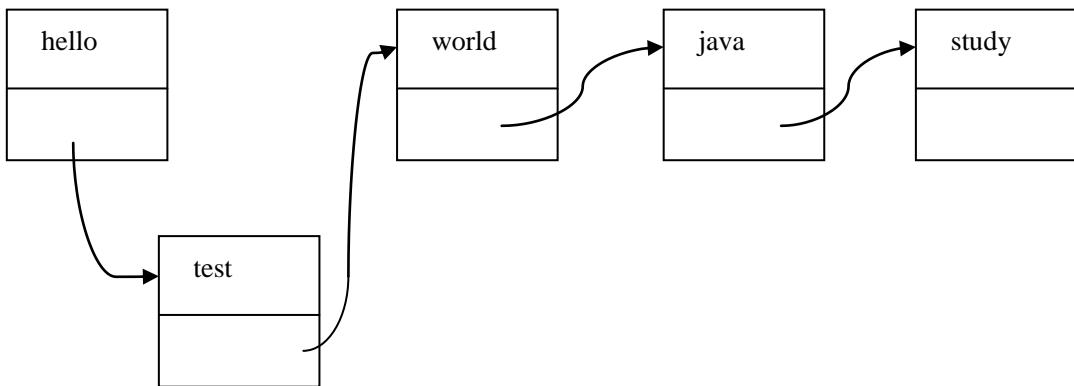
```
list.add(1, "test");
```

则内存中会进行下面的操作：

1. 创建一个新节点。如下图：



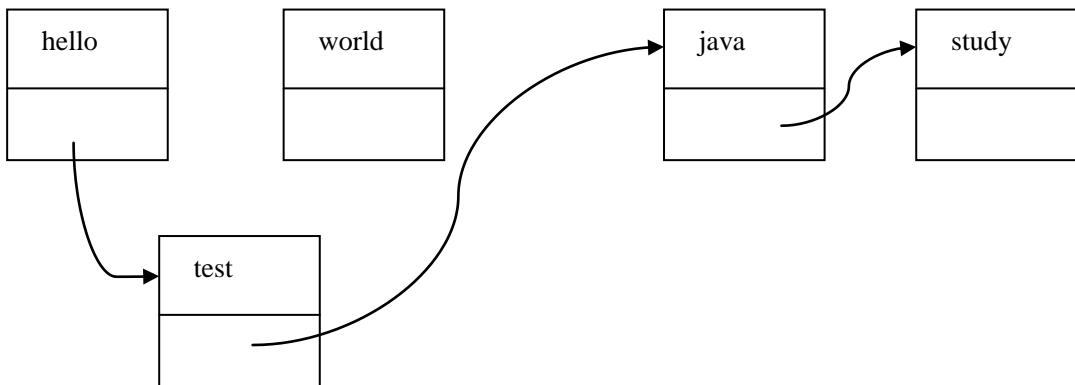
2. 修改 hello 和 test 的指针指向即可。如下图：



与之类似的，使用链表进行删除也需要改动某个指针的指向即可。例如，假设运行了如下代码：

```
list.delete(2);
```

则在内存中，完成的操作如下：



由上面的例子可知，相对使用数组实现 List，使用链表实现 List 中的插入和删除功能，由于没有数组扩容以及移动数据等问题，因此效率要远远高于使用数组实现。

ArrayList 和 LinkedList 之间的区别如下表：

	实现方式	特点
ArrayList	数组实现	增删慢, 查询快
LinkedList	链表实现	增删快, 查询慢

#### 4.3.2 Vector

Vector 是 JDK1.0 遗留下的产物。Vector 同样实现了 List 接口, 而且也是使用数组实现。Vector 和 ArrayList 之间的比较如下:

	实现方式	特点
ArrayList	数组实现	轻量级, 速度快, 线程不安全
Vector	数组实现	重量级, 速度慢, 线程安全

这两个类实现的方式都采用数组实现, 所不同的是, Vector 为了保证线程安全, 采用了重量级的实现方式。在 Vector 中, 所有的方法都被设计为了同步方法。这样, 当多线程共同访问同一个 Vector 对象时, 不会产生同步问题, 但却牺牲了访问的效率。

而 ArrayList 中所有方法并没有被作成同步方法, 因此访问效率较快。当然, 当多线程同时访问同一个 ArrayList 对象时, 可能会造成线程的同步问题。

关于线程的同步, 在本书线程的章节中有更加详细的描述, 请读者参考。

## 5 Set

### 5.1 Set 特点和基本操作

就像之前提到的一样, Set 接口的特点是元素不可以重复, 无顺序。具体例子不再赘述。

那 Set 接口有哪些基本操作呢? Set 接口中所有的操作都继承自 Collection 接口, 也就是说, Set 接口没有自己特有的操作, 其所有操作都来源于父接口 Collection。因此, 它具有 Collection 接口中定义的那些诸如 add、remove 等方法。

特别要注意的是, 由于 Set 集合中的元素没有顺序, 因此 Set 集合中的元素没有下标的概念。因此, 和 List 接口不同, Set 接口中没有定义与下标相关的操作。

Set 接口相关的内容请参考对 Collection 接口的描述。

### 5.2 遍历

与 List 接口一样, 我们先介绍一个 Set 接口的实现类, HashSet。我们利用这个类来测试 Set 接口的遍历。

Set 接口中没有跟下标相关的方法, 也就是说, Set 接口中没有类似 List 接口中的 get 方法, 因此, 无法使用跟下标紧密联系的 for 循环遍历。

但是, Set 接口可以使用迭代遍历。Collection 接口中定义了 iterator 方法, 因此 Set 接口中也包含了这个方法。对于 Set 集合来说 (尤其是 JDK1.5 以前的版本), 只能采用迭代器的方式来遍历。

示例代码如下:

```
public class TestSet {
    public static void main(String[] args) {
        Set set = new HashSet();
        set.add("hello");
        set.add("world");
        set.add("java");
        //加入重复元素是, add 方法会返回 false
```

```
    set.add("hello");

    //迭代遍历
    Iterator iter = set.iterator();
    while(iter.hasNext()) {
        Object value = iter.next();
        System.out.println(value);
    }
}
```

要注意的是，迭代遍历输出的结果为：

```
hello
java
world
```

注意到对 set 调用了两次 add("hello")方法，但是输出结构只有一个 hello 字符串。同时，注意到输出结果的排列顺序与加入 set 的顺序完全无关。这就是 Set 集合的特点：元素无顺序，不可以重复。

### 5.3 实现类

对于 Set 集合的基本操作，相对而言比较容易掌握。对于 Set 接口而言，比较难掌握的地方在于 Set 接口的实现类相关内容。下面这部分内容是学习 Set 接口的重点。

#### 5.3.1 HashSet

HashSet 实现了 Set 接口，因此要求元素不可以重复。那么，HashSet 是怎么来判断元素是否可以重复的呢？

我们首先看下面这个代码的例子：

```
class Student{
    private int age;
    private String name;

    public Student() {
    }

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
```

```

        public String getName() {
            return name;
        }
        public void setName(String name) {
            this.name = name;
        }
        public String toString(){
            return name + " " + age;
        }
    }

    public class TestStudent{
        public static void main(String args[]){
            Set set = new HashSet();
            Student stu1 = new Student("Tom", 18);
            Student stu2 = new Student("Tom", 18);
            set.add(stu1);
            set.add(stu2);
            System.out.println(set.size());
        }
    }
}

```

看上述代码。我们创建了两个 `Student` 对象，这两个对象具有相同的属性。根据 `Set` 接口的含义，`Set` 集合中不应该有内容重复元素，因此我们希望调用了两次 `add` 方法之后，`set` 的长度依然为 1。然而运行结果却是 2。

问题出在哪儿呢？首先来说，要判断两个对象内容是否相等，会调用对象的 `equals` 方法。而 `Student` 类中没有覆盖 `equals` 方法，因此 `Student` 类中的 `equals` 方法来源于 `Object` 类，判断的是引用中保存的地址是否相等。显然，`stu1` 和 `stu2` 这两个引用分别指向了一个 `Student` 对象，这两个对象地址不相同，因此用 `equals` 方法判断，结果为 `false`。

为了能让 `Student` 类能够正确的进行判断，我们应该为 `Student` 类覆盖 `equals` 方法。示例代码如下：

```

public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null) return false;
    if (getClass() != obj.getClass()) return false;
    Student stu = (Student) obj;
    if (this.age == stu.age && this.name.equals(stu.name)) {
        return true;
    } else {
        return false;
    }
}

```

覆盖了 `equals` 方法之后，再次运行。但是，运行结果还是 2！这次问题又出在哪里呢？

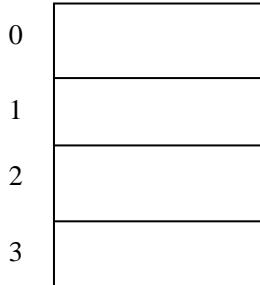
这里面涉及到了 `HashSet` 的实现机制：Hash 算法。下面我们简单的来介绍一下 Hash 算法的原理。

在 Object 类中，有一个 hashCode 方法，这个方法的签名如下：

```
public int hashCode()
```

这是一个 Object 类中定义的公开方法，意味着所有对象中都具有这个方法。这个方法没有参数，返回值为一个 int 类型的数值。

在我们把一个对象放到 HashSet 中时，HashSet 的 add 方法会调用对象的 hashCode 方法。假设，我们的 HashSet 的大小为 4，为这四个位置设置下标为 0~3。内存中情况如下：



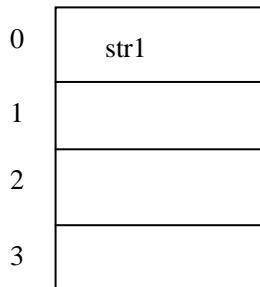
然后，假设调用 add 方法。假设我们有三个对象：str1、str2、str3 三个不同的字符串对象，假设对这三个对象调用 hashCode 方法的返回值为 96、99、100。

调用四次 add 方法如下：

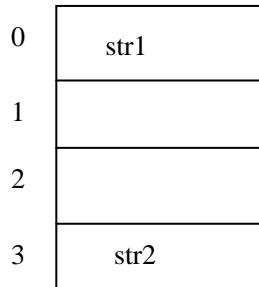
```
set.add(str1);  
set.add(str2);  
set.add(str3);  
set.add(str1);
```

在第一个 add 方法中，会调用 str1 的 hashCode 方法，返回值为 96。str1 对象在 HashSet 中的位置，是根据这个整数 96 对数组长度取模，计算出来的。

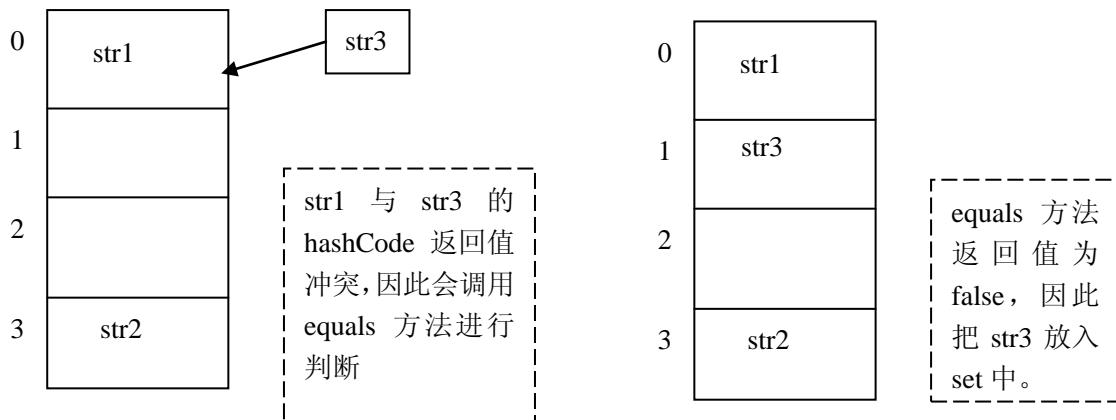
由于  $96 \% 4 = 0$ ，因此会把 str1 放入下标为 0 的位置，如下图：



在第二个 add 方法中，同样会调用 str2 的 hashCode 方法，返回值为 99。由于  $99 \% 4 = 3$ ，因此会把 str2 放入下标为 3 的位置，如下图：

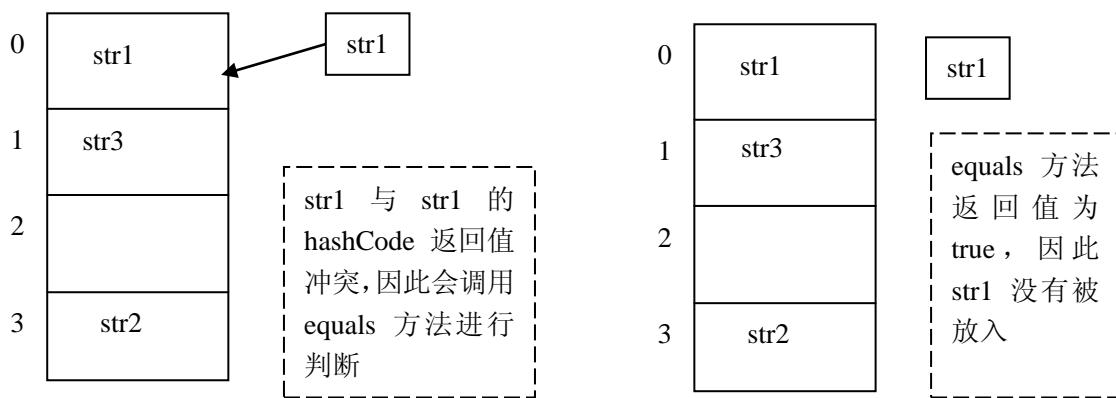


在第三个 add 方法中，会调用 str3 的 hashCode 方法，返回值为 100。由于  $100 \% 4 = 0$ ，但是下标为 0 的位置已经有了一个 str1 元素。此时，就产生了 hashCode 冲突。当产生 hashCode 冲突时，HashSet 会调用 equals 方法进行判断。这是，由于 str1.equals(str3) 返回值为 false，这两个对象的值不相等，因此 str3 同样会被加入到 HashSet 中。示意图如下：



在第四次调用 add 方法时，会再次调用 str1 的 hashCode 方法，返回值为 96。这时，由于  $96 \% 4 = 0$ ，此时产生了 hashCode 冲突。而这时，HashSet 会调用 equals 方法进行判断。由于判断的结果是返回 true，因此 HashSet 认为这是重复元素，从而不会把 str1 对象再次加入 Set，从而避免了重复元素。

示意图如下：



从上述我们对 Hash 算法的描述中，可以看出 HashSet 只有在 hashCode 返回值冲突的时候才会调用 equals 方法进行判断。也就是说，两个对象，如果 hashCode 没有冲突，HashSet 就不会调用 equals 方法判断而直接认为这两个对象是不同的对象。

而对我们自己写的 Student 类调用 hashCode 方法时，由于 Student 类没有覆盖 Object 类中的 hashCode 方法，因此得到的返回值是 Object 类中的 hashCode 方法返回值。参考下面的代码：

```
Student stu1 = new Student("Tom", 18);
Student stu2 = new Student("Tom", 18);
System.out.println(stu1.hashCode());
System.out.println(stu2.hashCode());
```

程序输出结果如下：

```
33263331
6413875
```

可以看出，虽然这两个对象的值相同，并且也覆盖了 equals 方法，但是 hashCode 方法返回值并不相同。这样，HashSet 就认为这两个对象是两个不同的对象，直接把这两个对象放入 HashSet。但是这样一来，就破坏了“Set 中的元素不可重复”这个原则。

那为什么 Student 类有这个问题，而 String 类没有这个问题呢？因为 String 类是 Sun 公司类库的一部分，在 Sun 公司提供 String 类的时候，就为 String 类提供了正确的 hashCode

方法的实现。从而保证了，相同字符串对象，调用 hashCode 方法的返回值都是相同的。

而 Student 类是我们自己写的，这个类中没有覆盖 hashCode 方法，因此调用的 hashCode 方法来源于 Object 类。Object 类中的方法不能够满足我们的要求，无法保证相同的对象返回的 hashCode 相同。

那怎么解决这个问题呢？我们应该从 Student 类本身入手，应该在 Student 类中覆盖 hashCode 方法。例如，在 Student 类中添加如下方法：

```
public int hashCode() {
    return age + name.hashCode();
}
```

这样，就能保证，当两个 Student 对象的 age 和 name 属性的值都相同时，返回的 hashCode 值必定相同。

因此，应当这样覆盖 hashCode：相同对象的 hashCode() 返回值应当相同。

接下来考虑下面的情况。如果我们把 Student 类中的 hashCode 的覆盖写成下面的形式：

```
public int hashCode() { return 0; }
```

这样是否能满足 hashCode 方法的要求呢？

这样的实现，从结果上来说，是对的。由于任何对象返回的 hashCode 值均为 0，因此符合之前所说的：相同对象的 hashCode 相同。

但是这样的实现也有问题：由于任何情况之下返回的 hashCode 值都为 0，因此在往 HashSet 中放入对象时，每次都会产生 hashCode 的冲突，从而每次调用 add 方法都必须要调用 equals 方法比较。而第一个 hashCode 的实现，不同对象造成的 hashCode 冲突的可能性要小得多，因此调用 equals 方法的次数也会少很多。

因此，基于性能方面的考虑，不同的对象 hashCode 返回值应当尽量不同。

总结一下，如果要正常使用 HashSet 存放对象，为了保证对象的内容不重复，则要求这个对象满足：

1. 覆盖 equals 方法。要求相同的对象，调用 equals 方法返回 true。
2. 覆盖 hashCode 方法。要求，相同对象的 hashCode 相同，不同对象的 hashCode 尽量不同。

完整代码如下：

```
import java.util.*;

class Student{
    private String name;
    private int age;

    public Student() {
    }

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }
}
```

```
        }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public int hashCode() {
        return age + name.hashCode();
    }

    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        Student stu = (Student) obj;
        if ( (this.age == stu.age)
            && (this.name.equals(stu.name)) ) {
            return true;
        }else{
            return false;
        }
    }

    public String toString(){
        return name + " " + age;
    }
}

public class TestHashSet {
    public static void main(String[] args) {
        Set set = new HashSet();
        set.add(new Student("Tom", 18));
        set.add(new Student("Jim", 20));
        set.add(new Student("Fred", 22));
        set.add(new Student("Tom", 18));

        Iterator iter = set.iterator();
        while(iter.hasNext()){
            System.out.println(iter.next());
        }
    }
}
```

```
        }
    }
}
```

输出结果如下：

```
Fred 22
```

```
Jim 20
```

```
Tom 18
```

注意，`add()`方法被调用了四次，但是遍历 `set` 集合时，只读到了三个元素。

### 5.3.2 LinkedHashSet

`HashSet` 的特点是元素不可重复且元素无顺序。某些情况下，我们依然需要元素不可以重复，但是希望按照我们加入 `Set` 的先后顺序来加入这些元素。这个时候，我们就可以使用 `LinkedHashSet`。例如下面的例子：

```
import java.util.*;
public class TestLinkedHashSet {
    public static void main(String args[]) {

        Set set = new LinkedHashSet();
        set.add("hello");
        set.add("world");
        set.add("java");
        set.add("hello");

        Iterator iter = set.iterator();
        while(iter.hasNext()){
            System.out.println(iter.next());
        }
    }
}
```

输出结果如下：

```
hello
```

```
world
```

```
java
```

我们可以看到，字符串的打印顺序和它们添加到 `LinkedHashSet` 中的顺序是一致的。同时，`hello` 字符串被添加了两次，但只打印了一次。

要注意的是，如果要使用 `LinkedHashSet` 的话，也必须正确的覆盖对象的 `hashCode` 和 `equals` 方法。

## 6 Map

### 6.1 Map 特点和基本操作

`Map` 接口与 `Collection` 接口不同，这个接口的元素是“键值对”。其中，键值对的特点

是：键可以重复，值不可以重复。

之前解释过，所谓“键值对”，可以理解成一种一一对应的关系。在这种关系中，我们可以通过“键”来找到特定的值。例如，如果把举办世界杯的年份当做键，把该年获得世界杯冠军的球队作为值，则这就形成了一个典型的键值对的关系。在这个关系中，我们可以通过年份查询对应年份的世界杯冠军，这就是“通过键，找到对应的值”的操作；并且举办世界杯的年份不会有重复，而不同年份的世界杯冠军有可能相同，这就对应着“键可以重复，值不可以重复”。

Map 接口中的一些基本操作罗列如下：

- `Object get(Object key)`

这个方法完成的功能是，通过键对象 key，来找到相应的值对象。

- `put(Object key, Object value)`

这个方法是把一个键值对放入 Map 中。如果键不存在，则在 Map 中新增一个键值对。如果键已存在，则把新值替换旧值。例如，有如下代码：

```
System.out.println(map.get("2002"));
map.put("2002", "Brazil");
System.out.println(map.get("2002"));
map.put("2002", "China");
System.out.println(map.get("2002"));
```

在第一个输出语句中，由于 Map 中不存在以 2002 作为键的键值对，因此第一个输出语句输出为 null。

之后，调用 put 方法。此时，由于 Map 中不存在 2002 这个键，因此会在 Map 中增加一个新的键值对。第二个输出语句就会输出 “Brazil”。

之后，再次调用 put 方法。此时，由于在 Map 中 2002 这个键已经存在，因此会用新值 “China” 替换旧值 “Brazil”。于是，第三个输出语句就会输出 “China”。

- `remove(Object key)`

这个方法根据一个键，删除一个键值对。

- `Set keySet()`

这个方法返回所有键的集合。由于在 Map 中，键没有顺序，且不可以重复，因此所有的键对象组成的就是一个 Set。也就是说，keySet 方法返回的是一个 Set，这个 Set 就是所有键对象的集合。

- `Collection values()`

values 方法返回类型是一个 Collection，返回的是所有值对象的集合。

- `containsKey / containsValue`

这两个方法用来判断在 Map 中键是否存在，或者值是否存在。

- `size()`

这个方法返回 Map 中键值对的个数

- `isEmpty()`

判断 Map 是否为空

- `clear()`

清空 Map

- `entrySet`

这个方法返回值类型是一个 Set 集合，集合中放的是 Map.Entry 类型。这个方法是

用来做键值对遍历的，在讲解遍历的时候还会给大家讲到。

## 6.2 遍历

与之前一样，在真正开始讲解遍历之前，首先先使用一个 Map 接口的实现类：HashMap。

创建相应的 HashMap 对象，并放入一些初始值，如下面代码所示：

```
import java.util.*;
public class TestMap {
    public static void main(String args[]) {
        Map map = new HashMap();

        map.put("2006", "Italy");
        map.put("2002", "Brazil");
        map.put("1998", "France");
        map.put("1994", "Brazil");
    }
}
```

在这个 Map 的基础上，我们开始对 Map 进行遍历。

由于 Map 管理的是键值对，因此对于 Map 而言，有多种遍历的方式：键遍历、键值遍历、利用 Map.Entry 进行键值遍历。

### 6.2.1 键遍历与键值遍历

键遍历指的是遍历所有的键。键遍历的实现非常简单：通过调用 Map 接口中的 keySet 方法，就能获得所有键的集合。然后，就可以像遍历普通 Set 一样遍历所有键对象的集合。键遍历参考代码如下：

```
Set set = map.keySet();
Iterator iter = set.iterator();
while(iter.hasNext()) {
    System.out.println(iter.next());
}
```

键遍历输出结果如下：

```
2006
1998
2002
1994
```

可以看到，键遍历输出了集合中所有的键，并且，键并没有顺序。

在键遍历的基础上更进一步，能够遍历所有的键值对。思路如下：

利用键遍历能够遍历所有的键，而在遍历键的时候，可以使用 get 方法，通过键找到对应的值。键值遍历的参考代码如下：

```
Set set = map.keySet();
Iterator iter = set.iterator();
while(iter.hasNext()) {
    Object key = iter.next();
```

```
Object value = map.get(key);
System.out.println(key + "---->" + value);
}
```

键值遍历的结果如下：

```
2006--->Italy
1998--->France
2002--->Brazil
1994--->Brazil
```

可以看到，键值遍历时能够输出键值对这种一一对应的关系。

### 6.2.2 值遍历

除了键遍历以及键值遍历之外，Map 接口还有一种遍历方式：值遍历。值遍历表示的是遍历 Map 中所有的值对象。与键遍历类似，我们对 Map 进行值遍历的思路也很简单：首先利用 Map 的 values() 方法获得 Map 中所有值的集合。需要注意的是，values() 方法返回的是一个 Collection 类型的对象，因此，应当用迭代遍历的方式，遍历这个 Collection。参考代码如下：

```
Collection conn = map.values();
Iterator iter = conn.iterator();
while(iter.hasNext()) {
    System.out.println(iter.next());
}
```

这样，我们就遍历了 Map 中的所有值。输出结果如下：

```
Italy
France
Brazil
Brazil
```

### 6.2.3 利用 Map.Entry 进行遍历

在 Map 接口中，有一个方法叫做 entrySet。这个方法返回一个 Set 集合，这个集合中装的元素的类型是 Map.Entry 类型。

Map.Entry 是 Map 接口的一个内部接口。这个接口封装了 Map 中的一个键值对。在这个接口中，主要定义了这样几个方法：

- getKey(): 获得该键值对中的键
- getValue(): 获得该键值对中的值
- setValue(): 修改键值对中的值

因此，利用 Map.Entry 也可以进行遍历。相关代码如下：

```
Set set = map.entrySet();
Iterator iter = set.iterator();
while(iter.hasNext()) {
    Map.Entry entry = (Map.Entry) iter.next();
    System.out.println(entry.getKey() + "---->" + entry.getValue());
}
```

注意，在赋值的时候，应当把 iter.next 的返回值强转成 Map.Entry 类型才可以。结果如下：

```
2006-->Italy  
1998-->France  
2002-->Brazil  
1994-->Brazil
```

可以看到，用 Map.Entry 进行遍历，以及使用 keySet()方法以及 get()方法进行键值，这两种遍历的结果是一样的。

### 6.3 实现类

Map 接口主要的实现类就是 HashMap 和 LinkedHashMap，此外还有一个使用较少的 Hashtable。

HashMap 的特点是：在判断键是否重复的时候，采用的算法是 Hash 算法，因此要求作为 HashMap 的键的对象，也应该正确覆盖 equals 方法和 hashCode 方法。

LinkedHashMap 和 HashMap 之间的区别有点类似于 LinkedHashSet 和 HashSet 之间的区别：LinkedHashMap 能够保留键值对放入 Map 中的顺序。

例如，如果我们把上一小节的例子中，Map 接口的实现类由 HashMap 改为 LinkedHashMap，修改后的完整的代码如下：

```
import java.util.*;  
public class TestLinkedHashMap {  
    public static void main(String args[]){  
        Map map = new LinkedHashMap();  
        map.put("2002", "Brazil");  
        map.put("1998", "France");  
        map.put("1994", "Brazil");  
        map.put("2006", "Italy");  
  
        Set set = map.keySet();  
        Iterator iter = set.iterator();  
        while(iter.hasNext()){  
            Object key = iter.next();  
            Object value = map.get(key);  
            System.out.println(key + "---->" + value);  
        }  
    }  
}
```

键值遍历之后，输出结果如下：

```
2002-->Brazil  
1998-->France  
1994-->Brazil  
2006-->Italy
```

可以看到，进行键值遍历时，输出的顺序，与我们在 Map 中进行 put 的顺序相同。这就是 LinkedHashMap 的特点，这个类能够保留键值对放入 Map 中的顺序。

Hashtable 也是 Map 接口的一个实现类。HashMap 和 Hashtable 之间的区别罗列如下：

		null 值处理
HashMap	轻量级，速度快，线程不安全	允许 null 作为键/值
Hashtable	重量级，速度慢，线程安全	null 作为键/值时会抛出异常

要注意，HashMap 和 Hashtable 有两方面的区别。一方面，这两个实现类一个是重量级，一个是轻量级。这类似于 ArrayList 和 Vector 的区别，也就是说，Hashtable 中的所有方法都是同步方法，因此是线程安全的。另一方面，在于对 null 值的处理。例如，有如下代码：

```
Map map = new HashMap();
map.put("2010", null);
```

上面的代码创建了一个 HashMap 作为 Map 接口的实现类，并增加了一个“2010→null”的键值对。在这个键值对中，2010 是键，null 作为值。

而如果把实现类改为 Hashtable，则上面的代码会抛出一个异常。也就是说，不允许把 null 作为键值对中的键或者值。

## 7 Comparable 与排序

### 7.1 Collections 类与 Comparable

在 java.util 包中，提供了一个 Collections 的类（注意这个类的名字，与我们的 Collection 接口只相差最后一个字母 s）。这个类中所有的方法都是静态方法，也就是说，Collections 类所有方法都能够通过类名直接调用。这个类为我们提供了很多使用功能，例如：sort 方法，这个方法能够对 List 进行排序。再例如，max 方法可以找到集合中的最大值，而 min 方法可以找到集合中的最小值，等等。

调用 Collections.sort 方法，可以对一个 List 进行排序。与在接口那一章中讲述的一样，要想对 List 进行排序，就要求 List 中的对象实现 Comparable 接口。具体应该如何实现请参考“接口”的章节。

### 7.2 TreeSet 与 TreeMap

Set 接口有一个子接口：SortedSet，这个接口的特点是：元素不可重复，且经过排序。这个接口有个典型的实现类：TreeSet。要注意的是，因为 TreeSet 中要对对象进行排序，因此要求放入 TreeSet 接口中的对象都必须实现 Comparable 接口。

例如，在讲述接口的知识时，我们曾经介绍过如何实现 Comparable 接口。我们利用 Student 类实现 Comparable 接口。在实现 Comparable 接口时，必须要实现 compareTo 方法，表示对学生进行排序。排序时，我们按照如下排序规则：对学生的年龄进行排序，年龄较小的学生排前面，年龄较大的学生排后面。Student 类的代码如下：

```
class Student implements Comparable<Student>{
    int age;
    String name;
    public Student() {
    }
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

```

    }

    public int compareTo(Student stu) {
        if (this.age > stu.age) {
            return 1;
        } else if (this.age < stu.age) {
            return -1;
        } else {
            return 0;
        }
    }
}

```

然后，就可以把学生对象放入 TreeSet 中。代码如下：

```

public class TestTreeSet {
    public static void main(String args[]) {
        Set set = new TreeSet();
        set.add(new Student("Tom", 18));
        set.add(new Student("Jim", 17));
        set.add(new Student("Jerry", 20));

        Iterator iter = set.iterator();
        while(iter.hasNext()) {
            Student stu = (Student) iter.next();
            System.out.println(stu.name + " " + stu.age);
        }
    }
}

```

输出结果如下：

```

Jim 17
Tom 18
Jerry 20

```

可以看出，遍历输出时，按照年龄的顺序，从小到大依次输出。这说明了在 TreeMap 内部进行排序时，使用了我们定义的 compareTo 方法比较两个元素的大小。

特别要注意的一点，TreeSet 进行比较时，会把利用 compareTo 方法比较时，返回值为 0 的两个对象，当做是相同对象。例如下面的代码：

```

set.add(new Student("Tom", 18));
set.add(new Student("Jim", 18));

```

上面的两行代码，在 set 中放入了两个对象。这两个对象用 compareTo 进行比较时，由于两个对象的 age 属性相同，因此会返回 0。则 TreeSet 认为这两个元素是相同元素，所以在 TreeSet 中只保留了一个对象。

完整代码如下：

```

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

```

```

Set set = new TreeSet();
set.add(new Student("Tom", 18));
set.add(new Student("Jim", 18));

Iterator iter = set.iterator();
while(iter.hasNext()) {
    Student stu = (Student) iter.next();
    System.out.println(stu.name + " " + stu.age);
}
}

```

输出结果如下：

```
Tom 18
```

可以看到，虽然调用了两次 add 方法，但两个元素被认为是相同元素，因此在 TreeSet 中只有一个元素。

与之类似的，Map 接口有一个子接口：SortedMap。这个接口的特点是：对 Map 的键进行了排序。这个接口的典型实现类是 TreeMap，如果要把某个键值对放入 TreeMap，则要求键对象必须实现 Comparable 接口。

我们把前一小节中世界杯的例子，使用 TreeMap 来改写。要注意的是，在这个例子中，Map 的键是 String 类型，这个类型是由 Sun 公司提供的，已经实现了 Comparable 接口。

代码如下：

```

import java.util.*;
public class TestTreeMap {
    public static void main(String args[]) {
        Map map = new TreeMap();
        map.put("2002", "Brazil");
        map.put("1998", "France");
        map.put("1994", "Brazil");
        map.put("2006", "Italy");

        Set set = map.keySet();
        Iterator iter = set.iterator();
        while(iter.hasNext()) {
            Object key = iter.next();
            Object value = map.get(key);
            System.out.println(key + "--->" + value);
        }
    }
}

```

运行结果如下：

```
1994--->Brazil
1998--->France
2002--->Brazil
```

2006--->Italy

可以看到，遍历 Map 时，对键进行了排序，按照键对象的排序结果，依次输出 Map 中的键值对。

## 8 5.0 新特性：foreach 循环

在 JDK5.0 中，Sun 公司对集合框架部分进行了比较大的修改和调整，为集合框架增加了很多新的特性。首先介绍一下一个简单的新特性：foreach 循环。

foreach 循环主要解决的是遍历的问题。对于 List 来说，我们可以采用 for 循环遍历，而对于 Set 而言，我们只能采用迭代遍历。相对 for 循环遍历而言，迭代遍历的代码比较繁琐和复杂。例如，对于一个 Set 而言，采用迭代遍历的代码如下：

```
Iterator iter = set.iterator();
while(iter.hasNext()){
    Object value = iter.next();
    System.out.println(value);
}
```

为了简化遍历的代码，在 5.0 中引入了 foreach 循环。基本语法如下：

```
for(变量 : 集合){
    循环体;
}
```

这段代码表示，遍历整个集合，每次迭代时都把集合中的一个元素赋值给 foreach 循环中的变量，并执行循环体。

例如，上面采用迭代遍历的代码，可以写成：

```
for(Object value : set){
    System.out.println(value);
}
```

这段代码表示，遍历 set 集合，在每次迭代时把 set 集合的元素赋值给 value 变量。可以看出，使用 foreach 循环遍历，语法比迭代遍历要简洁的多。

实际上，foreach 循环遍历和迭代遍历是完全等价的。在写代码时，如果程序员使用了 foreach 循环的语法，那么 5.0 的编译器会把 foreach 循环自动的翻译成对应的迭代遍历。

那么什么样的集合能够用 foreach 循环来遍历呢？在 Java5.0 中，只要是实现了 java.lang.Iterable 接口的集合对象，都能使用 foreach 方式来遍历。前面介绍的所有 Collection 接口的实现类，都符合这个特点。

此外，foreach 循环还能够用来遍历数组。例如下面的代码：

```
String[] ss = new String[]{"hello", "world", "java"};
for(String obj : ss){
    System.out.println(obj);
}
```

上面的代码演示了如何使用 foreach 循环来遍历数组。

## 9 5.0 新特性：泛型

泛型本身是一个非常大的话题，要彻底的掌握以及用好泛型，并不是一朝一夕的事情。在本书中，我们会介绍泛型最常用、最需要掌握的概念和语法。

在正式介绍泛型知识之前，我们首先谈一下 Java5.0 以前集合框架的缺点。在 5.0 以前，`ArrayList` 这个类相比直接使用数组来管理多个对象而言，具有很多优势，例如方法更多，使用更加方便等等。但是 `ArrayList` 也有缺点。例如，如果定义三个类：`Animal`、`Dog` 和 `Cat` 类如下：

```
abstract class Animal{
    abstract public void eat();
}

class Dog extends Animal{
    public void eat(){
        System.out.println("dog eat bones");
    }
    public void bark(){
        System.out.println("wang");
    }
}

class Cat extends Animal{
    public void eat(){
        System.out.println("Cat eat fish");
    }
    public void miaow(){
        System.out.println("miaow");
    }
}
```

如果创建一个 `Dog` 类型的数组，并且对每个对象调用 `bark` 方法，代码如下：

```
Dog[] dogs = new Dog[2];
dogs[0] = new Dog();
dogs[1] = new Dog();
for(int i = 0; i<dogs.length; i++){
    dogs[i].bark();
}
```

由于这是一个 `Dog` 数组，因此数组中的每一个元素都是 `Dog` 类型，从而可以直接把数组中的元素赋值给一个 `Dog` 类型的变量。而对这个变量，则可以直接调用 `bark` 方法。

而如果不是一个数组，使用一个 `ArrayList`，同样对 `ArrayList` 中每个对象调用 `bark` 方法，则代码如下：

```
List list = new ArrayList();
list.add(new Dog());
list.add(new Dog());
list.add(new Dog());
for(int i = 0; i<list.size(); i++){
    Dog d = (Dog) list.get(i);
    d.bark();
}
```

由于 ArrayList 的 get 方法返回的是一个 Object 类型，为了调用 bark 方法，还必须进行强制类型转换。这就是集合不如数组的第一个地方：存入集合的是 Dog 类型，而取出来的时候则变成了 Object 类型，需要进行强制类型转换。

此外，如果在 list 中调用了下面的代码：

```
list.add(new Cat());
```

这个程序在编译时没有错误，但是在运行时会产生一个 ClassCastException。这是集合不如数组的第二个地方：使用集合时，由于一个集合中能够装多个类型的对象，因此在使用时很有可能发生类型转换异常。

为什么会有这两个问题呢？原因很简单：ArrayList 为了设计的更加通用，其内部保存数据的时候，保存数据的时候都采用的是 Object 类型。因为只有设计成这样，ArrayList 才能用来保存所有的 Java 对象。但是，如果把 ArrayList 设计成这样的话，就会造成之前的两个问题：我们无法象定义某个类型的数组那样，定义一个专门用于存放某个类型对象的集合。因此，我们称传统的集合对象是：类型不安全的。

5.0 引入的泛型机制能够很好的解决我们上面所说的问题。

## 9.1 泛型的基本使用

使用 5.0 的泛型机制非常简单。例如，我们希望创建一个只能用来保存 Dog 对象的 List，可以使用如下代码：

```
List<Dog> list = new ArrayList<Dog>();
```

注意，跟原有的代码相比，在 List 接口和 ArrayList 后面，都有个后缀：<Dog>。这表明，创建的 ArrayList 只能够放置 Dog 类型。此时，如果对 list 放入非 Dog 类型对象，则会产生一个编译错误，示例如下：

```
list.add(new Dog()); //OK  
list.add(new Cat()); //!编译错误
```

这样，就能保证 ArrayList 中所有的对象都是 Dog 类型的对象。于是，get 方法返回值就可以确定，一定是 Dog 类型，也就省略了强制类型转换的步骤。所以原有的代码就可以修改成：

```
for(int i = 0; i<list.size(); i++) {  
    Dog d = list.get(i);  
    d.bark();  
}
```

注意到在调用 get 方法的时候，没有进行强制类型转换。

现在我们更仔细的探讨一下上面的这段程序。在 5.0 以后的 Java 文档中，List 接口定义为：List<E>，其中，E 就表示 List 的泛型。

上面的代码中，我们定义 list 变量的类型为 List<Dog>，这就意味着，我们把 E 类型设置为 Dog 类型。在 List 中定义的 get 方法，其声明为：

```
E get(int index)
```

其返回值类型为 E。由于我们设置了 E 为 Dog 类型，因此，我们对我们定义的 list 变量调用 get 方法，其返回值类型为 Dog 类型。

对 Set 集合使用泛型的方法，和 List 接口使用泛型的方法类似，在此不再赘述。

此外，考虑 foreach 循环。如果不使用泛型的话，foreach 循环每次迭代的时候，只能确定元素是 Object 类型，因此 foreach 循环只能写成：

```
for(Object obj : list){
```

```
...
```

```
}
```

而是用了泛型以后，由于能够确定集合中元素的类型，因此 foreach 循环可以写成：

```
for(E e : list) {
```

```
...
```

```
}
```

例如，上面遍历包含 Dog 的 list 的代码，就可以修改成：

```
for(Dog d : list) {  
    d.bark();  
}
```

可以看出，泛型与 foreach 循环结合，大大简化了遍历集合的代码。

除了可以对 List 和 Set 使用泛型，对 Map 类型也可以使用泛型。但是要注意的是，Map 由于管理的是键值对，键有一个类型，值也有一个类型，因此 Map 的泛型需要有两个参数。示例代码如下：

```
Map<Integer, String> map = new HashMap<Integer, String>();  
map.put(2002, "Brazil");  
map.put(1998, "France");  
Set<Integer> set = map.keySet();  
for(Integer i : set){  
    System.out.println(i + "→" + map.get(i));  
}
```

请注意，map 的 keySet 方法返回值为一个 Set<Integer>类型。另外，上述代码中，map 对象的键类型被设置为 Integer，而我们在调用 put 方法的时候采用了 2002, 1998 这样的 int 数据，根据 JDK5.0 中自动封箱的语法，int 类型的数据会被自动封装为 Integer 类型的对象。

## 9.2 泛型与多态

请看下面的例子：

```
List<Dog> dogList = new ArrayList<Dog>();  
List<Animal> aniList = dogList; //! 编译出错!
```

这两行代码中，第一行编译正确，第二行编译出错。

在第一行代码中，把一个 ArrayList<Dog>直接赋值给一个 List<Dog>类型的引用。在这个赋值过程中，类型里有多态（把 ArrayList 赋值给 List），但是泛型是一样的（均是 Dog 的泛型）。

在第二行代码中，把一个 List<Dog>赋值给一个 List<Animal>，这样赋值是错误的！在这个过程中，类型中没有多态（List 是相同的），而泛型有多态（一个是 Dog 的泛型，一个是 Animal 的泛型）。这句话会导致一个编译错误！

请记住这个结论：类型可以有多态，但是泛型不能够有多态！

为什么会有这么个结论呢？假设可以把 dogList 直接赋值给 aniList，则可以对 aniList 调用 add 方法：

```
aniList.add(new Cat());
```

这句代码能够编译通过，因为根据泛型，add 方法接受一个 Animal 类型的参数，而 Cat 对象能够当做一个 Animal 对象。

而事实上，对象是被添加到了 dogList 当中，而这个 list 中只能存放 Dog 对象，不能存

放 Cat 对象，这就出现了前后矛盾的问题。

为了避免这样的问题出现，因此，在 java 中，泛型不能有多态。不同泛型的引用之间不能相互赋值。

这个结论可以扩展到把泛型用在函数参数上。例如，写一个 playWithDog 方法如下：

```
public static void playWithDog(List<Dog> dogs) {  
    for(Dog d : dogs) {  
        d.bark();  
    }  
}
```

这个函数接受一个参数，这个参数的类型是 List<Dog> 类型。根据我们的结论，类型可以有多态，因此我们可以给这个函数传递一个 ArrayList<Dog> 类型的对象作为参数，也可以给这个函数传递一个 LinkedList<Dog> 类型的对象作为参数。但是，泛型上没有多态。假设有一个类 Courser 表示猎狗：

```
class Courser extends Dog{}
```

在传递参数给 playWithDog 的时候，不能够传递一个 ArrayList<Courser> 类型的对象作为实参。这同样是因为，不同泛型的引用之间不能相互赋值。

### 9.3 自定义泛型化类型

现在我们定义一个类，用来表示“一对”这个概念。如果不用泛型的话，示例代码如下：

```
class Pair{  
    private Object valueA;  
    private Object valueB;  
  
    public Object getValueA() {  
  
        return valueA;  
    }  
  
    public void setValueA(Object valueA) {  
        this.valueA = valueA;  
    }  
  
    public Object getValueB() {  
        return valueB;  
    }  
  
    public void setValueB(Object valueB) {  
        this.valueB = valueB;  
    }  
  
}  
  
public class TestPair {
```

```

public static void main(String[] args) {
    Pair p = new Pair();
    p.setValueA(new Dog());
    p.setValueB(new Dog());
}
}

```

可以看到，Pair 类为了尽可能通用，使用了 Object 类型来保存一对值。但是这样就会有类型方面的问题，例如：

```

p.setValueA(new Dog());
p.setValueB(new Cat());

```

这样，这个代码就把一只猫和一条狗硬生生配成了一对，显然，我相信无论是猫还是狗都不会愿意的。

为此，我们应该考虑让我们的 Pair 类具有更加安全的类型，即：要求对 Pair 类来说，valueA 属性和 valueB 属性具有相同的类型。为此，我们可以为 Pair 类使用泛型。

首先修改 Pair 类的定义：

```
class Pair<T>
```

在 Pair 类后面，写一对尖括号，表明 Pair 类要使用泛型。在尖括号中的大写字母 T 称为泛型参数，T 用来标识 Pair 的泛型类型。

然后，把 valueA 和 valueB 的声明也进行修改：

```

private T valueA;
private T valueB;

```

表明 valueA 和 valueB 属性为 T 类型。

最后，把所有的方法也进行修改：

```

public T getValueA() {
    return valueA;
}

public void setValueA(T valueA) {
    this.valueA = valueA;
}

public T getValueB() {
    return valueB;
}

public void setValueB(T valueB) {
    this.valueB = valueB;
}

```

注意，set 方法的参数以及 get 方法的返回值类型，都为 T 类型。这样，使用泛型的 Pair 类就定义好了。在使用的时候，我们就可以指定泛型：

```
Pair<Dog> p= new Pair<Dog>();
```

这就指定了，对于 p 对象来说，泛型 T 被赋值为 Dog 类型。在代码中凡是出现 “T”的地方，都会被 “Dog” 所取代。

此时，如果对 p 调用 set 方法而给出一个不是 Dog 的类型，则会编译出错。例如：

```
p.setValueA(new Dog()); //OK  
p.setValueB(new Cat()); //编译出错!
```

完整的代码如下：

```
class Pair<T>{  
    private T valueA;  
    private T valueB;  
  
    public T getValueA() {  
        return valueA;  
    }  
  
    public void setValueA(T valueA) {  
        this.valueA = valueA;  
    }  
  
    public T getValueB() {  
        return valueB;  
    }  
  
    public void setValueB(T valueB) {  
        this.valueB = valueB;  
    }  
  
}  
  
public class TestPair {  
    public static void main(String[] args){  
        Pair<Dog> p = new Pair<Dog>();  
        p.setValueA(new Dog());  
        p.setValueB(new Dog());  
        // p.setValueA(new Cat()); 编译出错!  
    }  
}
```

# Chp12 异常处理

## 本章导读

异常处理，处理的是程序的错误。对于我们来说，程序出错并不是最可怕的事情，最可怕的是程序出错为用户带来损失。例如，如果我们去 ATM 机上取钱，假设输入密码和金额之后，ATM 机没有吐出钱来，而是程序崩溃了。这个时候，我们关心的不是程序是否还能正常运行，这台机器有多大的几率能够修复。我们关心的是，我们的账户上的余额是否被修改了，关心的是程序崩溃了是否给我们带来了损失。

因此，异常处理不是为了让程序不出错，而是为了一旦程序出错，能够有一个相关的机制让程序执行一些代码来减少损失。这些代码是事先写好的，只有在错误发生的时候才会运行。就好像生活中的医院：开设医院并不能阻止人们生病，而是在人们生病之后，能有一个地方处理人的病情，通过各种手段来让人们恢复健康从而减少健康方面的损失。

## 1 异常的概念和分类

首先我们来介绍一下 Java 中所有错误的分类。在面向对象的概念中，一个错误也是一个对象，犯了一个错误，也就是创建了一个错误对象。在 Java 中，有一个 `java.lang.Throwable` 类，这个类是所有错误的父类。Java 中所有的错误类都是 `Throwable` 的子类。

`Throwable` 有两个子类，一个叫做 `Error`，一个叫做 `Exception`。其中，`Error` 指的是非常严重的错误。这种错误往往来源自 Java 底层，一旦发生这种错误，我们连减少损失的机会都没有。例如，虚拟机运行时崩溃，这种错误就是非常典型的 `Error`。因为虚拟机一旦崩溃，我们完全没有办法再执行任何代码，因此也没有任何机会来做一些减少损失的操作。这就好比当一个人如果生病了，可以去医院看病，从而减少损失。但是如果这个人死了，那无论做什么，都已经没有挽回损失的余地了。因此，对于这种严重的底层错误，我们的态度是：不做处理。并不是我们不想对这些错误做处理，而是我们根本没有机会对这种严重的底层错误进行处理。

相对于 `Error` 来说，`Exception` 就是指的，还不那么严重，有挽回余地的错误，这个单词被翻译成“异常”，在 Java 中的异常处理，指的就是 `Exception` 的处理。对于 `Exception` 而言，这个类有很多很多的子类，其中有一个类叫做 `RuntimeException`，这个类也有很多的子类。这样，所有 `Exception` 的子类就被 `RuntimeException` 分为两大部分：一种是 `Exception` 的子类，但不是 `RuntimeException` 的子类，被称为“已检查异常”；另一种是 `Exception` 子类但不是 `RuntimeException` 的子类，被称为“未检查异常”。

如果拿到一个异常类，如何分辨其是已检查异常还是未检查异常呢？只要看这个类的继承体系：如果这个类的直接或者间接父类中有 `RuntimeException`，则这个类是一个未检查异常；如果没有的话，则这个类是一个已检查异常。

那这两种异常有什么区别呢？所谓的“未检查异常”，指的是在写程序过程中可以避免的异常。可以这么来理解，之所以发生“未检查异常”，原因就是程序员写完程序以后没有好好检查；如果程序员能够好好检查自己的代码，则这些异常都可以避免发生。下面我们就为大家介绍一些常见的未检查异常。

```
import java.util.Scanner;
```

```

public class TestRuntimeException {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int a = sc.nextInt();
        int b = sc.nextInt();
        System.out.println(a/b);
    }
}

```

这段代码读入两个整数，输出他们的商。乍看之下，这段代码没什么问题，但是这段代码有一个隐患：当读入的整数 **b** 为 0 时，这段代码会输出什么？

当 **b** 为 0 时，这段代码就会产生一个异常，异常信息如下：

```

Exception in thread "main" java.lang.ArithmetricException: / by zero
at TestRuntimeException.main(TestRuntimeException.java:7)

```

这段代码里面有几个重要的信息。一个，是产生的异常类的名字，当除数为 0 时，会产生一个 **ArithmetricException** 的异常，产生这个异常的原因则写在类名后面：“/ by zero”。另外，显示产生这个异常的原因代码位置是 **TestRuntimeException.java** 文件的第 7 行。

在上面的代码中，**ArithmetricException** 异常是一个非常典型的未检查异常，完全可以通过编程时检查代码，通过各种手段加以避免。例如，我们可以在输出语句之前加一句判断：

```
if (b != 0) System.out.println(a/b);
```

这样，就能避免我们的输出语句产生异常。

因此，产生的 **ArithmetricException** 就是一个未检查异常，通过分析这个类的父类，我们同样可以得出这个结论。通过查阅 API 文档，我们发现这个类的父类是：**RuntimeExcepiton**。

## java.lang 类 ArithmetricException

```

java.lang.Object
  ↘ java.lang.Throwable
    ↘ java.lang.Exception
      ↘ java.lang.RuntimeException
        ↘ java.lang.ArithmetricException

```

除了 **ArithmetricException** 之外，还有一些其他异常的例子，例如下面的代码：

```
int[] n = new int[3];
n[3] = 10;
```

这段代码会产生一个 **ArrayIndexOutOfBoundsException**，也就是典型的数组下标越界异常。这个异常同样是一个未检查异常，可以通过细心的编写数组代码来避免出现这样的异常。

再例如，下面的代码：

```

class Animal{}
class Dog extends Animal{}
class Cat extends Animal{}
public class TestAnimal{
    public static void main(String args[]){
        Animal a = new Dog();
        Cat c = (Cat) a
    }
}
```

```
}
```

上面的代码会在运行时抛出一个 `ClassCastException`。这个异常也是一个未检查异常，原因是类型转换失败。我们可以通过在强制类型转换之前调用 `instanceof` 判断，从而来避免产生这样的类型转换异常。

最后，还有一个常见的未检查异常：`NullPointerException`。这个异常在我们对 `null` 引用使用属性或者调用方法时产生。这个异常也能够通过仔细编写代码来避免。

对于未检查异常，我们都可以通过仔细检查，在程序中加入适当的 `if` 语句，从而避免这种异常。因此，对于这个异常来说，系统并不要求我们必须处理它们。再好的异常处理，也不如不让异常发生，所谓“防范胜于救灾”，就是这个道理。

未检查异常的对应的是已检查异常。所谓的“已检查异常”，指的是一类无法避免的异常。可以这么来记忆：“已检查异常”是程序员已经仔细把代码检查过了之后，依然会发生的异常。这种异常因为程序员无法避免，因此必须要处理。如果一个程序可能发生已检查异常，而程序中缺少处理异常的代码，那么编译时我们会得到编译错误。

例如，假设程序员在 `Java` 中写了一段用来连接网络服务器的程序。对于程序员来说，无论在编程方面多么小心，都可能面临这样的异常：网络不通。一个 `java` 程序员有再大的本事，也无法保证运行程序时网络一定是通的。因此，在这种情况下，程序员必须要处理这个异常，也就是说，程序员必须要写下一个预案，向 `JVM` 说明：如果网络不通，程序应当如何处理。当网络不通时，系统会抛出 `java.net.SocketException`。通过查阅 API 文档可知，这个类是 `Exception` 的子类，但不是 `RuntimeException` 的子类，因此是已检查异常。

`java.net`

## 类 `SocketException`

`java.lang.Object`

  └ `java.lang.Throwable`

    └ `java.lang.Exception`

      └ `java.io.IOException`

       └ `java.net.SocketException`

举个生活中的例子。地震，就是一个典型的已检查异常。地震这种异常，任何人都无法预防。那各地的政府会做什么呢？一方面是根据当地实际情况，用来设定当地建筑物的抗震标准，这样一旦发生地震时，建筑物能够在一定程度上抵抗这个异常，从而让人有逃生的机会，减少损失；一方面是由地震监测部门，监测并预报地震的发生，另一方面，是一旦发生地震之后，进行救灾和援助的组织工作。以上工作皆可认为是政府部门对地震这种已检查异常的处理。

而生活中未检查异常的例子，比较典型的是吸烟引起的火灾。根据资料统计，人在劳累时在床上或者沙发上抽烟，因为烟头点燃纺织物而造成的火灾，是产生火灾的重要原因之一。由于这种原因而产生的异常（火灾），就是一种完全能够避免的异常。这种异常就属于未检查异常。也许我们的家中很少会事先配备灭火器，来及时扑灭抽烟引起的大火。取而代之的，是加强防范措施，来避免这种火灾的发生。

对于已检查异常来说，由于无法避免，因此应该做好充分的预案，因此已检查异常必须要处理；对于未检查异常来说，因为这种异常可以避免，因此可处理可不处理，在实际编程过程中，未检查异常应当以避免为主。

下面是对 `Java` 中异常分类的总结：

`Throwable` 类：所有错误的父类

```
|-- Error: 严重的底层错误，无法处理  
|-- Exception：异常，异常处理的主要对象  
    |-- RuntimeException 的子类：未检查异常，可以避免，可处理可不处理  
    |-- 非 RuntimeException 的子类：已检查异常，无法避免，必须处理
```

## 2 异常对象的产生和传递

看下面的代码：

```
import java.util.Scanner;  
public class TestException{  
    public static void main(String[] args){  
        Scanner sc = new Scanner(System.in);  
        int i = sc.nextInt();  
        System.out.println("main 1");  
        ma(i);  
        System.out.println("main 2");  
    }  
    static void ma(int i) {  
        System.out.println("ma 1");  
        mb(i);  
        System.out.println("ma 2");  
    }  
    static void mb(int i){  
        System.out.println("mb 1");  
        mc(i);  
        System.out.println("mb 2");  
    }  
    static void mc(int i){  
        System.out.println("mc 1");  
        if (i==0) throw new NullPointerException();  
        System.out.println("mc 2");  
    }  
}
```

在这段代码中，主方法调用 `ma`，`ma` 调用 `mb`，`mb` 调用 `mc`。注意 `mc` 方法，当 `i==0` 时这个语句的语法：

```
throw new NullPointerException();
```

这是一个 `throw` 语句，`throw` 语句表明要抛出一个错误。要注意的是 `throw` 语句的语法：

```
throw + Throwable 对象
```

也就是说，`throw` 语句后面跟的是一个对象，这个对象代表了发生的错误。例如，在 `mc` 方法中，我们创建了一个 `NullPointerException` 对象，并把这个对象抛出。`throw` 语句的作用类似于 `return` 语句，表示将一个异常对象作为方法的返回值返回。

需要注意的是，如果我们没有使用 `throw`，而仅仅是创建了一个对象，就只是在 JVM 中分配了一块内存空间而已，和创建一个普通对象一样。而只有使用了 `throw`，才能表明真正抛出异常对象。

由于 `NullPointerException` 是一个未检查异常，因此可处理可不处理。在这个程序中，我们没有对 `NullPointerException` 进行了处理。

当程序正常时（既在命令行上读入的整数不为 0 时），输出如下：

```
main 1  
ma 1  
mb 1  
mc 1  
mc 2  
mb 2  
ma 2  
main 2
```

上面的代码很容易理解，主方法输出 `main1`，然后调用 `ma` 方法；`ma` 输出 `ma1`，然后调用 `mb` 方法；`mb` 方法输出 `mb1`，然后调用 `mc` 方法；`mc` 方法输出 `mc1` 和 `mc2`；返回 `mb`；`mb` 方法输出 `mb2`，返回 `ma`；`ma` 方法输出 `ma2`，返回主方法；最后主方法输出 `main2`。

当我们在命令行上输入 0 时，运行结果如下：

```
main 1  
ma 1  
mb 1  
mc 1  
Exception in thread "main" java.lang.NullPointerException  
at TestException.mc(TestException.java:23)  
at TestException.mb(TestException.java:18)  
at TestException.ma(TestException.java:13)  
at TestException.main(TestException.java:8)
```

上面的运行结果是怎么来的呢？首先依然是 `main → ma → mb → mc` 这个调用过程。然后，在 `mc` 方法中产生了一个异常对象，并且向上抛出。`mc` 方法产生异常之后，后面的正常代码就不执行了，程序从 `throw` 语句处把异常向上抛出。`mc` 方法抛出异常之后，这个异常对象就到了 `mc` 的调用者：`mb` 方法中，即在 `mb` 方法的

```
mc(i);
```

这个语句处产生一个异常对象。对于 `mb` 方法而言，也就是调用 `mc` 方法时产生了一个异常。由于 `mb` 方法中没有任何处理异常的代码，因此 `mb` 方法后面的代码也停止执行，而直接把这个异常抛出，向上抛给了 `ma` 方法。`ma` 方法获得了这个异常之后，也没有处理，抛给了 `main` 方法，`main` 方法也没有处理，于是这个异常就抛到了 JVM 中。JVM 获得这个异常之后，会打印这个异常的信息，并且让程序终止。

也就是说，方法调用时，是 `main→ma→mb→mc` 的链状结构，这叫做方法调用链。而当产生异常的时候，函数的代码会在产生异常的地方终止，然后把异常对象返回给函数的调用者。对于我们这个例子来说，产生异常之后，异常传递的方式是 `main←ma←mb←mc`。这个结论就是说：当函数产生并抛出一个异常时，异常会沿着方法调用链反向传递。

### 3 异常对象的处理

在上一小节代码的基础上，我们修改一下，让 `mc` 方法有更多的选择：

```
static void mc(int i){  
    System.out.println("mc 1");  
    if (i==0) throw new NullPointerException();  
    if (i==1) throw new java.io.FileNotFoundException();  
    if (i==2) throw new java.io.EOFException();  
    if (i==3) throw new java.sql.SQLException();  
    System.out.println("mc 2");  
}
```

首先我们来看一下这几个异常类。我们可以查看 JDK 文档，在文档中显示的 `FileNotFoundException` 和 `EOFException` 这两个类的继承关系如下：

`FileNotFoundException`:

**java.io**  
**Class FileNotFoundException**

```
java.lang.Object  
└ java.lang.Throwable  
    └ java.lang.Exception  
        └ java.io.IOException  
            └ java.io.FileNotFoundException
```

`EOFException`:

**java.io**  
**Class EOFException**

```
java.lang.Object  
└ java.lang.Throwable  
    └ java.lang.Exception  
        └ java.io.IOException  
            └ java.io.EOFException
```

可以看出，这两个异常都是 `IOException` 的子类，并且，由于继承树中不存在 `RuntimeException` 这个类，因此这两个类都是已检查异常。

而下面是 `java.sql.SQLException` 的继承关系：

`SQLException`:

**java.sql**  
**Class SQLException**

```
java.lang.Object  
└ java.lang.Throwable  
    └ java.lang.Exception  
        └ java.sql.SQLException
```

从它的继承关系可以看出，这个类同样是一个已检查异常。

在 `mc` 方法抛出 `NullPointerException` 的时候，由于这个异常是一个未检查异常，可处理可不处理，因此我们的程序中没有任何处理 `NullPointerException` 的代码。而对于后面的代码，由于抛出的三个异常都是已检查异常，因此必须要处理。如果不处理的话，就会产生编译时错误，信息如下：

```
D:\book>javac TestException.java
TestException.java:23: 未报告的异常 java.io.FileNotFoundException; 必须对其进行
捕捉或声明以便抛出
        if (i==1) throw new java.io.FileNotFoundException();
                           ^
TestException.java:24: 未报告的异常 java.io.EOFException; 必须对其进行捕捉或声明
以便抛出
        if (i==2) throw new java.io.EOFException();
                           ^
TestException.java:25: 未报告的异常 java.sql.SQLException; 必须对其进行捕捉或声
明以便抛出
        if (i==3) throw new java.sql.SQLException();
                           ^
3 错误
```

接下来我们要介绍的是，如何处理异常。

### 3.1 throws 声明抛出异常

让我们想想这样的场景，JVM 和 main、ma、mb、mc 这四个方法坐在一起开会。首先，JVM 问 mc：

JVM：你可能会抛出 NullPointerException？你打算怎么处理？

mc：...，我没有任何要说的，要注意，它是未检查异常，我有不处理它的权利！

JVM：OK，那好，那你还可能会抛出 FileNotFoundException, EOFException, SQLException，关于这些异常，你打算怎么办？

mc：我不知道，让我想一下……

JVM：你不知道？？！！这些都是已检查异常，如果你不处理的话，我想编译器都不会让你过关的！

为了解决这个尴尬的局面，mc 方法决定回答 JVM。

mc：你说的那些异常，我作为一个小小的被调用的方法，实在是处理不了。如果发生这些问题的话，我会向上级领导反映……

mc 这种处理异常的方式，体现在代码上，就是下面的 throws 语句：

```
import java.io.*;
import java.sql.*;
...
static void mc(int i)
    throws FileNotFoundException, EOFException, SQLException{
    System.out.println("mc 1");
    if (i==0) throw new NullPointerException();
    if (i==1) throw new FileNotFoundException();
    if (i==2) throw new EOFException();
    if (i==3) throw new SQLException();
    System.out.println("mc 2");
}
```

在方法的参数表最后，写上“throws”加上一系列异常的名字，表示声明抛出。如果要抛出多个异常的话，多个异常之间用逗号隔开。

注意，一定要区分“throws”和“throw”这两个关键字。throw 是一个动作，如果执行时遇到 throw，这表明在这个语句的地方真的会抛出一个异常。例如，在 mc 方法内部的

`throw`, 每个 `throw` 都会真的抛出一个异常。而 `throws` 表示的是一个声明, 这个声明表示这个方法有可能抛出异常。例如, `mc` 方法声明 `throws FileNotFoundException, EOFException, SQLException`, 这表明调用 `mc` 方法有可能会抛出这些异常。也可以这么理解: 如果你要调用 `mc` 方法的话, `mc` 方法告诉其他函数: 调用我可以, 但是我可能会出“`FileNotFoundException, EOFException, SQLException`”这些错, 如果出了这些错, 我不会管, 而由调用我的函数负责。

简单来说: `throw` 是一个动作, 表示抛出; 而 `throws` 是一个声明, 表示本方法一旦发生这些异常, 本方法不作处理, 异常由调用这个方法的方法来处理。

OK, 回到 JVM 和四个方法的会议。`mc` 声明了这些异常它不管, 接下来 JVM 就开始问另一个方法: `mb`。如果 `mb` 不处理 `mc` 声明抛出的异常, 则会产生编译时的错误。

JVM : `mb, mc` 说他要 `throws` 那三个异常。他说调用他的函数要对那三个异常负责。所以, 现在我问你, 你对这三个异常打算怎么处理?

`mb`: 首先我想说, `FileNotFoundException` 和 `EOFException`, 这都属于 `IOException`。只要是 `IOException`, 我就不管。

JVM: .....好吧, 那 `SQLException` 呢?

`mb`: 我同样不想管。

这段对话翻译成代码如下:

```
static void mb(int i) throws IOException, SQLException{
    System.out.println("mb 1");
    mc(i);
    System.out.println("mb 2");
}
```

`mb` 方法声明抛出 `IOException` 和 `SQLException`。需要注意的是, 由于 `FileNotFoundException` 和 `EOFException` 与 `IOException` 有父子类的关系, 因此声明抛出 `IOException`, 就包含了声明抛出所有 `IOException` 的子类。这是把多态用在声明抛出异常上面。

到了这一步, `mb` 同样把异常往上抛: `mb` 声明抛出 `IOException` 和 `SQLException`。作为调用 `mb` 的方法, `ma` 难辞其咎。于是 JVM 与 `ma` 有了下面的对话:

JVM : `ma, ma!` 醒醒! 我们正在开会呢!

`ma` : 啊? 哦.....你们刚刚说了什么?

JVM: `mb` 方法说他不处理 `IOException` 和 `SQLException`, 你作为他的调用者, 你应该.....

`ma`: 啥都别说了, 有问题, 找我的调用者! 我接着睡觉去了.....

JVM: .....

把这段场景翻译成代码如下:

```
static void ma(int i) throws Exception{
    System.out.println("ma 1");
    mb(i);
    System.out.println("ma 2");
}
```

也就是说，只要是异常，`ma` 方法都会往上抛出！

终于到了最后，JVM 面对着 `main` 方法……

JVM : `main` 方法，我们相处这么多年了，我启动以后第一个寻找的就是你，你不会让我失望吧……

`main`: 唉，大哥不好当啊，小弟如果出了问题，我也很难办……

JVM: 你的意思是？

`main`: 如果出了问题，我也拦不住，所以……

JVM: 难道……

是的，在主方法后面，同样可以加上 `throws` 语句。换句话说，主方法同样可以抛出异常！

JVM: 我没的问了，所有的方法都选择了逃避。那么一旦 `mc` 方法抛出了异常，`mc` 推给 `mb`，`mb` 推给 `ma`，`ma` 推给 `main`，`main` 方法又推给了我，那么我，只好选择停止这个程序的运行了。

完整的代码如下：

```
import java.util.*;
import java.io.*;
import java.sql.*;
public class TestException{
    public static void main(String[] args) throws Exception{
        Scanner sc = new Scanner(System.in);
        int i = sc.nextInt();
        System.out.println("main 1");
        ma(i);
        System.out.println("main 2");
    }
    static void ma(int i) throws Exception{
        System.out.println("ma 1");
        mb(i);
        System.out.println("ma 2");
    }
    static void mb(int i) throws IOException, SQLException{
        System.out.println("mb 1");
        mc(i);
        System.out.println("mb 2");
    }
    static void mc(int i) throws FileNotFoundException,
EOFException, SQLException{
        System.out.println("mc 1");
        if (i==0) throw new NullPointerException();
        if (i==1) throw new java.io.FileNotFoundException();
        if (i==2) throw new java.io.EOFException();
        if (i==3) throw new java.sql.SQLException();
        System.out.println("mc 2");
    }
}
```

```
    }  
}
```

当我们在命令行上输入 1 时，运行结果如下：

```
main 1  
ma 1  
mb 1  
mc 1  
Exception in thread "main" java.io.FileNotFoundException  
  at TestException.mc(TestException.java:24)  
  at TestException.mb(TestException.java:18)  
  at TestException.ma(TestException.java:13)  
  at TestException.main(TestException.java:8)
```

与抛出 `NullPointerException` 的情况类似，异常对象由 `mc` 方法抛出后，被逐层传递至 JVM，最终导致程序的中止运行。

通过 `throws` 关键字，函数可以声明抛出异常。虽然这样处理异常比较“消极”，但是同样是一种处理方式。

虽然我们在任何一个函数后面都没有声明抛出 `NullPointerException`，但是由于这个异常是未检查异常，因此任何一个函数都可以向外抛出这个异常。换句话说，我们可以认为每个函数都默认 `throws RuntimeException`。

### 3.2 try-catch 捕获异常

除了通过 `throws` 进行消极处理之外，我们同样可以进行一些积极处理。我们可以采用 `try-catch` 的语法来捕获可能抛出的异常。例如，现在 `ma` 方法不向抛出异常，而决定自己来捕获和处理 `mb` 有可能发生的异常。`try-catch` 的语法结构如下：

```
try{  
    可能发生异常的代码;  
}catch(捕获的异常类型 1 e){  
    异常处理 1...  
}catch(捕获的异常类型 2 e){  
    异常处理 2...  
} catch(捕获的异常类型 n e){  
    异常处理 n...  
}
```

如果在 `try` 语句块中没有出现异常，那么任何一个 `catch` 语句块都不会得到执行。但一旦 `try` 块中出现了异常，程序的流程会马上跳出 `try` 块，根据异常类型的不同，进入某一个 `catch` 语句块。随后，这个异常被宣告处理完毕，程序将继续向下正常运行。

例如，我们可以把 `ma` 方法按照上面的语法结构改成如下形式：

```
static void ma(int i) {  
    try{  
        System.out.println("ma 1");  
        mb(i);  
        System.out.println("ma 2");  
    }
```

```

} catch (IOException ioe) {
    System.out.println("IOException");
} catch (SQLException sqle) {
    System.out.println("SQLException");
} catch (Exception e) {
    System.out.println("Exception");
}
}

```

注意上面的代码，我们把 `ma` 方法签名后面的 `throws Exception` 声明给去掉了，并把 `ma` 方法中原有的三行代码都放入了 `try` 块中。由于这其中 `mb` 方法有可能抛出异常，我们这么做也就是把可能抛出异常的代码放入了 `try` 块中。

在 `try` 块之后，是三个 `catch` 子句。这三个 `catch` 语句分别捕获不同的异常类型，三个分别为 `IOException`、`SQLException` 和 `Exception` 类型。`catch` 语句表示，当 `try` 块中的代码抛出异常的时候，会根据抛出异常类型的不同，而进行对应的不同的处理。例如，当 `mb` 方法抛出 `SQLException` 时，程序会进入 `SQLException` 的 `catch` 语句块。

当我们在 `ma` 方法中处理完异常之后，就可以把 `main` 方法中的 `throws Exception` 语句给去掉。完整代码如下：

```

import java.util.*;
import java.io.*;
import java.sql.*;
public class TestException{
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int i = sc.nextInt();
        System.out.println("main 1");
        ma(i);
        System.out.println("main 2");
    }
    static void ma(int i) {
        try{
            System.out.println("ma 1");
            mb(i);
            System.out.println("ma 2");
        } catch (IOException ioe) {
            System.out.println("IOException");
        } catch (SQLException sqle) {
            System.out.println("SQLException");
        } catch (Exception e) {
            System.out.println("Exception");
        }
    }
}

static void mb(int i) throws IOException, SQLException{
    System.out.println("mb 1");
}

```

```

        mc(i);
        System.out.println("mb 2");
    }
    static void mc(int i) throws FileNotFoundException,
EOFException, SQLException{
    System.out.println("mc 1");
    if (i==0) throw new NullPointerException();
    if (i==1) throw new java.io.FileNotFoundException();
    if (i==2) throw new java.io.EOFException();
    if (i==3) throw new java.sql.SQLException();
    System.out.println("mc 2");
}
}

```

下面我们分析一下，当正常情况以及异常情况时程序执行的流程。当输入 *i* 不等于 0~4 之间的值时，*mc* 方法不抛出异常，也就意味着 *ma* 中的 *try* 块中没有发生异常。在这种情况下，*ma* 方法执行时，会顺利执行完 *try* 块中的所有语句，然后返回 *main* 方法中。

那如果发生异常情况呢？假设给定 *i == 3*，根据 *mc* 方法中的代码，*mc* 会抛出一个 *SQLException*。*mb* 调用 *mc*，由于 *mb* 中没有处理这个异常的代码，因此会把这个异常抛给 *ma* 方法。也就是说，在 *ma* 调用 *mb* 方法的这个 *try* 块中，产生了一个异常。当产生异常之后，程序的会马上跳出 *try* 块，因此在 *try* 块中最后一句输出“*ma2*”会被跳过。然后，由于 *try* 块中产生了一个 *SQLException*，得到这个异常之后会依次跟后面的 *catch* 语句中的异常类型进行比较，因此会进入捕获 *SQLException* 的语句块中，输出“*SQLException*”。当 *catch* 语句执行完毕之后，*ma* 方法正常返回 *main* 方法中。因此输出结果如下：

```

main 1
ma 1
mb 1
mc 1
SQLException
main 2

```

假设给定 *i == 2*。此时程序中产生的异常是 *EOFException*。当这个异常从 *mc* 传递到 *ma* 时，相当于在 *try* 块中产生了一个 *EOFException*。这个异常被抛出之后，会依次对 *catch* 语句进行匹配。由于 *EOFException* 是 *IOException* 的子类，因此在遇到第一个 *catch* 语句：

```
catch(IOException ioe){...}
```

时，*EOFException* 会作为 *IOException* 的子类被捕获。这是非常典型的多态用在异常的捕获上。类似的，当 *i == 1* 时抛出的 *FileNotFoundException* 也会被捕获 *IOException* 的 *catch* 语句捕获。

当 *i == 0* 时，程序中抛出的是 *NullPointerException*。由于 *NullPointerException* 既不是 *IOException*，也不是 *SQLException*，因此这个异常无法被前两个 *catch* 语句所捕获。但是由于 *NullPointerException* 是 *Exception* 的子类，因此它能够被第三个 *catch* 语句捕获。

上面这个例子说明了 1、未检查异常同样可以被捕获、被处理。2、如果在一系列 *catch* 语句中存在一个捕获 *Exception* 的语句，就意味着任何类型的异常都能够被这个 *catch* 语句

所捕获。换句话说，由于 `Exception` 是所有异常类的父类，根据多态，捕获 `Exception` 类也就是捕获其任何一种子类异常。

由于多态可以用在异常的捕获上面，因此上面的代码还有一个细节值得注意。如果修改上述 `ma` 方法中的代码：

```
static void ma(int i) {  
    try{  
        System.out.println("ma 1");  
        mb(i);  
        System.out.println("ma 2");  
    }  
    //! 下面的代码编译出错!  
    catch(Exception e){  
        System.out.println("Exception");  
    }catch(IOException ioe){  
        System.out.println("IOException");  
    }catch(SQLException sqle){  
        System.out.println("SQLException");  
    }  
}
```

注意，上述代码就是把捕获 `Exception` 类型的 `catch` 语句放到了前面。如果代码写成这样的话，编译就会失败，原因在于：由于任何一种异常都能被当做 `Exception` 来捕获，因此 `try` 块中产生的任何一种异常都会被第一个 `catch` 语句捕获。这也就意味着，即使产生了 `IOException` 或者 `SQLException`，后面的两个 `catch` 语句也不会被执行。由于这两个 `catch` 语句永远无法被执行到，因此会产生一个编译时的错误。

为了避免上述的问题，往往我们写捕获异常的代码时，会遵循这样一个原则：捕获子类异常的 `catch` 语句写在前面，捕获父类异常的 `catch` 语句写在后面。

此外，还要注意的一点是，由于 `try` 块中的代码表示的是可能会出错的代码，因此对于 `java` 来说，`try` 块中的代码是有可能不执行的代码。于是，就可能会衍生出一些特别的语法现象。例如，定义下面的函数：

```
public static int m1(){  
    try{  
        return 100;  
    }catch(Exception e){}  
}
```

上面的函数无法编译通过，原因在于，上面的函数要求必须返回一个 `int` 类型的值，但是由于 `return` 语句是在 `try` 块中，因此 `return` 语句有可能不执行，这样就会使得这个函数不返回任何值，从而产生编译错误。

再看下面这段代码的例子：

```
public static void m2(){  
    int n;  
    try{
```

```
    n = 10;
} catch (Exception e) {}
System.out.println(n);
}
```

上面的代码同样编译出错。在 `m2` 方法中定义了一个局部变量 `n`，这个局部变量在 `try` 块中被赋予了初始值。但是由于 `java` 认为 `try` 块中的语句不一定会被执行，所以在 `try` 块外输出 `n` 的值时，会提示“可能尚未初始化变量 `n`”，因此编译出错。

### 3.3 finally

`finally` 语句是 `try-catch` 语句的一个补充。在 `try-catch` 语句块之后，可以加上一个 `finally` 语句块，这个语句块中的代码，无论程序执行时是否发生异常，最终都会被执行。例如，我们可以为 `ma` 方法的 `try-catch` 语句增加一个 `finally` 代码块：

```
static void ma(int i) {
    try{
        System.out.println("ma 1");
        mb(i);
        System.out.println("ma 2");
    }catch (IOException ioe){
        System.out.println("IOException");
    }catch (SQLException sqle){
        System.out.println("SQLException");
    }catch (Exception e){
        System.out.println("Exception");
    }finally{
        System.out.println("in finally of ma");
    }
}
```

如上面代码所示，我们为 `try-catch` 语句增加了一个 `finally` 语句块。当程序正常运行的时候（也就是调用 `mb` 方法时没有产生异常），此时 `try` 块正常执行结束。在程序跳出 `try-catch-finally` 而返回主方法之前，`finally` 语句会得到执行。体现在输出结果中，就是在 `ma2` 和 `main2` 之间输出“`in finally of ma`”。输出结果如下：

```
main 1
ma 1
mb 1
mc 1
mc 2
mb 2
ma 2
in finally of ma //当程序正常时会运行 finally
main 2
```

当产生异常时（例如 `i == 1`），此时由于 `try` 块中抛出异常，因此跳出 `try` 块并且执行相应的 `catch` 语句块。当 `catch` 语句块执行之后，同样会执行 `finally` 语句块中的内容之后，然后才会返回主方法。当 `i == 1` 时，输出结果如下：

```
main 1
ma 1
mb 1
mc 1
IOException //进入 catch IOException 的 catch 语句
in finally of ma //产生异常也要运行 finally
main 2
```

因此，`finally` 语句块中的代码，意味着无论程序是否发生异常，都一定要执行。关于这一点，我们还可以看下面的代码

```
static int ma2(int i) {
    try{
        mb(i);
        return 100;
    }catch(Exception e){
        System.out.println("Exception");
        return 200;
    }finally{
        System.out.println("in finally of ma");
        return 300;
    }
}
```

上述代码，在 `try` 块中有一个 `return 100`，在 `catch` 块中有一个 `return 200`，在 `finally` 中有一个 `return 300`。如果在其他方法中调用这个 `ma2` 方法，返回值是什么呢？

如果 `mb` 方法不抛出异常，程序正常执行的话，这样执行到 `try` 块最后的 `return 100` 时，程序不会在 `try` 块内返回，而会先执行 `finally` 语句块中的内容，再执行 `return 100`。而由于 `finally` 语句块中存在一个 `return 300`，因此 `ma2` 方法的返回值为 300。

如果 `mb` 方法抛出异常，则程序会跳出 `try` 块，进入 `catch` 子句。在 `catch` 子句中有一个 `return` 语句。这个 `return` 语句在执行之前，同样要先执行 `finally` 语句块中的内容，执行完之后才能返回。由于 `finally` 语句块中同样具有 `return` 语句，因此在执行 `finally` 语句过程中，程序就会返回，返回值为 300。

也就是说，如果 `finally` 语句块中存在 `return` 语句，最终函数一定会从 `finally` 中返回，而不是 `try` 块或者 `catch` 语句中返回。这也进一步说明：`finally` 块中的代码一定会执行。

正因为 `finally` 语句块中的代码一定会被执行，因此 `finally` 语句块中我们往往会放上一些释放资源的代码。例如，有下面一些步骤：

1. 申请数据库连接资源
2. 通过数据库的验证，得到数据库连接
3. 使用数据连接，完成数据库的操作
4. 释放数据库连接

在上面的四个步骤中，释放数据库连接就是一个非常典型的释放资源的过程。在申请到了数据库资源之后，无论是第二步进行数据库验证，还是第三步完成数据库操作，都有可能发生异常。但是无论发生异常与否，数据库连接的释放都应该执行，因此这一步也就是所谓的一定要执行的一步。为了保证数据库连接的释放一定被执行，因此这一步应当放在 `finally`

语句块中。

## 4 异常与面向对象

上面介绍了一些异常的基本操作和语法，接下来这部分内容，将结合面向对象与异常处理，对异常进行进一步的介绍。

### 4.1 异常与方法覆盖

我们首先介绍一下异常与方法覆盖相关的内容。方法覆盖对于方法声明抛出的异常也有要求，具体的来说，要求：子类的覆盖方法不能比父类的被覆盖方法抛出更多的异常。

如何来理解所谓“不能抛出更多的异常”呢？考虑下面的代码：

```
class Super{  
    public void m() throws IOException{}  
}  
  
class Sub extends Super{  
    public void m() throws IOException{}  
}
```

上面两个类中，Sub 类继承自 Super 类，并且覆盖了 Super 类的 m 方法。在 Super 类中的 m 方法抛出 IOException，而子类的 m 方法也同样抛出 IOException。这样，子类的覆盖方法与父类的被覆盖方法抛出的异常相同，这样能够编译通过。

同样的，如果父类方法中 throws IOException，而子类方法中不抛出任何异常，这样代码同样可以编译通过，这样也不算抛出更多的异常。示例代码如下：

```
class Super{  
    public void m() throws IOException{}  
}  
  
class Sub extends Super{  
    public void m() {}  
}
```

但是，如果父类方法中没有抛出 IOException，而子类方法抛出这个异常，则会产生问题。例如：

```
//此处代码编译出错！  
class Super{  
    public void m() {}  
}  
  
class Sub extends Super{  
    public void m() throws IOException{}  
}
```

上述代码就是子类比父类抛出了更多的异常。

此外，异常的抛出还有多态的问题。例如下面的代码：

```

class Super{
    public void m() throws IOException{}
}

class Sub extends Super{
    public void m() throws FileNotFoundException, EOFException{}
}

```

在上面的代码中，父类只抛出一个异常，子类抛出了两个异常，但是由于子类抛出的异常 `FileNotFoundException` 和 `EOFException`，是父类抛出的异常的子类，因此并不能算子类抛出了更多的异常。与之对应的是下面这个例子：

```

//此处代码编译出错!
class Super{
    public void m() throws IOException{}
}

class Sub extends Super{
    public void m() throws SQLException{}
}

```

这个例子同样编译出错。因为父类抛出的是 `IOException`，而子类抛出的 `SQLException`，这两个异常类之间完全没有任何的父子类之间的关系。

换句话说，子类不能比父类抛出更多的异常，我们可以这么来理解：子类要么抛出跟父类相同的异常，要么不抛出异常，要么抛出的异常是父类抛出异常的子类。只有这样三种选择。

## 4.2 异常类介绍

`Exception` 类作为 `java` 中异常处理的核心，下面我们就研究一下这个类的相关方法和属性。

### 4.2.1 message 属性

在 `Exception` 类中，有一个 `getMessage` 方法。该方法签名如下：

```
public String getMessage()
```

这个方法是在 `Throwable` 中定义的，因此能够被 `Exception` 子类继承，并且也就意味着所有的异常类都包含这个方法。这个方法返回一个字符串，JDK 的文档中，称这个方法返回的是“详细消息”。那么什么是这个详细消息呢？

从 `getMessage` 这个方法的签名来看，非常类似 `getXXX` 方法，我们有理由相信，在 `Throwable` 这个类中包含一个私有的 `message` 属性，而这个 `getMessage()` 方法就是用来获得那个 `message` 属性的。

虽然在 `java` 中提供了 `getMessage` 方法用来获得这个属性，但是却没有提供 `setMessage` 方法来设置这个属性的值。那个如何来设置属性值呢？这就只能在创建异常的时候，调用异常的构造方法来完成。在 `Exception` 类中包含这样一个构造方法：

```
public Exception(String message)
```

这个构造方法接受一个字符串参数，从 JDK 文档的形参名我们就可以猜测出，这个构造方法的参数能够用来设置 `Throwable` 中的 `message` 属性。也就是说，如果我们创建异常时给定一个字符串参数，则在后来捕获异常之后调用 `getMessage` 方法时就能获得这个字符串的

值。例如下面的例子：

```
import java.util.*;
import java.io.*;
import java.sql.*;
public class TestExceptionArgs{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        ma(n);
    }

    public static void ma(int n) {
        try{
            mb(n);
        }catch(Exception e){
            System.out.println(e.getMessage());
        }
    }

    public static void mb(int n) throws Exception{
        if (n == 0) throw new SQLException("n==0");
        if (n == 1) throw new EOFException("n==1");
        if (n == 2) throw new FileNotFoundException("n==2");
    }
}
```

上面的代码就是非常典型的使用 `message` 属性的例子：在创建异常时，给定一个字符串参数，而当捕获异常时，利用 `getMessage` 方法获得当时传递的参数。

当输入 `n` 为 1 时，输出结果为：

`n==1`

也就是创建异常时，给定的 `message` 参数。

`message` 属性往往被用来对异常进行一些解释和说明，使得程序员在得到异常的同时，能够得到关于该异常的更多信息，方便程序员调试。就像我们到银行取款，如果出现了异常，我们总是希望更多的了解这个异常的信息，究竟为什么取款失败，是因为密码错误，还是因为余额不足呢？

#### 4.2.2 printStackTrace

除了 `getMessage` 方法之外，在异常类中还有一个常用方法：`printStackTrace`。这个方法的签名如下：

```
public void printStackTrace()
```

这个方法同样是 `Exception` 类从 `Throwable` 继承来的方法，同样是所有异常类都具有的方法。这个方法的作用是：在标准错误输出上打印出产生异常时的方法调用栈的信息。怎么来理解呢？所谓的标准错误，在默认情况下往往指的就是程序员的屏幕。而所谓的方法调用栈的信息是什么意思呢？我们可以试验一下打印的信息。把上面 `TestExceptionArgs.java` 程序进行修改如下：

```
import java.util.*;
```

```

import java.io.*;
import java.sql.*;
public class TestExceptionArgs{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        ma(n);
    }

    public static void ma(int n) {
        try{
            mb(n);
        }catch(Exception e){
            e.printStackTrace();
        }
    }

    public static void mb(int n) throws Exception{
        if (n == 0) throw new SQLException("n==0");
        if (n == 1) throw new EOFException("n==1");
        if (n == 2) throw new FileNotFoundException("n==2");
    }
}

```

在 catch 语句中，我们调用 `printStackTrace` 来打印方法调用栈的信息。当 n 输入为 1 时，输出结果如下：

```

java.io.EOFException: n==1
at TestExceptionArgs.mb(TestExceptionArgs.java:21)
at TestExceptionArgs.ma(TestExceptionArgs.java:13)
at TestExceptionArgs.main(TestExceptionArgs.java:8)

```

我们可以看到，打印的内容分为了三个部分。首先，是抛出异常的类型名：`java.io.EOFException`。随后，在异常类名之后，是一个冒号，冒号后面的内容，就是我们创建异常时给定的“详细信息”，也可以理解为就是异常的 `message` 属性。

之后，有三行信息，这三行说明了异常产生时的方法调用关系。异常是在 21 行，`mb` 方法中产生的，调用 `mb` 方法的是 13 行的 `ma` 方法，调用 `ma` 方法的是第 8 行的 `main` 方法。通过这样的格式，打印出了产生异常时程序运行的状态和方法调用的关系。

在实际开发过程中，在开发和调试阶段，`printStackTrace` 方法往往会被程序员用来做异常处理，因为这个方法能够为程序员提供非常多的信息，能够帮助程序员更好的找到错误并改正错误。

## 5 自定义异常

我们除了可以使用 Sun 公司已经提供好的异常之外，也可以创建自己的异常体系和异常结构，从而完善自己的软件系统。在 Java 中，自定义异常是相当简单的事情，下面我们就

对自定义异常进行一下介绍。

## 5.1 创建异常类

在 Java 中自定义一个异常类非常简单，只要保证这个异常类继承自 `Exception` 类就可以；而如果想要自定义一个未检查异常，则只要创建一个类继承自 `RuntimeException` 就可以了。例如下面的代码：

```
//自定义已检查异常
class MyException1 extends Exception{
}

//自定义未检查异常
class MyException2 extends RuntimeException{
}
```

定义了这两个自定义异常之后，使用的方式跟 Sun 公司定义的异常类相比，没有任何区别。例如下面的例子：

```
public class TestMyException{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        ma(n);
    }

    public static void ma(int n) {
        try{
            mb(n);
        }catch(MyException1 e){
            e.printStackTrace();
        }catch(Exception e){
            e.printStackTrace();
        }
    }

    public static void mb(int n) throws MyException1{
        if (n == 0) throw new MyException1();
        if (n == 1) throw new MyException2();
    }
}
```

从上面的代码我们可以看出，创建 `MyException1` 和 `MyException2` 的对象，以及抛出这两个异常，以及 `throws` 语句和 `try-catch` 语句，无论是异常的哪一个方面，我们创建的自定义异常都和 Sun 公司提供的异常完全一样。

也就是说，只要继承自 `Exception` 类或者 `RuntimeException` 类，自定义异常最基本的部分就完成了。

## 5.2 设定 message 属性

但是，如果仅仅让 `MyException` 继承 `Exception` 或 `RuntimeException` 类的话，这样的功

能还不够完善。我们应当让自定义异常和 Sun 公司提供的异常一样，能够设置 `message` 属性。由于 `message` 属性在 `Throwable` 类中，并且没有提供 `setMessage` 的方法，我们只能让异常在起构造方法中对 `message` 属性进行赋值。

由于我们的 `MyException` 本身并没有定义 `message` 属性，因此为了设置 `message` 属性，我们必须为 `MyException` 提供接受字符串作为参数的构造方法。例如下面的代码：

```
public MyException(String msg) {  
    ...  
}
```

在构造方法中应当做哪些事情呢？在构造方法中，需要做的就只有一件事情，那就是调用父类（`Exception` 类或者 `RuntimeException` 类）的带字符串参数的构造方法。修改后的 `MyException1` 和 `MyException2` 代码如下：

```
//自定义已检查异常  
class MyException1 extends Exception{  
    public MyException1() {}  
    public MyException1(String str){  
        super(str);  
    }  
}  
  
//自定义未检查异常  
class MyException2 extends RuntimeException{  
    public MyException2() {}  
    public MyException2(String str){  
        super(str);  
    }  
}
```

代码非常简单。当修改过后，就可以使用带字符串参数的构造方法以及 `getMessage` 方法来使用自定义异常类的 `message` 属性了。

# Chp13 多线程

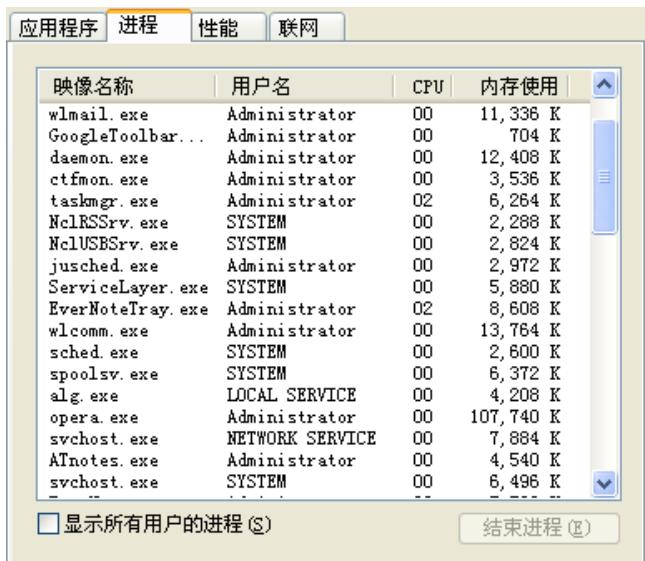
## 本章导读

并发编程是现代操作系统以及现代编程语言中很重要的一个组成部分。Java 语言中，对并发编程的支持很好，使用 Java 语言能够非常简单的写出多线程编程的代码。

### 1 多线程与并发的概念

#### 1.1 并发与多进程

首先，我们来介绍一下并发的基本概念和原理。对于现代操作系统来说，大部分操作系统都是多任务操作系统。什么是“多任务”呢？指的是每个系统在同时能够运行多个进程。例如，在 windows 中，可能开着一个 Word 窗口写文档，开着一个 eclipse 写代码，开着一个 QQ 聊天，开着一个 IE 浏览网页，开着一个 Winamp 听歌……这就意味着，在一个操作系统中，同时有多个程序在内存中运行着。每一个运行着的程序，就是操作系统中运行着的一个任务，也就是我们所说的“进程”。例如，在 windows 中，可以通过“任务管理器”来查看系统中有多少进程正在运行。如下图：



从上面的例子我们可以看出，在一个操作系统中可以同时运行多个进程，这就是进程的并发。

我们知道，一般来说，在个人电脑上，往往只有一个 CPU。同时，每个程序运行的时候，都需要在 CPU 上完成运算，也就是说，所有的程序运行时，都需要占用 CPU。那么，在系统中，是如何做到使用单 CPU 同时运行多个程序的呢？

接下来，我们来阐述一下单 CPU 执行多任务的原理。

假设我们同时开了多个程序：Word, IE, QQ, Winamp，对于操作系统来说，这意味着有四个进程要同时运行。为了解决这个问题，计算机规定了：让这四个进程轮流使用 CPU，每个进程用一小会儿。这个“一小会儿”，往往是若干个毫秒，这段时间被称为一个“CPU 时间片”。这样，每一秒钟可能有上百个 CPU 时间片，也就是说，在一秒钟之内，这四个程

序可能各自能够占用一小会儿 CPU，从而运行一下。从微观上来看，每一个特定的时刻，CPU 上只有一个程序在运行。示意图如下：



如上图所示，在某个特定的 CPU 时间片中，只运行一个程序。而操作系统控制 CPU，让多个程序不停的切换，从而保证多个程序能够轮流使用 CPU。

从本质上说，单 CPU 一次只能运行一个任务。但是，由于 CPU 时间片非常短，每次从一个程序切换到另一个程序的时候速度很快。在一秒钟内，可能有上百个 CPU 时间片，也就是说系统可能进行了上百次任务的切换。由于人的反应相对计算机来说，是比较慢的，因此对于人来说，感受就是在一秒内，这几个进程同时在运行。

这就是单 CPU 执行多任务的原理：利用很短的 CPU 时间片运行程序，在多个程序之间进行 CPU 时间片的快速切换。可以把这种运行方式总结为一句话：宏观上并行，微观上串行。这是说，从宏观上来看，一个系统同时运行多个程序，这些程序是并行执行的；而从微观上说，每个时刻都只有一个程序在运行，这些程序是串行执行的。

## 1.2 线程的概念

上面所说的，是操作系统与多进程的概念。但是，由于 Java 代码是运行在 JVM 中的，对于某个操作系统来说，一个 JVM 就相当于一个进程。而 Java 代码不能够越过 JVM 直接与操作系统打交道，因此，Java 语言中没有多进程的概念。也就是说，我们无法通过 Java 代码写出一个多进程的程序来。

Java 中的并发，采用的是线程的概念。简单的来说，一个操作系统可以同时运行多个程序，也就是说，一个系统并发多个进程；而对于每个进程来说，可以同时运行多个线程，也就是：一个进程并发多个线程。

从上面的描述上我们可以看出，线程就是在一个进程中并发运行的一个程序流程。当若干的 CPU 时间片分配给 JVM 进程的时候，系统还可以把时间片进一步细分，分给 JVM 中的每一个线程来执行，从而达到“宏观上并行，微观上串行”的线程执行效果。

目前为止，我们见到的 Java 程序只有一个线程，也就是说，只有一个程序执行流程。这个流程从 main 方法的第一行开始执行，以 main 方法的退出作为结束。这个线程我们称之为“主线程”。

那么，一个线程运行，需要哪些条件呢？首先，必须要给线程赋予代码。通俗的来说，就是必须要为线程写代码。这些代码是说明，启动线程之后，这个线程完成了什么功能，我们需要这个线程来干什么。

其次，为了能够运行，线程需要获得 CPU。只有获得了 CPU 之后，线程才能真正启动并且执行线程的代码。CPU 的调度工作是由操作系统来完成的。

第三，运行线程时，线程必须要获得数据。也就是说，在进行运算的时候，线程需要从内存空间中获得数据。关于线程的数据，有一个结论，叫做“堆空间共享，栈空间独立”。我们之前提到过，所谓的“堆空间”，保存的是我们利用 new 关键字创建出来的对象；而所谓的栈空间，保存的是程序运行时的局部变量。“堆空间共享，栈空间独立”的意思是：多线程之间共享同一个堆空间，而每个线程又拥有各自独立的栈空间。因此，运行程序时，多个不同线程能够访问相同的对象，但是多个不同线程彼此之间的局部变量时独立的，不可能出现多个线程访问同一个局部变量的情况。

CPU、代码、数据，是线程运行时所需要的三大条件。在这三大条件中，CPU 是操作系

统分配的，Java 程序员无法控制；数据这部分，需要把握住“堆空间共享，栈空间独立”的概念（关于这个概念，我们在后面还会继续提到）；而下一个章节，将为大家介绍，如何为线程赋予代码。

## 2 Thread 类与 Runnable 接口

在 Java 中，要为线程赋予代码，有两种方式：一种是继承 `Thread` 类，一种是实现 `Runnable` 接口。下面我们分别来看一下如何来使用这两种方式为线程赋予代码。

### 2.1 Thread 类

第一种方式是创建一个类，让这个类继承 `java.lang.Thread` 类，然后覆盖 `Thread` 类中的 `run()` 方法，在 `run()` 中提供线程的代码。注意，`Thread` 类中的 `run()` 签名如下：

```
public void run()
```

这就意味着我们创建的 `Thread` 中的 `run()` 签名也应当写成这样。

例如，假设我们想要创建两个线程：一个线程输出 1000 遍“\$\$\$”；另一个线程输出 1000 遍“###”。在为线程赋予代码的时候，我们只需要创建两个新的类继承 `Thread`，并且覆盖 `Thread` 中的 `run()` 即可。示例代码如下：

```
class MyThread1 extends Thread{
    public void run(){
        for(int i = 1; i<=1000; i++){
            System.out.println(i + " $$$");
        }
    }
}

class MyThread2 extends Thread{
    public void run(){
        for(int i = 1; i<=1000; i++){
            System.out.println(i + " ###");
        }
    }
}
```

上面的代码中，我们定义了两个线程类，并为这两个线程类赋予了代码。

有了这两个线程类之后，我们就可以利用这两个类来创建和处理线程了。那应该如何利用自定义的线程类来创建新线程呢？

首先，应当利用自定义的线程类创建线程对象，之后，调用线程的 `start()`，就能够启动新线程。注意，在我们自定义的线程类（`MyThread1` 和 `MyThread2`）中，我们并没有自己添加 `start()` 方法，这个方法是从 `Thread` 类中继承来的。相关代码如下：

```
public class TestThread{
    public static void main(String args[]){
        Thread t1 = new MyThread1();
        Thread t2 = new MyThread2();

        t1.start();
    }
}
```

```
    t2.start();  
}  
}
```

我们可以执行一下这个 `TestThread` 程序，在我当前的 Windows XP 机器上，执行中的部分结果如下：

```
1 $$$  
1 ###  
2 ###  
3 ###  
4 ###  
2 $$$  
5 ###  
3 $$$  
6 ###  
7 ###  
4 $$$  
8 ###  
9 ###  
10 ###  
11 ###  
12 ###  
13 ###  
14 ###  
15 ###  
16 ###  
17 ###  
18 ###  
19 ###  
20 ###  
21 ###  
5 $$$  
6 $$$  
7 $$$  
8 $$$  
9 $$$  
10 $$$  
11 $$$  
12 $$$  
13 $$$  
.....  
###和$$$一直交替输出，一直到最后：  
949 $$$  
.....  
973 $$$
```

```
988 ###
989 ###
.....
998 ###
999 ###
1000 ###
974 $$$
975 $$$
976 $$$
.....
997 $$$
998 $$$
999 $$$
1000 $$$
```

可以看出，两个线程交替打印出\$\$\$和###字符串，说明这两个线程交替的占用了 CPU。

上面的代码中，有好几处需要注意的地方。首先，上面的程序中有这样两行代码：

```
Thread t1 = new MyThread1();
Thread t2 = new MyThread2();
```

这两行代码创建了 t1 和 t2 两个线程对象，但是，并没有在系统中创建新的线程。我们利用 new 关键字创建出来的，是 JVM 中的两个对象。这两个对象并不等于系统中的线程，但是通过操作这两个对象，能够来操作系统中的线程。

怎么来理解呢？就好比我们去银行办理业务，归根结底是去银行修改我们在银行的账户信息，比如存款就是增加账户余额，而取款就是减少账户余额，等等。但是我们去银行，是不能直接操作自己的账户的，而必须通过银行的职员来操作。换句话说，我们以银行职员为“代理”来操作自己的账户。

与上面的例子相似，我们创建的线程对象，就好像是系统中的线程的“代理”。也就是说，我们能够通过线程对象来对系统中的线程进行操作，可以请求系统创建线程、销毁线程等等。但是，线程对象本身不是线程：创建线程对象时，系统中没有创建新的线程；而销毁线程时，线程对象有可能还存在，要一直等到垃圾回收时，线程对象才会被销毁。

创建完线程对象之后，只有调用 start()，系统中才真正启动了一个新线程。就像我们之前所说的那样，start()是 Thread 类中的方法。特别要注意的是，当我们创建线程时，覆盖的是 run()，而不要去覆盖 start()！

调用完 start()之后，线程就开始真正启动了。可以猜测，在 start()内部，会通过 JVM 跟底层的操作系统进行交互，请求系统创建一个新线程。创建完新线程之后，这个线程执行的就是 run()中的代码。换句话说，我们可以当做在 start()内部，调用了我们覆盖以后的 run()。

另外，当我们对两个对象调用了 start 方法之后，在我们的 JVM 中总共有几个线程在运行呢？要注意，现在总共有三个线程在运行：

```
t1 线程：执行 MyThread1 中的 run 方法
t2 线程：执行 MyThread2 中的 run 方法
主线程：执行 main 方法。
```

注意，除了我们创建的 t1 和 t2 两个线程之外，还有一个主线程。主线程是所有 Java 程序运行时首先启动的第一个线程。而 t1 和 t2 这两个线程，都是在主方法内部调用 start 以后

才启动起来的，因此也就是说，这两个线程是从主线程中启动的。

那能不能直接调用 `run()` 方法呢？从语法上来说，完全可以对线程对象调用 `run()` 方法。但是，那样的话，并没有创建新线程。例如，如果在主方法中这样来调用：

```
t1.run();
t2.run();
```

这样就意味着调用了这两个对象的 `run` 方法，执行的时候，会先执行 `t1` 对象的 `run` 方法，当 `t1` 的 `run` 方法返回之后，再执行 `t2` 线程的 `run` 方法。也就是说，在主线程中运行了这两个 `run` 方法，程序自始至终只有一个线程在执行。

## 2.2 Runnable 接口

除了继承 `Thread` 类之外，我们还可以采用另外一种方式创建线程，就是实现 `Runnable` 接口。`Runnable` 接口在 `java.lang` 包中定义，在这个接口中，只包含一个方法：`run()` 方法。该方法签名如下：

```
void run()
```

由于这个方法定义在接口中，因此默认就是 `public` 的。在实现 `Runnable` 接口的时候，只需要实现这个 `Run` 方法就可以了。

下面我们修改刚刚的 `MyThread2` 这个类，把这个类改名为 `MyRunnable2`，并且由继承 `Thread` 类改为实现 `Runnable` 接口。相应代码如下：

```
class MyRunnable2 implements Runnable{
    public void run() {
        for(int i = 1; i<=1000; i++) {
            System.out.println(i + " ###");
        }
    }
}
```

在上面的代码中，对 `run()` 方法的实现没有任何修改，而只是改动了第一行：把继承 `Thread` 类改成了实现 `Runnable` 接口。

修改完了线程的实现之后，还必须要修改一下创建线程的代码。由于 `MyRunnable2` 类不是 `Thread` 类的子类，因此创建的 `MyRunnable2` 类型的对象不能直接赋值给 `Thread` 类型的引用。换句话说，我们利用 `MyRunnable2` 创建出来的不是一个线程对象，而是一个所谓的“目标”对象。代码如下：

```
Runnable target = new MyRunnable2();
```

创建了目标对象之后，可以利用目标对象再来创建线程对象，代码如下：

```
Thread t2 = new Thread(target);
```

通过上面的两行代码，我们就可以利用 `Runnable` 接口的实现类来创建线程对象。创建完线程对象之后，需要启动线程时，同样要调用线程对象的 `start()` 方法。完整的代码如下：

```
class MyThread1 extends Thread{
    public void run() {
        for(int i = 1; i<=1000; i++) {
            System.out.println(i + " $$$");
        }
    }
}
```

```

    }

class MyRunnable2 implements Runnable{
    public void run(){
        for(int i = 1; i<=1000; i++){
            System.out.println(i + " ###");
        }
    }
}

public class TestThread{
    public static void main(String args[]){
        Thread t1 = new MyThread1();
        Runnable target = new MyRunnable2();
        Thread t2 = new Thread(target);

        t1.start();
        t2.start();
    }
}

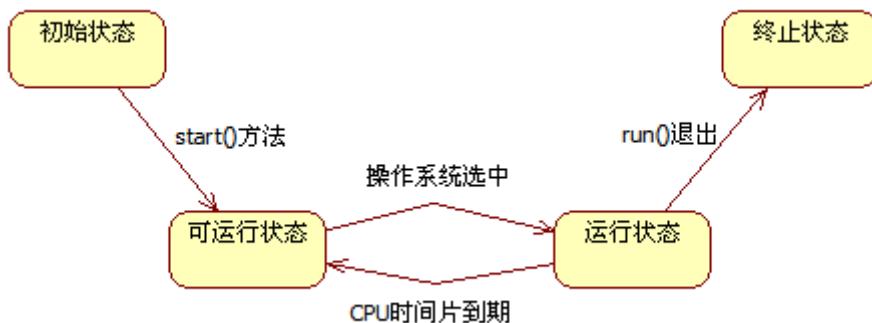
```

### 3 线程的状态

在上一节的内容中，我们为大家介绍了如何为线程写代码。在这一小节中，我们将更加详细的为大家介绍线程的相关知识，首先来介绍线程在整个程序运行中的状态转换。

#### 3.1 线程状态图

在上一节的例子中，我们创建的程序总共有这样四个状态：初始状态、可运行状态、运行状态、终止状态。这四个状态的转换如下图：



下面对这四种状态进行详细的介绍。

**初始状态：**当我们创建了一个线程对象，而没有调用这个线程对象的 `start()` 方法时，此时线程处于初始状态。在上一小节中，我们介绍过，创建了一个线程对象，不等于在系统中创建了一个新线程。当我们创建一个线程对象时，此时线程处于初始状态。也就是说，线程

对象自身进行了一些初始化的操作，而并没有向操作系统申请创建系统中的线程。

另外，要注意的是，当一个线程调用 `start()` 方法之后，由初始状态进入了可运行状态。这个状态的转换过程是一个单向的过程，只能有初始状态转换为可运行状态，而不能由可运行状态转换为初始状态。也就是说，一旦调用完了 `start` 方法之后，线程就进入了可运行状态而不会回到初始状态，因此在线程对象的整个生命周期中，只能调用一次 `start()` 方法。如果对同一个线程对象多次调用 `start()` 方法，会产生一个 `IllegalStateException` 异常。

**可运行状态：**处于可运行状态下的线程，具有这样的特点：线程已经为运行做好了完全的准备，只等着获得 CPU 来运行。也就是说，线程是万事俱备，只欠 CPU。

**运行状态：**处于这种状态的线程获得了 CPU 时间片，正在执行代码。由于系统中只有一个 CPU，因此，同时只能有一个线程处于运行状态。

当 CPU 空闲的时候，操作系统会检查是否有可运行状态的线程。如果有一个或多个线程处于可运行状态，系统会根据一定的规则挑选一个线程，为这个线程分配一个 CPU 时间片，从而使得这个线程进入了运行状态。

当然，一个线程不能永远的占用 CPU，不可能永远的处于运行状态。系统为了实现多任务同时进行，会为线程分配一个 CPU 时间片。当 CPU 时间片到期而线程没有执行完毕时，操作系统会把处于运行状态的线程转换成可运行状态，然后再重新从可运行状态中选取一个线程，为其分配 CPU 时间片。这就是运行状态和可运行状态之间的转换。

**终止状态：**当一个线程执行完了 `run()` 方法中的代码，该线程就会进入终止状态。要注意的是，这个状态转换也是一个单向箭头。也就是说，一旦一个线程从运行状态进入了终止状态，那这个线程就进入了生命的尾声，我们无法通过任何手段重新启动这个线程。

特别要提醒的是，在上一小节中，除了我们创建的两个线程之外，系统中还存在第三个线程：主线程。主线程的启动没有必要经历初始状态，当我们启动 JVM 执行程序时，JVM 会执行某个类的主方法，此时主方法就在主线程中执行。此后，当主方法结束之后，主线程就进入了终止状态。需要注意的是，主线程进入终止状态，并不意味着整个程序就结束了。例如上一小节的例子：

```
public class TestThread{  
    public static void main(String args[]){  
        Thread t1 = new MyThread1();  
        Runnable target = new MyRunnable2();  
        Thread t2 = new Thread(target);  
  
        t1.start();  
        t2.start();  
    }  
}
```

主方法调用完了两个线程的 `start()` 方法之后，就进入了终止状态。但是由于现在的程序中，除了主线程之外，还有两个线程 `t1` 和 `t2`，这两个线程还在没有终止，因此程序不会终止执行。

我们之前曾经说过，程序启动之后执行主方法，主方法退出时整个程序退出。应当说，这样的说法是不准确的，必须等程序中所有线程都进入终止状态之后，整个程序才会终止。只不过之前我们写的所有程序都只有主线程一个线程而已，当这个线程进入终止状态之后，

由于没有其他的线程存在，程序就终止了。而现在我们既然学习了多线程，就应当更新我们的概念。请记住：主方法退出时，程序未必退出；只有整个程序中所有线程都进入了终止状态，整个程序才会退出。

此外，还有一个小问题：为什么对一个线程调用 `start()` 方法之后，这个线程不能马上进入运行状态，而一定要首先进入可运行状态呢？

要回答这个问题，请思考：对于上面的例子来说，当程序运行到调用 `t1.start()` 方法启动 `t1` 线程时，有没有线程处于运行状态？

事实上，由于 `t1.start()` 这行代码处于主方法中，因此当这行代码被执行时，意味着主线程正处于运行状态！换句话说，由于运行状态有一个线程正在运行，因此被启动的新线程只能先处于可运行状态。

启动线程时，必须有别的线程调用 `start` 方法（例如，主线程中调用 `t1.start()` 方法），而调用 `start()` 方法又意味着，运行状态有别的线程正在运行（例如，调用 `t1.start()` 方法意味着主线程正在运行），因此新启动的线程只能处于可运行状态。

### 3.2 `sleep()`与阻塞

除了上面所说的四种状态之外，还有一个很重要的状态：阻塞状态。

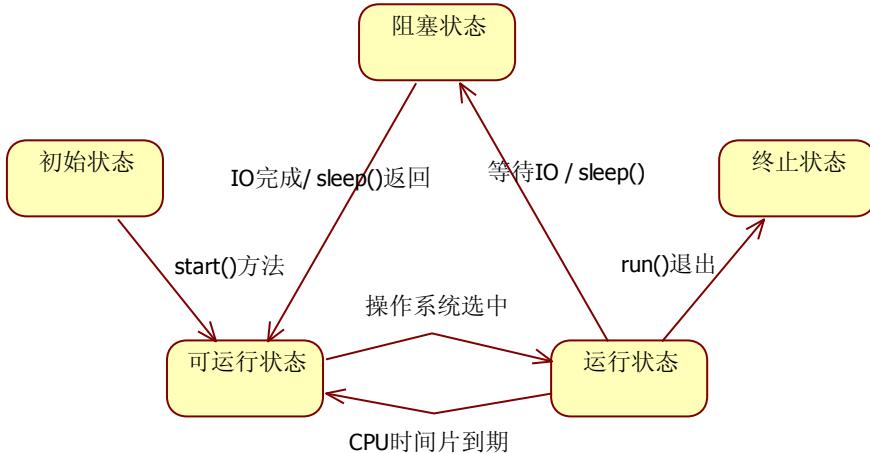
什么是阻塞状态呢？我们知道，如果一个线程要进入运行状态，则必须要获得 CPU 时间片。但是，在某些情况下，线程运行不仅需要 CPU 进行运算，还需要一些别的条件，例如等待用户输入、等待网络传输完成，等等。如果线程正在等待其他的条件完成，在这种情况下，即使线程获得了 CPU 时间片，也无法真正运行。因此，在这种情况下，为了能够不让这些线程白白占用 CPU 的时间，会让这些线程会进入阻塞状态。最典型的例子就是等待 I/O，也就是说，如果线程需要与 JVM 外部进行数据交互的话（例如等待用户输入、读写文件、网络传输等），这个时候，当数据传输没有完成时，线程即使获得 CPU 也无法运行，因此就会进入阻塞状态。

可以这么来理解：我们可以把 CPU 当做是银行的柜员，而去银行的准备办理业务的顾客就好比是线程。顾客希望能够让银行的柜员为自己办事，就好像线程希望获得 CPU 来运行线程代码一样。但有时会出现这种情况：比如有个顾客想去银行汇款，排队等到了银行的职员为自己服务，却事先没有填好汇款单。这个时候，这个银行职员即使想为你服务，也无法做到；这就相当于线程没有完成 I/O，此时线程即使获得 CPU 也无法运行。一般这个时候，银行的工作人员也不会傻等，而是会非常礼貌的给顾客一张单子，让他去旁边找个地方填写汇款单，而职员自己则开始接待下一位顾客；这也就相当于让一个线程进入阻塞状态。

当所等待的条件完成之后，处于阻塞状态的线程就会进入可运行状态，等待着操作系统为自己分配时间片。还是用我们银行职员的比喻：当顾客填写汇款单的时候，就好比是线程进入了阻塞状态；而当顾客把汇款单填完了之后，这就好比是线程的 I/O 完成，所等待的条件已经满足了。这样，这个顾客就做好了办理业务所需要的准备，就等着能够有一个银行职员来为自己服务了。

那为什么从阻塞状态出来之后，线程不能马上进入运行状态呢？因为当线程所等待的条件完成的时候，可能有一个线程正处于运行状态。由于处于运行状态的线程只能有一个，因此当线程从阻塞状态出来之后，只能在可运行状态中暂时等待。这就好比，当顾客填完汇款单之后，银行的职员可能正在为其他顾客服务。这个时候，你不能把其他用户赶走，强行让银行职员为你服务，而只能重新在后面排队，等着有银行职员为你服务。

加上阻塞状态之后，线程的转换图如下：



如上图所示，除了等待 IO 会进入阻塞状态之外，还可以调用 `Thread` 类的 `sleep` 方法进入阻塞状态。顾名思义，`sleep` 方法就是让线程进入“睡眠”状态。你可以想象，如果一个顾客在银行办理业务的时候睡着了，那样的话，银行职员会让这个顾客在一旁先呆着，等什么时候顾客睡醒了，再什么时候为这个顾客服务。

下面我们来看一下怎么调用 `sleep` 方法。首先，`sleep` 方法的方法签名为：

```
public static void sleep(long millis) throws InterruptedException
```

这个方法是一个 `static` 方法，也就意味着可以通过类名来直接调用这个方法。`sleep` 方法的参数是一个 `long` 类型，表示要让这个线程“睡”多少个毫秒（注意，1 秒=1000 毫秒）。另外，这个方法抛出一个 `InterruptedException` 异常，这个异常是一个已检查异常，根据异常处理的规则，这个异常必须要处理。

我们修改上一节的代码，让两个线程 `MyThread1` 和 `MyRunnable2` 每次输出之后都“睡”200 毫秒。首先修改 `MyThread1` 类：

```
class MyThread1 extends Thread{  
    public void run(){  
        for(int i = 1; i<=1000; i++){  
            System.out.println(i + " $$$");  
            try{  
                Thread.sleep(200);  
            } catch(InterruptedException e){}  
        }  
    }  
}
```

在 `for` 循环中增加了 `Thread.sleep` 方法。要注意的是，由于 `sleep` 方法抛出一个已检查异常，所以必须要处理。由于 `Thread` 类中的 `run` 方法没有抛出任何异常，根据方法覆盖的要求，`MyThread1` 类中的 `run` 方法也不能抛出任何异常。因此，对于 `sleep` 方法抛出的异常，程序员只能用 `try-catch` 的方法处理。

下面是修改后的 `MyRunnable2` 的代码：

```
class MyRunnable2 implements Runnable{  
    public void run(){  
        for(int i = 1; i<=1000; i++){  
    }
```

```

        System.out.println(i + " ###");
        try{
            Thread.sleep(200);
        }catch(InterruptedException e){}
    }
}
}
}

```

然后编译运行主程序，输出结果如下：

```

1 $$$
1 ###
2 $$$
2 ###
.....
998 $$$
998 ###
999 $$$
999 ###
1000 $$$
1000 ###

```

从输出结果可以看出，这两个线程基本上是交替进行输出。我们可以结合线程状态图来分析一下运行的流程。

首先，创建了 t1 对象和 t2 对象之后，这两个线程都进入了初始状态。之后，随着主方法中调用了 t1.start() 以及 t2.start()，这两个方法就进入了可运行状态。接着主线程进入终止状态，操作系统会从 t1 和 t2 中挑选一个线程进入运行状态。

假设首先选中的是 t1 线程，此时这个线程会输出“1 \$\$\$”，执行完这个输出语句之后，执行 sleep 语句，使得 t1 线程就进入了阻塞状态。此时，没有线程处于运行状态，于是操作系统会从可运行状态中挑选一个线程进入运行状态。由于可运行状态中只有 t2 一个线程，因此 t2 线程得到运行。

t2 线程运行时，会首先输出“2 ###”，输出之后，会执行 sleep 语句，使得 t2 线程也进入了阻塞状态。此时，t1 和 t2 线程都处于阻塞状态，而没有线程处于可运行状态以及运行状态，因此这时没有代码执行。

当 200 毫秒过去之后，t1 线程的 sleep 方法返回，此时 t1 线程由阻塞状态进入可运行状态。由于之前可运行状态和运行状态都没有线程，因此，这时操作系统会选中唯一一个处于可运行状态的 t1 线程进入运行状态。

再之后，t2 线程的 sleep 方法返回，t2 也进入了可运行状态。于是……

.....

进过多次状态的反复和转换，t1 和 t2 两个线程都输出完毕之后，程序中所有线程都进入了终止状态，整个程序结束。

当然，如果多次运行的话，可以发现，t1 和 t2 两个线程的交替输出不是绝对的，有可能出现下面的情况：

```

5 ###
5 $$$
6 $$$

```

```
6 ###
7 ###
7 $$$
8 $$$
8 ###
9 $$$
9 ###
```

也就是说，利用 `sleep` 方法对线程的控制是非常不精确的。试图用 `sleep` 方法严格的要求数线程间进行交替输出，是非常的不可靠的。

### 3.3 join()

除了使用 `sleep()` 和等待 `IO` 之外，还有一个方法会导致线程阻塞，这就是线程的 `join()` 方法。我们首先来看下面一段代码：

```
class MyThread1 extends Thread{
    public void run(){
        for(int i = 0; i<100; i++){
            System.out.println(i + " $$$");
        }
    }
}

class MyThread2 extends Thread{
    public void run(){
        for(int i = 0; i<100; i++){
            System.out.println(i + " ###");
        }
    }
}

public class TestJoin{
    public static void main(String args[]){
        MyThread1 t1 = new MyThread1();
        MyThread2 t2 = new MyThread2();
        t1.start();
        t2.start();
    }
}
```

这段代码创建了两个线程对象，并且启动了两个新线程。这两个线程会交替输出`$$$`和`###`。

接下来，我们修改一下 `MyThread2` 类，并修改主方法，修改后的代码如下：

```
class MyThread2 extends Thread{
    Thread t;
```

```

public void run(){
    try{
        t.join();
    }catch(Exception e){}
    for(int i = 0; i<100; i++){
        System.out.println(i + " ###");
    }
}
}

public class TestJoin{
    public static void main(String args[]){
        MyThread1 t1 = new MyThread1();
        MyThread2 t2 = new MyThread2();
        t2.t = t1;
        t1.start();
        t2.start();
    }
}

```

上面这个程序中，`MyThread2` 对象增加了一个属性 `t`。在主方法中，把 `t1` 对象赋值给 `t`，也就是让 `t` 属性和 `t1` 引用指向同一个对象。

最重要的一点是，在 `MyThread2` 类的 `run()` 方法中，调用了 `t` 属性的 `join()` 方法。这个方法能够让 `MyThread2` 线程由运行状态进入阻塞状态，直到 `t` 线程结束。下面结合程序的状态转换图，来说明一下执行的过程。

首先 `main` 方法中，创建了两个线程对象，并且调用了 `t1` 和 `t2` 线程的 `start()` 方法，这样，这两个线程就进入了可运行状态。假设操作系统首先挑选了 `t1` 线程进入了可运行状态，于是输出若干个 “\$\$\$”。经过一段时间之后，由于 CPU 时间片到期，`t1` 线程进入了可运行状态。假设经过了一段时间之后，操作系统选择了 `t2` 线程进入了可运行状态。进入了 `t2` 线程的 `run()` 方法之后，调用了 `t` 属性的 `join()` 方法。由于 `t` 属性与 `t1` 指向同一个对象，因此这也意味着在 `t2` 线程中，调用了 `t1` 线程的 `join()` 方法。调用之后，`t2` 线程会进入阻塞状态。此时，运行状态没有线程在执行，因此系统会从可运行状态中选择一个线程执行。由于可运行状态此时只有一个 `t1` 线程，因此这个线程会一直占用 CPU，直到线程代码执行结束。当 `t1` 线程结束之后，`t2` 才会由阻塞状态进入可运行状态，此时才能够执行 `t2` 的代码。因此，从输出结果上来看，会先执行 `t1` 线程的所有代码，然后再执行 `t2` 线程的所有代码。部分输出结果如下：

```

0 $$$
1 $$$
2 $$$
.....
97 $$$
98 $$$
99 $$$
0 ###
1 ###

```

```
2 ###
3 ###
.....
97 ###
98 ###
99 ###
```

要注意的是，我们在 t2 线程中调用 t1 线程的 `join()` 方法，结果是 t2 阻塞，直到 t1 线程结束。t2 线程是 `join()` 方法的调用者，而 t1 线程是被调用者，在调用 `join()` 方法的过程中，方法的调用者被阻塞，阻塞到被调用的线程结束。

我们可以用一个生活中的比喻来解释 `join()` 方法。假设顾客到饭店里去吃饭，那么每一个顾客就可以认为是一个线程。顾客点菜，就可以当做是顾客线程要求启动一个厨师线程为自己做饭，在做饭过程中，顾客线程只能等待。因此，这也可以当做是顾客线程调用了厨师线程的 `join()` 方法，等厨师做完饭了，顾客才能继续下一步：吃饭。在这个例子中，顾客线程就调用了厨师线程的 `join()` 方法。

在调用 `join` 方法的过程中，要注意，不能让两个线程相互 `join()`。例如，如果一个顾客点菜，相当于调用了厨师的 `join()` 方法，顾客打算等厨师做完饭以后，吃完饭再给钱；而厨师呢，在拿到顾客下的单之后，希望顾客先给钱，之后再开始做饭，于是厨师也调用了顾客的 `join()` 方法。这样的结果就是两边互相等待，结果谁都无法继续下去。我们拿代码模拟一下这种情况：

```
class MyThread1 extends Thread{
    Thread t;
    public void run() {
        try{
            t.join();
        }catch(Exception e){}
        for(int i = 0; i<100; i++){
            System.out.println(i + " $$$");
        }
    }
}

class MyThread2 extends Thread{
    Thread t;
    public void run() {
        try{
            t.join();
        }catch(Exception e){}
        for(int i = 0; i<100; i++){
            System.out.println(i + " ###");
        }
    }
}
```

```

public class TestJoin{
    public static void main(String args[]){
        MyThread1 t1 = new MyThread1();
        MyThread2 t2 = new MyThread2();
        t2.t = t1;
        t1.t = t2;
        t1.start();
        t2.start();
    }
}

```

这段代码会迟迟无法运行下去，原因就在于，`t1` 线程等待 `t2` 线程结束，而 `t2` 线程等待 `t1` 线程结束，这样的结果就是两边的线程都处于阻塞状态而无法运行。

那怎么解决这个问题呢？首先，写程序的时候应当小心，尽量不应该出现这样的代码。因为这样的代码在编译和运行时都不会出现任何错误和异常。另一方面，Java 也为 `join()` 方法提供了一个替换的方案。

在 `Thread` 类中，除了有一个无参的 `join()` 方法之外，还有一个有参的 `join()` 方法，方法签名如下：

```
public final void join(long millis) throws InterruptedException
```

这个方法接受一个 `long` 类型作为参数，表示 `join()` 最多等待多少毫秒。也就是说，调用这个 `join()` 方法的时候，不会一直处于阻塞状态，而是有一个时间限制。就好像顾客等待厨师做饭时，不会无限制的等下去，如果菜一段时间内还不上，则顾客就会离开，而不会一直傻等下去。利用这个方法，修改上面的 `MyThread1` 类：

```

class MyThread1 extends Thread{
    Thread t;
    public void run(){
        try{
            t.join(1000);
        } catch(Exception e){}
        for(int i = 0; i<100; i++){
            System.out.println(i + " $$$");
        }
    }
}

```

这样，`MyThread1` 就不会无限制的等待下去，而是当等待 1000 毫秒之后，就会从阻塞状态转为可运行状态，从而执行下去。

## 4 线程同步

下面要介绍的是线程中非常重要的一部分：线程同步。这部分的课程并不是太好理解，希望各位读者在学习过程中，能够耐心和细致一些。

### 4.1 临界资源与数据不一致

我们首先来看一个例子。我们使用数组来实现一个“栈”的数据结构。

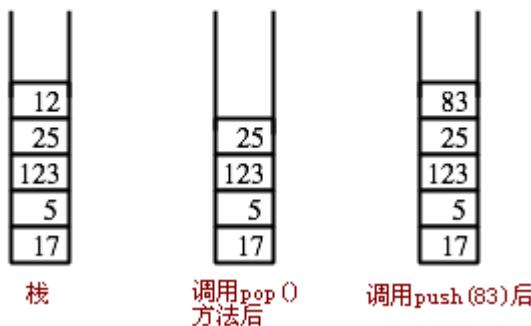
栈在表示上，就如同一个单向开口的盒子，每当有新数据进入时，都是进入栈顶。其基本操作为 push 和 pop。push 表示把一个元素加入栈顶，pop 表示把栈顶元素弹出。

示意图如下：

### 栈是一种数据结构

pop()方法表示把栈顶端的元素返回，并删除该元素。如下图：调用pop()方法后

push()方法表示把数据加入栈，加入时新数据放在栈顶。如下图：调用push(83)后



我们利用一个数组来实现栈操作，相关代码如下：

```
class MyStack{
    char[] data = {'A', 'B', ' '};
    int index = 2;
    public void push(char ch){
        data[index] = ch;
        index++;
    }

    public void pop(){
        index--;
        data[index] = ' ';
    }

    public void print(){
        for(int i = 0; i<data.length; i++){
            System.out.print(data[i] + "\t");
        }
        System.out.println();
    }
}

public class TestMyStack{
    public static void main(String args[]){
        MyStack ms = new MyStack();
        ms.push('C');
        ms.print();
        ms.pop();
    }
}
```

```
        ms.print();
    }
}
```

上面的代码会输出：

```
A  B  C  
A  B
```

符合我们对栈的认识。MyStack 类中的 data 属性表示用来储存栈的信息，而 index 这个变量保存的是栈中有效元素的个数。

然后，我们为 push 方法增加一个 Thread.sleep()语句。修改后的 push()方法如下：

```
public void push(char ch) {
    data[index] = ch;
    try{
        Thread.sleep(1000);
    }catch(Exception e){}
    index++;
}
```

在修改 data 值和维护 index 中间，增加了一个 sleep()方法。增加了这个方法之后，运行结果没有区别。

接下来，我们使用多线程来访问 MyStack 类。创建两个线程类：PopThread 和 PushThread，这两个类用来访问 MyStack 对象。PushThread 线程负责向栈中添加一个字符，而 PopThread 线程负责从栈中取出一个字符。修改后完整的代码如下：

```
class MyStack{
    char[] data = {'A', 'B', ' '};
    int index = 2;
    public void push(char ch) {
        data[index] = ch;
        try{
            Thread.sleep(1000);
        }catch(Exception e){}
        index++;
    }

    public void pop() {
        index--;
        data[index] = ' ';
    }

    public void print(){
        for(int i = 0; i<data.length; i++){
            System.out.print(data[i] + "\t");
        }
        System.out.println();
    }
}
```

```

}

class PopThread extends Thread{
    MyStack ms;
    public PopThread(MyStack ms) {
        this.ms = ms;
    }
    public void run(){
        ms.pop();
        ms.print();
    }
}

class PushThread extends Thread{
    MyStack ms;
    public PushThread(MyStack ms) {
        this.ms = ms;
    }
    public void run(){
        ms.push('C');
        ms.print();
    }
}

public class TestMyStack{
    public static void main(String args[]){
        MyStack ms = new MyStack();
        Thread t1 = new PushThread(ms);
        Thread t2 = new PopThread(ms);

        t1.start();
        t2.start();
    }
}

```

上述代码的输出结果为：

```

A      C
A      C

```

注意，这就产生了问题：我们的栈要求后入先出，栈内原有元素 A、B，结果先启动了 Push 线程，后启动 Pop 线程之后，却把 B 元素 pop 出去，而把 C 元素保留了。更关键的是，C 元素和 A 元素中有一个空位，也就是说，栈内的元素变得不连续了！

很显然，这是一个程序运行中的错误。那为什么会造成这个错误呢？

我们结合线程状态的转换来看这个问题。在主方法中，t1 和 t2 线程都被调用了 start()

方法被启动。假设 t1 线程先进入运行状态，而 t2 线程保持在可运行状态。此外，要注意的是，t1 线程和 t2 线程中，都包含有一个 MyStack 类型的引用。在主方法中，我们只创建了一个 MyStack 类型的对象，并且通过构造方法，让 t1 和 t2 的 ms 属性都指向了同一个 MyStack 类型的对象。

在 t1 线程进入运行状态之后，它会调用 MyStack 类中的 push('C') 方法。这个在执行这个方法过程中，会首先修改 data 数组的值，把下标为 2 的位置设为 C。之后，应当把 index 加 1，但是线程调用了 sleep() 方法，使得 t1 线程进入了阻塞状态。此时，data 数组中有 3 个元素，但是 index 却是 2！这个时候，表示栈的两个数据 data 数组和 index，出现了信息不一致的情况。

当 t1 处于阻塞状态的时候，t2 线程处于可运行状态。于是，操作系统就会选中 t2 线程进行运行。此时，t2 的 run() 方法中会调用 pop 方法，会首先把 index 减 1，使得 index 的值为 1，然后把 data 数组中下标为 1 的元素设为空。要注意的是，现在栈顶元素应当是 data 数组中下标为 2 的 C，而不是下标为 1 的元素 B！这样就造成了我们上面运行的结果，破坏了栈的数据结构。

下图说明了元素入栈和出栈的正确情况：

0	1	2	index=	运行状态线程	代码	备注
A	B		2			初始状态
A	B	C	2	t1	data[index]='C'	t1 线程执行了 C 字符入栈
A	B	C	3	t1	index++	t1 线程执行 index 加 1，入栈过程完成
A	B	C	2	t2	index--	
A	B		2	t2	data[index]=''	t2 线程执行出栈过程完成 打印 ‘C’ 出栈

而下图则揭示了出现错误的数组情况：

0	1	2	index=	运行状态线程	代码	备注
A	B		2			初始状态
A	B	C	2	t1	data[index]='C'	t1 线程执行了 C 字符入栈
A	B	C	1	t2	index--	t1 休眠 t2 获得 CPU 时间片
A		C	1	t2	data[index]=''	t2 执行出栈过程完成 打印 ‘B’ 出栈
A		C	2	t1	index++	t1 恢复运行，执行 index++

造成这种错误的原因是什么呢？第一个原因：t1 线程中的两个步骤：1、修改 data 数组；2、index 变量加 1。这两个步骤当一起完成，才能组成一个完整的 push() 操作；如果这两个步骤中只完成了一个，就会产生问题。具体来说，就会产生数据不一致的问题，对于 MyStack

来说，所谓的数据不一致，指的就是 `data` 数组中元素的个数和 `index` 所表示的元素的个数不一致。

第二个原因，则是因为有 `t1` 和 `t2` 两个线程，这两个线程并发访问同一个 `MyStack` 对象。由于当 `t1` 线程没有完成的操作的时候，`t2` 线程就开始对 `MyStack` 对象进行了访问，从而就会造成数据不一致。

总结一下上面的两个原因：多个线程并发访问同一个对象，如果破坏了不可分割的操作，则有可能产生数据不一致的情况。

这其中，有两个专有名词：被多个线程并发访问的对象，也被称之为“临界资源”；而不可分割的操作，也被称之为“原子操作”。产生的数据不一致的问题，也被称之为“同步”问题。要产生同步问题，多线程访问“临界资源”，破坏了“原子操作”，这两个条件缺一不可。

## 4.2 synchronized

### 4.2.1 锁的概念

分析了产生同步问题的原因之后，下面就应该设法解决同步问题。那同步问题应该怎么处理和解决呢？

首先，由于在 `push()` 方法中存在 `sleep()` 方法才造成了原子操作被破坏，那如果把 `sleep()` 方法去掉，是不是就能解决同步的问题了呢？

要注意的是，把 `sleep()` 方法去掉之后，这样的代码可能运行时一时不会出问题，但是并不代表这代码是没有安全问题的。例如，当 `push` 线程进行了 `push` 的第一步操作之后，CPU 时间片到期了，然后接下来换成 `pop` 线程运行，此时，上一节我们描述的同步问题，同样有可能发生。

虽然上面所说的情况可能发生的概率非常的低，但是却不能不说这是代码中的一个隐患。并且，当我们的程序成为一个大型企业应用系统的一部分时，由于每天都有大量的客户访问我们的程序，也就有大量的线程在同时访问同一个对象，此时，无论发生同步问题的概率有多小，在大量访问和重复的过程中，发生问题几乎是必然的。而同步问题一旦发生，就有可能造成极为严重的损失。因此，我们在写代码，尤其是有可能被多线程访问的类和对象时，一定要慎重设计，把所有的隐患都杜绝。

那应该怎么杜绝这个隐患呢？我们首先从生活中的一个例子说起。

有两位电工，长期驻扎在一个小区作为物业，一个电工 `A`，上早班，上班时间是 `6: 00 ~ 18: 00`，另一位电工 `B`，上班时间为 `18: 00 ~ 6: 00`。这两位电工轮流在小区值班。

某一天，在下午 `17: 55` 的时候，电工 `A` 接到小区居民投诉：小区中的电压不稳，希望电工能够修复一下。虽然马上要到下班时间了，但是电工 `A` 还是决定去查看一下。为了爬上电线杆进行高空带电作业，于是电工 `A` 暂时把小区的电闸给关了，然后再到高空进行电压不稳的检查。

在 `A` 在工作过程中，不知不觉到了 `18: 01`，电工 `B` 上班了。这个时候，电工 `B` 接到小区居民投诉：小区停电了。电工 `B` 也决定去查看一下。当他查看到电闸时，一下就明白了：怪不得小区停电呢，谁把闸关了啊。于是，电工 `B` 就把电闸打开了。

于是，听到一声惨叫，`A` 从天上掉了下来……

半个月以后，等 `A` 把伤养得差不多了以后，变电所决定解决吸取教训，争取再也不要发生这样的问题。那这个问题是怎么发生的呢？

我们可以把 A 和 B 当做是两个线程，而电闸就当做是临界资源。因此，这样就形成了两个线程共同访问临界资源，由于 A 检修电路时，“关掉电闸、爬上电线杆检修、打开电闸恢复供电”是不可分割的原子操作，而一旦其中的某一步被打断，就有可能产生问题，这就是两个线程数据不一致的情况。

那怎么办呢？变电所决定，要解决这个问题，这就借助于一样东西：锁。在电闸上挂一个挂锁。平时没有问题时，锁是打开的；但是一旦有一个电工需要操作电闸的话，为了防止别人动电闸，他可以把电闸给锁上，并把唯一的钥匙随身携带。这样，当他进行原子操作时，由于临界资源被他锁上了，其他线程就访问不了这个临界资源，因此就能保证他的原子操作不被破坏。

#### 4.2.2 **synchronized** 与同步代码块

Java 中采取了类似的机制，也采用锁来保护临界资源，防止数据不一致的情况产生。下面我们就来介绍 Java 中的同步机制以及 **synchronized** 关键字。

在 Java 中，每个对象都拥有一个“互斥锁标记”，这就好比是我们说的挂锁。这个锁标记，可以用来分给不同的线程。之所以说这个锁标记是“互斥的”，因为这个锁标记同时只能分配给一个线程。

光有锁标记还不行，还要利用 **synchronized** 关键字进行加锁的操作。**synchronized** 关键字有两种用法，我们首先介绍第一种：**synchronized + 代码块**。

这种用法的语法如下：

```
synchronized(obj){  
    代码块...  
}
```

**synchronized** 关键字后面跟一个圆括号，括号中的是某一个引用，这个引用应当指向某一个对象。后面紧跟一个代码块，这个代码块被称为“同步代码块”。

这种语法的含义是，如果某一个线程想要执行代码块中的代码，必须要先获得 **obj** 所指向对象的互斥锁标记。也就是说，如果有线程 **t1** 要想进入同步代码块，必须要获得 **obj** 对象的锁标记；而如果 **t1** 线程正在同步代码块中运行，这意味着 **t1** 有着 **obj** 对象的互斥锁标记；而这个时候如果有一个 **t2** 线程想要访问同步代码块，会因为拿不到 **obj** 对象的锁标记而无法继续运行下去。

需要注意的是，**synchronized** 与同步代码块是与对象紧密结合在一起的，加锁是对对象加锁。例如下面的例子，假设有两个同步代码块：

```
synchronized(obj1){  
    代码块 1;  
}
```

```
synchronized(obj1){  
    代码块 2;  
}
```

```
synchronized(obj2){  
    代码块 3;
```

```
}
```

假设有一个线程 t1 正在代码块 1 中运行，那假设另有一个线程 t2，这个 t2 线程能否进入代码块 2 呢？能否进入代码块 3 呢？

由于 t1 正在代码块 1 中运行，这也就意味着 obj1 对象的锁标记被 t1 线程获得，而此时 t2 线程如果要进入代码块 2，也必须要获得 obj1 对象的锁标记。但是由于这个标记正在 t1 手中，因此 t2 线程无法获得锁标记，因此 t2 线程无法进入代码块 2。

但是 t2 线程能够进入代码块 3，原因在于：如果要进入代码块 3 中，要获得的是 obj2 对象的锁标记，这个对象与 obj1 不是同一个对象，此时 t2 线程能够顺利的获得 obj2 对象的锁标记，因此能够成功的进入代码块 3。

从上面这个例子中，我们可以看出，在分析、编写同步代码块时，一定要搞清楚，同步代码块锁的是哪个对象。只有把这个问题搞清楚了之后，才能正确的分析多线程以及同步的相关问题。

我们下面利用同步代码块修改一下 MyStack 类，为这个类增加同步的机制。

首先，要为 MyStack 类增加一个属性 lock，这个属性用来表示我们所说的“锁”。利用这个对象的互斥锁标记，我们完成对 MyStack 的同步。

修改之后的 MyStack 代码如下：

```
class MyStack{
    char[] data = {'A', 'B', ' '};
    int index = 2;
    private Object lock = new Object();
    public void push(char ch){
        synchronized(lock){
            data[index] = ch;
            try{
                Thread.sleep(1000);
            }catch(Exception e){}
            index++;
        }
    }

    public void pop(){
        synchronized(lock){
            index--;
            data[index] = ' ';
        }
    }

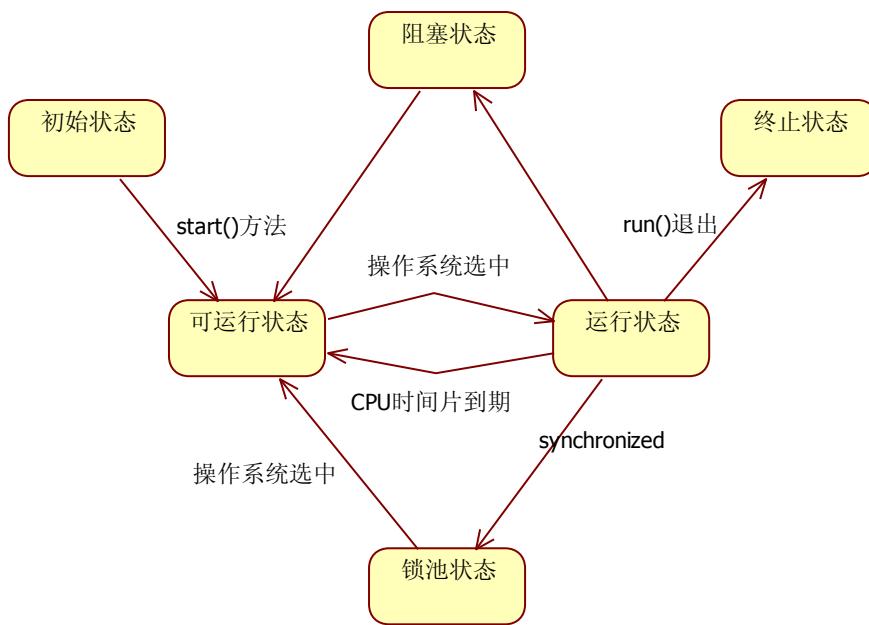
    public void print(){
        for(int i = 0; i<data.length; i++){
            System.out.print(data[i] + "\t");
        }
        System.out.println();
    }
}
```

}

在 `push` 方法和 `pop` 方法中，我们对 `lock` 属性进行加锁。也就是说，线程如果要执行 `pop` 或者 `push` 时，需要获得 `lock` 对象的互斥锁标记。

下面我们结合线程的状态转换，来考察一下 `synchronized` 关键字在程序运行中的作用。

首先，如果一个线程获得不了某个对象的互斥锁标记，这个线程就会进入一个状态：锁池状态。如下图：



当运行中的线程，运行到某个同步代码块，但是获得不了对象的锁标记时，会进入锁池状态。在锁池状态的线程，会一直等待某个对象的互斥锁标记。如果有多个线程都需要获得同一个对象的互斥锁标记，则可以有多个线程进入锁池，而某个线程获得锁标记，执行同步代码块中的代码。

当对象的锁标记被某一个线程释放之后，其他在锁池状态中的线程就可以获得这个对象的锁标记。假设有多个线程在锁池状态中，那么会由操作系统决定，把释放出来的锁标记分配给哪一个线程。当在锁池状态中的线程获得锁标记之后，就会进入可运行状态，等待获得 CPU 时间片，从而运行代码。

下面我们结合线程状态的转换，来分析一下修改之后的 `MyStack` 代码。

当 `t1` 线程启动的时候，它会首先执行 `push` 方法。要想执行 `push` 方法，必须要获得 `lock` 对象的互斥锁标记。由于此时 `lock` 对象的锁标记没有分配给其他线程，因此这个锁标记被分配给 `t1` 线程。当 `t1` 线程调用 `sleep()` 方法之后，`t1` 进入阻塞状态，于是 `t2` 线程开始运行。由于 `t2` 线程调用 `pop()` 方法，要调用这个方法，也必须获得 `lock` 对象的互斥锁标记。由于这个锁标记已经分配给了 `t1` 线程，因此 `t2` 线程只能进入 `lock` 对象的锁池，从而进入了锁池状态。

之后，当 `t1` 线程 `sleep()` 方法结束，`t1` 重新进入可运行状态→运行状态，执行完了整个 `push()` 方法，退出对 `lock` 加锁的同步代码块。此时，`t2` 线程就能获得 `lock` 对象的锁标记，于是 `t2` 线程就由锁池状态转为了可运行状态。

从上面的分析我们可以看出，虽然 `MyStack` 对象还是被两个线程 `t1`、`t2` 同时访问，但是由于对 `lock` 对象加锁的原因，当 `t1` 线程执行 `push` 方法执行到一半进入阻塞状态时，`t2` 线程同样无法操作 `MyStack` 对象。这样，就不会破坏原子操作 `push()`，从而保护了临界资源，解决了同步问题。

加入了锁机制之后的数组情况如下图：

0	1	2	index=	运行状态 线程	代码	备注
A	B		2			初始状态
A	B	C	2	t1	<code>data[index]='C'</code>	t1 线程执行了 C 字符入栈
A	B	C	2	t2		t1 休眠，但 t2 无法获得锁对象的锁标记，因此 t2 无法执行代码
A	B	C	3	t1	<code>index++</code>	t1 线程执行 index 加 1，入栈过程完成
A	B	C	2	t2	<code>index--</code>	t2 获得锁标记，执行代码
A	B		2	t2	<code>data[index] = ''</code>	t2 线程执行出栈过程完成 打印 ‘C’ 出栈

#### 4.2.3 同步方法

在上面的例子中，我们专门创建了一个 `Object` 类型的 `lock` 对象，用来在 `pop` 方法和 `push` 方法中加锁。然而，对于上面的程序来说，除了 `lock` 对象有锁标记之外，`MyStack` 对象本身，也具有互斥锁标记。对于 `pop` 方法和 `push` 方法来说，也能够对 `MyStack` 对象（也就是所谓的“当前对象”）加锁。例如：

```
class MyStack{
    char[] data = {'A', 'B', ' '};
    int index = 2;
    public void push(char ch){
        synchronized(this) {
            data[index] = ch;
            try{
                Thread.sleep(1000);
            }catch(Exception e){}
            index++;
        }
    }

    public void pop(){
        synchronized(this){
            index--;
            data[index] = ' ';
        }
    }
}
```

```

    }

    public void print(){
        for(int i = 0; i<data.length; i++){
            System.out.print(data[i] + "\t");
        }
        System.out.println();
    }

}

```

在上面的代码中，我们去掉了 `lock` 属性，并且对 “`this`” 加锁，也就是对当前对象加锁。上面的代码同样能够完成我们同步的要求。

对于这种，在整个方法内部对 “`this`” 加锁的情况，我们可以使用 `synchronized` 作为修饰符修饰方法，来表达同样的意思。

用 `synchronized` 关键字修饰的方法称之为同步方法，所谓的同步方法，指的是同步方法中整个方法的实现，需要对 “当前对象” 加锁。哪个线程能够拿到对象的锁标记，哪个线程才能调用对象的同步方法。

上面的 `MyStack` 代码可以等价的改为下面的情况：

```

class MyStack{
    char[] data = {'A', 'B', ' '};
    int index = 2;
    public synchronized void push(char ch){
        data[index] = ch;
        try{
            Thread.sleep(1000);
        }catch(Exception e){}
        index++;
    }

    public synchronized void pop(){
        index--;
        data[index] = ' ';
    }

    public void print(){
        for(int i = 0; i<data.length; i++){
            System.out.print(data[i] + "\t");
        }
        System.out.println();
    }

}

```

当一个线程 `t1` 在访问某一个对象的同步方法时（例如 `pop`），另一个线程 `t2` 能否访问同

一个对象的其他同步方法（例如 `push`）？

我们举例来说：假设有一个 `MyStack` 对象 `ms`，一个线程 `t1` 正在访问 `pop` 方法，这意味着 `ms` 对象的互斥锁标记正在 `t1` 线程手中。而另一个线程 `t2`，如果想要方法 `push` 方法的话，必须要获得 `ms` 对象的互斥锁标记。由于这个锁标记正在 `t1` 线程手中，因此 `t2` 线程无法获得，从而 `t2` 线程就无法访问 `push` 方法。因此，这是一个结论：

当一个线程正在访问某个对象的同步方法时，其他线程不能访问同一个对象的任何同步方法。

## 5 wait 与 notify

在 `synchronized` 关键字的作用下，还有可能产生新的问题：死锁。

### 5.1 死锁

考虑下面的代码，假设 `a` 和 `b` 是两个不同的对象。

```
synchronized(a){  
    ...//1  
    synchronized(b){  
        ...  
    }  
    synchronized(b){  
        ... //2  
        synchronized(a){  
            ...  
        }  
    }  
}
```

假设现在有两个线程，`t1` 线程运行到了//1 的位置，而 `t2` 线程运行到了//2 的位置，接下来会发生什么情况呢？

此时，`a` 对象的锁标记被 `t1` 线程获得，而 `b` 对象的锁标记被 `t2` 线程获得。对于 `t1` 线程而言，为了进入对 `b` 加锁的同步代码块，`t1` 线程必须获得 `b` 对象的锁标记。由于 `b` 对象的锁标记被 `t2` 线程获得，`t1` 线程无法获得这个对象的锁标记，因此它会进入 `b` 对象的锁池，等待 `b` 对象锁标记的释放。而对于 `t2` 线程而言，由于要进入对 `a` 加锁的同步代码块，由于 `a` 对象的锁标记在 `t1` 线程手中，因此 `t2` 线程会进入 `a` 对象的锁池。

此时，`t1` 线程在等待 `b` 对象锁标记的释放，而 `t2` 线程在等待 `a` 对象锁标记的释放。由于两边都无法获得所需的锁标记，因此两个线程都无法运行。这就是“死锁”问题。

### 5.2 wait 与 notify

在 Java 中，采用了 `wait` 和 `notify` 这两个方法，来解决死锁机制。

首先，在 Java 中，每一个对象都有两个方法：`wait` 和 `notify` 方法。这两个方法是定义在 `Object` 类中的方法。对某个对象调用 `wait()` 方法，表明让线程暂时释放该对象的锁标记。例如，上面的代码就可以改成：

```
synchronized(a){  
    ...//1  
    a.wait();  
    synchronized(b){  
        ...  
    }  
}
```

```

    }
}

synchronized(b){
    ... //2
    synchronized(a){
        ...
        a.notify();
    }
}

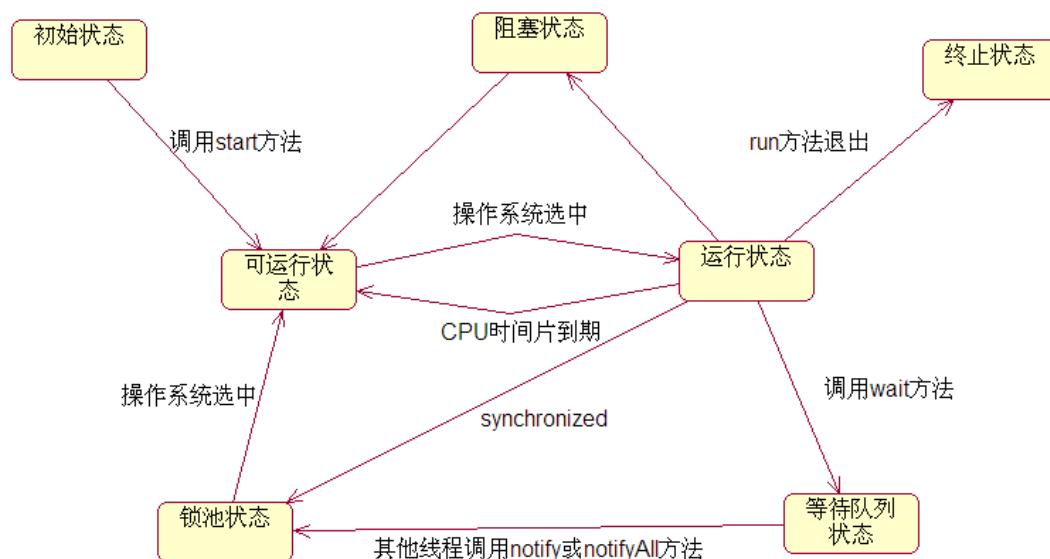
```

这样的代码改完之后，在//2 后面，t1 线程就会调用 a 对象的 wait 方法。此时，t1 线程会暂时释放自己拥有的 a 对象的锁标记，而进入另外一个状态：等待状态。

要注意的是，如果要调用一个对象的 wait 方法，前提是线程已经获得这个对象的锁标记。如果在没有获得对象锁标记的情况下调用 wait 方法，则会产生异常。

由于 a 对象的锁标记被释放，因此，t2 对象可以获得 a 对象的锁标记，从而进入对 a 加锁的同步代码块。在同步代码块的最后，调用 a.notify()方法。这个方法与 wait 方法相对应，是让一个线程从等待状态被唤醒。

那么 t2 线程唤醒 t1 线程之后，t1 线程处于什么状态呢？由于 t1 线程唤醒之后还要在对 a 加锁的同步代码块中运行，而 t2 线程调用了 notify()方法之后，并没有立刻退出对 a 加锁的同步代码块，因此此时 t1 线程并不能马上获得 a 对象的锁标记。因此，此时，t1 线程会在 a 对象的锁池中进行等待，以期待获得 a 对象的锁标记。也就是说，一个线程如果之前调用了 wait 方法，则必须要被另一个线程调用 notify()方法唤醒。唤醒之后，会进入锁池状态。线程状态转换图如下：



由于可能有多个线程先后调用 a 对象 wait 方法，因此在 a 对象等待状态中的线程可能有多个。而调用 a.notify()方法，会从 a 对象等待状态中的多个线程里挑选一个线程进行唤醒。与之对应的，有一个 notifyAll()方法，调用 a.notifyAll() 会把 a 对象等待状态中的所有线程都唤醒。

### 5.3 wait 与 notify 应用：生产者/消费者问题

下面我们结合一个实际的例子，来为大家演示如何使用 `wait` 和 `notify / notifyAll` 方法。我们使用一个数组来模拟一个我们比较熟悉的数据结构：栈。代码如下所示：

```
class MyStack{  
    private char[] data = new char[5];  
    private int index = 0;  
  
    public char pop(){  
        index -- ;  
        return data[index];  
    }  
  
    public void push(char ch){  
        data[index] = ch;  
        index++;  
    }  
  
    public void print(){  
        for (int i=0; i<index; i++){  
            System.out.print(data[i] + "\t");  
        }  
        System.out.println();  
    }  
    public boolean isEmpty(){  
        return index == 0;  
    }  
  
    public boolean isFull(){  
        return index == 5;  
    }  
}
```

注意，我们为 `MyStack` 增加了两个方法，一个用来判断栈是否为空，一个用来判断栈是否已经满了。

然后我们创建两个线程，一个线程每隔一段随机的时间，就会往栈中增加一个数据；另一个线程每隔一段随机的时间，就会从栈中取走一个数据。为了保证 `push` 和 `pop` 的完整性，在线程中应当对 `MyStack` 对象加锁。

但是我们发现，入栈线程和出栈线程并不是在任何时候都可以工作的。当数组满了的时候，入栈线程将不能工作；当数组空了的时候，出栈线程也不能工作。违反了上面的条件，我们将得到一个数组下标越界异常。

为此，我们可以用 `wait/notify` 机制。在入栈线程执行入栈操作时，如果发现数组已满，则会调用 `wait` 方法，去等待。同样，出栈线程在执行出栈操作时，如果发现数组已空，同样调用 `wait` 方法去等待。在入栈线程结束入栈工作之后，会调用 `notifyAll` 方法，释放那些正在等待的出栈线程（因为数组现在已经不是空的了，他们可以恢复工作了）。同样，当出栈线程结束出栈工作之后，也会调用 `notifyAll` 方法，释放正在等待的入栈线程。

相关代码如下：

```
class Consumer extends Thread{
    private MyStack ms;

    public Consumer(MyStack ms) {
        this.ms = ms;
    }

    public void run() {
        while(true) {
            //为了保证push和pop操作的完整性
            //必须加synchronized
            synchronized(ms) {
                //如果栈空间已满，则wait()释放ms的锁标记
                while(ms.isEmpty()) {
                    try {
                        ms.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                char ch = ms.pop();
                System.out.println("Pop " + ch);
                ms.notifyAll();
            }
            //push之后随机休眠一段时间
            try {
                sleep( (int) Math.abs(Math.random() * 100) );
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

//生产者
class Producer extends Thread{
    private MyStack ms;

    public Producer(MyStack ms) {
        this.ms = ms;
    }
```

```
public void run() {
    while(true) {
        //为了保证 push 和 pop 操作的完整性
        //必须加 synchronized
        synchronized(ms) {
            //如果栈空间已满，则 wait() 释放 ms 的锁标记
            while(ms.isFull()) {
                try {
                    ms.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            ms.push('A');
            System.out.println("push A");
            ms.notifyAll();
        }
        //push 之后随机休眠一段时间
        try {
            sleep( (int) Math.abs( Math.random() * 200 ) );
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class TestWaitNotify {
    public static void main(String[] args) {
        MyStack ms = new MyStack();
        Thread t1 = new Producer(ms);
        Thread t2 = new Consumer(ms);
        t1.start();
        t2.start();
    }
}
```

# Chp14 I/O 框架

在程序运行的过程当中，JVM 的内存中必然会存放很多数据，包括基本类型和对象类型。但是当程序结束，JVM 关闭的时候，这些数据必然会随之消失。我们可能希望通过某种方式，让这些数据能够保存下来，以备在此使用。因此我们会把数据存入文件，或通过网络发送出去，或存入数据库。反之，我们当然也需要用某种方式，把保存的数据重新读回 JVM。这些，都涉及到 JVM 与外部进行数据交换。

将 JVM 中的数据写出去，我们称为数据的输出。反之，将数据读入 JVM，我们称之为数据的输入。因此，Java 中解决这部分问题的 API 被称为 I/O。（I 是英语 Input 的首字母，表示输入，O 是英语 Output 的首字母，表示输出）

## 1 Java I/O 概览

### 1.1 文件系统和 File 类

首先介绍一下 Java 中的 File 类。这个类在 java.io 包中，对于一个 File 对象来说，它能够代表硬盘上的一个文件或者文件夹。在这句描述中，有两个要点值得注意：

- 1、File 对象不仅能够代表一个文件，还能够代表一个文件夹。
- 2、File 对象是“代表”一个文件或者文件夹。怎么来理解“代表”两个字呢？

首先，当我们创建一个 File 对象时，指的是在内存中分配了一块数据区域，也就是说，创建一个 File 对象并不会在系统中真的创建一个文件或者文件夹，而只是在 JVM 的内存中创建了一个对象。当然，这个对象能够用来跟底层系统打交道，从而通过这个对象能够跟 OS 打交道，从而操作底层的文件。

其次，既然 File 对象是“代表”OS 中的文件，因此并不要求 File 对象所代表的文件或者文件夹在 OS 中一定存在。也就是说，File 对象所代表的文件或者文件夹可能存在。

下面我们来结合 JDK 的文档，来看一下 File 类中都有哪些值得注意的方法，以及如何使用 File 类。

首先是 File 类的构造方法。File 对象有四个构造方法，其中三个构造方法比较常用。罗列如下：

`File(String pathname)`：利用字符串作为参数，表示一个路径名。用来创建一个代表给定参数的文件或者文件夹。

`File(String parent, String child)`：parent 表示父目录，child 表示在 parent 目录下的路径。这种写法用来代表名字为 parent/child 的文件或者文件夹。

`File(File parent, String child)`：同样用 parent 表示父目录，只不过这个 parent 是用 File 类型来表示。

需要注意的是，在创建 File 对象的时候，需要指定文件的路径，指定的时候，可以用绝对路径，也可以用相对路径。

另外，要注意路径分隔符的问题。在 Windows 中，路径分隔符使用的是反斜杠 “\”，而在 Java 中反斜杠是用来转义的，因此如果要使用反斜杠的话，必须使用 “\\” 来表示一个反斜杠。

例如，如果要表示 D 盘的 abc 目录，则在 Java 中的字符串应当这样写：

`"D:\\abc"`

另外，也可以用一个正斜杠来做 Windows 中的路径分隔，这样就不需要转义。因此，

也可以使用下面的表示方式：

```
"D:/abc"
```

这两种路径的写法都是正确的。

其次是 File 类中的一些基本操作。

`createNewFile()`：这个方法可以用来创建一个新文件。需要注意的是，如果这个文件在系统中已经存在，`createNewFile` 方法不会覆盖原有文件。

`mkdir()` / `mkdirs()`：这两个方法都可以用来创建文件夹。所不同的是，`mkdir` 只能创建一层文件夹，而 `mkdirs` 能够创建多层文件夹。

例如，如果当前目录下已经存在一个 abc 目录，则下面的代码能够成功创建一个目录：

```
File dir = new File("abc/def");
dir.mkdir();
```

由于已经有了 abc 目录，所以需要创建的仅仅是一层 def 目录而已。此时可以调用 `mkdir()` 方法来创建目录。

而如果当前目录下不存在 abc 目录，创建时需要先创建一个 abc 目录，然后创建 def 目录，属于创建两层目录。因此，这就需要调用 `mkdirs()` 方法来创建多层文件夹。

`delete()`：这个方法能够删除 File 所代表的文件或者文件夹。

`deleteOnExit()`：这个方法也能用来删除文件或者文件夹。所不同的是，`delete()` 方法被调用时，这个文件或者文件夹会被立刻删除，而 `deleteOnExit()` 方法被调用后，文件或者文件夹并不会立刻被删除，而会等到程序退出以后再删除。

`getPath()`：返回路径。

`getName()`：返回文件名

`getParent()`：返回所在的文件夹

对于上面这三个方法，我们用一段代码来说明。我们创建一个 File 对象，表示当前目录的 abc 目录下的 test.txt 文件，并分别调用上述三个方法。代码如下：

```
import java.io.*;
public class TestFile{
    public static void main(String args[]){
        File f = new File("abc/test.txt");
        System.out.println(f.getPath());
        System.out.println(f.getParent());
        System.out.println(f.getName());
    }
}
```

输出结果如下：

```
D:\book>java TestFile
abc\test.txt
abc
test.txt
D:\book>
```

可以看出，`getPath()` 返回的是我们给出的路径，`getParent()` 返回的所在的父文件夹，而

getName()返回的是文件名。

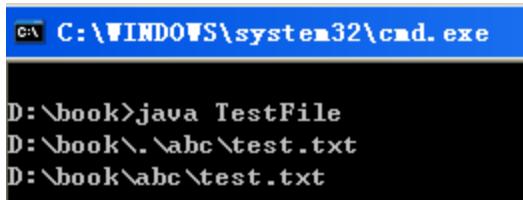
getAbsolutePath()：返回绝对路径。

getCanonicalPath()：返回规格化以后的路径。需要注意的是这个方法会抛出一个异常。

我们同样可以使用一个程序来说明这两个方法。代码如下：

```
import java.io.*;
public class TestFile{
    public static void main(String args[]) throws Exception{
        File f = new File("./abc/test.txt");
        System.out.println(f.getAbsolutePath());
        System.out.println(f.getCanonicalPath());
    }
}
```

上述代码运行结果如下：



```
C:\WINDOWS\system32\cmd.exe
D:\book>java TestFile
D:\book\.\\abc\\test.txt
D:\book\\abc\\test.txt
```

getAbsolutePath 和 getCanonicalPath 都会返回绝对路径，只不过，如果在路径中存在“.”以及“..”这两个符号的话，则 getCanonicalPath 会修正路径，而不让返回值中存在“.”和“..”这两个符号。

下面是一些判断：

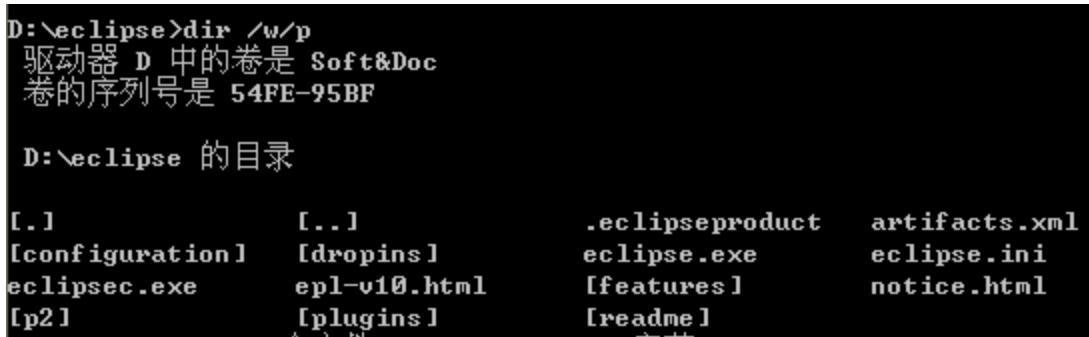
exists()：这个方法用来判断 File 对象表示的文件或者文件夹是否存在。

isFile()：判断 File 对象是否代表的是一个文件

isDirectory()：判断 File 对象是否代表的是一个文件夹

最后，是一个文件夹特有的操作：

listFiles()：这个方法能够返回一个 File 数组，这个数组表示 File 对象所代表文件夹下所有的内容。例如，如果 D:\eclipse 目录下的内容如下图：



```
D:\eclipse>dir /w/p
驱动器 D 中的卷是 Soft&Doc
卷的序列号是 54FE-95BF

D:\eclipse 的目录

[.]           [..]           .eclipseproduct      artifacts.xml
[configuration] [dropins]      eclipse.exe          eclipse.ini
[eclipsec.exe]  [epl-v10.html] [features]          notice.html
[p2]           [plugins]       [readme]
```

我们可以创建一个表示这个目录的 File 对象，并对其调用 listFiles 方法，代码如下：

```
import java.io.*;
public class TestListFiles{
```

```
public static void main(String args[]) {  
    File f = new File("D:/eclipse");  
    File[] fs = f.listFiles();  
    for(int i = 0; i<fs.length; i++) {  
        System.out.println(fs[i].getName());  
    }  
}
```

结果如下：

```
D:\book>javac TestListFiles.java  
D:\book>java TestListFiles  
.eclipseproduct  
artifacts.xml  
configuration  
dropins  
eclipse.exe  
eclipse.ini  
eclipsesec.exe  
epl-v10.html  
features  
notice.html  
p2  
plugins  
readme
```

上面就是 `File` 对象的使用。请在继续下一部分的学习之前，巩固一下 `File` 对象的一些基本操作。

## 1.2 I/O 分类

对于 Java 来说，进行 I/O 操作需要使用“流”对象。所谓的流，指的是：用来传输数据的对象。例如，在生活中，电线就是一种流，这个对象用来传输电力；水管是一种流，这个对象是用来传输水；输油管也是流，用来传输石油，等等。

对于 Java 中的流来说，有三种分类的方式：按照流的方向分，按照流的数据单位分，按照流的功能来分。

首先，流可以按照方向分类，分为输入流和输出流。这里，需要解决的问题是，哪个方向是输入流，哪个方向是输出流。例如，如果从硬盘上读取一个文件，算输入还是输出呢？

所谓输入、输出的方向，总是相对于 JVM 而言的。所谓读取文件，指的是从硬盘中的文件里读取数据，然后这些数据就会传入 JVM 中。这个过程，就是数据从虚拟机的外部“进入”JVM 的过程，这就是“输入”。而写文件，就是把 JVM 中的数据保存到文件中，是数据从 JVM “输出”到文件中，这就是“输出”。

从生活上来说，流也都是有方向的。我们的楼房中，有上水管和下水管，上水管就如同是输入流，而下水管就如同是输出流。

其次，流可以按照数据单位分类，分为字节流和字符流。顾名思义，字节流传输的单位是字节，而字符流传输的单位是字符。

首先，对于任何系统中的所有文件来说，底层都是 0 和 1 组成的，这就是二进制的位(bit)

的概念。而 8 个 bit 组成一个字节，这也就是计算机中处理数据的最小单位。换而言之，由于任何数据都是 bit 组成的，而我们可以每次都传输 8 个 bit 形成一个字节，也就是说，任何数据都可以按照字节的方式进行传输。

因此，字节流可以用来传输任何一种文件类型，包括 mp3、电影、图片、网页、文本……等等。

而系统中，有一种文件比较特殊：文本文件。这种文件大量存在于系统中，例如源代码、html 源码、xml 配置文件，等等。我们在进行 I/O 的时候可能会频繁跟文本文件打交道。字节流同样可以处理文本文件，但是会有一些小问题。例如，大部分中文的文本，一个汉字可能占用的空间不止一个字节。假设一个汉字需要占用两个字节的空间，如果要用字节流处理文本的话，就需要读入两个字节，然后再把这两个字节拼成一个完整的汉字。更有可能在传输错误的时候，产生只保存了“半个汉字”这种问题。为了解决这种问题，我们提供了字符流。

字符流传输数据的单位是字符。这种流专门用于处理文本，能够方便的处理字符编码的问题。关于字符编码的问题，在后面关于字符流的介绍中再进行详细阐述。

最后，流可以按照功能分类，分为节点流和过滤流。什么是节点流呢？这指的是：真正能够完成传输功能的流。而相对的，过滤流并不能完成真正的数据传输，过滤流是用来为其他流增强功能。

如何来理解呢？在输电线中，真正能够传输电力的，是输电线中的金属丝，这就相当于节点流。而输电线往往包一层绝缘的胶布，这层胶布并不能用来传输电力，而是为节点流增强功能（增加了绝缘保护的功能），这一层绝缘胶布就称之为过滤流。

在节点流和过滤流的设计上，I/O 框架中使用了一种设计模式，这种设计模式被称为“装饰模式”。

### 1.3 初识装饰模式

首先，我们简单介绍一下设计模式。所谓的设计模式，我们可以把它理解为设计的“套路”。在面向对象编程的发展过程中，有一些程序员总结出了在设计中有可能经常要遇到的一些问题，为了解决这些问题，在业界有一些套路，这些套路就是所谓的设计模式。在学习设计模式的时候，学习模式的编码和实现固然重要，但是不能忽略这种模式用来解决什么问题。

下面我们就来提出一个问题。

假设我们要设计一种第一人称射击游戏。这种设计游戏，当角色刚刚产生时，手里的枪是一把默认的枪，一把单发的威力较小的枪。

之后，在游戏过程中，有可能遇到一些字母。例如，如果遇到英文字母 S，此时，手里的枪就会变成一把散弹枪，一按扳机会同时发出 5 发子弹；如果遇到英文字母 L，手里的枪则会变成一把激光枪；如果遇到英文字母 F，手里的枪就会变成一把火焰枪。等等

当然，作为第一人称射击游戏，我们的设计和经典游戏魂斗罗还是有区别的。例如，如果先遇到 S 后遇到 L，则此时，我们手中的枪会变成一把散弹激光枪，也就是说，扣动扳机之后会有 5 道激光同时从枪中发出。

此外，除了基本的英文字母之外，还可能遇到一些其他的道具，例如，可能可以捡到瞄准镜，或者消声器等道具。

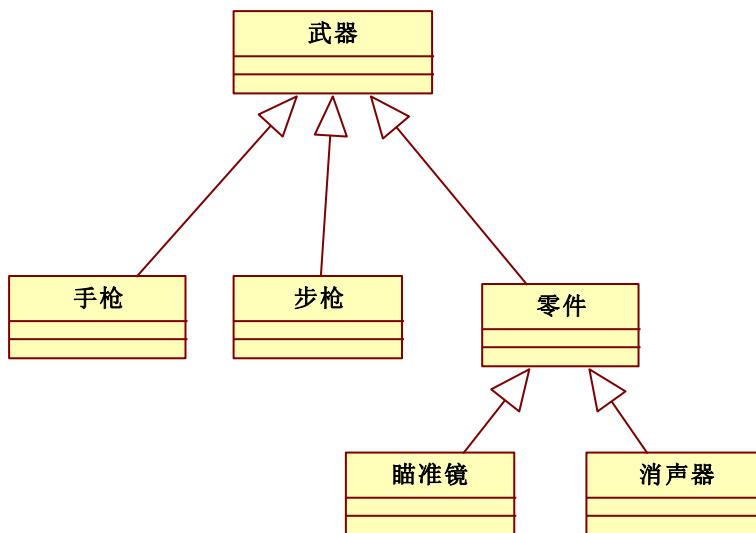
下面我们要为这种游戏来设计枪械类。如果按照一般的设计的话，则对于每一种不同的枪都要设计一个特定的类。

例如：散弹枪、激光枪、火焰枪；散弹激光枪、散弹火焰枪、激光火焰枪；散弹激光火

焰枪。还得加上道具：带瞄准镜的火焰枪，带瞄准镜的激光枪，带瞄准镜的散弹枪（或许，散弹枪不需要瞄准镜？）；带瞄准镜的散弹火焰枪……；带消声器的散弹枪，带消声器的火焰枪，带消声器的……

目前仅仅是两个零件而已。如果随着游戏的更改，又增加了新的道具，比如枪榴弹、刺刀等等，我们就需要写带刺刀的散弹火焰枪、带枪榴弹的散弹激光枪……等等大量的类。每当增加一种新道具，我们所需要写的类都是成几何级数增长的！

怎么解决这个问题呢？为此，我们引入了装饰模式，采用下面的类关系来设计整个武器的体系：



我们可以看到，我们写一个父类武器，是所有武器的父类。此外，这个父类有一些子类，例如手枪、步枪等等。而在子类中，有一个子类是零件，零件的子类就包括标准镜、消声器等等。

从类的设计上来说，除了武器类之外，手枪、步枪等类都是真正完成射击功能的类，也就是真正完成功能的类。在 I/O 框架中对应的概念就是节点流。而零件的那些子类，是为枪械增强功能的，本身并不能完成射击的功能，因此对应于 I/O 的概念，就是过滤流。

## 2 字节流

介绍完流的分类之后，我们首先来介绍一下字节流。字节流的特点是传输的数据单位是字节，也意味着字节流能够处理任何一种文件。

### 2.1 InputStream/OutputStream 的基本操作

首先，介绍一下所有字节流的父类，也就是在装饰模式中扮演“武器”这个角色的类。所有输入字节流的父类是 `InputStream`，所有输出字节流的父类是 `OutputStream`，他们都处于 `java.io` 包下。要学习这两个类，就需要创建这两个类的对象。但是，查阅 JDK 文档，我们可以知道这两个类都是抽象类，无法创建对象。因此，为了学习这两个类的用法，我们需要先获得这两个类的某个子类。

我们将使用 `FileInputStream` 和 `FileOutputStream` 这两个类来学习 `InputStream` 和

`OutputStream`。`FileInputStream` 是文件输入流，从功能上说，这是一个节点流，能够读取硬盘上的文件；而 `FileOutputStream` 是文件输出流，能够写入文件。

下面，我们就以 `FileInputStream / FileOutputStream` 为例，来学习一下字节流。

### 2.1.1 FileInputStream

首先我们研究一下 `FileInputStream` 这个类。这个类由若干个构造方法，其中两个比较常用，罗列如下：

`FileInputStream(String filename)` : 通过文件路径，获得文件输入流

`FileInputStream(File file)` : 通过文件对象，获得文件输入流。

需要注意的是，`FileInputStream` 在创建对象的时候，当要读取的文件不存在时，就会抛出异常：`FileNotFoundException`。由于这是一个已检查异常，因此必须要处理。

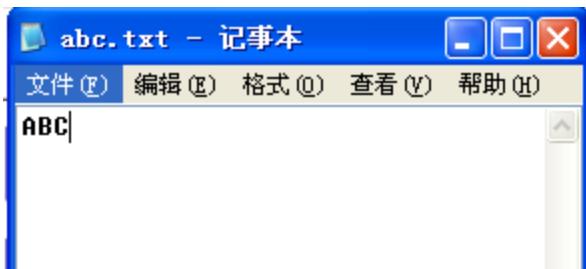
除此之外，`FileInputStream` 还有一些值得注意的方法：

`close()`：这个方法顾名思义，可以用来关闭文件流，释放资源。当我们结束输入操作时，应该调用该方法来关闭流。

`int read()`：无参的 `read` 方法。这个方法每次都从文件中读取一个字节，并且把读到的内容返回。要强调的是，虽然返回值是一个 `int` 类型，但是每次读文件时只读取一个字节，这个字节作为 `int` 四个字节中的最低位返回。

而当读到流末尾时，返回-1。

我们可以结合代码来理解 `read` 方法。假设我们在当前目录下准备一个文件 `abc.txt`，这个文件中保存了三个英文字母：ABC。如下所示：



保存这个文件，本质上是保存了三个字节，这三个字节分别代表了三个英文字母 A、B、C 的底层编码。根据 ASCII 编码的规范，这三个字母的编码分别为 65、66、67。

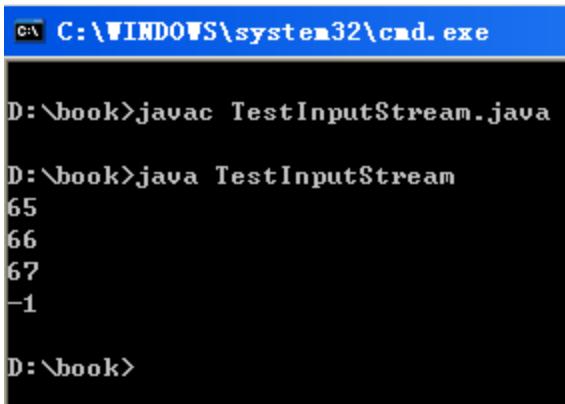
然后有下面的代码：

```
import java.io.*;
public class TestInputStream{
    public static void main(String args[]) throws Exception{
        FileInputStream fin = new FileInputStream("abc.txt");
        System.out.println(fin.read());
        System.out.println(fin.read());
        System.out.println(fin.read());
        System.out.println(fin.read());
        fin.close();
    }
}
```

我们总共调用了四次 `read` 方法。

当我们调用第一次 `read` 方法时，会从文件中读取一个字节，因此会输出 65；第二次时，

会读取下一个字节，输出 66，第三次输出 67；当第四次调用 read 方法时，由于此时已经读到了流的末尾，因此此时 read 方法返回值为 -1。运行结果如下：



```
c:\ C:\WINDOWS\system32\cmd.exe
D:\book>javac TestInputStream.java
D:\book>java TestInputStream
65
66
67
-1
D:\book>
```

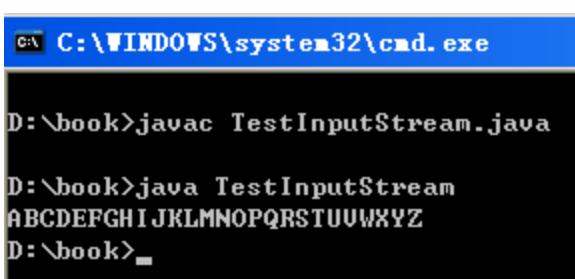
下面，我们把 abc.txt 增加内容，让这个文件中保存 26 个英文字母，如下：



如果我们要读取这个文件中的所有内容，此时再一次一次调用 read 方法显然不现实，我们应当用 while 循环来完成。同时，由于 read 方法返回值为 int 类型，为了能够输出正常的字母，我们需要把 read 方法的返回值强制转换为 char 类型。典型代码如下：

```
import java.io.*;
public class TestInputStream{
    public static void main(String args[]) throws Exception{
        FileInputStream fin = new FileInputStream("abc.txt");
        int ch = 0;
        while( (ch=fin.read()) != -1){
            System.out.print((char)ch);
        }
        fin.close();
    }
}
```

这个代码利用 while 循环读取文件内容，并且把文件中所有内容都输出。运行结果如下：



```
c:\ C:\WINDOWS\system32\cmd.exe
D:\book>javac TestInputStream.java
D:\book>java TestInputStream
ABCDEFGHIJKLMNOPQRSTUVWXYZ
D:\book>
```

介绍完无参的 read 方法之后，下面我们来介绍有参的 read 方法。首先是带 byte 数组参数的 read 方法：

`int read(byte[] bs)`：这个方法的特点是，每次调用这个方法的时候，会把读取到的数据放入 `bs` 数组中，一次调用尽量读取 `bs.length` 个字节。为什么说尽量读取 `bs.length` 个字节呢？假设 `bs.length` 长度为 6，如果流中剩余的字节数超过 6 个，那么这时候，调用这个 `read` 方法会把 `bs` 数组读满；而如果流中剩余的字节数不到 6 个的时候，那么调用 `read` 方法会把流中剩下所有的数据都读入到数组中。由于剩余的数据少于 6 个字节，因此数组不会被读满。

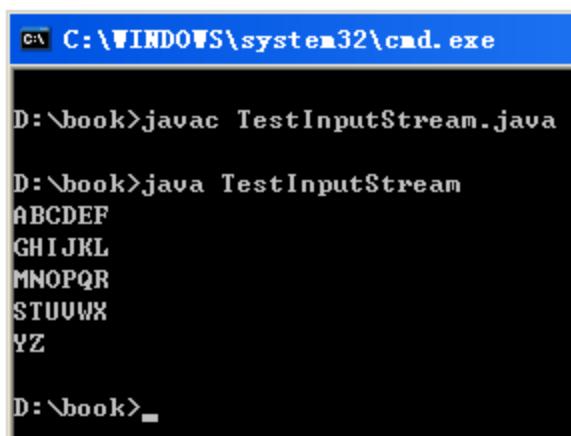
我们可以这么来理解：`read` 方法就好像是去食堂打饭一样。无参的 `read` 方法，相当于每次去食堂都只打一粒饭。而对于 `read(byte[])` 来说，就相当于每次去食堂打饭时都带着一个饭盒。比如这个饭盒能打 6 两饭，每次调用 `read` 方法，都尽量能够把这个饭盒打满。如果食堂里面剩下的饭超过 6 两，那么要去打满一盒饭没有问题；而如果去食堂的时候已经比较晚了，食堂里只剩下 3 两饭了，这个时候，那只能打三两饭，即使饭盒不满也没办法。

理解了上面的问题，返回值也就好理解了：`int read(byte[] bs)` 这个方法的返回值，返回的是读到的字节数。对于大多数情况来说，读到的字节数和 `bs.length` 相同；而对于流中剩下的字节数少于 `bs.length` 的情况，`read` 方法返回的就是实际读到的字节数。

特别要注意的是，当读到流末尾时，返回 -1。

我们可以利用这个方法，改写一下之前的 `TestInputStream` 程序。修改过的代码和运行结果如下：

```
import java.io.*;
public class TestInputStream{
    public static void main(String args[]) throws Exception{
        FileInputStream fin = new FileInputStream("abc.txt");
        byte[] bs = new byte[6];
        int len = 0;
        while( (len=fin.read(bs)) != -1) {
            for(int i = 0; i < len; i++) {
                System.out.print((char)bs[i]);
            }
            System.out.println();
        }
        fin.close();
    }
}
```



```
C:\> C:\WINDOWS\system32\cmd.exe
D:\book>javac TestInputStream.java
D:\book>java TestInputStream
ABCDEF
GHIJKL
MNOPQR
STUUVW
YZ

D:\book>
```

需要注意的是，上面的代码我们使用了一个 `len` 变量用来保存每次 `read` 方法的返回值。

读者可以思考一下为什么。

最后，简单介绍一下带多个参数的 `read` 方法：

`int read(byte[] bs, int off, int len)`：这个方法同样是把数据读入数组中，只不过这个方法读取数据的时候，会让数据在数组中以下标为 `off` 的地方开始，并且最多只读取 `len` 个。也就是说，这个方法并没有使用整个 `bs` 数组来存放读到的数据，而是使用了数组的一部分。究竟是哪部分，是由 `off` 和 `len` 这两个参数决定的。这就好比我们去食堂打饭的时候，使用的是分过格子的饭盒，我们要打的也不是一盒饭，而是一格饭。

返回值是真正读取的字节数，当读到流末尾时，返回-1。

### 2.1.2 FileOutputStream

相对于 `InputStream`，`OutputStream` 更简单一些。首先同样是研究一下 `OutputStream` 的构造方法，罗列如下：

`OutputStream(String path)`：根据路径创建文件输出流

`OutputStream(File file)`：根据文件对象创建文件输出流

还有其他的构造方法，会在后面的内容中再次提到。

此外，下面是一些基本操作：

`close()`：关闭流

`OutputStream` 最重要的是 `write` 方法，其 `write` 方法罗列如下：

`void write(int v)`：这个 `write` 方法每次调用时写入一个字节。注意，虽然这个 `write` 方法接受的参数类型是 `int` 类型，虽然 `int` 类型有 4 个字节，但是这个 `write` 方法每次只写入 `int` 类型中最后的那个字节。

`void write(byte[] bs)`：写入一个 `byte` 数组。

`void write(byte[] bs, int off, int len)`：同样是写入一个 `byte` 数组，所不同的是，写入这个 `byte` 数组的时候，数据内容从数组下标 `off` 的位置开始，写入 `len` 个字节。

下面我们演示一下如何使用 `write` 方法。假设我们要写入一个字符串：`Hello World`，由于 `OutputStream` 中没有一个类能够接受一个字符串作为参数，因此我们必须要把字符串转化为 `byte` 数组。转换的方式也很简单，`String` 类中有一个方法叫做 `getBytes()`，签名如下：

`public byte[] getBytes()`

这个方法没有参数，并且返回一个 `byte` 数组，这就是字符串转为 `byte` 数组的方法。

我们利用这个方法，编写代码如下：

```
import java.io.*;
public class TestOutputStream{
    public static void main(String args[]) throws Exception{
        String hello = "Hello World";
        byte[] bs = hello.getBytes();
        FileOutputStream fout= new FileOutputStream("test.txt");
        fout.write(bs);
        fout.close();
    }
}
```

运行这个代码之前，当前目录下并不存在 `test.txt` 文件。

运行之后，在当前目录下产生 test.txt 文件，其中的内容为：Hello World。这也就提示我们：如果文件不存在，`FileOutputStream` 会创建新文件。

那么文件已存在会怎么样呢？修改 `test.txt` 文件的内容并保存，然后再次运行程序，你会发现 `test.txt` 文件中的内容依然是 Hello World。

这说明：默认情况下，当文件已存在时，`FileOutputStream` 会覆盖同名文件。

我们也可以使用 `FileOutputStream` 另外两个构造函数：

`FileOutputStream(String path, boolean append)`

`FileOutputStream(File file, boolean append)`

这两个构造方法与原来的构造方法相比，增加了一个 `boolean` 参数：`append`。这个参数表示什么含义呢？

当这个参数为 `false` 的时候，这两个构造函数与只有一个参数的构造函数用起来相同：如果文件不存在，则 `FileOutputStream` 会创建新文件；如果文件已存在，则覆盖原文件。

如果这个参数为 `true`，则：当如果文件不存在，依然会创建新文件；而如果文件已存在，则会用追加的方式写文件。

我们可以写一个实验代码：

```
import java.io.*;
public class TestOutputStream{
    public static void main(String args[]) throws Exception{
        String hello = "Hello World";
        byte[] bs = hello.getBytes();
        FileOutputStream fout
            = new FileOutputStream("test.txt", true);
        fout.write(bs);
        fout.close();
    }
}
```

把上面的例子进行修改，在创建 `FileOutputStream` 的时候增加一个新的参数 `true`。如果 `test.txt` 文件不存在，第一次运行时，会创建这个文件，文件内容为：Hello World；反复运行这个程序多次，就会出现多个 Hello World。这就是用追加的方式写文件。

### 2.1.3 处理异常

之前，我们的程序处理异常的时候，都是使用 `throws` 的方式处理。下面，我们对 `TestInputStream` 程序进行修改，使用相对比较积极的 `try-catch` 的方式进行处理。

首先，在 `FileInputStream` 这个类中，有下面这些部分是需要进行异常处理的：

- 1、创建对象时
- 2、调用 `read` 方法时
- 3、调用 `close` 方法时

另外，要注意的是，`close` 方法属于释放资源的代码，应当写在 `finally` 代码块中。

修改之后的 `TestInputStream` 代码如下：

```
import java.io.*;
public class TestInputStream{
    public static void main(String args[]) {
        FileInputStream fin = null;
        try{
```

```

        fin = new FileInputStream("abc.txt");
        byte[] bs = new byte[6];
        int len = 0;
        while( (len=fin.read(bs)) != -1) {
            for(int i = 0; i < len; i++) {
                System.out.print((char)bs[i]);
            }
            System.out.println();
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally{
        if(fin!=null)
            try{
                fin.close();
            }catch(IOException e){
                e.printStackTrace();
            }
    }
}

```

有这样几个要点需要注意：

- 1、在 try 块之外定义 FileInputStream 变量 fin。原因在于：如果在 try 块内定义这个变量的话，则这个变量的作用范围就是 try 块内部，这样这个变量就无法在 finally 中使用。
- 2、给 fin 赋初值 null。原因在于，在 try 块内的代码被 Java 认为有可能不执行，如果不给 fin 赋初值的话，在 finally 中对 fin 的使用有可能没有赋初值，违反了对局部变量先赋值后使用的原则。
- 3、在 finally 中，调用 close 方法之前，需要判断一下 fin 变量是否为 null。

请读者根据上面的例子，修改 TestOutputStream 程序，使用 try-catch-finally 处理异常。

## 2.2 过滤流基础

介绍完节点流之后，下面我们开始介绍字节流的过滤流。首先介绍一下过滤流的基本使用。在学习过滤流的时候，关键的关键在于，一定要搞清楚过滤流增强了什么功能，以及在什么情况之下要使用过滤流的这种功能。

### 2.2.1 Data Stream

首先我们来介绍一对过滤流：DataInputStream 和 DataOutputStream。这两个类有什么作用呢？首先来思考下面的需求：

假设，要把一个 double 类型的数据写入文件中（例如 3.14），应当怎么做呢？由于 FileOutputStream 没有一个接受 double 类型作为参数的 write 方法，因此必须要想别的方法。

一个 double 变量占据 8 个字节，因此写一个 double 类型的时候，比较直观的方法应当是把这个 double 类型的数据拆分成 8 个字节，然后把这 8 个字节写入到流中去。当然，把 double

类型进行拆分是一件比较麻烦的事情。

幸运的是，拆分 double 这件事情不需要程序员自己去做，而可以使用 Java 中现成的类。Sun 公司提供了 DataInputStream 以及 DataOutputStream 这两个类，这两个类为流增强的功能就是：增强了读写八种基本类型和字符串的功能。

我们可以看一下 DataOutputStream 的方法：除了有 OutputStream 中有的几个 write 方法之外，还有 writeBoolean, writeByte, writeShort ... 等一系列方法，这些方法接受某一种基本类型，把基本类型写入到流中。

需要注意的是，有一个 writeInt(int n)方法，这个方法接受一个 int 类型的参数。这个方法和 write(int v)方法不同。writeInt 方法是 DataOutputStream 特有的方法，这个方法一次写入参数 n 的四个字节。而 write 方法则一次写入参数 v 的最后一个字节。

与之对应的，DataInputStream 的方法中，除了有几个 read 方法之外，还有 readBoolean, readByte, readInt 等一系列方法，这些方法能够读入若干个字节，然后拼成所需要的数据。例如 readDouble 方法，就会一次读入 8 个字节，然后把这 8 个字节拼接成一个 double 类型。

最后要提示的是，DataXXXStream 中有 readUTF 和 writeUTF 这两个方法用来读写字符串，但是一般来说，我们读写字符串的时候几乎不使用 Data 流。Data 流主要是用在 8 种基本类型的读写上。

下面，我们来写一个程序，在一个文件中存入 3.14 这个 double 值，然后再从文件中把这个值读取出来。在写代码之前，我们先来研究一下过滤流的使用。

### 2.2.1.1 过滤流使用的基本步骤

首先，过滤流的构造方法中，一般都会有一个构造方法，接受其他类型的流。例如，在 DataInputStream 的构造方法中，唯一的构造方法如下：

DataInputStream(InputStream is)

而 DataOutputStream 类唯一的构造方法如下：

DataOutputStream(OutputStream os)

这个参数表示什么呢？可以这么来理解：过滤流使用来为其他流增强功能的，而构造方法中的这个参数，表明的是过滤流为哪一个流增强功能。

过滤流的使用分为下面四个步骤：

1、创建节点流。这个步骤是使用过滤流的先决条件，由于过滤流无法直接实现数据传输功能，因此必须先有一个节点流，才能够进行数据传输。

2、封装过滤流。所谓的“封装”，指的是创建过滤流的时候，必须以其他的流作为构造函数的参数。需要注意的是，可以为一个节点流封装多个过滤流（虽然这样的情况并不是很常见）

3、读/写数据

4、关闭外层流。这指的是，关闭流的时候，只需要关闭最外层的过滤流即可，内层流会随着外层流的关闭而一起被关闭。

我们结合上面对过滤流的使用，给出下面的代码：

```
import java.io.*;
public class TestDataStream{
    public static void main(String args[]) throws Exception{
        //创建节点流
        FileOutputStream fout = new FileOutputStream("pi.dat");
        //封装过滤流
        DataOutputStream dout = new DataOutputStream(fout);
        //写数据
        dout.writeDouble(3.14);
        //关闭外层流
        dout.close();

        //创建节点流
        FileInputStream fin= new FileInputStream ("pi.dat");
        //封装过滤流
        DataInputStream din = new DataInputStream(fin);
        //读数据
        double pi = din.readDouble();
        //关闭外层流
        din.close();

        System.out.println(pi);
    }
}
```

有兴趣的读者可以尝试着处理一下上述代码的异常。

需要注意的是，由于 Data 流保存八种基本类型的方式采用的是拆分子节的方式，而不是采用文本的方式，因此保存的 pi.dat 这个文件无法用文本编辑器直接进行编辑。

## 2.2.2 Buffered Stream

下面我们要介绍的是一对流：BufferedInputStream 和 BufferedOutputStream。这两个流增强了缓冲区的功能。

什么叫缓冲区呢？在之前的代码中，我们每调用一次 read 或者 write 方法，都会触发一次 I/O 操作。而由于 I/O 操作要跨越 JVM 的边界，因此进行 I/O 操作的时候，事实上效率会非常低，这非常不利于程序的高效。为了让程序的效率得到提升，我们引入了缓冲机制。

我们会在内存中开辟一块空间，当调用 read 或者 write 方法时，并不真正进行 I/O 操作，而是对内存中的这块空间进行操作。我们以 write 操作为例，使用了缓冲机制之后，我们调用 write 方法时，并不真正把数据写入到文件中，而是先把数据放到缓冲区里。等到缓冲区满了之后，再一次性把数据写入文件中。

为什么这样就能提高效率了呢？考虑下面这个生活中的例子。在大学里面，我们都用手

洗衣服。手洗衣物有一个很大的问题，往往衣服洗完以后很难拧干，晾在阳台上之后会滴水。如果任由衣物在阳台上滴水的话，有可能会让阳台变得很脏，如果阳台上放了一些其他的杂物的话，更有可能因为滴水而损坏那些物品。因此，我们需要使用一个方式，把衣服上滴的水运输到洗手间，倒掉。

那怎么运输呢？一般来说，我们不会在阳台傻等，等一滴水下来以后马上就用手接着然后跑到洗手间倒掉。我们往往会用一个盆先接水，当这个盆满了以后，我们才会真正把这个盆中的水倒掉，也就是真正完成 I/O 操作。这个盆，就好比是我们的缓冲区。通过这个盆，我们减少了 I/O 的次数，从而提高了 I/O 的效率。

Buffered 流几乎没有为流增加新的方法，我们给出一个输出流的例子代码：

```
import java.io.*;
public class TestBufferedStream{
    public static void main(String args[]) throws Exception{
        String data = "Hello World";
        byte[] bs = data.getBytes();
        //创建节点流
        FileOutputStream fout
            = new FileOutputStream("test.txt");
        //封装过滤流
        BufferedOutputStream bout
            = new BufferedOutputStream(fout);
        //写数据
        bout.write(bs);
        //关闭外层流
        bout.close();
    }
}
```

需要注意的是，如果把 `bout.close()` 方法去掉，此时在看 `test.txt` 文件，会发现文件的内容为空。这是因为，我们在调用 `write` 方法的时候，其实并没有真正把数据写入到文件中，而只是把数据写入到缓冲区中。那么什么时候缓冲区中的数据会真正写入到文件中呢？有三种情况：第一种情况是缓冲区已满，第二种情况是调用 `close` 方法。

除了这两种情况之外，假设程序员希望在缓冲区没有满并且不关闭流的情况下，把缓冲区内的东西真正写入流中，应当调用一个方法：`flush()`。这个方法用来清空缓冲区，往往用在输出流上面。当一个带缓冲的输出流调用 `flush()` 之后，就能保证之前在缓冲区中的内容真正进行了 I/O 操作，而不是仅仅停留在缓冲区。

### 2.2.3 PrintStream

`PrintStream` 是一个比较特殊的过滤流，我们简单介绍一下，读者作为一般性的了解即可。

`PrintStream` 作为过滤流，增强的功能有以下几个：

- 1、缓冲区的功能
- 2、写八种基本类型和字符串

### 3、写对象。

需要注意的是，这个流写基本类型和写对象的时候，是按照字符串的方式写的。也就是说，这个流写八种基本类型的时候，会把基本类型转换成字符串以后再写，而写对象的时候，会写入对象的 `toString()` 方法返回值。

这个类具体如何使用我们不多介绍了。需要介绍的是这个类的一个对象：我们所熟知的向屏幕输出数据的对象：`System.out` 对象，这就是一个 `PrintStream` 类型的对象。

## 2.3 对象序列化

### 2.3.1 序列化的概念

下面我们为大家介绍另外一对过滤流：`ObjectInputStream` 和 `ObjectOutputStream`。这两个也是过滤流，增强的功能如下：

- 1、增强了缓冲区功能
- 2、增强了读写八种基本类型和字符串的功能。读写基本类型和字符串的方式，与 Data 流完全一样。
- 3、增强了读写对象的功能。这是这两个流最主要的作用。在 `ObjectInputStream` 类有一个 `readObject` 方法，这个方法能够往流中写入一个对象；而 `ObjectOutputStream` 类中有一个 `writeObject` 方法，这个方法能够从流中读取一个对象。

如上所述，`ObjectInputStream` 和 `ObjectOutputStream` 能够完成对对象的读写。这种把对象放到流上进行传输的过程，称之为“对象序列化”。一个对象如果能够放到流上进行传输，则我们称这个对象是“可序列化”的。

### 2.3.2 `Serializable` 接口和 `transient` 关键字

需要注意的是，并不是所有对象都是“可序列化”的。举个例子说，搬家就是一个传输对象的过程。然而，搬家的时候并不是所有对象都能够搬走的。例如，家具、电器，这些对象往往在搬家的时候是能够搬走的，但是，门、窗户、地板，这些对象无法搬走。那我们可以说家具、电器是可序列化的对象，而窗户、地板是不可序列化的对象。

那怎么让对象能够在流上进行传输呢？如果要让一个类成为可序列化的，只要让这个类实现一个接口：`java.io.Serializable` 接口即可。

要实现这个接口，就要实现这个接口中的所有方法。好了，现在请去查一下 `Serializable` 接口，看看这个接口中定义了哪些方法？

让你惊讶吧，这个接口中没有任何的方法。也就是说，如果要实现这个 `Serializable` 接口，只需要写上 `implements Serializable` 就可以了。

我们可以尝试一下。定义一个 `Student` 类，创建两个对象并保存到文件中，然后再利用另一个流读取文件。代码如下：

```
import java.io.*;
class Student implements Serializable{
    String name;
    int age;
    public Student(String name, int age) {
```

```

        this.name = name;
        this.age = age;
    }

}

public class TestSerializable {
    public static void main(String[] args) throws Exception {
        Student stu1 = new Student("tom", 18);
        Student stu2 = new Student("jerry", 18);

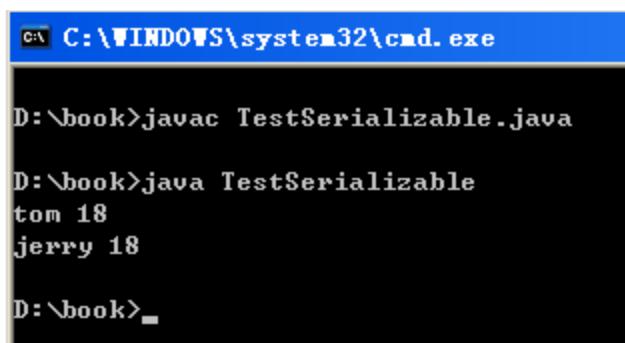
        FileOutputStream fout = new FileOutputStream("stu.dat");
        ObjectOutputStream oout = new ObjectOutputStream(fout);
        oout.writeObject(stu1);
        oout.writeObject(stu2);
        oout.close();

        FileInputStream fin = new FileInputStream("stu.dat");
        ObjectInputStream oin = new ObjectInputStream(fin);
        Student s1 = (Student) oin.readObject();
        Student s2 = (Student) oin.readObject();
        oin.close();
        System.out.println(s1.name + " " + s1.age);
        System.out.println(s2.name + " " + s2.age);

    }
}

```

需要注意的是，由于 `readObject` 方法返回值为 `Object` 类型，因此需要对返回值进行强转。  
运行结果如下：



```

C:\> C:\WINDOWS\system32\cmd.exe
D:\book>javac TestSerializable.java
D:\book>java TestSerializable
tom 18
jerry 18
D:\book>

```

同时，也产生了一个 `stu.dat` 文件。这个文件中保存的都是一些二进制数据，这些二进制数据就是保存对象时保存的数据。

下面，我们介绍一下一个新的关键字：`transient`。这个关键字是一个修饰符，这个修饰符可以用来修饰属性，用 `transient` 修饰的属性表示：这个属性不参与序列化。

我们可以修改原有的代码，把 `Student` 类的 `age` 属性修改为 `transient` 的。这样之后产生

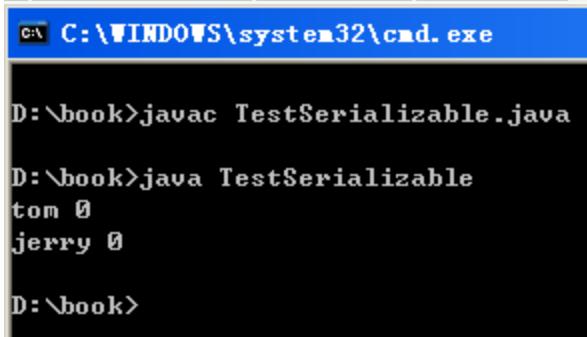
的改变是：1、stu.dat 文件的大小变小了。这很容易理解，因为由于参与序列化的属性变少了，因此序列化之后保存的数据也变少了，从而导致文件也变小了。2、输出的 age 属性都为 0。这是因为由于在 writeObject 的时候没有保存 age 属性，而读取时从文件中也读取不到 age 属性，从而导致这个属性的值只能是默认值：0。

修改后的代码以及运行结果如下：

(其他代码与上一个例子相同)

```
class Student implements Serializable{  
    String name;  
    transient int age;  
    public Student(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

运行结果：



```
C:\WINDOWS\system32\cmd.exe  
D:\book>javac TestSerializable.java  
D:\book>java TestSerializable  
tom 0  
jerry 0  
D:\book>
```

另外，在使用对象序列化的时候，注意这样两个问题：

1、不要使用追加的方式写对象。也就是说，如果我们创建一个文件输出流采用 FileOutputStream(file, true) 的方式创建节点流，然后再在外面封装 ObjectOutputStream，这样将无法完成我们设想的结果。如果对一个文件多次写入的话，读取对象的时候只能读取第一次写入的对象，而后面用追加的方式写入的对象将无法被读取。这是对象序列化底层机制所决定的。

2、如果一个对象的属性又是一个对象，则要求这个属性对象也实现了 Serializable 接口，如果一个对象的属性是一个集合，则要求集合中所有对象都实现 Serializable 接口。除非这个对象的属性被标记为 transient，不参与序列化。

这就是最基本的对象序列化的内容。关于对象序列化的更多内容，可以参考 Sun 公司的网站以及相关文档。

### 3 字符流

研究完字节流之后，我们开始介绍字符流。要理解字符流，首先要理解字符编码的含义。

### 3.1 字符编码

计算机中显示文字的时候，本质上是在屏幕上绘制一些图像用来显示文字。从这个意义上说，文字就是一种特殊的图片。

然而，在计算机中保存文字的时候，并不是按照图片的方式保存。当保存文件的时候，计算机底层会把文字转换成数字，然后再进行保存。计算机把字符转换为数字的过程，称之为“编码”。而读取文件的时候，则过程相反，计算机把数字转化为文字，并绘制到屏幕上。计算机把数字转换为字符的过程，称之为“解码”。

很显然，不同的字符必须对应不同的数字，不然，在解码时会遇到问题。

那么，什么字符对应于什么数字呢？有些标准化组织，会规定字符和数字之间的对应关系，这种对应关系就是所谓的编码规范。常见的编码规范如下：

ASCII：最早的编码方式，规定了英文字母和英文标点对应的编码

ISO-8859-1：这种编码方式包括了所有的西欧字符以及西欧标点。

GB2312/ GBK：大陆广泛使用的简体中文编码。其中，GB2312 是 GBK 的一个子集，也就是说，在 GB2312 中有的汉字，在 GBK 中也有，且同一个汉字在 GB2312 和 GBK 中的编码相同。GBK 主要在 GB2312 的基础上扩展了很多新的字符。

GBK 是简体中文 Windows 的默认编码方式。

Big5：台湾地区广泛使用的繁体中文编码。

UTF-8：一种国际通用编码，包括简体和繁体中文。与 GB2312/GBK 不兼容，也就是说，同一个汉字，在 GBK 和 UTF-8 的编码是不同的。大部分简体中文 Linux 使用的是 UTF-8 编码。

由于有了多种编码规范，因此就会有乱码的问题。乱码问题是怎么产生的呢。例如，我们在保存文件的时候，使用了 GBK 编码，这个时候，假设一个字符“程”，被编码成了数字 31243，于是这个文件在底层保存的数据就是 31243。之后，这个文件被传送到了台湾，台湾地区的工程师打开文件的时候，使用的是软件对这个文件采用 Big5 解码。此时，就会把 31243 这个数字给解码成字符“最”。这样，就会产生理解上的误会，从而产生乱码。

换而言之，产生乱码的根源在于：编解码方式不一致。

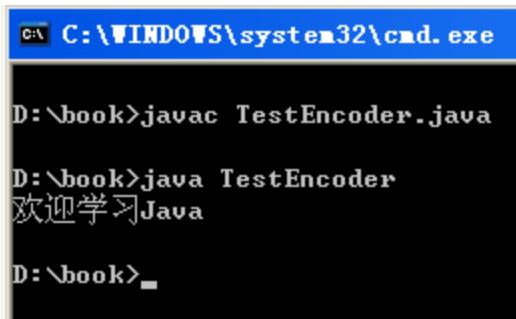
我们可以用程序来演示编解码。之前我们说过，String 类有一个方法 getBytes()，这个方法能够把字符串转换成一个 byte 类型的数组，实际上就是把字符转化为数字的过程，本质上，就是在进行编码。在调用 getBytes 的时候，也可以指定编码方式。

那得到 byte 数组之后，如何解码呢？String 类有一个构造方法，能够接受 byte 数组作为参数，这就是能够把数字转化为字符串，本质上是解码的过程。在构造的时候，也可以指定解码的方式。

示例代码如下：

```
public class TestEncoder {  
    public static void main(String[] args) throws Exception {  
        String str = "欢迎学习 Java";  
        //编码，指定编码方式为 GBK  
        byte[] bs = str.getBytes("GBK");  
        //解码，指定解码方式为 GBK  
        String str2 = new String(bs, "GBK");  
        System.out.println(str2);  
    }  
}
```

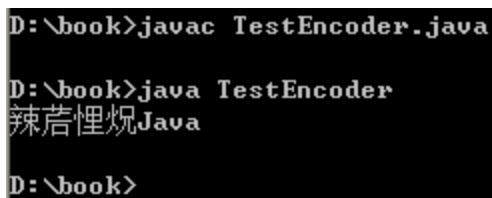
上面的程序中，编码和解码的方式都为 GBK，输出结果为：



```
C:\WINDOWS\system32\cmd.exe
D:\book>javac TestEncoder.java
D:\book>java TestEncoder
欢迎学习Java
D:\book>
```

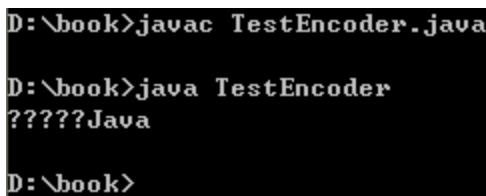
可以看到，没有乱码。

如果修改一下，编码用 GBK，解码用 Big5，则输出结果如下：



```
D:\book>javac TestEncoder.java
D:\book>java TestEncoder
辣啓惺忼Java
D:\book>
```

可以看到，产生了乱码。同样的，如果编码用 GBK，而解码用 UTF-8，则输出结果如下：



```
D:\book>javac TestEncoder.java
D:\book>java TestEncoder
?????Java
D:\book>
```

同样产生了乱码。

需要注意的是，英文字母（例如上面字符串中的“Java”），无论采用什么方式编码和解码，都不会产生乱码。世界上任何一种编码方式，都与 ASCII 编码兼容，也就是说，任何一种编码方式下面，A 都对应 65，a 都对应 97，没有例外。

### 3.2 获得字符流与桥转换

由于编码方式的不一致，导致了传输文本的时候会有一些比较棘手的问题。为了让传输文本文件更加方便，我们使用字符流。

首先是字符流的父类。所有输入字符流的父类是 Reader，所有输出字符流的父类是 Writer。与 InputStream 和 OutputStream 类似，这两个类也是抽象类。

此外，与 FileInputStream 以及 FileOutputStream 类似，有两个类 FileReader 和 FileWriter，这两个类分别表示文件输入字符流和文件输出字符流。这两个流的使用与 FileInputStream 以及 FileOutputStream 也非常雷同，在此不多介绍，需要注意的是，使用这两个流的时候，无法指定编解码方式。

通过 FileReader 和 FileWriter 可以直接获得文件字符流。

下面，我们介绍两个流：InputStreamReader 和 OutputStreamWriter。

InputStreamReader 这个类本身是 Reader 类的子类，因此这个类的对象是一个字符流。而这个流的构造函数如下：

## Constructor Summary

[InputStreamReader\(InputStream in\)](#)

Creates an InputStreamReader that uses the default charset.

[InputStreamReader\(InputStream in, Charset cs\)](#)

Creates an InputStreamReader that uses the given charset.

[InputStreamReader\(InputStream in, CharsetDecoder dec\)](#)

Creates an InputStreamReader that uses the given charset decoder.

[InputStreamReader\(InputStream in, String charsetName\)](#)

Creates an InputStreamReader that uses the named charset.

可以看到，这个流所有构造函数，都可以接受一个 InputStream 类型的参数。也就是说，通过这个流，可以接受一个字节流作为参数，创建一个字符流。这个对象就起到了字节流向字符流转换的功能，我们往往称之为：桥转换。

类似的， OutputStreamWriter 类能够把一个输出字节流转换为一个输出字符流。

在桥转换的过程中，我们还可以指定编解码方式。如果不指定的话，则编码方式采用系统默认的编码方式。

因此，通过桥转换获得字符流，也是一个获得字符流的方式。这种方式有两个用法：

- 1、如果需要制定编码方式，则应当使用桥转换。
- 2、在无法直接获得字符流的情况下，可以先获得字节流，再通过桥转换获得字符流。

利用桥转换进行编程，需要以下五个步骤：

- 1、创建节点流
- 2、桥转换为字符流
- 3、在字符流的基础上封装过滤流
- 4、读/写数据
- 5、关闭外层流

### 3.3 字符过滤流

介绍完如何获得字符流之后，下面单刀直入，介绍一些字符流的过滤流。对于字符流来说，常用的过滤流只有两个：读入使用 BufferedReader，写出使用 PrintWriter。下面我们分别进行介绍。

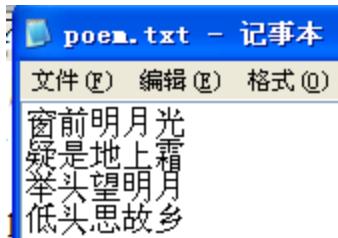
#### 3.3.1 BufferedReader

顾名思义， BufferedReader 提供了缓冲区功能。但是更重要的是， BufferedReader 中有一个 readLine()方法，签名如下：

```
public String readLine()
```

这个方法也很容易理解：每次读入一行文本，并把读入的这一行文本当做返回值返回。当读到流末尾时，返回一个 null 值。

下面给出一个示例代码，介绍 BufferedReader 的使用。首先在当前目录下准备一个文本文件，文本文件中保存一段文字。对于 Windows 来说，默认的编码为 GBK。如下：



然后有如下代码：

```
import java.io.*;
public class TestPoem {
    public static void main(String[] args) throws Exception {
        //创建节点流
        FileInputStream fin = new FileInputStream("poem.txt");
        //桥转换
        Reader r = new InputStreamReader(fin, "GBK");
        //封装过滤流
        BufferedReader br = new BufferedReader(r);
        //读/写数据
        String line = null;
        while( (line=br.readLine()) !=null) {
            System.out.println(line);
        }
        //关闭外层流
        br.close();
    }
}
```

上面的代码演示了如何使用 `BufferedReader` 读取文件。

### 3.3.2 PrintWriter

`PrintWriter` 是一个很特殊的类。

首先, `PrintWriter` 可以作为一个过滤流。这个流可以接受一个 `Writer` 作为参数。增强了如下一些功能:

- 1、缓冲区的功能。因此使用 `PrintWriter` 应当及时关闭或刷新
- 2、写八种基本类型和字符串的功能。
- 3、写对象的功能。

在 `PrintWriter` 类中, 有一系列 `print` 方法, 这些方法能够接受八种基本类型、字符串和对象。同样的, 还有一系列 `println` 方法, 这些方法在写入数据之后, 会在数据后面写入一个换行符。

要注意的是, `PrintWriter` 写基本类型的方式, 是把基本类型转换为字符串再写入流中, 与 `Data` 流不同。举例来说, 对于 3.14 这个 `double` 类型的数, `Data` 流会把这个数拆分成 8 个字节写入文件, 而 `PrintWriter` 会把这个数字转化为字符串 “3.14”, 写入文件中。

此外，`PrintWriter` 写对象的时候，写入的是对象的 `toString()`方法返回值，与对象序列化有本质区别。

`PrintWriter` 除了可以作为过滤流之外，还可以作为节点流。`PrintWriter` 类的构造方法中，可以直接接受一个文件名或 `File` 对象作为参数，直接获得一个输出到文件的 `PrintWriter`。当然，编码方式采用的是系统默认的编码方式。

最后，`PrintWriter` 的构造方法可以接受一个 `InputStream`，也就是说，可以使用 `PrintWriter` 进行桥转换。只不过使用 `PrintWriter` 进行桥转换的时候，无法指定编码方式，采用的是系统默认的编码方式。

下面，我们把 `PrintWriter` 当做过滤流，给出一段代码的例子：

```
import java.io.*;
public class TestPrintWriter {
    public static void main(String[] args) throws Exception {
        FileOutputStream fout
            = new FileOutputStream("poem2.txt");
        Writer w = new OutputStreamWriter(fout, "GBK");
        PrintWriter pw = new PrintWriter(w);
        pw.println("一个人在清华园");
        pw.println("我写的 Java 程序");
        pw.println("是全天下");
        pw.println("最面向对象的");
        pw.close();
    }
}
```

上面的程序在当前目录下写入了一首现代诗。读者可以思考一下，如果把 `PrintWriter` 当做节点流、以及把 `PrintWriter` 用来做桥转换，分别应当怎么改写上面的代码。

# Chp15 网络编程

## 1 网络编程基础

### 1.1 网络协议的概念

要掌握网络编程，首先要掌握网络协议的概念。我们知道，网络主要是一种沟通的媒介。在不同机器上的不同程序，可以通过网络来交换数据，从而达到互联互通的功能。为了能够相互连通，首先要规定的是协议。所谓协议，可以理解为一些规则，这些规则规定了两个需要连通的点之间，应当如何进行数据交互。

举例来说，我们可以把语言当做一种协议。人在社会上需要交流，为了能够让两个人能够相互交流相互理解，我们定义了一种协议，这种协议就是语言。当两个需要沟通的人说同一种语言时，我们就说这两个人使用了同一种协议，这样就能够进行交流。而当两个人使用不同的语言时，就无法进行交流。

此外，协议是分层次的。对于不同的层次来说，规定的协议是不同的。例如，对于底层来说，为了连通网络必须要有网线。那么网线应该怎么做，水晶头的规格是什么，这是底层的协议。与之相对应的，`ftp`、`http`等，就是在网络已经连通的情况下，不同的程序之间应当按照什么规则进行交互，这属于高层协议。

下面我们就介绍一下协议的分层。

### 1.2 OSI 七层结构和 TCP/IP 协议族

对于网络来说，有两种模型。一种是 OSI 七层模型，一种是 TCP/IP 协议族的 5 层模型。我们首先介绍 OSI 七层模型。

#### 1.2.1 OSI 模型



OSI 七层模型如上图所示。对于这个模型，我们简单进行一下阐述。在这个模型中，物理层协议，规定的是网络连接中的一些电气特性，例如网线的电压电阻等等物理参数。数据链路层则规定了在网络中传输时，底层的包结构。这两个层次相对比较低层。

网络层规定了一个数据包从一台电脑（或者其他网络终端）发出之后，应当通过什么样的方式寻找路径，从而能够传输到另一台电脑上。传输层则规定了在能够找到终端之间的通路的情况下，应当如何收发数据包。这两层是网络通信软件的基础。

在之上的会话层、表示层和应用层，表示的是如何在网络已经能够收发数据包的情况下完成网络程序之间的通信。这部分内容我们不做进一步的具体阐述。

OSI 网络模型是一种标准化的网络模型。然而，这种网络模型在设计的时候过于理想化，

并且有些层次分的过细。在实际工业生产中，用的更多的是我们接下来要介绍的：TCP/IP 网络模型。

### 1.2.2 TCP 模型



上图是 TCP 网络模型。这个模型中，物理层、数据链路层与 OSI 网络模型中的两层相对应，而 IP 层对应着 OSI 模型中的网络层，传输层对应着 OSI 中的传输层。另外，TCP 模型中的应用层，对应着 OSI 模型中的会话层、表示层和应用层，可以说是把三层并为了一层。

对于 IP 层来说，在 TCP/IP 协议栈中，网络中某一台计算机的定位，靠的是 IP 地址这种方式。因此在 TCP/IP 模型中，把网络层也称之为 IP 层。如果拿生活中的例子来比喻的话，那 IP 就可以当做是电话号码，通过电话号码能够唯一定位城市中的一台电话机。

而传输层，则可以认为是相互沟通的基础。从技术上说，传输层决定了一个数据包是如何从一台电脑传输到另一台电脑。传输层有两个协议：TCP 和 UDP 协议，这两个协议各有不同。TCP 协议是一个有连接、可靠的协议；而 UDP 协议是一个无连接，不可靠的协议。关于这两个协议的使用，我们在后面的编程中会进一步详细阐述。如何使用传输层进行网络通信，这是本章的重点。

应用层指的是应用程序使用底层的网络服务，典型的应用层协议包括 ftp、http 协议等。

事实上，大部分 Java 开发者，绝大多数时间都是在开发应用层方面的内容。而这一章介绍的内容，也许在你的 Java 程序员生涯中并不会经常接触。但是，掌握这些相对底层的技术，了解底层工作的原理，对以后理解 Web 编程是有非常大的帮助的。

下面我们就分别介绍传输层的两个协议。

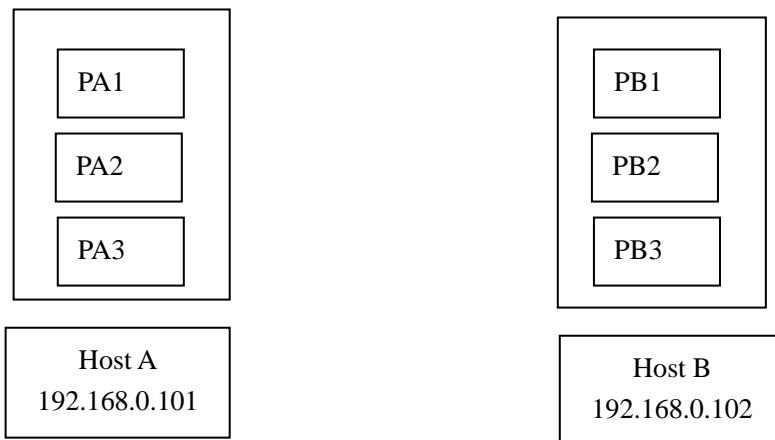
## 2 TCP 编程

### 2.1 TCP 协议简介

TCP 协议是传输层的协议，这个协议也是使用最广泛的协议之一。与 UDP 协议相比，TCP 协议的特点是：TCP 协议是一个有连接、可靠的协议。所谓有连接，指的是在进行 TCP 通信之前，两个需要通信的主机之间要首先建立一条数据通道，就好像打电话进行交流之前，首先要让电话接通一样。所谓可靠，指的是 TCP 协议能够保证：1、发送端发送的数据不会丢失；2、接收端接受的数据包的顺序，会按照发送端发送的包的顺序接受。也就是说，TCP 协议能够保证数据能够完整无误的传输。

此外，要进行 TCP 编程，还要理解端口号的概念。在网络通信的时候，我们连接服务器时，往往不仅要知道服务器的 ip 地址，还要知道服务器的端口号。那什么是端口号呢？

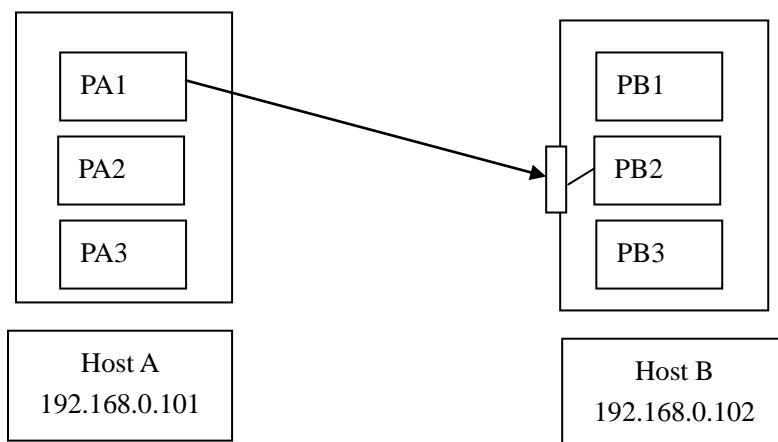
网络通信的本质，就是进程间通信。比如，两台电脑进行 qq 聊天，本质上就是两台电脑中的 qq 进程进行相互通信。对于这种情况的描述，如下图所示：



Host A 中运行了三个进程：PA1、PA2、PA3，Host B 中也运行了三个进程：PB1、PB2、PB3。所谓的网络通信，指的就是 HostA 中的某个进程和 HostB 中的某个进程进行通信。假设 PA1 要和 PB2 进行通信。

那么首先，PA1 要定位到网络中的某一台电脑，它可以用 IP 地址，在本例中，使用 192.168.0.102，就能定位到 Host B。下一步，由于 HostB 中存在多个进程，那如何才能说明要跟哪一个进程通信呢？

HostB 中的每个进程会绑定在一个端口号上，假设 PB2 绑定了 9000 端口，因此，HostA 如果要通信的话，就可以连接到 HostB 的 9000 端口上。如下图：



可以这么来理解端口号：IP 地址就好比是一个单位的电话号码，而端口号则好比是某一部电话的分机号。通过电话号码找到某一个单位，而通过分机号则可以找到具体的个人；这就类似于通过 IP 地址可以定位某一台电脑，而通过端口号可以找到电脑中某一个的进程。

值得注意的是，在一台主机中，每个进程端口号上只能绑定唯一的一个进程。

## 2.2 TCP 编程的基本步骤

我们下面通过一个例子，来解释一下基本的 TCP 编程。TCP 编程又称为 Socket 编程，主要的类有两个：`java.net.ServerSocket` 和 `java.net.Socket`。

首先，从服务器端开始。在服务器端首先要做的是创建一个 `ServerSocket` 类型的对象。创建这个对象的时候，使用下面的方式：

```
ServerSocket ss = new ServerSocket(9000);
```

这样，就创建了一个 `ServerSocket` 对象，并把这个对象绑定到了 9000 端口。而 IP 地址，则是电脑当前的地址。在 Windows 中，可以使用 `ipconfig` 命令来查看当前地址，如下：

```
D:\book>ipconfig

Windows IP Configuration

Ethernet adapter 本地连接:

    Media State . . . . . : Media disconnected

Ethernet adapter 无线网络连接:

    Connection-specific DNS Suffix . :
    IP Address . . . . . : 150.236.56.101
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : 150.236.56.1
```

可以看到，当前地址为 150.236.56.101。

创建完 `ServerSocket` 对象之后，下一步就应该调用 `ServerSocket` 的 `accept` 方法。这个方法返回一个 `Socket` 对象。怎么来理解这个方法呢？我们可以理解为，这就是服务器端在等待连接的过程。如果没有一个客户端连接过来的话，则这个 `accept` 方法会一直不会返回。而一旦有一个客户端连接服务器，则这个方法就会返回。返回的 `Socket` 对象，我们可以比喻为一部电话机，很显然，通过一个电话号码和一个分机号，可以唯一的找到一台电话机。

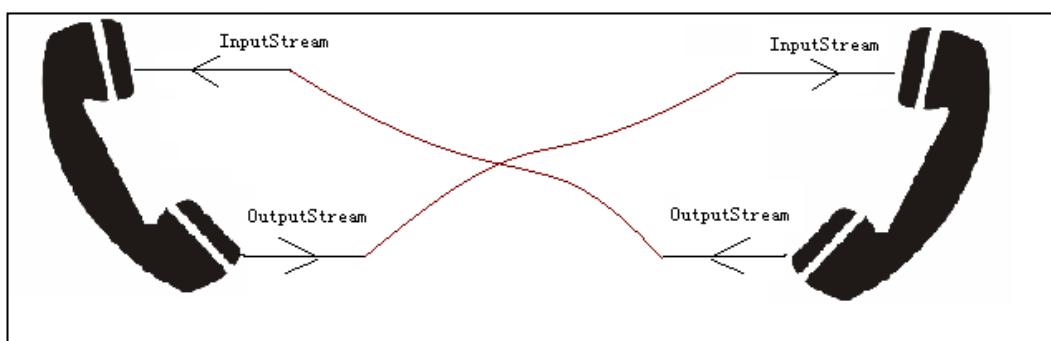
那客户端怎么连接服务器端呢？可以在客户端创建一个 `Socket` 对象连接服务端，如下：

```
Socket s = new Socket("150.236.56.101", 9000);
```

在创建 `Socket` 对象的时候，给出 IP 地址并给出端口号。这样就能创建一个 `Socket` 对象并连接服务器。

此时，客户端和服务器端都有一个 `Socket` 对象，并且保持连通，这就意味着两部电话已经接通了。接下来，就要进行数据传输了。

在数据传输的时候，需要调用 `Socket` 对象的 `getInputStream` 方法获得一个输入流，调用 `getOutputStream` 方法获得一个输出流。输入流就好比是电话机的听筒，能够听到从对方传来的声音，而输出流就好像是话筒，能够向对方说话。示意图如下：



上面的示意图说明，一端的 `InputStream` 连接在另一端的 `OutputStream` 上面，换句话说，在某一端的 `Socket` 上利用 `OutputStream` 写数据，就可以在另一端的 `Socket` 上用 `InputStream` 读数据。这就好比打电话的时候，向某一端的话筒说话，就能传到另一端的听筒。

获得 `InputStream` 和 `OutputStream` 之后，就能进行 I/O 操作了。当 I/O 操作完成之后，注意，不应该关闭 I/O 流，而应该调用 `socket` 对象的 `close` 方法，关闭 `socket`。这一点与之前的 I/O 不同。

总结一下通信的过程。服务器端：

- 1、创建 ServerSocket 对象（并绑定端口）
- 2、调用 accept 方法，等待来自客户端的连接
- 3、调用 getXXXStream 方法，进行 I/O
- 4、关闭 Socket

客户端：

- 1、创建 Socket 对象，并连接服务器
- 2、调用 getXXXStream 方法，进行 I/O
- 3、关闭 Socket

下面给出一个示例代码。在这个例子中，我们创建一个 TCP 服务器，和一个 TCP 客户端。客户端向服务器端发送一个“hello”字符串。服务端接受客户端发送的字符串，在后面增加一个“ from server”字符串，再返回给客户端。

服务器端代码：

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

public class TCPServer {
    public static void main(String[] args) throws Exception {
        //创建 ServerSocket 对象（并绑定端口）
        ServerSocket ss = new ServerSocket(9000);
        //调用 accept 方法
        Socket s = ss.accept();

        //调用 getXXXStream 方法，进行 I/O
        BufferedReader br =
            new BufferedReader(
                new InputStreamReader(s.getInputStream())
            );
        String line = br.readLine();

        PrintWriter pw = new PrintWriter(
            s.getOutputStream());
        pw.println(line + " from server");
        pw.flush();
        //关闭 Socket
        s.close();
    }
}
```

客户端代码：

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

public class TCPClient {
    public static void main(String[] args) throws Exception {
```

```

//创建 Socket 对象（并连接服务器）
Socket s = new Socket("150.236.56.101", 9000);
//调用 getXXXStream 方法，进行 I/O
PrintWriter pw = new PrintWriter(s.getOutputStream());
pw.println("hello world");
pw.flush();

BufferedReader br = new BufferedReader(
    new InputStreamReader(s.getInputStream())
);
String line = br.readLine();
System.out.println(line);
//关闭 Socket
s.close();
}
}

```

### 2.3 多线程的服务器

上述的程序演示了基本的 TCP 服务器，但是这样的服务器是有很大的缺陷：启动服务器之后，只能让一个用户进行访问，访问之后服务器就会被关闭。这种服务器在现实生活中根本无法使用。

为了解决上面的问题，我们可以使用一个 while(true) 的死循环，这样来保证服务器程序不会终止。这样，服务器端的步骤就改成：

创建 ServerSocket 对象（并绑定端口）  
while(true){  
 调用 accept 方法  
 调用 getXXXStream 方法，进行 I/O  
 关闭 Socket  
}

注意，创建 ServerSocket 对象只需要一次，可以反复调用 accept 接听电话。

然而，这样的服务器依然不完善：这个服务器同时只能有一个客户端进行 I/O 操作。现实生活中，所有的服务器都应当是允许多用户同时浏览和访问的。如果一个网站做成这个样子：当一个用户浏览网站的时候，其他用户如果只能等待，那这样的网站是不会有人愿意访问的。

为了解决这个问题，我们可以把这个服务器设计成一个多线程的服务器。主线程只负责等待客户端的连接请求，一旦有客户端成功的连接过来，主线程负责创建并启动一个新的线程，由这个新线程负责与该客户端进行 I/O 操作，而此时的主线程，则继续去等待其他客户端的连接。修改成多线程的服务器基本工作步骤如下：

创建 ServerSocket 对象（并绑定端口）  
while(true){  
 调用 accept 方法  
 创建并启动新线程进行 I/O 操作  
}

我们可以看到，原有的 I/O 操作被挪到了新线程中，主线程只负责接受用户的连接。这

样，把速度较慢的 I/O 操作多线程处理，就能允许多个用户同时访问服务器。修改后的服务器代码如下：

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

class ServerThread extends Thread{
    private Socket s;

    public ServerThread(Socket s) {
        this.s = s;
    }

    public void run(){
        try{
            BufferedReader br =
                new BufferedReader(
                    new InputStreamReader(s.getInputStream())
                );
            String line = br.readLine();

            PrintWriter pw = new PrintWriter(
                s.getOutputStream());
            pw.println(line + " from server");
            pw.flush();
        } catch (IOException e) {
            e.printStackTrace();
        } finally{
            try {
                s.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

public class TCPServer {
    public static void main(String[] args) throws Exception {
        //创建ServerSocket 对象（并绑定端口）
        ServerSocket ss = new ServerSocket(9000);
        while(true){
            //调用 accept 方法
            Socket s = ss.accept();
            //创建新线程进行 I/O
```

```
        Thread t = new ServerThread(s);
        t.start();
    }
}
}
```

## 3 UDP 编程

### 3.1 UDP 协议简介

介绍完了 TCP 编程，我们开始介绍 UDP 编程。与 TCP 协议相比，UDP 是一个无连接，不可靠的协议。即：数据的发送方只负责将数据发送出去，数据的接受方只负责接受数据。发送方和接收方不会相互确认数据的传输是否成功。

使用 UDP 通信有点类似于写信，当我们寄信的时候，不需要想打电话一样事先准备一个连接，寄信人只知道把信寄了出去，但是对方有没有收到信，寄信人则一无所知。

相对于 TCP 而言，UDP 有一个优点：效率较高。因此，当我们在对数据传输的正确率不太关心，但是对传输效率要求较高的情况下，可以采用 UDP 协议。典型的使用 UDP 协议的是网络语音以及视频聊天应用。

下面我们来采用 UDP 协议进行 Java 网络编程。

### 3.2 UDP 编程基本步骤

UDP 编程所要用到的两个主要的类：DatagramSocket 和 DatagramPacket。我们可以把进行 UDP 通信比作收发传真。其中，DatagramSocket 可以当做是一台传真机，传真机既可以发传真，又可以收传真。而 DatagramPacket 则是需要传输的数据。

假设我们现在要从客户端向服务器端发送一个“hello server”，而服务器端回给客户端一个字符串：“hello client”。

首先看服务器端。先要创建一个 DatagramSocket 类型的对象，代码如下：

```
DatagramSocket socket = new DatagramSocket(9000);
```

此时，创建了一个绑定到端口号 9000 的 DatagramSocket。这样，就准备好了个传真机。

再看客户端，客户端要发送数据，也得先创建一个 DatagramSocket，代码如下：

```
DatagramSocket socket = new DatagramSocket();
```

上面的代码没有指定客户端的端口，这样的话，系统会自动为客户端分配一个随机的端口号。

为什么服务器端的端口号不能让系统随机分配呢？因为服务器端的地址和端口号是必须要向外界公布，供客户端去访问的，如果一个网站向外公布：网站地址：xxxxxxxx，网站端口：随机，这样的话让用户究竟怎么访问你的网站？

因此，服务器端必须手动指定端口号。

客户端创建了一个 DatagramSocket 之后，就可以准备发送的数据了。首先应当准备一个 byte 数组，如下：

```
String str = "hello server";
byte[] data = str.getBytes();
```

这样，`data` 数组中保存的就是需要发送的数据内容。

然后，应当把 `data` 封装到一个 `DatagramPacket` 中，代码如下：

```
DatagramPacket packet = new DatagramPacket(  
    data, 0, data.length,  
    new InetSocketAddress("150.236.56.101", 9000)  
);
```

在创建 `packet` 的时候，给出了四个参数。前三个参数表示，发送的数据是 `data` 数组，从数组下标为 0 的位置开始发送，发送长度为 `data.length` 个字节。

第四个参数，是一个 `InetSocketAddress` 对象，这个对象表示数据要发送到哪个地址去。

当一切准备就绪之后，客户端就可以调用 `socket` 的 `send` 方法，发送 `packet` 对象。代码如下：

```
socket.send(packet);
```

这样，数据就从客户端发送到了服务器。

接下来，就是服务器如何接收了。在接收数据的时候，同样需要一个 `DatagramPacket`。这个 `DatagramPacket` 对象就好比是传真机上的纸，在收传真的时候，需要传真机里放上白纸，然后传真机根据发过来的内容，在白纸上打印出来。当传真接受完毕之后，这张纸上就记录了接受到的内容。

我们首先要创建一个空数组，这个数组就好像是白纸。

```
byte[] buf = new byte[100];
```

然后，根据这张白纸，创建一个 `DatagramPacket`：

```
DatagramPacket paper = new DatagramPacket(buf, 0, buf.length);
```

创建了 `paper` 对象之后，就可以调用 `socket` 对象的 `receive` 方法接受数据。

```
socket.receive(paper);
```

接受到数据之后，可以通过 `paper` 对象的下面几个方法获得相关信息：

`paper.getSocketAddress()`：获得发送者的地址。可以理解为获得发送传真的对方的号码，等一会儿回传真的时候，就可以用这个地址。

`paper.getLength()`：获得发送的数据的长度。虽然我们用一个长度为 100 的数据包去接受，但是接受到的数据长度有可能不满 100 个字节，可以通过调用这个 `getLength` 方法来获取实际接受的数据长度。

这样，我们就能获得客户端给服务器端发送的数据，代码如下：

```
String str = new String(buf, 0, paper.getLength());
```

之后，我们就可以从服务器端向客户端发送数据。部分代码如下：

```
byte[] data = "hello client".getBytes();  
DatagramPacket packet = new DatagramPacket(  
    data, 0 ,data.length,  
    paper.getSocketAddress()  
>;
```

需要注意的是，由于是回信给客户端，因此，我们这次发送数据的收信人，就是 `paper` 数据包的发信人。所以，`packet` 对象的地址，就是 `paper` 对象的 `getSocketAddress()` 方法的返回值。

最后，当两边完成通信之后，应当关闭 `socket`。

其余代码比较简单，不再赘述。完整代码如下：

服务器端:

```
import java.io.*;

public class UDPServer {
    public static void main(String[] args) throws Exception {
        //创建socket
        DatagramSocket socket = new DatagramSocket(9000);

        //收数据
        byte[] buf = new byte[100];
        DatagramPacket paper = new DatagramPacket(
            buf, 0, buf.length);
        socket.receive(paper);
        String str = new String(buf, 0 , paper.getLength());
        System.out.println(str);

        //发送数据
        byte[] data = "hello client".getBytes();
        DatagramPacket packet = new DatagramPacket(
            data, 0 ,data.length,
            paper.getSocketAddress()
        );
        socket.send(packet);
        //关闭socket
        socket.close();
    }
}
```

客户端:

```
import java.net.*;

public class UDPClient {
    public static void main(String[] args) throws Exception {
        DatagramSocket socket = new DatagramSocket();
        String str = "hello server";
        byte[] data = str.getBytes();
        DatagramPacket packet = new DatagramPacket(
            data, 0, data.length,
            new InetSocketAddress("150.236.56.101", 9000)
        );
        socket.send(packet);
    }
}
```

```

        byte[] buf = new byte[100];
        DatagramPacket paper = new DatagramPacket(
            buf, 0, buf.length
        );
        socket.receive(paper);
        String msg = new String(buf, 0, paper.getLength());
        System.out.println(msg);
        socket.close();
    }
}

```

## 4 URL 编程

最后，简单介绍一下 Java 中的 URL 编程。

URL 是统一资源定位符（Uniform Resource Locator）的简称，用于表示 Internet 上某一资源的地址。Internet 上的网络资源非常丰富，如常见的万维网和 FTP 站点上的各种文件、目录等。URL 的语法格式通常如下所示：

协议名 :// 主机名（或者 IP 地址）: 端口号 / 资源路径

如：<http://localhost:8080/web/image/a.jpg> 这就是一个 URL，指向了网络上的一个图片文件。（注意：localhost 是一个主机名，对应的 IP 地址是 127.0.0.1。这个地址指向的是本地主机，也就是当前这台计算机）

Java 语言同样提供了用 URL 来访问网络资源的编程方法：

Java 中的 URL 编程主要用到的类有两个：URL 和 URLConnection。

使用基本的 URL 编程非常简单：

1、创建 URL 对象

2、调用 URL 对象的 openConnection 方法，获得 URLConnection

3、调用 URLConnection 方法的 getInputStream，获得输入流，从而读取资源

4、I/O 操作

5、关闭 I/O 流

下面的代码能够读取新浪 (<http://www.sina.com.cn>) 首页的 html 源代码，并输出到屏幕上。

```

import java.net.*;
import java.io.*;

public class TestURL {
    public static void main(String[] args) throws Exception {
        // 创建 URL 对象
        URL url = new URL("http://www.sina.com.cn");
        // 调用 URL 对象的 openConnection 方法，获得 URLConnection
        URLConnection conn = url.openConnection();
        // 调用 URLConnection 方法的 getInputStream
        InputStream in = conn.getInputStream();
        // I/O 操作
        BufferedReader br = new BufferedReader(

```

```
    new InputStreamReader(in));
String line = null;
while( (line=br.readLine()) != null ){
    System.out.println(line);
}
//关闭 I/O 流
br.close();
}
}
```

该程序会显示一些“杂乱无章”的字符，不过不用担心，这些是一个 html 文件的源代码，这些代码如果在浏览器上显示出来，你就会看到一张漂亮的页面了！

# Chp16 反射

反射是 Java 中非常重要的一个语言特性，反射的强大和完善，让 Java 语言在工程实践中的灵活性大大的增强，使得 Java 程序在运行时可以探查类的信息，动态的创建类的对象，获知对象的属性，调用对象的方法。因此，反射技术被广泛的应用在一些工具和框架的开发上。

也许，并不是每一个程序员都有机会利用反射 API 进行他们的 Java 开发，但是，学习反射是一个 Java 程序员必须要走过的道路之一，对反射的掌握能够帮助程序员更好的理解后面很多的框架和 Java 工具，毕竟这些框架和工具都是采用反射作为底层技术的。

首先，先来看看几个编程中的问题。给出下面两个需求：

1、给定一个对象，要求输出这个对象所具有的的所有方法的名字。即，写出类似下面的函数：

```
public static void printMethod(Object obj)
```

2、给定一个字符串参数，这个参数表示一个类的名字。根据类名，创建该类的一个对象并返回。即写出类似下面这种定义的函数：

```
public static Object createObject(String className)
```

思考一下，用现在的知识，能做到这一点么？有一点困难吧。

## 1 类对象

### 1.1 概念

要理解反射，首先要理解的是“类对象”的概念。

Java 中有一个类，`java.lang.Class` 类。这个类的对象，就称之为类对象。

那类对象用来干什么呢？比如，以前我们写过学生类，一个学生对象都是用来保存一个学生的信息。而一个类对象呢，则用来保存一个类的信息。所谓类的信息，包括：这个类继承自哪个类，实现了哪些接口，有哪些属性，有哪些方法，有哪些构造方法……等等。

我们之前提到过类加载的概念。当 JVM 第一次遇到某个类的时候，会通过 CLASSPATH 找到相应的.class 文件，读入这个文件并把读到的类的信息保存起来。而类的信息在 JVM 中，则被封装在了类对象中。

这种阐述比较抽象，我们可以举一个非常形象的例子。我们在动物园中，能够看到动物，我们见到的都是活生生的对象。例如，我们在笼子中见到了三条狗，那实际上是说，我们遇到了三个狗对象。

在关着狗的笼子外面，一般会有一个牌子。方便游人更好的认识这种动物。

例如，牌子上会有这样的信息：狗，脊椎动物门，哺乳纲，食肉目，犬科。这部分信息，实际上是在说明，狗的父类是什么，介绍的是狗这个类的继承关系。

狗能当宠物，这说明的是狗实现了什么接口。

狗有四条腿，脚上有每个脚上有 4 个脚趾，有尾巴…… 这些，表明的都是狗有什么，实际上说明的是狗的属性。

狗吃肉，能看家，能拉雪橇……这说明的是狗有哪些方法。

狗什么时候繁殖，一胎生多少只小狗……这是狗的构造方法。

换句话说，在动物园的牌子上，写满了狗这个类的信息。（注意，牌子上写的是对象的信息。对象信息是什么呢？比如，笼子里的狗叫什么名字，是公还是母，年龄多大……显然这些不会写在牌子上）

下面，我们思考这个牌子。首先，这个牌子也是一个对象；其次，这个牌子对象的作用，就是用来保存狗这个类的信息。

因此，我们所说的类对象，就非常类似于动物园里的牌子：这种对象的创建，就是为了保存类的信息。

## 1.2 获取类对象

接下来，我们来介绍一下如何获得类对象。获得类对象总共有三种方式。

### 1.2.1 类名.class

可以通过类名.class 的方法直接获得某个类的类对象。例如，如果要获得 Student 类的类对象，就可以使用 Class c = Student.class。

这种方法获得类对象比较直接，并且还有一个非常重要的特点。对于基本类型来说，他们也有自己的类对象，但是要获得基本类型的类对象，只能通过类型名.class 来获得类对象。例如下面的代码，就能获得 int 类型的类对象

```
Class c = int.class;
```

### 1.2.2 getClass()方法

Object 类中定义了一个 getClass()方法，这个方法也能获得类对象。我们之前曾经介绍过这个方法，前文中，把这个方法称之为获得对象的实际类型。而现在我们可以知道，这个方法实际上是返回某个对象的类对象。

例如，我们可以对一条狗（狗对象）调用它的 getClass()方法，此时，它会叼着那块牌子返回给你。

另外，由于多条狗公用一个牌子，也就是说，同一类型的对象公用一个类对象，因此，对同一类型的任何一个对象调用 getClass()方法，返回的应该是同一个类对象。

正因为如此，对类对象的比较，可以使用 “==”。可以回顾一下 equals 方法的写法，在比较实际类型时，使用的是 getClass()方法，用 “==” 比较两个类对象。

当我们有一个对象，而想获得这个对象的类对象时，应当调用这个对象的 getClass()方法。因此，getClass()方法主要用于：通过类的对象获得类对象。

### 1.2.3 Class.forName()方法

在 Class 类中有一个静态方法，这个静态方法叫做 forName。方法的签名如下：

```
public static Class forName(String className) throws ClassNotFoundException
```

这个方法接受一个字符串作为参数，这个字符串参数表示一个类的类名。这个静态方法能够根据类名返回一个类对象。

需要注意的是两点：

1、当 className 所代表的类不存在时，这个方法会抛出一个已检查异常 ClassNotFoundException。

2、这个方法接受的字符串参数，必须带包名。举例来说，如果想要利用 Class.forName 获得 ArrayList 这个类的类对象的话，必须使用 Class.forName("java.util.ArrayList")这样的方式获得类对象，而不是 Class.forName("ArrayList")。

这种获得类对象的方式还有其他的作用。考虑下面的代码：

```
public static void main(String args[]) throws Exception{
    Class c = Class.forName("p1.p2.TestClass");
}
```

在上面的代码中，主方法中利用 `Class.forName()` 获得 `p1.p2.TestClass` 类的类对象。而我们知道，类对象是在类加载之后才会产生的。在调用 `Class.forName()` 方法的时候，`p1.p2.TestClass` 这个类并没有被加载。因此，要获得这个类的类对象，必须要先加载这个类。`Class.forName()` 就会触发类加载的动作。

有些时候，我们可以用 `Class.forName()` 这个方法，来进行“强制类加载”的操作。

获得类对象之后，接下来要做的事情就是使用类对象。获得的类对象究竟有什么用处呢？应该如何使用呢？这就是我们接下来要介绍的。

为了介绍类对象的使用，我们首先创建一个 `Student` 类。代码如下：

```
public class Student {
    public String name;
    private int age;

    public Student() {}

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void study() {
        System.out.println(name + " study");
    }

    public void study(int h) {
        System.out.println(name + " study for " + h + " hours");
    }

    public double study(int a, double b) {
```

```

        System.out.println(name + " study " + a + " " + b);
        return a * b;
    }

    private void play(){
        System.out.println(name + " play");
    }

}

```

这个 Student 类中包括两个属性，一个公开的 name 属性和一个私有的 age 属性，并对这两个私有属性提供了相应的 get/set 方法。

此外，Student 类还定义了三个重载的 study 方法，并定义了一个私有的 play 方法。

接下来，用 Student 类的类对象来演示如何使用类对象。

### 1.3 使用类对象获取类的信息

当我们获得了类对象，当然就可以调用类对象中的方法。例如：

- getName(): 获得类的名称，包括包名
- getSimpleName(): 获得类的名称，不包括包名
- getSuperClass(): 获得本类的父类的类对象
- getInterfaces(): 获得本类所实现的所有接口的类对象，返回值类型为 Class[], 当然，这是对的，一个类可以实现多个接口。

我们来看如下代码：

```

import java.util.ArrayList;
public class TestClass1 {
    public static void main(String[] args) {
        Class c = ArrayList.class;

        String className = c.getName();
        System.out.println("类名: "+className);

        String simpleName = c.getSimpleName();
        System.out.println("简单类名: "+simpleName);

        Class superClass = c.getSuperclass();
        System.out.println("父类: "+superClass.getName());

        Class[] interfaces = c.getInterfaces();
        System.out.println("接口: ");
        for(int i =0 ; i < interfaces.length ; i++){
            System.out.println(interfaces[i].getName());
        }
    }
}

```

运行结果：

```
C:\> C:\WINDOWS.1\system32\cmd.exe
D:\book>java TestClass1
类名: java.util.ArrayList
简单类名: ArrayList
父类: java.util.AbstractList
接口:
java.util.List
java.util.RandomAccess
java.lang.Cloneable
java.io.Serializable

D:\book>
```

该程序通过分析类对象，打印出了 `ArrayList` 类的父类以及所实现的接口，和 API 文档中提示的是一致的。

## 1.4 使用类对象获取类中方法的信息

在 `Class` 类中，有两个方法，这两个方法签名如下：

```
public Method[] getDeclaredMethods() throws SecurityException
public Method[] getMethods() throws SecurityException
```

这两个方法都抛出 `SecurityException` 异常，这个异常是一个未检查异常，可处理可不处理。

这两个方法都返回一个 `Method` 类型的数组。`Method` 类是在 `java.lang.reflect` 包中定义的类。`Method` 类用来表示方法，一个 `Method` 对象封装了一个方法的信息。我们可以调用 `Method` 对象的 `getName()` 方法获得方法名，也可以直接调用 `Method` 对象的 `toString()` 方法直接返回方法的签名。

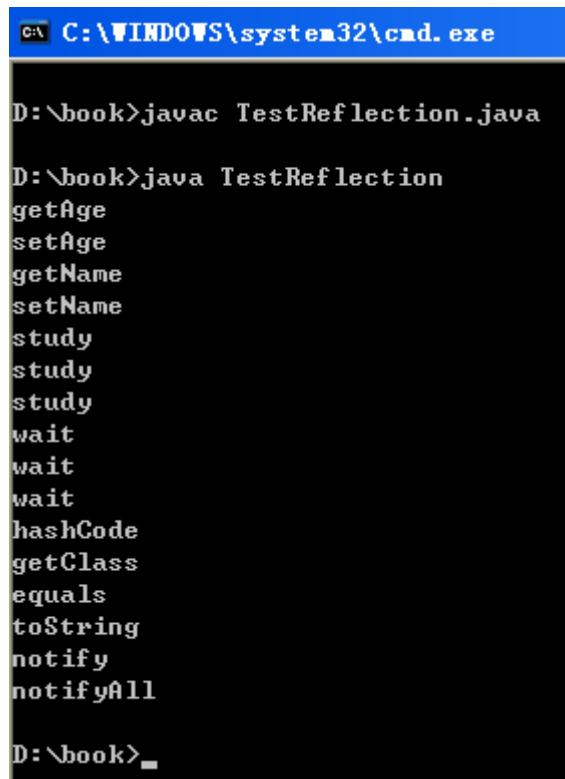
以上所述的两个方法，都可以返回类中所有方法的信息，由于一个方法的信息会封装在一个 `Method` 对象中，因此，返回值类型均为 `Method[]`。

同样是返回 `Method` 数组，那 `getMethods()` 和 `getDeclaredMethods()` 有什么区别呢？

对于 `getMethods()` 来说，返回的 `Method` 类型的数组中，包括所有的公开方法，也包括父类中定义的公开方法。对于 `Student` 类来说，既包括在 `Student` 类中定义的 `study` 以及一系列 `get/set` 方法，也包括在 `Object` 类中定义而被 `Student` 类继承的方法，例如 `toString`、`equals` 方法等。但是，私有方法不会被获取。演示代码如下：

```
import java.lang.reflect.*;
public class TestReflection {
    public static void main(String[] args) {
        Class c = Student.class;
        Method[] ms = c.getMethods();
        for(int i = 0; i<ms.length; i++) {
            System.out.println(ms[i]);
        }
    }
}
```

运行结果如下：



```
C:\> C:\WINDOWS\system32\cmd.exe

D:\book>javac TestReflection.java

D:\book>java TestReflection
getAge
setAge
getName
setName
study
study
study
wait
wait
wait
hashCode
getClass
equals
toString
notify
notifyAll

D:\book>
```

可以看出，包括了父类的中的公开方法，但是不包括任何的私有方法（例如 play 方法）。

而 getDeclaredMethods 方法，则会返回在本类中定义的所有方法，包括私有方法。也就是说，对于 Student 类的类对象而言，调用 getDeclaredMethods 方法会获得在 Student 类中定义的所有方法，包括私有的 play 方法。但是，不能获得父类中的任何方法。演示代码如下：

```
import java.lang.reflect.*;
public class TestReflection {
    public static void main(String[] args) {
        Class c = Student.class;
        Method[] ms = c.getDeclaredMethods();
        for(int i = 0; i<ms.length; i++){
            System.out.println(ms[i].toString());
        }
    }
}
```

运行结果如下：

```
cmd C:\WINDOWS\system32\cmd.exe
D:\book>javac TestReflection.java
D:\book>java TestReflection
public int Student.getAge()
public void Student.setAge(int)
public java.lang.String Student.getName()
public void Student.setName(java.lang.String)
public void Student.study(int)
public double Student.study(int,double)
public void Student.study()
private void Student.play()

D:\book>
```

可以看到，相比前一个程序，结果中多出了 play 方法，但是少了很多 Object 类中的方法。并且，相对于 getName 只能获得方法名，Method 对象的 toString 方法能够获得对象的完整签名。

至此，我们可以完成本章开始时提出的需求 1：给定一个对象，输出这个对象的所有方法的名字。示例代码如下：

```
public static void printMethod(Object obj) {
    //获取obj对象所对应的类对象
    Class c = obj.getClass();

    //通过类对象，获取其中的所有方法对象
    Method[] ms = c.getMethods();

    //打印每个方法的方法名
    for(int i = 0 ; i < ms.length ; i++) {
        System.out.println(ms.getName());
    }
}
```

## 1.5 使用类对象创建类的对象

类对象除了能够获知类中有哪些方法之外，还有着很多其他很有价值的功能。例如，在 Class 类中有一个方法： newInstance()，这个方法能够通过类的无参构造方法，创建一个对象。也就是说，我们可以通过类对象创建一个类的对象。（当然，这有些不符合刚刚我们说的例子，我们不能拿着一个写满狗的信息的牌子，就创建一条狗吧……）

例如，如果要创建一个 Student 对象，除了可以直接 new 之外，还能够用下面的方法：

```
Class c = Student.class;
Student stu = (Student) c.newInstance();
```

这样创建对象的时候，会调用对象的无参构造方法。为了证明这一点，我们修改 Student 类的无参构造方法如下：

```

public Student() {
    System.out.println("Student()");
}

然后，运行示例代码：
import java.lang.reflect.*;
public class TestReflection {
    public static void main(String[] args) throws Exception {
        Class c = Student.class;
        Student stu = (Student) c.newInstance();
    }
}

```

结果如下：

```

C:\WINDOWS\system32\cmd.exe

D:\book>javac TestReflection.java

D:\book>java TestReflection
Student()

```

输出 Student(), 说明无参构造方法被调用，创建了一个 Student 对象。

至此，开头的需求 2 也可以完成。示例代码如下：

```

public static Object createObject(String className) {
    Object result = null;
    try{
        Class c = Class.forName(className);
        result = c.newInstance();
    }catch(Exception e){
        e.printStackTrace();
    }
    return result;
}

```

## 2 反射包

上一部分介绍了反射的一些基本内容。这一节将在上一节的基础上，进一步学习反射的使用，利用反射完成更多的事情。

接下来要学习的这几个类，都在 `java.lang.reflect` 这个反射包下面。

### 2.1 Field 类

`Field` 类封装了属性信息，一个 `Field` 对象封装了一个属性的信息。

#### 2.1.1 获取特定属性

首先，学习 `Field` 类第一部分，就是如何获得 `Field` 对象。在 `Class` 类中，有以下两个方法：

```
Field getDeclaredField(String name)
```

```
Field getField(String name)
```

故名思意，这两个方法能够根据属性名，获得相应的 Field 对象。`getField` 方法可以获得本类的公开属性以及从父类继承到的公开属性，但是无法获得非公开属性；而 `getDeclaredField` 方法只能获得本类属性，但这包括本类的非公开属性。这点区别类似于 `getMethods` 方法和 `getDeclaredMethods` 方法，不是吗？

例如，如果要获得代表 `Student` 类的 `name` 属性的 `Field` 对象，则可以使用下面的代码：

```
Class c = Student.class;
Field nameField = c.getField("name");
```

这样，就可以获得相应的 `Field` 对象。

### 2.1.2 修改、读取属性

有了 `Field` 对象之后，我们还可以使用反射来获取、修改属性的值。

首先，我们分析一下，如果不使用反射的话，应当如何对属性的值进行读取的修改。代码如下：

```
Student stu = new Student();
stu.name = "tom"; //修改属性值
String data = stu.name; //获取属性值
```

要注意的是，修改属性值，有三个要素：

- 1、`stu`。这个要素说明的是，我们要修改哪一个对象的属性；
- 2、`.name`。这个要素说明的是，我们希望修改的是对象的哪一个属性。
- 3、`"tom"`。这个要素说明的是，我们希望把对象的属性值修改成什么。

分析清楚了这三个要素之后，我们就可以学习利用反射来修改属性。

首先我们必须获得即将被修改的属性所对应的 `Field` 对象，然后对 `Field` 对象调用 `set` 方法。`set` 方法签名如下（不包括抛出的异常）

```
public void set(Object obj, Object value)
```

这个方法有两个参数，第一个参数表示要修改属性的对象，第二个参数表示属性值要修改成什么。

我们可以用代码来表示：

```
Student stu = new Student();
Class c = stu.getClass();
Field nameField = c.getField("name"); //1
nameField.set(stu, //2
    "tom"); //3
```

上面的代码，同样把 `stu` 对象的 `name` 属性设为了 `tom`。我们来分析一下反射的这部分代码。

//1 的位置，是获取相应的 `Field` 对象。我们可以认为这是对应着上面所说的要素 2：要修改对象的哪一个属性。我们通过调用 `getField` 方法，说明要修改的是名字为“`name`”的属性。

//2 的位置，是 `set` 方法的第一个参数，这个参数对应着要素 1：要修改哪一个对象的属性。这里说的很明确，要修改 `stu` 对象的属性。

//3 的位置，是 `set` 方法的第二个参数。这个参数对应着要素 3：要把属性值修改成什么。

我们可以看到，使用反射设置属性，与直接使用代码设置属性，所需要的信息是一致的。

理解了怎么设置属性之后，再学习怎么获取属性就变得比较简单了。Field 类中有一个 get 方法，签名如下（不包括抛出的异常）

```
public Object get(Object obj)
```

get 方法的参数，表明了要读取哪一个对象的属性。而 get 方法的返回值，则表明了读取到的属性值。例如，要获得 stu 对象的 name 属性值，代码如下：

```
Class c = stu.getClass();
Field f = c.getDeclaredField("name");
Object value = f.get(stu);
```

### 2.1.3 私有属性

除了能够获得 Student 类中的公开属性之外，利用反射还能获得并修改对象的私有属性，如下面的代码：

```
Class c = Student.class;
```

```
Field ageField = c.getDeclaredField("age");
```

由于 age 属性是私有的，因此只能用 getDeclaredField 方法。

对于一般的途径来说，是不能直接读取、修改私有属性的。然而，反射却可以突破属性私有的限制，只需要在读取和修改之前调用一个方法：

```
public void setAccessible(boolean flag)
```

为这个方法传递一个参数 true 就可以解决问题。参考代码如下：

```
Student stu = new Student();
```

```
// stu.age = 18; 不能直接修改 age 属性，这句代码将无法编译通过
```

```
Field f = stu.getClass().getDeclaredField("age");
```

```
f.setAccessible(true);
```

```
f.set(stu, new Integer(18));
```

需要注意的是，虽然 age 属性是一个 int 类型，但是由于 set 方法第二个参数是 Object 类型，因此必须要把 18 这个整数值封装成 Integer 对象。

从上面的例子中我们可以看到，反射可以获取一个对象的私有属性，并且可以读取和修改私有属性。这也是我们不使用反射做不到的。

那么，这样算不算破坏封装呢？严格的说，算。但是，这种对封装的破坏并不可怕。要明确的是，反射是一种非常底层的技术，而封装相对来说是一个比较高级的概念。例如，一台服务器，要防止外部的破坏，有可能会假设一道网络防火墙。防火墙这个概念就是一个相对比较高级的概念。而这个防火墙设计的再合理，如果服务器机房的钥匙被人偷走了，让人能够进入机房偷走服务器，那么防火墙设计的再好也拦不住。防火墙防止的是高层的攻击，而底层的破坏，不需要防火墙处理。

封装也一样。封装防止的是程序员直接访问和操作一些私有的数据；而封装是一个非常底层的技术，利用反射，完全可以打破封装。

## 2.2 Method 类

Method 类在之前已经接触过，因此关于 Method 类的基本概念不再赘述。

### 2.2.1 获取特定方法

除了之前我们说的 getMethods 和 getDeclaredMethods 方法之外，还有两个方法能够获得

特定的方法对象：

```
public Method getMethod(String name, Class[] parameterTypes)  
public Method getDeclaredMethod(String name, Class[] parameterTypes)
```

两个方法的区别同样与 `getMethods` 方法和 `getDeclaredMethods` 方法类似。`getMethod` 可以获得公开方法，包括父类的；`getDeclaredMethod` 只能获得本类的方法，但不限于公开方法。

我们可以看到，`getMethod` 以及 `getDeclaredMethod` 方法有两个参数。

第一个参数是一个字符串参数，表示的是方法的方法名。

但是，光有方法名还不能确定一个方法，因为类中有可能有方法重载的情况。

为了能唯一确定一个方法，除了要给一个方法名之外，还要给出这个方法的参数表。我们用一个 `Class` 数组来表示参数表。

对于无参的方法来说，参数表就是一个空数组：`new Class[]{}`

对于有一个参数的方法来说（例如一个 `int` 类型参数），则参数表是长度为 1 的数组：  
`new Class[]{int.class}`

对于有多个参数的方法来说，则把多个参数的类型依次罗列在数组中。例如，如果一个方法接受一个 `int` 类型与一个 `double` 类型的话，则表示参数表的 `Class` 数组为：`new Class[]{int.class, double.class}`

因此，如果想要获得两个参数的 `study` 方法，则可以用下面的代码：

```
Class c = Student.class;  
Method m = c.getMethod("study",  
                      new Class[]{int.class, double.class});
```

## 2.2.2 利用反射，调用对象的方法

接下来，我们学习怎么用反射来调用方法。

首先，还是分析一下不用反射应当怎么来调用方法。

```
Student stu = new Student();  
double result = stu.study(10, 1.5);
```

上面就是调用方法的例子。这里有四个要素  
1、stu。需要说明对哪个对象调用方法  
2、study。需要说明调用的是哪个方法  
3、(10, 1.5) 需要传入实参  
4、方法可以有返回值

利用反射调用方法，首先我们必须获得即将被调用的方法所对应的 `Method` 对象，然后对 `Method` 对象调用 `invoke` 方法。`invoke` 方法签名如下（不包括抛出的异常）

```
public Object invoke(Object obj, Object[] args)
```

在这个方法中，`invoke` 方法有两个参数

- 1、第一个参数 `obj` 表示对哪一个对象调用方法
- 2、第二个参数表示调用方法时的参数表
- 3、`invoke` 方法的返回值对应于 `Method` 对象所代表的方法的返回值。

我们给出利用反射调用 `study` 方法的代码：

```
Student stu = new Student();
```

```

Class c = stu.getClass();
Method m = c.getDeclaredMethod("study", new Class[]{int.class,
double.class}); //1
Object result //2
= m.invoke(stu, //3
new Object[]{new Integer(10), new Double(1.5) } ); //4

```

我们来分析上述代码。

//1 的位置获得一个代表带两个参数的 study 方法的 Method 对象，表示调用哪个方法，对应于要素 2；

//2 是 invoke 方法的返回值，对应于要素 4；

//3 是 invoke 方法的第一个参数，表明对 stu 对象调用方法，对应于要素 1；

//4 的位置传入了两个参数，这两个参数形成调用方法时的实参，对应于要素 3。

上面就是利用反射调用方法。与 Field 对象类似，也可以调用一个类中的私有方法，只需要在调用 Method 对象的 invoke 方法之前，先调用 setAccessible(true)即可。

## 2.3 Constructor 类

我们简单介绍一下 Constructor 类，故名思意，这个类封装了构造函数的信息，一个 Constructor 对象代表了一个构造函数。

首先，可以通过 Class 类中的 getConstructors() / getDeclaredConstructors() 获得 Constructor 数组。

其次，可以通过 Class 类中的 getConstructor() / getDeclaredConstructor() 来获得指定的构造方法。与 getMethod 不同，这两个方法只有一个参数：一个 Class 数组。原因也很简单：构造方法的方法名与类名相同，不需要指定。

最后，可以调用 Constructor 类中的 newInstance 方法创建对象。创建对象的时候，会调用相应的构造方法。

如果创建对象只需要调用无参构造方法的话，就可以直接使用 Class 类中的 newInstance 方法，如果在创建对象的时候需要指定调用其他构造方法的话，就需要使用 Constructor 类。

下面的代码利用这两种不同的方式创建对象。

```

import java.lang.reflect.*;
class Dog{
    String name;
    int age;

    public Dog(){
        System.out.println("Dog()");
    }
    public Dog(String name, int age) {
        System.out.println("Dog(String, int)");
        this.name = name;
        this.age = age;
    }
    public String toString(){

```

```

        return name + " " + age;
    }
}

public class TestConstructor {
    public static void main(String[] args) throws Exception {
        Class c = Dog.class;

        Dog d1 = (Dog) c.newInstance();
        System.out.println(d1);
        //获得构造方法
        Constructor con = c.getConstructor(
            new Class[]{String.class, int.class});
        //创建对象时指定构造方法
        Dog d2 = (Dog) con.newInstance(
            //为构造方法传递的参数
            new Object[]{"Snoopy", new Integer(5)});
        System.out.println(d2);
    }
}

```

很明显，在上面的代码中 d1 对象是利用 Dog 类的无参构造方法创建出来的；而 d2 对象则是利用有参构造方法创建出来的，在创建的同时，name 属性被赋值为“snoopy”，age 属性被赋值为 5。

### 3 反射的作用

学到这里，你可能会感到疑惑。反射有什么用呢？我们在前面的学习中已经掌握了创建对象，调用方法的办法，利用反射做这些事情的优势在哪里呢？

我们来对比以下的代码：

```

Student s = new Student();
s.study();

```

这是最常规的创建 Student 对象，并调用 study() 方法的代码。

```

String className = "Student";
Class c = Class.forName(className);
Object o = c.newInstance();

String methodName = "study";
Method m = c.getMethod(methodName, new Class[] {});
m.invoke(o, new Object[] {});

```

这是利用反射的代码，做的是同样的事情。

很显然用反射写出的代码比较繁琐，可是除了繁琐呢？你看出别的端倪了么？

在反射代码中，创建对象所采用的类名“`Student`”，以及调用方法时的方法名“`study`”都是以字符串的形式存在的，而字符串的值完全可以不写在程序中，比如，从文本文件中读取。这样，如果需求改变了，需要创建的对象不再是 `Student` 对象，需要调用的方法也不再是 `study` 方法，那么程序有没有可能不做任何修改呢？当然可能，你需要修改的可能是那个文本文件。

反观不用反射的代码，它只能创建 `Student` 对象，只能调用 `study` 方法，如有改动则必须修改代码重新编译。明白了吧，用反射的代码，会更通用，更万能！

因此，利用反射实现的代码，可以在最大程度上实现代码的通用性，而这正是工具和框架在编写的时候所需要的。因此，反射才能在这些领域得到用武之地。

我们在前面的章节中提到过，利用多态，可以使代码通用，例如：

```
public void feed (Dog d ) {  
    d.eat ();  
}  
public void feed (Animal a ) {  
    a.eat ();  
}
```

很显然，以 `Animal` 为参数的 `feed` 方法要比以 `Dog` 为参数的 `feed` 方法具有更大的通用性，因为它不仅可以传入 `Dog` 对象，还可以传入其他的 `Animal` 的子类对象。但是按照这个思路，能不能再通用一点呢？请看以下代码：

```
public void feed (Object o) throws Exception{  
    Class c = o.getClass ();  
    Method m = c.getMethod ("eat", new Class [] {});  
    m.invoke (o , new Object [] {});  
}
```

这个 `feed` 方法的参数类型为 `Object`，可以传入任何对象。只要这个对象具有 `eat` 方法，就可以通过反射来实现对 `eat` 方法的调用。好了，利用反射，我们已经把多态用到极致了。因为 `Object` 类不会再有父类了。

当然，这里并不是鼓励大家滥用反射。反射技术有着非常显著的几个缺点。

1. 运行效率与不用反射的代码相比会有明显下降。
2. 代码的复杂度大幅提升，这个从代码量上大家就能比较出来
3. 代码会变得脆弱，不易调试。使用反射，我们就在一定程度上绕开了编译器的语法检查，例如，用反射去调用一个对象的方法，而该对象没有这个方法，那么在编译时，编译器是无法发现的，只能到运行时由 JVM 抛出异常。

因此，反射作为一种底层技术，只适合于工具软件和框架程序的开发，在大部分不需要使用反射的场合，没有必要为了追求程序的通用性而随意使用反射。滥用反射绝对是一个坏的编程习惯。

# Chp17 OOAD 初步

OOAD，指的是面向对象的分析与设计。作为本书的最后一章，本章将介绍很多基本的面向对象的分析和设计内容，从根本上阐述面向对象设计思想的精髓。

首先，我们将介绍 UML 图以及类关系。

## 1 UML 图以及类关系

UML 是统一建模语言的缩写，使用 UML，能够用图形的方式比较清晰的表达出软件设计师的设计意图，也能够让程序员之间更加容易的交流软件的结构和设计。

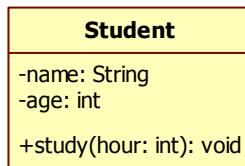
在 UML 中，共有 9 种基本的图形，包括：用例图，类图，对象图，时序图，协作图，组件图，部署图，活动图和状态转换图。其中，出现频率最高的当属类图。

类图，顾名思义，就是描述系统中类的组成，以及类与类之间的关系。

### 1.1 类图

#### 1.1.1 UML 表示类

用类图表示一个类，则可以绘制如下：



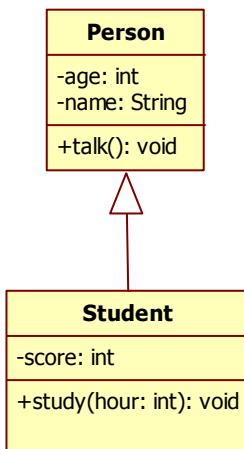
我们用一个矩形来表示一个类。把这个矩形用横线分为三个部分：第一部分用来写类名；第二部分用来写属性；第三部分用来写方法。

属性的写法：先写属性名，后写属性的类型。另外，对于访问权限修饰符，用“-”表示私有，用“+”表示公开。

方法的写法：先写方法名，然后写参数表，最后是返回值类型。

#### 1.1.2 继承关系

两个有继承关系的类，用类图表示如下：

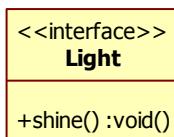


我们可以看到，子类 Student 继承 Person 类，则从子类 Student 出发，用一个空心的三角箭头指向父类。这就是用类图表示的继承关系。

### 1.1.3 接口

接口有两种画法。

接口的第一种画法是完整画法，如下：

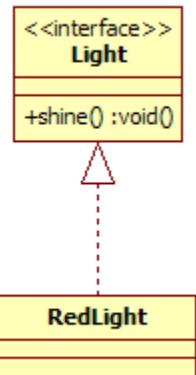


这种画法与类相似，也是用一个矩形来表示。所不同的是，在接口的名字上方，会有

`<<interface>>`

的字样，强调这是一个接口。

如果用一个类来实现这个接口，则 UML 图描述如下：

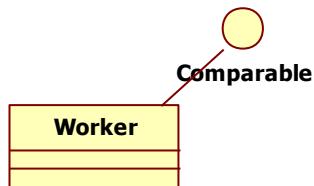


用一个空心的三角箭头指向被实现的接口，接口和实现类之间用虚线连接。

接口的第二种画法是简化的画法，如下图：



这种画法只需要画一个空心的圆形，然后写上接口的名字即可。对应于这种简化画法，一个类实现一个接口，也可以画成下面的图形：



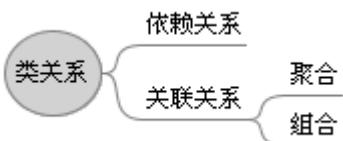
用一根实线连接接口和实现类，表示接口的实现。

## 1.2 类关系

接下来，我们要为大家介绍的是类与类之间的关系。类与类之间除了我们所熟知的继承关系之外，还可能存在以下的关系。

### 1.2.1 依赖、聚合与组合

类的关系，可以用下面的图来表示：



我们对这些关系依次来进行解释。

#### 1.2.1.1 依赖关系

所谓依赖关系，又被称为 use-a 关系。这种关系的特点是：A 类和 B 类之间如存在依赖关系，则 A 类中有一个 B 类型的局部变量（当然，方法参数也可以认为是特殊的局部变量）。

怎么来理解这个关系呢？例如，有一个人要过河，也就是说，有一个 Person 对象要调用 crossRiver 方法。而这个人要过河，需要一条小船载他过去，也就是说，他需要一个 Boat 对象。我们可以这样来理解：

```
class Person{  
    public void crossRiver(Boat b){  
        ...  
    }  
}
```

上面的代码，Boat 类是 Person 类的某一个函数的方法参数。表示如果一个人要过河的话，则他需要一条船。我们可以理解为，调用 Person 对象的 crossRiver 方法的话，需要用一下某一个船对象。而当这个方法调用完毕之后，作为局部变量的 Boat 引用即失效，Person 对象和 Boat 对象之间就毫无瓜葛，没有联系了。用英语表述的话，就是：

Person use a Boat to crossRiver.

你看，这就说明 Person 类和 Boat 类是“use-a”关系，表明他们之间是依赖关系。

用类图来表示，如下：



Person 类依赖 Boat 类，则可以用一根虚线连接 Person 和 Boat 两个类，并在被依赖的一方画一个箭头。

### 1.2.1.2 关联关系

所谓 A 类和 B 类存在关联关系，指的是 A 类中有一个 B 类型的属性。由于属性表示对象“有”什么，所以关联关系也称之为“has-a”关系。

当然，“有”也有所不同。例如，人有一双手，以及人有一辆自行车。虽然都是“有”，但是这两者还是有区别的。

人“有”自行车，在这种关系中，人和自行车是相对比较独立的对象。这个独立，指的是外部对象“人”和内部对象“自行车”的生命周期之间，没有必然的联系。有可能人还在，自行车没了；也有可能自行车还在，人没了。这种外部对象和内部对象在失掉关联关系之后，依然可以分别存在的关系，称之为“聚合”。用 UML 图表示如下：



在外部对象的一端，用一个空心的菱形表示聚合关系。

人“有”手，在这种关系中，人和手是密切不可分割的对象。外部对象一旦不存在，则内部对象也一定不存在了，换句话说，外部对象管理内部对象的生命周期。这种对象的关系称之为“组合”，用 UML 图表示如下：



在外部对象的一端，用一个实心的菱形表示组合关系。

### 1.2.2 关联关系

下面我们将对关联关系作进一步的阐述。关联关系是有方向性和多重性的。

#### 1.2.2.1 关联关系的方向性

我们来看如下代码：

```
class Person{  
    Bike bike;  
}  
class Bike{}
```

很明显，在 Person 类中存在一个 Bike 类型的属性，因此我们可以认定，Person 类和 Bike 类之间存在关联关系，但是反过来，在 Bike 类中并不存在一个 Person 类型的属性，也就是说 Bike 类和 Person 类并不存在关联关系。也就是说，关联关系有方向性，A 类关联 B 类并不意味着 B 类也关联 A 类。

对于上述代码给出的关系，Person 类关联 Bike 类，但 Bike 类并不关联 Person 类，我们称之为“单向关联”。但是对于以下代码：

```
class Man{  
    Woman wife;  
}  
  
class Woman{  
    Man husband;  
}
```

我们则可以称之为“双向关联”。因为在 Man 类关联 Woman 类的同时，Woman 类也关联了 Man 类。

对于单向关联，类图表示如下：



箭头由 Person 类指向 Bike 类，表示只有 Person 类关联 Bike 类，Bike 类没有关联 Person 类。

而对于双向关联，类图表示如下：



在 Man 类和 Woman 类之间只有一道实线，没有箭头，表示实线两端的类之间双向均存在关联关系。

#### 1.2.2.2 关联关系的多重性

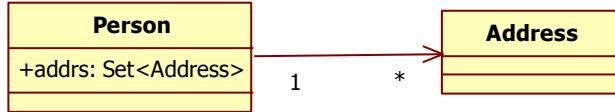
最简单的关联关系是一对一关联，也就是说，关联关系的两端往往是一个对象对应着一个对象。例如前面的例子：人和自行车（往往一个人只有一辆自行车）；丈夫和妻子(按照一夫一妻制，一个丈夫对象所关联的妻子对象也只能是一个)。这些都是一对一关联。

除了一对一关联之外，还有可能有一对多关联。例如，一个人（Person）可能有多个地址（Address）：公司地址、家庭地址、户口所在地等等。在设计代码时，Person 类中就必须采用数组或是集合来保存其所关联的所有 Address 对象。

一对多关联也有方向。例如，Person 类与 Address 类，就是单向一对多关联。我们可以从一个 Person 对象中获得它关联的所有地址对象。示例代码如下：

```
class Address{ ... }  
  
class Person{  
    Set<Address> addrs;  
}
```

一对多单向关联的 UML 图如下：



在单向关联的基础上，在“一”的一段用一个1来表示，而在“多”的一段用一个\*来表示。

当然，也有一些其他的表示方式，例如，在“一”的一段，还可以使用“0..1”来表示，在“多”的一段，可以用“n”、“0..\*”、“1..\*”、“0..n”、“1..n”等方法来表示。

类似的，双向一对多关联也可以这么表示。例如老师和学生之间的关联：一个老师可以管理多个学生，而一个学生被一个老师管理。示例代码如下：

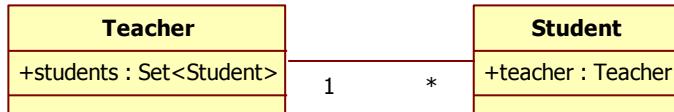
```

class Teacher{
    Set<Student> students;
}

class Student{
    Teacher teacher;
}

```

用 UML 图表示双向一对多关联，如下：



多对多关联，表示两个类中，都有一个属性，是另一个类的一个集合。因此，多对多关联没有单向，只有双向。例如，一个学生可以选多门课程，而每门课程都有多个学生选。学生和课程之间就形成了多对多的关联。示例代码如下：

```

class Course{
    Set<Student> students;
}

class Student{
    Set<Course> courses;
}

```

表示多对多关系的 UML 图如下：



## 2 常用设计模式

我们在介绍 I/O 框架的时候，曾经介绍过设计模式的概念。设计模式是值在设计面向对

象软件的过程中，用固定的套路去解决一些通用的问题。

在这一部分的内容中，我们将介绍单例模式、简单工厂模式。

## 2.1 单例模式

单例模式要解决的是这样一个问题：某一个类在整个程序中，只有唯一的一个对象。例如，在封建社会，“皇帝”这个类永远只能有一个唯一的对象。在实际代码中，我们也可能认为某个类在整个程序中只需要创建一个对象：例如唯一的一个数据库连接池对象，唯一的一个网络连接对象等等，而不希望程序在运行过程中创建很多新的连接而浪费资源影响性能。

那么我们怎么能做到这一点呢。例如下面的代码：

```
public class Singleton{  
    public Singleton(){}  
}
```

由于 Singleton 类提供了一个公开的构造方法，我们当然可以创建出任意多个 Singleton 类的对象，因此，我们要把构造方法改为私有。这样，这个方法只能在 Singleton 类内部调用。由于创建对象必须调用构造方法，这也意味着，不能在 Singleton 类外部创建 Singleton 对象。

代码形式如下：

```
public class Singleton{  
    private Singleton(){}  
}
```

同时我们可以在 Singleton 类中增加一个静态方法，由这个静态方法负责返回一个 Singleton 对象。这个静态方法由于在 Singleton 内部，因此可以调用构造方法；并且，这个静态方法可以通过 Singleton 的类名，在外部直接调用。如下：

```
public class Singleton{  
    private Singleton(){}  
    public static Singleton getInstance(){  
        return new Singleton();  
    }  
}
```

经过上面两步修改，我们可以通过 Singleton.getInstance() 获取对象了，但是还是没法做到全局唯一，因为每次对该方法的调用都会新创建一个对象。为此，我们为 Singleton 类增加一个静态的 instance 属性，并且修改 getInstance 方法如下：

```
public class Singleton{  
    private Singleton(){}  
    private static Singleton instance = null;  
    public static Singleton getInstance(){  
        if (instance == null) instance = new Singleton();  
        return instance;  
    }  
}
```

上面的代码中，当第一次调用 getInstance 方法时，由于静态属性 instance 为 null，会调用构造方法创建一个对象，并把这个对象返回。当后面再调用 getInstance 方法时，则会直

接返回 instance 对象。这样，在整个程序中，每次调用 getInstance 方法都能保证返回的是同一个对象，这就是基本的单例模式。

但是上面的单例模式的写法还有不完善的地方。注意 getInstance 方法：

```
01: public static Singleton getInstance() {  
02:     if (instance == null)  
03:         instance = new Singleton();  
04:     return instance;  
05: }
```

考虑多线程的情况。假设一开始，就有两个线程 t1 和 t2 同时调用 getInstance 方法。当 t1 执行到 02 行的时候，进行判断，假设现在 instance 依然为 null，因此判断为 true，t1 线程创建对象。在 t1 线程创建对象的过程中，CPU 时间片到期，t1 进入了可运行状态，t2 进入运行状态。

此时，由于依然没有创建对象，因此 instance 依然为 null，所以 t2 线程进行判断之后，创建一个 Singleton 对象，并返回。

这样，t1 和 t2 线程总共创建了两个 Singleton 对象，破坏了单例模式的含义！

我们上面描述的多线程问题，出现的概率非常低，但是，如果在一个高并发的服务器上，一旦出现这种问题，可能就会造成非常大的影响。因此，我们必须要解决这个问题。

怎么解决这个问题呢？非常简单，我们只要做一个简单的修改即可：

```
public class Singleton{  
    private Singleton(){}
    private static Singleton instance = new Singleton();
    public static Singleton getInstance(){  
        return instance;  
    }
}
```

在初始化 instance 属性的时候直接创建对象，这样，创建对象的过程在类加载的时候完成。这就解决了多线程的问题。

上面就是单例模式的介绍。在写单例模式的时候，有三个要点：1、私有的构造方法；2、静态的 instance 属性；3、静态的 getInstance()方法。掌握这三个要点，就能顺畅的写出单例模式的代码。

## 2.2 简单工厂模式

下面要介绍的是简单工厂模式。严格的说，简单工厂模式并不算是一种设计模式，因为它解决的问题非常基本，解决的方式也非常简单。事实上，我们可以把简单工厂模式当做常见的编程的写法。

这种模式主要解决的是创建对象的问题。我们之前创建对象，都使用 new 关键字。这种创建对象的方式当然可以，但假如是复杂的对象，可能就不那么简单。

例如，举一个生活中的例子来说。吃早餐，如果早餐我们打算吃一个煮鸡蛋，那可能早起 5 分钟，自己就可以创建一个煮鸡蛋对象出来。但如果早餐打算吃煎饼，如果自己创建的话，则需要和面、炸薄脆或者油条、准备面酱、切葱花香菜，最后，摊煎饼。很显然，这么多步骤都让自己来完成，是不现实的。为了解决这个问题，我们可以去煎饼摊买一个煎饼。在这个过程中，我们可以把煎饼摊当做是一个煎饼工厂，这个工厂有一个 createJianBing 方

法，能够被调用并返回一个煎饼对象。

示例代码如下：

```
public class JianBingFactory{
    //返回一个煎饼对象
    public static JianBing createJianBing(){
        return new JianBing();
    }
}

class JianBing{}
```

由工厂对象负责对象的创建，可以把创建一个复杂对象的代码从其他代码中分离出来，使得代码的功能更加单一，符合面向对象“各司其职”的原则。

### 3 三层体系结构介绍

接下来我们将介绍的是软件设计中非常重要也非常常用的内容：软件的三层体系结构。

#### 3.1 需求的变化

首先，我们从代码开始。假设我们是在公司中工作的程序员，每天的工作就是完成老板提出的需求，写出相应的代码。好了，下面，我们开始一天的工作！老板的需求正源源不断的到来！

需求 1：写一个程序，在屏幕上打印出 Hello World

这个代码非常简单，如下：

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

需求 2：从文件中读取一行文本，在屏幕上打印

我们将前面的代码修改如下：

```
import java.io.*;
public class HelloWorld {
    public static void main(String[] args) {
        BufferedReader br = null;
        try{
            FileReader fr = new FileReader("hello.txt");
            br = new BufferedReader(fr);
            String line = br.readLine();
            System.out.println(line);
        }catch(IOException e){
            e.printStackTrace();
        }finally{
            if (br != null){
                try{

```

```
        br.close();
    } catch (IOException e) { }
}
}
```

我们可以看到，代码已经变得复杂了很多。

需求3：把读到的文本转成大写，然后输出。

```
import java.io.*;
public class HelloWorld {
    public static void main(String[] args) {
        BufferedReader br = null;
        try{
            FileReader fr = new FileReader("hello.txt");
            br = new BufferedReader(fr);
            String line = br.readLine();
            line = line.toUpperCase();
            System.out.println(line);
        }catch(IOException e){
            e.printStackTrace();
        }finally{
            if (br != null){
                try{
                    br.close();
                }catch(IOException e){ }
            }
        }
    }
}
```

需求4：不从文件中读取，改为从网络中读取。代码如下：

```
import java.io.*;
import java.net.*;
public class HelloWorld {
    public static void main(String[] args) {
        Socket s = null;
        try{
            s = new Socket("127.0.0.1", 9000);
            BufferedReader br = new BufferedReader(
                new InputStreamReader(s.getInputStream()));
            String line = br.readLine();
            line = line.toUpperCase();
            System.out.println(line);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```

        } catch (IOException e) {
            e.printStackTrace();
        } finally{
            if (s != null){
                try{
                    s.close();
                } catch (IOException e) {}
            }
        }
    }
}

```

注意，在进行网络编程的时候，需要修改的代码已经相当多了。

需求 5：把读到的字符串全都转为倒置输出。

```

import java.io.*;
import java.net.*;
public class HelloWorld {
    public static void main(String[] args) {
        Socket s = null;
        try{
            s = new Socket("127.0.0.1", 9000);
            BufferedReader br = new BufferedReader(
                new InputStreamReader(s.getInputStream()));
            String line = br.readLine();
            StringBuffer sb = new StringBuffer(line);
            sb = sb.reverse();
            line = sb.toString();
            System.out.println(line);
        } catch (IOException e){
            e.printStackTrace();
        } finally{
            if (s != null){
                try{
                    s.close();
                } catch (IOException e) {}
            }
        }
    }
}

```

需求 6：把从网络上读取再改成从文件中读取。

**STOP！**你会发现，你的代码越来越混乱，随着需求的改变和不断的调整，我们的代码也不断进行调整。在刚刚修改代码的过程中，你有没有这样的想法：为啥要在原来的代码上

修改呢？还不如推倒重新写一遍呢……

恭喜你，在整个软件行业中，并不是只有你有这样的想法，很多大项目，在维护到后面的时候，很多程序员都在抱怨，觉得前人留下的代码非常的混乱而难于理解，不希望继续在前人的基础上工作，从头开始做项目可能更加容易一些。现在我们应该能够理解他们了，面对杂乱无章的代码，确实令人抓狂。而代码为什么凌乱呢？并不是一开始就乱的，而是在一遍又一遍修改的过程中，逐渐被“改乱”的。

当一个程序无法应对新的需求变化的时候，当改动原有程序比重新写一个程序需要花更多时间的时候，这时，我们当然会选择重新写程序，那么原有的程序就走向了死亡。在软件行业中，有大量的软件，花了长的时间进行开发之后，没有多久就因为种种的原因而必须被淘汰。这造成了软件的短命，也造成了软件的成本居高不下，无法充分利用已有的资源。

我们首先来分析一下，造成软件短命的原因是什么。有人会说，需求的变化是造成软件短命的主要原因。如果一个软件的需求没有变化或者变化很少，则需要对这个软件的改动也非常少，这样这个软件就能够一直使用下去，而不用进行大的修正。

但是，一方面，由于需求是由客户提出的，对于程序员来说，无法控制需求的变化。另一方面，随着技术的进步和时代的发展，技术和商业一定会有着非常巨大的变化。这些变化，都会使得人们对软件的要求有变化。例如，最早，用电话线上网的时候，腾讯公司的 qq 软件只有文字聊天功能；之后，随着带宽的增加和需求的变化，qq 软件增加了传送文件等功能；再之后，随着 ADSL 的兴起，如今的 qq 软件有了语音以及视频聊天、qq 游戏、qq 空间等等非常丰富的功能。这就是一个典型的需求不断变化的例子。

因此我们发现，需求的变化是必然的，程序员只能去适应需求的变化。于是，人们延长软件寿命的工作重点放在了“避免修改代码”上。是啊，如果能够在不修改代码的情况下，满足新的需求，那么就能在最大程度上避免软件被“越改越乱”了。于是，软件行业提出了一个设计原则：“开闭原则”。

开闭原则，指的是：在软件设计的过程中，要求软件能够做到：对扩展开放，对修改关闭。程序员可以通过在原有代码基础上添加新代码的方式，来满足新的需求，而不是修改原有的代码。如果一个软件能够做到这一点，那么软件的功能可以自由扩展，从而应对需求的变化；而原有的部分保持成熟和稳定。这样就能更好的保持程序结构的清晰易读。

为了实现开闭原则，有一些更加具体的要求。例如，修改关闭，就意味着原有的代码，在新的扩展以后的系统中继续能够使用，也就是代码的“可重用性”；而扩展开放，就意味着新的代码能够很方便的扩展原有的系统，而不影响原有的代码，这也就是代码的“可扩展性”；为了能够达到开闭原则，就要求模块之间的联系应当尽可能的弱，这样才能够保证方便的扩展新功能而不影响其他功能。同时，我们应该能够根据不同的新需求，扩展相应的软件模块，这也就有了软件“各司其职”的要求，即：软件的不同模块在功能上应该有明确的职责划分。也就是说，为了实现开闭原则，我们的软件应该具备以下特点：

- ✓ 可重用性
- ✓ 可扩展性
- ✓ 弱耦合性
- ✓ 各司其职

细心的读者可以看出，这些正是面向对象编程思想的特点和要求。由此可见，面向对象

的思想不是凭空产生的，而是软件行业为了应对需求的变化，为了能够更好的实现开闭原则，在编程思想领域的重大进步。

举个例子，我们都知道中国古代的四大发明，分别是造纸术，指南针，火药和活字印刷术。这其中，造纸术、指南针和火药都是从无到有的发明，唯有活字印刷术比较特别，北宋时期的毕昇只是将原有的印刷术加以改进，发明了活字印刷术。这难道不奇怪吗？我们往往认为，技术的发明者要比技术的改良者更值得纪念。就像我们记住了灯泡的发明者是爱迪生，却淡忘了节能环保型灯泡的发明者。可是针对印刷术，谁又能说清印刷术的发明者是谁呢？我们记住的只是那个改良者—毕昇。

这不难理解，传统的印刷术，印刷工人要在一整块木板上刻下所有的文字。一个错字就可能使得整个版作废。而活字印刷术高明之处在于，将每个字做成独立的“个体”，由多个“个体”组成词语，句子。这样，当文字发生改变的时候，只需要替换或增加有改动的文字即可，使得印刷工作符合了“开闭原则”。具体的说，每个字是独立的个体，这符合“各司其职”的要求；做好的字可以反复使用，这符合“可重用性”的要求；字与字之间彼此独立，互不影响，这符合“弱耦合性”的要求；整个版面可以在不影响其他字的情况下，随意添加新的文字，这又符合了“可扩展性”的要求。总之我们可以戏称，活字印刷术位列四大发明，体现了开闭原则的价值，闪烁着面向对象的光芒。

而为了更好的使用面向对象思想，为了使得我们的程序更加符合开闭原则，下面我们将为大家介绍非常典型的软件职责划分的方法：软件的三层体系结构。

## 3.2 三层体系结构介绍

在介绍三层体系结构之前，我们先分析一下之前提出的那些需求。我们可以把所有的需求分成三大类。

第一类：数据从哪儿来。之前提出的需求中，有的需求数据是从文件中读取，而有些情况，数据是从网络中读取。这一类需求的变化，是数据来源的变化，也可以认为是访问数据的方式的变化。

第二类，数据怎么处理。之前提出的需求中，数据获得之后，有些需求要求把数据全部转为了大写，有些需求要求把数据都转为倒置。这一类需求的变化，是对数据处理的变化，也可以认为是处理数据方式的变化。

第三类，数据怎么显示。在我们这些需求中，数据的显示比较简单，通过输出语句直接输出数据。但是我们可以想象，在以后的编程实践中，数据的现实会有各种各样的方式，例如通过图形界面显示，通过网页显示等等。

本着各司其职的思想，我们把软件设计成三个层次。这三个层次分别对应于三类需求。

首先是数据访问层。数据访问层是用来和数据打交道，具体的说，负责数据的增加，删除，修改和查询（当然，在我们的例子中，只涉及数据的查询）。而数据访问层的对象，被称之为数据访问对象（Data Access Object），简称 DAO。因此，数据访问层也被称为 DAO 层。DAO 层对应着第一类需求：数据从哪儿来。

其次是业务逻辑层。业务逻辑层，是专门用来处理数据的，这一层的对象被称之为业务对象（Business Object），简称 BO。而数据访问层也被称为 biz 层。

需要注意的是，业务逻辑层处理的数据，往往是从 DAO 层来的。也可以认为，DAO 负责获取数据，然后把数据传递给 biz 层，让 biz 层对数据进行处理。

用户提交的请求数据需要被接收，当数据处理完之后，结果数据需要显示给用户。而负责接收用户请求，并显示数据的是显示层，也被称为 view 层。而 view 层中的负责与用户交

互的对象称之为 View Object，简称 VO。

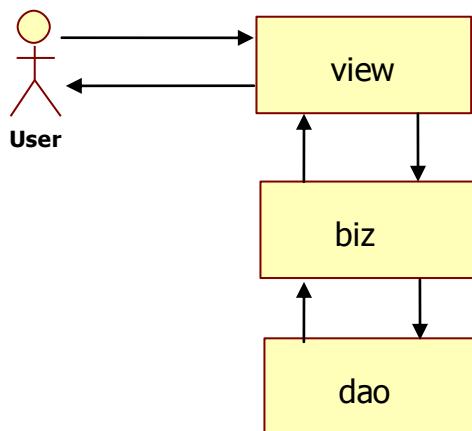
把软件分成三个层次之后，典型的情况如下：

view 层与用户交互的过程中，接受用户的一个指令。如果 view 层用图形界面显示数据，则这个指令有可能是图形界面上的一次点击；如果 view 层用网页显示数据，则这个指令有可能是网页上发送的一个 http 请求。

当 view 层获得一个用户的指令之后，会把这个指令交给 biz 层。在 view 层中，VO 会调用 biz 层中的方法，把用户跟 view 层交互时输入的一些数据，传递给 biz 层。然后，等 biz 层把数据处理完成之后，再把处理完成的数据返回给 view 层，让 view 层显示结果。

而 biz 层如果要处理数据，可能要先获得数据。为了获得数据，biz 层需要调用 dao 层的方法。当 dao 层把数据获取之后，dao 对象会将结果返回给 biz 层，biz 层才能根据数据进行下一步的处理。

因此，view、biz、dao 三个层次之间，是从上到下依次调用方法的关系。示意图如下：



我们以一个现实生活中的例子，来说明三层结构的概念。例如，一家汽车 4S 店，这种店为客户提供多种服务，例如购车、保养、修车等等。比如，客户的车坏了，需要修车。这个时候，当客户来到 4S 店时，与客户交互的往往是前台的接待员。这些接待员能够跟用户清晰、友好的沟通，获得客户的指令。我们可以把这些接待员当做就是 view 层中的 VO。

当客户告诉接待员，“我需要修车”，这就相当于客户发送了一个指令给了 VO。接待员知道用户需要修车之后，自己不会替用户完成修车这个过程。修车的指令，被接待员发送给了真正的汽车修理工。汽车哪部分有毛病，哪部分需要检修，这些数据都由接待员告诉修理工，由修理工真正来完成“修车”这个业务。在这个关系中，修理工就相当于 biz 层中的 BO，而上述流程就相当于 view 层的 VO 调用 biz 层的 BO 的方法。

当修理工修车时，有可能需要进行零部件的更换。例如，可能需要更换发动机，为此，修理工必须要获得一个新的发动机。往往备用的零配件是存放在仓库中，而仓库显然不能让每个人都随意进行访问，往往公司会安排一个专人做仓库管理员。仓库管理员的职责，就是负责向仓库存放物资，以及从仓库中取出物资。如果我们把零配件当做数据，那么仓库管理员的工作就是存取数据，扮演的就是 DAO 的角色。而修理工修车时，会根据需要向管理员要零配件，可以认为这就是 biz 层的 BO 对象在调用 dao 层的 DAO 对象的方法。

当仓库管理员找到相应的零配件时，会把这些物品交给修理工；而当修理工修车完毕之后，会通知前台的接待员；接待员最后，会把修好的车以及其他的一些信息（例如修车花了多少钱……）显示给客户。类比 Java 代码就是：dao 层返回到 biz 层，biz 层返回给 view 层，

view 层把运算的结果显示给客户。

上面我们介绍了软件的三层体系结构。那么，把软件设计成三层结构有什么好处呢？好处在于：当需求发生改变时，我们可以把改变局限在某个层次中，而不影响其他层次。例如，如果仓库的地点以及放置物品的位置发生改变的话，我们不需要对前台接待员和汽车修理工做过多的说明，只要让仓库保管员能够清楚应该怎么工作就可以了。同样的，如果某一个层次的需求发生变化，则我们只需要针对那个特定的层次，修改相应的代码，而不用改变其他层次的代码。

为了让层次与层次之间，实现弱耦合性，我们使用接口来定义三个不同的层次。

### 3.3 三层结构的 HelloWorld 程序

下面我们以 Hello World 程序为例，来看一下应当如何应用三层体系结构。

首先，应当定义三层的接口。先是 dao 层的接口。

```
package dao;
```

```
public interface Dao {  
    String getData();  
}
```

dao 层中所有的 DAO 对象都应该实现 Dao 接口。

然后，是 biz 层的接口。需要注意的是，因为 biz 层需要调用 dao 层的方法，因此，在 biz 对象中，需要维护一个 Dao 对象的引用。为此，所有 Biz 层接口的实现类都应当有一个 Dao 类型的属性，并且提供一个 setDao 方法。因此，在 Biz 接口中，我们定义了 setDao 方法。

```
package biz;  
import dao.Dao;  
  
public interface Biz {  
    void setDao(Dao dao);  
    String dealData();  
}
```

biz 层中所有的 BO 对象都应该实现 Biz 接口。

最后，是 view 层的接口。与之前的情况类似，view 层也应当有一个 Biz 类型的引用，用以调用 biz 层的方法。因此，在 View 接口中，我们定义了 setBiz 方法。

```
package view;  
import biz.Biz;  
  
public interface View {  
    void setBiz(Biz biz);  
    void showData();  
}
```

view 层中所有的 VO 对象都应该实现 View 接口。

定义完了三个接口之后，接下来应该给出的是接口的实现类。例如，我们首先给出 Dao

接口的实现类。假设我们希望从当前目录下的“test.txt”文件中获取数据，则可以给出一个实现类：FileDaoImpl 实现 Dao 接口，代码如下：

```
package dao;
import java.io.*;
public class FileDaoImpl implements Dao {

    public String getData() {
        String data = null;
        BufferedReader br = null;
        try{
            br = new BufferedReader(new FileReader("test.txt"));
            data = br.readLine();
        }catch(IOException e){
            e.printStackTrace();
        }
        finally{
            if (br != null){
                try{
                    br.close();
                }catch(IOException e){e.printStackTrace();}
            }
        }
        return data;
    }

}
```

下面是 Biz 接口的实现。假设我们要实现把所有数据都转成大写的逻辑，则可以给出一个UpperCaseBizImpl 的实现类。代码如下：

```
package biz;
import dao.Dao;

public class UpperCaseBizImpl implements Biz {
    private Dao dao;

    public String dealData() {
        String data = dao.getData();
        if (data != null){
            data = data.toUpperCase();
        }
        return data;
    }

    public void setDao(Dao dao) {
```

```
        this.dao = dao;
    }

}
```

需要注意的是，我们为 Biz 的实现类增加了 Dao 类型的属性，原因是在 dealData 方法中，我们使用了 Dao 接口中定义的方法。

最后，View 接口的实现比较简单。我们给出 TextViewImpl 的实现代码：

```
package view;

import biz.Biz;

public class TextViewImpl implements View {
    private Biz biz;
    public void setBiz(Biz biz) {
        this.biz = biz;
    }

    public void showData() {
        String data = biz.dealData();
        System.out.println(data);
    }
}
```

当把三个接口的实现类完成之后，接下来，就可以写主方法，设置类之间的关联关系。  
代码如下：

```
package test;

import dao.*;
import biz.*;
import view.*;

public class TestMain {
    public static void main(String[] args) {
        //创建对象并调用 set 方法进行组装
        Dao dao = new FileDaoImpl();
        Biz biz = new UpperCaseBizImpl();
        biz.setDao(dao);
        View view = new TextViewImpl();
        view.setBiz(biz);

        view.showData();
    }
}
```

完成三层结构之后，再有新的需求到来时，我们就可以更好的应对。例如，现在有新的需求，需要把所有的字符串转为小写。

由于这个需求是“数据如何处理”，属于 biz 层的需求，我们无需改动原有的 biz 层对象，而可以为 Biz 接口扩展出一个新的实现类 LowerCaseBizImpl，代码如下：

```
package biz;
import dao.Dao;
public class LowerCaseBizImpl implements Biz {
    private Dao dao;

    public String dealData() {
        String data = dao.getData();
        if (data != null) {
            data = data.toLowerCase();
        }
        return data;
    }

    public void setDao(Dao dao) {
        this.dao = dao;
    }

}
```

我们只是实现 Biz 接口，很轻松的就完成了对代码的扩展，而没有对原有代码进行任何的改动。这就达到了开闭原则中，“扩展开放”的要求。

之后，我们需要修改的代码只有这样一条：

```
public static void main(String[] args) {
    //创建对象并调用 set 方法进行组装
    Dao dao = new FileDaoImpl();
    Biz biz = new LowerCaseBizImpl();
    biz.setDao(dao);
    View view = new TextViewImpl();
    view.setBiz(biz);

    view.showData();
}
```

这样，与“修改关闭”的要求，也已经非常接近了。

## 3.4 简单工厂模式的应用

然而，接近了“修改关闭”，但是依然会在需求变化的时候，修改原有的代码。能不能完全不修改代码呢？

首先，修改代码的原因，与创建对象相关。为此，我们可以利用简单工厂模式，把所有

床架对象的过程，都挪到一个简单工厂中。工厂代码如下：

```
package factory;
import dao.*;
import biz.*;
import view.*;
public class SimpleFactory {
    public SimpleFactory() {}

    public Dao createDao() {
        return new FileDaoImpl();
    }

    public Biz createBiz() {
        return new UpperCaseBizImpl();
    }

    public View createView() {
        return new TextViewImpl();
    }
}
```

这样，TestMain 程序就可以改成：

```
package test;

import dao.*;
import biz.*;
import view.*;
import factory.SimpleFactory;

public class TestMain {
    public static void main(String[] args) {
        SimpleFactory factory = new SimpleFactory();
        Dao dao = factory.createDao();
        Biz biz = factory.createBiz();
        biz.setDao(dao);
        View view = factory.createView();
        view.setBiz(biz);

        view.showData();
    }
}
```

这样，当实现类改变的时候，主方法不需要改变，因为创建对象的代码都使用 factory

来完成了。

但是，这样如果需求改变的话，还是要该 factory 的代码。有没有办法不改代码，就能能够在某个层次用一个实现类替换另一个实现类呢？

我们知道，利用反射可以灵活的创建对象。使用 Class.forName(String className)，通过一个字符串获得类对象。然后，通过类对象，可以创建一个相应类型的对象。简单的说，可以通过给定一个字符串，就获得一个该类型的对象。

而字符串，既可以写在代码中，同样可以从别的途径获得。例如，可以从一个配置文件中获得。因此，修改配置文件，就可以让 Class.forName 获得不同的字符串，从而创建出不同类型的对象来。换句话说，我们可以通过配置文件+反射的方式，来完成对象的创建。这样，当我们需要修改实现类的时候，只需要修改配置文件，而完全不需要修改代码。

首先，我们写一个方法，这个方法通过字符串来创建一个对象。

```
private Object createObject(String name){  
    Object result = null;  
    try {  
        Class c = Class.forName(name);  
        result = c.newInstance();  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
    return result;  
}
```

然后，我们在 SimpleFactory 的构造方法中，读某一个配置文件，然后把配置文件的信息保存起来。

我们创建一个配置文件 conf.props，并把配置文件设计成下面这种格式：

```
view=view.TextViewImpl  
biz=biz.UpperCaseBizImpl  
dao=dao.FileDaoImpl
```

用“=”分开两个部分，左边的部分是 view、biz、dao，分别表示三层；而右面表示的是该层我们采用的实现类的名字。注意，给出名字的时候，给出的是“包名 + 类名”的全限定名。我们可以看到，这样的配置文件非常类似于“键值对”的结构，等号左边为键，等号右边为值。并且，键和值都是字符串。

在 Java 中，解析这种“键值对”形式的配置文件，有一个非常方便的类：java.util.Properties。这个类是 Hashtable 的子类，他是一个特殊的 Map。特殊的地方有两种：

1) 这个类的键和值都是字符串。因此，这个类提供了一个方法：getProperty。这个方法接受一个字符串类型的参数，表示“键”；返回值也是字符串，对应的是值。

2) 我们可以通过这个类的 load 方法，来读入配置文件。load 方法可以接受一个 InputStream 参数，如果要读文件的话，那创建一个 FileInputStream 作为 load 方法参数即可。load 方法会自动解析输入流，例如我们上面的 conf.props，就会被自动解析出三个键值对放入 Properties 中，键分别为“view”、“biz”、“dao”，对应的值为“view.TextViewImpl”、“biz.UpperCaseBizImpl”、“dao.FileDaoImpl”。

为此，我们可以为 SimpleFactory 增加一个 Properties 的属性，并且在构造方法中，利用 load 方法，读入 conf.props。完整的 SimpleFactory 代码如下：

```
package factory;
import dao.*;
import biz.*;
import view.*;
import java.util.Properties;
import java.io.*;
public class SimpleFactory {
    private Properties props;
    public SimpleFactory() {
        props = new Properties();
        InputStream is = null;
        try{
            is = new FileInputStream("conf.props");
            props.load(is);
        }catch(IOException e){
            e.printStackTrace();
        }
        finally{
            if (is!=null) {
                try{
                    is.close();
                }
                catch(Exception e){
                    e.printStackTrace();
                }
            }
        }
    }

    public Dao createDao() {
        String className = props.getProperty("dao");
        Dao dao = (Dao) createObject(className);
        return dao;
    }

    public Biz createBiz() {
        String className = props.getProperty("biz");
        Biz biz = (Biz) createObject(className);
        return biz;
    }
}
```

```
public View createView() {
    String className = props.getProperty("view");
    View view = (View) createObject(className);
    return view;
}

private Object createObject(String name) {
    Object result = null;
    try {
        Class c = Class.forName(name);
        result = c.newInstance();
    } catch (Exception e) {
        e.printStackTrace();
    }
    return result;
}
}
```

配置 conf.props 文件如下：

```
view=view.TextViewImpl
biz=biz.UpperCaseBizImpl
dao=dao.FileDaoImpl
```

test.txt 文件内容如下：

```
Hello World
```

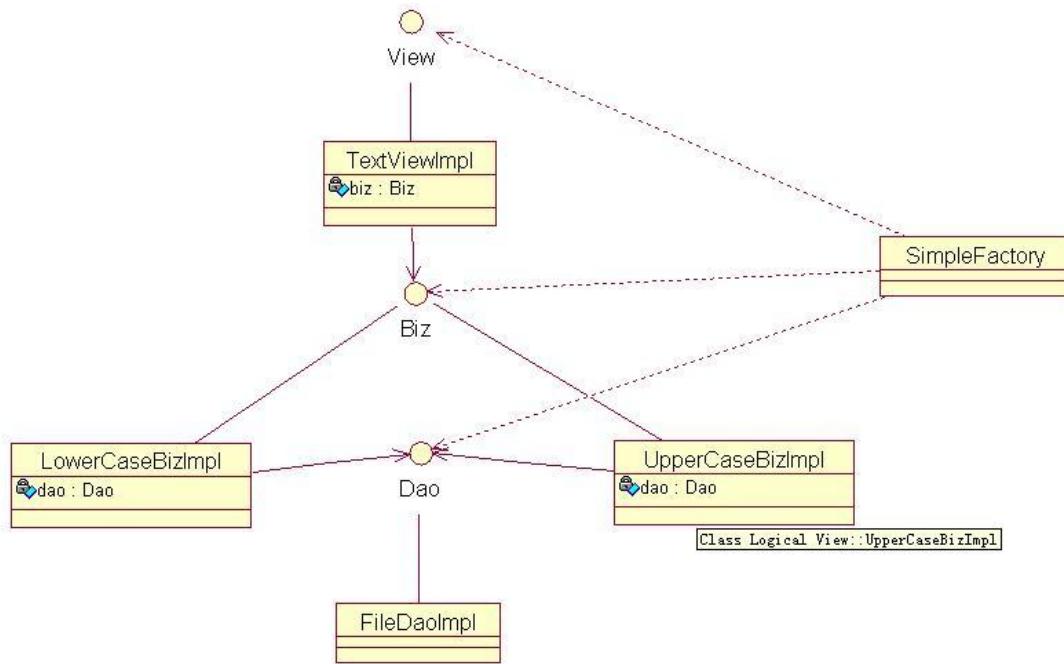
当上述简单工厂完成之后，运行结果如下：

```
HELLO WORLD
```

而当有新需求，希望把所有字符改成小写，只需要修改配置文件：

```
biz=bizLowerCaseBizImpl
即可。
```

整个应用的类图如下：



读者也可以在该代码上，尝试着完成其他的一些需求。例如：文字从网络中获取（改变的是数据的获取方式，需要为 Dao 接口添加一个实现类，替换掉 FileDaolmpl）；将文字倒置处理（改变的是数据的处理方式，需要为 Biz 接口添加一个实现类，替换 LowerCaseBizImpl）。

也许读者会迷惑，写了这么多的接口和类，最终只是完成了最初级的 HelloWorld 输出的功能，是不是有点小题大做了？诚然，这个程序的功能并不复杂，但这种程序结构，使得在需求变化的时候，我们总是能够有针对性的扩展出某个接口的实现类，而不需要改动任何原有代码，从而保证了开闭原则的要求，也使得我们的代码能够“健康长寿”。我们可以认为，这才是一个真正贯彻了面向对象思想，发挥了面向对象优势的 HelloWorld！