1 Java 入门

Java 是一门很优秀的编程语言，具有面向对象、与平台无关、安全、稳定和多线程等特点，是目前软件设计中极为健壮的编程语言。Java 不仅可以用来开发大型的应用程序，而且特别适合于 Internet 的应用开发。Java 确实具备了“一次编写，处处运行”的特点，Java 已成为网络时代最重要的编程语言之一。

1.1 Java 的诞生

在 1990 年，Sun 公司成立了一个由 James Gosling 领导的软件设计团队，他们合作的项目称为“绿色计划”。他们认为计算机技术发展的一个趋势是数字家电之间的通讯。James 开始负责为设备和用户之间的交流创建一种能够实现网络交互的语言。随着大量的时间和金钱投入到“绿色计划”，他们创建了一种语言。这种语言一开始被叫做“Oak”，这个名字得自于 Gosling 想名字时看到了窗外的一棵橡树，后来被改为了“Java”。

Java 的快速发展得利于 Internet 和 Web 的出现，到了 2000 年，Java 已经成为世界上最流行的电脑语言。绿色小组当初设计 Java 是为了面向数字家庭，支持各种家电设备。他们没有想到的是，Java 支持的计算模式，实际上就是互联网的模式 。

Java 的重要历史事件：

时间	事件
1995 年 5 月 23 日	Java 语言诞生
1996 年 1 月	第一个 JDK-JDK1.0 诞生
1996 年 4 月	10 个最重要的操作系统供应商申明将在其产品中嵌入 JAVA 技术
1997 年 2 月 18 日	JDK1.1 发布
1998 年 12 月 8 日	JAVA2 企业平台 J2EE 发布
1999 年 6 月	SUM 公司发布 Java 的三个版本： 标准版（J2SE）、企业版（J2EE）和微型版（J2ME）
2004 年 9 月 30 日	J2SE1.5 发布，成为 Java 语言发展史上的又一里程碑。为了表示该版本的重要性，J2SE1.5 更名为 Java SE 5.0

2006 年 12 月	SUN 公司发布 JDK6.0

目前主流的 JDK 是 Sun 公司发布的 JDK，除了 Sun 之外，还有很多公司和组织都开发了自己的 JDK，例如 IBM 公司开发的 JDK，BEA 公司的 Jrocket，还有 GNU 组织开发的 JDK 等等。

印度尼西亚有一个重要的盛产咖啡的岛屿，中文名叫爪哇，开发人员为这种新的语言起名为 Java，其寓意是为世人端上一杯香浓的热咖啡。现在就让我们来一起品尝吧。

1.2 Java 编程语言是什么

Java 的内容很丰富，实现的功能也很多，我们从以下几个角度来描述它。

一种计算机编程语言
一种软件开发平台
一种软件运行平台
一种软件部署环境
句法与 C++相似，语义与 Small Talk 相似
用来开发 applets，又用来开发 applications

1、Java 是一种计算机编程语言

语言

我们说的普通话、英语都是语言，语言是一种交流的工具，语言具有创造性和结构性，并且代表一定的意义。比如我说下课了，大家都明白什么意思，证明这个语句的意思表达清楚了，正规的语言在交流上是不能有歧义的。

计算机编程

计算机编程就是：把程序员的要求和设想，按照能够让计算机看得懂的规则和约定，编写出来的过程，就是编程。编程的结果就是一些计算机能够看懂并能够执行和处理的东西， 我们把它叫做软件或者程序。事实上，程序就是我们对计算机发出的命令集（指令集）。

Java 是一种计算机编程语言

首先，Java 是一种语言，也就是 Java 是用来交流的，那么用来谁和谁交流呢？很明显就是程序员和计算机交流，换句话说把我们的要求和设想用 Java 语言表达出来，那么计算机能看懂，就能够按照我们要求运行，而这个过程就是我们所说的使用 Java 编程，所以我们讲 Java 是一种计算机编程语言。为了让计算机看懂，Java 会有一系列的规则和约定，这些就是 Java 的语法。

2、Java 是一种软件开发平台

什么是软件开发

可以简单地理解为：编程的结果是软件或者程序，而编程的过程就是软件开发。软件开发的基本步骤包括：需求分析、概要设计、详细设计、编码、测试、维护等阶段。

需求分析：这里指的需求不仅仅是用户需求，应该是开发中遇到的所有的需求。比如，你首

先要知道做这个项目是为了解决什么问题；测试案例中应该输入什么数据…… 为了清楚地知道这些需求，你经常要和客户、项目经理以及项目伙伴调查研究，这就是需求分析。

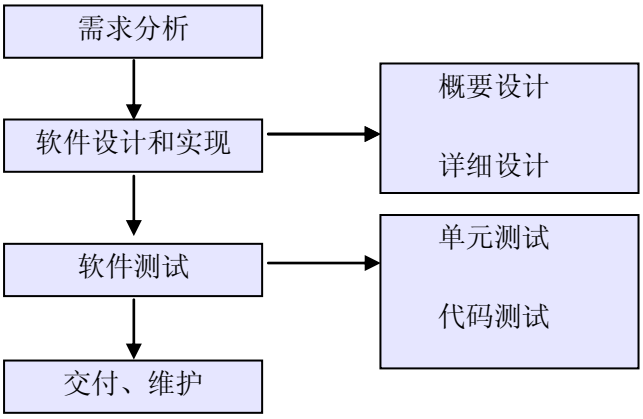
概要设计、详细设计：根据软件系统需求完成对系统的设计，确定强壮的系统架构，设计模块层次、用户界面和数据库表结构。

编码：开发代码，完成设计的具体实现。

测试：利用测试工具按照测试方案和业务流程对产品进行功能和性能测试，对测试方案可能出现的问题进行分析和评估，并修改代码。

维护：根据用户需求的变化或硬件环境的变化，对应用程序进行部分或全部的修改。

用以下的流程图来表达这个过程：



什么是开发平台

在软件开发的过程中，我们需要很多的工具来辅助我们的工作，不可能什么都从头自己做。我们把编程的环境和相应的辅助工具统称为开发环境，开发平台就是用来提供这个开发环境的。车床工人需要一个车床才能工作一样。

Java 是一种开发平台

Java 不单纯是一个编程的语言，它自身提供了一系列开发 Java 所需要的环境和工具，来进行编译、解释、文档生成、打包等，比如：javac.exe、java.exe 等等，这些我们后面会讲到，所以我们讲 Java 是一个开发平台。

3、Java 是一种软件运行平台

什么是软件的运行平台

如同人类需要阳光、空气、水和食物才能正常存活一样，软件最终要能够运行，也需要一系列的外部环境，来为软件的运行提供支持，而提供这些支持的就是运行平台。

Java 是一种软件运行平台

Java 本身提供 Java 软件所需要的运行环境，Java 应用可运行在安装了 JRE(Java Runtime Environment)的机器上，所以我们说 Java 是一个运行平台。

JRE：Java Runtime Environment，Java 运行环境。

4、Java 是一种软件部署环境

什么是软件的部署

简单地讲，部署就是安装，就是把软件放置到相应的地方，并且进行相应的配置（一般称作部署描述）让软件能够正常运行起来。

Java 是一种软件部署环境

Java 本身是一个开发的平台，开发后的 Java 程序也是运行在 Java 平台上的。也就是说，开发后的 Java 程序也是部署在 Java 平台上的，这个尤其在后面学习 JEE（Java 的企业版）的时候，体现更为明显。

1.3 Java 能干什么

Java 能做的事情很多，涉及到编程领域的各个方面。

桌面级应用：尤其是需要跨平台的桌面级应用程序。

桌面级应用：简单的说就是主要功能都在我们本机上运行的程序，比如 word、excel 等运行在本机上的应用就属于桌面应用。

企业级应用

企业级应用：简单的说就是大规模的应用，一般使用人数较多，数据量较大， 对系统的稳定性、安全性、可扩展性和可装配性等都有比较高的要求。举例说明一下：

组件化：企业级应用通常比较复杂，组件化能够更好对业务进行建模，提高系统的扩展性和维护性，做到组件复用。

分布式：企业组织机构复杂，同一地有多个分部，或者跨省，甚至跨国，COABA， RMI，Web Services 是 JavaEE 中支持的分布式访问技术，还有分布式的连接，如系统需要接入多个数据源，可以用 JNDI 来透明实现。

事务管理：为了保证数据的安全操作、安全访问，事务是不可缺少的，事实上只要操作数据库，就离不开事务管理。

消息管理：通过消息来实现异步触发从而降低系统耦合性，提高系统吞吐量。一个电子商务网站也可以使用消息来进行异步发邮件，但在企业级应用当中，根据实际需求还可以演变成更多复杂的应用，JEE 提供 JMS 实现消息管理。

安全性：企业级应用的数据都更为敏感（比如公司的销售数据、财务数据），需要为此提供严格的安全性保护，企业级组织的复杂性、接入访问的多样性增加了安全策略实施的难度，JAAS 为此提供了一整套的安全策略，方便企业级应用以安全、一致、便捷的方式实现安全机

制。

目前企业级应用是 Java 应用最广泛的一个领域，几乎一枝独秀。包括各种行业应用、企业信息化、电子政务等，包括办公自动化 OA，人力资源 HR，客户关系管理 CRM， 企业资源计划 ERP、供应链管理 SCM、企业设备管理系统 EAM、产品生命 周期管理 PLM、面向服务体系架构 SOA、商业智能 BI、项目管理 PM、营销管理、流程管理 WorkFlow、财务管理……等等几乎所有你能想到的应用。

嵌入式设备及消费类电子设备

包括无线手持设备、智能卡、通信终端、医疗设备、信息家电（如数字机顶盒、 电冰箱）、汽车导航系统等都是近年以来热门的 Java 应用领域，尤其是手机上的 Java 应用程序和 Java 游戏，更是普及。

嵌入式装置答题上区分为两种：一种是运算功能有限、电力供应也有限的嵌入式装置，例如：PDA、手机；另外一种则是运算能力相对较佳、并且电力供应上相对比较充足的嵌入式装置，比如：冷气机、电冰箱、电视机顶盒。

除了上面提到的，Java 还有很多功能：如进行数学运算、显示图形界面、进行网络操作、进行数据库操作、进行文件的操作等等。

1.4 Java 有什么

Java 体系比较庞杂，功能繁多，这也导致很多人在自学 Java 的时候总是感觉无法建立全面的知识体系，无法从整体上把握 Java 的原因。在这里我们先简单了解一下 Java 的版本。

Java 分成三种版本，分别是： Java 标准版(JSE)、Java 微缩版(JME)和 Java 企业版(JEE)。

每一种版本都有自己的功能和应用方向。

Java 标准版：JSE(Java Standard Edition）

JSE 是 Sun 公司针对桌面开发以及低端商务计算解决方案而开发的版本，例如：我们平常熟悉的 Application 桌面应用程序。这个版本是个基础，它也是我们平常开发和使用最多的技术，Java 的主要的技术将在这个版本中体现。本书主要讲的就是 JSE。

Java 微缩版：JME(Java Micro Edition）

JME 是对标准版 JSE 进行功能缩减后的版本，于 1999 年 6 月由 Sun Microsystems 第一次推向 Java 团体。它是一项能更好满足 Java 开发人员不同需求的广泛倡议的一部分。Sun Microsystems 将 JME 定义为“一种以广泛的消费性产品为目标的高度优化的 Java 运行时环境”，应用范围包括掌上电脑、移动电话、可视电话、数字机顶盒和汽车导航系统及其他无线设备等。

JME 是致力于消费产品和嵌入式设备的开发人员的最佳选择。尽管早期人们对它看好而且 Java 开发人员团体中的热衷人士也不少，然而 JME 最近才开始从其影响更大的同属产品 JEE 和

JSE 的阴影中走出其不成熟期。

JME 在开发面向内存有限的移动终端(例如掌上电脑、移动电话)的应用时，显得尤其实用。因为它是建立在操作系统之上的，使得应用的开发无须考虑太多特殊的硬件配置类型或操作系统。因此，开发商也无须为不同的终端建立特殊的应用，制造商也只需要简单地使它们的 操作平台可以支持 JME 便可。

Java 企业版：JEE（Java Enterprise Edition）

JEE 是一种利用 Java 平台来简化企业解决方案的开发、部署和管理相关的复杂问题的体系结构。JEE 技术的基础就是核心 Java 平台或 Java 平台的标准版，JEE 不仅巩固了标准版中的许多优点，例如"一次编写、处处运行"的特性、方便存取数据库的 JDBC API、CORBA 技术以及能够在 Internet 应用中保护数据的安全模式等等， 同时还提供了对 EJB(Enterprise Java Beans)、Java Servlets API、JSP(Java Server Pages) 以及 XML 技术的全面支持。其最终目的就是成为一个能够使企业开发者大幅缩短投放市场时 间的体系结构。

JEE 体系结构提供中间层集成框架来满足无需太多费用而又需要高可用性、高可靠性以及可扩展性的应用的需求。通过提供统一的开发平台，JEE 降低了开发多层应用的费用和复 杂性，同时提供对现有应用程序集成强有力支持，完全支持 Enterprise Java Beans，有良好的向导支持打包和部署应用，添加了目录支持，增强了安全机制，提高了性能。

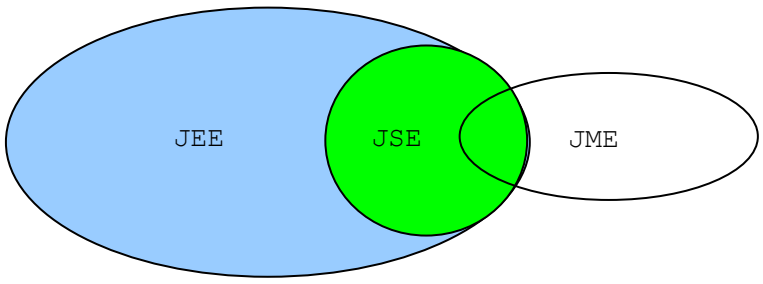
JEE 是对标准版进行功能扩展，提供一系列功能，用来解决进行企业应用开发中所面临的复杂的问题。具体的我们会放到后面 JEE 的课程去讲。

三个版本之间的关系：

JEE 几乎完全包含 JSE 的功能，然后在 JSE 的基础上添加了很多新的功能。

JME 主要是 JSE 的部分功能子集，然后再加上一部分额外添加的功能。

如下图所示：



Java 的 API 类库之中有一组所谓的核心类(CoreClass，即 java.*)，在核心类之外还有所谓的扩充类(Extended Class，即 javax.*)。根据对这两种类的支持程度，进而区分出几种不同的 Java 版本。

我们必须以 Java Standard Edition(JSE)作为基准，这个版本做了所有 Java 标准规格之中所定义的核心类，也支持所有的 Java 基本类。JSE 定位在客户端程序的应用上。

从 JSE 往外延伸，其外面为 Java Enterprise Edition(JEE)，此版本除了支持所有的标准核心类外，而且还增加了许多支持企业内部使用的扩充类，支持 Servlet / JSP 的 javax.servlet.*类、支持 Enterprise Java Bean 的 javax.ejb.* 类。当然，JEE 必定支 持所有的 Java 基本类。JEE 定位在服务器端(server-side)程序的应用上。

从 JSE 向内看，是 Java Micro Edition(JME)，它所支持的只有核心类的子集合，在 JME CLDC 的规格之中，只支持 java.lang.* 、java.io.* 、以及 java.util.* 这些类。此版本 也增加了一些支持"微小装置"的扩充类，如 javax.microedition.io.* 类。然而，此版本并不支持所有的 Java 基本类，就 标 准的 JMECLDC， 也就是能在 PalmOS 上执行的 KVM(KVirtualMachine)来说，它就不支持属于浮点数(float、double)的 Java 基本类。JME 定位在嵌入式系统的应用上。

最里层，还有一个 Java 的 Smart Card 版本，原本在 Java 的文件之中并没有这样定义，但是将它画在 JME 内部是很合理的。因为 SmartCard 版本只支持 java.lang.* 这个核心类，比起 JME 所支持的核心类更少，但它也有属于自己的扩充类，如 javacard.*、javacardx.* 这些类。SmartCard 版本只支持 Boolean 与 Byte 这两种 Java 基本类，此版本定位在 SmartCard 的应用上。

1.5 Java 的特点

简单地说，Java 具有如下特点：简单的、面向对象的、平台无关的、多线程的、安全的、高效的、健壮的、动态的等特点。

简单的

简单是指 Java 既易学又好用。不要将简单误解为这门编程语言很干瘪。你可能很赞同这样的观点：英语要比阿拉伯语言容易学。但这并不意味着英语就不能表达丰富的内容和深刻的思想，许多荣获诺贝尔文学奖的作品都是用英文写的。如果你学习过 C++，你会感觉 Java 很眼熟，因为 Java 中许多基本语句的语法和 C++一样，像常用的循环语句、控制语句等和 C++几乎一样，但不要误解为 Java 是 C++的增强版，Java 和 C++是两种完全不同的编程语言，它们各有各的优势，将会长期并存下去，Java 和 C++已成为软件开发者应当掌握的编程语言。如果从语言的简单性方面看，Java 要比 C++简单，C++中有许多容易混淆的概念，或者被 Java 弃之不用了，或者以一种更清楚更容易理解的方式实现，例如，Java 不再有指针的概念。

面向对象的

面向对象是指以对象为基本粒度，其下包含属性和方法。对象的说明用属性表达，而通过使用方法来操作这个对象。面向对象技术使得应用程序的开发变得简单易用，节省代码。基于对象的编程更符合人的思维模式，使人们更容易编写程序。Java 是一种面向对象的语言，也继承了面向对象的诸多好处，如代码扩展、代码复用等。我们将在以后的章节中详细地讨论类、对象等概念。

平台无关的

与平台无关是 Java 最大的优势。其他语言编写的程序面临的一个主要问题是：操作系统的变化，处理器升级以及核心系统资源的变化，都可能导致程序出现错误或无法运行。而用 Java 写的程序不用修改就可在不同的软硬件平台上运行。这样就能实现同样的程序既可以在

Windows 下运行，到了 Unix 或者 Linux 环境不用修改就直接可以运行了。Java 主要靠 Java 虚拟机（JVM）实现平台无关性。平台无关性就是一次编写，到处运行：Write Once, Run Anywhere。

多线程的

Java 实现了内置对多线程的支持。多线程允许同时完成多个任务。实际上多线程使人产生多个任务在同时执行的错觉，因为，目前的计算机的处理器在同一时刻只能执行一个线程，但处理器可以在不同的线程之间快速地切换，由于处理器速度非常快，远远超过了人接收信息的速度，所以给人的感觉好像多个任务在同时执行。C++没有内置的多线程机制，因此必须调用操作系统的多线程功能来进行多线程程序的设计。

安全的

安全性可以分为四个层面，即编译、类装载、字节码校验、沙箱机制。

高效的

高级语言程序必须转换为机器语言程序才能执行，但不同的计算机系统所使用的机器语言不同。Java 为了实现"一次编译，随处运行"的目标，Java 的源程序在编译时，并不直接编译成特定的机器语言程序，而是编译成与系统无关的**字节码**，由 Java 虚拟机（JVM ）来执行。当 JVM 解释执行 Java 程序时，Java 实时编译器（Just-In-Time，JIT）会将字节码译成目标平台对应的机器语言的指令代码。

早先的许多尝试解决跨平台的方案对性能要求都很高。其他解释执行的语言系统，如 BASIC、PERL 都有无法克服的性能缺陷。然而，Java 却可以在非常低档的 CPU 上顺畅运行，这是因为 Java 字节码是经过精心设计的，能够直接使用 JIT 编译技术将字节码转换成高性能的本机代码。事实上，Java 的运行速度随着 JIT 编译器技术的发展已接近于 C++。

健壮的

Java 是健壮的语言。为了更好地理解 Java 的健壮性，先讨论一下在传统编程环境下程序设计失败的主要原因：内存管理错误和误操作引起的异常或运行时异常。

在传统的编程环境下，内存管理是一项困难、乏味的工作。例如，在 C 或 C++ 中，必须手工分配、释放所有的动态内存。如果忘记释放原来分配的内存，或是释放了其他程序正在使用的内存时，就会出错。在传统的编程环境下，异常情况可能经常由“被零除”、“Null 指针操作”、“文件未找到”等原因引起，必须用既繁琐又难理解的一大堆指令来进行管理。

对此，Java 通过自行管理内存分配和释放，从根本上消除了有关内存的问题。Java 提供垃圾收集器，可自动收集闲置对象占用的内存。通过提供面向对象的异常处理机制来解决异常处理的问题。通过类型检查、Null 指针检测、数组边界检测等方法，在开发早期发现程序错误。

动态的

Java 语言具有动态特性。Java 动态特性是其面向对象设计方法的扩展，允许程序动态地装入运行过程中所需的类，这是 C++进行面向对象程序设计所无法实现的。C++程序设计过程中，每当在类中增加一个实例变量或一种成员函数后，引用该类的所有子类都必须重新编译，否则将导致程序崩溃。

1.6 构建 Java 开发环境

我们先来学习构建 Java 开发环境，让大家对 Java 有更实际的了解。下面以 JDK6.0 在 Windows XP 上的安装配置为例来讲述：

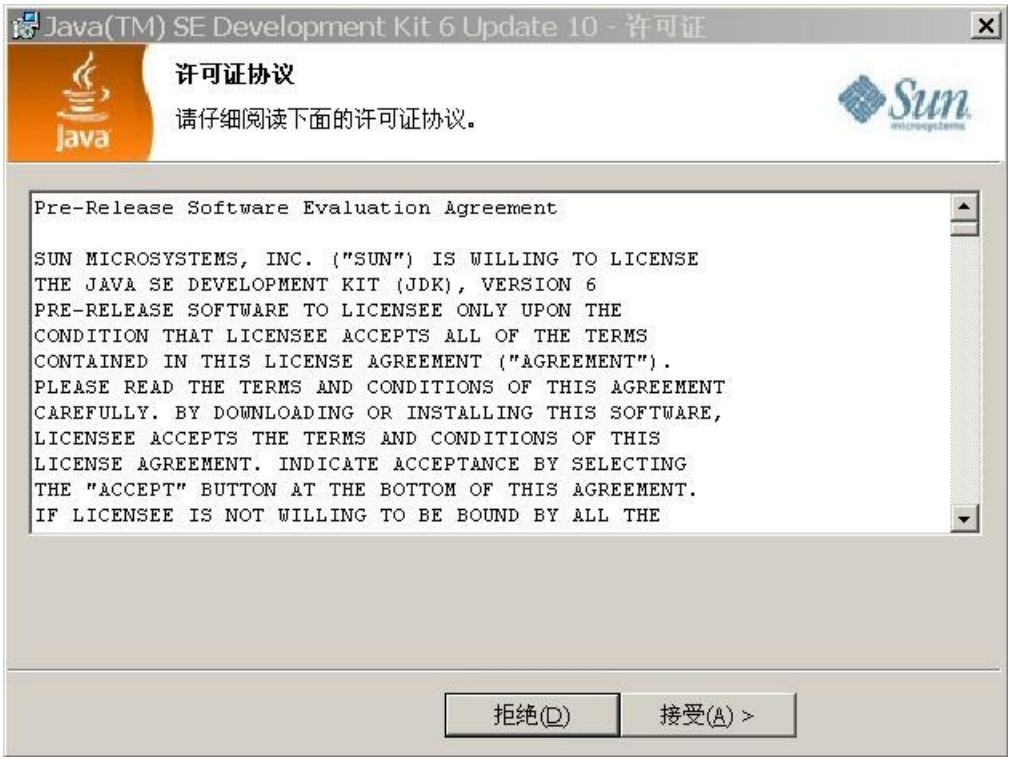
1.6.1 第一步：下载 JDK

从 SUN 网站下载 JDK6 或以上版本，地址是 http://java.sun.com/javaee/downloads/index.jsp，这里以 jdk-6u10-rc2-bin-b32-windows-i586-p-12_sep_2008.exe 版为例，如下图：

1.6.2 第二步：安装 JDK

（1）双击 jdk-6u10-rc2-bin-b32-windows-i586-p-12_sep_2008.exe 文件，出现如下安装界面：

（2）然后出现下面的界面



（3）点击“接受”按钮，然后出现下列界面

（4）点击界面上的“更改”按钮，可以设置需要安装到的路径，然后点击下一步，会进行 JDK 的安装，请耐心等待。

（5）直到出现 JRE 的安装，如下图：

（6）可以选择更改，然后在弹出的界面选择安装 JRE 的路径，然后点击确定。也可以不更改安装路径，点击下一步，出现下面的界面。这里介绍了 Java 语言开发的一个 Office 软件，和微软的 Office 各有千秋，因为是 Java 开发的，大家可以支持一下，最主要的它是免费的。

（7）直到出现如下界面，表示安装完成：

（8）不想查看产品注册信息的话，就直接点击完成按钮，安装 JDK 就完成了。

（9）安装完成过后，JDK 文件夹如下图：

C:\Program Files\Java\jdk1.6.0_10：是 JDK 的安装路径

bin：是 binary 的简写，存放的是 Java 的各种可执行文件，常用的命令有编译器 javac.exe、解释器 java.exe 。

include：需要引入的一些头文件，主要是 c 和 c++的，JDK 本身是通过 C 和 C++实现的。

jre：Java 运行环境。

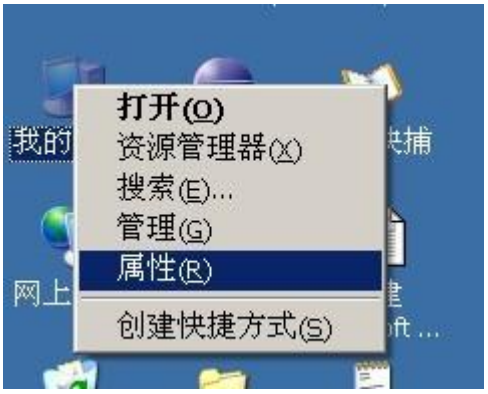
lib：是 library 的简写，JDK 所需要的一些资源文件和资源包。

C:\Program Files\Sun\JavaDB ：JDK6 新加入的 Apache 的 Derby 数据库，支持 JDBC4.0 的规范。

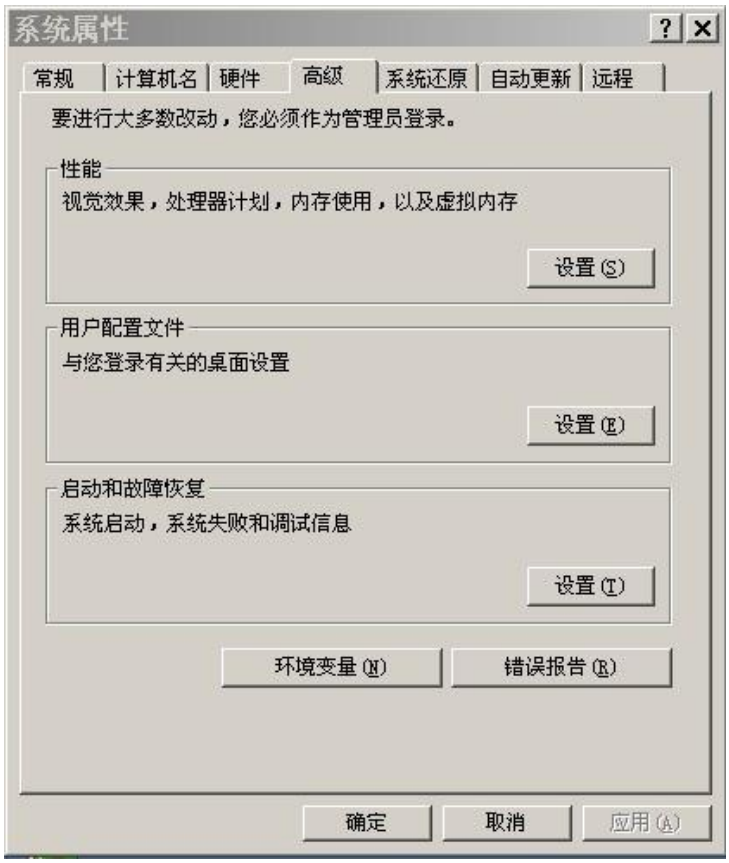
1.6.3 第三步：配置环境变量

安装完成后，还要进行 Java 环境的配置，才能正常使用，步骤如下：

（1）在我的电脑点击右键→选择属性，如下图所示：



（2）在弹出界面上 ：选择高级→环境变量，如下图所示：



（3）弹出如下界面，我们的设置就在这个界面上：

（4）点击环境变量界面的**系统变量**栏的“**新建**”按钮，弹出界面如下图：

在上面填写变量名为：JAVA_HOME，变量值为：C:\Program Files\Java\jdk1.6.0_10，如下图所示：

点击确定按钮。

（5）再次点击环境变量界面的**系统变量**栏的“**新建**”按钮。 在弹出的窗口中填写变量名为：classpath，变量值为：.; ，注意是点和分号，如下图所示：

点击确定按钮。

（6）如下图所示：在系统变量里面找到“Path”这一项，然后双击它，在弹出的界面上，在变量值开头添加如下语句“%JAVA_HOME%\bin;”，注意不要忘了后面的分号，如下图所示：

然后点击“编辑系统变量”界面的确定按钮。

（7）点击“环境变量”界面的确定按钮。

（8）点击“系统属性”界面的确定按钮。Java 开发环境的设置就完成了。

那么为何要设置这些环境变量呢？

JAVA_HOME：配置到 JDK 安装路径。**注意**：变量名必须书写正确，全部大写，中间用下划线。有以下好处：

为了方便引用。比如，JDK安装在C:\Program Files\Java\jdk1.6.0_10 目录里，则设置JAVA_HOME为该目录路径， 那么以后要使用这个路径的时候，只需输入%JAVA_HOME%即可， 避免每次引用都输入很长的路径串。

归一原则。 当JDK路径被迫改变的时候， 仅需更改JAVA_HOME的变量值即可， 否则，你就要更改任何用绝对路径引用JDK目录的文档， 要是万一你没有改全，某个程序找不到JDK，后果是可想而知。

为第三方软件服务。基于 Java 的第三方软件会引用约定好的JAVA_HOME变量，那么它们能够找到 JDK 的位置，不然， 将不能正常使用该软件。如果某个软件不能正常使用，不妨想想是不是这个问题。

PATH：提供给操作系统寻找到 Java 命令工具的路径。通常是配置到 JDK 安装路径\bin。完成配置以后，使用编译器和解释器就会很方便，可以在任何路径下使用 bin 目录下的 Java 命令，而不需要写出完整的路径：C:\Program Files\Java\jdk1.6.0_10 \bin\java 。

CLASSPATH：提供程序在运行期寻找所需资源的路径，比如：.class 类文件、文件、图片等等。在 windows 环境下配置“.”代表当前路径，那么在执行 java 命令时，就会先到当前路径寻找 class 类文件。这个配置对于 Java 初学者比较难理解，我们在后面的 Java 运行过程中，再详细体会。

注意：在 windows 操作系统上，最好在 CLASSPATH 的配置里面，始终在前面保持“.;”的配置。

通过在 windows 窗口设置环境变量，可以永久更改系统环境变量。也可以在 dos 命令行窗口，通过 dos 命令的方式修改环境变量，具体命令如下：

```
set JAVA_HOME=C:\Program Files\Java\jdk1.6.0_10;
set path=%JAVA_HOME%\bin;
set classpath=.;
```

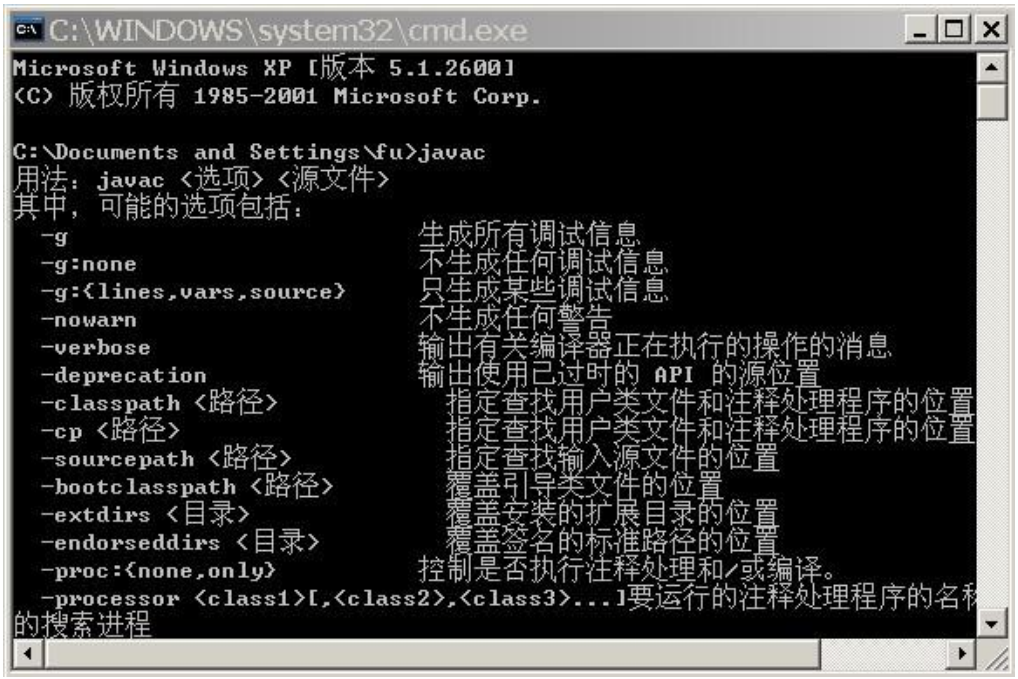
但通过这种方式修改的环境变量只在当前的 dos 命令行窗口有效，对于其他的 dos 命令行窗口则无效。

1.6.4 第四步：检测安装配置是否成功

进行完上面的步骤，基本的安装和配置就好了，怎么知道安装成功没有呢？

（1）点击开始—〉点击运行，在弹出的对话框中输入“cmd”，如下图示：

（2）然后点击确定，在弹出的 dos 命令行窗口里面，输入“javac”，然后回车，出现如下界面则表示安装配置成功。



现在 Java 的开发环境就配置好了，如果出现下面的提示，说明刚才的环境变量的配置有问题，需要从头一步一步检查。

注意：如果更改了系统环境变量的配置，必须重新打开 dos 窗口，就是打开一个新的 dos 窗口，环境变量的新配置才会有效，否则，dos 窗口就还应用旧的环境变量的旧配置。

有些人已经跃跃欲试的想要马上开始学习如何编写 Java 程序了，下面先来体会第一个 Java 程序。

1.7 一个基本的 Java 应用程序—HelloWorld

像其它编程语言一样，Java 编程语言也被用来创建应用程序。一个共同的应用程序范例是在屏幕上显示字串“Hello World!”。下列代码给出了这个 Java 应用程序。

如果是第一次看到 Java 代码，可能都不明白每一个代码的意义，没有关系，主要是来体会一下 Java 程序是什么样子，在脑海里有个印象，然后可以先模仿着做。

1.7.1 HelloWorld

```
1.//这里是注释
2.// HelloWorld 应用示例
3.//
4.public class HelloWorld{
5.  public static void main  (String args[]){
6.     System.out.println("你好，欢迎来到Java快车！");
7.  }
8.}
```

以上程序行是在 dos 窗口上打印“你好，欢迎来到 Java 快车！”所需的最少组件。

1.7.2 描述 HelloWorld

```
1.// 这里是注释
2.// HelloWorld 应用示例
3.//
```

程序中的 1-3 行是注释行

```
4. public class HelloWorld{
```

第 4 行声明类名为 HelloWorld。类名（ClassName）是在源文件中指明的，它可在与源代码相同的目录上创建一个 ClassName.class 文件。在本例题中，编译器创建了一个称为 HelloWorld.class 的文件，它包含了公共类 HelloWorld 的编译代码。

```
5.  public static void main  (String args[]){
```

第 5 行是程序执行的起始点(入口)。Java 解释器必须发现这一严格定义的点，否则将拒绝运行程序。

其它程序语言（特别是 C 和 C++）也采用 main()声明作为程序执行的起始点。此声明的不同部分将在本课程的后几部分介绍。

如果在程序的命令行中给出了任何自变量（命令行参数），它们将被传递给 main()方法中被称作 args 的 String 数组。在本例题中，未使用自变量。

public———说明方法 main()可被任何程序访问，包括 Java 解释器。

static———是一个告知编译器 main()是用于类 HelloWorldApp 中的方法的关键字。为使 main()在程序做其它事之前就开始运行，这一关键字是必要的。

void——— 表明 main()不返回任何信息。这一点是重要的，因为 Java 编程语言要进行谨慎的类型检查，包括检查调用的方法确实返回了这些方法所声明的类型。

String args[]———是一个字符串类型的数组声明，它是 main 方法的参数，它将包含位于类名之后的命令行中的自变量。

Java HelloWorld args[0] args[1]····

{}———两个大括号一起中间的内容叫方法体，代表这个方法所执行的代码。

```
6. System.out.println  ("你好，欢迎来到Java快车！" );
```

第 6 行声明如何使用类名、对象名和方法调用。它使用由 System 类的 out 成员引用的 PrintStream 对象的 println ()方法，将字串“你好，欢迎来到 Java 快车！”打印到标准输出上。

在这个例子中，println()方法被输入了一个字串自变量并将其写在了标准输出流上。

```
7.   }
8. }
```

本程序的 7-8 行分别是方法 main()和类 HelloWorld 的下括号。

1.7.3 编译并运行 HelloWorld

编译

当你创建了 HelloWorld.java 源文件后，用下列命令行进行编译：

```
javac HelloWorld.java
```

如果编译器未返回任何提示信息，表示编译完成，生成一个新文件 HelloWorld. class，该文件称为字节码文件，class 文件名和对应的 java 文件名是相同的。class 文件被存储在与源文件相同的目录中，除非另有指定。

如果在编译中遇到问题，请参阅本模块的查错提示信息部分。

运行

为运行你的 HelloWorld 应用程序，需使用 Java 解释器和位于 bin 目录下的 java 程序：

```
java HelloWorld
你好，欢迎来到Java快车!
```

1.7.4 编译查错

以下是编译时的常见错误：

--javac:Command not found

PATH 变量未正确设置以包括 javac 编译器。javac 编译器位于 JDK 目录下的 bin 目录。

--HelloWorld.java:6: Method printl(java.lang.String)
not found in class java.io.PrintStream.System.
out.printl ("你好，欢迎来到 Java 快车！");
方法名是 println 而不是 printl，少敲了一个 n。

--In class HelloWorld:main must be public or static

该错误的出现是因为 main 方法没有 static 或 public 修饰符。

以下是运行时的错误：

--can't find class HelloWorld

（这个错误是在控制台敲入 java HelloWorld 时产生的）

通常，它表示在命令行中所指定的类名的拼写与 filename.class 文件的拼写不同。Java 编程语言是一种大小写区别对待的语言。

例如： public class Helloworld

创建了一个 Helloworld.class，它不是编译器所预期的类名(HelloWorld.class)。

--命名

如果.java 文件包括一个公共类，那么它必须使用与那个公共类相同的文件名。例如在前例中的类的定义是

public class Helloworld

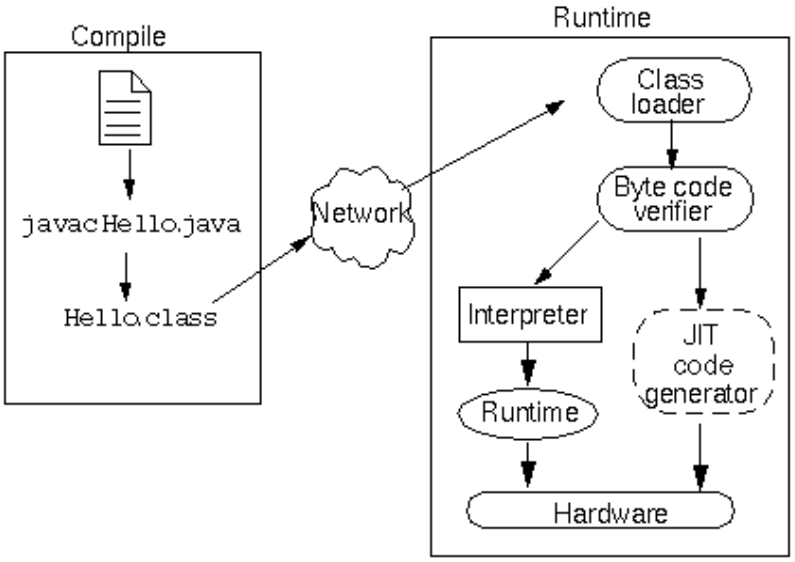
源文件名则必须是 Helloworld.java

--类的数量

在源文件中每次只能定义一个公共类。

1.8 Java 的运行过程

用一个图来描述这个过程会比较容易理解：



编写代码

首先把我们想要计算机做的事情，通过 Java 表达出来，写成 Java 文件，这个过程就是编写代码的过程。如上图所示的 Hello.java 文件。

编译

写完 Java 代码后，机器并不认识我们写的 Java 代码，需要进行编译成为字节码，编译后的文件叫做 class 文件。如上图所示的 Hello.class 文件。

类装载 ClassLoader

类装载的功能是为执行程序寻找和装载所需要的类。

ClassLoader 能够加强代码的安全性，主要方式是：把本机上的类和网络资源类相分离，在调入类的时候进行检查，因而可以限制任何“特洛伊木马”的应用。

字节码（byte-code）校验

字节码校验的功能是对 class 文件的代码进行校验，保证代码的安全性。Java 软件代码在实际运行之前要经过几次测试。JVM 将代码输入一个字节码校验器以测试代码段格式并进行规则检查－－检查伪造指针、违反对象访问权限或试图改变对象类型的非法代码。

字节码校验器对程序代码进行四遍校验，这可以保证代码符合 JVM 规范并且不破坏系统的完整性。如果校验器在完成四遍校验后未返回出错信息，则下列各点可被保证：

- 类符合 JVM 规范的类文件格式
- 无访问限制异常
- 代码未引起操作数栈上溢或下溢
- 所有操作代码的参数类型将总是正确的
- 无非法数据转换发生，如将整数转换为对象引用
- 对象域访问是合法的

解释

可是机器也不能认识 class 文件，还需要被解释器进行解释，解释成机器码，计算机的处理器才能最终理解 Java 程序员所要表达的指令。

运行

最后由运行环境中的 Runtime 对指令进行运行，真正实现我们想要机器完成的工作。

说明

由上面的讲述，大家看到，Java 通过一个**编译阶段**和一个**运行阶段**，来让机器最终理解我们想要它完成的工作，并按照我们的要求进行运行。在这两个阶段中，需要我们去完成的就是编译阶段的工作，也就是说：我们需要把我们想要机器完成的工作用 Java 语言表达出来，写成 Java 源文件，然后把源文件进行编译，形成 class 文件，最后就可以在 Java 运行环境中运行了。运行阶段的工作由 Java 平台自身提供，我们不需要做什么工作。

1.9 Java 技术三大特性

1.9.1 虚拟机

Java 虚拟机 JVM（Java Virtual Machine）在 Java 编程里面具有非常重要的地位，相当于前面学到的 Java 运行环境，虚拟机的基本功能如下：

（1）通过 ClassLoader 寻找和装载 class 文件
（2）解释字节码成为指令并执行，提供 class 文件的运行环境
（3）进行运行期间垃圾回收
（4）提供与硬件交互的平台

Java 虚拟机是在真实机器中用软件模拟实现的一种想象机器。Java 虚拟机代码被存储在 .class 文件中；每个文件都包含最多一个 public 类。Java 虚拟机规范为不同的硬件平台提供了一种编译 Java 技术代码的规范，该规范使 Java 软件独立于平台，因为编译是针对作为虚拟机的“一般机器”而做。这个“一般机器”可用软件模拟并运行于各种现存的计算机系统，

也可用硬件来实现。编译器在获取 Java 应用程序的源代码后，将其生成字节码，它是为 JVM 生成的一种机器码指令。每个 Java 解释器，不管它是 Java 技术开发工具，还是可运行 applets 的 Web 浏览器，都可执行 JVM。

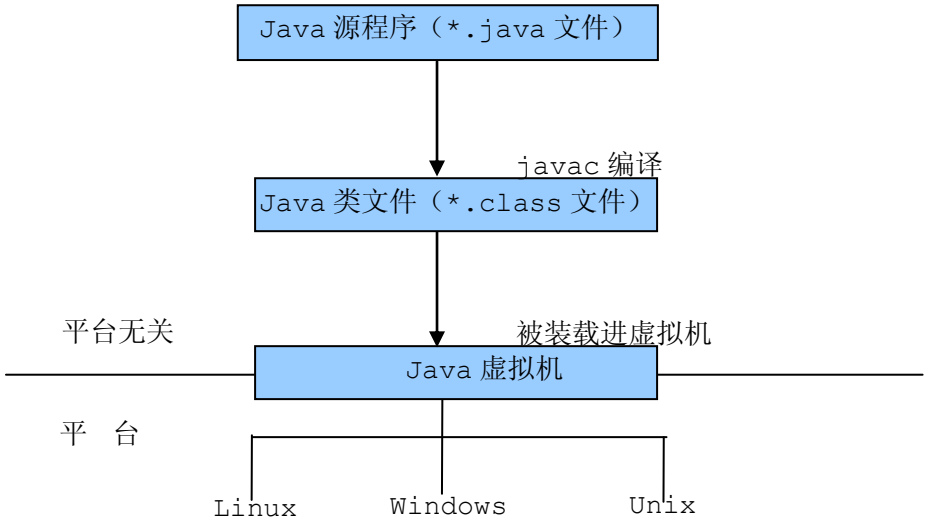
JVM 为下列各项做出了定义：

- 指令集（相当于中央处理器 [CPU]）
- 寄存器
- 类文件格式
- 栈
- 垃圾收集堆
- 存储区

JVM 的代码格式由紧缩有效的字节码构成。由 JVM 字节码编写的程序必须保持适当的类型约束。大部分类型检查是在编译时完成。任何从属的 Java 技术解释器必须能够运行任何含有类文件的程序，这些类文件应符合 Java 虚拟机规范中所指定的类文件格式。

虚拟机是 Java 平台无关的保障

正是因为有虚拟机这个中间层，Java 才能够实现与平台无关。虚拟机就好比是一个 Java 运行的基本平台，所有的 Java 程序都运行在虚拟机上，如下图所示：



1.9.2 垃圾回收

什么是垃圾

在程序运行的过程中，存在被分配了的内存块不再被需要的情况，那么这些内存块对程序来讲就是垃圾。

产生了垃圾，自然就需要清理这些垃圾，更为重要的是需要把这些垃圾所占用的内存资源，回收回来，加以再利用，从而节省资源，提高系统性能。

垃圾回收

- 不再需要的已分配内存应取消分配(释放内存)
- 在其它语言中，取消分配是程序员的责任
- Java 编程语言提供了一种系统级线程以跟踪内存分配：垃圾收集机制
- 可检查和释放不再需要的内存
- 可自动完成上述工作

许多编程语言都允许在程序运行时动态分配内存，分配内存的过程由于语言句法不同而有所变化，但总是要将指针返回到内存的起始位置，当分配内存不再需要时(内存指针已溢出范围)，程序或运行环境应释放内存。

在 C，C++ 或其它语言中，程序员负责释放内存。有时，这是一件很困难的事情。因为你并不总是事先知道内存应在何时被释放。当在系统中没有能够被分配的内存时，可导致程序瘫痪，这种问题被称作**内存漏洞**。

Java 编程语言解除了程序员释放内存的责任。它可提供一种系统级线程以跟踪每一次内存的分配情况。在 Java 虚拟机的空闲周期，垃圾收集线程检查并释放那些可被释放的内存。垃圾收集在 Java 技术程序的生命周期中自动进行，它解除了释放内存的要求，这样能够有效避免内存漏洞和内存泄露(内存泄露就是程序运行期间，所占用的内存一直往上涨， 很容易造成系统资源耗尽而降低性能或崩溃） 。

提示

（1）在 Java 里面，垃圾回收是一个自动的系统行为，程序员不需要控制垃圾回收的功能和行为。比如垃圾回收什么时候开始，什么时候结束，还有到底哪些资源需要回收等。

（2）程序员可以通过设置对象为 null（后面会讲到）来标示某个对象不再被需要了， 这只是表示这个对象可以被回收了，并不是马上被回收。

（3）有一些跟垃圾回收相关的方法，比如：System.gc() ，调用 gc 方法暗示着 Java 虚拟机做了一些努力来回收未用对象，以便能够快速地重用这些对象当前占用的内存。当控制权从方法调用中返回时，虚拟机已经尽最大努力从所有丢弃的对象中回收了空间。

1.9.3 代码安全

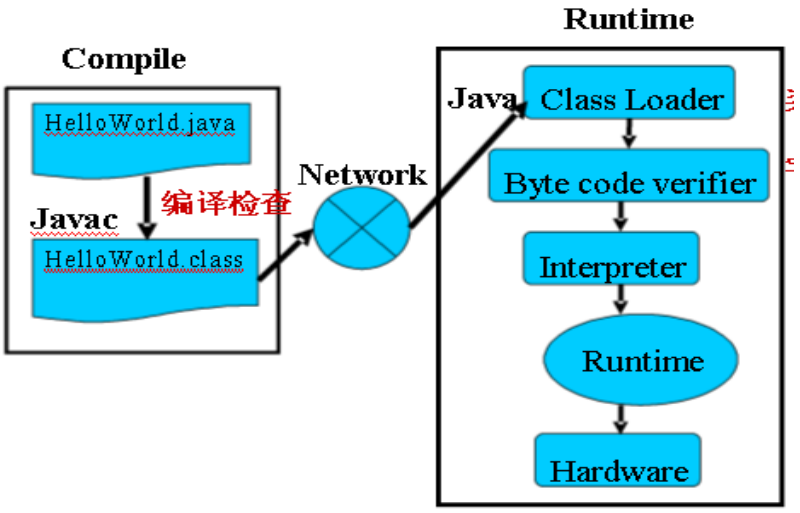
Java 如何保证编写的代码是安全可靠的呢？

第一关：编写的代码首先要被编译成为 class 文件，如果代码写得有问题，编译期间就会发现，然后提示有编译错误，无法编译通过。

第二关：通过编译关后，在类装载的时候，还会进行类装载检查，把本机上的类和网络资源类相分离，在调入类的时候进行检查，因而可以限制任何“特洛伊木马”的应用。

第三关：类装载后，在运行前，还会进行字节码校验，以判断你的程序是安全的。

第四关：如果你的程序在网络上运行，还有沙箱（Sand Box）的保护，什么是沙箱呢？ 就是如果你的程序没有获得授权,只能在沙箱限定的范围内运行，是不能够访问本地资源的， 从而保证安全性。如下图所示：



学习到这里，大家应该对 Java 有了一定的了解了。Java 的内容很丰富，对于初学者来说，不可能一次性将所有看到的知识点理解透彻。在学习完第二章后，对第一章的理解就深入一层；学习完第三章后，对第一章和第二章的理解就又深入一层；直到学习完第八章后，对整个 Java 基础才会有比较全面的了解，所以这并不是“一步一个脚印”的学习过程，而是一个“循序渐进、不断加深”的学习过程，需要同学们反复揣摩，体会 Java 的思想。

1.10 学习目标

学习目标中标注的知识点十分详细，非常适合初学者，能帮助初学者打好坚实的 Java 基础。按照学习目标复习，避免漏掉知识点。对于标注“理解”的，表示能够描述知识点，就是能对别人讲解，能说出相关原因或现象。

1. 了解 Java 语言是什么。（从四种不同的角度理解 Java 语言）
2. 了解 Java 能干什么。
3. 了解 Java 分哪三个版本，分别擅长哪方面的开发。
4. 熟悉如何构建 Java 的开发环境。
5. 掌握如何配置环境变量、为什么配置这些环境变量。
6. 掌握简述开发 HelloWorld 的步骤。
7. 能够描述 HelloWorld 的组成部分及中文解释。
8. 总结在开发 HelloWorld 程序时遇到的**编译错误**和**运行异常**，并在笔记中记录下来。
9. 能够描述 Java 程序的运行过程。
10. 能够描述 Java 虚拟机的功能。
11. 能够描述垃圾回收机制。
12. 能够描述 Java 代码的安全性。

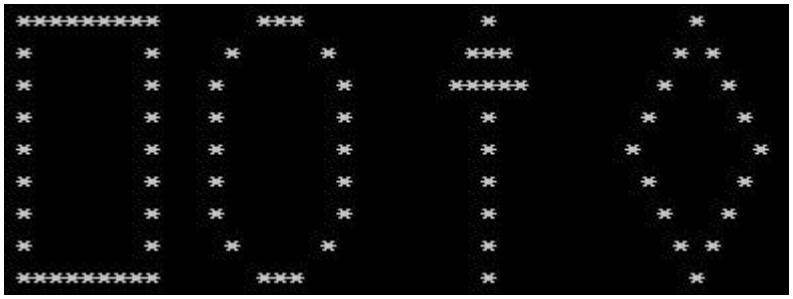
1.11 练习

1. 简述 Java 从代码到运行的全过程。
2. 简述虚拟机的工作机制。
3. 简述 Java 的垃圾回收机制。
4. 简述 path、classpath、JAVA_HOME 各自的含义和配置方式。
5. 简述软件开发基本步骤。
6. 设计一个最简单的输出程序，要求在屏幕上输出以下结果：

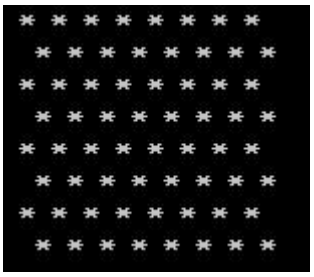
   在 Java 快车学习 Java 是一个很轻松的事

   我很喜欢 Java

7. 写四个程序，分别打印一个矩形、一个椭圆、一个箭头以及一个菱形，如下所示：



8. 用 8 个输出语句显示一个棋盘图案，如下所示：



9. 写一个程序计算从 0 到 10 的数字的平方和立方，并按如下格式打印出来：

number	square	cube
0	0	0
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

2 基础语法

我们在学习汉语的时候，开始学的是一些单个的字，只有认识了单个的字，然后才能组成词，然后才能慢慢的到句子，然后到文章。学习计算机语言跟这个过程是一样的，首先我们要学习一些计算机看得懂的单个的字，然后写成一句代码，最后很多句代码组成程序。 那么这些单个字在 Java 里面就是关键字，让我们的 Java 语法学习从关键字开始吧。

2.1 关键字

2.1.1 什么是关键字

关键字对 Java 技术编译器有特殊的含义，它们可标识数据类型名或程序构造（construct）名。

其实就是个约定或者规定，比如我们看到红灯就知道要停下来，看到绿灯就可以前进了。这些都是人类约定好的游戏规则，在我们的日常生活中有特殊的意义，不可改变，违反它就要付出代价。

关键字是 Java 语言和 Java 的开发和运行平台之间的约定，程序员只要按照这个约定使用了某个关键字，Java 的开发和运行平台就能够认识它，并正确地处理，展示出程序员想要的效果。

2.1.2 Java 中的关键字

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

- abstract：表明类或类中的方法是抽象的；
- assert：声明断言；
- boolean：基本数据类型之一，布尔类型；
- break：提前跳出一个块；
- byte：基本数据类型之一，字节类型；
- case：在 switch 语句中，表明其中的一个分支；
- catch：用于处理例外情况，用来捕捉异常；

- char：基本数据类型之一，字符类型；
- class：类；
- continue：回到一个块的开始处；
- default：用在 switch 语句中，表明一个默认的分支；
- do：用在"do while"循环结构中；
- double：基本数据类型之一，双精度浮点数类型；
- else：在条件语句中，表明当条件不成立时的分支；
- extends：用来表明一个类是另一个类的子类；
- final：用来表明一个类不能派生出子类，或类中的方法不能被覆盖，或声明一个变量是常量；
- finally：用于处理异常情况，用来声明一个肯定会被执行到的块；
- float：基本数据类型之一，单精度浮点数类型；
- for：一种循环结构的引导词；
- if：条件语句的引导词；
- implements：表明一个类实现了给定的接口；
- import：表明要访问指定的类或包；
- instanceof：用来测试一个对象是否是一个指定类的实例；
- int：基本数据类型之一，整数类型；
- interface：接口；
- long：基本数据类型之一，长整数类型；
- native：用来声明一个方法是由与机器相关的语言(如 C/C++/FORTRAN 语言)实现的；
- new：用来申请新对象；
- package：包；
- private：一种访问方式：私有模式；
- protected：一种访问方式：保护模式；
- public：一种访问方式：公共模式；
- return：从方法中返回值；
- short：基本数据类型之一，短整数类型；
- static：表明域或方法是静态的，即该域或方法是属于类的；
- strictfp：用来声明 FP-strict(双精度或单精度浮点数)表达式，参见 IEEE 754 算术规范；
- super：当前对象的父类对象的引用；
- switch：分支结构的引导词；
- synchronized：表明一段代码的执行需要同步；
- this：当前对象的引用；
- throw：抛出一个异常；
- throws：声明方法中抛出的所有异常；
- transient：声明不用序列化的域；
- try：尝试一个可能抛出异常的程序块
- void：表明方法不返回值；
- volatile：表明两个或多个变量必须同步地发生变化；
- while：用在循环结构中；
- enum：声明枚举类型；

说明：

1. **这些关键字的具体含义和使用方法，会在后面使用的时候详细讲述。**
2. **Java 的关键字也是随新的版本发布在不断变动中的，不是一成不变的。**
3. **所有关键字都是小写的。**

2.2 标识符

每个人都有名字，每个事物也都有名字，有了名字，就可以通过语言表示出来。Java 的文件、类、方法、变量也都有名字。中国人的名字有默认的命名规则，比如不使用阿拉伯数字，不使用标点符号，不使用长辈的名字，等等。Java 中有什么命名规则呢？

2.2.1 什么是标识符

在 Java 编程语言中，标识符是标识变量、类或方法的有效字符序列。简单地说标识符就是一个名字。

2.2.2 标识符命名规则

命名规则如下：

（1）首字母只能以字母、下划线、$开头,其后可以跟字母、下划线、$和数字。

示例：$abc 、 _ab 、ab123 等都是有效的。

（2）标识符不能是关键字。

（3）标识符区分大小写 （事实上整个 Java 编程里面都是区分大小写的）。

Girl 和 girl 是两个不同的标识符。

（4）标识符长度没有限制，但不宜过长。

（5）如果标识符由多个单词构成，那么从第二个单词开始，首字母大写。

示例：getUser、setModel、EmployeeModel 等。

（6）标识符尽量命名的有意义，让人能够望文知意。

（7）尽量少用带$符号的标识符，主要是习惯问题，大家都不是很习惯使用带$符号的标识符；还有内部类中，$具有特殊的含义。

（8）虽然 Java 语言使用 16-bit 双字节字符编码标准（Unicode 字符集），最多可以识别 65536 个字符(0-65535)。建议标识符中最好使用 ASCII 字母。虽然中文标识符也能够正常编译和运行，却不建议使用。

```
public class Test {
    public static void main(String[] args) {
       String Java快车 = "中文标识符测试";
       System.out.println("Java快车==" + Java快车);
    }
}
```

运行结果：Java 快车==中文标识符测试

2.2.3 示例

下列哪些是正确的标识符：

```
my Variable
9javakc
a+b
book1-2-3
java&c
It's
```

错误的标识符及其原因分析如下：

```
My Variable //含有空格
9javakc //首字符为数字
a+b //加号不是字母
book1-2-3 //减号不是字母
java&c //与号不是字母
It's //单引号不是字母
```

2.3 数据类型

2.3.1 什么叫数据类型

数据类型简单的说就是对数据的分类，对数据各自的特点进行类别的划分，划分的每种数据类型都具有区别于其它类型的特征，每一类数据都有相应的特点和操作功能。例如数字类型的就能够进行加减乘除的操作。

在现实生活中，我们通常会针对不同的提问，做出不同类型的回答，比如：

你叫什么名字？ --刘德华
你今天多大年纪了？ --24
你家住哪里？ --北京市海淀区上地信息路
请告诉我，你的身高？ --1.75
你带课本了，是吗？ --是的
1+1=2，对吗？ --对

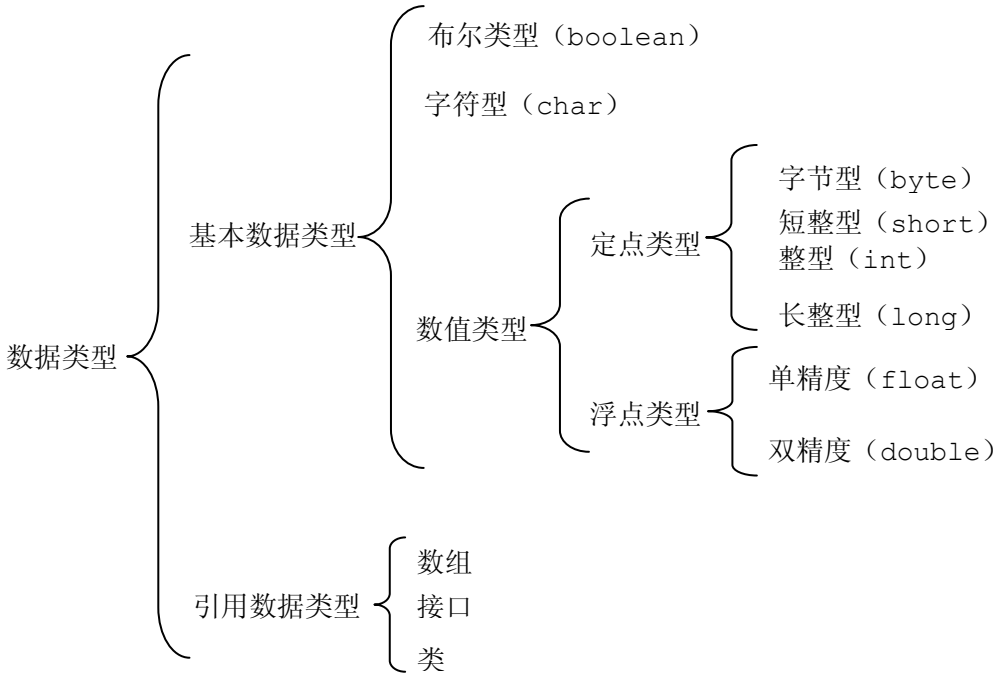
大家仔细分析一下回答的信息（数据），是不是有类别之分？如果类别搞错了，是不是会出笑话？

类似的在程序中，计算机也需要某种方式来判断某个数字是什么类型的。这通常是需要程序员显示来声明某个数据是什么类型的，Java 就是这样的。Java 是一种强类型的语言，凡是使用到的变量，在编译之前一定要被显示的声明。

Java 的安全和健壮性部分来自于它是强类型语言这一事实。首先，每个变量有类型，每个表达式有类型，而且每种类型是严格定义的。其次，所有的数值传递，不管是直接的还是通过方法调用经由参数传过去的都要先进行类型相容性的检查。有些语言没有自动强迫进行数据类型相容性的检查或对冲突的类型进行转换的机制。Java 编译器对所有的表达式和参数都要进行类型相容性的检查以保证类型是兼容的。任何类型的不匹配都是错误的，在编译器完成编译以前，错误必须被改正。

2.3.2 Java 数据类型的分类

Java 里面的数据类型从大的方面分为两类，一是基本数据类型，一是引用类型。基本的 JAVA 数据类型层次图如下：



2.3.3 Java 中的基本数据类型

Java 中的基本数据类型可分为四种：

（1）逻辑型：boolean

（2）文本型：char

（3）整数型：byte、short、int、long

（4）浮点型：float、double

2.3.3.1 逻辑型 boolean

逻辑值有两种状态，即人们经常使用的“on”和“off” 或“true”和“false”或“yes”和 “no”，这样的值是用 boolean 类型来表示的。boolean 有两个文字值，即 true 和 false。 以下是一个有关 boolean 类型变量的声明和初始化：

```
boolean truth = true; //声明变量 truth
```

注意：在整数类型和 boolean 类型之间无转换计算。有些语言（特别值得强调的是 C 和 C++）允许将数字值转换成逻辑值（所谓“非零即真”），这在 Java 编程语言中是不允许的；boolean 类型只允许使用 boolean 值（true 或 false）。

2.3.3.2 文本型 char

char 类型用来表示单个字符。一个 char 代表一个 16-bit 无符号的（不分正负的）Unicode 字符，一个 char 字符必须包含在单引号内。

示例：

```
'a' //表示简单的字符
'1' //用数字也可以表示字符
下面就错了，只能使用单个字符。
'ab' //错误
'12' //错误
```

什么是 Unicode 编码

Unicode 编码又叫统一码、万国码或单一码，是一种在计算机上使用的字符编码。它为每种语言中的每个字符设定了统一并且唯一的二进制编码，以满足跨语言、跨平台进行文本转换、处理的要求。1990 年开始研发，1994 年正式公布。随着计算机工作能力的增强，Unicode 也在面世以来的十多年里得到普及。

Unicode 字符集最多可以识别 65535 个字符，每个国建的“字母表”的字母都是 Unicode 表中的一个字符，比如汉字中的“你”字就是 Unicode 表中的第 20320 个字符，还包含日文里的片假名、平假名、韩文以及其他语言中的文字。

在 Java 中的定义示例：

```
char c = 'a';
char c = '1';
```

取值范围和默认值：

名称	长度	范围	默认值
char	16 位	$0\sim2^{16}-1$	'\u0000'

Java 里面的转义字符

转义字符是指，用一些普通字符的组合来代替一些特殊字符，由于其组合改变了原来字符表示的含义，因此称为“转义”。 常见的转义字符：

```
\n 回车(\u000a)
\t 水平制表符(\u0009)
\b 退格(\u0008)
\r 换行(\u000d)
\f 换页(\u000c)
\' 单引号(\u0027)
\" 双引号(\u0022)
\\ 反斜杠(\u005c)
```

2.3.3.3 整数型 byte、short、int、long

```
byte：字节型
short：短整型
int：整型
long：长整型
```

在 Java 中，整数型的值都是带符号的数字，可以用十进制、八进制和十六进制来表示。所谓几进制，就是满多少就进位的意思，如十进制表示逢十进位，八进制就表示逢八进位。 示例：

```
2：十进制的 2
077：首位的 0 表示这个一个 8 进制的数值，相当于十进制的 63，计算公式：7*8+7=63
0x7C：首位的 0x表示这个一个 16 进制的数值，相当于十进制的 124，计算公式：
7*16+12=124
```

在 Java 中的定义示例：

```
示例：byte bt = 5;
表示在 Java 中定义一个变量 bt，类型是 byte 类型，值是 5
同理可以定义其它的类型： 比如：
short sh = 5;
int i = 5;
long lon = 5;
```

这些都是可以的，如果要明确表示是 long 型的值，可以在后面直接跟一个字母“L”或者“l”。L 表示一个 long 值。

也就是写成：long abc4 =5L;

```
请注意：
    在 Java 编程语言中使用大写或小写 L 同样都是有效的，但由于小写 l 与数字 1 容易混淆，因而，尽量不要使用小写。
    整数型的值，如果没有特别指明，默认是 int 型。
```

取值范围和默认值：

名称	长度	范围	默认值
byte	8 位	-2^7~2^7-1	0
short	16 位	-2^{15}~2^{15}-1	0
int	32 位	-2^{31}~2^{31}-1	0
long	64 位	-2^{63}~2^{63}-1	0L

2.3.3.4 浮点型 float、double

Java 用浮点型来表示实数，简单地说就是带小数的数据。

用关键字 float 或 double 来定义浮点类型，如果一个数字包括小数点或指数部分，或者在数字后带有字母 F 或 f（float）、D 或 d（double），则该数字文字为浮点型的。

示例：

```
12.3 //浮点型数据
12.3E10 //一个大浮点数据，E或e表示指数值，相当于 12.3*10^10
```

在 Java 中的定义示例 ：

```
float f1 = 3.14F;
float f2 = 3.14f;
double d1 = 3.14;
double d2 = 3.14D;
double d3 = 3.14d;
```

请注意：

浮点型的值，如果没有特别指，默认是 double 型的。

定义 float 型的时候，一定要指明是 float 型的，可以通过在数字后面添加“F”或者“f”来表示。

定义 double 型的时候，可以不用指明，默认就是 double 型的，也可以通过在数字后面添加“D”或者“d”来表示。

取值范围和默认值：

名称	长度	范围	默认值
float	32 位		0.0f
double	64 位		0.0d

Java 技术规范的浮点数的格式是由电力电子工程师学会（IEEE）754 定义的，是独立于平台的。可以通过 Float.MAX_VALUE 和 Float.MIN_VALUE 取得 Float 的最大最小值；可以通过 Double.MAX_VALUE 和 Double.MIN_VALUE 来取得 Double 的最大最小值。

2.3.3.5 特别介绍:字符串型 String

字符型只能表示一个字符，那么多个字符怎么表示呢？

Java 中使用 String 这个类来表示多个字符，表示方式是用双引号把要表示的字符串引起来，字符串里面的字符数量是任意多个。字符本身符合 Unicode 标准， char 类型的反斜线符号(转义字符)适用于 String。与 C 和 C++不同，String 不能用\0 作为结束。String 的文字应用双引号封闭，如下所示：

“The quick brown fox jumped over the lazy dog.”

char 和 String 类型变量的声明和初始化如下所示：

```
char ch = 'A'; // 声明并初始化一个字符变量
String str1 = "java快车"; //字符串类型
String str2 = ""; //表示空字符串
String str3 = null; //表示空（注意不是空字符串）
```

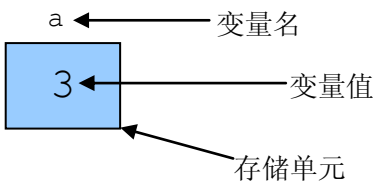
注意：

（1）String不是原始的数据类型，而是一个类(class)。
（2）String 包含的字符数量是任意多个，而字符类型只能是一个。
要特别注意：
"a"表示的是字符串，而'a'表示的是字符类型，它们的意义和功能都不同。
（3）String 的默认值是 null。

2.4 变量和常量

2.4.1 变量

变量是 Java 程序的一个基本存储单元，它可以用来引用另一个存储单元。变量由一个标识符，类型及一个可选初始值的组合定义。此外，所有的变量都有一个作用域，定义变量的可见性，生存期。变量的值是可以改变的，可以通过操作变量来操作变量所对应的内存区域或值块的值。



变量的创建通过声明完成，执行变量声明语句时，系统根据变量的数据类型在内存中开辟相应的存储空间并赋予初始值。

赋值就是为一个声明的变量或者常量赋予具体的值，也就是赋予值的意思。使用一个等号“=”来表示。

变量的定义规则：

（1）遵从所有标识符的规则。
（2）所有变量都可大小写混用，但首字符应小写。
（3）尽量不要使用下划线和$符号。
（4）可以先声明再赋值，如：

```
int i;
i=9;
```

也可以声明的同时进行赋值：

```
int i=9;
```

这句话的意思就是，声明一个类型为 **int** 的变量 **i**，并将它赋值为 **9**。
（5）没有赋值的变量是不可以使用的。如：

```
int i;
System.out.println(i);//这句代码是错误的。
```

几点说明：

（1）变量在计算机内部对应着一个存储单元，而且总是具有某种数据类型：基本数据类型或引用数据类型。
（2）变量总是具有与其数据类型相对应的值。
（3）每个变量均具有：名字、类型、一定大小的存储单元以及值。
（4）变量有一个作用范围，超出它声明语句所在的块就无效。

2.4.2 常量

常量是变量中的一个特例，用 final 关键字修饰，常量是值是不可以改变的，也就是说常量引用的存储单元中的数据是不可更改的。

对常量命名的定义规则：建议大家尽量全部大写，并用下划线将词分隔。

如：JAVAKC_CLASS、PI、FILE_PATH

2.5 注释

什么是注释呢？就是标注解释的意思，主要用来对 Java 代码进行说明。Java 中有三种

注释方式

（1）// 注释单行语句

示例：

```
//定义一个值为 10 的 int 变量
int a = 10;
```

（2）/* */ 多行注释

示例：

```
/*
这是一个注释，不会被 Java 用来运行 这是第二行注释，可以有任意多行
*/
```

（3）/ */ 文档注释**

紧放在变量、方法或类的声明之前的文档注释， 表示该注释应该被放在自动生成的文档中(由 javadoc 命令生成的 HTML 文件)以当作对声明项的描述。

示例：

```
/**
*  这是一个文档注释的测试
*  它会通过 javadoc 生成标准的 java 接口文档
*/
```

在 javadoc 注释中加入一个以“@”开头的标记，结合 javadoc 指令的参数，可以在生成的 API 文档中产生特定的标记

常用的 javadoc 标记：

```
@author：作者
@version：版本
@deprecated：不推荐使用的方法
@param：方法的参数类型
@return：方法的返回类型
@see："参见"，用于指定参考的内容
@exception：抛出的异常
@throws：抛出的异常，和 exception 同义
```

javadoc 标记的应用范围：

```
在类和接口文档注释中的标记有：@see  @deprecated  @author  @version
在方法或者构造方法中的标记有：@see  @deprecated  @param  @return
@exception  @throws
 在属性文档注释中的标记：@see  @deprecated
```

2.6 运算符和表达式

程序的基本功能是处理数据，任何编程语言都有自己的运算符。为实现逻辑和运算要求，编程语言设置了各种不同的运算符，且有优先级顺序，所以有的初学者使用复杂表达式的时候搞不清楚。

下面按优先顺序列出了各种运算符：

优先级	运算符分类	运算符
由高到低	分隔符	. [] () ; ,
	一元运算符	! ++ -- - ~
	算术运算符	* / % + -
	移位运算符	<< >> >>>
	关系运算符	< > <= >= == != instanceof(Java 特有)
	逻辑运算符	! && || ~ & | ^
	三目运算符	布尔表达式?表达式 1:表达式 2
	赋值运算符	= *= /= %= += -= <<= >>= >>>= &= ~= |= ^=

表达式是由常量、变量、对象、方法调用和操作符组成的式子。表达式必须符合一定的规范，才可被系统理解、编译和运行。表达式的值就是对表达式自身运算后得到的结果。

根据运算符的不同，表达式相应地分为以下几类：算术表达式、关系表达式、逻辑表达式、赋值表达式，这些都属于数值表达式。

2.6.1 一元运算符

因操作数是一个故称为一元运算符。

运算符	含义	示例
-	改变数值的符号，取反	-10 -x
~	右结合	~10 ~x
++	左结合	++x; x++;
--	左结合	--x; x--;

需要注意的是 **++** 或 **--** 操作：

++x 因为**++**在前，所以先加后用。
x++ 因为**++**在后，所以先用后加。

有一种特殊的情况：a+ ++b 和 a+++b 是不一样的（因为有一个空格）。

```
int a = 10;
int b = 10;
int sum = a + ++b;
System.out.println("a=" + a + ",b=" + b + ",sum=" + sum);
```

运行结果是： a=10,b=11,sum=21

```
int a = 10;
int b = 10;
int sum = a++ + b;
System.out.println("a=" + a + ",b=" + b + ",sum=" + sum);
```

运行结果是：a=11,b=10,sum=20

```
n=10;
m=~n;
```

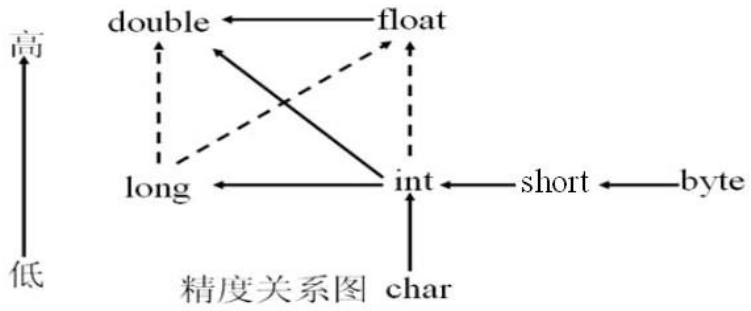
变量 n 的二进制数形式： 00000000 00000000 00000000 00001010
逐位取反后，等于十进制的-11 11111111 11111111 11111111 11110101

2.6.2 算术运算

算术运算是指：＋、－、*、/ 等基本运算。

运算符	含义	代码示例	运算示例
+	加法	x+y	1+2 结果：3 ； 1.2+1 结果：2.2
-	减法	x-y	1-2 结果：-1 ； 1.2-1 结果：0.2
*	乘法	x*y	1*2 结果：2 ； 1.2*1 结果：1.2
/	除法	x/y	5/2 结果：2 ； 5.2/2 结果：2.6
%	求模(余)	x%y	5%2 结果：1

这些操作可以对不同类型的数字进行混合运算，为了保证操作的精度，系统在运算过程中会做相应的转化。数字精度的问题，我们在这里不再讨论。下图中展示了运算过程中，数据自动向上造型的原则。



1. 实线箭头表示没有信息丢失的转换，也就是安全性的转换，虚线的箭头表示有精度损失的转化，也就是不安全的。
2. 当两个操作数类型不相同时，操作数在运算前会子松向上造型成相同的类型，再进行运算。

示例如下：

```
int a = 22;
int b = 5;
double c = 5.0;
System.out.println(b + "+" + c + "=" + (b + c)); //10.0
System.out.println(b + "-" + c + "=" + (b - c)); //0.0
System.out.println(b + "*" + c + "=" + (b * c)); //25.0
System.out.println(a + "/" + b + "=" + (a / b)); //4
System.out.println(a + "%" + b + "=" + (a % b)); //2
System.out.println(a + "/" + c + "=" + (a / c)); //4.4
System.out.println(a + "%" + c + "=" + (a % c)); //2.0
```

用加号（+）进行串链接。

运算符 + 能够进行 String 对象的链接并生成一个新的 String：

```
String salutation = "Dr. ";
String name = "Jack " + "Arthur";
String title = salutation + name;
```

最后一行的结果是：

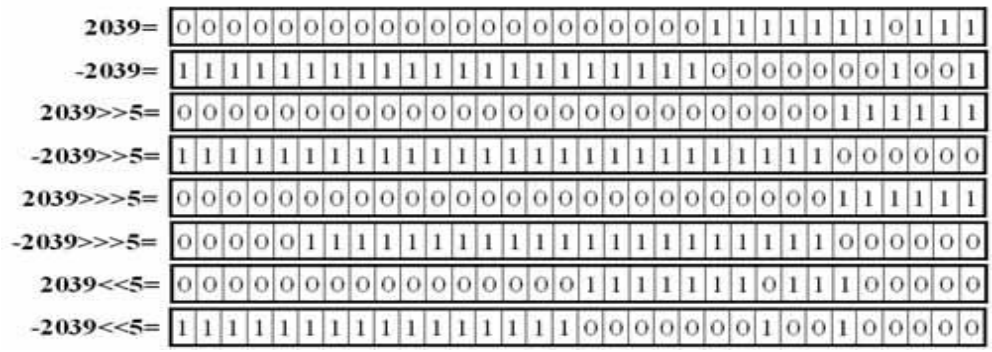
```
Dr. Jack Arthur
```

如果+运算符中有一个自变量为 String 对象，则其它自变量将被转换成 String。所有对象都可被自动转换成 String，尽管这样做的结果可能是意义含糊的。不是字符串的对象是通过使用 toString()成员方法而转换成串的等价物的。

2.6.3 移位运算符

移位运算符操作的对象就是二进制的位，可以单独用移位运算符来处理 int 型或 long 型整数，在运算 byte 和 short 类型数据时，会自动向上造型成 int，再运算。

运算符	含义	示例
<<	左移运算符，将运算符左边的对象向左移动运算符右边指定的位数（在低位补 0）	100<<3 : 800
>>	"有符号"右移运算符，将运算符左边的对象向右移动运算符右边指定的位数。使用符号扩展机制，也就是说，如果值为正，则在高位补 0，如果值为负，则在高位补 1.	100>>3 :12
>>>	"无符号"右移运算符，将运算符左边的对象向右移动运算符右边指定的位数。采用 0 扩展机制，也就是说，无论值的正负，都在高位补 0.	-100>>>3 :536870899



以 int 类型的 2039 为例，代码如下：

```
System.out.println(Integer.toBinaryString(2039));
System.out.println(Integer.toBinaryString(-2039));
System.out.println(Integer.toBinaryString(2039 >> 5));
System.out.println(Integer.toBinaryString(-2039 >> 5));
System.out.println(Integer.toBinaryString(2039 >>> 5));
System.out.println(Integer.toBinaryString(-2039 >>> 5));
```

```
        System.out.println(Integer.toBinaryString(2039 << 5));
        System.out.println(Integer.toBinaryString(-2039 << 5));
```

运行结果如下：

```
11111110111
11111111111111111111100000001001
111111
11111111111111111111111111000000
111111
11111111111111111111111000000
1111111011100000
11111111111111110000000100100000
```

注意：

负数的二进制表现形式是正数取反加一。

x<<y 相当于 $x*2^y$ ； x>>y相当于$x/2^y$

从计算速度上讲，移位运算要比算术运算快。

如果x是负数，那么x>>>3 没有什么算术意义，只有逻辑意义。

移位运算符将它们右侧的操作数模 32 简化为 int 类型左侧操作数，模 64 简化为 long 类型右侧操作数。因而， 任何 int x, x >>> 32 都会导致不变的 x 值，而不是你可能预计的零 。

2.6.4 关系（比较）运算符

Java 具有完备的关系运算符，这些关系运算符同数学中的关系运算符是一致的。具体说明如下：

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

关系运算符用于比较两个数据之间的大小关系，产生的结果都是布尔型的值，一般情况下，在逻辑与控制中会经常使用关系运算符，用于选择控制的分支，实现逻辑要求。

instanceof 操作符用于判断一个引用类型所引用的对象是否是某个指定的类或子类的实例，如果是，返回真(true)，否则返回假(false)。

操作符左边的操作元是一个引用类型，右边的操作元是一个类名或者接口，形式如下：

```
    obj instanceof ClassName   或者  obj instanceof InterfaceName
```

需要注意的是：关系运算符中的"=="和"!="既可以操作基本数据类型，也可以操作引用数据类型。操作引用数据类型时，比较的是引用的内存地址。所以在比较非基本数据类型时，应该使用 equals 方法。

简单示例如下：

```
public class TestRelation {
	public static void main(String args[]) {
		// 变量初始化
		int a = 30;
		int b = 20;
		// 定义结果变量
		boolean r1, r2, r3, r4, r5, r6;
		// 计算结果
		r1 = a == b;
		r2 = a != b;
		r3 = a > b;
		r4 = a < b;
		r5 = a >= b;
		r6 = a <= b;
		// 输出结果
		System.out.println("a = " + a + "   b = " + b);
		System.out.println("a==b = " + r1);
		System.out.println("a!=b = " + r2);
		System.out.println("a>b = " + r3);
		System.out.println("a<b = " + r4);
		System.out.println("a>=b = " + r5);
		System.out.println("a<=b = " + r6);
	}
}
```

运行结果如下：

```
a = 30   b = 20
a==b = false
a!=b = true
a>b = true
a<b = false
a>=b = true
a<=b = false
```

2.6.5 逻辑运算

运算符！(定义为“非”)、&&(定义为“与”)、||(定义为“或”)执行布尔逻辑表达式。

逻辑非关系值表	
A	! A
true	false
false	true

逻辑与关系值表		
A	B	A&&B
true	true	true
true	false	false
false	true	false
false	false	false

逻辑或关系值表		
A	B	A||B
true	true	true
true	false	true
false	true	true
false	false	false

在运用逻辑运算符进行相关的操作，就不得不说“短路”现象。代码如下：

```
if(1==1 && 1==2 && 1==3){ }
```

代码从左至右执行，执行第一个逻辑表达式后：true && 1==2 && 1==3

执行第二个逻辑表达式后：true && false && 1==3

因为其中有一个表达式的值是 false，可以判定整个表达式的值是 false，就没有必要执行第三个表达式了，所以 Java 虚拟机不执行 1==3 代码，就好像被短路掉了。

逻辑或也存在“短路”现象，当执行到有一个表达式的值为 true 时，整个表达式的值就为 true，后面的代码就不执行了。

“短路”现象在多重判断和逻辑处理中非常有用。我们经常这样使用：

```
public void a(String str) {
    if (str != null && str.trim().length() > 0) {
        //do some thing;
    }
}
```

如果 str 为 null，那么执行 str.trim().length()就会报错，短路现象保证了我们的代码能够正确执行。

在书写布尔表达式时，首先处理主要条件，如果主要条件已经不满足，其他条件也就失去了处理的意义。也提高了代码的执行效率。

2.6.6 位运算

位运算符包括：&（与），|（或），~（取反），^（异或）；位运算是对整数的二进制位进行相关操作，详细运算如下：

非位关系值表	
A	~A
1	0
0	1

与位关系值表		
A	B	A
1	0	0
0	1	0
1	1	1
0	0	0

或位关系值表		
A	B	A
1	0	1
0	1	1
1	1	1
0	0	0

异或位关系值表		
A	B	A
1	0	1
0	1	1
1	1	0
0	0	0

位运算示例：

```
    int a = 15;
    int b = 2;

    System.out.println(a + "&" + b + "=" + (a & b));
    System.out.println(a + "|" + b + "=" + (a | b));
    System.out.println(a + "^" + b + "=" + (a ^ b));
    System.out.println("~" + b + "=" + (~b));
```

运算结果如下：

```
15&2=2
    15|2=15
    15^2=13
    ~2=-3
```

2.6.7 三目运算

三目运算符(?:)相当于条件判断，表达式 x?y:z 用于判断 x 是否为真，如果为真，表达式的值为 y，否则表达式的值为 z。

例如：

```
public class Test {
    public static void main(String[] args) {
        int i = (5 > 3) ? 6 : 7;
        System.out.println("the i=" + i);
    }
}
```

运行结果为：the i=6

其实三目运算符的基本功能相当于 if-else（马上就要学到了），使用三目运算符是因为它的表达比相同功能的 if-else 更简洁。上面的例子改成用 if-else 表达如下：

```
public class Test {
    public static void main(String[] args) {
        int i = 0;
        if (5 > 3) {
            i = 6;
        } else {
            i = 7;
        }
        System.out.println("the i=" + i);
    }
}
```

运行结果为：the i=6

2.7 控制语句

2.7.1 分支控制语句

分支语句又称条件语句，条件语句使部分程序可根据某些表达式的值被有选择地执行。Java 编程语言支持双路 if 和多路 switch 分支语句。

2.7.1.1 if 语句

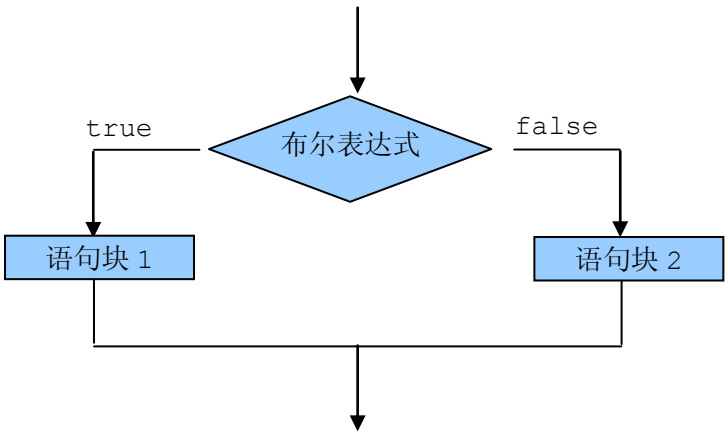
if-else 语句的基本句法是：

```
if(布尔表达式) {
    语句块 1;
} else {
    语句块 2;
```

```
}
```

说明：

(1) 布尔表达式返回值为true或false。
(2) 如果为true，则执行语句或块 1，执行完毕跳出if-else语句。
(3) 如果为false，则跳过语句或块 1，然后执行else下的语句或块 2。



例如：

```
int a=10;
int b=10;
if (a >= b) {
    System.out.println("a 大于等于 b");
} else {
    System.out.println("a 小于 b");
}
```

在 Java 编程语言中，if ()用的是一个布尔表达式，而不是数字值，这一点与 C/C++不同。前面已经讲过，布尔类型和数字类型不能相互转换。因而，如果出现下列情况：

```
if (x) // x 是 int 型
你应该使用下列语句替代：
if (x != 0)
```

注意：

(1)if 块和 else 块可以包含任意的 java 代码，自然也就可以包含新的 if-else，也就是说：if-else 是可以嵌套的，嵌套的规则是不可以交叉，必须完全包含。比如：

```
if (a1 > a2) {
    if (a1 > a3) {
        // 包含在里面的块必须先结束
    } else {
        // 同样可以包含 if-else 块
    }
}
```

（2）else 部分是选择性的，并且当测试条件为假时如不需做任何事，else 部分可被省略。也就是说 if 块可以独立存在，但是 else 块不能独立存在，必须要有 if 块才能有 else 块。

（3）如果 if 块和 else 块的语句只有一句时，可以省略{}，例如：

```
if (a1 > a2)
    System.out.println("这是 if块");
else
    System.out.println("这是 else 块");
```

上面的代码从语法角度是完全正确的，但是从代码的可阅读性上，容易让人产生迷惑，所以我们不建议大家这么写。

（4）还可以把 else 和 if 组合使用，就是使用 else if，表达"否则如果"的意思，示例如下：

```
if (score >= 90) {
    grade = "very good";
} else if (score >= 70) {
    grade = "good";
} else if (score >= 60) {
    grade = "pass";
} else {
    grade = "No pass";
}
```

（5）如果不用“{ }”，则“else”总是与最接近它的前一个“if”相匹配，例如：

```
if (x > 5)
    if (y > 5)
        System.out.println("x and y are > 5");
else
    System.out.println("x is <= 5");
```

执行顺序与上面的对应匹配形式不同，而是与下面形式匹配

```
if (x > 5)
    if (y > 5)
        System.out.println("x and y are > 5");
    else
        System.out.println("x is <= 5");
```

2.7.1.2 switch 语句

switch 表示选择分支的情况，switch 语句的句法是：

```
switch (表达式 1) {
    case 表达式 2:
```

```
            语句块 2;
            break;
        case  表达式 3:
            语句块 3;
            break;
        default:
            语句块 4;
            break;
    }
```

说明：

(1)表达式 1 的值必须与整型兼容或者 enum 枚举类型的常量值，包含 byte、short、int 和 char，不能是字符串或对象，也不能是 long 型的值。

(2)case 分支要执行的程序语句。

(3)表达式 2、3、4 是可能出现的值。

(4)不同的 case 分支对应着不同的语句或块序列。

(5)break 表示跳出这一分支。

(6)当变量或表达式的值不能与任何 case 值相匹配时，可选缺省符（defauit）指出了应该执行的程序代码。

示例：

```
public class Test {
    public static void main(String[] args) {
        int colorNum = 5;
        switch (colorNum) {
        case 0:
            System.out.println(Color.red);
            break;
        case 1:
            System.out.println(Color.green);
            break;
        default:
            System.out.println(Color.black);
            break;
        }
    }
}
```

运行结果：

```
java.awt.Color[r=0,g=0,b=0]
```

(6)如果没有 break 语句作为某一个 case 代码段的结束句，则程序的执行将继续到下一个 case，而不检查 case 表达式的值。示例：

```
import java.awt.Color;
public class  Test {
    public static void main(String[] args) {
        int colorNum = 0;
        switch (colorNum) {
        case 0:
            System.out.println(Color.red);
        case 1:
            System.out.println(Color.green);
        default:
            System.out.println(Color.black);
            break;
        }
    }
}
```

运行结果：

```
java.awt.Color[r=255,g=0,b=0]
java.awt.Color[r=0,g=255,b=0]
java.awt.Color[r=0,g=0,b=0]
```

2.7.2 循环控制语句

循环语句使语句或块的执行得以重复进行。Java 编程语言支持三种循环构造类型：for，while 和 do 循环。for 和 while 循环是在执行循环体之前测试循环条件，而 do 循环是在执行完循环体之后测试循环条件。这就意味着 for 和 while 循环可能连一次循环体都未执行，而 do 循环将至少执行一次循环体。

2.7.2.1 for 循环

for 循环的句法是：

```
for (初值表达式; 测试表达式; 改变量表达式/步长) {
    语句块
}
```

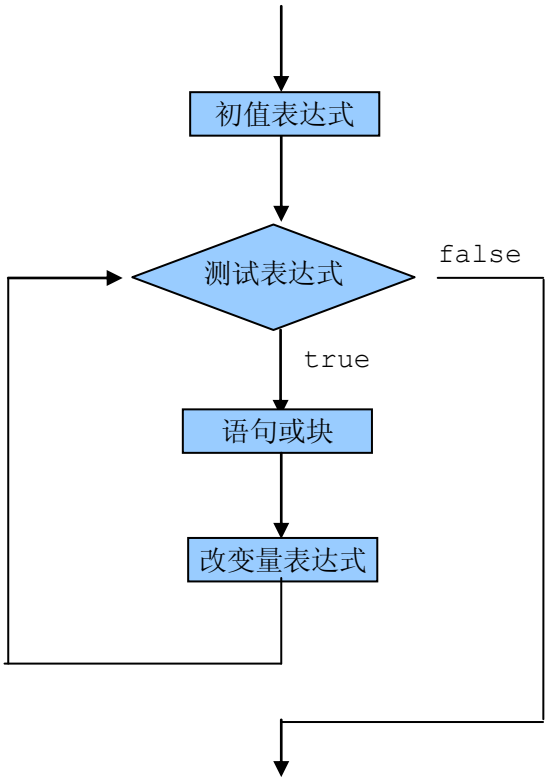
其执行顺序如下：

（1）首先运行初值表达式。

（2）然后计算测试表达式，如果表达式为true，执行语句或块；如果表达式为false，退出for循环。

（3）最后执行步长，第一次循环结束。

（4）第二次循环开始：首先判断测试表达式，转到第 2 步继续运行。



示例：

```
for (int i = 0; i < 10; i++) {
    System.out.println("Java快车");
}
System.out.println("Finally!");
```

注意：for 循环的执行顺序：先执行初始值表达式 **int** i = 0;再执行一遍测试表达式 i < 10; 如果测试表达式返回 true，则执行循环体，就是 System 的输出语句，如果测试表达式返回 false，则整个循环结束，然后执行增量表达式。我们称这是第一次循环结束了，初始值表达式只执行一次，第二次循环从测试表达式开始，方法和步骤和第一次一样。循环就这样一次一次地进行，直到测试表达式返回 false，整个循环就结束了，这个 for 语句的生命周期就结束了。

for 语句里面的 3 个部分都可以省略，但是我们不建议这么做。示例如下：

示例 1：3 个部分全部省略，就是一个无限循环。

```
public class Test {
    public static void main(String[] args) {
        for (;;) {
            System.out.println("Java快车");
        }
    }
}
```

示例 2：省略测试表达式和增量表达式部分，就是一个无限循环。

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

```
        for (int i = 0;;) {
            System.out.println("Java快车" + i);
        }
    }
}
```

示例 3：可以省略增量表达式部分，就是一个无限循环。

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

示例 4：可以省略增量表达式部分，就是一个无限循环。

```
public class Test {
    public static void main(String[] args) {
        for (int i = 0;; i++) {
            System.out.println("Java快车" + i);
        }
    }
}
```

示例 5：在 for 语句的括号里的表达式省略后，仍然能完成全部功能。

```
public class Test {
        public static void main(String[] args) {
            int i = 0;
            for (;; ) {
                if(i==10){
                    //如果条件成立，使用break跳出循环。
                    break;
                }
                System.out.println("Java快车" + i);
                i++;
            }
        }
}
```

for 循环的组合很多，要灵活运行，避免死循环。

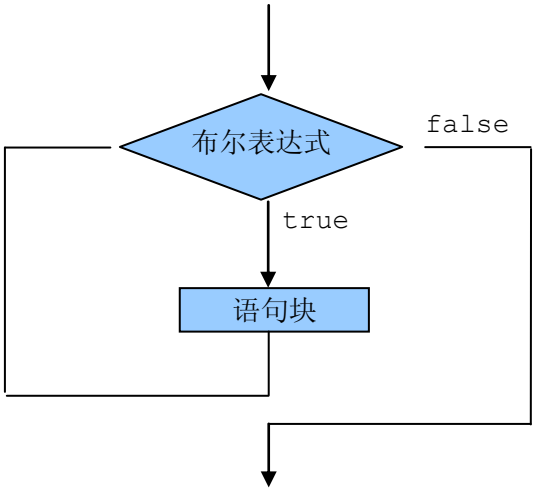
2.7.2.2 while 循环

while 循环的句法是：

```
while (布尔表达式) {
    语句块
```

```
}
```

说明：当布尔表达式为 true 时，执行语句或块，否则跳出 while 循环。



示例：

```
int i = 0;
while (i < 10) {
    System.out.println("javakc");
    i++;
}
System.out.println("Finally!");
```

请确认循环控制变量在循环体被开始执行之前已被正确初始化，并确认循环控制变量是真时，循环体才开始执行。控制变量必须被正确更新以防止死循环。

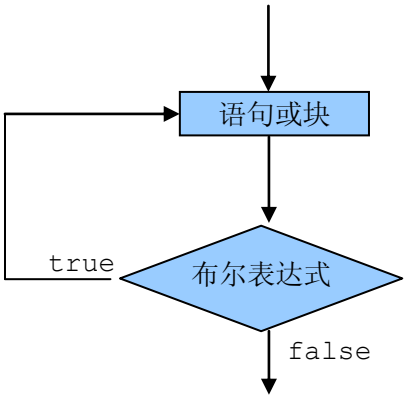
2.7.2.3 do-while 循环

do-while 循环的句法是：

```
do  {
    语句块;
} while (测试值表达式);
```

说明：

先执行语句或块，然后再判断布尔表达式。与 while 语句不同，当布尔表达式一次都不为 true 时，while 语句一开始判断就跳出循环，一次都不执行语句或块，而在 do 语句中则要执行一次。



示例：

```
int i = 0;
do {
    System.out.println("javakc");
    i++;
} while (i < 10);
System.out.println("Finally!");
```

像 while 循环一样，请确认循环控制变量在循环体中被正确初始化和测试并被适时更新。作为一种编程惯例，for 循环一般用在那种循环次数事先可确定的情况，而 while 和 do 用在那种循环次数事先不可确定的情况。

2.7.2.4 特殊循环流程控制

下列语句可被用在更深层次的控制循环语句中：

```
break  [label];
continue [label];
label:  语句; //这里必须是任意的合法的语句
```

break 语句被用来从 switch 语句、循环语句和预先给定了 label 的块中退出，跳出离 break 最近的循环。

continue 语句被用来略过并跳到循环体的结尾，终止本次循环，继续下一次循环。

label 可标识控制需要转换到的任何有效语句，它被用来标识循环构造的复合语句。当嵌套在几层循环中想退出循环时，正常的 break 只退出一重循环，你可以用标号标出你想退出哪一个语句。它类似以前被人诟病的“goto”语句，我们应该尽量避免使用。

例 1：break 的使用

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

```
        }
    }
}
```

运行的结果：i==0 一直到 i==4。因为当 i==5 的时候，执行 break，跳出 for 循环。

例 2：continue 的使用

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

运行的结果：i==0 一直到 i==4，就是不包括 i==3。因为当 i==3 的时候，执行 continue，终止本次循环，继续下一次循环。

例 3：label 的使用

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

运行的结果是：i==0 一直到 i==2。在这里看不出执行效果，如果是两层嵌套的循环，使用 label 就可以跳出外层的循环了。

例 4：label 的使用

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

运行的结果是：i==0 一直到 i==4

例 5：label 的使用

```
public class Test {
```

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

运行的结果是：i==0 i==1 i==2 i==4

2.7.2.5 循环的使用技巧：

1. 如果两个或两个以上的 for 嵌套使用，则执行循环次数多的放最里面，即执行次数由内到外布局，这样可以提高执行速度。
2. 变量的定义等应当尽量放在 for 外面，而不是放里面。
3. 知道循环次数时使用 for 循环，不知道循环次数时使用 while 循环。
4. 尽量使用 for 而不是 while:因为 for 初值，结束条件，循环增量都放在一起，看起来方便。

2.8 学习目标

1. 了解什么是关键字？
2. 了解学习 Java 中的关键字需要注意什么？
3. 了解什么是标识符？
4. 能够描述标识符的命名规则。
5. Java 中数据类型的分类。
6. Java 中有哪些基本数据类型。
7. 掌握 8 个基本数据类型的范围，并使用数据类型声明变量。
8. 深刻理解 Java 中的变量，变量的赋值，变量的存储。
9. 代码写出 Java 中的注释。
10. 学习 Java 中的运算符，能够说出每种运算符的运算法则，并能够代码示例。
11. 描述 if 语句的句法结构，能够说出 if 语句的运行过程，并能够灵活使用 if 语句和 if 语句的嵌套。
12. 描述 switch 语句的句法结构，能够说出 switch 语句的运行过程，并能够灵活使用 switch 语句，理解 break。
13. 描述 for 语句的句法结构，能够说出 for 语句的运行过程，并能够灵活使用 for 语句和 for 语句的嵌套。
14. 描述 while 语句的句法结构，能够说出 while 语句的运行过程，并能够灵活使用 while 语句。
15. 描述 do-while 语句的句法结构，能够说出 do-while 语句的运行过程，并能够灵活使用 do-while 语句。
16. 总结循环语句的本质。
17. 理解循环语句中的 break 和 continue，并代码示例。
18. 理解循环语句中的 Label，并代码示例。

2.9 练习

1. 叙述标识符的定义规则，指出下面的标识符中那些是不正确的，并说明理由
   here, _there, this, it, 2to1, _it

2. Java 中有那些基本数据类型？分别用什么符号来表示，各自的取值范围是多少？

3. 指出正确的表达式
   A. byte b=128;
   B. char c=65536;
   C. long len=0xfffL;
   D. double dd=0.9239d;

4. 下面哪几个语句将引起编译错？
   A. float f=2039.0;
   B. double d=2039.0;
   C. byte b=2039;
   D. char c=2039;

5. 描述短路现象。

6. 执行下列代码后的 x 和 y 的结果分别是什么？

```
int x,y,a=2;
x=a++;
y=++a;
```

7. 下面的程序输出结果是：a=6 b=5,请将程序补充完整。

```
public class  A{
    public  static  void  main(String args[]){
        int  a=5,b=6;
        a=_______;
        b=a-b;
        a=_______;
        System.out.println("a="+a+ "b="+b);
    }
}
```

8. 下面哪个语句序列没有错误，能够通过编译？

A.

```
int  i=0;
if(i){
    System.out.println("Hi");
}
```

B.

```
boolean  b=true;
boolean  b2=true;
if(b==b2)  {
   System.out.println("So  true");
}
```

C.

```
int  i=1;
int  j=2;
if(i==1|| j==2) System.out.println("OK");
```

D.

```
int  i=1;
int  j=2;
if  (i==1  &|  j==2) System.out.println("OK");
```

9. 阅读以下代码行：

```
boolean  a=false; boolean  b=true;
boolean  c=(a&&b)&&(!b);
int  result= c==false?1:2;
```

这段程序执行完后，c 与 result 的值是：

A. c=false; result=1;

B. c=true; result=2;

C. c=true; result=1;

D. c=false; result=2;

10.下列代码哪行会出错？

```
1)  public  void  modify()  {
2)    int  i,  j,  k;
3)    i  =  100;
4)    while  (  i  >  0  )  {
5)      j  =  i  *  2;
6)      System.out.println  ("  The  value  of  j  is  "  +  j  );
7)      k  =  k  +  1;
8)      i--;
9)    }
10)}
```

A 第 4 行 B 第 6 行 C 第 7 行 D 第 8 行

11.指出下列程序的运行结果。

```
int  i  =  9;
switch  (  i  )  {
   default:
      System.out.print("default");
   case  0:
```

```
            System.out.print("zero");
            break;
        case  1: System.out.print("one");
        case  2: System.out.print("two");
    }
```

A. default
B. defaultzero
C. 编译错
D. 没有任何输出

以下是编程题：

12. 将 1 到 1000 内的全部数字打印出来，数字之间用“,”分隔。

13. 声明两个 int 类型的变量 x、y，并将 62、97 分别赋值给 x、y，比较 x、y 的大小，输出其中的大者。

14. 将 1 到 1000 之间的奇数打印出来。

15. 判断一个数能否同时被 3 和 5 整除。

16. 输入三个数，找出最大一个数，并打印出来。

17. 给出一百分制成绩，要求输出成绩等级'A','B','C','D','E'。90 分以上为'A',80~89 分为'B',70~79 分为'C',60~69 分为'D',60 分以下为'E'。

```
请输入成绩：67
D

请输入成绩：89
B

请输入成绩：56
D
```

18. 设计一个程序，输入一个数字（0~6），用中文显示星期几。

```
请输入数字：6
星期日

请输入数字：2
星期三

请输入数字：8
错误
```

19. 写一个程序，要求输入一个数字，数字中包含 5 个数位。把数字分解成单独的数位，并打印每一个数位。例如，假定用于键入 43263 这个数字，那么程序应打印结果：4 3 2 6 3。

20. 某公司计划提高员工工资，若基本工资大于等于 3000 元，增加工资 20%；若小于 3000 元大于等于 2000 元，则增加工资 15%；若小于 2000 元则增加工资 10%。请根据用户输入的基本工资，计算出增加后的工资。

```
请输入员工工资：3600
现在的工资是：4320 元

请输入员工工资：2800
现在的工资是：3220 元

请输入员工工资：1700
现在的工资是：1870 元
```

21. 设 s=1*2*3*4*5*……*n，求 s 不大于 400000 时最大的 n。

22. 编写程序，在控制台输出如下图案：

```
*
**
***
****
*
**
***
****
```

23. 编写一个程序，说明 while 和 do/while 的区别

24. 使用 for 语句打印显示下列数字形式：n=4

```
1
1  2  1
1  2  3  2  1
1  2  3  4  3  2  1
```

25. 编程实现求一个十进制数字的二进制形式。

26. 每位司机都关心车辆的油耗情况。有位司机记录了自己行使的公里数，以及每次加油多少升。请设计一个程序，要求输入行使的英里数，以及每次加了多少升汽油。程序应计算并显示每次加油后，每升汽油可供行驶多少公里。程序还应综合所有的输入，计算并

输出每升汽油可以供行驶多少公里。输出结果如下：

```
请输入加油量：30
请输入公里数：300
结果：每升油行使 10 公里
平均：每升油行使 10 公里

请输入加油量：20
请输入公里数：210
结果：每升油行使 10.5 公里
平均：每升油行使 10.2 公里

请输入加油量：10
请输入公里数：110
结果：每升油行使 11 公里
平均：每升油行使 10.33 公里
```

27. 一家大型化工厂采用佣金方式为推销员父薪酬。推销员每月领到基本工资 900 元，再加上一周销售毛利的 9%。例如，一名推销员在某一周销售了毛利为 5000 元的化工产品，那么除了领取固定的 900 元之外，还要加上 5000 元的 9%，总计 1350 元。开发一个程序，用于输入推销员上一周的毛利，然后计算并显示那名推销员的收入。

```
请输入推销员上一周的毛利：6000
推销员的收入是：1440 元

请输入推销员上一周的毛利：7000
推销员的收入是：1530 元

请输入推销员上一周的毛利：8000
推销员的收入是：1620 元
```

28. 开发一个程序，计算每名员工的薪水。公司规定，每周每名员工在其工作的前 40 个小时内，每小时都领取固定工资 60 元。超出 40 小时后，每工作一小时，算一个半小时。你的程序根据输入的小时数，计算并显示员工上一周的薪水总额。

```
员工上周工作总时间是：45
员工上周的薪水是：2850 元

员工上周工作总时间是：50
员工上周的薪水是：3300 元

员工上周工作总时间是：55
员工上周的薪水是：3750 元
```

29. 企业发放的奖金根据利润提成。利润(I)低于或等于 10 万元时，奖金可提 10%；利润高于 10 万元，低于 20 万元时，低于 10 万元的部分按 10%提成，高于 10 万元的部分，可可提成 7.5%；20 万到 40 万之间时，高于 20 万元的部分，可提成 5%；40 万到 60 万之间时高于 40 万元的部分，可提成 3%；60 万到 100 万之间时，高于 60 万元的部分，可提成 1.5%，高于 100 万元时，超过 100 万元的部分按 1%提成，从键盘输入当月利润 I，求应发放奖金总数？

3 程序的灵魂-算法

一个程序包括以下两个方面的内容：

(1)对数据的描述。在程序中要指定数据的类型和数据的组织形式，即数据结构。
(2)对操作的描述。即操作步骤，也就是算法。

数据是操作的对象，操作的目的是对数据进行加工处理，以得到期望的结果。作为程序设计人员，必须认真考虑和设计数据结构和操作步骤（即算法）。著名计算机科学家沃思提出一个公式：**数据结构+算法=程序**。

实际上，一个程序除了以上两个主要要素之外，还应当采用结构化程序设计方法进行程序设计，并且用某一种计算机语言表示。因此，算法、数据结构、程序设计方法和语言工具 4 个方面是一个程序设计人员所应具备的知识。

3.1 算法的概念

广义地说，**为解决一个问题而采取的方法和步骤，就称为算法**(algorithm)。例如，描述太极拳动作的图解，就是太极拳的算法。一首歌曲的乐谱，也可以称为该歌曲的算法，因为它指定了演奏该歌曲的每一个步骤，按照它的规定就能演奏出预定的曲子。

对同一个问题，可以有不同的解题方法和步骤。

计算机算法可分为两大类别：

数值运算算法
非数值运算算法

3.2 简单算法举例

[例 3.1]求 1×2×3×4×5

算法 1：

步骤 1：先求 1*2，得到结果 2。
步骤 2：将步骤 1 得到的乘积 2 再乘以 3，得到结果 6。
步骤 3：将 6 再乘以 4，得 24。
步骤 4：将 24 再乘以 5，得 120。

算法 2：

S_1：使t=1
S_2：使i=2
S_3：使t*i，乘积仍放在变量t中，可表示为t=t*i
S_4：使i的值加 1，即i=i+1
S_5：如果i不大于 5，返回重新执行步骤S_3以及其后的步骤S_4和S_5；否则，算法结束。最后得t的值就是 5！的值。

[例 3.2]有 50 个学生，要求将他们之中成绩在 80 分以上的学生的学号和成绩输出。

n_i 代表第 i 个学生学号。g_i 代表第 i 个学生成绩，算法如下：

S_1：i=1
S_2：如果g_i>80，则输出n_i和g_i；否则不输出
S_3：i=i+1
S_4：如果i<=50，返回S_2，继续执行；否则，算法结束。

[例 3.3]判定 2000-2500 年中的每一年是否是闰年

闰年的条件是：
能被 4 整除，但不能被 100 整除的年份是闰年，如 1996 年、2004 年；
能被 400 整除的年份是闰年，如 1600 年、2000 年。

设 y 为被检测的年份。可采用以下步骤：

S_1：y=2000
S_2：若y不能被 4 整除，则输出“y不是闰年”。然后转到S_6
S_3：若y能被 4 整除，不能被 100 整除，则输出“y是闰年”。然后转到S_6
S_4：若y能被 400 整除，输出“y是闰年”，然后转到S_6
S_5：输出“y不是闰年”
S_6：y=y+1
S_7：当y<=2500 时，转S_2继续执行，否则算法停止。

[例 3.4]对一个大于或等于 3 的正整数，判断它是不是一个素数

判断一个数 n(n>=3)是否素数的方法是将 n 作为被除数，将 2 到 n-1 各个整数先后作为除数，如果都不能被整除，则 n 为素数。算法如下：

S_1：输入n的值
S_2：i=2（i作为除数）
S_3：n被i除，得余数r
S_4：如果r=0，表示n能被i整除，则输出“n不是素数”，算法结束；否则执行S_5
S_5：i=i+1
S_6：如果i<=n-1，返回S_3；否则输出“n是素数”，然后结束。

3.3 算法的特性

（1）有穷性。一个算法应包含有限的操作步骤，而不能是无限的。

（2）确定性。算法中的每一个步骤都应当是确定的，而不应当是含糊的、模棱两可的。

（3）有零个或多个输入。所谓输入是指在执行算法时需要从外界取得必要的信息。

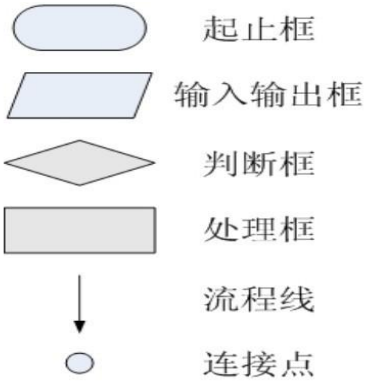
（4）有一个或多个输出。算法的目的是为了求解，“解”就是输出。

（5）有效性。算法中的每一个步骤都应当能有效地执行，并得到确定的结果。

3.4 表示一个算法

为了表示一个算法，可以用不同的方法。常用的方法有：

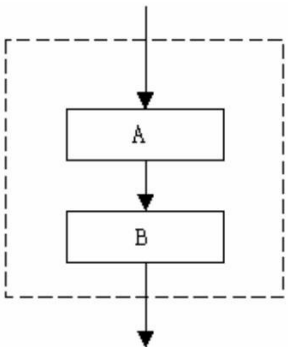
自然语言
传统流程图
N-S流程图
伪代码
PAD图



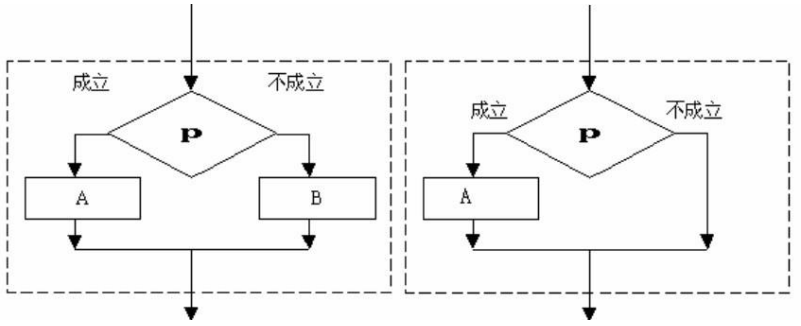
3.4.1 用流程图表示算法

3.4.2 三种基本结构

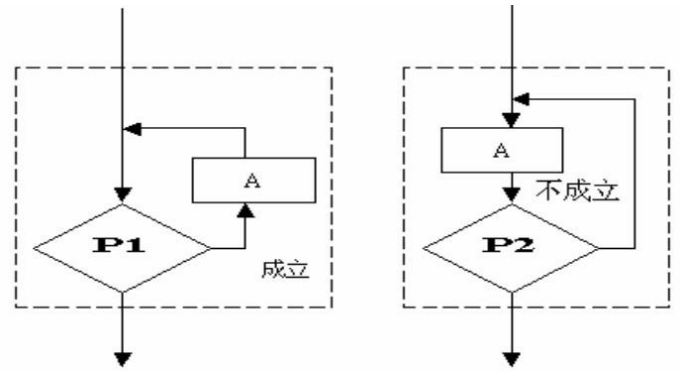
顺序结构：



选择结构：

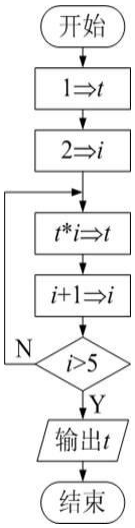


循环结构：

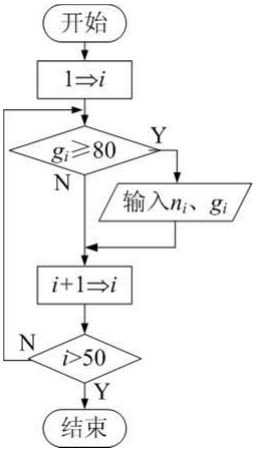


3.4.3 示例

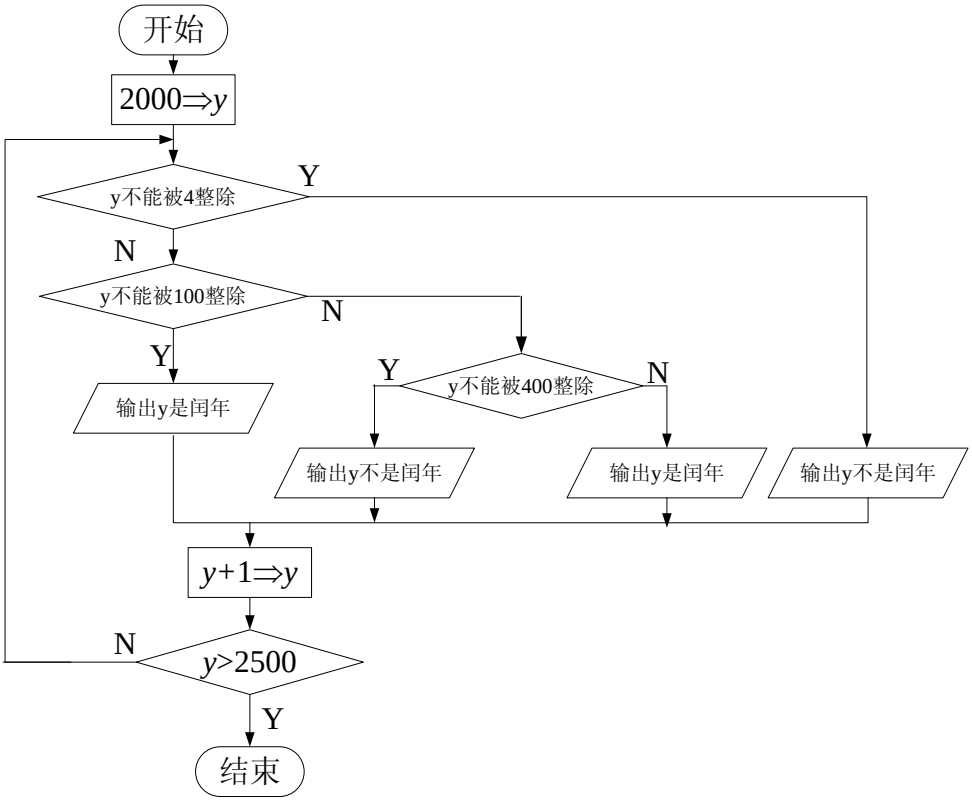
将[例 3.1]求 5!的算法用流程图表示：



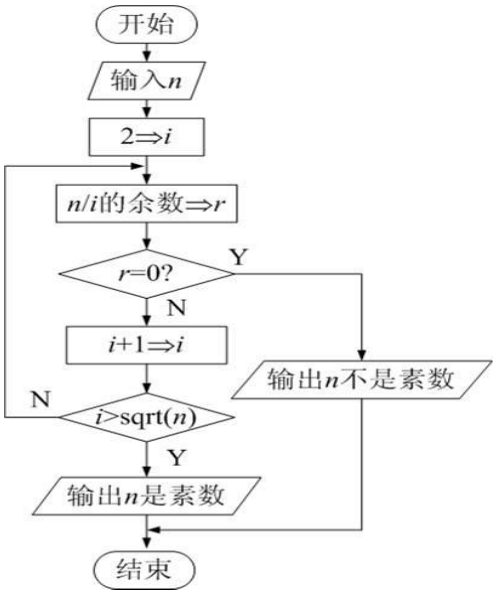
将[例 3.2]的算用流程图表示。将 50 名学生中成绩在 80 分以上者的学号和成绩输出：



将[例 3.3]判定闰年的算法用流程图表示：



将[例 3.4]判断素数的算法用流程图表示：



3.5 用计算机语言表示算法

[例 3.1]求 1×2×3×4×5

```
int t=1;
for(int i=2;i<=5;i++){
    t*=i;
}
System.out.println("5!="+t);
```

[例 3.3]判定 2000-2500 年中的每一年是否是闰年

```
int y=2000;
String result="平年";
if(y%4==0){
    if(y%100!=0){
        result="闰年";
    }else{
        if(y%400==0){
            result="闰年";
        }
    }
}
System.out.println(y+"年是"+result);
```

[例 3.4]对一个大于或等于 3 的正整数，判断它是不是一个素数

```
int n=23;
int i=2;
String result="是素数";
while(i<=Math.sqrt(n)){
```

```
        if(n%i==0){
            result="不是素数";
            break;
        }
        i++;
    }
    System.out.println(n+result);
```

4 Java 类和对象

4.1 理解面向对象

4.1.1 为什么要面向对象

传统的开发方法是面向过程的，面向过程是一种以事件为中心的编程思想。就是分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候一个一个依次调用就可以了。

面向过程其实是最为实际的一种思考方式，就是算面向对象的方法也是含有面向过程的思想，可以说面向过程是一种基础的方法，它考虑的是实际的实现。一般的面向过程是从上往下步步求精，当程序规模不是很大时，面向过程的方法还会体现出一种优势，因为程序的流程会很清楚。

比如拿学生早上做的事情来说说这种面向过程，粗略的可以将过程拟为：

```
(1)起床
(2)穿衣
(3)洗脸刷牙
(4)去学校
```

这 4 步就是一步一步的完成，它的顺序很重要，你只须一个一个的实现就行了。

面向过程开发具有如下缺点：

软件重用性差

重用性是指同一事物不经修改或稍加修改就可多次重复使用的性质。软件重用性是软件工程追求的目标之一。

软件可维护性差

软件工程强调软件的可维护性，强调文档资料的重要性，规定最终的软件产品应该由完整、一致的配置成分组成。在软件开发过程中，始终强调软件的可读性、可修改性和可测试性是软件的重要的质量指标。实践证明，用传统方法开发出来的软件，维护时其费用和成本仍然很高，其原因是可修改性差，维护困难，导致可维护性差。

开发出的软件不能满足用户需要

用传统的结构化方法开发大型软件系统涉及各种不同领域的知识，在开发需求模糊或需求动态变化的系统时，所开发出的软件系统往往不能真正满足用户的需要，用户需求的变化往往造成系统结构的较大变化，从而需要花费很大代价才能实现这种变化

面向对象(Object-Oriented,简称 OO)是一种以事物为中心的编程思想。

对象是真实世界中的事物在人脑中的映象。在实际生活中，我们每时每刻都与“对象”在打交道，我们用的钢笔，骑的自行车，乘坐的公共汽车等都是对象。这些对象是能看得见、摸得着，

实际存在的东西，我们称之为实体对象；有的对象是针对非具体物体的，但是在逻辑关系上的反映，比如：钢笔与墨水的关系，人与自行车的关系，我们称之为逻辑对象。

如果是用面向对象的方法来模拟学生早上做的事情。可以抽象出一个学生的类，它包括四个方法，但是具体的顺序就不能体现出来。根据类创建一个对象，再调用对象的方法，在调用方法时体现出顺序。

```
张三起床
张三穿衣服
张三洗脸刷牙
张三去学校
```

基于对象的编程更符合人的思维模式，编写的程序更加健壮和强大，更重要的是，面向对象编程鼓励创造性的程序设计。

4.1.2 对象的基本构成

现实中的人是一个实体对象，分析实体对象的构成，发现有这样一些共同点，这些实体对象都有自己的状态描述，比如：人有姓名、身高、体重、发型、着装等，有了这些描述，我们可以想象出一个人的样子。我们把这些描述称之为属性，属性是静态的，这些属性用来决定了对象的具体表现形式。

除了这些静态的，实体对象还有自己的动作，通过这些动作能够完成一定的功能，我们称之为方法，比如：人能写字，能刷牙，能跑步，打篮球，踢足球等。我们知道了对象的方法，也就知道了这个对象可以做什么，有什么用。

依照这个理论我们再分析一下汽车，首先想到的是静态的属性，有颜色、车牌号、标志、发动机的功率、乘载人数、自重、轮子数目等等。然后是动态的功能：加速、减速、刹车、转弯等等。

总之一句话，对象同时具备**静态的属性**和**动态的功能**。

4.1.3 如何进行对象抽象

抽象是在思想上把各种对象或现象之间的共同的本质属性抽取出来,而舍去个别的非本质的属性的思维方法。也就是说把一系列相同或类似的实体对象的特点抽取出来，采用一个统一的表达方式，这就是抽象。

```
比如：
张三这个人身高 180cm，体重 75kg，会打篮球，会跑步
李四这个人身高 170cm，体重 70kg，会打篮球，会跑步
```

现在想要采用一个统一的类来描述张三和李四，那么我们就可以采用如下的表述方法来表述：

```
人{
    静态属性：
        姓名；
        身高；
```

```
        体重;
    动态动作:
        打篮球();
        跑步();
}
```

4.1.4 Java 中的类和对象

4.1.4.1 Java 中的类

把抽象出来的类型使用 Java 表达出来，那就是类 class。

类是对具有相似性质的一类事物的抽象，类封装了一类对象的属性和方法。实例化一个类，可以获得属于该类的一个实例（即对象）。

类在 Java 编程语言中作为定义新类型的一种途径，类是组成 Java 程序的基本要素。类声明可定义新类型并描述这些类型是如何实现的。

比如前面讨论过的“人”、“汽车”，使用 Java 表达出来就是一个类。

4.1.4.2 Java 中的对象

对象是类的一个实例，类的具体化，也称实例对象。实例就是实际例子，就好像真实存在一样。

类可被认为是一个模板---你正在描述的一个对象模型。一个对象就是你每次使用的时候创建的一个类的实例的结果。

比如前面讨论的张三和李四，他们就是通过“人”这个类的创建出来的实例，这样就在计算机内存中，把现实中的张三、李四表达出来了，就像张三、李四已经活在计算机的内存中了。

4.2 Java 类的基本构成

一个完整的 Java 类通常由下面六个部分组成：

```
包定义语句
import 语句
类定义{
    成员变量
    构造方法
    成员方法
}
```

其中：只有类定义和“{}”是不可或缺的，其余部分都可以根据需要来定义。 下面分别来学

习各个部分的基本规则，看看如何写 Java 的类，建议初学者，先看类、成员变量、方法部分，再看包、import 部分。

4.2.1 包

4.2.1.1 什么是包

计算机操作系统使用文件夹或者目录来存放相关或者同类的文档，在 Java 编程语言中，提供了一个包的概念来组织相关的类。包在物理上就是一个文件夹，逻辑上代表一个分类概念。

包是类、接口或其它包的集合，包对类进行有效管理的机制。

包对于下列工作非常有用：

- 包将包含类代码的文件组织起来，易于查找和使用适当的类。
- 包不止是包含类和接口，还能够包含其它包。形成层次的包空间。
- 有助于避免命名冲突。当使用很多类时，确保类和方法名称的唯一性是非常困难的。包能够形成层次命名空间，缩小了名称冲突的范围，易于管理名称。
- 控制代码访问权限。

为便于管理数目众多的类，Java 语言中引入了“包”的概念，可以说是对定义的 Java 类进行“分组”，将多个功能相关的类定义到一个“包”中，以解决命名冲突、引用不方便、安全性等问题。

比如户籍制度，每个公民除有自己的名字“张三”、“李四”外还被规定了他的户籍地。假定有两个人都叫张三，只称呼名字就无法区分他们，但如果事先登记他们的户籍分别在北京和上海，就可以很容易的用“北京的张三”、“上海的张三”将他们区分开来。如果北京市仍有多个张三，还可以细分为“北京市.海淀区的张三”、“北京市.西城区.平安大街 的张三”等等，直到能惟一标识每个“张三”为止。

4.2.1.2 JDK 中常用的包

JDK 中定义的类就采用了“包”机制进行层次式管理，简而言之：从逻辑上讲，包是一组相关类的集合；从物理上讲，同包即同目录。

java.lang----包含一些 Java 语言的核心类，包含构成 Java 语言设计基础的类。在此包中 定义的最重要的一个类是“Object”，代表类层次的根，Java 是一个单根系统，最终的根就是 Object。

java.awt----包含了构成抽象窗口工具集（abstract window toolkits）的多个类，这些类被用来构建和管理应用程序的图形用户界面(GUI)。

javax.swing----完全 Java 版的图形用户界面(GUI)解决方案，提供了很多完备的组件，可以应对复杂的桌面系统构建。

java.net----包含执行与网络相关的操作的类，如 URL, Socket, ServerSocket 等。

java.io----包含能提供多种输入/输出功能的类。

java.util----包含一些实用工具类，如定义系统特性、使用与日期日历相关的方法。还有重要的集合框架。

4.2.1.3 代码中如何表达包

Java 语言使用 package 语句来实现包的定义。package 语句必须作为 Java 源文件的非注释语句第一条语句，指明该文件中定义的类所在的包。若缺省该语句，则指定为无名包，其语法格式为：

```
package pkg1[.pkg2[.pkg3…]]; //“[]”表示可选
```

Java 编译器把包对应于文件系统的目录管理，因此包也可以嵌套使用，即一个包中可以含有类的定义也可以含有子包，其嵌套层数没有限制。package 语句中，用“.”来指明包的层次。

程序中 package 的使用：Test.java

```
package p1;
public class Test {
    public void display() {
       System.out.println("in  method display()");
    }
}
```

Java 语言要求包声明的层次和实际保存类的字节码文件的目录结构存在对应关系，以便将来使用该类时能通过包名（也就是目录名）查找到所需要的类文件。简单地说就是包的层次结构需要和文件夹的层次对应。

注意：每个源文件只有一个包的声明，而且包名应该全部小写。

4.2.1.4 编译和生成包

如果在程序 Test.java 中已定义了包 p1，就必须将编译生成的字节码文件 Test.class 保存在与包名同名的子目录中，可以选用下述两种方式之一：

直接编译：

```
javac  Test.java
```

则编译器会在当前目录下生成 Test.class 文件，再在手动创建一个名为 p1 的文件夹，将 Test.class 复制到该 p1 目录下。

带包编译：

带包编译是简化的编译命令。

```
javac -d .\  Test.java
```

编译器会自动在当前目录下建立一个子目录 p1，并将生成的.class 文件自动保存到 p1 文件夹下。

```
javac -d .\test\  Test.java
```

编译器会自动在当前目录下的 test 文件夹中建立一个子目录 p1，并将生成的.class 文件自动保存到 p1 文件夹下。前提是当前目录下必须有 test 文件夹，否则报错：

“未找到目录：.\test\”

```
javac -d D:\test\  Test.java
```

编译器会自动在 D 盘根目录下的 test 文件夹中建立一个子目录 p1，并将生成的.class 文件自动保存到 p1 文件夹下。前提是 D 盘根目录下必须有 test 文件夹，否则报错：

“未找到目录：D:\test\”

4.2.1.5 带包运行

运行带包的程序，需要使用类的全路径，也就是带包的路径，比如上面的那个程序，就使用如下的代码进行运行：

```
java p1.Test
```

4.2.2 import 语句

4.2.2.1 用法

为了能够使用某一个包的成员，我们需要在 Java 程序中明确导入该包。使用“import”语句可完成此功能。在 java 源文件中 import 语句应位于 package 语句之后，所有类的定义之前，可以没有，也可以有多条，其语法格式为：

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

java 运行时环境将到 CLASSPATH + package1.[package2…]路径下寻找并载入相应的字节码文件 classname.class。“*”号为通配符，代表所有的类。也就是说 import 语句为编译器指明了寻找类的途径。

例，使用 import 语句引入类程序：TestPackage.java

```
import p1.Test;
//或者 import p1.*;
public class TestPackage {
    public static void main(String args[]) {
       Test t = new Test(); // Test 类在 p1 包中定义
       t.display();
    }
}
```

java 编译器默认为所有的 java 程序引入了 JDK 的 java.lang 包中所有的类（import

java.lang.*;），其中定义了一些常用类：System、String、Object、Math 等。因此我们可以直 接使用这些类而不必显式引入。但使用其它非无名包中的类则必须先引入、后使用。

4.2.2.2 Java 类搜寻方式

程序中的 import 语句标明要引入 p1 包中的 Test 类，假定环境变量 CLASSPATH 的值为“.;C:\jdk6\lib;D:\ex”，java 运行环境将依次到下述可能的位置寻找并载入该字节码文件 Test.class：

```
.\p1\Test.class
 C:\jdk6\lib\p1\Test.class
 D:\ex\p1\Test.class
```

“.”代表当前路径，如果在第一个路径下就找到了所需的类文件，则停止搜索。否则依次搜索后续路径，如果在所有的路径中都未找到所需的类文件，则编译或运行出错。

4.2.3 访问修饰符

Java 语言允许对类中定义的各种属性和方法进行访问控制，即规定不同的保护等级来限制对它们的使用。为什么要这样做？Java 语言引入类似访问控制机制的目的在于实现信息的封装和隐藏。Java 语言为对类中的属性和方法进行有效地访问控制，将它们分为四个等级：private、无修饰符、protected、public，具体规则如下：

可否直接访问 / 控制等级	同一个类中	同一个包中	不同包中的子类的对象	任何场合
private	Yes			
无修饰符	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes

变量和方法可以使用四个访问级别中的任意一个修饰，类可以使用公共或无修饰级别修饰。

变量、方法或类有缺省（无修饰符）访问性，如果它没有显式受保护修饰符作为它的声明的一部分的话。这种访问性意味着，访问可以来自任何方法，当然这些方法只能在作为对象的同一个包中的成员类当中。

以修饰符 protected 标记的变量或方法实际上比以缺省访问控制标记的更易访问。一个 protected 方法或变量可以从同一个包中的类当中的任何方法进行访问，也可以是从任何子类中的任何方法进行访问。当它适合于一个类的子类但不是不相关的类时，就可以使用这种受保护访问来访问成员。

4.2.4 类定义

Java 程序的基本单位是类，你建立类之后，就可用它来建立许多你需要的对象。Java 把每一个可执行的成分都变成类。

类的定义形式如下：

```
<权限修饰符> [一般修饰符] class <类名> [extends 父类][implements 接口]{
        [<属性定义>]
    [<构造方法定义>]
    [<方法定义>]
}
```

这里，类名要是合法的标识符。在类定义的开始与结束处必须使用花括号。你也许想建立一个矩形类，那么可以用如下代码：

```
public class Rectangle{
    ......//矩形具体的属性和方法
}
```

4.2.5 构造方法

什么是构造方法

类有一个特殊的成员方法叫作构造方法，它的作用是创建对象并初始化成员变量。在创建对象时，会自动调用类的构造方法。

构造方法定义规则

Java 中的构造方法**必须与该类具有相同的名字，并且没有方法的返回类型（包括没有 void）**。另外，构造方法一般都应用 public 类型来说明，这样才能在程序任意的位置创建类的实例——对象。

下面是一个 Rectangle 类的构造方法，它带有两个参数，分别表示矩形的长和宽：

```
public class Rectangle {
    int width;
    int height;
    public Rectangle(int w, int h) {
       width = w;
       height = h;
    }
    public Rectangle() {
    }
}
```

每个类至少有一个构造方法。如果不写一个构造方法，**Java 编程语言将提供一个默认的，**

该构造方法没有参数，而且方法体为空。

注意：**如果一个类中已经定义了构造方法则系统不再提供默认的构造方法。**

4.2.6 析构方法

当垃圾回收器将要释放无用对象的内存时，先调用该对象的 finalize()方法。如果在程序终止前垃圾回收器始终没有执行垃圾回收操作，那么垃圾回收器将始终不会调用无用对象的 finalize()方法。在 Java 的 Object 基类中提供了 protected 类型的 finalize()方法，因此任何 Java 类都可以覆盖 finalize()方法，通常，在析构方法中进行释放对象占用的相关资源的操作。

Java 虚拟机的垃圾回收操作对程序完全是透明的，因此程序无法预料某个无用对象的 finalize()方法何时被调用。如果一个程序只占用少量内存，没有造成严重的内存需求，垃圾回收器可能没有释放那些无用对象占用的内存，因此这些对象的 finalize()方法还没有被调用，程序就终止了。

程序即使显式调用 System.gc()或 Runtime.gc()方法，也不能保证垃圾回收操作一定执行，也就不能保证对象的 finalize()方法一定被调用。

当垃圾回收器在执行 finalize()方法的时候，如果出现了异常，垃圾回收器不会报告异常，程序继续正常运行。

```
@Override
protected void finalize(){
    System.out.println("in finalize");
}
```

在 Java 编程里面，一般不需要我们去写析构方法，这里只是了解一下就可以了。

4.2.7 成员变量

成员变量是指类的一些属性定义，标志类的静态特征，它的基本格式如下：

```
<权限修饰符> [一般修饰符] 类型 <属性名称> [=初始值];
```

访问修饰符：可以使用四种不同的访问修饰符中的一种，包括 public(公共的)、protected（受保护的），无修饰符和 private（私有的）。public 访问修饰符表示属性可以从任何其它代码调用。private 表示属性只可以由该类中的其它方法来调用。protected 将 在以后的课程中讨论。

修饰符：是对属性特性的描述，例如后面会学习到的：static、final 等等。

类型：属性的数据类型，可以是任意的类型。

属性名称：任何合法标识符

初始值：赋值给属性的初始值。如果不设置，那么会自动进行初始化，基本类型使用缺省值，对象类型自动初始化为 null。

成员变量有时候也被称为属性、实例变量。

4.2.8 方法

4.2.8.1 定义

方法就是对象所具有的动态功能。Java 类中方法的声明采用以下格式：

```
<权限修饰符> [修饰符] 返回值类型 <方法名称> (参数列表) [throws 异常列表]{
[方法体]
}
```

访问修饰符： 可以使用四种不同的访问修饰符中的一种，包括 public、 protected、无修饰符和 private。public 访问修饰符表示方法可以从任何其它代码调用。private 表示方法只可以由该类中的其它方法来调用。protected 将在以后的课程中讨论。

修饰符：是对方法特性的描述，例如后面会学习到的：static、final、abstract、synchronized 等等。

返回值类型：表示方法返回值的类型。如果方法不返回任何值，它必须声明为 void(空)。Java 技术对返回值是很严格的，例如，如果声明某方法返回一个 int 值，那么方法必须从所有可能的返回路径中返回一个 int 值（只能在等待返回该 int 值的上下文中被调用。）

方法名称：可以是任何合法标识符，并带有用已经使用的名称为基础的某些限制条件。

参数列表：允许将参数值传递到方法中。列举的元素由逗号分开，而每一个元素包含一个类型和一个标识符。在下面的方法中只有一个形式参数，用 int 类型和标识符 days 来声明：
public void test(int days){}

throws 异常列表：子句导致一个运行时错误（异常）被报告到调用的方法中，以便以合适的方式处理它。异常在后面的课程中介绍。

花括号内是方法体，即方法的具体语句序列。

4.2.8.2 示例

比如现在有一个“车”的类—Car，“车”具有一些基本的属性，比如四个轮子，一个方向盘，车的品牌等等。当然，车也具有自己的功能，也就是方法，比如车能够“开动”—run。要想车子能够开动，需要给车子添加汽油，也就是说，需要为 run 方法传递一些参数“油”进去。车子就可以跑起来，这些油可以供行驶多少公里？就需要 run 方法具有返回值“行驶里程数”。

```
package com.javakc;
public class Car {// 车这个类
```

```
    private String make;// 一个车的品牌
    private int tyre;// 一个车具有轮胎的个数
    private int wheel;// 一个车具有方向盘的个数
    public Car() {
       // 初始化属性
       make = "BMW";// 车的品牌是宝马
       tyre = 4;// 一个车具有 4 个轮胎
       wheel = 1;// 一个车具有一个方向盘
    }
    /**
     * 车这个对象所具有的功能，能够开动
     *
     * @param oil 为车辆加汽油的数量
     * @return 车辆行驶的公里数
     */
    public double run(int oil) {
       // 进行具体的功能处理
       return 100*oil/8;
    }
    public static void main(String[] args){
        Car c=new Car();
        double mileage=c.run(100);
        System.out.println("行驶了 "+mileage+" 公里");
    }
}
```

main 方法是一个特殊的方法，如果按照 **public static void** main(String[] args) 的格式写，它就是一个类的入口方法，也叫主函数。当这个类被 java 指令执行的时候，首先执行的是 main 方法，如果一个类没有入口方法，就不能使用 java 指令执行它，但可以通过其他的方法调用它。

4.2.8.3 形参和实参

对于方法的参数的理解，分为形参和实参：

形参：就是形式参数的意思。是在定义方法的时候使用的参数，用来标识方法接收的参数类型，在调用该方法时传入。

实参：就是实际参数的意思。是在调用方法时传递给该方法的实际参数。

比如：上面的例子中“int oil”就是个形式参数，这里只是表示需要加入汽油，这个方法才能正常运行，但具体加入多少，要到真正使用的时候，也就是调用这个方法的时候才具体确定，加入调用的时候传入“100”，这就是个实际参数。

形参和实参有如下基本规则：

形参和实参的类型必须要一致，或者要符合隐含转换规则。

形参类型不是引用类型时，在调用该方法时，是**按值传递**的。在该方法运行时， 形参和实参是不同的变量，它们在内存中位于不同的位置，形参将实参的值复制一份，在该方法运行结束的时候形参被释放，而实参内容不会改变。

形参类型是引用类型时，在调用该方法时，是**按引用传递**的。运行时，传给方法的是实参的地址，在方法体内部使用的也是实参的地址，即使用的就是实参本身对应的内存空间。所以在函数体内部可以改变实参的值。

4.2.8.4 参数可变的方法

从 JDK5.0 开始，提供了参数可变的方法。

当不能确定一个方法的入口参数的个数时，5.0 以前版本的 Java 中，通常的做法是将多个参数放在一个数组或者对象集合中作为参数来传递，5.0 版本以前的写法是：

```
Int sum(Integer[] numbers){…}
//在另一个方法中调用该方法
sum(new Integer[] {12,13,20});
```

而在 5.0 版本中可以写为：

```
//注意：方法定义中是三个点
int  sum(Integer...  numbers){
    //方法内的操作
}
//调用该方法
sum(12,13,20);//正确
sum(10,11);  //正确
```

也就是说，传入参数的个数并不确定。但请注意：传入参数的类型必须是一致的，究其本质，就是一个数组。

显然，JDK5.0 版本的写法更为简易，也更为直观，尤其是方法的调用语句，不仅简化很多，而且更符合通常的思维方式，更易于理解。

4.3 如何使用一个 Java 类

前面学习了如何定义一个类，下面来学习如何使用一个类。

4.3.1 new 关键字

假如定义了一个表示日期的类，有三个整数变量；日、月和年的意义即由这些整数变量给出。如下所示：

```
class MyDate {
    int day;
    int month;
```

```
    int year;
   public String toString() {
      int num=0;
      return day+","+month+","+year;
   }
}
```

名称 MyDate 按照类声明的大小写约定处理，而不是由语意要求来定。

在可以使用变量之前，实际内存必须被分配。这个工作是通过使用关键字 new 来实现的。如下所示：

```
MyDate  myBirth;
myBirth = new MyDate()
```

第一个语句(声明)仅为引用分配了足够的空间，而第二个语句则通过调用对象的构造方法为构成 MyDate 的三个整数分配了空间。对象的赋值使变量 myBirth 重新正确地引用新的对象。这两个操作被完成后，MyDate 对象的内容则可通过 myBirth 进行访问。

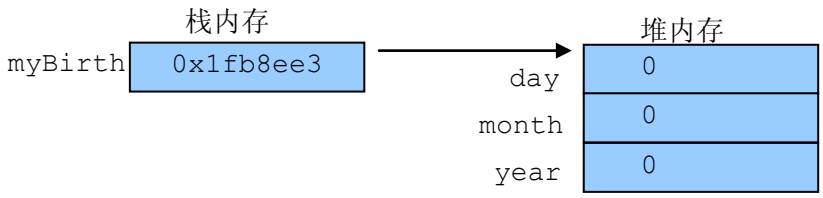
关键字 new 意味着内存的分配和初始化，new 调用的方法就是类的构造方法。

假使定义任意一个 class XXXX，可以调用 new XXXX()来创建任意多的对象，对象之间是分隔的。就像有一个汽车的类，可以使用 new 关键字创建多个具体的对象，比如有红旗的汽车、奇瑞的汽车、大众的汽车等等，它们都是独立的单元，相互之间是隔离的。

一个对象的引用可被存储在一个变量里，因而一个变量点成员(如 myBirth.day)可用来访问每个对象的单个成员。请注意在没有对象引用的情况下，仍有可能使用对象，这样的对象称作“匿名”对象。

使用一个语句同时为引用 myBirth 和由引用 myBirth 所指的对象分配空间也是可能的。

```
MyDate  myBirth  =  new  MyDate();
```



4.3.2 使用对象中的属性和方法

要调用对象中的属性和方法，使用“.”操作符。

对象创建以后就有了自己的属性，通过使用“.”操作符实现对其属性的访问。例如：

```
myBirth.day  =  26;
myBirth.month  =  7;
myBirth.year  =  2000;
```

对象创建以后，通过使用“.”操作符实现对其方法的调用，方法中的局部变量被分配内存空间，方法执行完毕，局部变量即刻释放内存。例如：

```
myBirth.toString();
```

4.3.3 this 关键字

关键字 this 是用来指向当前对象或类实例的，功能说明如下：

4.3.3.1 点取成员

this.day 指的是调用当前对象的 day 字段，示例如下：

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

Java 编程语言自动将所有实例变量和方法引用与 this 关键字联系在一起，因此，this 可以省略。下面的代码与前面的代码是等同的。

```
public class MyDate {
    private int day, month, year;

    public void tomorrow() {
        day = day + 1; // 在 day 前面没有使用 this
        // 其他代码
    }
}
```

4.3.3.2 区分同名变量

在类属性上定义的变量和方法内部定义的变量相同的时候，到底是调用谁？例如：

```
public class Test {
    int i = 2;
    public void t() {
        int i = 3; // 跟属性的变量名称是相同的
        System.out.println("实例变量 i=" + this.i);
        System.out.println("方法内部的变量 i=" + i);
    }
}
```

也就是说：“this.变量“调用的是当前属性的变量值，直接使用变量名称调用的是相对距离最近的变量的值。

4.3.3.3 作为方法名来初始化对象

this 用在构造方法中，this()也就是相当于调用本类的其它构造方法，它必须作为构造方法的第一句。示例如下：

```
public class Test {
    public Test() {
        this(3);// 在这里调用本类的另外的构造方法
    }
    public Test(int a) {
    }
    public static void main(String[] args) {
        Test t = new Test();
    }
}
```

4.3.3.4 作为参数传递

需要在某些完全分离的类中调用一个方法，并将当前对象的一个引用作为参数传递时。例如：

```
    Birthday bDay = new Birthday (this);
```

4.4 引用类型

早些时候的编程语言和初级程序员将每个变量看作相互无关的实体。例如，如果一个程序需处理某个日期，则要声明三个单独的整数：

```
        int day, month, year;
```

上述语句作了两件事，一是当程序需要日、月或年的有关信息时，它将操作一个整数； 二是为那些整数分配存储器。

尽管这种作法很容易理解，但它存在两个重大缺陷。首先，如果程序需同时记录几个日期，则需要三个不同的声明。例如，要记录两个生日，你可能使用：

```
    int myBirthDay, myBirthMonth, myBirthYear;
        int yourBirthDay, yourBirthMonth, yourBirthYear;
```

这种方法很快就会引起混乱，因为需要的名称很多。

第二个缺陷是这种方法忽视了日、月和年之间的联系并把每个变量都作为一个独立的值，每个变量都是一个独立单元。

为了解决这个问题，**Java** 引入了引用类型，允许程序员自己定义类型。

4.4.1 什么是引用类型

引用类型（reference type）指向一个对象，不是原始值，指向对象的变量是引用变量。

在 Java 里面除去基本数据类型的其它类型都是引用数据类型，自己定义的 class 类都是引用类型，可以像基本类型一样使用。在 Java 程序运行时，会为引用类型分配一定量的存储空间并解释该存储空间的内容。

示例如下：

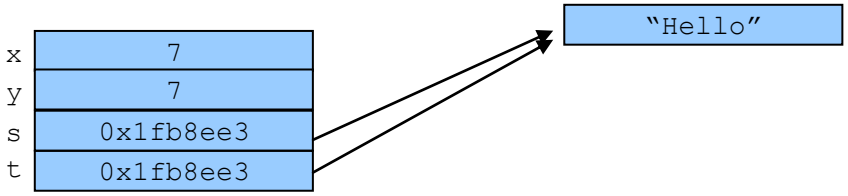
```
public class MyDate {
    private int day = 8;
    private int month = 8;
    private int year = 2008;
    public MyDate(int day, int month, int year){…}
    public void print(){…}
}
public class TestMyDate {
    public static void main(String args[]) {
       // 这个 today 变量就是一个引用类型的变量
       MyDate today = new MyDate(23, 7, 2008);
    }
}
```

4.4.2 引用类型的赋值

在 Java 编程语言中，用类的一个类型声明的变量被指定为引用类型，这是因为它正在引用一个非原始类型，这对赋值具有重要的意义。请看下列代码片段：

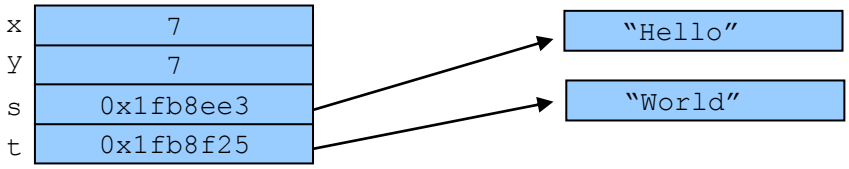
```
    int x = 7;
    int y = x;
    String s = "Hello";
    String t = s;
```

四个变量被创建：两个原始类型 int 和两个引用类型 String。x 的值是 7，而这个值被复制到 y；x 和 y 是两个独立的变量且其中任何一个的进一步的变化都不对另外一个构成影响。至于变量 s 和 t，只有一个 String 对象存在,它包含了文本“Hello”，s 和 t 均引用这个单一的对象。



将变量 t 重新定义为：t=”World”； 则新的对象 World 被创建，而 t 引用这个对象。上

述过程被描述如下：



4.4.3 按值传递还是按引用传递

这个在 Java 里面是经常被提起的问题，也有一些争论，似乎最后还有一个所谓的结论：“在 Java 里面参数传递都是按值传递”。事实上，这很容易让人迷惑，下面先分别看看什么是按值传递，什么是按引用传递，只要能正确理解，至于称作按什么传递就不是个大问题了。

4.4.3.1 按值传递

指的是在方法调用时，传递的参数是按值的拷贝传递。示例如下：

```
1. public class TempTest {
2.   private void test1(int a) {
3.     // 做点事情
4.     a++;
5.   }

6.   public static void main(String[] args) {
7.     TempTest t = new TempTest();
8.     int a = 3;
9.     t.test1(a);// 这里传递的参数a就是按值传递。
10.    System.out.println("main方法中的a===" + a);
11.  }
12.}
```

按值传递重要特点：传递的是值的拷贝，也就是说传递后就互不相关了。第 9 行的 a 和第 2 行的 a 是两个变量，当改变第 2 行 a 的值，第 9 行 a 的值是不变的，所以打印结果是 3。

我们把第 9 行的 a 称之为实参，第 2 行的 a 称之为形参；对于基本数据类型，形参数据的改变，不影响实参的数据。

4.4.3.2 按引用传递

指的是在方法调用时，传递的参数是按引用进行传递，其实传递的引用的地址，也就是变量所对应的内存空间的地址。示例如下：

```
1. public class TempTest {
2.   private void test1(A a) {
```

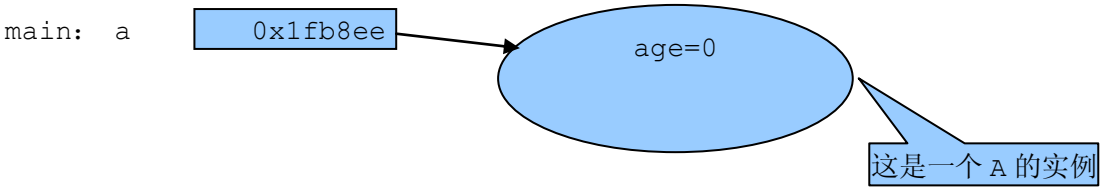
```
3.      age = 20;
4.      System.out.println("test1 方法中的age="+a.age);
5.    }
6.    public static void main(String[] args) {
7.      TempTest t = new TempTest();
8.      A a = new A();
9.      a.age = 10;
10.     t.test1(a); // 这里传递的参数a就是按引用传递
11.     System.out.println("main方法中的age="+a.age);
12.   }
13.}
14.class A {
15.   public int age = 0;
16.}
```

运行结果如下： test1 方法中的 age=20 main 方法中的 age=20

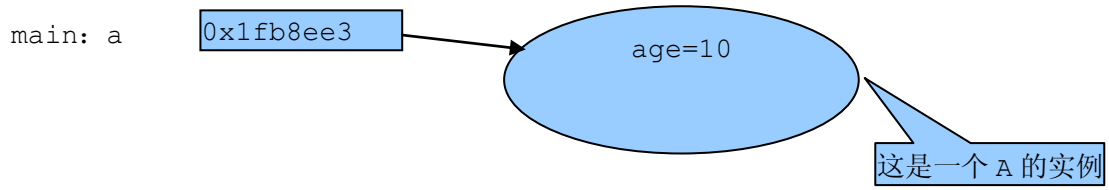
按引用传递的重要特点：
传递的是值的引用，也就是说传递前和传递后都指向同一个引用（也就是同一个内存空间）。

要想正确理解按引用传递的过程，就必须学会理解内存分配的过程，**内存分配示意图**可以辅助我们去理解这个过程。

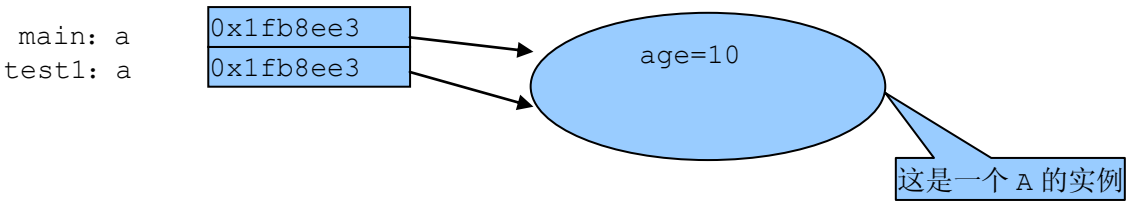
用上面的例子来进行分析：
（1）运行开始，运行第 8 行，创建了一个 A 的实例，内存分配示意图如下：



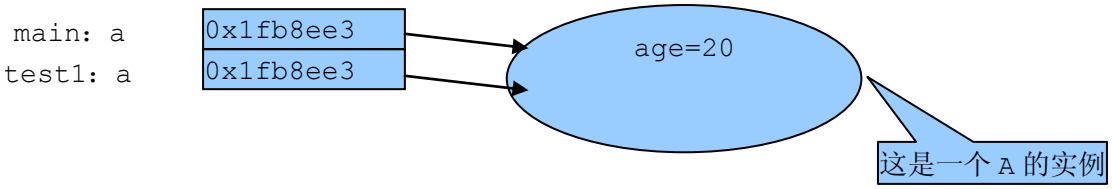
（2）运行第 9 行，是修改 A 实例里面的 age 的值，运行后内存分配示意图如下：



（3）运行第 10 行，是把 main 方法中的变量 a 所引用的内存空间地址，按引用传递给 test1 方法中的 a 变量。请注意：这两个 a 变量是完全不同的，不要被名称相同所蒙蔽，但它们指向了同一个 A 实例。内存分配示意图如下：



（4）运行第 3 行，为 test1 方法中的变量 a 指向的 A 实例的 age 进行赋值，完成后形成新的内存示意图如下：



此时 A 实例的 age 值的变化是由 test1 方法引起的。

（5）运行第 4 行，根据此时的内存示意图，输出 test1 方法中的 age=20
（6）运行第 11 行，根据此时的内存示意图，输出 main 方法中的 age=20

对上述例子的改变
理解了上面的例子，可能有人会问，那么能不能让按照引用传递的值，相互不影响呢？ 就是 test1 方法里面的修改不影响到 main 方法里面呢？

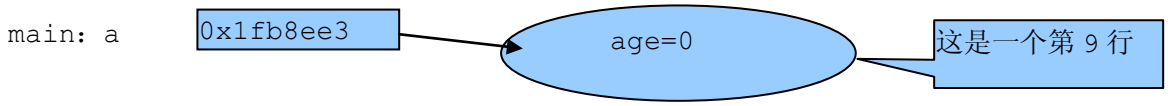
方法是在 test1 方法里面新 new 一个实例就可以了。改变成下面的例子，其中第 3 行为新加的：

```
1. public class TempTest {
2.   private void test1(A a) {
3.     a = new A();// 新加的一行
4.     age = 20;
5.     System.out.println("test1 方法中的age=" + a.age);
6.   }
7.   public static void main(String[] args) {
8.     TempTest t = new TempTest();
9.     A a = new A();
10.    age = 10;
11.    t.test1(a);
12.    System.out.println("main方法中的age=" + a.age);
13.  }
14.}
15.class A {
16.  public int age = 0;
17.}
```

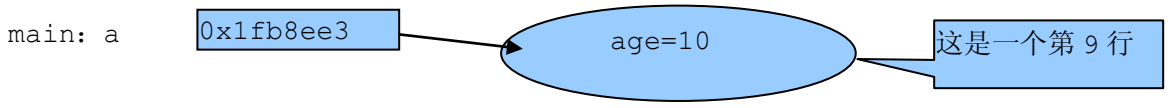
运行结果为： test1 方法中的 age=20 main 方法中的 age=10

为什么这次的运行结果和前面的例子不一样呢，还是使用内存示意图来理解一下：

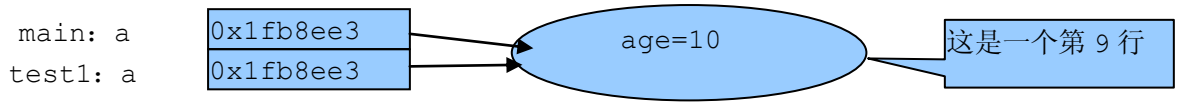
（1）运行开始，运行第 9 行，创建了一个 A 的实例，内存分配示意图如下：



（2）运行第 10 行，是修改 A 实例里面的 age 的值，运行后内存分配示意图如下：

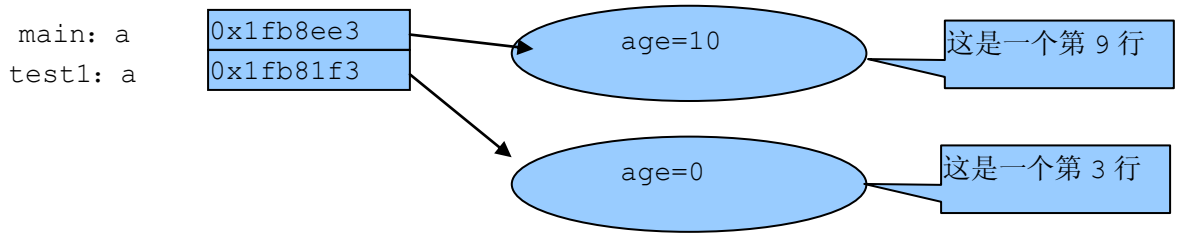


（3）运行第 11 行，是把 main 方法中的变量 a 所引用的内存空间地址，按引用传递给 test1 方法中的 a 变量。请注意：这两个 a 变量是完全不同的，不要被名称相同所蒙蔽。

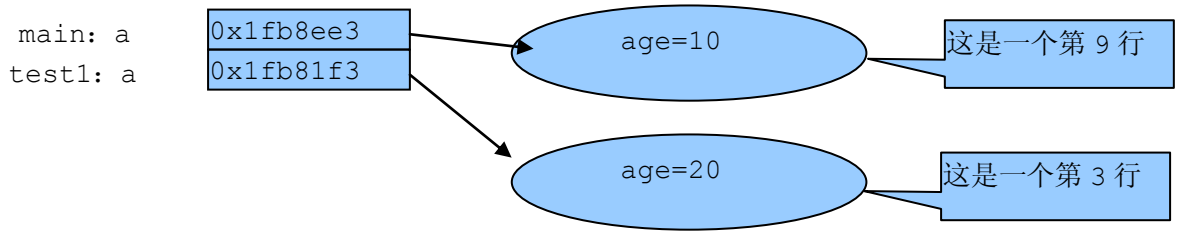


也就是说：是两个变量都指向同一个空间。

（4）运行第 3 行，为 test1 方法中的变量 a 重新生成了新的 A 实例的，完成后形成的新的内存示意图如下：



（5）运行第 4 行，为 test1 方法中的变量 a 指向的新的 A 实例的 age 进行赋值，完成后形成的新的内存示意图如下：



注意：这个时候 test1 方法中的变量 a 的 age 被改变，而 main 方法中的是没有改变的。

（6）运行第 5 行，根据此时的内存示意图，输出 test1 方法中的 age=20

（7）运行第 12 行，根据此时的内存示意图，输出 main 方法中的 age=10

说明：

“在 Java 里面参数传递都是按值传递”这句话的意思是：按值传递是传递的值的拷贝，按引用传递其实传递的是引用的地址值，所以统称按值传递。

在 Java 里面只有基本类型和按照下面这种定义方式的 String 是按值传递，其它的都是按引用传递。就是直接使用双引号定义字符串方式：String str = "Java 快车";

4.5 类中的变量

4.5.1 实例变量和局部变量

在方法外定义的变量主要是实例变量，它们是在使用 new Xxxx ()创建一个对象时被分配内存空间的。每当创建一个对象时，系统就为该类的所有实例变量分配存储空间；创建多个对象就有多份实例变量。通过对象的引用就可以访问实例变量。

在方法内定义的变量或方法的参数被称为局部 (local) 变量 ，有时也被用为自动（automatic）、临时（temporary）或栈（stack）变量。

方法参数变量定义在一个方法调用中传送的自变量，每次当方法被调用时，一个新的变量就被创建并且一直存在到程序的运行跳离了该方法。当执行进入一个方法遇到局部变量的声明语句时，局部变量被创建，当执行离开该方法时，局部变量被取消，也就是该方法结束时局部变量的生命周期也就结束了。

因而，局部变量有时也被引用为“临时或自动”变量。在成员方法内定义的变量对该成员变量是“局部的”，因而，你可以在几个成员方法中使用相同的变量名而代表不同的变量。该方法的应用如下所示：

```
public class Test {
    private int i; // Test类的实例变量
    public int firstMethod() {
        int j = 1; // 局部变量
        // 这里能够访问i和j
        System.out.println("firstMethod 中  i=" + i + ",j=" + j);
        return 1;
    } // firstMethod()方法结束
    public int secondMethod(float f) {
        // method parameter
        int j = 2; // 局部变量，跟firstMethod()方法中的j是不同的
        // 这个j的范围是限制在secondMethod()方法中的
        // 在这个地方，可以同时访问i,j,f
        System.out.println("secondMethod中 i=" + i + ",j=" + j + ",f=" +f);
        return 2;
    }
```

```
    public static void main(String[] args) {
        Test t = new Test();
        t.firstMethod();
        t.secondMethod(3);
    }
}
```

4.5.2 变量初始化

在 Java 程序中，任何变量都必须经初始化后才能被使用。当一个对象被创建时，实例变量在分配内存空间时按程序员指定的初始化值赋值，否则系统将按下列默认值进行初始化：

数据类型	初始值
byte	0
short	0
int	0
long	0L
char	'\u0000'
float	0.0f
double	0
boolean	false
所有引用类型	null

注意：一个具有空值“null”的引用不引用任何对象。试图使用它引用的对象将会引起一个异常。异常是出现在运行时的错误，这将在模块“异常”中讨论。

在方法外定义的变量被自动初始化。局部变量必须在使用之前做“手工”(由程序员进行)初始化。如果编译器能够确认一个变量在初始化之前可能被使用的情形，编译器将报错。

```
public class Test {
    private int i; // Test类的实例变量
    public void test1() {
        int x = (int) (Math.random() * 100);
        int y;
        int z;
        if (x > 50) {
            y = 9;
        }
        z = y + x; // 将会引起错误，因为y可能还没有被初始化就使用了
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.test1();
    }
```

```
}
```

4.5.3 变量的范围(scope)

Java 变量的范围有四个级别:类级、对象实例级、方法级、块级。

类级变量又称全局级变量，在对象产生之前就已经存在，就是用 static 修饰的属性。

对象实例级，就是属性变量。

方法级：就是在方法内部定义的变量，就是局部变量。

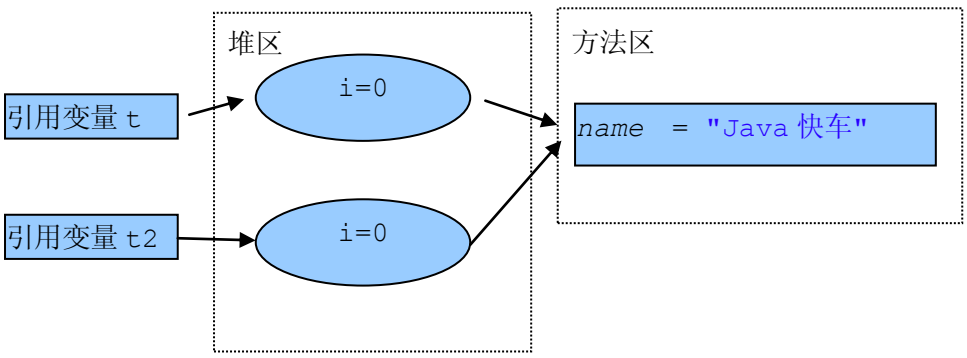
块级：就是定义在一个块内部的变量，变量的生存周期就是这个块，出了这个块就消失了，比如 if、for 语句的块。

示例如下：

```
public class Test {
    private static String name = "Java快车";// 类级
    private int i; // 对象实例级，Test类的实例变量
    {// 属性块，在类初始化属性时候运行
        int j = 2;// 块级
    }
    public void test1() {
        int j = 3;// 方法级
        if (j == 3) {
            int k = 5;// 块级
        }
        // 这里不能访问块级的变量，块级变量只能在块内部访问
        System.out.println("name=" + name + ",i=" + i + ",j=" + j);
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.test1();
       Test t2 = new Test();
    }
}
```

运行结果：

```
name=Java 快车,i=0,j=3
```



说明：

（1）方法内部除了能访问方法级的变量，还可以访问类级和实例级的变量。

（2）块内部能够访问类级、实例级变量，如果块被包含在方法内部，它还可以访问方法级的变量。

（3）方法级和块级的变量必须被显示地初始化，否则不能访问。

4.6 包装类

虽然 Java 语言是典型的面向对象编程语言，但其中的 8 种基本数据类型并不支持面向对象的编程机制，基本类型的数据不具备“对象”的特性----不携带属性、没有方法可调用。 沿用它们只是为了迎合人类根深蒂固的习惯，并的确能简单、有效地进行常规数据处理。

这种借助于非面向对象技术的做法有时也会带来不便，比如引用类型数据均继承了 Object 类的特性，要转换为 String 类型（经常有这种需要）时只要简单调用 Object 类中定义的 toString()即可，而基本数据类型转换为 String 类型则要麻烦得多。为解决此类问题，Java 语言引入了封装类的概念，在 JDK 中针对各种基本数据类型分别定义相应的引用类型，并称之为包装类（Wrapper Classes）。

下表描述了基本数据类型及对应的包装类

基本数据类型	对应的包装类
byte	Byte
short	Short
int	Integer
long	Long
char	Character
float	Float
double	Double
boolean	Boolean

每个包装类的对象可以封装一个相应的基本类型的数据，并提供了其它一些有用的功能。包装类对象一经创建，其内容（所封装的基本类型数据值）不可改变。

例，包装类用法程序：Wrapper.java

```
public class Wrapper {
    public static void main(String args[]) {
        int i = 500;
        Integer t = new Integer(i);
        int j = t.intValue(); // j = 500
        String s = t.toString(); // s = "500"
        System.out.println(t);
        Integer t1 = new Integer(500);
        System.out.println(t.equals(t1));
    }
}
```

程序运行结果为：

```
500
true
```

包装类一个常用的功能就是把字符串类型的数据造型成为对应的基本数据类型，如下示例：

```
String str = "123";
int a = Integer.parseInt(str);
```

更过的功能还请查看 JDK 文档。

4.7 强制类型转换

把某种类型强制转换成另外一种类型就叫做强制类型转换。

例如，可以将一个 long 值“挤压”到一个 int 变量中。显式转型做法如下：

```
long  bigValue  =  99L;
int squashed  =  (int)(bigValue);
```

在上述程序中，期待的目标类型被放置在圆括号中，并被当作表达式的前缀，该表达式必须被更改。一般来讲，建议用圆括号将需要转型的全部表达式封闭。否则，转型操作的优先级可能引起问题。

注意：强制类型转换只能用在原本就是某个类型，但是被表示成了另外一种类型的时候，可以把它强制转换回来。强制转换并不能在任意的类型间进行转换。

比如上面的例子：99 这个数本来就是一个 int 的数，但是它通过在后面添加 L 来表示成了一个 long 型的值，所以它才能够通过强制转换来转换回 int 类型。

4.8 Java 类的基本运行顺序

作为程序员，应该对自己写的程序具备充分的掌控能力，应该清楚程序的基本运行过程，否则糊里糊涂的，不利于对程序的理解和控制，也不利于技术上的发展。 我们以下面的类来说明一个基本的 Java 类的运行顺序：

```
1. public class Test {
2.   private String name;
3.   private int age;
4.
5.   public Test() {
6.     name = "Tom";
7.     age = 20;
8.   }
9.   public static void main(String[] args) {
10.    Test t = new Test();
11.    System.out.println(t.name + "的年龄是" + t.age );
12.  }
13.}
```

运行的基本顺序是：

（1）先运行到第 9 行，这是程序的入口。

（2）然后运行到第 10 行，这里要 new 一个 Test，就要调用 Test 的构造方法。

（3）就运行到第 5 行，注意：可能很多人觉得接下来就应该运行第 6 行了，错！**初始化一个类，必须先初始化它的属性**。

（4）因此运行到第 2 行，然后是第 3 行。

（5）**属性初始化完过后，才回到构造方法**，执行里面的代码，也就是第 6 行、第 7 行。

（6）然后是第 8 行，表示 new 一个 Test 实例完成。

（7）然后回到 main 方法中执行第 11 行。

（8）然后是第 12 行，main 方法执行完毕。

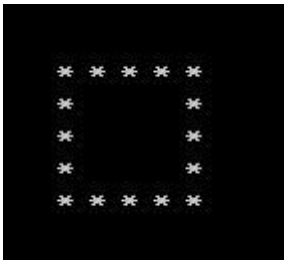
说明：这里只是说明一个基本的运行过程，没有考虑更多复杂的情况。

4.9 学习目标

1. 理解为什么要面向对象，理解现实中的对象和类型与 Java 中的类和对象的相互关系。
2. 能够描述 Java 中类的 6 个组成部分。
3. 简述什么是包？包的功能，描述使用包的好处。
4. 了解 Java 中的常见包，能够写出 5 个以上的 JDK 中的包，以及他们的基本功能
5. 能够描述 import 关键字的用法。
6. 简述 Java 的访问修饰符类型？分别有什么功能？
7. 能够描述**类定义**的完整的语法结构。
8. 能够描述**属性定义**的完整的语法结构。
9. 能够描述**构造方法定义**的完整的语法结构。
10. 理解**构造方法**的调用和系统提供的默认的构造方法
11. 理解析构方法
12. 能够描述**方法定义**的完整的语法结构。
13. 分别说明：在类上、在属性上、在方法上等能使用那些访问修饰符
14. 什么是形参（形式参数）？什么是实参（实际参数）？
15. 如何通过类创建对象，如何调用对象的方法，如何调用对象的属性。
16. 什么是引用数据类型？
17. 什么是按值传递，什么是按引用传递？
18. this 关键字的用法？
19. 什么时候使用 this 关键字？
20. 什么是实例变量，什么是局部变量？
21. 如何理解属性的初始化？
22. 什么是强制类型转换。
23. 8 个基本数据类型对应 8 个包装类，说出两者的不同之处。
24. 理解自动包装和解包。
25. 简述 Java 类的基本运行顺序。

4.10 练习

1. 请编写一个方法实现如下功能：将 0 至 6 的数字转换为星期日到星期六的字符串。要求：输入一个数字参数，返回字符串。
2. 请编写一个方法实现如下功能：将任意三个整数 a,b,c 按从小到大排序。要求：输入三个参数，返回一个排好顺序的字符串。
3. 请编写一个方法实现如下功能：计算 1 加到 n（n>=2 的整数）的总和。
4. 请编写一个方法实现如下功能：得到一个整数的绝对值。
5. 请编写一个方法，输入两个参数，判断第一个数字是否是第二个数字的整数倍数，如果是，返回 true；如果不是，返回 false。
6. 请编写一个方法，输入正方形的边长，然后利用星号和空格，打印具有那个边长的一个空心正方形。假定输入参数是 5，则输出效果应该是下面这样：



7. “回文”是一种特殊的数字或文字短语，无论顺读，还是倒读，结果都是一样的。 例如，下面这些整数其实都是“回文”： 11、1221、5555、4554、12321、11611。编写一个方法判断是否是“回文”：输入数字，如果是“回文”，返回 true，否则返回 false
8. 请编写一个方法实现如下功能：输入三个非零的整数值，判断它们是否能代表一个三角形的 3 个边长，如果能，返回 true；如果不能返回 false。（提示：判断三个边长能否组成三角形的规则是：任意两个边长之和大于第三个边长）
9. 请编写一个方法实现如下功能：用程序找出每位数的立方和等于该数本身值的所有的 3 位数。（水仙花数）
10. 有一个序列，首两项为 0，1，以后各项值为前两项值之和。写一个方法来实现求这个序列的和
11. 写一个 MyPoint 完全封装类，其中含有私有的 int 类型的 x 和 y 属性，分别用公有的 getX 和 setX、getY 和 setY 方法访问，定义一个 toString 方法用来显示这个对象的 x、y 的值，如显示（1，2），最后用 main 方法测试。
12. 为 MyPoint 类添加 public boolean equals(MyPoint mp)方法，创建两个 MyPoint 对象，使用 equals 方法比较两个对象是否相等，再使用==比较两个对象是否相等，思考两种比较的不同之处。

5 Java 高级类特性

面向对象有三大特征，即封装、继承、多态。

5.1 封装

封装这个词听起来好象是将什么东西包裹起来不要别人看见一样，就好象是把东西装进箱子里面，这样别人就不知道箱子里面装的是什么东西了。其实 JAVA 中的封装这个概念也就和这个是差不多的意思。

封装是 JAVA 面向对象的特点的表现，封装是一种信息隐蔽技术。它有两个含义：即把对象的全部属性和全部服务结合在一起，形成一个不可分割的独立单位；以及尽可能隐藏对象的内部结构。也就是说，如果我们使用了封装技术的话，别人就只能用我们做出来的东西而看不见我们做的这个东西的内部结构了。

封装的功能：

- 迫使用户去使用一个界面访问数据
- 使代码更好维护
- 隐藏对象的实现细节

封装迫使用户通过方法访问数据能保护对象的数据不被误修改，还能使对象的重用变得更简单。数据隐藏通常指的就是封装。它将对象的外部界面与对象的实现区分开来，隐藏实现细节。迫使用户去使用外部界面，即使实现细节改变，还可通过界面承担其功能而保留原样，确保调用它的代码还继续工作。封装使代码维护更简单。

5.2 继承

5.2.1 继承

在面向对象世界里面，常常要创建某对象（如：一个职员对象），然后需要一个该基本对象的更专业化的版本，比如，可能需要一个经理的对象。显然经理实际上是一个职员，经理和职员具有 is-a 的关系，经理只是一个带有附加特征的职员。因此，需要有一种办法从现有对象来创建一个新对象，这个方式就是继承。现实中的事务，只要具有 is-a 的关系，在 java 中都可以用继承表示。

“继承”是面向对象软件技术当中的一个概念。如果一个对象 **A** 继承自另一个对象 **B**，就把这个 **A** 称为“**B** 的子对象”，而把 **B** 称为“**A** 的父对象”。继承可以使得子对象具有父对象的各种属性和方法，而不需要再次编写相同的代码。在令子对象继承父对象的同时，可以重新定义某些属性，并重写某些方法，即覆盖父对象的原有属性和方法，使其获得与父对象不同的功能。

5.2.2 extends 关键字

在 Java 中使用 extends 关键字来表达继承的关系，比如：经理这个类继承雇员这个类，示例如下：

```
public class Employee {
    String name;
    Date hireDate;
    Date dateOfBirth;
    String jobTitle;
    int grade;
    ...
}
public class Manager extends Employee {
    String department;
    Employee[] subordinates;
    ...
}
```

在这样的定义中，Manager 类被定义，具有 Employee 所拥有的所有变量及方法。所有这些变量和方法都是从父类的定义中继承来的。所有的程序员需要做的是定义额外特征或规定将适用的变化。

注意：这种方法是在维护和可靠性方面的一个伟大进步。如果在 Employee 类中进行修改，那么，Manager 类就会自动修改，而不需要程序员做任何工作，除了对它进行编译。

5.2.3 父子类的初始化顺序

Java 技术安全模式要求在子类执行任何东西之前，描述父类的一个对象的各个方面都必须初始化。因此，Java 编程语言总是在执行子构造方法前调用父类构造方法的版本。有继承的类在运行的时候，一定要记得：**初始化子类必先初始化父类**，这是 Java 程序的一个基本运行过程。比如：

```
public class Test extends Parent {
    private String name;
    private int age;
    public Test() {
        name="Tom";
        age=20;
    }
    public static void main(String[] args) {
        Test t = new Test();
        System.out.println(t.name + "的年龄是" + t.age);
    }
}
class Parent {
```

```
    private int num = 1;
    public Parent() {
        System.out.println("现在初始化父类");
    }
    public void test() {
        System.out.println("这是父类的test方法");
    }
}
```

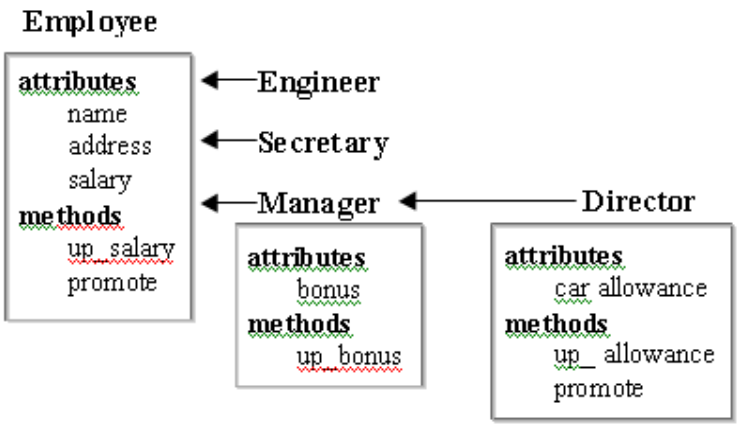
上述类的基本运行顺序是：

（1）先运行到第 8 行，这是程序的入口。
（2）然后运行到第 9 行，这里要 new 一个 Test，就要调用 Test 的构造方法。
（3）就运行到第 4 行，注意：初始化子类必先初始化父类。
（4）要先初始化父类，所以运行到第 15 行。
（5）然后是第 14 行，初始化一个类，必须先初始化它的属性。
（6）然后是第 16 行。
（7）然后是第 17 行，表示父类初始化完成。
（8）然后是回到子类，开始初始化属性，因此运行到第 2 行，然后是第 3 行。
（9）子类属性初始化完过后，才回到子类的构造方法，执行里面的代码，也就是第 5、6 行。
（10）然后是第 7 行，表示 new 一个 Test 实例完成。
（11）然后回到 main 方法中执行第 10 行。
（12）然后是第 11 行。

5.2.4 单继承性

单继承性：当一个类从一个唯一的类继承时，被称做单继承性。单继承性使代码更可靠。接口提供多继承性的好处，而且没有（多继承的）缺点。

Java 编程语言允许一个类仅能继承一个其它类，即一个类只能有一个父类。这个限制被称做单继承性。单继承性与多继承性的优点是面向对象程序员之间广泛讨论的话题。Java 编程语言加强了单继承性限制而使代码更为可靠，尽管这样有时会增加程序员的工作。后面会学到一个被叫做接口（interface）的语言特征，它允许多继承性的大部分好处，而不受其缺点的影响。使用继承性的子类的一个例子如图所示：



5.2.5 构造方法不能被继承

尽管一个子类从父类继承所有的方法和变量，但它不继承构造方法，掌握这一点很重要。一个类能得到构造方法，只有两个办法。或者写构造方法，或者根本没有写构造方法，类有一个默认的构造方法。

5.2.6 关键字 super

关键字 super 可被用来引用该类的父类，它被用来引用父类的成员变量或方法。父类行为被调用，就好象该行为是本类的行为一样，而且调用行为不必发生在父类中，它能自动向上层类追溯。

super 关键字的功能：
点取父类中被子类隐藏了的数据成员。
点取已经覆盖了的方法。
作为方法名表示父类构造方法。

5.2.6.1 点取父类中被子类隐藏了的数据成员或方法

```
public class Employee {
    private String name;
    private int salary;
    public String getDetails() {
        return "Name:  " + name + "\nSalary:  " + salary;
    }
}
public class Manager extends Employee {
    private String department;
    public String getDetails() {
        return super.getDetails() + // 调用父类的方法
               "\nDepartment:  " + department;
    }
}
```

请注意，super.method()格式的调用，如果对象已经具有父类类型，那么它的方法的整个行为都将被调用，也包括其所有负面效果。该方法不必在父类中定义，它也可以从某些祖先类中继承。也就是说可以从父类的父类去获取，具有追溯性，一直向上去找，直到找到为止，这是一个很重要的特点。

5.2.6.2 调用父类构造方法

在许多情况下，使用默认构造方法来对父类对象进行初始化。当然也可以使用 super 来显示调用父类的构造方法。

```
public class Employee {
    String name;
```

```
    public Employee(String n) {
        name = n;
    }
}
public class Manager extends Employee {
    String department;
    public Manager(String s, String d) {
        super(s);
        department = d;
    }
}
```

注意：无论是 super 还是 this，都必须放在构造方法的第一行。

通常要定义一个带参数的构造方法，并要使用这些参数来控制一个对象的父类部分的构造。可能通过从子类构造方法的第一行调用关键字 super 的手段调用一个特殊的父类构造方法作为子类初始化的一部分。要控制具体的构造方法的调用，必须给 super()提供合适的参数。当不调用带参数的 super 时，缺省的父类构造方法（即，不带参数的构造方法）被隐含地调用。**在这种情况下，如果没有缺省的父类构造方法，将导致编译错误。**

```
public class Employee {
    String name;
    public Employee(String n) {
        name = n;
    }
}
public class Manager extends Employee {
    String department;
    public Manager(String s, String d) {
        super(s); // 调用父类参数为 String 类型的构造方法，没有这句话，编译会出错
        department = d;
    }
}
```

当被使用时，super 或 this 必须被放在构造方法的第一行。显然，两者不能被放在一个单独行中，但这种情况事实上不是一个问题 。如果写一个构造方法，它既没有调用 super(…)也没有调用 this(…)，编译器自动插入一个调用到父类构造方法中，而不带参数。 其它构造方法也能调用 super(…)或 this(…)，调用一个 static 方法和构造方法的数据链。 最终发生的是父类构造方法（可能几个）将在链中的任何子类构造方法前执行。

5.3 方法的覆盖和重载

5.3.1 方法的覆盖

5.3.1.1 什么是方法的覆盖（Overridden Methods）

在类继承中，子类可以修改从父类继承来的行为，也就是说子类能创建一个与父类方法有不同功能的方法，但具有相同的：名称、返回类型、参数列表。如果在子类中定义一个方法，其方法名称、返回值类型及参数列表正好与父类中方法的名称、返回值类型及参数列表相匹配，那么称，子类的方法覆盖了父类的方法。

5.3.1.2 示例

如下在 Employee 和 Manager 类中的这些方法：

```
public class Employee {
    String name;
    int salary;
    public String getDetails() {
        return " Name: " + name + " \n " + "Salary: " + salary;
    }
}
public class Manager extends Employee {
    String department;
    public String getDetails() {
        return " Name: " + name + " \n " + " Manager of " + department;
    }
}
```

Manager 类有一个定义的 getDetails()方法，因为它是从 Employee 类中继承的。基本的方法被子类的版本所代替或覆盖了。

5.3.1.3 方法的覆盖规则

子类的方法的名称以及子类方法参数的顺序必须与父类中的方法的名称以及参数的顺序相同，以便该方法覆盖父类版本。下述规则适用于覆盖方法：

覆盖方法不能比它所覆盖的方法访问性差（即访问权限不允许缩小）。
覆盖方法不能比它所覆盖的方法抛出更多的异常。

这些规则源自多态性的属性和 Java 编程语言必须保证"类型安全"的需要。考虑一下这个无效方案：

```
public class Parent {
    public void method() {
    }
}
```

```
public class Child extends Parent {
    private void method() {// 编译就会出错
    }
}
public class Test {
    public void otherMethod() {
        Parent p1 = new Parent();
        Parent p2 = new Child();
        p1.method();
        p2.method();
    }
}
```

Java 编程语言语义规定，p2.method()导致方法的 Child 版本被执行，但因为方法被声明为 private，p2（声明为 Parent）不能访问它。于是，语言语义冲突。

5.3.2 方法的重载

假如你必须在不同情况下发送不同的信息给同一个成员方法的话，该怎么办呢？你可以通过对此成员方法说明多个版本的方法来实现重载。重载的本质是创建了一个新的成员方法：你只需给它一个不同的参数列表。

5.3.2.1 什么是重载

在同一个 Java 类中（包含父类），如果出现了方法名称相同，而参数列表不同的情况就叫做重载。

参数列表不同的情况包括：个数不同、类型不同、顺序不同等等。特别提示，仅仅参数变量名称不同是不可以的。

5.3.2.2 示例

如下例所示：

```
    void getArea(int w,int h);
    void getArea(float w,float h);
```

在第二种情况下，成员方法 getArea()接受两个浮点变量作为它的参数，编译器根据调用时的不同参数来决定该调用哪一种成员方法，假如你把两个整数提供给成员方法，就调用第一个成员方法；假如你把两个浮点数提供给成员方法，第二个成员方法就被调用。

当写代码来调用这些方法中的一个方法时，便以其会根据提供的参数的类型来选择合适的方法。

注意：跟成员方法一样，构造方法也可以重载。

5.3.2.3 方法的重载规则

方法名称必须相同
参数列表必须不同（个数不同，或类型不同，或参数排列顺序不同）。
方法的返回类型可以相同也可以不相同。仅仅返回类型不同不足以成为方法的重载。

注意：调用语句的参数表必须有足够的不同，以至于允许区分出正确的方法被调用。正常的拓展晋升（如，单精度类型 float 到双精度类型 double）可能被应用，但是这样会导致在某些条件下的混淆。

5.3.2.4 比较方法的覆盖和重载

重载方法： 在一个类（或父子类）中用相同的名字创建多个方法（每个方法的参数表不同）

方法覆盖： 在一个类中创建的方法与父类中方法的名字、返回类型和参数表相同，覆盖是针对两个类说的，而且必须是子类（或孙类，孙孙类等）覆盖掉父类的方法

5.4 多态性

5.4.1 什么是多态

多态是同一个行为具有多个不同表现形式或形态的能力。

比如我们说“宠物”这个对象，它就有很多不同的表达或实现，比如有小猫、小狗、蜥蜴等等。那么我到宠物店说“请给我一只宠物”，服务员给我小猫、小狗或者蜥蜴都可以，我们就说“宠物”这个对象就具备多态性。

再回想一下经理和职员的关系，经理类具有父类职员类的所有属性、成员和方法。这就是说，任何在 Employee 上的合法操作在 Manager 上也合法。如果 Employee 有 raiseSalary()和 fire()两个方法，那么 Manager 类也有。在这种 Manager 继承 Employee 的情况下，一个 Employee 既可以是一个普通的 Employee 类，也可以是一个 Manager 类。也就是说下述表示都是对的：

```
Employee e = new Employee();
Employee e = new Manager();
```

从上面可以看到：同一个行为 Employee 具有多个不同的表现形式（既可以是一个普通的 Employee 类，也可以是一个 Manager 类），这就被称为多态。

注意：方法没有多态的说法，严格说多态是类的特性。但是也有对方法说多态的，了解一下，比如前面学到的方法**覆盖称为动态多态**，是一个运行时问题；**方法重载称为静态多态**， 是一个编译时问题。

5.4.2 多态与类型

一个对象只有一个格式（是在构造时给它的）。但是，既然变量能指向不同格式的对象， 那么变量就是多态性的。也就是说一个对象只有一种形式，但一个变量却有多种不同形式。 象大多数面向对象语言一样，Java 实际上允许父类类型的引用变量指向一个子类的对象。因此，可以说：

```
Employee e = new Manager()
```

使用变量 e 是因为，你能访问的对象部分只是 Employee 的一个部分；Manager 的特殊部分是隐藏的。这是因为编译者应意识到，e 是一个 Employee，而不是一个 Manager。因而， 下述情况是不允许的：

```
e.department = " Finance " ; //非法的，编译时会出错
```

可能有的人会不理解，为什么明明是 new 的一个 Manager，却不能访问 Manager 的属性数据。原因在于编译的时候，变量 e 是一个 Employee 的类型，编译器并不去管运行时 e 指向的具体对象是一个 Employee 的对象，还是一个 Manager 的对象，所以它只能访问到 Employee 里面定义的属性和方法。所以说编译时看数据类型。

那么要想访问到 Manager 里面的 department 该怎么办呢？这就需要先对 e 进行强制类型转换，把它还原成为 Manager 类型，就可以访问到 Manager 里面的属性和方法了，如下：

```
Employee e = new Manager();
Manager m = (Manager)e;
m.department = "开发部"; //这就是合法的了
```

5.4.3 到底运行哪一个方法？

这里会给我们带来一个麻烦，父子类中有相同的方法，那么在运行时到底调用哪一个方法呢？假设下述方案：

```
Employee e = new Employee();
Manager m = new Manager();
```

如果请求 e.getDetails()和 m.getDetails()，就会调用不同的行为。Employee 对象将执行与 Employee 有关的 getDetails 版本， Manager 对象将执行与 Manager 有关的 getDetails()版本。

不明显的是如下所示：

```
Employee e = new Manager();
e.getDetails();
```

或某些相似效果，比如一个通用方法参数或一个来自异类集合的项。事实上，你得到与变量的运行时类型（即，变量所引用的对象的类型）相关的行为，而不是与变量的编译时类型相关的行为。这是面向对象语言的一个重要特征。它也是多态性的 一个特征，并通常被称作虚拟方法调用。

在前例中，被执行的 e.getDetails()方法来自对象的真实类型 Manager。因此规则是：编译时看数据类型，运行时看实际的对象类型（new 操作符后跟的构造方法是哪个类的）。一句话：**new 谁就调用谁的方法。**

5.4.4 instanceof 运算符

多态性带来了一个问题：如何判断一个变量所实际引用的对象的类型。C++使用 runtime-type information(RTTI)，Java 使用 instanceof 操作符。

instanceof 运算符功能：用来判断某个实例变量是否属于某种类的类型。一旦确定了变量所引用的对象的类型后，可以将对象恢复给对应的子类变量，以获取对象的完整功能。 示例如下：

```
public class Employee extends Object{}
public class Manager extends Employee {}
public class Contractor extends Employee{}
```

如果通过 Employee 类型的引用接受一个对象，它变不变成 Manager 或 Contractor 都可以。可以象这样用 instanceof 来测试：

```
public void method(Employee e) {
    if (e instanceof Manager) {
        // 如果雇员是经理，可以做的事情写在这里
    }else if (e instanceof Contractor) {
        // 如果雇员是普通的职员，可以做的事情写在这里
    }else {
       // 说明是临时雇员，可以做的事情写在这里
    }
}
```

5.4.5 多态对象的类型转换

在你接收父类的一个引用时，你可以通过使用 instanceof 运算符判定该对象实际上是你所要的子类，并可以用类型转换该引用的办法来恢复对象的全部功能。

```
public void method(Employee e) {
    if (e instanceof Manager) {
        Manager m = (Manager)e;
        System.out.println(" This is the manager of " + m.department);
    }
    // rest of operation
}
```

如果不用强制类型转换，那么引用 e.department 的尝试就会失败，因为编译器不能将被称做 department 的成员定位在 Employee 类中。

如果不用 instanceof 做测试，就会有类型转换失败的危险。通常情况下，类型转换一个对象引用的尝试是要经过几种检查的：

向上强制类型转换类层次总是允许的，而且事实上不需要强制类型转换运算符。可由简单的赋值实现。

严格讲不存在向下类型转换，其实就是强制类型转换，编译器必须满足类型转换至少是可能的这样的条件。比如，任何将 Manager 引用类型转换成 Contractor 引用的尝试是肯定不允许的，因为 Contractor 不是一个 Manager。类型转换发生的类必须是当前引用类型的子类。

如果编译器允许类型转换 ， 那么， 该引用类型就会在运行时被检查 。比如，如果 instanceof 检查从源程序中被省略，而被类型转换的对象实际上不是它应被类型转换进去的类型，那么，就会发生一个运行时错误(exception)。异常是运行时错误的一种形式，而且是后面章节的主题。

5.5 static

5.5.1 static 修饰符

static 修饰符能够与**属性、方法和内部类**一起使用，表示是“静态”的。

类中的静态变量和静态方法能够与“类名”一起使用，不需要创建一个类的对象来访问该类的静态成员。所以 static 修饰的变量又称作“类变量”。这与实例变量不同。实例变量总是用对象来访问，因为它们的值在对象和对象之间有所不同。

下列示例展示了如何访问一个类的静态变量：

```
class StaticModifier {
    static int i = 10;
    int j;
    StaticModifier() {
        j = 20;
    }
}
public class Test {
    public static void main(String args[]) {
        System.out.println("类变量  i=" + StaticModifier.i);
        StaticModifier s = new StaticModifier();
        System.out.println("实例变量  j=" + s.j);
    }
}
```

上述程序的输出是：

```
类变量 i=10
实例变量 j=20
```

5.5.2 static 属性的内存分配

在上面的例子中，无需创建类的对象即可访问静态变量 i。之所以会产生这样的结果，是因为编译器只为整个类创建了一个静态变量的副本，因此它能够用类名进行访问。也就是说：一个类中，一个 static 变量只会有一个内存空间，虽然有多个类实例，但这些类实例中的这个 static 变量会共享同一个内存空间。示例如下：

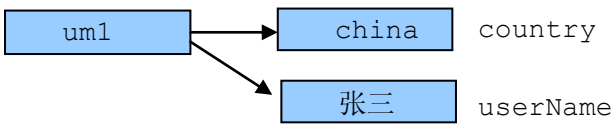
```
public class Test {
    public static void main(String[] args) {
        UserModel um1 = new UserModel();
        um.userName = "张三";
        um.country = "china";
        UserModel um2 = new UserModel();
        um2.userName = "李四";
        um2.country = "中国";
        System.out.println("um1.userName==" + um1.userName +
                                " um1.country==" +  um1.country );
        System.out.println("um2.userName==" + um2.userName +
                                " um2.country==" +  um2.country );
    }
}
class UserModel {
    public static String country
    public String userName;
}
```

运行结果：

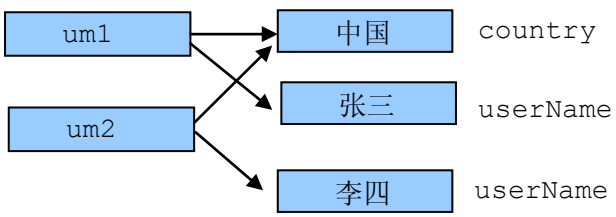
```
um1.userName==张三 um1.country==中国
um2.userName==李四 um2.country==中国
```

为什么会是一样的值呢？就是因为多个 UserModel 实例中的静态变量 country 是共享同一内存空间，um1.country 和 um2.country 其实指向的都是同一个内存空间，所以就得到上面的结果了。

在对象 um2 创建前的运行后内存分配示意图如下：



在对象 um2 创建后的运行后内存分配示意图如下：



要想看看是不是 static 导致这样的结果，你可以尝试去掉 country 前面的 static，然后再试一试，看看结果，应该如下：

```
um1.userName==张三 um1.country==china
um2.userName==李四 um2.country==中国
```

还有一点也很重要：static 的变量是在类装载的时候就会被初始化。也就是说，只要类被装载，不管你是否使用了这个 static 变量，它都会被初始化。

小结一下：类变量（class variables）用关键字 static 修饰，在类加载的时候，分配类变量的内存，以后再生成类的实例对象时，将共享这块内存（类变量），任何一个对象对类变量的修改，都会影响其它对象。外部有两种访问方式：通过对象来访问或通过类名来访问。

5.5.3 static 的基本规则

有关静态变量或方法的一些要点如下：

- 在同一个类中，静态方法只能访问静态属性或静态方法；
- 在同一个类中，静态方法不能够直接调用非静态方法；
- 如访问控制权限允许，静态属性和静态方法可以使用类名加“.”方式调用；当然也可以使用实例加“.”方式调用；
- 静态方法中不存在当前对象，因而不能使用this，当然也不能使用super；
- 静态方法不能被非静态方法覆盖；
- 构造方法不允许声明为static 的；
- 局部变量不能使用static修饰；

static 方法可以用类名来访问，如：

```
public class GeneralFunction {
    public static int addUp(int x, int y) {
        return x + y;
    }
}
public class UseGeneral {
    public void method() {
        int a = 9;
        int b = 10;
        int c = GeneralFunction.addUp(a, b);
        System.out.println("addUp() gives " + c);
    }
}
```

因为 static 方法不需它所属的类的任何实例就会被调用，因此没有 this 值。结果是，static 方法不能访问与它本身的参数以及 static 变量之外的任何变量，访问非静态变量的尝试会引起编译错误。

注： 非静态变量只限于实例，并只能通过实例引用被访问。

5.5.4 静态初始器—静态块

静态初始器（Static Initializer）是一个存在与类中方法外面的静态块。静态初始器仅仅在类装载的时候（第一次使用类的时候）执行一次。静态初始器的功能是：往往用来初始化静态的类属性。

示例：

```
class Count {
    public static int counter;
    static {// 只运行一次
        counter = 123;
        System.out.println("Now in static block.");
    }
    public void test() {
        System.out.println("test method==" + counter);
    }
}
public class Test {
    public static void main(String args[]) {
        System.out.println("counter=" + Count.counter);
        new Count().test();
    }
}
```

运行结果是：

```
Now in static block. counter=123
test method==123
```

5.5.5 静态 import

当我们要获取一个随机数时，写法是：

```
public class Test {
    public static void main(String[] args) {
        double randomNum = Math.random();
        System.out.println("the randomNum==" + randomNum);
    }
}
```

从 JDK5.0 开始可以写为：

```
import static java.lang.Math.random;
public class Test {
    public static void main(String[] args) {
        double randomNum = random();
        System.out.println("the randomNum==" + randomNum);
    }
}
```

静态引用使我们可以象调用本地方法一样调用一个引入的方法，当我们需要引入同一个类的多个方法时，只需写为“import static java.lang.Math.*”即可。这样的引用方式对于枚举也同样有效。

5.6 final

在 Java 中声明类、属性和方法时，可使用关键字 final 来修饰。final 所标记的成分具有“终态”的特征，表示“最终的”意思。

其具体规定如下：

- final 标记的类不能被继承。
- final 标记的方法不能被子类重写。
- final 标记的变量（成员变量或局部变量）即成为常量，只能赋值一次。
- final 标记的成员变量必须在声明的同时赋值，如果在声明的时候没有赋值，那么只有一次赋值的机会，而且只能在构造方法中显式赋值，然后才能使用。
- final 标记的局部变量可以只声明不赋值，然后再进行一次性的赋值。

final 一般用于标记那些通用性的功能、实现方式或取值不能随意被改变的成分，以避免被误用，

例如实现数学三角方法、幂运算等功能的方法，以及数学常量 π=3.141593、e=2.71828 等。事实上，为确保这终态性，提供了上述方法和常量的 java.lang.Math 类也已被定义为 final 的。

需要注意的是，如果将引用类型（即，任何类的类型）的变量标记为 final，那么该变量不能指向任何其它对象。但可以改 0 变

对象的内容，因为只有引用本身是 final 的。

如果变量被标记为 final，其结果是使它成为常数。想改变 final 变量的值会导致一个编译错误。下面是一个正确定义 final 变量的例子：

```
public final int MAX_ARRAY_SIZE = 25;
```

例 final 关键字程序：Test.java

```
public final class Test {
    public static final int TOTAL_NUMBER = 5;
    public int id;
    public Test() {
        id = ++TOTAL_NUMBER;// 非法，对final变量TOTAL_NUMBER进行二次赋值了。
        // 因为++TOTAL_NUMBER相当于：TOTAL_NUMBER=TOTAL_NUMBER+1
    }
    public static void main(String[] args) {
        final Test t = new Test();
        final int i = 10;
        final int j;
        j = 20;
        j = 30; // 非法，对final变量进行二次赋值
```

```
    }
}
```

Java 编程语言允许关键字 final 被应用到类上（放在 class 关键字前面）。如果这样做了， 类便不能被再派生出子类。比如，类 Java.lang.String 就是一个 final 类。这样做是出于安全原因，因为它保证，如果方法有字符串的引用，它肯定就是类 String 的字符串，而不是某个其它类的字符串，这个类是 String 的被修改过的子类，因为 String 可能被恶意窜改过。

方法也可以被标记为 final。被标记为 final 的方法不能被覆盖。这是由于安全原因。如果方法具有不能被改变的实现，而且对于对象的一致状态是关键的，那么就要使方法成为 final。被声明为 final 的方法有时被用于优化。编译器能产生直接对方法调用的代码，而不是通常的涉及运行时查找的虚拟方法调用。被标记为 static 或 private 的方法被自动地 final，因为动态联编在上述两种情况下都不能应用。

思考题：
使用 final 修饰的变量是最终的不可修改，如果变量指向引用数据类型，请问是变量的引用不可以修改，还是引用的对象不可以修改。

5.7 内部类

5.7.1 什么是内部类

内部类（Inner Classes）的概念是在 JDK1.1 版本中开始引入的。在 Java 中，允许在一个类（或方法、语句块）的内部定义另一个类，称为内部类，有时也称为嵌套类（Nested Classes）。内部类和外层封装它的类之间存在逻辑上的所属关系，一般只用在定义它的类或语句块之内，实现一些没有通用意义的功能逻辑，在外部引用它时必须给出完整的名称。 引入内部类的好处在于可使源代码更加清晰并减少类的命名冲突，就好比工厂制定内部通用的产品或工艺标准，可以取任何名称而不必担心和外界的标准同名，因为其使用范围不同。内部类是一个有用的特征，因为它们允许将逻辑上同属性的类组合到一起，并在另一个类中控制一个类的可视性。下述例子表示使用内部类的共同方法：

```
class MyFrame extends Frame {
    Button myButton;
    TextArea myTextArea;
    int count;
    public MyFrame(String title) {
        super(title);
        myButton = new Button("click me");
        myTextArea = new TextArea();
        add(myButton, BorderLayout.CENTER);
        add(myTextArea, BorderLayout.NORTH);
        ButtonListener bList = new ButtonListener();
        myButton.addActionListener(bList);
```

```
    }
    class ButtonListener implements ActionListener{ // 这里定义了一个内部类
        public void actionPerformed(ActionEvent e) {
            count++;
            myTextArea.setText("button clicked " + count + " times");
        }
    } // end of innerclass ButtonListener
    public static void main(String args[]) {
        MyFrame f = new MyFrame("Inner Class Frame");
        f.setSize(300, 300);
        f.setVisible(true);
    }
}
```

前面的例子包含一个类 MyFrame，它包括一个内部类 ButtonListener。编译器生成两个类文件，MyFrame$ButtonListener.class 以及 MyFrame.class。

5.7.2 内部类特点

（1）嵌套类（内部类）可以体现逻辑上的从属关系。同时对于其他类可以控制内部类对外不可见等。

（2）外部类的成员变量作用域是整个外部类，包括嵌套类。但外部类不能访问嵌套类的 private 成员

（3）逻辑上相关的类可以在一起，可以有效的实现信息隐藏。

（4）内部类可以直接访问外部类的成员。可以用此实现多继承！

（5）编译后，内部类也被编译为单独的类，不过名称为 outclass$inclass 的形式。

再来个例子：

```
public class Outer {
    private int size;
    public class Inner {
        private int counter = 10;
        public void doStuff() {
            size++;
        }
    }
    public static void main(String args[]) {
        Outer outer = new Outer();
        Inner inner = outer.new Inner();
        inner.doStuff();
        System.out.println(outer.size);
        System.out.println(inner.counter);
        // 编译错误，外部类不能访问内部类的private 变量
```

```
        System.out.println(counter);
    }
}
```

5.7.3 内部类的分类

内部类按照使用上可以分为四种情形：

（1）类级：成员式，有 static 修饰

（2）对象级：成员式，普通，无 static 修饰

（3）本地内部类：局部式

（4）匿名级：局部式

5.7.3.1 成员式内部类

内部类可以作为外部类的成员，示例如下：

```
public class Outer1 {
    private int size;
    public class Inner {
        public void dostuff() {
            size++;
        }
    }
    public void testTheInner() {
        Inner in = new Inner();
        in.dostuff();
    }
}
```

成员式内部类如同外部类的一个普通成员。

成员式内部类的基本规则

（1）可以有各种修饰符，可以用 4 种权限、static、final、abstract 定义（这点和普通的类是不同的）；

（2）若有 static 限定，就为类级，否则为对象级。类级可以通过外部类直接访问；对象级需要先生成外部的对象后才能访问。

（3）内外部类不能同名

（4）非静态内部类中不能声明任何 static 成员

（5）内部类可以互相调用，如下：

```
class A {
    // B、C 间可以互相调用
    class B {
    }
    class C {
    }
}
```

成员式内部类的访问

内部类的对象以属性的方式记录其所依赖的外层类对象的引用，因而可以找到该外层类对象并访问其成员。该属性是系统自动为非 static 的内部类添加的，名称约定为“外层类名.this”。

在其它场合则必须先获得外部类的对象，再由外部类对象加“.new”操作符调用内部类的构造方法创建内部类的对象，此时依赖关系的双方也可以明确。这样要求是因为：外部类的 static 方法中不存在当前对象，或者其它无关类中方法的当前对象类型不符合要求。

（1）在另一个外部类中使用非静态内部类中定义的方法时，要先创建外部类的对象，再创建与外部类相关的内部类的对象，再调用内部类的方法，如下所示：

```
class Outer2 {
    private int size;
    class Inner {
        public void dostuff() {
            size++;
        }
    }
}
class TestInner {
    public static void main(String[] args) {
        Outer2 outer = new Outer2();
        Outer2.Inner inner = outer.new Inner();
        inner.dostuff();
    }
}
```

（2）static 内部类相当于其外部类的 static 成分，它的对象与外部类对象间不存在依赖关系，因此可直接创建。示例如下：

```
class Outer2 {
    private static int size;
    static class Inner {
        public void dostuff() {
            size++;
            System.out.println("size=" + size);
        }
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Outer2.Inner inner = new Outer2.Inner();
        inner.dostuff();
    }
}
```

程序运行结果为：

```
size=1
```

（3）由于内部类可以直接访问其外部类的成分，因此当内部类与其外部类中存在同名属性或方法时，也将导致命名冲突。所以在多层调用时要指明，如下所示：

```
public class Outer3{
    private int size;
    public class Inner{
        private int size;
        public void dostuff(int size) {
            size++;   // 本地的 size;
            this.size; // 内部类的 size
            Outer3.this.size++;      // 外部类的 size
        }
    }
}
```

5.7.3.2 本地内部类

本地类(Local class)是定义在代码块中的类。它们只在定义它们的代码块中是可见的。

本地类有几个重要特性：

（1）仅在定义了它们的代码块中是可见的；

（2）可以使用定义它们的代码块中的任何本地 final 变量；

（3）本地类不可以是 static 的，里边也不能定义 static 成员。

（4）本地类不可以用 public、private、protected 修饰，只能使用缺省的。

（5）本地类可以是 abstract 的。

示例如下：

```
public final class Outter {
    public static final int TOTAL_NUMBER = 5;
    public int id = 123;
    public void t1() {
        final int a = 15;
        String s = "t1";
```

```
        class Inner {
            public void innerTest() {
                System.out.println(TOTAL_NUMBER);
                System.out.println(id);
                System.out.println(a);
                // System.out.println(s);不合法，只能访问本地方法的final变量
            }
        }
        new Inner().innerTest();
    }
    public static void main(String[] args) {
        Outter t = new Outter();
        t.t1();
    }
}
```

5.7.3.3 匿名内部类

匿名内部类是本地内部类的一种特殊形式，也就是没有变量名指向这个类实例，而且具体的类实现会写在这个内部类里面。把上面的例子改造一下，如下所示：

```
public final class Test {
    public static final int TOTAL_NUMBER = 5;
    public int id = 123;

    public void t1() {
        final int a = 15;
        String s = "t1";

        new Aclass() {
            public void testA() {
                System.out.println(TOTAL_NUMBER);
                System.out.println(id);
                System.out.println(a);
                // System.out.println(s);不合法，只能访问本地方法的final变量
            }
        }.testA();
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.t1();
    }
}
```

注意：匿名内部类是在一个语句里面，所以后面需要加“;”。

匿名类的规则：

（1）匿名类没有构造方法；

（2）匿名类不能定义静态的成员；

（3）匿名类不能用 4 种权限、static、final、abstract 修饰；

（4）只可以创建一个匿名类实例

再次示例：

```
public class Outter {
    public Contents getCont() {
        return new Contents() {
            private int i = 11;

            public int value() {
                return i;
            }
        };
    }
    public static void main(String[] args) {
        Outter p = new Outter();
        Contents c = p.getCont();
    }
}
```

5.7.4 内部类规则小结

总结一下，内部类有如下特点：

（1）类名称只能用在定义过的范围中，除非用在限定的名称中。内部类的名称必须与所嵌套的类不同。

（2）内部类可以被定义在方法中。这条规则较简单，它支配到所嵌套类方法的变量的访问。任何变量，不论是本地变量还是正式参数，如果变量被标记为 final，那么，就可以被内部类中的方法访问。

（3）内部类可以使用所嵌套类的类变量和实例变量以及所嵌套的块中的本地变量。

（4）内部类可以被定义为 abstract。

（5）只有内部类可以被声明为 private 或 protected，以便防护它们不受来自外部类的访问。访问保护不阻止内部类使用其它类的任何成员，只要一个类嵌套另一个。

（6）一个内部类可以作为一个接口，由另一个内部类实现。

（7）被自动地声明为 static 的内部类成为顶层类。这些内部类失去了在本地范围和其它内部类中使用数据或变量的能力。

（8）内部类不能声明任何 static 成员；只有顶层类可以声明 static 成员。因此，一个需求 static 成员的内部类必须使用来自顶层类的成员。

5.8 再谈 Java 内存分配

Java 程序运行时的内存结构分成：方法区、栈内存、堆内存、本地方法栈几种。栈和堆都是数据结构的知识，如果不清楚，没有关系，就当成一个不同的名字就好了，下面的讲解不需要用到它们具体的知识。

5.8.1 方法区

方法区存放装载的类数据信息包括：

（1）基本信息：

- 每个类的全限定名
- 每个类的直接超类的全限定名(可约束类型转换)
- 该类是类还是接口
- 该类型的访问修饰符
- 直接超接口的全限定名的有序列表

（2）每个已装载类的详细信息：

- 运行时常量池：存放该类型所用的一切常量(直接常量和对其它类型、字段、方法的符号引用)，它们以数组形式通过索引被访问，是外部调用与类联系及类型对象化的桥梁。它是类文件(字节码)常量池的运行时表示。
- 字段信息：类中声明的每一个字段的信息(名，类型，修饰符)。
- 方法信息：类中声明的每一个方法的信息(名，返回类型，参数类型，修饰符，方法的字节码和异常表)。
- 静态变量：到类 classloader 的引用：即到该类的类装载器的引用。
- 到类 class 的引用：虚拟机为每一个被装载的类型创建一个 class 实例，用来代表这个被装载的类。

5.8.2 栈内存

Java 栈内存以帧的形式存放本地方法的调用状态(包括方法调用的参数，局部变量，中间结果等)。每调用一个方法就将对应该方法的方法帧压入 Java 栈，成为当前方法帧。当调用结束(返回)时，就弹出该帧。

编译器将源代码编译成字节码(.class)时，就已经将各种类型的方法的局部变量，操作数栈大小确定并放在字节码中，随着类一并装载入方法区。当调用方法时，通过访问方法区中的类的信息，得到局部变量以及操作数栈的大小。

也就是说：在方法中定义的一些基本类型的变量和对象的引用变量都在方法的栈内存中分配。当在一段代码块定义一个变量时，Java 就在栈中为这个变量分配内存空间，当超过变量的作用域后，Java 会自动释放掉为该变量所分配的内存空间，该内存空间可以立即被另作它用。

栈内存的构成：

Java 栈内存由局部变量区、操作数栈、帧数据区组成。

（1）局部变量区为一个以字为单位的数组，每个数组元素对应一个局部变量的值。 调用方法时，将方法的局部变量组成一个数组，通过索引来访问。若为非静态方法，则加入一个隐含的引用参数this,该参数指向调用这个方法的对象。而静态方法则没有this参数。 因此，对象无法调用静态方法。

（2）操作数栈也是一个数组，但是通过栈操作来访问。所谓操作数是那些被指令操作的数据。当需要对参数操作时如 a=b+c,就将即将被操作的参数压栈，如将 b 和 c 压栈，然后由操作指令将它们弹出，并执行操作。虚拟机将操作数栈作为工作区。

（3）帧数据区处理常量池解析，异常处理等

5.8.3 堆内存

堆内存用来存放由 new 创建的对象和数组。在堆中分配的内存，由 Java 虚拟机的自动垃圾回收器来管理。 在堆中产生了一个数组或对象后，还可以在栈中定义一个特殊的变量，让栈中这个变量的取值等于数组或对象在堆内存中的首地址，栈中的这个变量就成了数组或对象的引用变量。引用变量就相当于是为数组或对象起的一个名称，以后就可以在程序中使用栈中的引用变量来访问堆中的数组或对象。

栈内存和堆内存比较

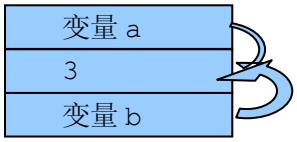
栈与堆都是 Java 用来在内存中存放数据的地方。与 C++不同，Java 自动管理栈和堆，程序员不能直接地设置栈或堆。

Java 的堆是一个运行时数据区，对象从中分配空间。堆的优势是可以动态地分配内存大小，生存期也不必事先告诉编译器，因为它是在运行时动态分配内存的，Java 的垃圾收集器会自动收走这些不再使用的数据。但缺点是，由于要在运行时动态分配内存，存取速度较慢。

栈的优势是，存取速度比堆要快，仅次于寄存器，栈数据可以共享。但缺点是，存在栈中的数据大小与生存期必须是确定的，缺乏灵活性。栈中主要存放一些基本类型的变量（int, short, long, byte, float, double, boolean, char）和对象句柄。 栈有一个很重要的特殊性，就是存在栈中的数据可以共享。假设我们同时定义：

```
int a = 3;
int b = 3;
```

编译器先处理 int a=3;首先它会在栈中创建一个变量为 a 的引用，然后查找栈中是否有 3 这个值，如果没找到，就将 3 存放进来，然后将 a 指向 3。接着处理 int b=3;在创建完 b 的引用变量后，因为在栈中已经有 3 这个值，便将 b 直接指向 3。这样，就出现了 a 与 b 同时均指向 3 的情况。内存示意图如下：



这时，如果再令 b=4;那么编译器会重新搜索栈中是否有 4 值，如果没有，则将 4 存放进来，并令 a 指向 4；如果已经有了，则直接将 a 指向这个地址。因此 a 值的改变不会影响到 b 的值。要注意这种数据的共享与两个对象的引用同时指向一个对象的这种共享是不同的，因为这种情况 a 的修改并不会影响到 b，它是由编译器完成的，它有利于节省空间。此时的内存分配示意图如下：

而一个对象引用变量修改了这个对象的内部状态，会影响到另一个对象引用变量。

5.8.4 本地方法栈内存

当一个线程调用本地方法时，它就不再受到虚拟机关于结构和安全限制方面的约束，它既可以访问虚拟机的运行期数据区，也可以使用本地处理器以及任何类型的栈。例如，本地栈是一个 C 语言的栈，那么当 C 程序调用 C 函数时，函数的参数以某种顺序被压入栈，结果则返回给调用函数。在实现 Java 虚拟机时，本地方法接口使用的是 C 语言的模型栈，那么它的本地方法栈的调度与使用则完全与 C 语言的栈相同。

Java 通过 Java 本地接口 JNI（Java Native Interface）来调用其它语言编写的程序，在 Java 里面用 native 修饰符来描述一个方法是本地方法。这个了解一下就好了，在我们的课程中不会涉及到。

5.8.5 String 的内存分配

String 是一个特殊的包装类数据。可以用：

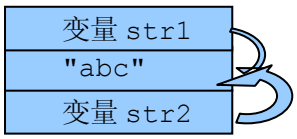
```
String str = new String("abc");
String str = "abc";
```

两种的形式来创建，第一种是用 new()来新建对象的，它会在存放于堆中。每调用一次就会创建一个新的对象。而第二种是先在栈中创建一个对 String 类的对象引用变量 str，然后查找栈中有没有存放"abc"，如果没有，则将"abc"存放进栈，并令 str 指向"abc"，如果已经有"abc" 则直接令 str 指向"abc"。

比较类里面的数值是否相等时，用 equals()方法；当测试两个包装类的引用是否指向同一个对象时，用== =，下面用例子说明上面的理论。

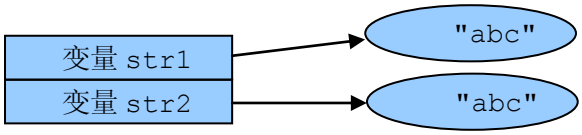
```
String str1 = "abc";
String str2 = "abc";
System.out.println(str1==str2); //true
```

可以看出 str1 和 str2 是指向同一个对象的。




```
String str1 = new String ("abc");
String str2 = new String ("abc");
System.out.println(str1==str2); // false
```

用 new 的方式是生成不同的对象。每一次生成一个。



因此用第一种方式创建多个"abc"字符串，在内存中其实只存在一个对象而已。这种写法有利于节省内存空间。同时它可以在一定程度上提高程序的运行速度，因为 JVM 会自动根据栈中数据的实际情况来决定是否有必要创建新对象。而对于 String str = new String("abc"); 的代码，则一概在堆中创建新对象，而不管其字符串值是否相等，是否有必要创建新对象，从而加重了程序的负担。

另一方面， 要注意:我们在使用诸如 String str = "abc"; 的格式时，总是想当然地认为，创建了 String 类的对象 str。担心陷阱！对象可能并没有被创建！而可能只是指向一个先前已经创建的对象。只有通过 new() 方法才能保证每次都创建一个新的对象。 由于 String 类的值不可变性（immutable），当 String 变量需要经常变换其值时，应该考虑使用 StringBuffer 或 StringBuilder 类，以提高程序效率。

5.9 学习目标

1. OOP（面向对象）语言的三大特征是？
2. 理解什么是封装，封装的好处。
3. 理解什么是类的继承，继承的好处。
4. 理解 is-a 关系。
5. 理解父子类的初始化顺序。
6. 理解类的单继承性。
7. 理解构造方法不能被继承。
8. super 关键字的用法，为什么使用 super？
9. 什么是方法的覆盖？什么情况下出现？
10. 方法覆盖的规则？
11. 什么是重载？什么情况下出现？
12. 简述重载的规则
13. 如何调用父类的构造函数？如何调用自己的构造函数？
14. 如何确定在多态的调用中，究竟是调用的那个方法？
15. Instanceof 运算符的用法
16. static 修饰符的功能是？可以用在什么地方？怎么访问？
17. static 的基本规则
18. final 修饰符的功能是？可以用在什么地方？
19. final 的基本规则
20. java 中什么数据存储在堆内存，什么数据存储在栈内存。
21. 堆内存有什么特点，栈内存有什么特点。

5.10 练习

1. 创建一个构造方法重载的类，并用另一个类调用。

2. 创建 Rodent（啮齿动物）：Mouse（老鼠），Gerbil（鼹鼠），Hamster（大颊鼠）等的一个继承分级结构。在基础类中，提供适用于所有 Rodent 的方法，并在衍生类中覆盖它们，从而根据不同类型的 Rodent 采取不同的行动。创建一个 Rodent 数组，在其中填充不同类型的 Rodent，然后调用自己的基础类方法，看看会有什么情况发生。

3. 编写 MyPoint 的一个子类 MyXYZ，表示三维坐标点，重写 toString 方法用来显示这个对象的 x、y、z 的值，如显示（1，2，3），最后用 main 方法测试。

4. 当你试图编译和执行下面的程序时会发生什么？

```
class Mystery {
    String s;
    public static void main(String[] args) {
        Mystery m = new Mystery();
        m.go();
    }
    void Mystery() {
        s = "constructor";
    }
    void go() {
        System.out.println(s);
    }
}
```

选择下面的正确答案：

A. 编译不通过
B. 编译通过但运行时产生异常
C. 代码运行但屏幕上看不到任何东西
D. 代码运行，屏幕上看到 constructor
E. 代码运行，屏幕上看到 null

5. 当编译和运行下列程序段时，会发生什么？

```
class Person {
}
class Woman extends Person {
}
class Man extends Person {
}
public class Test {
    public static void main(String argv[]) {
        Man m = new Man();
        Woman w = (Woman) new Man();
```

```
    }
}
```

A. 通过编译和并正常运行。
B. 编译时出现例外。
C. 编译通过，运行时出现例外。
D. 编译不通过

6. 对于下列代码：

```
1 class Person {
2     public void printValue(int i, int j) {...}
3     public void printValue(int i){ ...}
4 }
5 public class Teacher extends Person {
6     public void printValue() {  ...}
7     public void printValue(int i) { ...}
8     public static void main(String args[]){
9         Person t = new Teacher();
10        t.printValue(10);
11    }
12 }
```

第 10 行语句将调用哪行语句?

A. line 2
B. line 3
C. line 6
D. line 7

7. 下列代码运行结果是什么?

```
public class Bool {
    static boolean b;
    public static void main(String[] args) {
        int x = 0;
        if (b) {
            x = 1;
        } else if (b == false) {
            x = 2;
        } else if (b) {
            x = 3;
        } else {
            x = 4;
        }
        System.out.println("x= " + x);
    }
}
```

8. 完成此段代码可以单独添加哪些选项？

```
1. public class Test {
2.
3.    public static void main(String[] args) {
4.
5.       System.out.println("c=" + c);
6.    }
7. }
```

A. 在第 2 行加上语句 static char c;
B. 在第 2 行加上语句 char c;
C. 在第 4 行加上语句 static char c;
D. 在第 4 行加上语句 char c='f';

9. 下列代码运行结果是什么？

```
public class A {
    public static void main(Stirng[] args) {
        int m = 2;
        int p = 1;
        int t = 0;
        for (; p < 5; p++) {
            if (t++ > m) {
                m = p + t;
            }
        }
        System.out.println("t equals" + t);
    }
}
```

A. 2
B. 4
C. 6
D. 7

10. 设计个 Circle 类，其属性为圆心点（类型为前面设计的类 MyPoint）和半径，并为此类编写以下三个方法：
一是计算圆面积的 calArea()方法；
二是计算周长的 calLength()；
三是 boolean inCircle(MyPoint mp)方法，功能是测试作为参数的某个点是否在当前对象圆内（圆内，包括圆上返回 true；在圆外，返回 false。

6 数组和枚举

6.1 数组的声明和创建

6.1.1 数组的声明

数组是由相同类型的若干项数据组成的一个数据集合，数组中的每个数据称为元素。也就是说数组是用来集合相同类型的对象，可以是原始数据类型或引用数据类型。

数组声明实际是创建一个引用，通过代表引用的这个名字来引用数组。数组声明格式如下：

```
数据类型 标识符[]
```

示例：

```
char s[];    //  声明一个数据类型为字符型的数组s
Point arr[]; //  声明一个数据类型为Point的数组arr
```

在 Java 编程语言中，数组是一个对象，声明不能创建对象本身，而创建的是一个引用，该引用可被用来引用数组。数组元素使用的实际内存可由 new 语句或数组初始化软件动态分配。在后面，你将看到如何创建和初始化实际数组。

上述这种将方括号置于变量名之后的声明数组的格式，是用于 C、C++和 Java 编程语言的标准格式。这种格式会使声明的格式复杂难懂，因而，Java 编程语言允许一种替代的格式，该格式中的方括号位于变量名的左边：

```
char[] s;
Point[] arr;
```

这样的结果是，你可以认为类型部分在左，而变量名在右。上述两种格式并存，你可选择一种你习惯的方式。声明不指出数组的实际大小。

注意----当数组声明的方括号在左边时，该方括号可应用于所有位于其右的变量。

6.1.2 创建数组

数据对象和其他 Java 对象一样，使用关键字 new 创建。创建的时候要指明数组的长度。

```
s = new char [20];
p = new Point [100];
```

第一行创建了一个 20 个 char 类型元素的数组，在堆区为数组分配内存空间，每个元素都是 char 类型的，占 2 个字节，因此整个数组对象再内存中占用 40 个字节。为每个元素赋予其数据类型的默认值，即'\u0000'。返回数组对象的引用赋值给变量 s。

第二行创建了一个 100 个类型 Point 的变量，然而，它并不是创建 100 个 Point 对象；创

建 100 个对象的工作必须分别完成如下：

```
p[0] = new  Point();
p[1] = new  Point();
……
……
```

用来指示单个数组元素的下标必须总是从 0 开始，并保持在合法范围之内——大于 0 或等于 0 并小于数组长度。任何访问在上述界限之外的数组元素的企图都会引起运行时出错。

数组的下标也称为数组的索引，必须是整数或者整数表达式，如下：

```
int i[] = new  int[(9 - 2) * 3];// 这是合法的
```

其实，声明和创建可以定义到一行，而不用分开写。

6.1.3 数组的初始化

当创建一个数组时，每个元素都被自动使用默认值进行初始化。在上述 char 数组 s 的例子中，每个值都被初始化为 0 (\u0000-null)字符；在数组 p 的例子中， 每个值都被初始 化为 null，表明它还未引用一个 Point 对象。在经过赋值 p[0]=new Point()之后，数组 的第一个元素引用为实际 Point 对象。

注意——所有变量的初始化(包括数组元素)是保证系统安全的基础，变量绝不能在未初始化状态使用。

Java 编程语言允许使用下列形式快速创建数组，直接定义并初始化：

```
String names[] = { "Georgianna", "Jen", "Simon" };
```

其结果与下列代码等同：

```
String names[];
names = new String[3];
names[0]="Georgianna";
names[1]="Jen";
names[2]="Simon";
```

这“速记”法可用在任何元素类型。例如：

```
Myclass array[] = {
            new Myclass(),
            new Myclass(),
            new Myclass()
            };
```

适当的类类型的常数值也可被使用：

```
import java.awt.Color;
   Color palette [] ={
                Color.blue,
                Color.red,
                Color.white
                  };
```

6.1.4 数组的内存分配

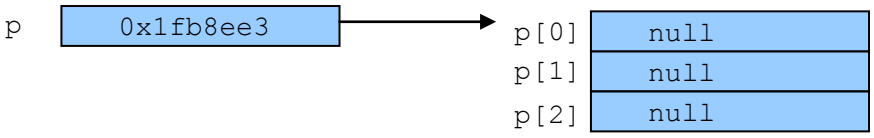
数组一旦被创建，在内存里面占用连续的内存地址。

数组还具有一个非常重要的特点—数组的静态性：数组一旦被创建，就不能更改数组的长度。

比如，定义数组如下：

```
Point[] p = new Point [3];
```

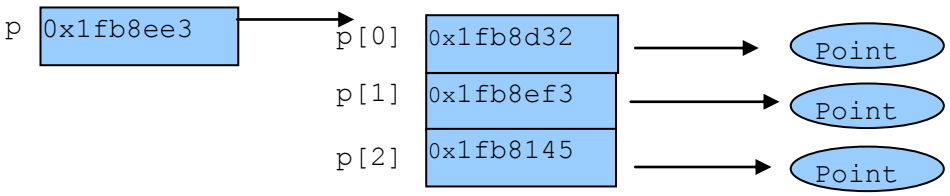
其中 p 是数组名，数组长度是 3，数组在被创建的时候，内存示意图如下：



为数组初始化：

```
p[0] = new Point() ;
p[1] = new Point() ;
p[2] = new Point() ;
```

内存示意图如下：



6.2 数组元素的访问

6.2.1 使用 length 属性

在 Java 编程语言中，所有数组的下标都从 0 开始。 一个数组中元素的数量被作为具有 length 属性的部分数组对象而存储； 这个值被用来检查所有运行时访问的界限。如果发生了一个越出界限的访问，那么运行时的报错也就出现了。使用 length 属性的例子如下：

```
int[] list =new int [10];
for(int i=0;i<list.length;i++){
   System.out.println( list[i] );
}
```

使用 length 属性使得程序的维护变得更简单。

所有元素的访问就通过数组的下标来访问，如上例的 list[i]，随着 i 的值发生变化，就依次访问 list[0]、list[1]、list[2]…

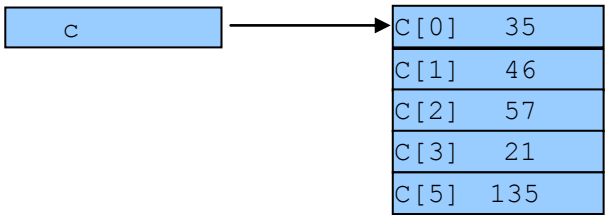
如果想要给某个数组元素赋值，如下方式：

```
list[0]=5;  list[1]=6;…
```

示例：假如定义一个数组：

```
int c [] = new int[5];
……//进行赋值的语句
```

对数组进行赋值后，内存示意图如下：



然后就可以根据数组名[下标]来取值了。如：

```
int  a  =  c[3];
```

结果就是：从数组中取出下标为 3 的元素的值“21”,然后赋值给 a。

6.2.2 更优化的 for 循环语句

在访问数组的时候，经常使用 for 循环语句。从 JDK5.0 开始，提供了一个更好的 for 循环语句的写法，示例如下：

```
public class Test {
    public static void main(String args[]) {
        int a[] = new int[3];
        // 旧的写法，赋值
        for (int i = 0; i < a.length; i++) {
            a[i] = i;
        }
        // 新的写法，取值
        for (int i : a) {
            System.out.println(i);
        }
    }
}
```

显然 JDK5.0 版本的写法比以前是大大简化了。

6.3 多维数组

6.3.1 多维数组的基础知识

Java 编程语言没有象其它语言那样提供多维数组。因为一个数组可被声明为具有任何基础类型，所以你可以创建数组的数组(和数组的数组的数组，等等)。一个二维数组如下例所示：

```
int twoDim[][] = new int[4][];
twoDim[0] = new int[5];
twoDim[1] = new int[5];
```

首次调用 new 而创建的对象是一个数组，它包含 4 个元素，每个元素对类型 array of int 的元素都是一个 null 引用并且必须将数组的每个点分别初始化。

因为这种对每个元素的分别初始化，所以有可能创建非矩形数组的数组。也就是说，twoDim 的元素可按如下方式初始化：

```
twoDim[0] = new int [2];
twoDim[1] = new int [4];
twoDim[2] = new int [6];
twoDim[3] = new int [8];
```

由于此种初始化的方法烦琐乏味，而且矩形数组的数组是最通用的形式，因而产生了一种“速记”方法来创建二维数组。例如：

```
int twoDim[][] = new int [3][4];
```

可被用来创建一个每个数组有 4 个整数类型的 3 个数组的数组。可以理解成为如下表格所示：

```
twoDim[0][0]    twoDim[0][1]    twoDim[0][2]    twoDim[0][3]
twoDim[1][0]    twoDim[1][1]    twoDim[1][2]    twoDim[1][3]
twoDim[2][0]    twoDim[2][1]    twoDim[2][2]    twoDim[2][3]
```

第一个方括号中的数字表示行，第二个方括号中的数字表示列，注意都是从 0 开始的。

尽管声明的格式允许方括号在变量名左边或者右边，但此种灵活性不适用于数组句法的其它方面。例如： new int [][4]是非法的。

6.3.2 示例

```
class FillArray {
   public static void main(String args[]) {
      int[][] matrix = new int[4][5]; // 二维数组的声明和创建
      for (int row = 0; row < 4; row++) {
         for (int col = 0; col < 5; col++) {
```

```
                matrix[row][col] = row + col; // 二维数组的访问，为元素赋值
            }
        }
    }
}
```

当然也可以直接定义并赋值，如下：

```
    double[][] c = {
                { 1.0, 2.0, 3.0, 4.0 },
                { 0.0, 1.0, 0.0, 0.0 },
                { 0.0, 0.0, 1.0, 0.0 }
                };
```

从上面可以看得很清楚，二维数组其实就是一维的一维数组。

6.3.3 多维数组的本质

N 维数组就是一维的 N-1 维数组，比如：三维数组就是一维的二维数组。

三维以至多维数组都是一个思路，一维数组—〉二维数组—〉三维数组的实例：

```
class Fill3DArray {
    public static void main(String args[]) {
        int[][][] M = new int[4][5][3];
        for (int row = 0; row < 4; row++) {
            for (int col = 0; col < 5; col++) {
                for (int ver = 0; ver < 3; ver++) {
                    M[row][col][ver] = row + col + ver;
                }
            }
        }
    }
}
```

6.4 数组的复制

数组一旦创建后，其大小不可调整。然而，你可使用相同的引用变量来引用一个全新的数组：

```
    int myArray []= new int [6];
    myArray = new int [10];
```

在这种情况下，第一个数组被有效地丢失，除非对它的其它引用保留在其它地方。

Java 编程语言在 System 类中提供了一种特殊方法拷贝数组，该方法被称作 arraycopy()。例如，araycopy 可作如下使用：

```
    // 原始数组
```

```
    int myArray[] = { 1, 2, 3, 4, 5, 6 };
    // 新的数组，比原始数组大
    int hold[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
    // 把原始数组的值拷贝到新的数组
    System.arraycopy(myArray, 0, hold, 0, myArray.length);
```

拷贝完成后，数组 hold 有如下内容：1,2,3,4,5,6,4,3,2,1。

注意–在处理对象数组时，System.arraycopy()拷贝的是引用，而不是对象。对象本身不改变。

6.5 数组的排序

6.5.1 基本的排序算法

6.5.1.1 冒泡排序

对几个无序的数字进行排序，比较常用的方法是冒泡排序法。冒泡法排序是一个比较简单的排序方法，在待排序的数列基本有序的情况下排序速度较快。

基本思路：对未排序的各元素从头到尾依次比较相邻的两个元素是否逆序(与欲排顺序相反)，若逆序就交换这两元素，经过第一轮比较排序后便可把最大（或最小）的元素排好，然后再用同样的方法把剩下的元素逐个进行比较，就得到了你所要的顺序。

可以看出如果有 N 个元素，那么一共要进行 N-1 轮比较，第 I 轮要进行 N-I 次比较。（如： 有 5 个元素，则要进行 5-1 轮比较。第 3 轮则要进行 5-3 次比较）示例如下：

```
public class Test3 {
    public static void main(String[] args) {
        // 需要排序的数组，目前是按照升序排列的
        int a[] = new int[5];
        a[0] = 3;
        a[1] = 4;
        a[2] = 1;
        a[3] = 5;
        a[4] = 2;
        // 冒泡排序
        for (int i = 0; i < a.length-1; i++) {
            for (int j = 0; j < a.length-1; j++) {
                // 判断相邻的两个元素是否逆序
                if (a[j] > a[j+1]) {
                    int temp = a[j];
                    a[j] = a[j+1];
                    a[j+1] = temp;
                }
            }
        }
```

```
        // 检测一下排序的结果
        for (int i : a) {
            System.out.println("i=" + i);
        }
    }
}
```

运行结果：

```
i=1
i=2
i=3
i=4
i=5
```

如果你想要按照降序排列，很简单，只需把：if(a[j] > a[j+1])改成：if(a[j] < a[j+1]) 就可以了。

6.5.1.2 选择排序

基本思路：从所有元素中选择一个最小元素 a[i]放在 a[0]（即让最小元素 a[i]与 a[0]交换），作为第一轮；第二轮是从 a[1]开始到最后的各个元素中选择一个最小元素，放在 a[1]中；……依次类推。n 个数要进行（n－1）轮。比较的次数与冒泡法一样多，但是在每一轮中只进行一次交换，比冒泡法的交换次数少，相对于冒泡法效率高。示例如下：

```
public class Test {
    public static void main(String[] args) {
        // 需要排序的数组，目前是按照升序排列的
        int a[] = new int[5];
        a[0] = 3;
        a[1] = 4;
        a[2] = 1;
        a[3] = 5;
        a[4] = 2;
        // 选择法排序
        int temp;
        for (int i = 0; i < a.length; i++) {
            int lowIndex = i;
            // 找出最小的一个的索引
            for (int j = i + 1; j < a.length; j++) {
                if (a[j] < a[lowIndex]) {
                    lowIndex = j;
                }
            }
            // 交换
            temp = a[i];
            a[i] = a[lowIndex];
            a[lowIndex] = temp;
        }
        // 检测一下排序的结果
        for (int i : a) {
```

```
            System.out.println("i=" + i);
        }
    }
}
```

运行结果：

```
i=1
i=2
i=3
i=4
i=5
```

如果你想要按照降序排列，很简单，只需要把：if (a[j] <a[lowIndex])这句话修改成：if (a[j] > a[lowIndex])就可以了。

6.5.1.3 插入法排序

基本思路：每拿到一个元素，都要将这个元素与所有它之前的元素遍历比较一遍，让符合排序顺序的元素挨个移动到当前范围内它最应该出现的位置。

举个例子来说，就用前面的数组，我们要对一个有 5 个元素的数组进行升序排列，假设第一个元素的值被假定为已排好序了，那么我们就将第 2 个元素与数组中的部分进行比较， 如果第 2 个元素的值较小，则将它插入到第 1 个元素的前面，现在就有两个元素排好序了， 我们再将没有排序的元素与排好序的元素列表进行比较，同样，如果小于第一个元素，就将 它插入到第一个元素前面，但是，如果大于第一个元素的话，我们就将它再与第 2 个元素的 值进行比较，小于的话就排在第 2 个元素前面，大于的话，就排在第 2 个元素的后面。以此类推，直到最后一个元素排好序。

示例如下：

```
public class Test {
    public static void main(String[] args) {
        // 需要排序的数组，目前是按照升序排列的
        int a[] = new int[5];
        a[0] = 3;
        a[1] = 4;
        a[2] = 1;
        a[3] = 5;
        a[4] = 2;
        // 插入法排序
        int temp;
        for (int i = 1; i < a.length; i++) {
            // i=1 开始，因为第一个元素认为 是已经排好序了的
            for (int j = i; (j > 0) && (a[j] < a[j - 1]); j--) {
                // 交换
                temp = a[j];
                a[j] = a[j - 1];
                a[j - 1] = temp;
            }
```

```
        }
        // 检测一下排序的结果
        for (int i : a) {
            System.out.println("i=" + i);
        }
    }
}
```

运行结果：

```
i=1
i=2
i=3
i=4
i=5
```

如果你想要按照降序排列，很简单，只需要把：`if (a[j] <a[lowIndex])`这句话修改成：`if (a[j] > a[lowIndex])`就可以了。

6.5.1.4 希尔(Shell)法排序

从前面介绍的冒泡排序法，选择排序法，插入排序法可以发现，如果数据已经大致排好序的时候，其交换数据位置的动作将会减少。例如在插入排序法过程中，如果某一整数 `d[i]` 不是较小时，则其往前比较和交换的次数会更少。如何用简单的方式让某些数据有一定的大小次序呢？Donald Shell（Shell 排序的创始人）提出了希尔法排序。

基本思路：先将数据按照固定的间隔分组，例如每隔 4 个分成一组，然后排序各分组的数据，形成以分组来看数据已经排序，从全部数据来看，较小值已经在前面，较大值已经在后面。将初步处理了的分组再用插入排序来排序，那么数据交换和移动的次数会减少。可以得到比插入排序法更高的效率。示例如下：

```
public class Test {
    public static void main(String[] args) {
        // 需要排序的数组，按照升序排列
        int a[] = new int[5];
        a[0] = 3;
        a[1] = 4;
        a[2] = 1;
        a[3] = 5;
        a[4] = 2;
        // shell法排序
        int j = 0;
        int temp = 0;
        // 分组
        for(int increment = a.length / 2; increment > 0; increment /= 2)
{
            // 每个组内排序
            for (int i = increment; i < a.length; i++) {
```

```
                temp = a[i];
                for (j = i; j >= increment; j -= increment) {
                    if (temp < a[j - increment]) {
                        a[j] = a[j - increment];
                    } else {
                        break;
                    }
                }
                a[j] = temp;
            }
        }
        // 检测一下排序的结果
        for (int i2 : a) {
            System.out.println("i=" + i2);
        }
    }
}
```

运行结果：

```
i=1
i=2
i=3
i=4
i=5
```

如果你想要按照降序排列，很简单，只需要把：`if (temp < a[j - increment])`这句话修改成：`if (temp > a[j - increment])`就可以了。

6.5.2 数组排序

事实上，数组的排序不用那么麻烦，上面只是想让大家对一些基本的排序算法有所了解而已。在 `java.util.Arrays` 类中有一个静态方法 `sort`，可以用这个类的 `sort` 方法来对数组进行排序。示例如下：

```
public class Test {
    public static void main(String[] args) {
        // 需要排序的数组，目前是按照升序排列的
        int a[] = new int[5];
        a[0] = 3;
        a[1] = 4;
        a[2] = 1;
        a[3] = 5;
        a[4] = 2;
        // 数组排序
        java.util.Arrays.sort(a);
        // 检测一下排序的结果
        for (int i2 : a) {
            System.out.println("i=" + i2);
```

```
        }
    }
}
```

注意：现在的 sort 方法都是升序的，要想实现降序的，还需要 Comparator 的知识，这个在后面会学到。

6.6 数组实用类 Arrays

在 java.util 包中，有一个用于操纵数组的实用类 Arrays。它提供了一系列静态方法，帮助开发人员操作数组。

public static <T> List<T> asList(T...a)

返回一个受指定数组支持的固定大小的列表。（对返回列表的更改会“直接写”到数组。）此方法同 Collection.toArray()一起，充当了基于数组的 API 与基于 collection 的 API 之间的桥梁。返回的列表是可序列化的，并且实现了 RandomAccess。

```
List<Integer> list=Arrays.asList(2,4,6,7,9,10,20);
for(int i=0;i<list.size();i++){
    System.out.print(list.get(i)+",");
}
```

运行结果：2,4,6,7,9,10,20,

public static boolean equals（数组参数 1,数组参数 2）

比较两个数组参数是否相同，数组参数可以是基本数据类型，也可以是引用数据类型。只有当两个数组中的元素数目相同，并且对应位置的元素也相同时，才表示数组相同。如果是引用类型的数组，比较的是引用类型的 equals 方法。示例如下：

```
String[] arr={"Java快车","javakc","JavaKC"};
String[] arr2={"Java快车","javakc","JavaKC"};
System.out.print(Arrays.equals(arr, arr2));
```

运行结果：true。

public static void fill（数组,数据参数）

向数组中填充数据参数，把数组中所有元素的值设置为该数据。数组和数据参数的类型必须一致，或可以自动转化，数组和元素可以是基本数据类型，也可以是引用数据类型。示例如下：

```
//基本数据类型或字符串
String[] arr=new String[5];
Arrays.fill(arr, "Java快车");
for(String s:arr){
    System.out.print(s+",");
}
//引用数据类型，使用了同一个引用
A[] arr2=new A[5];
Arrays.fill(arr2, new A());
for(A a:arr2){
```

```
        System.out.print(a+",");
    }
```

运行结果如下：

```
Java快车,Java快车,Java快车,Java快车,Java快车,
A@5224ee,A@5224ee,A@5224ee,A@5224ee,A@5224ee,
```

public static void fill（数组,int fromIndex, int toIndex,数据参数）

向数组中指定的范围填充数据参数，此范围包含 fromIndex，但不包含 toIndex。数组和数据参数的类型必须一致，或可以自动转化，数组和元素可以是基本数据类型，也可以是引用数据类型。示例如下：

```
String[] arr=new String[5];
Arrays.fill(arr,1,3,"Java快车");
for(String s:arr){
    System.out.print(s+",");
}
```

运行结果如下：

```
null,Java快车,Java快车,null,null,
```

public static int binarySearch（数组,数据参数）

查找数组中元素的值与给定数据相同的元素。数组和数据参数的类型必须一致，或可以自动转化，数组和数据参数可以是基本数据类型，也可以是引用数据类型。

因为此方法采用二分法进行查找数据，所以当调用该方法时，必须保证数组中的元素已经按照升序排列，这样才能得到正确的结果。如果该数组包含此数据参数，则返回对应的数组下标，否则返回一个负数。示例如下：

```
int[] arr={2,4,6,7,9,10,20};
System.out.print(Arrays.binarySearch(arr, 9));
```

运行结果：4。

public static int binarySearch（数组,int fromIndex,int toIndex,数据参数）

在数组中指定的范围查找元素的值与给定数据相同的元素。其他说明如上。

示例如下：

```
double[] arr={2,4,6,7,9,10,20};
System.out.print(Arrays.binarySearch(arr,1,5,10));
```

运行结果：-6。

public static void sort（数组）

把数组中的数组按升序排列。数组可以是基本数据类型，也可以是引用数据类型。

public static void copyOf（数组,int newLength）

赋值指定的数组，截取下标 0（包括）至 newLength（不包括）范围。示例如下：

```
int[] arr={2,4,6,7,9,10,20};
```

```
    int[] arr2=Arrays.copyOf(arr, 4);
    for(int i:arr2){
        System.out.print(i+",");
    }
```

运行结果：2,4,6,7,

public static 数组 copyOfRange（数组,int from,int to）

将数组的指定范围复制到一个新数组。数组可以是基本数据类型，也可以是引用数据类型。示例如下：

```
    int[] arr={2,4,6,7,9,10,20};
    int[] arr2=Arrays.copyOfRange(arr,1,5);
    for(int i:arr2){
        System.out.print(i+",");
    }
```

运行结果：4,6,7,9,

public static String toString（数组）

返回指定数组内容的字符串表示形式。数组可以是基本数据类型，也可以是引用数据类型。示例如下：

```
    int[] arr={2,4,6,7,9,10,20};
    System.out.print(Arrays.toString(arr));
```

运行结果：[2, 4, 6, 7, 9, 10, 20]

6.7 枚举类型

6.7.1 什么是枚举类型

枚举类型 enum 是一种新的类型，在 JDK5.0 加入，允许用常量来表示特定的数据片断，这些数据是分配时预先定义的值的集合，而且全部都以类型安全的形式来表示。

在枚举类型没有加入到 Java 前，我们要想表达常量的集合，通常采用如下的方式：

```
public class Test {
    public static final int A = 1;
    public static final int B = 2;
    public static final int C = 3;
    public static final int D = 4;
    public static final int E = 5;
}
```

那么我们在使用的时候就采用如下代码：

```
    Test.A    或者  Test.B  之类的代码。
```

但是在这样做的时候，我们需要记住这类常量是 Java 中 int 类型的常量，这意味着该方法可以接受任何 int 类型的值，即使它和 Test 中定义的所有级别都不对应。因此需要检测上界和下界，在出现无效值的时候，可能还要包含一个 IllegalArgumentException。而且，如果后来又添加另外一个级别（例如 TEST.F，那么必须改变所有代码中的上界，才能接受这个新值。

换句话说，在使用这类带有整型常量的类时，该解决方案也许可行，但并不是非常有效。

枚举就为处理上述问题提供了更好的方法。把上面的例子改成用枚举的方式：

```
public class Test {
    public enum StudentGrade {
        A, B, C, D, E, F
    };
}
```

可以采用如下的方式进行使用：

```
public class Test {
    public enum StudentGrade {
        A, B, C, D, E, F
    };

    public static void main(String[] args) {
        System.out.println("学生的平均成绩为==" + StudentGrade.B);
    }
}
```

上面的示例都相当简单，但是枚举类型提供的东西远不止这些。您可以逐个遍历枚举值，也可以在 switch 语句中使用枚举值，枚举是非常有价值的。

6.7.2 遍历枚举类型

示例如下：

```
public class Test {
    public enum StudentGrade {
        A, B, C, D, E, F
    };
    public static void main(String[] args) {
        for (StudentGrade score : StudentGrade.values()) {
            System.out.println("学生成绩取值可以为==" + score);
        }
    }
}
```

运行结果：

```
学生成绩取值可以为==A
学生成绩取值可以为==B
学生成绩取值可以为==C
学生成绩取值可以为==D
```

```
学生成绩取值可以为==E
学生成绩取值可以为==F
```

values()方法返回了一个由独立的 StudentGrade 实例构成的数组。

还有一个常用的方法：valueOf(String)：功能是以字符串的形式返回某一个具体枚举元素的值，示例如下：

```
public class Test {
    public enum StudentGrade {
        A, B, C, D, E, F
    };
    public static void main(String[] args) {
        Test t = new Test();
        StudentGrade score = StudentGrade.valueOf("A");
        System.out.println("你的成绩是:" + score);
    }
}
```

运行结果：你的成绩是:A

6.7.3 在 switch 中使用枚举类型

示例如下：

```
public class Test {
    public enum StudentGrade {
        A, B, C, D, E, F
    };

    public static void main(String[] args) {
        Test t = new Test();
        StudentGrade score = StudentGrade.C;
        switch (score) {
        case A:
            System.out.println("你的成绩是优秀");
            break;
        case B:
            System.out.println("你的成绩是好");
            break;
        case C:
            System.out.println("你的成绩是良");
            break;
        case D:
            System.out.println("你的成绩是及格");
            break;
        default:
            System.out.println("你的成绩是不及格");
            break;
        }
    }
```

```
    }
```

运行结果：

```
你的成绩是良
```

在这里，枚举值被传递到 switch 语句中，而每个 case 子句将处理一个特定的值。该值在提供时没有枚举前缀，这意味着不用将代码写成 case StudentGrade.A，只需将其写成 case A 即可，编译器不会接受有前缀的值。

6.7.4 枚举类型的特点

从上面的示例中可以看出，枚举类型大概有如下特点：

- 类型安全
- 紧凑有效的枚举数值定义
- 运行的高效率
- 类型安全

枚举的申明创建了一个新的类型。它不同于其它的已有类型，包括原始类型（整数，浮点数等等）和当前作用域（Scope）内的其它的枚举类型。当你对方法的参数进行赋值操作的时候，整数类型和枚举类型是不能互换的（除非是你进行显式的类型转换），编译器将强制这一点。比如说，用上面申明的枚举定义这样一个方法：

```
public void foo(Day);
```

如果你用整数来调用这个函数，编译器会给出错误的。必须用这个类型的值进行调用。

```
foo(4); // compilation error
```

比较前面写的例子，你看看是枚举定义写得紧凑，还是直接使用 static final 紧凑呢。答案是不言而喻的。

枚举的运行效率和原始类型的整数类型基本上一样高。在运行时不会由于使用了枚举而导致性能有所下降。

6.8 学习目标

1. 熟练使用一维数组的声明、创建、初始化、访问。
2. 了解一维数组的内存分配，能画出内存分配图。
3. 会用 for 循环访问一维数组。
4. 熟练使用二维数组的声明、创建、初始化、访问。
5. 描述多维数组的本质：n 维数组是一维的 n-1 维。
6. 了解二维数组的内存分配，能画出内存分配图。
7. 会用嵌套 for 循环访问二维数组。
8. 能画出引用数据类型的数组的内存分配图。
9. 如何完成数组的复制
10. 了解冒泡排序、选择排序、插入排序、希尔排序算法，最少能默写出其中一种排序算法。
11. 熟练掌握 Arrays 工具类中的方法。能说出大部分方法的方法名称、完成的功能，甚至参数列表、返回值类型、抛出的异常。

6.9 练习

1. 写一个方法，在方法内部定义一个一维的 int 数组，然后为这个数组赋上初始值，最后再循环取值并打印出来。

2. 下面的数组定义那些是正确的？

```
A. int a[ ][ ] = new int[3,3];
B. int a[3][3] = new int[ ][ ];
C. int a[ ][ ] = new int[3][3];
D. int [ ]a[ ] = new int[3][3];
E. int [ ][ ]a = new int[3][3];
```

3. 定义一个长度为 10 的一维字符串数组，在每一个元素存放一个单词；然后运行时从命令行输入一个单词，程序判断数组是否包含有这个单词，包含这个单词就打印出“Yes”，不包含就打印出“No”。

4. 请在下面语句中找出一个正确的。

```
A. int arr1[2][3];
B. int[ ][ ] a2 = new int[2][ ];
C. int[ ][ ] arr2 = new int [ ][4];
D. int arr3[ ][4] = new int [3][4];
```

5. 用二重循环求出二维数组 b 所有元素的和：

```
int[ ][ ] b={{11},{21,22},{31,32,33}}
```

6. 编写一个方法实现将班级同学的名单存放在数组中，并利用随机数(Math.random())随机输出一位同学的姓名。

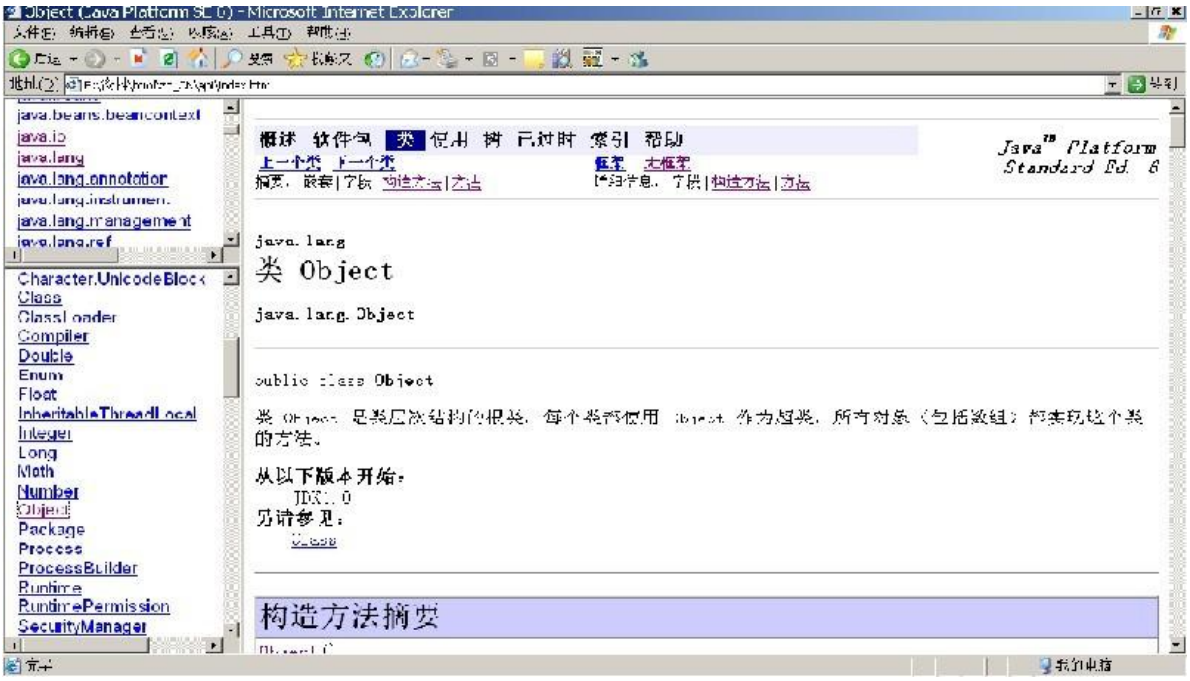
7. 生成一百个随机数，放到数组中，然后排序输出。

8. 统计字符串中英文字母、空格、数字和其它字符的个数。
   提示：String 类中有一个 toCharArray 方法，可以将此字符串转换成一个 char 类型的数组，另外还需要用到 asc 码。

7 常见类的使用

在 java 中有成百上千的类，每个类又有很多不同的方法，要记住所有的类和方法是不可能的。因此，熟练掌握如何查阅 API 是相当重要的。从中可以查到标准库中所有的类及方法。Api 文档是 Java SDK 的一部分，它以 HTML 形式发布。

Api 的主页面有三个窗口组成。左上端的窗口列出了所有可以利用的包，坐下端是所有可用的类，如果点击一个类名，关于这个 Api 的文档将会显示在右边最大的窗格。关于类的声明、属性、构造方法、成员方法及其他们的说明文字都在这个大窗口打开。



下面我们介绍几个在 Java 中常用的类。

7.1 Object 类

java.lang 包中定义的 Object 类是所有 Java 类的根父类，其中定义了一些实现和支持面向对象机制的重要方法。任何 Java 对象，如果没有父类，就默认它继承了 Object 类。因此， 实际上，以前的定义是下面的简略：

```
public class Employee extends Object
public class Manager extends Employee
```

Object 类定义许多有用的方法，包括 toString(),它就是为什么 Java 软件中每样东西都能转换成字符串表示法的原因。（即使这仅具有有限的用途）。

7.1.1 equals 方法

Object 类定义的 equals 方法用于判别某个指定的对象与当前对象（调用 equals 方法

的对象）是否等价。在 Java 语言中数据等价的基本含义是指两个数据的值相等。

在 equals 和“==”进行比较的时候，引用类型数据比较的是引用，即内存地址，基本数据类型比较的是值。

7.1.1.1 equals 方法与==运算符的关系

equals()方法只能比较引用类型，“==”可以比较引用类型及基本类型；

特例：当用 equals()方法进行比较时，对类 File、String、Date 及包装类来说，是比较类 型及内容而不考虑引用的是否是同一个实例；

用“==”进行比较时，符号两边的数据类型必须一致(可自动转换的数据类型除外)，否则编译出错，而用 equals 方法比较的两个数据只要都是引用类型即可。

示例如下：

```
class MyDate {
    private int day, month, year;
    public MyDate(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }
}
public class Test {
    public static void main(String[] args) {
        MyDate m1 = new MyDate(8, 8, 2008);
        MyDate m2 = new MyDate(8, 8, 2008);
        if (m1 == m2) {
            System.out.println("m1==m2");
        } else {
            System.out.println("m1!=m2");
        }
        if (m1.equals(m2)) {
            System.out.println("m1 is equal to m2");
        } else {
            System.out.println("m1 is not equal to m2");
        }
        m2 = m1;
        if (m1 == m2) {
            System.out.println("m1==m2");
        } else {
            System.out.println("m1!=m2");
        }
    }
}
```

程序运行结果为：

```
m1!=m2
m1 is not equal to m2
m==m2
```

小结一下：

在引用类型的比较上，Object 里面的 equals 方法默认的比较方式，基本上等同于“==”，都是比较内存地址，只有那几个特殊的是比较的值。

7.1.1.2 覆盖 equals 方法

对于程序员来说，如果一个对象需要调用 equals 方法，应该在类中覆盖 equals 方法。如果覆盖了 equals 方法，那么具体的比较就按照你的实现进行比较了。

一般来讲：为了比较两个分离的对象（也就是内存地址不同的两个对象），自行覆盖的 equals 方法里面都是检查类型和值是否相同。上面那几个特殊的情况就是这样，比如 String 类，它覆盖了 equals 方法，然后在里面进行值的比较。

覆盖 equals 方法的一般步骤如下：

（1）用==检查是否参数就是这个对象的引用

（2）判断要比较的对象是否为 null，如果是 null，返回 false

（3）用 instanceof 判断参数的类型是否正确

（4）把参数转换成合适的类型

（5）比较对象属性值是不是匹配

示例如下：

覆盖前 equals 和==比较的都是内存地址：

```
public class Test {
    public static void main(String[] args) {
        A a1 = new A();
        a1.id = 3;
        A a2 = new A();
        a2.id = 3;
        System.out.println("a1 == a2 test =" + (a1 == a2));
        System.out.println("a1 equals a2 test =" + a1.equals(a2));
    }
}
class A {
    public int id = 0;
}
```

运行结果是：

```
a1== a2 test =false
a1 equals a2 test =false
```

覆盖后 equals 比较的是值，==比较的是内存地址：

```
public class Test {
    public static void main(String[] args) {
        Test t = new Test();
        A a1 = new A();
        a1.id = 3;
        A a2 = new A();
        a2.id = 3;
        System.out.println("a1 == a2 test =" + (a1 == a2));
        System.out.println("a1 equals a2 test =" + a1.equals(a2));
    }
}
class A {
    public int id = 0;
    public boolean equals(Object obj) {
        // 第一步先判断是否同一个实例
        if (this == obj) {
            return true;
        }
        // 第二步判断要比较的对象是否为null
        if (obj == null) {
            return false;
        }
        // 第三步判断是否同一个类型
        if (obj instanceof A) {
            // 第四步类型相同，先转换成为同一个类型
            A a = (A) obj;
            // 第五步然后进行对象属性值的比较
            if (this.id == a.id) {
                return true;
             } else {
                return false;
            }
        } else {
            // 类型不同，直接返回false
            return false;
        }
    }
}
```

说明：如果对象的属性又是一个引用类型的话，会继续调用该引用类型的 equals 方法，直到最后得出相同还是不同的结果。示例如下：

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

```
        Test t = new Test();
        A a1 = new A();
        a1.id = 3;
        A a2 = new A();
        a2.id = 3;
        System.out.println("a1 == a2 test =" + (a1 == a2));
        System.out.println("a1 equals a2 test =" + a1.equals(a2));
    }
}
class A {
    public int id = 0;
    public String name = "Java快车";
    public boolean equals(Object obj) {
        // 第一步先判断是否同一个实例
        if (this == obj) {
            return true;
        }
        // 第二步判断要比较的对象是否为null
        if (obj == null) {
            return false;
        }
        // 第三步判断是否同一个类型
        if (obj instanceof A) {
            // 第四步类型相同，先转换成为同一个类型
            A a = (A) obj;
            // 第五步然后进行对象属性值的比较
            if (this.id == a.id && this.name.equals(a.name)) {
                return true;
            } else {
                return false;
            }
        } else {
            // 类型不同，直接返回false
            return false;
        }
    }
}
```

最后重要的一点规则：覆盖 equals 方法应该连带覆盖 hashCode 方法。

7.1.2 hashCode 方法

hashCode 是按照一定的算法得到的一个数值，是对象的散列码值。主要用来在集合（后面会学到）中实现快速查找等操作，也可以用于对象的比较。

在 Java 中，对 hashCode 的规定如下：

（1）在同一个应用程序执行期间，对同一个对象调用 hashCode()，必须返回相同的整数

结果—前提是 equals()所比较的信息都不曾被改动过。至于同一个应用程序在不同执 行期所得的调用结果，无需一致。

（2）如果两个对象被 equals(Object)方法视为相等，那么对这两个对象调用 hashCode()必须获得相同的整数结果。

（3）如果两个对象被 equals(Object) 方法视为不相等，那么对这两个对象调用 hashCode()不必产生不同的整数结果。然而程序员应该意识到，对不同对象产生不同的整数结果，有可能提升 hashTable（后面会学到，集合框架中的一个类）的效率。

简单地说：如果两个对象相同，那么它们的 hashCode 值一定要相同；如果两个对象的 hashCode 相同，它们并不一定相同。

在 Java 规范里面规定，覆盖 equals 方法应该连带覆盖 hashCode 方法，这就涉及到一个如何实现 hashCode 方法的问题了。

实现一：偷懒的做法：对同一对象始终返回相同的 hashCode，如下：

```
public int hashCode() {
    return 1;
}
```

它是合法的，但是不好，因为每个对象具有相同的 hashCode，会使得很多使用 hashCode 的类的运行效率大大降低，甚至发生错误。

实现二：采用一定的算法来保证

在高效 Java 编程这本书里面，给大家介绍了一个算法，现在 eclipse 自动生成 equals 方法和 hashCode 方法就是用的这个算法，下面介绍一下这个算法：

（1）将一个非 0 常数，例如 31，储存于 int result 变量

（2）对对象中的每个有意义的属性 f（更确切的说是被 equals()所考虑的每一个属性）进行如下处理：

A. 对这个属性计算出类型为 int 的 hash 码 c：

如果属性是个 boolean，计算(f ? 0 : 1)。

如果属性是个 byte,char,short 或 int，计算(int)f。

如果属性是个 long，计算(int)(f ^ (f >>> 32))。

如果属性是个 float，计算 Float.floatToIntBits(f)。

如果属性是个 double，计算 Double.doubleToLongBits(f)，然后将计算结果按步骤 2.A.iii 处理。

如果属性是个对象引用，而且 class 的equals()通过递归调用 equals()的方式来比较这一属性，那么就同样也对该属性递归调用 hashCode()。如果需要更复 杂的比较，请对该属性运算一个范式（canonical representation），并对该范式调用 hashCode()。如果属性值是 null，就返回 0（或其它常数；返回 0 是传统做法）。

如果属性是个数组，请将每个元素视为独立属性。也就是说对每一个有意义的元素施行上述规则，用以计算出 hash 码，然后再依步骤 2.B 将这些数值组合起来。

B. 将步骤 A 计算出来的 hash 码 c 按下列公式组合到变量 result 中：

result = 31*result + c;

（3） 返回 result。

示例如下：这个就是用 eclipse 自动生成的

```
import java.util.Arrays;

public class Test {
    private byte byteValue;
    private char charValue;
    private short shortValue;
    private int intValue;
    private long longValue;
    private boolean booleanValue;
    private float floatValue;
    private double doubleValue;
    private String uuid;
    private int[] intArray = new int[3];

    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + (booleanValue ? 1231 : 1237);
        result = prime * result + charValue;
        long temp;
        temp = Double.doubleToLongBits(doubleValue);
        result = prime * result + (int) (temp ^ (temp >>> 32));
        result = prime * result + Float.floatToIntBits(floatValue);
        result = prime * result + Arrays.hashCode(intArray);
        result = prime * result + intValue;
        result = prime * result + (int) (longValue ^ (longValue >>> 32));
        result = prime * result + shortValue;
        result = prime * result + ((uuid == null) ? 0 : uuid.hashCode());
        return result;
    }
}
```

7.1.3 toString 方法

toString()方法是 Object 类中定义的另一个重要方法，是**对象的字符串表现形式**，其格式为：

```
public String toString(){……}
```

方法的返回值是 String 类型，用于描述当前对象的有关信息。Object 类中实现的 toString() 方法是返回当前对象的类型和内存地址信息，但在一些子类(如 String， Date 等）中进行了 重写，也可以根据需要在用户自定义类型中重写 toString()方法，以返回更适用的信息。

除显式调用对象的 toString()方法外，在进行 String 与其它类型数据的连接操作时，

会自动调用 toString()方法，其中又分为两种情况：

（1）引用类型数据直接调用其 toString()方法转换为 String 类型；

（2）基本类型数据先转换为对应的包装类型，再调用该包装类的 toString()方法转换为 String 类型。

另外，在 System.out.println()方法输出引用类型的数据时，也先自动调用了该对象的 toString()方法，然后再将返回的字符串输出。示例如下：

```
class MyDate {
    private int day, month, year;
    public MyDate(int d, int m, int y) {
        day = d;
        month = m;
        year = y;
    }
}
class YourDate {
    private int day, month, year;

    public YourDate(int d, int m, int y) {
        day = d;
        month = m;
        year = y;
    }
    public String toString() {
        return day + "-" + month + "-" + year;
    }
}
public class Test {
    public static void main(String args[]) {
        MyDate m = new MyDate(8, 8, 2008);
        System.out.println(m);
```

```
        System.out.println(m.toString());
        YourDate y = new YourDate(8, 8, 2008);
        System.out.println(y);
    }
}
```

运行结果：

```
cn.JavaDriver.java6.test.MyDate@1fb8ee3
cn.JavaDriver.java6.test.MyDate@1fb8ee3
8-8-2008
```

toString 方法被用来将一个对象转换成 String 表达式。当自动字符串转换发生时，它被用作编译程序的参照。System.out.println()调用下述代码：

```
Date now = new Date() ;
System.out.println(now);
//将被翻译成：
System.out.println(now.toString());
```

对象类定义缺省的 toString()方法，它返回类名称和它的引用的地址（通常情况下不是很有用）。许多类覆盖 toString()以提供更有用的信息。例如，所有的包装类覆盖 toString()以提供它们所代表的值的字符串格式。甚至没有字符串格式的类为了调试目的常常实现 toString()来返回对象状态信息。

7.2 String 类

7.2.1 String 的声明和创建

双引号括起来的字符序列就是 String 的直接量。实例:"John" 或"111222333"字符串赋值，可以在声明时赋值：

```
String color = "blue";
```

color 是 String 类型的引用。
"blue" 是 String 直接量。

String 类型的数据存在 String 常量池（栈内存）中，**一旦定义值就不能改变**，只能是让变量指向新的内存空间。比如：

```
color = "red";
```

如果采用 new 的方法定义 String，那么是需要分配堆内存空间的，如下：

```
String str = new String("Hello");
```

一共有两个对象，在栈和堆内存中各有一个对象，内容都是"Hello"。

String 的还有两个常用的构造方法：

String(byte[] arr) 使用一个字节数组 arr 创建一个字符串对象。

String(char[] arr) 使用一个字符数组 arr 创建一个字符串对象。

```
char[] arr={'H','e','l','l','o'};
String str = new String(arr);
//相当于
String str = new String("Hello");
```

String(char[] arr,int startIndex,int count) 提取字符数组 a 中的一部分字符创建一个字符串对象，参数 startIndex 和 count 分别指定在 a 中提取字符的起始位置和从该位置开始截取的字符个数，例如：

```
char[] arr={'H','e','l','l','o'};
String str = new String(arr,1,3);
//相当于
String str = new String("ell");
```

7.2.2 String 的常用方法

7.2.2.1 得到字符串的长度

public int length()

返回 String 的长度，是按照 char 返回的长度

与数组不同之处： String 类不含有 length 成员域（属性）

```
String str = "java快车www.javakc.com";
System.out.println(str.length());//20
```

7.2.2.2 比较字符串

对于字符串，使用“==”比较的是内存地址，一般不使用“==”比较字符串。

public boolean equals(Object s)

比较两个 String 对象的实体是否相等，这个是区分大小写的。实际上就是依次比较其所包含的字符的数值大小。

```
String s1= new String("java快车");
String s2= new String("java快车");
System.out.println(s1.equals(s2));//true
```

public int compareTo(String s)

比较两个字符串的大小。返回 0 表示相等，返回大于 0 的数表示前面的字符串大于后面的字符串，返回小于 0 表示前面的字符串小于后面的字符串，区分大小写的。如下：

```
public class Test {
```

```
    public static void main(String[] args) {
        String str = "这里是 JAVA 快车";
        String str2 = "这里是 java 快车";
        if (str.compareTo(str2) == 0) {
            System.out.println("the str 等于 str2");
        } else if (str.compareTo(str2) > 0) {
            System.out.println("the str 大于 str2");
        } else if (str.compareTo(str2) < 0) {
            System.out.println("the str 小于 str2");
        }
    }
}
```

运行结果：

```
the str 小于 str2
```

public int compareToIgnoreCase(String s)

忽略大小写，比较两个字符串的大小。返回 0 表示相等，返回大于 0 的数表示前面的字符串大于后面的字符串，返回小于 0 表示前面的字符串小于后面的字符串。如下：

```
public class Test {
    public static void main(String[] args) {
        String str = "这里是 JAVA 快车";
        String str2 = "这里是 java 快车";
        if (str.compareToIgnoreCase(str2) == 0) {
            System.out.println("the str 等于 str2");
        } else if (str.compareToIgnoreCase(str2) > 0) {
            System.out.println("the str 大于 str2");
        } else if (str.compareToIgnoreCase(str2) < 0) {
            System.out.println("the str 小于 str2");
        }
    }
}
```

运行结果：the str 等于 str2

public boolean equalsIgnoreCase(String s)

比较两个 String 对象的值是否相等，忽略大小写。

```
String s1= new String("java快车");
String s2= new String("JAVA快车");
System.out.println(s1.equalsIgnoreCase(s2));//true
```

public boolean startsWith(Stirng prefix)

测试此字符串是否以指定的前缀开始。

```
public class Test {
    public static void main(String[] args) {
        String str = "这里是 Java 快车";
```

```
        String str2 = "Java";
        System.out.println(str.startsWith(str2));
    }
}
```

运行结果：false

public boolean startsWith(String prefix,int toffset)

测试此字符串从指定索引开始的子字符串是否以指定前缀开始。

```
public class Test {
    public static void main(String[] args) {
        String str = "这里是 Java 快车";
        String str2 = "Java";
        System.out.println(str.startsWith(str2, 3));
    }
}
```

运行结果：true

public boolean endsWith(String suffix)

测试此字符串是否以指定的后缀结束。

```
public class Test {
    public static void main(String[] args) {
        String str = "这里是 Java 快车";
        String str2 = "快车";
        System.out.println(str.endsWith(str2));
    }
}
```

运行结果：true

7.2.2.3 字符串检索

public char charAt(int index)

获得字符串指定位置的字符。

public int indexOf(int ch)

返回指定字符 ch 在此字符串中第一次出现处的索引。

```
String name="CoolTools";
System.out.println(name.indexOf('T'));//4
```

public int indexOf(String str)

返回第一次找到字符串 str 时的索引，如果没有找到，则返回-1。实例：

```
String name="CoolTools";
System.out.println(name.indexOf("oo"));//1
System.out.println(name.indexOf("kc"));//-1
```

public int indexOf(int ch,int fromIndex)

从指定的索引开始搜索，返回在此字符串中第一次出现指定字符处的索引，如果没有找到，则返回-1。

```
String name="CoolTools";
System.out.println(name.indxOf('l',5));//7
```

public int indexOf(String str,int fromIndex)

指定的索引开始搜索，返回在此字符串中第一次出现指定字符串处的索引，如果没有找到，则返回-1。

```
String name="CoolTools";
System.out.println(name.indexOf("oo",3));//5
```

public int lastIndexOf(String str)

返回指定子字符串在此字符串中最右边出现处的索引。laseIndexOf 有四中重载方法，用法和 indexOf 相似。

7.2.2.4 截取字符串

public String substring(int beginIndex)

返回新的字符串，它是当前字符串的子串。该子串从指定的位置开始，并一直到当前字符串结束为止。

public String substring(int beginIndex,int endIndex)

返回新的字符串，它是当前字符串的子串。该子串从指定的位置(beginIndex)开始，到指定的位置(endIndex - 1)结束。例如：

```
"unhappy".substring(2);     //返回 "happy"
"Harbison".substring(3);    //返回 "bison"
"emptiness".substring(9);   //返回 ""(空串)
"emptiness".substring(10);  //返回 StringIndexOutOfBoundsException
"hamburger".substring(4,8); //返回 "urge"
"smiles".substring(1,5);    //返回 "mile"
```

7.2.2.5 替换

public String trim()

返回新字符串，截去了源字符串最前面和最后面的的空白符；如果字符串没有被改变，则返回源字符串的引用。

public String replace(char oldChar,char newChar)

public String replace(CharSequence target, CharSequence replacement)

返回一个新的字符串，它是将字符串中的所有 oldChar 替换成 newChar

-源字符串没有发生变化

-如果该字符串不含 oldChar，则返回源字符串的引用。

实例：

```
"mesquite in your cellar".replace('e','o');//返回"mosquito in your collar"
"JonL".replace('q','x'); //结果返回"JonL"  (没有发生变化)
```

public String replaceAll(String regex, String replacement)

使用给定的 replacement 替换此字符串中所有匹配给定的正则表达式的子字符串。

-源字符串没有发生变化

-如果该字符串不含 regex，则返回源字符串的引用。

```
String s= "collection,cool,connection";
System.out.print(s.replaceAll("tion", "all"));
//result:collecall,cool,connecall
```

public String toUpperCase( )

返回对应的新字符串，所有小写字母都变为大写的，其它的不变。

-如果没有字符被修改，则返回字符串的引用

public String toLowerCase ()

返回对应的新字符串，所有大写字母都变为小写的，其它的不变。

-如果没有字符被修改，则返回字符串的引用

7.2.2.6 字符串分解成数组

public byte[] getBytes()

使用平台的默认字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中。

public byte[] getBytes(Charset charset)

使用给定的 charset 将此 String 编码到 byte 序列，并将结果存储到新的 byte 数组。

对于字符串中的汉字，是按照 char 来计算的，一个中文汉字占两个字节，也就是说，通过 length()得到的是字符串 char 的长度，而不是字节数，利用这个特点，就可以进行中文判断了。

例如：如何判断一个字符串里面有没有中文呢？如果字符串对应的 byte[]和 char[]的长度是不一样的，那就说明包含中文，还可以顺带计算出有几个汉字。

```
public class Test {
    public static void main(String[] args) {
        String str = "欢迎来到Java快车";
        int charLen = str.length();
        int byteLen = str.getBytes().length;
        if (byteLen > charLen) {
            int chineseNum = byteLen - charLen;
            System.out.println("str包含汉字，汉字共" + chineseNum + "个");
        } else {
            System.out.println("str没有包含汉字");
        }
    }
}
```

运行结果是：str 包含汉字，汉字共 6 个

public void getChars(int srcBegin,intsrcEnd, char[] dst,int dstBegin)

拷贝字符串的部分字符序列到指定的字符数组的指定位置;

要复制的第一个字符位于索引 srcBegin 处;要复制的最后一个字符位于索引 srcEnd-1 处(因此要复制的字符总数是 srcEnd-srcBegin)。要复制到 dst 子数组的字符从索引 dstBegin 处开始,并结束于索引。

```
char[] arr={'1','2','3','4','5','6'};
String s= "collection";
s.getChars(4, 7, arr, 2);
for(char c:arr){
    System.out.print(c+",");
}
```

执行结果:1,2,e,c,t,6,

public char[] toCharArray()

将此字符串转换为一个新的字符数组。

public String[] split(String regex)

根据给定正则表达式的匹配拆分此字符串,得到拆分好的字符串数组,示例如下:

```
public class Test {
    public static void main(String[] args) {
        String str = "这里,是 Java,快车";
        String tempS[] = str.split(",");// 按照","对字符串进行拆分
        for (int i = 0; i < tempS.length; i++) {
            System.out.println("tempS[" + i + "]===" + tempS[i]);
        }
    }
}
```

运行结果: tempS[0]===这里 tempS[1]===是 Java tempS[2]===快车

注意:因为“.”和“|”都是转义字符,必须得加“\\”。

(1)如果用“.”作为分隔的话,必须是如下写法:String.split("\\."),这样才能正确的分隔开,不能用 String.split(".");

```
public class Test {
    public static void main(String[] args) {
        String str = "这里.是Java.快车";
        String tempS[] = str.split("\\.");// 按照"."对字符串进行拆分
        for (int i = 0; i < tempS.length; i++) {
            System.out.println("tempS[" + i + "]===" + tempS[i]);
        }
    }
}
```

(2)如果用“|”作为分隔的话,必须是如下写法:String.split("\\|"),这样才能正确的分隔开,不能用 String.split("|");

```
public class Test {
    public static void main(String[] args) {
        String str = "这里|是Java|快车";
        String tempS[] = str.split("\\|");// 按照"|"对字符串进行拆分
        for (int i = 0; i < tempS.length; i++) {
            System.out.println("tempS[" + i + "]===" + tempS[i]);
        }
    }
}
```

7.2.2.7 展示字符串

public String concat（String str）

拼接两个字符串，并返回一个新字符串。

-源字符串不会被修改。

-s1.concat(s2) 返回字符串 s1 和 s2 拼接的结果。

```
String s1 = "ABC";
String s2 = "XYZ";
s1 = s1.concat(s2); // 等同于 s1 = s1 + s2;
```

public static String valueOf（参数列表）

将参数的值转化成相应的字符串。

public static String valueOf(char[] data)

返回 `new String(data);`

public static String valueOf(char[] data, int offset,int count)

返回`new String(data,offset,count);`

public String toString( )

由于 源对象就是字符串了，所以返回字符串本身。

-其它引用类型也可以通过重写方法 toString，生成相应的字符串（详见 Object 的 toString 方法）。

7.3 正则表达式和相关的 String 方法

完整的正则表达式语法是比较复杂的，这里只是简单地入个门，更多的正则表达式请参考相应的书籍或者文章。不过没有什么大的必要，常见的正则表达式基本都是写好了的，拿过来用就可以了。

7.3.1 什么是正则表达式

在编写处理字符串的程序时，经常会有查找符合某些复杂规则的字符串的需要，正则表达式就是用于描述这些规则的工具。换句话说，**正则表达式就是记录文本规则的代码。**

回忆一下在 Windows 下进行文件查找，查找的通配符也就是*和?。如果你想查找某个目录下的所有的Word文档的话，你会搜索*.doc。在这里，*会被解释成任意的字符串。和通配符类似，正则表达式也是用来进行文本匹配的工具，只不过比起通配符，它能更精确地描述你的需求—当然，代价就是更复杂—比如你可以编写一个正则表达式，用来查找所有以0开头，后面跟着2-3 个数字，然后是一个连字号"-"，最后是 7 或 8 位数字的字符串(像 010-62972039 或 0311-8115213)。

简言之正则表达式是用于进行文本匹配的工具，也是一个匹配的表达式。

7.3.2 正则表达式基础入门

学习正则表达式的最好方法是从例子开始，理解例子之后再自己对例子进行修改，实验。下面给出了不少简单的例子，并对它们作了详细的说明。

假设你在一篇英文小说里查找 hi，你可以使用正则表达式 hi。 这是最简单的正则表达式了，它可以精确匹配这样的字符串：由两个字符组成，前一个字符是 h,后一个是 i。通常，处理正则表达式的工具会提供一个忽略大小写的选项，如果选中了这个选项，它可以匹配 hi,HI,Hi,hI 这四种情况中的任意一种。不幸的是，很多单词里包含 hi 这两个连续的字符，比如 him,history,high 等等。用 hi 来查找的话，这里边的 hi 也会被找出来。如果要精确地查找 hi 这个单词的话，我们应该使用\bhi\b。

\b 是正则表达式规定的一个特殊代码（某些人叫它元字符，meta character），代表着单词的开头或结尾，也就是单词的分界处。虽然通常英文的单词是由空格或标点符号或换行来分隔的，但是\b 并不匹配这些单词分隔符中的任何一个，它只匹配一个位置。假如你要找的是 hi 后面不远处跟着一个 Lucy，你应该用\bhi\b.*\bLucy\b。

"."是另一个元字符，匹配除了换行符以外的任意字符。*同样是元字符，不过它代表的不是字符，也不是位置，而是数量—它指定*前边的内容可以连续重复出现任意次以使整个表达式得到匹配。 因此，".*"连在一起就意味着任意数量的不包含换行符。现在 \bhi\b.*\bLucy\b 的意思就很明显了：先是一个单词 hi,然后是任意个任意字符(但不能是换行符)，最后是 Lucy 这个单词。

如果同时使用其它的一些元字符，我们就能构造出功能更强大的正则表达式。比如下面这个例子：

0\d\d-\d\d\d\d\d\d\d\d 匹配这样的字符串：以 0 开头，然后是两个数字，然后是一个连字号"-"，最后是 8 个数字(也就是中国的电话号码。当然，这个例子只能匹配区号为 3 位的情形)。

这里的\d 是一个新的元字符，匹配任意的数字(0，或 1，或 2，或……)。-不是元字符，只匹配它本身—连字号。

为了避免那么多烦人的重复，我们也可以这样写这个表达式：0\d{2}-\d{8}。 这里\d 后面的{2}和{8}的意思是前面\d 必须连续重复匹配 2 次和 8 次。

7.3.3 在 Java 中运行正则表达式

在 Java 中的 String 类中有好几个方法都跟正则表达式有关，最典型的就是

public boolean matches(String regex)

判断此字符是否匹配给定的正则表达式。

使用这个方法来测试和运行上面学到的正则表达式，示例如下：

```
public class Test {
    public static void main(String[] args) {
       String str = "010-62972039";
System.out.println("str是一个正确的电话号码？答案是： " +
                                  str.matches("0\\d{2}-\\d{8}"));
    }
}
```

运行结果：str 是一个正确的电话号码？答案是：true

注意：由于在 Java 里面“\”需要转义，应该变成“\\”。

在 Java 中，有专门进行正则表达式的类，在 java.util.regex 包里面。

7.3.3.1 java.util.regex.Pattern 类

Pattern 类是正则表达式的编译表示形式。

指定为字符串的正则表达式必须首先被编译为此类的实例。然后，可将得到的模式用于创建 Matcher 对象，依照正则表达式，该对象可以与任意字符序列匹配。执行匹配所涉及的所有状态都驻留在匹配器中，所以多个匹配器可以共享同一模式

常见方法如下：

static Pattern compile(String expression)

static Pattern compile(String expression, int flags)：编译正则表达式字符串到 pattern 对象用以匹配的快速处理

参数：

expression 正则表达式

flags 下列标志中的一个或多个：

CASE_INSENSITIVE,UNICODE_CASE,MULTILINE,UNIX_LINES,DOTALL,and CANON_EQ

Matcher matcher(CharSequence input)

返回一个 matcher 对象，它可以用来在一个输入中定位模式匹配

String[] split(CharSequence input)

String[] split(CharSequence input, int limit)

将输入字符串分离成记号，并由 pattern 来指定分隔符的形式。返回记号数组。分隔符并不是记号的一部分。

参数：

input 分离成记号的字符串

limit 生成的最大字符串数。

7.3.3.2 java.util.regex.Matcher 类

Mathcer 类通过解释 Pattern 对字符序列执行匹配操作的引擎。 常见方法如下：

boolean matches()：返回输入是否与模式匹配

boolean lookingAt()：如果输入的起始匹配模式则返回 True

boolean find()

boolean find(int start)：尝试查找下一个匹配，并在找到匹配时返回 True

参数：

start 开始搜索的索引

int start()：返回当前匹配的起始位置位置

int end()：返回当前匹配的结尾后位置

String group()：返回当前匹配

int groupCount()：返回输入模式中的分组数

int start(int groupIndex)

int end(int groupIndex) 返回一个给定分组当前匹配中的起始位置和结尾后位置

参数：

groupIndex 分组索引（从 1 开始），0 表示整个匹配

String group(int groupIndex)：返回匹配一个给定分组的字符串

参数：

groupIndex 分组索引（从 1 开始），0 表示整个匹配

String replaceAll(String replacement) String replaceFirst(String replacement) 返回从 matcher 输入得到的字符串，但已经用替换表达式替换所有或第一个匹配

参数:

replacement 替换字符串
Matcher reset()
Matcher reset(CharSequence input) 复位 mather 状态。

7.3.3.3 Java 中操作正则表达式示例

```
public class Test {
    public static void main(String[] args) {
       String str = "010-62972039";
       Pattern p = Pattern.compile("0\\d{2}-\\d{8}");
       Matcher m = p.matcher(str);
       boolean flag = m.matches();
       System.out.println("str是一个正确的电话号码？答案是：" + flag);
    }
}
```

运行结果：str 是一个正确的电话号码？答案是：true

7.3.4 元字符

现在你已经知道几个很有用的元字符了，如\b、.、*，还有\d.当然还有更多的元字符可用，比如\s 匹配任意的空白符，包括空格，制表符(Tab)，换行符，中文全角空格等。\w 匹配字母或数字或下划线等。

常用的元字符	
代码	说明
.	匹配除换行符以外的任意字符
\w	匹配字符或数字或下划线
\s	匹配任意的空白符
\d	匹配数字
\b	匹配字符串的开始或结束
^	匹配字符串的开始
$	匹配字符串的结束

下面来试试更多的例子:

例 1：\ba\w*\b

匹配以字母 a 开头的单词—先是某个单词开始处(\b)，然后是字母 a,然后是任意数量的字母或数字(\w*)，最后是单词结束处(\b)（好吧，现在我们说说正则表达式里的单词是什么意

思吧：就是几个连续的\w。不错，这与学习英文时要背的成千上万个同名的东西的确关系不大）。

例 2：\d+
匹配 1 个或更多连续的数字。这里的+是和*类似的元字符，不同的是*匹配重复任意次(可能是 0 次)，而+则匹配重复 1 次或更多次。

例 3：\b\w{6}\b
匹配刚好 6 个字母/数字的单词。

例 4：^\d{5,12}$
匹配必须为 5 位到 12 位数字的字符串

元字符^（和数字 6 在同一个键位上的符号）以及$和\b 有点类似，都匹配一个位置。^ 匹配你要用来查找的字符串的开头，$匹配结尾。这两个代码在验证输入的内容时非常有用， 比如一个网站如果要求你填写的 QQ 号时，可以使用。

这里的{5,12}和前面介绍过的{2}是类似的，只不过{2}匹配只能不多不少重复 2 次，{5,12}则是重复的次数不能少于 5 次，不能多于 12 次，否则都不匹配。

因为使用了^和$，所以输入的整个字符串都要用来和\d{5,12}来匹配，也就是说整个输入必须是 5 到 12 个数字，因此如果输入的 QQ 号能匹配这个正则表达式的话，那就符合要求了。

7.3.5 重复

已经看过了前面的*,+,{2},{5,12}这几个匹配重复的方式了。下面是正则表达式中所有的限定符(指定数量的代码，例如*,{5,12}等)：

常用的限定符	
代码/语法	说明
*	重复零次或多次
+	重复一次或多次
?	重复零次或一次
{n}	重复 n 次
{n,}	重复 n 次或更多次
{n,m}	重复 n 次到 m 次

下面是一些使用重复的例子：

```
Windows\d+ 匹配 Windows 后面跟 1 个或更多数字
13\d{9}                 匹配 13 后面跟 9 个数字(中国的手机号)
```

7.3.6 字符类

要想查找数字，字母或数字，空白是很简单的，因为已经有了对应这些字符集合的元字符，

但是如果你想匹配没有预定义元字符的字符集合(比如元音字母 a,e,i,o,u),应该怎么办?

很简单，你只需要在中括号里列出它们就行了，像[aeiou]就匹配任何一个英文元音字母，[.?!]匹配标点符号(.或?或!)(英文语句通常只以这三个标点结束)。

我们也可以轻松地指定一个字符范围，像[0-9]代表的含意与\d 就是完全一致的：一位数字，同理[a-z0-9A-Z_]也完全等同于\w（如果只考虑英文的话）。

7.3.7 常见正则表达式

（1）检测是否 Email 地址

```
^([\\w-\\.]+)@((\\[[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.)|(([\\w-]+\\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\\]?)$
```

（2）判断输入的字符串只包含汉字

```
^[\u4e00-\u9fa5]+$
```

（3）匹配 3 位或 4 位区号的电话号码，其中区号可以用小括号括起来，也可以不用，区号与本地号间可以用连字号或空格间隔，也可以没有间隔

```
^\\(0\\d{2}\\)[-]?\\d{8}$|^0\\d{2}[-]?\\d{8}$|^\\(0\\d{3}\\)[-]?\\d{7}$|^0\\d{3}[-]?\\d{7}$
```

（4）判断输入的字符串是否是一个合法的手机号，这个不完全，只是 13 开头的

```
^13\\d{9}$
```

（5）判断输入的字符串只包含数字，可以匹配整数和浮点数

```
^-?\\d+$|^(-?\\d+)(\\.\\d+)?$
```

（6）匹配非负整数

```
^\\d+$
```

（7）判断输入的字符串只包含英文字母

```
^[A-Za-z]+$
```

（8）判断输入的字符串是否只包含数字和英文字母

```
^[A-Za-z0-9]+$
```

好了，学到这里对基本的正则表达式就有了基本的认识，下面来看看 String 里面跟正则表达式有关的几个方法的使用。

7.4 StringBuffer 类和 StringBuilder 类

前面学到过 String 类有一个重要的特点，那就是 String 的值是不可变的，这就导致每次对 String 的操作都会生成新的 String 对象，不仅效率低下，而且大量浪费有限的内存空间。那么对于经常要改变值的字符串应该怎样操作呢？

答案就是使用 StringBuffer 和 StringBuilder 类，这两个类功能基本相似，区别主要在于 StringBuffer 类的方法是多线程安全的（多线程的课程在后面会学习到），而 StringBuilder 不是线程安全的，相比而言 StringBuilder 类会略微快一点。

7.4.1 StringBuffer 类

String
-字符串(String)对象一旦创建，其内容不能再被修改 (read-only)

StringBuffer
-StringBuffer 对象的内容是可以被修改的
-除了字符的长度之外，还有容量的概念
-通过动态改变容量的大小，加速字符管理

7.4.1.1 StringBuffer 的构造方法

StringBuffer buf1 = new StringBuffer();
创建空的 StringBuffer 对象，初始容量为 16 字符

StringBuffer buf2 = new StringBuffer(容量);
创建空的 StringBuffer 对象，指定容量大小。

StringBuffer buf3 = new StringBuffer(myString);
创建含有相应字符序列的 StringBuffer 对象，容量为 myString.length() +16

7.4.1.2 StringBuffer 的常用方法

public int length()
返回 StringBuffer 的长度

public int capacity()
返回 StringBuffer 的容量

public void setLength(int newLength)
增加或减小 StringBuffer 的长度

public char charAt(int index)

返回 StringBuffer 对象中指定位置的字符

public void setCharAt(int index, char ch)

设置 StringBuffer 对象中指定位置的字符

public void getChars(int srcBegin, int srcEnd,Char[] dst, int dstBegin)

将 StringBuffer 对象中指定的字符子序列，拷贝到指定的字符数组(dst)

public void reverse()

将 StringBuffer 对象中的字符序列按逆序方式排列，可用作字符串倒序

public StringBuffer append(……)

允许数值类型的值添加到 StringBuffer 对象中

public StringBuffer insert(……)

允许将各种数据插到 StringBuffer 对象的指定位置

public StringBuffer delete(int start, int end)
public StringBuffer deleteCharAt(int index)

允许删除 StringBuffer 对象中的指定字符

其中最常用的恐怕就要算 append 方法和 toString 方法了，如下示例：

```
public class Test {
    public static void main(String[] args) {
        StringBuffer buffer = new StringBuffer();
        buffer.append("这里");
        buffer.append("是");
        buffer.append("Java");
        buffer.append("快车");
        System.out.println("buffer==" + buffer.toString());
    }
}
```

运行结果：buffer==这里是 Java 快车

7.4.2 StringBuilder 类

StringBuilder 类是一个可变的字符序列。此类提供一个与 StringBuffer 兼容的 API，但不保证同步。该类被设计用作 StringBuffer 的一个简易替换，用在字符串缓冲区被单个线程使用的时候(这种情况很普遍)。如果可能，建议优先采用该类，因为在大多数实现中，它比 StringBuffer 要快。它的功能基本等同于 StringBuffer 类，就不再赘述了。

```
public class Test {
    public static void main(String[] args) {
        StringBuilder builder = new StringBuilder();
        builder.append("这里");
        builder.append("是");
```

```
        builder.append("Java");
        builder.append("快车");
        System.out.println("buffer==" + builder.toString());
    }
}
```

运行结果：builder==这里是 Java 快车

7.5 Math 类

Java 中的数学（Math）类是 final 类，不可继承。 其中包含一组静态方法和两个常数。

7.5.1 常数

PI ：double，圆周率
E ：double，自然对数

7.5.2 方法

7.5.2.1 截取

注意方法的返回类型

public static double ceil(double d)
返回不小于 d 的最小整数

public static double floor(double d)
返回不大于 d 的最大整数

public static int round(float f)
返回四舍五入后的整数

public static long round(double d)
返回四舍五入后的整数

7.5.2.2 变换

public static double abs(double d)
返回绝对值

public static double min(double d1, double d2)
返回两个值中较小的值

public static double max(double d1, double d2)
返回两个值中较大的值

7.5.2.3 对数

public static double log(double d)
自然对数

public static double exp(double d)
E 的指数

7.5.2.4 其它

public static double sqrt(double d)
返回平方根

public static double random()
返回随机数

还有三角函数的运算等，请参考 JDK 文档。

示例如下：

问题：请问有 101 条记录，按照每组 10 条记录进行分组，应该分多少组？

```
public class Test {
    public static void main(String[] args) {
      int records = 101;// 共有 101 条记录
      final int GROUP_NUM = 10;//每组 10 条

      int groups = (int) Math.ceil(1.0 * records / GROUP_NUM);
      //注意这里的 1.0,目的是要把类型变成double型的,而不是int,结果还是int,就错了
      System.out.println("应该分的组数为=" + groups);
    }
}
```

运行结果：应该分的组数为 11。

7.6 日期操作的类

7.6.1 Date 类

java.util 包里面的 Date 类，是 Java 里面进行日期操作常用类。Date 类用来表示特定的瞬间，精确到毫秒。

7.6.1.1 初始化 Date

Date 的构造方法：

public Date( )

分配 Date 对象并初始化此对象，以表示分配它的时候的当前时间（精确到毫秒）。使用 Date 类得到当前的时间。

public Date(long date)

分配 Date 对象并初始化此对象，以表示自从标准基准时间(称为“历元(epoch)“，即 1970 年 1 月 1 日 00:00:00 格林威治时间）以来的指定毫秒数。

7.6.1.2 常用方法

public boolean after(Date when)

测试此日期是否在指定日期之后

public boolean before(Date when)

测试此日期是否在指定日期之前方法：getTime()返回自 1970 年 1 月 1 日 00:00:00 GMT 以来此 Date 对象表示的毫秒数。

7.6.1.3 示例

简单的性能测试—监控一段代码运行所需要的时间

```
public class Test {
    public static void main(String args[]) {
       long d1 = new Date().getTime();// 得到此时的时间
       int sum = 0;
       for (int i = 1; i <= 1000000; i++) {// 一百万次
           sum += i;
       }
       System.out.println("从 1 加到 1000000 的和=" + sum);
       long d2 = new Date().getTime();// 得到此时的时间
       System.out.println("从 1 加到 1000000 所耗费的时间是=" + (d2 - d1) + "
毫秒");

    }
}
```

运行结果：

从 1 加到 1000000 的和=1784293664

从 1 加到 1000000 所耗费的时间是=20 毫秒。

7.6.2 DateFormat 类和 SimpleDateFormat 类

Date 对象表示时间的默认顺序是星期、月、日、小时、秒、年，例如

```
Wed May 12 14:33:11 CST 2009
```

我们可能希望按照某种习惯来输出时间，比如时间的顺序：年、月、星期、日。可以使用 java.text 包中的 DateFormat 类，是日期/时间格式化子类的抽象类，它以与语言无关的方式格式化并解析日期或时间。日期/时间格式化子类（如 SimpleDateFormat）允许进行格式化（也就是日期转换成文本）、解析（文本转换成日期）和标准化。

由于 DateFormat 是个抽象类，SimpleDateFormat 类是它的子类，所以下面就主要按照 SimpleDateFormat 类来讲解。

7.6.2.1 如何初始化

这里只讲述最常用到的构造方法，更多的请参看 JDK 文档。

构造方法：SimpleDateFormat(String pattern)

用给定的模式和默认语言环境的日期格式符号构造 SimpleDateFormat。

```
SimpleDateFormat sdf=new SimpleDateFormat("yyyy-MM-dd hh:mm:ss");
```

7.6.2.2 日期和时间模式

日期和时间格式由日期和时间模式字符串指定。在日期和时间模式字符串中，未加引号的字母 'A' 到 'Z' 和 'a' 到 'z' 被解释为模式字母，用来表示日期或时间字符串元素。文本可以使用单引号 (') 引起来，以免进行解释。"''" 表示单引号。所有其他字符均不解释；只是在格式化时将它们简单复制到输出字符串，或者在解析时与输入字符串进行匹配。

定义了以下模式字母（所有其他字符 'A' 到 'Z' 和 'a' 到 'z' 都被保留）：

字母	日期或时间元素	表示	示例
G	Era 标志符	Text	AD
y	年	Year	1996; 96
M	年中的月份	Month	July; Jul; 07
w	年中的周数	Number	27
W	月份中的周数	Number	2
D	年中的天数	Number	189
d	月份中的天数	Number	10
F	月份中的星期	Number	2
E	星期中的天数	Text	Tuesday; Tue
a	Am/pm 标记	Text	PM
H	一天中的小时数（0-23）	Number	0
k	一天中的小时数（1-24）	Number	24
K	am/pm 中的小时数（0-11）	Number	0
h	am/pm 中的小时数（1-12）	Number	12
m	小时中的分钟数	Number	30

s	分钟中的秒数	Number	55
S	毫秒数	Number	978
z	时区	General time zone	PST; GMT-08:00
Z	时区	RFC 822 time zone	-0800

7.6.2.3 常用方法

public Date parse(String source)

从给定字符串的开始解析文本，以生成一个日期。

public String format(Date date)

将一个 Date 格式化为日期/时间字符串。

这里只讲述最常用的方法，更多的方法请参看 JDK 文档。

7.6.2.4 示例

```
import java.util.*;
import java.text.*;
public class Test {
    public static void main(String args[]) {
       DateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss SSS");
       Date d = new Date();
       // 把当前时间转换成为我们熟悉的时间表达格式
       String str = df.format(d);
       System.out.println("当前时间是: " + str);
       // 然后再把字符串格式的日期转换成为一个Date类
       try {
          Date d2 = df.parse("2012-07-27 08:08:08 888");
          System.out.println("伦敦奥运会开幕时间是: " + d2.getTime());
       } catch (ParseException e) {
          e.printStackTrace();
       }
    }
}
```

运行结果：

```
当前时间是：2009-05-17 18:52:13 437
伦敦奥运会开幕时间是：1343347688888
```

也可以采用 System 类的静态方法 public long currentTimeMillis（）获取系统当前时间的毫秒数。

7.6.2.5 说明

虽然 JDK 文档上说 Date 的毫秒值，是相对于格林威治时间 1970 年 1 月 1 号的 0 点，但实际测试，这个 Date 是跟时区相关的，也就是说在中国测试这个基准值应该是 1970 年 1 月 1 日的 8 点，不过这个不影响我们的处理，因为只要是同一个基准时间就可以了，而不用关

心具体是多少，见下面的示例：

```
public class Test {
    public static void main(String args[]) {
       DateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss SSS");

       Date d = new Date(0L);// 把时间设为 0，表示到基准时间
       // 然后转换成为字符串看看是什么时候
       String str = df.format(d);

       System.out.println("基准时间是: " + str);
    }
}
```

运行结果：

基准时间是：1970-01-01 08:00:00 000

7.6.3 Calendar 类

java.util 包中的 Calendar 类是 Java 里面另外一个常用的日期处理的类。Calendar 类是一个抽象类，它为特定瞬间与一组诸如 YEAR、MONTH、DAY_OF_MONTH、HOUR 等日历字段之间的转换提供了一些方法，并为操作日历字段（例如获得下星期的日期）提供了一些方法。

7.6.3.1 初始化

Calendar 类是通过一个静态方法 getInstance()来获取 Calendar 实例。返回的 Calendar 基于当前时间，使用了默认时区和默认语言环境如下：

```
Calendar c = Calendar.getInstance();
```

7.6.3.2 使用 Calendar 对日期进行部分析取

Calendar 类一个重要的功能就是能够从日期里面按照要求析取出数据，如：年、月、日、星期等等。

public int get(int field)

返回给定日历字段的值，例如年份、月份、小时、星期等信息，参数 field 的有效值由 Calendar 得静态常量指定。

示例如下：

```
public class Test {
    public static void main(String args[]) {
       Calendar c = Calendar.getInstance();
       int year = c.get(Calendar.YEAR);
```

```
        int month = c.get(Calendar.MONTH);// 注意：month特殊，是从 0 开始的， 也
就是 0 表示 1 月
        int day = c.get(Calendar.DAY_OF_MONTH);
        System.out.println("现在是" + year + "年" + (month + 1) + "月" +
                                                        day + "日");
    }
}
```

运行结果：
现在是 2009 年 5 月 17 日

7.6.3.3 使用 Calendar 进行日期运算

这是 Calendar 另外一个常用的功能，也就是对日期进行加加减减的运算。

public void add(int field,int amount)

根据日历的规则，为给定的日历字段添加或减去指定的时间量。

示例如下：

```
public class Test {
    public static void main(String args[]) {
        Calendar c = Calendar.getInstance();
        c.add(Calendar.DATE, 12);//当前日期加 12 天，如果是-12 表示当前日期减去
12 天
        int year = c.get(Calendar.YEAR);
        //注意：month特殊, 是从 0 开始的, 也就是 0 表示 1 月
        int month = c.get(Calendar.MONTH);
        int day = c.get(Calendar.DAY_OF_MONTH);
        System.out.println("在当前日期加 12 天是" + year + "年" + (month + 1)
                                                + "月" + day+ "日");
    }
}
```

运行结果：在当前日期加 12 天是 2009 年 5 月 29 日

7.6.3.4 为 Calendar 设置初始值

public void setTime(Date date)

使用给定的 Date 设置此 Calendar 的当前时间。

public void setTimeInMillis(long millis)

用给定的 long 值设置此 Calendar 的当前时间值。

```
public class Test {
```

```
    public static void main(String args[]) {
        Calendar c = Calendar.getInstance();
        c.setTimeInMillis(1234567890123L);
        int year = c.get(Calendar.YEAR);
        //注意：month特殊,是从 0 开始的,也就是 0 表示 1 月
        int month = c.get(Calendar.MONTH);
        int day = c.get(Calendar.DAY_OF_MONTH);
        System.out.println("设置的时间是" + year + "年" + (month + 1) + "月"
                                                    + day+ "日");
    }
}
```

运行结果：设置的时间是 2009 年 2 月 14 日

7.7 System 类

7.7.1 命令行参数

当 Java 程序启动时，可以添加 0 或多个命令行参数（Command-line Arguments）。不管使用双引号与否都作为字符串自动保存到 main 函数的参数中。参数之间用空格分隔。

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

本段代码可以使用下面语句测试：java Test 这里 是 Java 快车

运行结果：

```
4
这里
是
Java
快车
```

7.7.2 控制台输入输出

许多应用程序要与用户进行文本 I/O（输入/输出）交互，标准输入是键盘；标准输出是终端窗口。Java SDK 支持控制台 I/O 使用三个 java.lang.System 类中定义的变量：

System.out 是一个 PrintStream 对象，初始引用启动 Java 的终端窗口。

System.in 是一个 InputStream 对象，初始指向用户键盘。

System.err 是一个 PrintStream 对象，初始引用启动 Java 的终端窗口。

这三个对象都可以重新定向(如文件)：System.setOut\setIn\setErr。

往标准输出写东西使用 PrintStream 对象的 println 或 print 方法。print 方法输出参数；但 println 方法输出参数并追加一个换行符。

println 或 print 方法都对原始类型进行重载，同时还重载了 char[]和 Object，String。参数是 Object 时，调用参数的 toString 方法。

输出例子：

```
public class Test {
    public static void main(String args[]) {
        char c[] = { 'a', 'b', 'c' };
        System.out.println(c);
    }
}
```

运行结果：abc

输入例子：

```
public class Test {
    public static void main(String args[]) {
        String s = "";
        InputStreamReader ir = new InputStreamReader(System.in);
        BufferedReader in = new BufferedReader(ir);
        System.out.println("Ctrl+z to exit");
        try {
            s = in.readLine();
            while (s != null) {
                System.out.println("Read:" + s);
                s = in.readLine();
            }
            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

运行的时候从控制台输入数据，然后回车，就看到输出了读入的数据。因为涉及到后面要学习的 I/O 的知识，所以这里先简单了解一下。

也可以使用 Scanner 类，Scanner 类用来扫描控制台的输入数据。控制台会一直等待输入，直到敲回车键结束，把所输入的内容传给 Scanner，作为扫描对象。如果要获取输入的内容，则只需要调用 Scanner 的 nextLine()方法即可，或者 next、nextInt、nextDouble 方法等。

```
import java.util.Scanner;
public class Test {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while(true){
            String s=sc.next();
            System.out.println(s);
        }
    }
}
```

7.7.3 格式化输出 printf

从 JDK5.0 开始，Java 里面提供了 C 风格的格式化输出方法 —— printf，比如输出一个加法算式，JDK5.0 版本以前的写法是：

```
public class Test {
    public static void main(String[] args) {
       int x = 5;
       int y = 7;
       int nSum = x + y;
       System.out.println(x + " + " + y + " = " + nSum);
    }
}
```

运行结果：5 + 7 = 12

而在 JDK5.0 以后版本中可以写为：

```
public class Test {
    public static void main(String[] args) {
       int x = 5;
       int y = 7;
       int nSum = x + y;
       System.out.printf("%d + %d = %d\n", x, y, nSum);
    }
}
```

以上两种写法的输出结构是一样的，即“5 + 7 = 12”。

这种改变不仅仅是形式上的，printf 还可以提供更为灵活、强大的输出功能，比如限定按照两位整数的形式输出，可以写为：

```
public class Test {
    public static void main(String[] args) {
       int x = 5;
       int y = 7;
       int nSum = x + y;
       System.out.printf("%02d + %02d = %02d\n", x, y, nSum);
    }
}
```

运行输出结果将是“05 + 07 = 12”。

其实这个功能在 Java 里面并没有什么大用，具体的 printf 格式化字符串格式请参见 JDK 文档中的具体说明。

7.8 学习目标

1. 学会使用 JDK API 文档
2. 掌握 Object 类中的 equals、hashCode、toString 方法的作用，以及相互的关系。
3. 掌握如何重写 Object 类中的 equals、hashCode、toString 方法。
4. equals 方法和”==”的功能和区别
5. 至少掌握 String 类的 4 中构造方法
6. 理解 String 类的存储结构。
7. 熟练掌握 String 类中的方法。能说出大部分方法的方法名称、完成的功能，甚至参数列表、返回值类型。
8. 了解正则表达式完成的功能。
9. 在 Java 中使用正则表达式完成验证功能。
10. 描述正则表达式中有哪些元字符。
11. 如何在正则表达式中表示“重复”。
12. 如何在正则表达式中表示“字符类”。
13. 为什么要用 StringBuffer？（String 有什么缺点）
14. 描述 StringBuffer 的容量和长度。
15. StringBuffer 为我们提供了哪些 String 没有的方法。
16. StringBuffer 和 StringBuilder 的区别。
17. 描述 Math 类中常用的方法。
18. 掌握课上画的关于日期的图，注意相互之间的转换。
19. 掌握如何通过 System.in 创建扫描仪 Scanner，如何使用 Scanner。

7.9 练习

1. 设计一个员工类 Employee，具有员工编号、姓名、性别、住址、出生日期属性，覆盖 Object 的 equals 和 toString 方法。在测试类中，分别创建两个员工对象，比较是否相等，并打印员工信息。

2. 将字符串“12-007,张三,男,上海浦东新区 15 号,1983-2-6”中的信息抽取出来，封装到第一题中创建的员工对象中。

3. 有 11 个人一起喝啤酒，每个人喝 3 瓶，每个酒箱内有 6 瓶啤酒，问要买几箱啤酒？（用 java 程序解答）

4. 编写一个程序，输入字符串，通过 substring 方法，将字符串逆序输出。

5. 下列代码编译并运行的结果是：

```
public class Test {
    public static void main(String[] args) {
        double num = 7.4;
        int a = (int) Math.abs(num + 0.5);
        int b = (int) Math.ceil(num + 0.5);
        int c = (int) Math.floor(num + 0.5);
        int d = (int) Math.round(num + 0.5);
        int e = (int) Math.round(num - 0.5);
        int f = (int) Math.floor(num - 0.5);
        int g = (int) Math.ceil(num - 0.5);
        int h = (int) Math.abs(num - 0.5);

        System.out.println("a=" + a);
        System.out.println("b=" + b);
        System.out.println("c=" + c);
        System.out.println("d=" + d);
        System.out.println("e=" + e);
        System.out.println("f=" + f);
        System.out.println("g=" + g);
        System.out.println("h=" + h);
    }
}
```

6. 在控制台输入字符串，用正则表达式验证是否只包含数字和英文字母。

7. （按不同格式打印日期）在上午信函中，日期可以几种不同的格式打印，两种比较常用的格式如下：09/21/2012 和 July 21,1013。
编写一个程序，读取第一种日期格式的日期，并以第二种日期格式返回。

8. 编写一个程序，用随机数生成语句。该程序应用 4 个 String 类型的数组，它们分别是 article，noun，verb，preposition。该程序按下列顺序从 4 个数组中随机选取一个元素生成一个语句：article，noun，verb，preposition，article，noun。当选取每个单词时，应该特别注意上述单词组成的数组是否足够大。要求：单词之间用空格分开，输出最后的语句时，应以大写字母开头，以圆点结尾。每次点击回车生成下

一条语句。

语句填充如下：article 数组包含冠词：the,a,one,some,any
noun 数组包含名词 boy,girl,doy,town,cat
verb 数组包含动词 drove,jumped,ran,walked,skipped
preposition 数组应包含介词 to,from,over,under,on

9. 计算当前时间距明年春节还有多少天多少小时多少分钟多少秒。

10. 将 26 个英文字母用逗号分隔，组成字符串打印出来。考虑效率问题，使用 StringBuffer 或者 StringBuilder。

11. 设计一个银行帐户类，具有户名，帐号，密码，余额等属性，在控制台模拟登录、退出、存款、取款等方法，并对此类进行测试

12. 编写程序，测试计算 1～50 的阶乘的和所耗费的毫秒级时间。
阶乘：例如 5 的阶乘等于 5*4*3*2*1

13. 找出 1~1000 之间的全部“同构数”。
注：如果一个数出现在其平方数的右端，则称此数为同构数。如：1 在 1*1=1 的右端，5 在 5*5=25 的右端，25 在 25*25=625 的右端等等。

8 抽象类和接口

8.1 抽象类

8.1.1 什么是抽象类

有时在开发中，要创建一个体现某些基本行为的类，并为该类声明方法，但不能在该类中实现该行为，而是在子类中实现该方法。

这种只给出方法定义而不具体实现的方法被称为抽象方法，抽象方法是没有方法体的，在代码的表达上就是没有“{}”。

怎么表示一个方法是抽象的呢？使用 abstract 修饰符来表达抽象。

abstract 修饰符可以与类和方法一起使用。被修饰的类不能被实例化，被修饰的方法必须在包含此方法的类的子类中被实现。 抽象类简单地说：使用 abstract 修饰的类就是抽象类。 示例如下：

```
public abstract class Test {// 抽象类定义
    public abstract void doItByHand();// 抽象方法定义
}
```

8.1.2 抽象类的使用

例如，考虑一个 Drawing 类。该类包含用于各种绘图设备的方法，但这些必须以独立平台的方法实现。它不可能去访问机器的录像硬件而且还必须是独立于平台的。其意图是绘图类定义哪种方法应该存在，但实际上，由特殊的从属于平台子类去实现这个行为。

正如 Drawing 类这样的类，它声明方法的存在而不是实现，以及带有对已知行为的方法的实现，这样的类通常被称做抽象类。通过用关键字 abstract 进行标记声明一个抽象类。被声明但没有实现的方法（即，这些没有程序体或{}），也必须标记为抽象。

```
public abstract class Drawing {
    public abstract void drawDot(int x, int y);
    public void drawLine(int x1, int y1, int x2, int y2) {
       // draw using the drawDot() method repeatedly.
    }
}
```

抽象类不能直接使用，必须用子类去实现抽象类，然后使用其子类的实例。然而可以创建一个变量，其类型是一个抽象类，并让它指向具体子类的一个实例，也就是可以使用抽象类来充当形参，实际实现类作为实参，也就是多态的应用。

不能有抽象构造方法或抽象静态方法。

abstract 类的子类为它们父类中的所有抽象方法提供实现，否则它们也是抽象类。

```
public class MachineDrawing extends Drawing {
    public void drawDot(int machX, int machY) {
       // 画点
    }
}
Drawing d = new MachineDrawing();
```

在下列情况下，一个类将成为抽象类：

（1）当一个类的一个或多个方法是抽象方法时；

（2）当类是一个抽象类的子类，并且不能为任何抽象方法提供任何实现细节或方法主体时；

（3）当一个类实现一个接口，并且不能为任何抽象方法提供实现细节或方法主体时；

注意：

（1）这里说的是这些情况下一个类将成为抽象类，没有说抽象类一定会有这些情况。

（2）一个典型的错误：抽象类一定包含抽象方法。 但是反过来说“包含抽象方法的类一定是抽象类”就是正确的。

（3）事实上，抽象类可以是一个完全正常实现的类

8.2 接口的基本概念

接口可以说是 Java 程序设计中最重要的概念之一了，“面向接口编程”是面向对象世界的共识，所以深刻理解并熟练应用接口是每一个学习 Java 编程人员的重要任务。

8.2.1 接口概念

Java 可以创建一种称作接口(interface)的类，在这个类中，所有的成员方法都是抽象的，也就是说它们都只有定义而没有具体实现，接口是抽象方法和常量值的定义的集合。从本质上讲，接口是一种特殊的抽象类，用 interface，可以指定一个类必须做什么，而不是规定它如何去做。定义接口的语法格式如下：

```
访问修饰符 interface  接口名称  {
  抽象属性集
  抽象方法集
}
```

现实中也有很多接口的实例，比如说串口电脑硬盘，Serial ATA 委员会指定了 Serial ATA 2.0 规范，这种规范就是接口。Serial ATA 委员会不负责生产硬盘，只是指定通用的规范。希捷、日立、三星等生产厂家会按照规范生产符合接口的硬盘，这些硬盘就可以实

现通用化，如果正在用一块 160G 日立的串口硬盘，现在要升级了，可以购买一块 320G 的希捷串口硬盘，安装上去就可以继续使用了。

在 Java 中可以模拟 Serial ATA 委员会定义以下串口硬盘接口。

```
//串行硬盘接口
public interface SataHdd{
    //连接线的数量
    public static final int CONNECT_LINE=4;
    //写数据
    public void writeData(String data);
    //读数据
    public String readData();
}
```

目前看来接口和抽象类差不多。确实如此，接口本就是从抽象类中演化而来的，因而除特别规定，接口享有和类同样的“待遇”。比如，源程序中可以定义多个类或接口，但最多只能有一个 public 的类或接口，如果有则源文件必须取和 public 的类和接口相同的名字。和类的继承格式一样，接口之间也可以继承，子接口可以继承父接口中的常量和抽象方法并添加新的抽象方法等。

但接口有其自身的一些特性，归纳如下：

接口中声明的成员变量默认都是 public static final 的，必须显示的初始化。因而在常量声明时可以省略这些修饰符。

接口中只能定义抽象方法，这些方法默认为 public abstract 的，因而在声明方法时可以省略这些修饰符。试图在接口中定义实例变量、非抽象的实例方法及静态方法，都是非法的。

```
public interface SataHdd{
    //连接线的数量
    public int connectLine;//编译出错,connectLine是静态常量，必须显式初始化
    //写数据
    protected void writeData(String data); //编译出错，必须是public类型
    //读数据
    public static String readData(){      //编译出错，接口中不能包含静态方法
        return "数据";                    //编译出错，接口中只能包含抽象方法，
    }
}
```

接口中没有构造方法，不能被实例化。

一个接口不实现另一个接口，但可以继承多个其他接口。接口的多继承特点弥补了类的单继承。

```
//串行硬盘接口
public interface SataHdd extends A,B{
    // 连接线的数量
```

```
    public static final int CONNECT_LINE = 4;
    // 写数据
    public void writeData(String data);
    // 读数据
    public String readData();
}
interface A{
    public void a();
}
interface B{
    public void b();
}
```

8.2.2 为什么使用接口

两个类中的两个类似的功能，调用它们的类动态地决定一种实现，那它们提供一个抽象父类，子类分别实现父类所定义的方法。

问题的出现：Java 是一种单继承的语言，一般情况下，哪个具体类可能已经有了一个父类，解决是给它的父类加父类，或者给它父类的父类加父类，只到移动到类等级结构的最顶端。这样一来，对一个具体类的可插入性的设计，就变成了对整个等级结构中所有类的修改。接口是可插入性的保证。

在一个等级结构中的任何一个类都可以实现一个接口，这个接口会影响到此类的所有子类，但不会影响到此类的任何父类。此类将不得不实现这个接口所规定的方法，而其子类可以从此类自动继承这些方法，当然也可以选择置换掉所有的这些方法，或者其中的某一些方法，这时候，这些子类具有了可插入性（并且可以用这个接口类型装载，传递实现了他的所有子类）。

我们关心的不是哪一个具体的类，而是这个类是否实现了我们需要的接口。

接口提供了关联以及方法调用上的可插入性，软件系统的规模越大，生命周期越长，接口使得软件系统的灵活性和可扩展性，可插入性方面得到保证。

接口把方法的特征和方法的实现分割开来。这种分割体现在接口常常代表一个角色，它包装与该角色相关的操作和属性，而实现这个接口的类便是扮演这个角色的演员。一个角色由不同的演员来演，而不同的演员之间除了扮演一个共同的角色之外，并不要求其它的共同之处。

对于下述情况，接口是有用的：

（1）声明方法，期望一个或更多的类来实现该方法。

（2）揭示一个对象的编程接口，而不揭示类的实际程序体。（当将类的一个包输送到其它开发程序中时它是非常有用的。）

（3）捕获无关类之间的相似性，而不强迫类关系。

（4）可以作为参数被传递到在其它对象上调用的方法中

面向对象程序设计讲究"提高内聚，降低耦合"，那么不同的程序模块怎么相互访问呢， 就是通过接口，也就是接口是各部分对外的统一外观。接口在 Java 程序设计中体现的思想就是隔离，因为接口只是描述一个统一的行为，所以开发人员在面向接口编程时并不关心具 体的实现。

由以上讲到的接口的作用和基本思想可以看到，接口在面向对象的 Java 程序设计中占有举足轻重的地位。事实上在设计阶段最重要的任务之一就是设计出各部分的接口，然后通过接口的组合，形成程序的基本框架结构。

8.3 接口作为类型使用

8.3.1 接口的使用

接口的使用与类的使用有些不同。在需要使用类的地方，会直接使用 new 关键字来构建一个类的实例进行应用：

```
ClassA a =new ClassA();
```

这是正确的 但接口不可以这样用，因为接口不能直接使用 new 关键字来构建实例。

```
SataHdd sh = new SataHdd();
```

这是错误的，接口在使用的时候要实例化相应的实现类。实现类的格式如下：

```
访问修饰符 修饰符 class 类名 extends 父类 implements 多个接口 {
        实现方法
}
```

说明：
接口必须通过类来实现它的抽象方法，类实现接口的关键字为 implements。

如果一个类不能实现该接口的所有抽象方法，那么这个类必须被定义为抽象方法。

不允许创建接口的实例，但允许定义接口类型的引用变量，该变量指向了实现接口的类的实例。

一个类只能继承一个父类，但却可以实现多个接口。

```
//希捷硬盘
public class SeagateHdd implements SataHdd,A{
    public String readData() {
        //希捷硬盘读取数据
```

```
            return "数据";
        }
        public void writeData(String data) {
            //希捷硬盘写入数据
        }
    }
    //三星硬盘
    public class SamsungHdd implements SataHdd{
        public String readData() {
            //三星硬盘读取数据
            return "数据";
        }
        public void writeData(String data) {
            //三星硬盘写入数据
        }
    }
    //某劣质硬盘，不能写数据
    public abstract class XXHdd implements SataHdd{
        public String readData() {
            //硬盘读取数据
            return "数据";
        }
    }
    public class Client{
        public static void main(String[] args) {
            SataHdd sh1=new SeagateHdd();  //初始化希捷硬盘
            SataHdd sh2=new SamsungHdd();  //初始化三星硬盘
        }
    }
```

8.3.2 接口作为类型使用

接口作为引用类型来使用,任何实现该接口的类的实例都可以存储在该接口类型的变量中，通过这些变量可以访问类中所实现的接口中的方法，Java 运行时系统会动态地确定应该使用哪个类中的方法，实际上是调用相应的实现类的方法。

示例如下：

```
public class Test {
    public void test1(A a) {
        a.doSth();
    }
    public static void main(String[] args) {
        Test t = new Test();
        A a = new B();
        t.test1(a);
    }
}
```

```
public interface A {
    public int doSth();
}
public class B implements A {
    public int doSth() {
        System.out.println("now in B");
        return 123;
    }
}
```

运行结果：now in B

大家看到接口可以作为一个类型来使用，把接口作为方法的参数和返回类型。

8.4 接口和抽象类的选择

由于从某种角度讲，接口是一种特殊的抽象类，它们的渊源颇深，有很大的相似之处，所以在选择使用谁的问题上很容易迷糊。我们首先分析它们具有的相同点。

都代表类树形结构的抽象层。在使用引用变量时，尽量使用类结构的抽象层，使方法的定义和实现分离，这样做对于代码有松散耦合的好处。

都不能被实例化。

都能包含抽象方法。抽象方法用来描述系统提供哪些功能，而不必关心具体的实现。

抽象类和接口的主要区别：

抽象类可以为部分方法提供实现，避免了在子类中重复实现这些方法，提高了代码的可重用性，这是抽象类的优势；而接口中只能包含抽象方法，不能包含任何实现。

```
public abstract class A{
    public abstract void method1();
    public void method2(){
        //A method2
    }
}
public class B extends A{
    public void method1(){
        //B method1
    }
}
public class C extends A{
    public void method1(){
        //C method1
    }
}
```

抽象类 A 有两个子类 B、C，由于 A 中有方法 method2 的实现，子类 B、C 中不需要重写 method2 方法，我们就说 A 为子类提供了公共的功能，或 A 约束了子类的行为。method2 就是代码可重用的例子。A 并没有定义 method1 的实现，也就是说 B、C 可以根据自己的特点实现 method1 方法，这又体现了松散耦合的特性。再换成接口看看：

```
public interface  A{
    public void method1();
    public void method2();
}
public class B implements A{
    public void method1(){
        //B method1
    }
    public void method2(){
        //B method2
    }
}
public class C implements A{
    public void method1(){
        //C method1
    }
    public void method2(){
        //C method2
    }
}
```

接口 A 无法为实现类 B、C 提供公共的功能，也就是说 A 无法约束 B、C 的行为。B、C 可以自由地发挥自己的特点现实 method1 和 method2 方法，接口 A 毫无掌控能力。

一个类只能继承一个直接的父类（可能是抽象类），但一个类可以实现多个接口，这个就是接口的优势。

```
interface A{
    public void method2();
}
interface B{
    public void method1();
}
class C implements A,B{
    public void method1(){
        //C method1
    }
    public void method2(){
        //C method2
    }
}
//可以如此灵活的使用C，并且C还有机会进行扩展，实现其他接口
A a=new C();
```

```
B b=new C();
```

```
abstract class A{
    public abstract void method1();
}
abstract class B extends A{
    public abstract void method2();
}
class C extends B{
    public void method1(){
        //C method1
    }
    public void method2() {
        //C method2
    }
}
```

对于 C 类，将没有机会继承其他父类了。

综上所述，接口和抽象类各有优缺点，在接口和抽象类的选择上，必须遵守这样一个原则：行为模型应该总是通过接口而不是抽象类定义。所以通常是：

（1）优先选用接口，尽量少用抽象类。

选择抽象类的时候通常是如下情况：

（2）需要定义子类的行为，又要为子类提供共性的功能。

8.5 学习目标

1. 什么是抽象方法？什么是抽象类？
2. 在 Java 中如何表示抽象类和抽象方法。
3. 抽象方法和抽象类的关系？
4. 抽象类中是否有属性，抽象类中是否有构造方法。
5. 如何使用抽象类。
6. 什么是接口？接口的定义规则？
7. 接口的作用？为什么使用接口？
8. 举例说明如何把接口当作类型使用
9. 如何选择接口和抽象类？为什么？

8.6 练习

1. 定义一个接口“交通工具”，说明交通工具可以移动。实现交通工具而产生汽车、飞机、轮船，并定义类来实现其移动的方法。

2. 定义一个类来使用上面的接口

9 异常

9.1 异常的定义

9.1.1 异常基础知识

在 Java 编程语言中，异常是指当程序出错时创建的一种特殊的运行时错误对象。注意这个错误不是编译时的语法错误。

Java 创建异常对象后，就发送给 Java 程序，即抛出异常(throwing an exception)。程序捕捉到这个异常后，可以编写相应的异常处理代码进行处理，而不是让程序中断。使用异常处理可以提高程序的健壮性，有助于调试和后期维护。

在程序执行中，任何中断正常程序流程的异常条件就是错误或异常。例如，发生下列情况时，会出现异常：

- 想打开的文件不存在
- 网络连接中断
- 受控操作数超出预定范围
- 正在装载的类文件丢失

在 Java 编程语言中，错误类定义被认为是不能恢复的严重错误条件。在大多数情况下，当遇到这样的错误时，建议让程序中断。

Java 编程语言实现异常处理来帮助建立弹性代码。在程序中发生错误时，发现错误的方法能抛出一个异常到其调用程序，发出已经发生问题的信号。然后，调用方法捕获抛出的异常，在可能时，再恢复回来。这个方案给程序员一个写处理程序的选择，来处理异常。

通过浏览 API，可以了解方法抛出的是什么样的异常。

9.1.2 异常实例

在学习在程序中处理异常之前，看一看如果不处理异常，会有什么情况发生。下面的小程序包括一个故意导致被零除错误的表达式。

```
class Exc0 {
    public static void main(String args[]) {
        int d = 0;
        int a = 42 / d;
```

```
        }
    }
```

当 Java 运行时系统检查到被零除的情况，它构造一个新的异常对象然后引发该异常。这导致 Exc0 的执行停止，因为一旦一个异常被引发，它必须被一个异常处理程序捕获并且被立即处理。该例中，没有提供任何异常处理程序，所以异常被 Java 运行时系统的默认处理程序捕获。任何不是被你程序捕获的异常最终都会被该默认处理程序处理。默认处理程序显示一个描述异常的字符串，打印异常发生处的堆栈轨迹并且终止程序。

下面是由标准 javaJDK 运行时解释器执行该程序所产生的输出：

```
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)
```

类名 Exc0，方法名 main，文件名 Exc0.java 和行数 4 被包括在一个堆栈使用轨迹中，用于提示异常所发生的位置,显示导致错误产生的方法调用序列。引发的异常类型是 Exception 的一个名为 ArithmeticException 的子类，该子类更明确的描述了何种类型的错误方法。Java 提供多个内置的与可能产生的不同种类运行时错误相匹配的异常类型。

下面是前面程序的另一个版本，它介绍了相同的错误，但是错误是在 main()方法之外的另一个方法中产生的：

```
class Exc1 {
    static void subroutine() {
        int d = 0;
        int a = 10 / d;
    }
    public static void main(String args[]) {
        Exc1.subroutine();
    }
}
```

默认异常处理器的堆栈轨迹结果表明了整个调用栈是怎样显示的：

```
java.lang.ArithmeticException: / by zero
at Exc1.subroutine(Exc1.java:4)
at Exc1.main(Exc1.java:7)
```

栈底是 main 的第 7 行，该行调用了 subroutine()方法。该方法在第 4 行导致了异常。调用堆栈对于调试来说是很重要的，因为它查明了导致错误的精确的步骤。

9.2 异常的处理

Java 提供了一种异常处理模型，它使您能检查异常并进行相应的处理。它实现的是异常处理的抓抛模型。使用此模型，您只需要注意有必要加以处理的异常情况。Java 提供的这种异常处理模型，代替了用返回值或参数机制从方法返回异常码的手段。

异常处理的抓抛方法有两大优点：

（1）异常情况能仅在有必要之处加以处理，防止程序自动终止，而不在其发生处和需要进行处理处之间的每一级上均进行处理。

（2）能够编写统一的、可重用的异常处理代码。

应该区别对待程序中的正常控制流和异常处理流。当然，异常处理流也是程序中的控制流。当异常发生时，抛出一个异常。异常伴随调用链，直到它们被捕获或程序退出为止。 下面是 Java 语言中的异常处理块的模型：

```
try {
    // 放置可能出现异常的代码
} catch (Exception1 e1) {
    // 如果 try 块抛出异常对象的类型为 Exception1，那么就在这里进行处理
} catch (Exception2 e2) {
    // 如果 try 块抛出异常对象的类型为 Exception2，那么就在这里进行处理
} catch (ExceptionN eN) {
    // 如果 try 块抛出异常对象的类型为 ExceptionN，那么就在这里进行处理
} finally {
    // 不管是否有异常发生，始终执行这个代码块
}
```

在未提供适当异常处理机制的程序中，无论何时发生异常，程序均会异常中断，而之前分配的所有资源则保持其状态不变。这会导致资源遗漏。要避免这一情况，在适当的异常处理机制中，我们可以将以前由系统分配的所有资源返还给系统。所以，当异常可能发生时，要牢记必须对每一异常分别进行处理。

例如我们处理文件 I/O,在打开文件时发生 IOException，程序异常中断而没有机会关闭该文件，这可能会毁坏文件而且分配给该文件的操作系统资源可能未返还给系统。

9.2.1 try-catch

try 块由一组可执行语句组成，在执行它们时可能会抛出异常。

catch 块，是用来捕获并处理 try 中抛出的异常的代码块。 catch 块不能单独存在，可以有多个 catch 块,以捕获不同类型的异常。

try 不可以跟随在 catch 块之后。

```
class Exc0 {
    public static void main(String args[]) {
        try {
            int d = 0;
            int a = 1 / d;
            System.out.println("This will not be printed");
        } catch (ArithmeticException e) {
            System.out.println("Division by zero");
        }
        System.out.println("After catch statement");
    }
}
```

程序将会发生异常而中断，异常可在 catch 块中被捕获，输出如下：

```
Division by zero
After catch statement
```

在 try 块中对 println()的调用是不会执行的。一旦异常被引发，程序控制由 try 块转到 catch 块。执行永远不会从 catch 块"返回"到 try 块。因此，第 6 行将不会被执行。一旦执行了 catch 语句，程序控制从整个 try/catch 机制的下面一行（第 10 行）继续。

一个 try 和它的 catch 语句形成了一个单元。catch 子句的范围限制于 try 语句前面所定义的语句。一个 catch 语句不能捕获另一个 try 声明所引发的异常（除非是嵌套的 try 语句情况）。被 try 保护的语句声明必须在一个大括号之内（也就是说，它们必须在一个块中）。不能单独使用 try。

构造 catch 子句的目的是解决异常情况并且像错误没有发生一样继续运行。例如，下面的程序中，每一个 for 循环的反复得到两个随机整数。这两个整数分别被对方除，结果用来除 1000。最后的结果存在 a 中。如果一个除法操作导致被零除错误，它将被捕获，a 的值设为零，程序继续运行。

```
import java.util.Random;
class HandleError {
    public static void main(String args[]) {
        int a = 0, b = 0, c = 0;
        Random r = new Random();
        for (int i = 0; i < 10; i++) {
            try {
                b = r.nextInt();
                c = r.nextInt();
                a = 1000 / (b / c);
            } catch (ArithmeticException e) {
                System.out.println("Division by zero.");
                a = 0; // set a to zero and continue
            }
            System.out.println("a: " + a);
        }
    }
}
```

try 块可以嵌套，也就是说，一个 try 语句可以在另一个 try 块内部。每次进入 try 语句，异常的前后关系都会被推入堆栈。如果一个内部的 try 语句不含特殊异常的 catch 处理程序，堆栈将弹出，下一个 try 语句的 catch 处理程序将检查是否与之匹配。这个过程将继续直到一个 catch 语句匹配成功，或者是直到所有的嵌套 try 语句被检查耗尽。如果没有 catch 语句匹配，Java 的运行时系统将处理这个异常。下面是运用嵌套 try 语句的一个例子：

```
class Exc0 {
    public static void main(String args[]) {
        try {
            int a = 1 / 0;
            try {
```

```
                int b = 2 / 0;
            } catch (Exception e2) {
                // 异常处理
                System.out.println("e2");
        }
    } catch (Exception e) {
        // 异常处理
        System.out.println("e1");
    }
 }
}
```

当执行到第 4 行代码时，发生了异常，被第 11 行的 catch 捕捉到，再执行第 14 行的异常处理代码，而不执行第 5 行至第 10 行的代码。如果第 4 行没有异常产生，将会捕捉到第 6 行代码的异常。

当多个 catch 块存在的时候，从上往下 catch 异常的范围应该从小到大，因为 catch 块的运行机制是找到一个匹配的就进行处理了，如果把范围大的放在前面，那么后面的代码就没有机会运行了，这会是一个编译异常。

比如下面这个是正确的：

```
public class Exc0 {
    public static void main(String[] args) {
        try {
            int a = 5 / 0;
        } catch (ArithmeticException e) {
            e.printStackTrace();
        } catch (Exception err) {
            err.printStackTrace();
        }
    }
}
```

而下面这个就是错误的了，编译都发生了错误：

```
public class Exc0 {
    public static void main(String[] args) {
        try {
            int a = 1 / 0;
        } catch (Exception err) {
            err.printStackTrace();
        } catch (ArithmeticException e) {
            e.printStackTrace();
        }
    }
}
```

9.2.2 finally 块

finally 块表示：无论是否出现异常，都会运行的块

通常在 finally 块中可以编写资源返还给系统的语句，通常，这些语句包括但不限于：

（1）释放动态分配的内存块；

（2）关闭文件；

（3）关闭数据库结果集；

（4）关闭与数据库建立的连接；

它紧跟着最后一个块，是可选的，不论是否抛出异常，finally 块总会被执行。finally 块的语法如下：

```
try{
}catch(异常类型 1  e){
}catch(异常类型 2  e){
}finally{
}
```

下面的程序显示的是 finally 块的使用

```
public class Test {
    static String name;
    static int n01, n02;
    public static void main(String args[]) {
       try {
          name = "Aptech Limited";
          n01 = Integer.parseInt(args[0]);
          n02 = Integer.parseInt(args[1]);
          System.out.println(name);
          System.out.println("Division is" + n01 / n02);
       } catch (ArithmeticException i) {
          System.out.println("Can not be divided by zero!");
       } finally {
          name = null;
          System.out.println("finally executed");
       }
    }
}
```

从下面的命令行执行此程序：

```
Java  Test 20  0
```

将会得到下面的输出：

```
Aptech Limited
Can not be divided by zero!
finally executed
```

现在从下面的命令行执行此程序：

```
Java  Test  20  4
```

则会得到下面这样的输出：

```
Aptech Limited
Division is 5
finally executed
```

说明：当用不同的命令行参数执行此程序时，均会看见“finally executed”的输出。

这意味着，无论 try 块是否抛出异常，都会执行 finally 块。

思考题：是否会执行第 7 行代码？

```
try{
       System.out.println(1);
       return ;
   }catch(Exception e){
       e.printStackTrace();
   }finally{
       System.out.println(11);
}
```

9.2.3 try、catch、finally 块的关系

（1）try 块不能单独存在，后面必须跟 catch 块或者 finally 块。

（2）三者之间的组合为：try—catch 、try—catch—finally 、 try—finally 这几种是合法的。

（3）一个 try 块可以有多个 catch 块，从上到下多个 catch 块的范围为从小到大。

9.2.4 throw 语句

throw 语句用来从代码中主动抛出异常，可以将它理解为一个向上抛出的动作。throw 的操作数是任一种异常类对象。是 Throwable 类类型或 Throwable 子类类型的一个对象。简单类型，例如 int 或 char，以及非 Throwable 类，例如 String 或 Object，不能用作异常。

程序执行在 throw 语句之后立即停止；后面的任何语句不被执行。最紧紧包围的 try 块用来检查它是否含有一个与异常类型匹配的 catch 语句。如果发现了匹配的块，控制转向该语句；如果没有发现，次包围的 try 块来检查，以此类推。如果没有发现匹配的 catch 块，默认异常处理程序中断程序的执行并且打印堆栈轨迹。

下面是 throw 关键字的一个示例：

```
try {
    int i = 5/0;
} catch (ArithmeticException i) {
    throw new Exception("Can not be divided by zero!");
    System.out.println("after throw");
}
```

第 2 行代码产生异常，然后执行 catch 块中的代码，第 4 行抛出新异常，将不再执行第 5 行代码。

下面是一个创建并引发异常的例子程序，与异常匹配的处理程序再把它引发给外层的处理程序。

```
class ThrowDemo {
    static void demoproc() {
        try {
            throw new NullPointerException("demo");
        } catch (NullPointerException e) {
            System.out.println("Caught inside demoproc.");
            throw e; // rethrow the exception
        }
    }
    public static void main(String args[]) {
        try {
            demoproc();
        } catch (NullPointerException e) {
            System.out.println("Recaught: " + e);
        }
    }
}
```

该程序有两个机会处理相同的错误。首先，main（）设立了一个异常关系然后调用 demoproc()。demoproc()方法然后设立了另一个异常处理关系并且立即引发一个新的 NullPointerException 实例，NullPointerException 在下一行被捕获。异常于是被再次引发。下面是输出结果：

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

该程序还阐述了怎样创建 Java 的标准异常对象，特别注意下面这一行：

```
throw new NullPointerException("demo");
```

这里，new 用来构造一个 NullPointerException 实例。所有的 Java 内置的运行时异常有两个构造函数：一个没有参数，一个带有一个字符串参数。当用到第二种形式时，参数指定描述异常的字符串。如果对象用作 print()或println()的参数时，该字符串被显示。这同样可以通过调用 getMessage()来实现，getMessage()是由 Throwable 定义的。

9.2.5 throws 语句

throws 用来在方法定义时声明异常。

Java 中对异常的处理有两种方法，一个就是 try-catch，然后自己处理；一个就是不做处理，向外 throws，由调用该方法的代码去处理。

Java 语言要求在方法定义中列出该方法抛出的异常：

```
public class Example{
    public static void exceptionExample() throws
                                        ExampleException,LookupException{
    }
}
```

在上面的示例中，exceptionExample()声明包括 throws 关键字，其后列出了此方法可能抛出的异常列表。在此案例中列出的是 ExampleException 和 LookupException。

比如前面那个例子写完整如下：

```
    public class Exc0 {
        public static void main(String args[]) throws Exception {
            try {
                int a = 1 / 0;
            } catch (ArithmeticException i) {
                throw new Exception("Can not be divided by zero!");
            }
        }
    }
```

9.2.6 调用栈机制

如果方法中的一个语句抛出一个没有在相应的 try/catch 块中处理的异常，那么这个异常就被抛出到调用方法中。如果异常也没有在调用方法中被处理，它就被抛出到该方法的调用程序。这个过程要一直延续到异常被处理。如果异常到这时还没被处理，它便回到 main()， 而且，即使 main()不处理它，那么，该异常就异常地中断程序。

考虑这样一种情况，在该情况中 main( )方法调用另一个方法（比如，first( )），然后它调用另一个（比如，second( )）。如果在 second( )中发生异常，那么必须做一个检查来看看该异常是否有一个 catch；如果没有，那么对调用栈（first( )）中的下一个方法进行检查，然后检查下一个（main()）。如果这个异常在该调用栈上没有被最后一个方法处理，那么就会发生一个运行时错误，程序终止执行

9.3 异常的分类

9.3.1 异常的分类

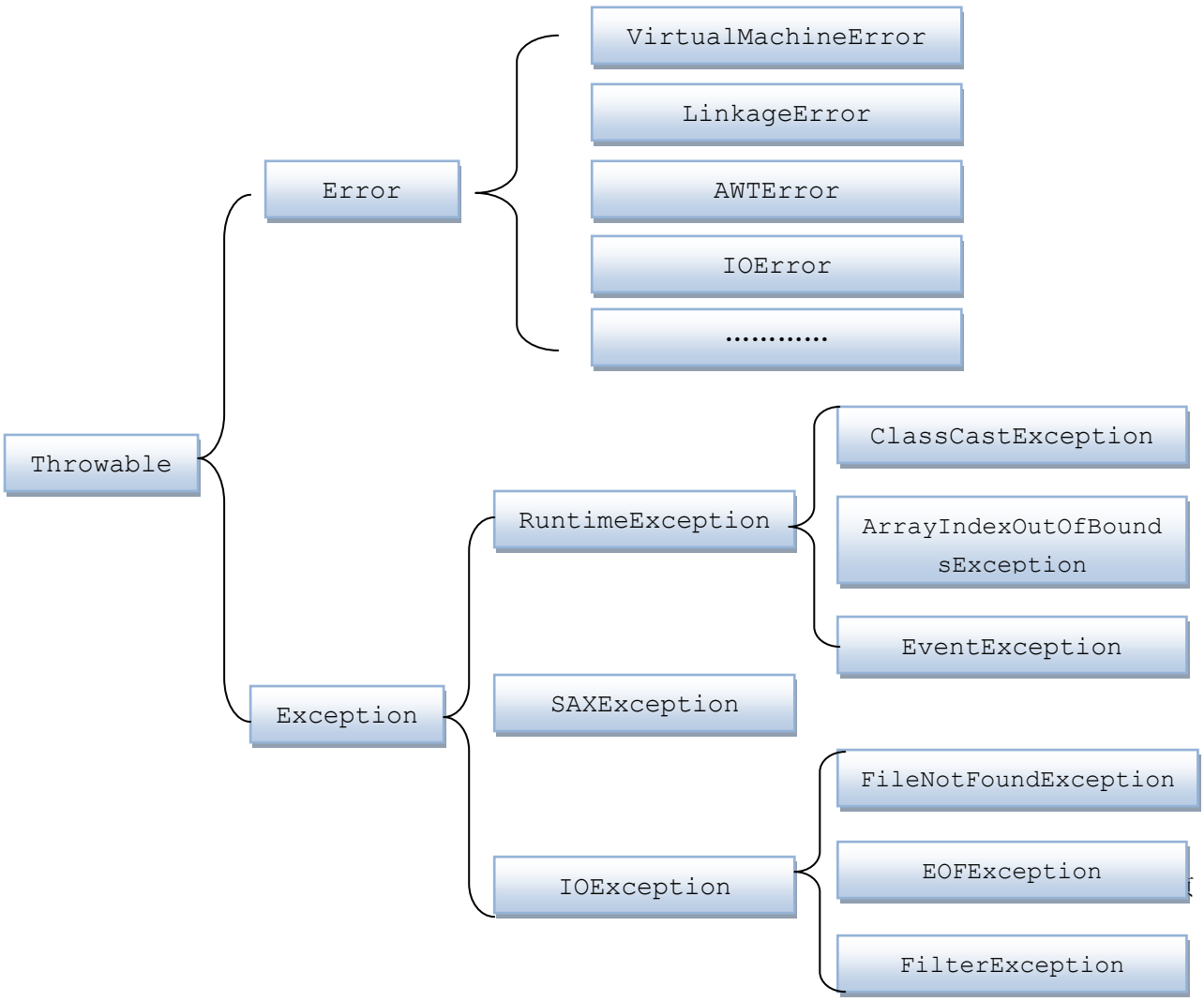
在 Java 编程语言中，异常有两种分类。java.lang.Throwable 类充当所有对象的父类，可以使用异常处理机制将这些对象抛出并捕获。在 Throwable 类中定义方法来检索与异常相关的错误信息，并打印显示异常发生的栈跟踪信息。它有 Error 和 Exception 两个基本子类。

错误（Error）：JVM 系统内部错误、资源耗尽等严重情况；

异常（Exception 违例）：其它因编程错误或偶然的外在因素导致的一般性问题，例如：对负数开平方根、空指针访问、试图读取不存在的文件、网络连接中断等。

当发生 Error 时，程序员根本无能为力，只能让程序终止。比如说内存溢出，不可能指望程序能处理这样的情况。而对于 Exception，而有补救或控制的可能，程序员也可以预先防范，本章主要讨论 Exception 的处理。

为有效地描述异常状况、传递有关的异常信息，JDK 中针对各种普遍性的异常情况定义了多种异常类型。其层次关系如下图所示：



异常分为运行时异常和受检查异常。

RuntimeException（运行时异常）是指因设计或实现方式不当导致的问题。也可以说，是程序员的原因导致的，本来可以避免发生的情况。比如，如果事先检查数组元素下标保证其不超出数组长度，那么，ArrayIndexOutOfBoundsException 异常从不会抛出；再如，先检查并确保一个引用类型变量值不为 null，然后在令其访问所需的属性和方法，那么，NullPointerException 也就从不会产生。

这种异常的特点是 Java 编译器不会检查它，也就是说程序，程序中可能出现这类异常时，即使没有用 try-catch 语句捕获它，也没有用 throws 语句声明抛出它，还是会编译通过的。由于没有处理它，当出现这类异常时，异常对象一直被传递到 main()方法，则程序将异常终止。如果采用了异常处理，异常将会被相应的程序执行处理。

除了 RuntimeException 以及子类，其他的 Exception 及其子类都是受检查异常。这种异常的特点是 Java 编译器不会检查它，也就是说，当程序中出现这类异常时，要么用 try-catch 语句捕获它，也没有用 throws 语句声明抛出它，否则编译不会通过。这些异常的产生不是因为程序员的过错，是程序员无法预见的。

总结一下，Java 程序异常处理的原则为：

（1）对于 Error 和 RuntimeException，可以在程序中进行捕获和处理，但不是必须的；

（2）对于其它异常，必须在程序中进行捕获和处理。

9.3.2 预定义异常

Java 编程语言中预先定义好的异常叫预定义异常，上面提到的异常都是预定义异常，这些异常的产生不需要程序员手动抛出，即不需要使用 throw 语句抛出异常，当产生异常时，系统会自动抛出。下面是几种常见的异常：

JDK 中定义的 RuntimeException 子类	
名称	描述
ArithmeticException	算术错误，如被 0 除
NullPointerException	非法使用空引用
ArrayIndexOutOfBoundsException	数组下标越界
NegativeArraySizeException	创建带负维数大小的数组的尝试
ClassCastException	非法强制转换类型
ArrayStoreException	数组元素赋值类型不兼容
IllegalArgumentException	调用方法的参数非法
IllegalMonitorStateException	非法监控操作，如等待一个未锁定线程
IllegalStateException	环境或应用状态不正确
IllegalThreadStateException	请求操作与当前线程状态不兼容
IndexOutOfBoundsException	某些类型索引越界
NumberFormatException	字符串到数字格式非法转换

SecurityException	试图违反安全性
StringIndexOutOfBounds	试图在字符串边界之外索引
UnsupportedOperationException	遇到不支持的操作

JDK 中的定义的受检查异常	
名称	描述
ClassNotFoundException	找不到类
CloneNotSupportedException	试图克隆一个不能实现 Cloneable 接口的对象
IllegalAccessException	对一个类的访问被拒绝
InstantiationException	试图创建一个抽象类或者抽象接口的对象
InterruptedException	一个线程被另一个线程中断
NoSuchFieldException	请求的字段不存在
NoSuchMethodException	请求的方法不存在

9.3.3 自定义异常

Java 语言允许用户在需要时创建自己的异常类型，用于表述 JDK 中未涉及到的其它异常状况，这些类型也必须继承 Throwable 类或其子类。用户自定义异常类通常属 Exception 范畴，依据命名惯例，应以 Exception 结尾。用户自定义异常未被加入 JRE 的控制逻辑中，因此永远不会自动抛出，只能由人工创建并抛出。

Throwable 定义的方法	
方法	描述
String getLocalizedMessage( )	返回一个异常的局部描述
String getMessage( )	返回一个异常的描述
void printStackTrace( )	显示堆栈轨迹
void printStackTrace(PrintStreamstream)	把堆栈轨迹送到指定的流
void printStackTrace(PrintWriterstream)	把堆栈轨迹送到指定的流
String toString( )	返回一个包含异常描述的 String 对象。当输出一个 Throwable 对象时，该方法被 println( )调用

看一个用户自定义异常的例子：

```
class MyException extends Exception {
    private int idnumber;

    public MyException(String message, int id) {
       super(message);
       this.idnumber = id;
    }

    public int getId() {
       return idnumber;
```

```
    }
}

public class Test {
    public void regist(int num) throws MyException {
        if (num < 0) {
            throw new MyException("人数为负值，不合理", 3);
        }
        System.out.println("登记人数" + num);
    }

    public void manager() {
        try {
            regist(-100);
        } catch (MyException e) {
            System.out.print("登记出错，类别: " + e.getId());
        }
        System.out.print("本次登记操作结束");
    }

    public static void main(String args[]) {
        Test t = new Test();
        t.manager();
    }
}
```

9.4 学习目标

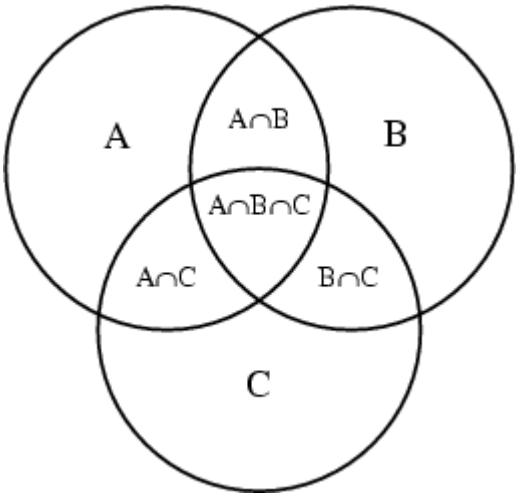
1. 什么是异常?
2. 简述处理异常的两种方式?
3. 简述 try 块的功能和规则
4. 简述 catch 块的功能和规则
5. 简述 finally 块的功能和规则
6. 简述 throw 和 throws 的功能和使用方法
7. 理解什么时候用抓，什么时候用抛。
8. 异常的分类?
9. 什么是受检查异常，描述其特点。
10. 什么是运行时异常，描述其特点。
11. 什么是预定义异常
12. 简述自定义异常的规则

10 集合框架

10.1 基本概念

10.1.1 数学背景

在常见用法中，集合（Collection）和数学上直观的集（set）的概念是相同的。集是一个唯一项组，也就是说组中没有重复项。实际上，“集合框架”包含了一个 Set 接口和许多具体的 Set 类。 但正式的集概念却比 Java 技术提前了一个世纪，那时英国数学家 George Boole 按逻辑正式的定义了集的概念。大部分人在小学时通过我们熟悉的维恩图引入的“集的交”和“集的并”学到过一些集的理论。



映射是一种特别的集。它是一种对（pair）集，每个对表示一个元素到另一元素的单向映射。一些映射示例有：

字典（词到含义的映射）

关键字到数据库记录的映射

IP 地址到域名（DNS）的映射

2 进制到 10 进制转换的映射

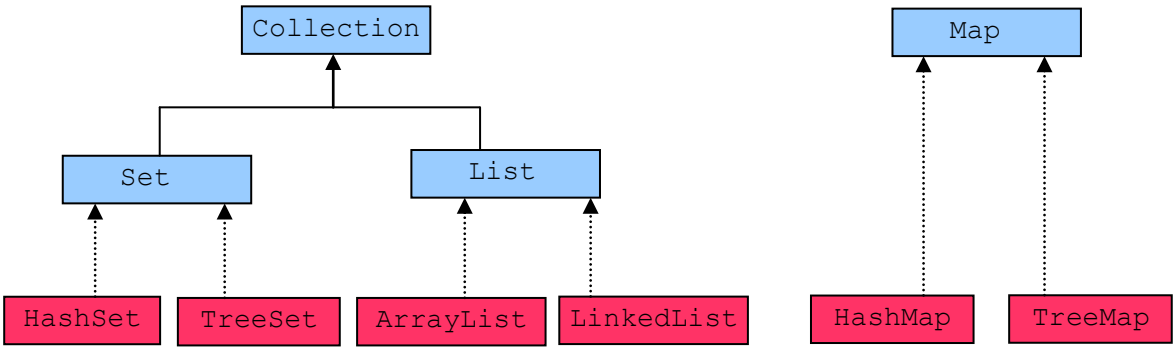
10.1.2 什么是集合

集合是包含多个对象的简单对象，所包含的对象称为元素。集合的典型应用是用来处理多种

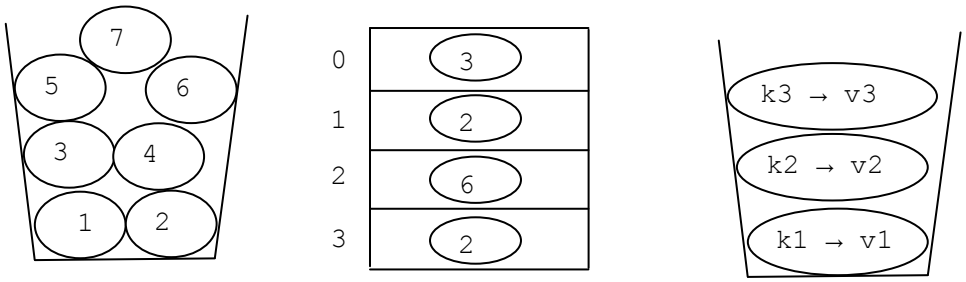
类型的对象，这些类型必须有共同的父类。

“集合框架”由一组用来操作对象的接口组成。不同接口描述不同类型的组。在很大程度上，一旦你理解了接口，就理解了框架。

虽然总要创建接口特定的实现，但访问实际集合的方法应该限制在接口方法的使用上；因此，允许更改基本的数据结构而不必改变其它代码。框架接口层次结构如下图所示。



从上图中，我们可以看出，集合主要分为 3 中类型：



Set(集) List(列表) Map(映射)

Set（集）中的对象不允许重复,无序。

List(列表)中的对象允许重复，有序。允许按照对象在集合中的索引位置检索对象。List 与数组有些相似。

Map（映射）集合中的每一个元素包含一对键对象和值对象，集合中没有重复的键对象，值对象可以重复。

10.2 Collection 接口和 Iterator 接口

在 Collection 接口中声明了适用于 Java 集合（只包含 Set 和 List）的通用方法：

方法	描述
boolean add(Object o)	向集合中添加一个对象的引用
void clear()	删除集合中所有的对象，即不再持有这些对象的引用
boolean contains(Object o)	判断在集合中是否持有特定对象的引用
boolean isEmpty()	判断集合是否为空
Iterator iterator()	返回一个 Iterator 对象，可用它来遍历集合中的元素
Boolean remove(Object o)	从集合中删除一个对象的引用
int size()	返回集合中元素的数目
Object[] toArray()	返回一个数组，该数组包含集合中所有的元素

Set 接口和 List 接口都继承了 Collection 接口，而 Map 接口没有集成 Collection 接口，因此可以对 Set 对象和 List 对象调用以上方法，但是不能对 Map 对象调用以上方法。

Collection 接口的 iterator()和 toArray()方法都用于获得集合中的所有对象，iterator()方法返回一个 Iterator 对象，toArray()方法返回一个包含集合中所有元素的数组。

Iterator 接口隐藏底层集合的数据结构，向客户程序提供了遍历各种类型的集合的统一接口。Iterator 接口中声明了如下方法：

方法	描述
boolean hasNext()	判断集合中的元素是否遍历完毕，如果没有，就返回 true
Object next()	返回下一个元素
void remove()	从集合中删除上一个由 next()方法返回的元素。

Collection 接口还支持元素组的操作：

方法	描述
boolean containsAll (Collection collection)	判断集合中是否包含了另一个集合的所有元素，即另一个集合是否是当前集合的子集，如果是，就返回 true
boolean addAll (Collection collection)	将另一个集合中的所有元素都添加到当前的集合中，通常称为并
void removeAll (Collection collection)	类似于 clear()，移除此集合中那些包含在指定 collection 中的所有元素
boolean retainAll (Collection collection)	仅保留此集合中那些也包含在指定 collection 的元素，即交集

10.3 Set(集)

Set 是最简单的一种集合，集合中的对象不按特定的方式排序，它不允许集合中存在重复项。

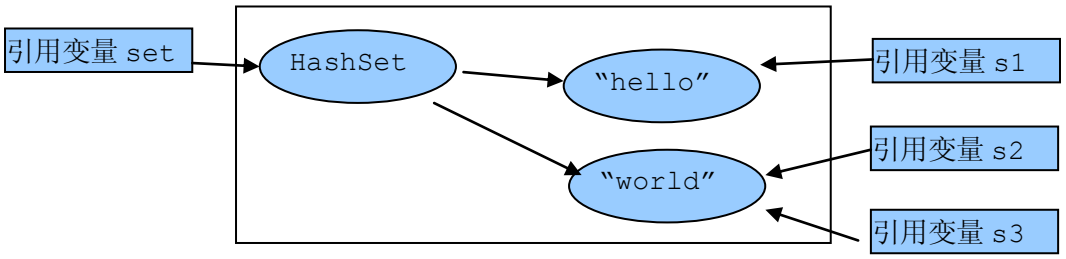
Set 接口主要有两个实现类：HashSet 和 TreeSet。HashSet 按照哈希算法来存取集合中的对象，存取速度比较快。Hash 类还有一个子类 LinkedHashSet 类，它不仅实现了 Hash 算法，而且实现了链表数据结构，链表数据结构能提高插入和删除元素的性能。TreeSet 类实现了 SortedSet 接口，具有排序功能。

10.3.1 Set 的一般用法

Set 集合中存放的是对象的引用，并且没有重复对象。以下代码创建了 3 个引用变量：s1、s2、s3。 s2 和 s3 变量引用同一个字符串对象“world”， s1 变量引用另一个字符串对象“hello”，Set 集合依次把这 3 个引用变量加入到集合中。

```
Set<String> set=new HashSet<String>();
String s1=new String("hello");
String s2=new String("world");
String s3=s2;
set.add(s1);
set.add(s2);
set.add(s3);
System.out.println(set.size());//打印集合中对象的数目 2
```

以上程序的打印结果为 2，实际上只向 Set 集合加入了两个对象，



当一个新对象加入到 Set 集合中时，Set 采用对象的 equals()方法比较两个对象是否相等，而不是采用“==”比较运算符。所以实际上只向集合中加入了一个对象。

10.3.2 HashSet 类

HashSet 类按照哈希算法来存取集合中的对象，具有很好的存取和查找性能。当向集合中加入一个对象时，HashSet 会调用对象的 hashCode()方法来获得哈希码，然后根据这个哈希码进一步计算出对象在集合中的存放位置。

在 object 类中定义了 hashCode()和 equals()方法，Object 类的 equals()方法按照内存地址比较对象是否相等，因此如果 object1.equals(object2)为 true，则表明 object1 和 object2 变量实际上引用同一个对象，那么两个对象的哈希码也肯定相等。

如果用户定义的 Customer 类覆盖了 Object 类的 equals()方法，但是没有覆盖 Object 类的 hashCode()方法，就会导致 customer1.equals(customer2)为 true 时，而

customer1 和 customer2 的哈希码不一定一样，这会使 HashSet 无法正常工作。

```
class Customer{
    private String name;
    private int age;
    //省略set、get方法
    public Customer(String name,int age){
        this.name=name;
        this.age=age;
    }
     public boolean equals(Object obj) {
         if (this == obj) {
            return true;
         }
         if (obj == null) {
            return false;
         }
         if (obj instanceof Customer) {
             Customer customer = (Customer) obj;
            if (this.name.equals(customer.getName()) &&
                                        this.age==customer.getAge()) {
               return true;
            } else {
               return false;
            }
         } else {
            return false;
         }
     }
}
```

以下程序向 HashSet 中加入两个 Customer 对象。

```
Set<Customer> set=new HashSet<Customer>();
Customer customer1=new Customer("Tom",25);
Customer customer2=new Customer("Tom",25);
set.add(customer1);
set.add(customer2);
System.out.println(set.size());
```

由于 customer1.equals(customer2)的比较结果为 true，按理说 HashSet 只应该把 customer1 加入集合中，但实际上以上程序的打印结果为 2，表明集合中加入了两个对象。出现这一非正常现象在于，customer1 和 customer2 的哈希码不一样，因此 HashSet 为 customer1 和 customer2 计算出不同的存放位置，于是把它们存放在集合中的不同地方。

为了保证 HashSet 正常工作，如果 Customer 类覆盖了 equals()方法，也应该覆盖 hashCode()方法，并且保证两个相等的 Customer 对象的哈希码也一样。Customer 类的 hashCode()方法代码如下：

```
public int hashCode(){
    int result;
```

```
        result=(name==null?0:name.hashCode());
        result=29*result+age;
        return result;
    }
```

判断 name 是否为 null 是为了保证程序代码的健壮性。

10.3.3 TreeSet 类

TreeSet 类实现了 SortedSet 接口，能够对集合中的对象进行排序。以下程序创建了一个 TreeSet 对象，然后向集合中加入 4 个 Integer 对象。

```
    Set<Integer> set = new TreeSet<Integer>();
    set.add(new Integer(3));
    set.add(new Integer(1));
    set.add(new Integer(4));
    set.add(new Integer(2));
    Iterator<Integer> it=set.iterator();
    while(it.hasNext()){
        System.out.print(it.next()+" ");
    }
```

以上程序的打印结果为：

```
1 2 3 4
```

当 treeSet 向集合中加入一个对象时，会把它插入到有序的对象序列中。那么 TressSet 是如何对对象进行排序的呢?TreeSet 支持两种排序方式：自然排序和客户化排序。在默认情况下 TreeSet 采用自然排序方式。

10.3.3.1 自然排序

在 JDK 类库中，有一部分类实现 Comparable 接口，如 Integer、Double 和 String 等。Comparable 接口有一个 compareTo(Object o)方法，它返回整数类型。

对于表达式 x.compareTo(y)，如果返回值为 0，则表示 x 和 y 相等，如果返回值大于 0，则表示 x 大于 y，如果返回值小于 0，则表示 x 小于 y。

TreeSet 调用对象的 compareTo()方法比较集合中对象的大小，然后进行升序排列，这种排序方式称为自然排序。下表中显示了 JDK 类库中实现了 Comparable 接口的一些类的排序方式。

类	排序
BigDecimal 、 BigInteger 、 Byte 、 Float、Integer、Long、Short	按数字大小排序
Character	按字符的 Unicode 值的数字大小排序
String	按字符串中字符的 Unicode 值排序

使用自然排序时，只能向 TreeSet 集合中加入同类型的对象，并且这些对象的类必须实现 Comparable 接口。以下程序先后向 TreeSet 集合中加入了一个 Integer 对象和 String 对象。

```
    Set<Object> set = new TreeSet<Object>();
    set.add(new Integer(3));
    set.add(new String("1"));//运行时抛出ClassCastException
```

当第二次调用 TreeSet 的 add()方法时抛出 ClassCastException。

```
Exception in thread "main" java.lang.ClassCastException:
java.lang.Integer cannot be cast to java.lang.String
    at java.lang.String.compareTo(String.java:92)
    at java.util.TreeMap.put(TreeMap.java:545)
    at java.util.TreeSet.add(TreeSet.java:238)
    at Client.main(Client.java:17)
```

在 string 类的 compareTo(Object o)方法中，首先对参数 o 进行类型转换。

```
    String s=(String)o;
```

如果参数 o 实际引用的不是 String 类型的对象，以上代码就会抛出 ClassCastException。

下面的示例中向 TreeSet 集合加入了 3 个 Customer 对象，但是 Customer 类没有实现 Comparable 接口。

```
    Set<Customer> set = new TreeSet<Customer>();
    set.add(new Customer("Tom",25));
    set.add(new Customer("Tom",24));
    set.add(new Customer("Jim",25));
```

当第二次调用 TreeSet 的 add()方法时，也会抛出 ClassCastException 异常。如果希望避免这种异常，应该使 Customer 类实现 Comparable 接口。

```
class Customer implements Comparable{
    private String name;
    private int age;

    // 省略set、get方法
    public Customer(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public int compareTo(Object o){
        Customer other=(Customer)o;
        //先按照name属性排序
        if(this.name.compareTo(other.getName())>0){
            return 1;
        }else if(this.name.compareTo(other.getName())<0){
            return -1;
        }
        //再按照age属性排序
```

```
        if(this.age>other.getAge()){
            return 1;
        }else if(this.age<other.getAge()){
            return -1;
        }
        return 0;

    }

    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (obj instanceof Customer) {
            Customer customer = (Customer) obj;
            if (this.name.equals(customer.getName())
                                    && this.age == customer.getAge()) {
                return true;
            } else {
                return false;
            }
        } else {
            return false;
        }
    }
    public int hashCode(){
        int result;
        result=(name==null?0:name.hashCode());
        result=29*result+age;
        return result;
    }
}
```

为了保证 TreeSet 能正确排序，要求 Customer 类的 compareTo()方法与 equals 方法按相同的规则比较两个 Customer 对象是否相等。也就是说，如果 customer1.equals(customer2)为 true，那么 customer1.compareTo(customer2)为 0。

测试代码如下：

```
Set<Customer> set = new TreeSet<Customer>();
set.add(new Customer("Tom",25));
set.add(new Customer("Tom",24));
set.add(new Customer("Jim",25));
Iterator<Customer> it=set.iterator();
while(it.hasNext()){
    Customer customer=it.next();
```

```
        System.out.println(customer.getName()+" "+customer.getAge());
    }
```

打印结果为：

```
Jim 25
Tom 24
Tom 25
```

10.3.3.2 客户化排序

除了自然排序，TreeSet 还支持客户化排序。使用 Java.util.Comparator<Type>接口提供具体的排序方式，<Type>指定被比较的对象的类型，Comparator 有个 compare(Type x,Type y)方法，用于比较两个对象的大小。当 compare(x,y)的返回值大于 0 时，表示 x 大于 y；当 compare(x,y)的返回值小于 0 时，表示 x 小于 y；当 compare(x,y)的返回值等于 0 时，表示 x 等于 y。

如果希望 TreeSet 按照 Customer 对象的 name 属性进行降序排列，可以先创建一个实现 Comparator 接口的类 CustomerComparator。

```
import java.util.Comparator;
import java.util.Iterator;
import java.util.Set;
import java.util.TreeSet;
public class CustomerComparator implements Comparator<Customer>{

    public int compare(Customer c1, Customer c2) {
        if(c1.getName().compareTo(c2.getName())>0){
            return -1;
        }else if(c1.getName().compareTo(c2.getName())<0){
            return 1;
        }
        return 0;
    }
    public static void main(String[] args){
        Set<Customer> set = new TreeSet<Customer>(new
                                            CustomerComparator());
        set.add(new Customer("Tom",25));
        set.add(new Customer("Tom",24));
        set.add(new Customer("Jim",25));
        Iterator<Customer> it=set.iterator();
        while(it.hasNext()){
            Customer customer=it.next();
            System.out.println(customer.getName()+" "+customer.getAge());
        }
    }
}
```

当 treeSet 向集合中加入 Customer 对象时，会调用 CustomerComparator 类的

compare()方法进行排序。以上 TreeSet 按照 Customer 对象的 name 属性进行降序排列，最后打印结果为：

```
Tom 25
Jim 25
```

因为 compare()方法使用 Customer 对象的 name 属性进行比较，set 集合只保存了第一个 Tom。

10.4 List(列表)

List 的主要特征是其元素以线性方式存储，集合中允许存放重复对象。list 接口主要的实现类包括：

ArrayList--ArrayList 代表长度可变的数组。允许对元素进行快速的随机访问，但是向 ArrayList 中插入与删除元素的速度较慢。

LinkedList—在实现中采用链表数据结构。对顺序访问进行优化，向 List 中插入和删除元素的速度较快，随机访问速度则相对较慢。随机访问是指检索位于特定索引位置的元素。LinkedList 单独具有 addFirst()、addLast()、getFirst()、getLast()、removeFirst() 和 removeLast()方法，这些方法使得 LinkedList 可以作为堆栈、队列和双向队列来使用。

10.4.1 访问列表的元素

List 中的对象按照索引位置排序，客户程序可以按照对象在集合中的索引位置来检索对象。以下程序向 List 中加入 4 个 Integer 对象。

```
List<Integer> list = new ArrayList<Integer>();
list.add(new Integer(3));
list.add(new Integer(2));
list.add(new Integer(1));
list.add(new Integer(2));
```

List 的 get(int index)方法返回集合中由参数 index 指定的索引位置的对象，第一个加入到集合中的对象的索引位置为 0，依次类推。以下程序依次检索出集合中的所有对象。

```
for(int i=0;i<list.size();i++){
    System.out.print(list.get(i)+" ");
}
```

以上程序的打印结果为：

```
3 2 1 2
```

List 的 iterator()方法和 Set 的 iterator()方法一样，也能返回 Iterator 对象，可以用 Iterator 来遍历集合中的所有对象，例如：

```
Iterator<Integer> it=list.iterator();
while(it.hasNext()){
    System.out.print(it.next());
```

```
    }
```

10.4.2 为列表排序

List 只能对集合中的对象按索引位置排序，如果希望对 List 中的对象按其他特定的方式排序，可以借助 Comparator 接口和 Collections 类。Collections 类是 Java 集合类库中的辅助类，它提供了操作集合的各种静态方法，其中 sort()方法用于对 List 中的对象进行排序。

sort(List list):对 List 中的对象进行自然排序。

sort(List list,Comparator comparator):对 List 中的对象进行客户化排序，comparator 参数指定排序方式。

以下程序对 List 中的 Integer 对象进行自然排序。

```
List<Integer> list = new ArrayList<Integer>();
list.add(new Integer(3));
list.add(new Integer(2));
list.add(new Integer(1));
list.add(new Integer(2));
Collections.sort(list);
for(int i=0;i<list.size();i++){
    System.out.print(list.get(i)+" ");
}
```

以上程序的打印结果为：

```
1 2 2 3
```

10.4.3 ListIterator 接口

List 的 listIterator()方法返回一个 ListIterator 对象，ListIterator 接口继承了 Iterator 接口，此外还提供了专门操作列表的方法。

add()：向列表中插入一个元素。

hasNext()：判断列表中是否还有下一个元素。

HasPrevious():判断列表中是否还有上一个元素。

next()：返回列表中的下一个元素。

previous()：返回列表中的上一个元素。

下面示例中的 insert()方法向一个排序的 List 列表中按顺序插入数据。

```
class ListInserter {
    //向List列表中按顺序插入元素
    public static void insert(List<Integer> list,int data) {
        ListIterator<Integer> it=list.listIterator();
        while(it.hasNext()){
            Integer in=it.next();
            if(data<=in.intValue()){
                //指针向前移动
                it.previous();
                //插入元素
                it.add(new Integer(data));
                break;
            }
        }
    }
    public static void main(String[] args){
        //创建一个链接列表
        List<Integer> list = new ArrayList<Integer>();
        list.add(new Integer(3));
        list.add(new Integer(5));
        list.add(new Integer(1));
        list.add(new Integer(2));
        //为列表排序
        Collections.sort(list);
        //向列表中插入一个元素
        insert(list,4);
        System.out.print(Arrays.toString(list.toArray()));
    }
}
```

以上程序的打印结果如下：

```
[1, 2, 3, 4, 5]
```

10.4.4 获得固定长度的 List 对象

Java.util.Arrays 类的 asList()方法能够把一个 Java 数据包装成一个 List 对象，这个 List 对象代表固定长度的数组。所有对 List 对象的操作都会被作用到底层的 Java 数组。由于数组的长度不能改变，因此不能调用这种 List 对象的 add()和 remove()方法，否则就会抛出 java.lang.UnsupportedOperationException 运行时异常。

```
String[] arr={"Smith","Tom","Jack"};
List<String> list=Arrays.asList(arr);
list.set(1, "jane"); //合法，可以修改某个位置的元素
System.out.println(Arrays.toString(arr)); //打印[Smith, jane, Jack]
list.remove("Tom");//非法，抛出UnsupportedOperationException
```

```
list.add("Mary");//非法，抛出UnsupportedOperationException
```

10.4.5 比较 Java 数组和各种 List 的性能

List 的两个实现类 ArrayList 和 LinkedList 都表示列表，在 JDK1.0 版本中有一个 Vector 类，也表示列表，在 JDK1.2 版本中把 Vector 类改为实现了 List 接口。

分别对 Java 数组、ArrayList、LinkedList、Vector 进行测试，测试的数据量是 10 万条数据，次数是 10 万次。测试结果如下(单位：毫秒)：

类型	Java 数组	ArrayList	LinkedList	Vector
随机访问操作(get)	10	16	56453	16
迭代操作(iterator)	0	0	47	16
插入操作(insert)	不适用	13281	232109	13297
删除操作(remove)	不适用	95031	167390	87687

总结如下：ArrayList 的效率相对平衡，对 LinkedList 进行操作，因为要先进行顺序查找，所以效率较低，Vector 类在各方面的性能也不是太差，属于历史集合类，已经不提倡使用它。

10.5 Map(映射)

Map(映射)是一种把键对象和值对象进行映射的集合，它的每一个元素都包含一对键对象和值对象，而值对象仍可以是 Map 类型，以此类推，这样就形成了多级映射。向 Map 集合中加入元素时，必须提供一对键对象和值对象，从 Map 集合中检索元素时，只要给出键对象，就会返回对应的值对象。以下程序通过 Map 的 put(Object key,Object value)方法向集合中加入元素，通过 Map 的 get(Obejct key)方法来检索与键对象对应的值对象。

```
Map<String,String> map=new HashMap<String,String>();
map.put("1", "Monday");
map.put("2", "Tuesday");
map.put("3", "Wendsday");
map.put("4", "Thursday");
String day=map.get("2");//day的值为"Tuesday"
```

Map 集合中的键对象不允许重复，也就是说任意两个键对象通过 equals()方法比较的结果都是 false。对于值对象则没有唯一性的要求，可以将任意多个键对象映射到一个值对象上。例如以下 Map 集合中的键对象“1”和“one”都和同一个值对象“Monday”对应。

```
Map<String,String> map=new HashMap<String,String>();
map.put("1", "Mon");
map.put("1", "Monday");
map.put("one", "Monday");

Set<String> set=map.keySet();
Iterator<String> it=set.iterator();
while(it.hasNext()){
    String key=it.next();
```

```
        String value=map.get(key);
        System.out.println("key="+key+",value="+value);
    }
```

由于第一次和第二次加入 Map 中的键对象都为"1"，因此第一次加入的值对象将被覆盖，Map 集合中最后只有两个元素，分别为：

```
    1--->Monday

    one--->Monday
```

Map 的 keySet()方法返回集合中所有键对象的集合。

Map 有两种比较常用的实现：HashMap 和 TreeMap。HashMap 按照哈希算法来存取键对象，有很好的存取性能，为了保证 HashMap 能正常工作，和 HashSet 一样，要求当两个键对象通过 equals()方法比较为 true 时，这两个键对象的 hashCode()方法返回的哈希码也一样。

TreeMap 实现了 SortedMap 接口，能对键对象进行排序。和 TreeSet 一样，TreeMap 也支持自然排序和客户化排序两种方式。以下程序中 TreeMap 会对 4 个 String 类型的键对象进行自然排序。

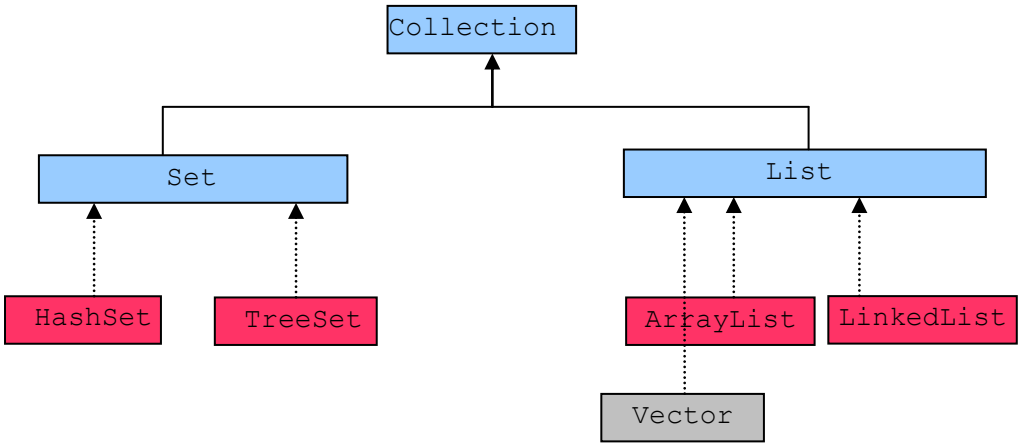
```
    Map<String,String> map=new TreeMap<String,String>();
    map.put("3", "Wendsday");
    map.put("1", "Monday");
    map.put("4", "Thursday");
    map.put("2", "Tuesday");

    Set<String> set=map.keySet();
    Iterator<String> it=set.iterator();
    while(it.hasNext()){
        String key=it.next();
        String value=map.get(key);
        System.out.println("key="+key+",value="+value);
    }
```

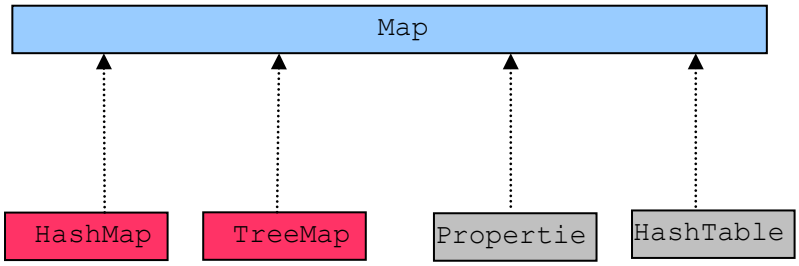
以上程序的打印结果为：

```
    key=1,value=Monday
    key=2,value=Tuesday
    key=3,value=Wendsday
    key=4,value=Thursday
```

如果希望 TreeMap 对键对象进行客户化排序，可调用它的另一个构造方法---TreeMap(Comparator comparator)，参数 comparator 指定具体的排序方式。

10.6 历史集合类





在早期的 JDK1.0 版本中，代表集合的类只有 Vector、Stack、Enumeration、Hashtable、Properties 和 BitSet。从 JDK1.2 版本开始，才出现了 Collection、Set、List 和 Map 接口及各种实现类，它们构成了完整的 Java 集合框架。JDK1.0 版本中的集合类也称为历史集合类。

历史集合类	描述	缺点	替代类
Vector	集合中的元素有索引位置，在新的集合框架中把它改为实现了 List 接口	采用了同步机制，影响操纵集合的性能	ArrayList LinkedList
Stack	表示堆栈，支持后进先出的操作	采用了同步机制，影响操作集合的性能；Stack 继承了 Vector 类，使得 Stack 不能作为严格的堆栈，还允许随机访问	LinkedList
Hashtable	集合中的每个元素包含一对键与值。在新的集合框架中把它改为实现了 Map 接口	采用了同步机制，影响操作集合的性能	HashMap
Properties	集合中的每个元素包含一对键与值，继承了 Hashtable	采用了同步机制，影响操作集合的性能	无
Enumeration	用于遍历集合中的元素	只能与 Vector 和 Hashtable 等历史集合配套使用；Enumeration 类的名	Iterator

		字较长，Iterator 类的名字简短	
BitSet	存放一组 boolean 类型数据，支持与、或和异或操作	无	无

10.7 数据结构*

10.7.1 线性表

线性表(List)是由叫作元素(Element)的数据项组成的一种有限并且有序的序列。有序是指线性表中的每一个元素都有自己的位置(index)。

线性表中的元素有相同的类型；

线性表中不包含任何元素时，我们称之为空表(empty List)；

当前实际存储的元素数据的个数叫做线性表的长度(length)；

线性表的开始节点（第一个元素）叫作表头(head)；

线性表的结尾节点（最后一个元素）叫作表尾(tail)；

如果线性表的元素按照值的递增顺序排列，我们称之为有序线性表(sorted list)；

如果线性表的元素与位置没有关系，我们称之为无序线性表(unsorted list)；

线性表的表示：$(a_0,a_1,a_2,\ldots\ldots,a_{n-1})$

线性表的操作

```
public interface List {
    //remove all Objects from list
    public void clear();

    //insert Object at current position
    public void insert(Object item);

    //insert Object at tail of list
    public void append(Object item);

    //remove/return current Object
    public Object remove();
```

```
    //set current to first position
    public void setFirst();

    //move current to next position
    public void next();

    //move current to prev position
    public void prev();

    //return current length of list
    public int length();

    //set current to specified pos
    public void setPost(int pos);

    //set current Object's value
    public void setValue(Object val);

    //return value of current Object
    public Object currValue();

    //return true if list is empty
    public boolean isEmpty();

    //return true if current is within list
    public boolean isInList();

    //print all of list's elements
    public void print();
}
```

线性表有两种标准的实现方法：顺序表(array-based list)和链表(linked list)。

10.7.1.1 顺序表

顺序表的实现是用数组来存储表中的元素，这就意味着将要分配固定长度的数组。因此当线性表生成时数组的长度必须是已知的。既然每个线性表可以有一个不等长的数组，那么这个长度就必须由线性表对象来记录。在任何给定的时刻，线性表都有一定数目的元素，这个数目应该小于数组允许的最大值。线性表中当前的实际元素数目也必须在线性表中记录。

顺序表把表中的元素存储在数组中相邻的位置上。数组的位置与元素的位置相对应。换句话说，就是表中第 i 个元素存储在数组的第 i 个单元中。表头总是在第 0 个位置，这就使得对表中任意一个元素的随机访问相当容易。给出表中的某一个位置，这个位置对应元素的值就可以直接获取。

如果在表尾进行插入和删除操作很方便；如果在表中插入一个元素，就要将此元素之后的所有元素向后移动一个位置；如果在表中删除一个元素，就要将此元素之后的所有元素向前移动一个位置。

```
public class ArrayList implements List {

    // default array size
    private static final int defaultSize = 10;
    // max size of list
    private int maxSize;
    // actual number of Objects in list
    private int length;
    // position of current Object
    private int curr;
    // Array holding list Objects
    private Object[] listArray;

    /**
     * Constructor use default size
     */
    public ArrayList() {
        setUp(defaultSize);
    }

    /**
     * Constructor use specified size
     */
    public ArrayList(int size) {
        setUp(size);
    }

    private void setUp(int size) {
        maxSize = size;
        length = curr = 0;
        listArray = new Object[size];
    }

    //remove all Objects from list
    public void clear(){
        length = curr = 0;
    }

    //insert Object at current position
    public void insert(Object item){
        if (isFull()){
            throw new RuntimeException("顺序表达到最大容量");
        }
        for(int i=length;i>curr;i--){
            listArray[i]=listArray[i-1];
        }
        listArray[curr]=item;
        length++;
    }
```

```
//insert Object at tail of list
public void append(Object item){
    if (isFull()){
        throw new RuntimeException("顺序表达到最大容量");
    }
    listArray[length] = item;
    length++;
}

//remove/return current Object
public Object remove(){
    if (isEmpty()){
        throw new RuntimeException("顺序表为空，无法再删除了");
    }
    Object o=listArray[curr];
    for(int i=curr;i<length-1;i++){
        listArray[i]=listArray[i+1];
    }
    length--;
    return o;
}

//set current to first position
public void setFirst(){
    curr=0;
}

//move current to next position
public void next(){
    curr++;
}

//move current to prev position
public void prev(){
    curr--;
}

//return current length of list
public int length(){
    return length;
}

//set current to specified post
public void setPost(int pos){
    curr=pos;
}

//set current Object's value
```

```
    public void setValue(Object val){
        listArray[curr]=val;
    }

    //return value of current Object
    public Object currValue(){
        return listArray[curr];
    }

    //return true if list is empty
    public boolean isEmpty(){
        if(length==0) {
            return true;
        } else {
            return false;
        }
    }

    //return true if list is full
    public boolean isFull(){
        if (length == maxSize){
            return true;
        }else{
            return false;
        }
    }

    //return true if current is within list
    public boolean isInList(){
        return curr>=0 && curr<length;
    }

    //print all of list's elements
    public void print(){
        if(isEmpty()){
            System.out.println("[]");
        }else{
            System.out.print("[");
            for(setFirst();isInList();next()){
                System.out.print(currValue()+" ");
            }
            System.out.print("]");
        }
    }
}
```

10.7.1.2 链表

链表是利用指针来实现的，是动态的，也就是说，能够按照需要为表中新的元素分配存储空间。

链表是由一系列叫作结点的对象组成，每一个结点都是一个独立的对象。下面代码中表示了结点的完整定义，Link 类包含一个存储单元值的 element 属性和一个存储表中下一个结点指针的 next 属性。因为在这种结点建立的链表中，每个结点只有一个指向表中下一个节点的指针，所以叫作单链表（singly linked list）

```
public class Link {
    //object for this node
    private Object element;
    //pointer to next node in list
    private Link next;

    Link(Object element,Link next){
        this.element=element;
        this.next=next;
    }
    Link(Link next){
        this.next=next;
    }
    Link next(){
        return next;
    }
    Link setNext(Link next){
        this.next=next;
        return next;
    }
    Object element(){
        return element;
    }
    Object setElement(Object element){
        return this.element=element;
    }
}
```

Link 类非常简单。它有两个构造方法，一个有初始化元素的值，另一个没有。其他方法帮助用户很方便的访问两个私有数据成员。

单链表类的实现代码如下：

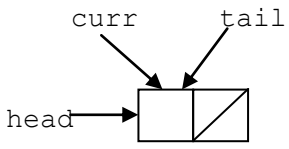
```
public class LinkList implements List{
    //pointer to list header
    private Link head;
    //pointer to last Object in list
    private Link tail;
    //pointer to current Object
    protected Link curr;
```

```
public LinkList(){
    tail=head=curr=new Link(null);
}

//remove all Objects from list
public void clear(){
    head.setNext(null);
    curr=tail=head;
}

//insert Object at current position
public void insert(Object item){
    curr.setNext(new Link(item,curr.next()));
    if(tail==curr){
        tail=curr.next();
    }
}

//insert Object at tail of list
public void append(Object item){
    tail.setNext(new Link(item,null));
    tail=tail.next();
}

//remove/return current Object
public Object remove(){
    if(!isInList()){
        throw new RuntimeException("当前节点为null");
    }
    Object o=curr.element();
    if(null==curr.next()){
        prev();
        curr.setNext(null);
        tail=curr;
    } else{
        Link temp=    curr.next();
        prev();
        curr.setNext(temp);
    }
    return o;
}

//set current to first position
public void setFirst(){
    curr=head.next();
}

//move current to next position
```

```
public void next(){
    if(curr!=null){
        curr=curr.next();
    }
}

//move current to prev position
public void prev(){
    if(curr==null || curr==head){ //no previous object
        return;                   //so just return
    }
    Link temp=head;               //start at front of list
    while(temp!=null && temp.next()!=curr){
        temp=temp.next();
    }
    curr=temp;                    //found previous link
}

//return current length of list
public int length(){
    int count=0;
    for(Link temp=head.next();temp!=null;temp=temp.next()){
        count++;
    }
    return count;
}

//set current to specified post
public void setPost(int pos){
    curr=head;
    for(int i=0;curr!=null && i<pos;i++){
        curr=curr.next();
    }
}

//set current Object's value
public void setValue(Object val){
    if(isInList()){
        curr.setElement(val);
    }else{
        throw new RuntimeException("当前节点为null");
    }
}

//return value of current Object
public Object currValue(){
    if(curr==null){
        return null;
    }
```

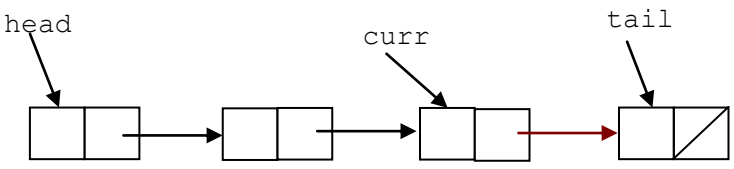
```
        return curr.element();
    }

    //return true if list is empty
    public boolean isEmpty(){
        return head.next()==null;
    }

    //return true if current is within list
    public boolean isInList(){
        return curr!=null && curr.next()!=null;
    }

    //print all of list's elements
    public void print(){
        if(isEmpty()){
            System.out.println("[]");
        }else{
            System.out.print("[");
            for(setFirst();isInList();next()){
                System.out.print(currValue()+ " " );
            }
            System.out.println("]");
        }
    }
}
```

单链表中增加了特殊的表态结点，这个表头结点是表中的第一个结点，它与表中其他元素一样，只是它的值被忽略，不被看做表中的实际元素。这样我们就不再需要考虑空链表，表中一定有一个结点，这样的设计节省了源代码，降低了源程序的复杂性。存储结构如下：



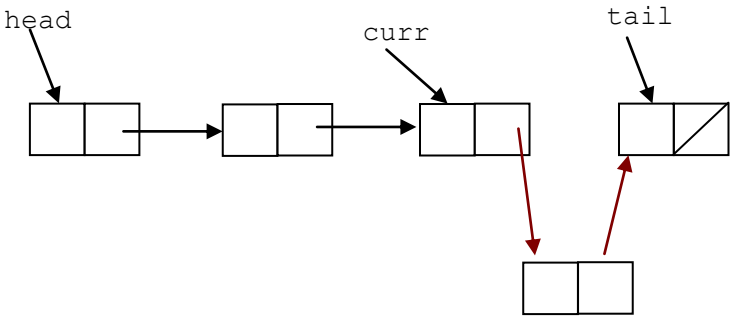
空的单链表（只有表头元素）

当使用 append 方法向表中添加新的元素时，只需要改变 tail 指向结点的 next 属性，将新的节点赋值给该属性即可。



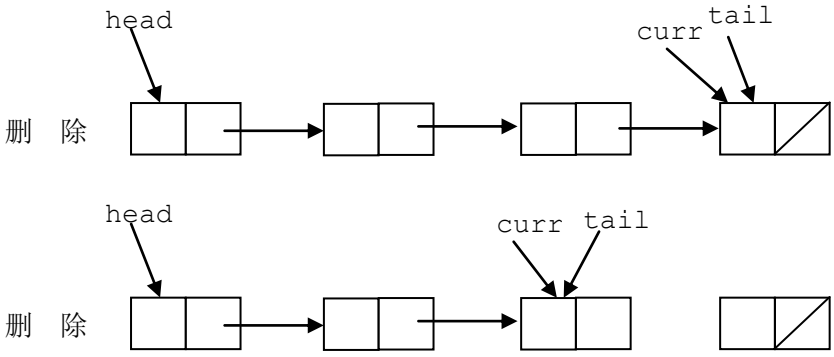
append 方法添加新元素

当使用 insert 方法时，向 curr 指向的结点后添加一个新的节点。

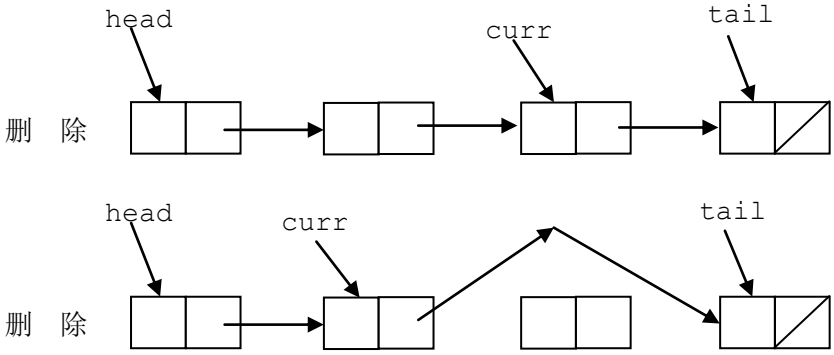


insert 方法添加新元素

当使用 remove 方法时，将 curr 指向的结点删除，如果 curr 指向尾结点，将 curr 指针向前移动，设置 next 指向空；如果 curr 不指向尾结点，取出 curr 的下一个结点 temp，将 curr 指针向前移动，设置 curr 指针的 next 为 temp 结点。



当 curr 指向尾结点时使用 remove



当 curr 没有指向尾结点时使用 remove

注：代码中设计删除当前结点后，指针向前移动；也可以设计成删除结点后，指针向后移动。

10.7.1.3 线性表实现方法的比较

前面已经给出线性表的两种截然不同的实现方法，哪一种更好呢，当使用线性表时如何选择呢？

顺序表的缺点是长度固定。虽然便于给数组分配空间，但它不仅不能超过预定的长度，而且当线性表中只有几个元素时，浪费了相当多的空间。

链表的优点是只有实际在链表中的对象需要空间。只有有可用的内存空间分配，链表中元素的个数就没有限制。

顺序表的优点是对于表中每一个元素没有浪费**空间**，而链表需要在每个结点上附加一个指针。如果 element 对象占据的空间较小，则链表的结构性开销就占去了整个存储空间的一大部分。当顺序表被填满时，存储上没有结构性开销。这这种情况下，书序表有更高的空间效率。

所以当线性表元素**数目变化**较大或者未知时，最好使用链表实现。而如果用户事先知道线性表的大致长度，使用顺序表的空间效率会更高。

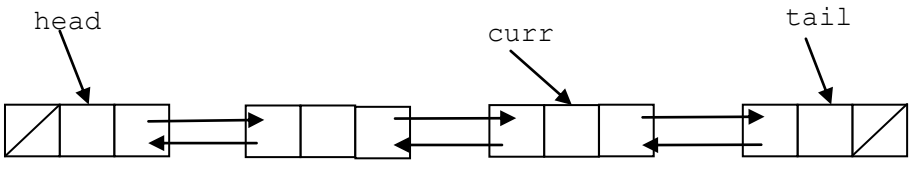
当要求对线性表进行**随机访问**时，使用顺序表更快一些；通过 next 和 prev 只可以调整当前位置向前或向后移动。其中 next 操作很容易实现，而对于 prev 操作，在单链表中不能直接访问前面的元素，只能从表头开始寻找那个特定的位置。

当进行**插入**和**删除**操作时，链表比顺序表的时间效率更高。

10.7.1.4 双链表

当链表只允许从一个表结点直接访问它的后继结点，而双链表（double linked list）可以从一个表结点出发，方便地再线性表中访问它的前驱结点和后继结点。

双链表存储了两个指针，一个指向它的后继结点（与单链表相同），另一个指向它的前驱结点。结构如下：



双链表结构

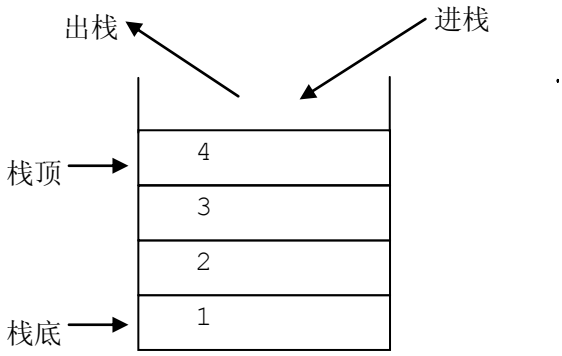
双链表比单链表更容易实现，因为有一个指针 prev 可以访问前一个结点，使得指针 curr 先前或向后移动变得非常方便。

10.7.2 栈

栈(stack)是限定仅在一端进行插入或删除的线性表。虽然这个限制减小了栈的灵活性，但

是会使栈更有效且更容易实现。

栈的访问遵循“后进先出”(last in first out)原则，就是栈的可访问元素为栈顶(top)元素，元素的插入栈称作压栈(push)，删除元素时称作出栈(pop)。图示如下：



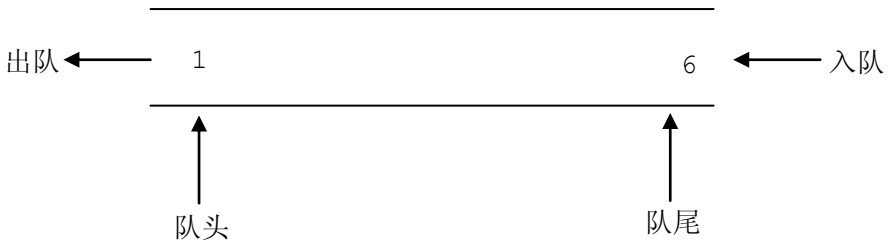
思考题：如果进栈顺序为 1、2、3、4，那么下面哪种出栈顺序是可以实现的？
1、4、3、2
1、3、4、2
4、3、2、1
3、4、2、1

两种常用实现方法：顺序栈(array-based stack)和链式栈(linked stack)

10.7.3 队列

同栈一样，队列(queue)也是一种受限的线性表。队列元素只能从队尾插入(称为入队操作)，从队首进行删除(称为出队操作)。队列操作就像在电影院前排队买票一样，如果没有人为破坏，那么新来者应该站在队列的后端，在队列最前端的人是下一个要被服务的对象。

队列的访问遵循“先进先出”(first in first out)原则，图示如下：



两种常用实现方法：顺序栈(array-based queue)和链式栈(linked queue)

10.8 学习目标

1. 能够描述集合框架的结构：Collection 接口、List 接口、Set 接口、Map 接口及实现类

2． 理解 List 接口和 Set 接口的特点。

3． 理解 Map 接口的特点。

4． 熟练掌握 Collection 接口的方法，重点是迭代方法。

5． 熟练掌握 Map 接口中的方法，重点是迭代方法。

6． 查看 JDK API 文档中和集合相关的 API。

7． 完成课后作业题，通过老师的讲解，理解 ColDB 的作用和原理。（讲过作业题后）

10.9 练习

1． 写一个方法 reverseList，该方法接收一个 List 参数，然后把该 List 倒序排列。

2． 综合设计题：定义一个类 ListDB，类里面有一个属性 list,类型是集合类型 List，实现下列方法：可以向 list 里面添加数据、修改数据、查询数据、删除数据。也就是把这个 list 当作一个数据存储的容器，对其实现数据的增删改查的方法。注意这些方法是否需要传入参数，是否应该有返回类型。

3． 综合设计题：已知有十六支男子足球队参加 2008 北京奥运会。写一个程序，把这 16 只球队随机分为 4 个组。
2008 北京奥运会男足参赛国家：科特迪瓦，阿根廷，澳大利亚，塞尔维亚，荷兰，尼日利亚、日本，美国，中国，新西兰，巴西，比利时，韩国，喀麦隆，洪都拉斯，意大利。

11 图形用户接口 AWT

11.1 GUI 的基本概念

11.1.1 Component

AWT 提供用于所有 Java applets 及应用程序中的基本 GUI 组件，还为应用程序提供与机器的界面。这将保证一台计算机上出现的东西与另一台上的相一致。

在学 AWT 之前，简单回顾一下对象层次。记住，超类是可以扩展的，它们的属性是可继承的。而且，类可以被抽象化，这就是说，它们是可被分成子类的模板，子类用于类的具体实现。

显示在屏幕上的每个 GUI 组件都是抽象类组件的子类。也就是说，每个从组件类扩展来的图形对象都与允许它们运行的大量方法和实例变量共享。

11.1.2 Container

Container 是 Component 的一个抽象子类，它允许其它的组件被嵌套在里面。这些组件也可以是允许其它组件被嵌套在里面的容器，于是就创建了一个完整的层次结构。在屏幕上布置 GUI 组件，容器是很有用的。Panel 是 Container 的最简单的类。Container 的另一个子类是 Window。

Window 是 Java.awt.Window 的对象。Window 是显示屏上独立的本机窗口，它独立于其它容器。

Window 有两种形式：Frame(框架)和 Dialog(对话框)。Frame 和 Dialog 是 Window 的子类。Frame 是一个带有标题和缩放角的窗口。对话没有菜单条。尽管它能移动，但它不能缩放。

Panel 是 Java.awt.Panel 的对象。Panel 包含在另一个容器中，或是在 Web 浏览器的窗口中。Panel 确定一个四边形，其它组件可以放入其中。Panel 必须放在 Window 之中（或 Window 的子类中）以便能显示出来。

容器不但能容纳组件，还能容纳其它容器，这一事实对于建立复杂的布局是关键的，也是基本的。

11.1.3 组件定位

容器里的组件的位置是由布局管理器决定的。容器对布局管理器的特定实例保持一个引用。当容器需要定位一个组件时，它将调用布局管理器来做。当决定一个组件的大小时，同样如此。

布局管理器完全控制容器内的所有组件。它负责计算并定义上下文中对象在实际屏幕中所需的大小。

11.2 容器的基本使用

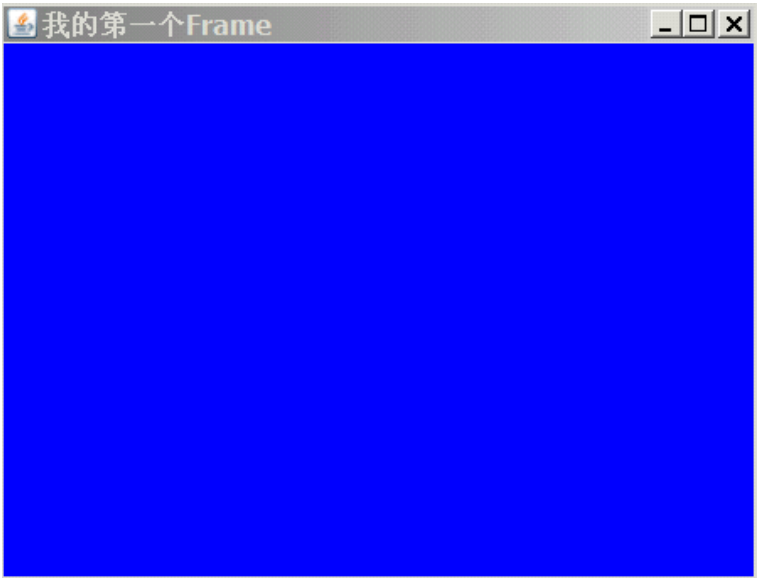
11.2.1 Frame 的使用

Frame 是 Window 的一个子类。它是带有标题和缩放角的窗口。它继承于 Java.awt.Container，因此，可以用 add()方式来给框架添加组件。框架的缺省布局管理器就是 Border Layout。它可以用 setLayout()方式来改变。

框架类中的构造程序 Frame(String)用由 String 规定的标题来创建一个新的不可见的框架对象。当它还处于不可见状态时，将所有组件添加到框架中。

```
import java.awt.*;

public class MyFrame extends Frame {
    public MyFrame(String str) {
        super(str);
    }
    public static void main(String args[]) {
        MyFrame fr = new MyFrame("我的第一个Frame");
        // 设置大小
        fr.setSize(500, 500);
        // 设置背景颜色
        fr.setBackground(Color.blue);
        // 设置框架可见
        fr.setVisible(true);
    }
}
```

上述程序创建了下述框架，它有一个具体的标题、大小及背景颜色。



在框架显示在屏幕上之前，必须做成可见的（通过调用程序 setVisible(true)），而且其大小是确定的（通过调用程序 setSize()或 pack()）。

11.2.2 Panel 的使用

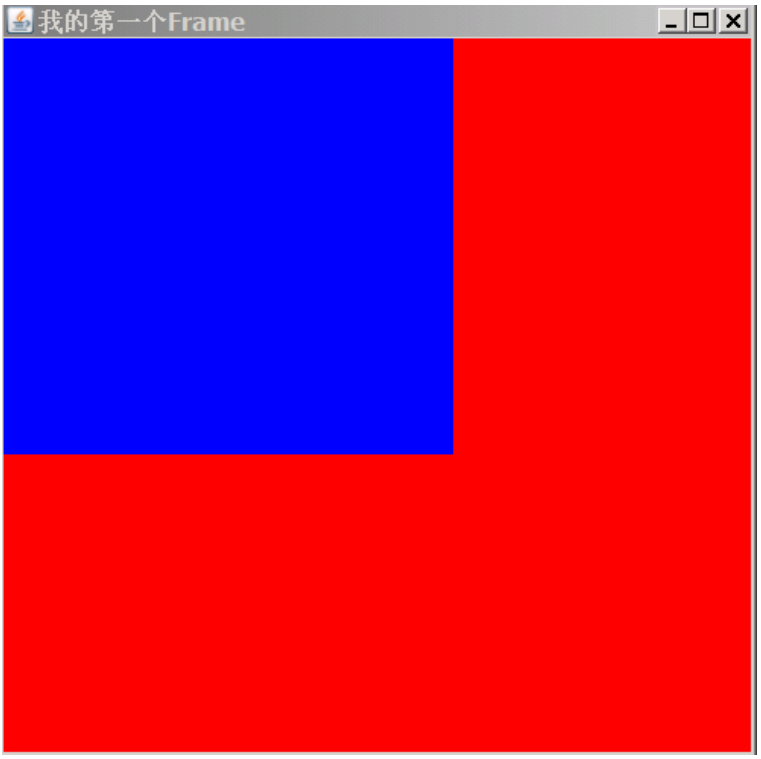
像 Frame 一样，Panel 提供空间来连接任何 GUI 组件，包括其它面板。每个面板都可以有它自己的布管理程序。

一旦一个面板对象被创建，为了能看得见，它必须添加到窗口或框架对象上。用 Container 类中的 add()方式可以做到这一点。

下面的程序创建了一个小的黄色面板，并将它加到一个框架对象上：

```
import java.awt.*;

public class FrameWithPanel extends Frame {

    // Constructor
    public FrameWithPanel(String str) {
        super(str);
    }

    public static void main(String args[]) {
        FrameWithPanel fr = new FrameWithPanel("我的第一个Frame");
        Panel pan = new Panel();

        fr.setSize(500, 500);
        fr.setBackground(Color.red);
```

```
        fr.setLayout(null); // override default layout mgr
        pan.setSize(300, 300);
        pan.setBackground(Color.blue);
        pan.setLayout(null); // override default layout mgr

        fr.add(pan);
        fr.setVisible(true);
    }
}
```



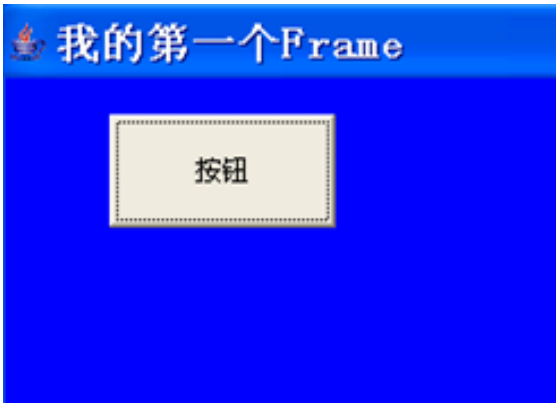
11.3 AWT 组件的使用

AWT 组件提供了控制界面外观的机制，包括用于文本显示的颜色和字体。此外，AWT 还支持打印。这个功能是在 JDK1.1 之后中引入的。

11.3.1 按钮(Button)

你已经比较熟悉 Button 组件了。这个组件提供了“按下并动作”的基本用户界面。可以构造一个带文标签的按钮，用来告诉用户它的作用。

```
    Button b=new Button("按钮");
    b.setSize(100,50);
        b.setLocation(100,100);
    pan.add(b);
```



11.3.2 复选框(Checkbox)

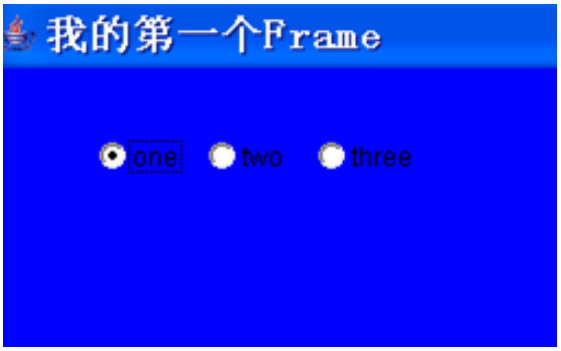
Checkbox 组件提供一种简单的“开/关”输入设备，它旁边有一个文本标签。

```
    Checkbox one=new Checkbox("one",true);
        Checkbox two =new Checkbox("two",false);
        Checkbox three =new Checkbox("three",false);

        one.setSize(50,50);
        two.setSize(50,50);
        three.setSize(50,50);

        one.setLocation(50,50);
        two.setLocation(100,50);
        three.setLocation(150,50);

        pan.add(one);
        pan.add(two);
        pan.add(three);
```

11.3.3 复选框组－单选框(Checkbox group-Radio Button)

复选框组提供了将多个复选框作为互斥的一个集合的方法，因此在任何时刻，这个集合中只有一个复选框的值是 true。值为 true 的复选框就是当前被选中的复选框。你可以使用带有一个额外的 CheckboxGroup 参数的构造函数来创建一组中的每个复选框。正是这个 CheckBoxGroup 对象将各个复选框连接成一组。如果你这么做的话，那么复选框的外观会发生改变，而且所有和一个复选框 组相关联的复选框将表现出"单选框"的行为。

```
CheckboxGroup cbg=new CheckboxGroup();

Checkbox one=new Checkbox("one",true,cbg);
Checkbox two =new Checkbox("two",false,cbg);
Checkbox three =new Checkbox("three",false,cbg);

one.setSize(50,50);
two.setSize(50,50);
three.setSize(50,50);

one.setLocation(50,50);
two.setLocation(100,50);
three.setLocation(150,50);

pan.add(one);
pan.add(two);
pan.add(three);
```



11.3.4 下拉列表(Choice)

下拉列表组件提供了一个简单的"从列表中选取一个"类型的输入。例如：

```
Choice c=new Choice();

c.add("First");
c.add("Second");
c.add("Third");

c.setSize(100,100);
c.setLocation(50,50);

pan.add(c);
```

点击下拉列表组件时，它会显示一个列表，列表中包含了所有加入其中的条目。注意所加入的条目是 String 对象。

11.3.5 标签(Label)

一个标签对象显示一行静态文本。程序可以改变文本，但用户不能改变。标签没有任何特殊的边框和装饰。

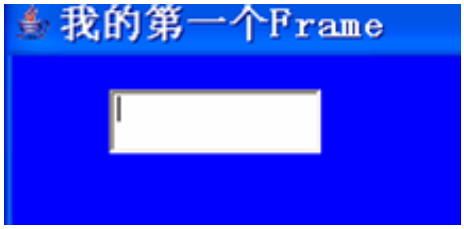
```
Label lbl=new Label("HelloWorld");
lbl.setSize(100,50);
lbl.setLocation(50,50);

pa.add(lbl);
```

标签通常不处理事件，但也可以按照和画布相同的方式来处理事件。也就是说，只有调用了requestFocus()方法后，才能可靠地检取击键事件。

11.3.6 文本域(Textfield)

文本域是一个单行的文本输入设备。例如：

```
TextField tf=new TextField();

tf.setSize(300,30);
tf.setLocation(50,50);

pan.add(tf);
```



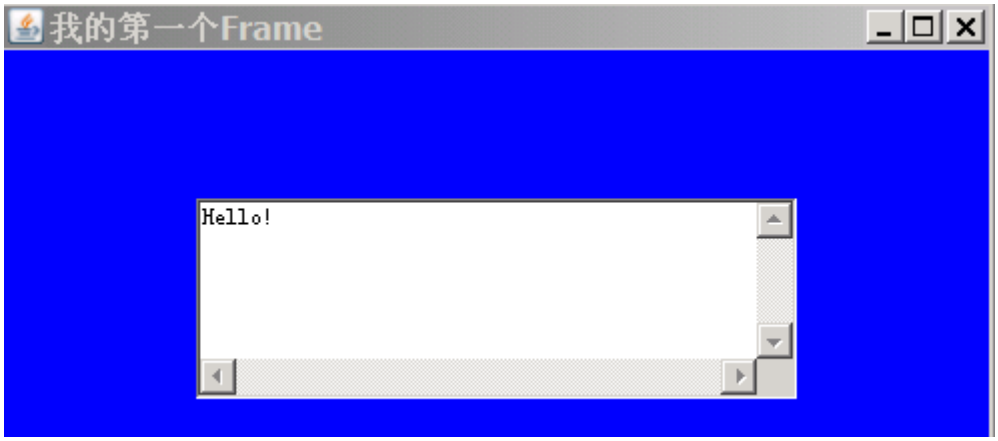
11.3.7 文本区(TextArea)

文本区是一个多行多列的文本输入设备。

文本区将显示水平和垂直的滚动条。

下面这个范例创建了一个文本区，最初它含有“Hello!”。

```
TextArea t = new TextArea("Hello!");
t.setLocation(100, 100);
t.setSize(300, 100);
add.add(t);
```



11.3.8 列表(list)

一个列表将各个文本选项显示在一个区域中，这样就可以在同时看到若干个条目。列表可以滚动，并支持单选和多选两种模式。例如：

```
List lst=new List();

lst.add("篮球");
lst.add("足球");
lst.add("排球");

lst.setSize(50,60);
lst.setLocation(50,50);
lst.setMultipleMode(true);

pa.add(lst);
```

11.4 菜单的实现

菜单与其他组件有一个重要的不同：你不能将菜单添加到一般的容器中，而且不能使用布局管理器对它们进行布局。你只能将菜单加到一个菜单容器中。然而，你可以将一个 Jmenuswing 组件加到一个 Jcontainer 中。你可以通过使用 setMenuBar()方法将菜单放到一个框架中，从而启动一个 菜单“树”。从那个时刻之后，你可以将菜单加到菜单条中，并将菜单或菜单项加到菜单中。

弹出式菜单是一个例外，因为它们可以以浮动窗口形式出现，因此不需要布局。

11.4.1 帮助菜单

菜单条的一个特性是你可以将一个菜单指定为帮助菜单。这可以用 setHelpMenu(Menu)来做到。 要作为帮助菜单的菜单必须加入到菜单条中；然后它就会以和本地平台的帮助菜单同样的方式被处理。对于 X/Motif 类型的系统，这涉及将菜单条放置在菜单条的最右边。

11.4.2 菜单条(MenuBar)

一个菜单条组件是一个水平菜单。它只能加入到一个框架中，并成为所有菜单树的根。在一个时刻，一个框架可以显示一个菜单条。然而，你可以根据程序的状态修改菜单条，这样在不同的时 刻就可以显示不同的菜单。例如：

```
MenuBar mb = new MenuBar();
frame.setMenuBar(mb);
```

没有菜单的菜单条是看不到的，所以大家在 Frame 中看不到菜单条的。

11.4.3 菜单组件

菜单组件提供了一个基本的下拉式菜单。它可以加入到一个菜单条或者另一个菜单中。例如：

```
MenuBar mb = new MenuBar();
Menu m1 = new Menu("File");
```

```
        Menu  m2  =  new  Menu("Edit");
        Menu  m3  =  new  Menu("Help");
        mb.add(m1);
        mb.add(m2);
        mb.setHelpMenu(m3);
        frame.setMenuBar(mb);
```

这里显示的菜单是空的，这正是 File 菜单的外观。

11.4.4 菜单项(MenuItem)

菜单项组件是菜单树的文本"叶"结点。它们通常被加入到菜单中，以构成一个完整的菜单。例如：

```
    MenuItem mi1 = new MenuItem("New");
    MenuItem mi2 = new MenuItem("Load");
    MenuItem mi3 = new MenuItem("Save");
    MenuItem mi4 = new MenuItem("Quit");
    m1.add(mi1);
    m1.add(mi2);
    m1.addSeparator();
    m1.add(mi3);
```

11.4.5 复选菜单项(CheckboxMenuItem)

复选菜单项是一个可复选的菜单项，所以你可以在菜单上有选项(“开”或“关”)。例如：

```
MenuItem mi1 = new MenuItem("Save");
    CheckboxMenuItem mi2 =new CheckboxMenuItem("Persistent");
    m1.add(mi1);
    m1.add(mi2);
```

11.4.6 弹出式菜单(PopupMenu)

弹出式菜单提供了一种独立的菜单，它可以在任何组件上显示。你可以将菜单条目和菜单加入到弹出式菜单中去。例如：

```
PopupMenu p = new PopupMenu("Popup");
MenuItem s = new MenuItem("Save");
MenuItem l = new MenuItem("Load");

p.add(s);
p.add(l);
frame.add(p);
//显示弹出式菜单
p.show(fr, 100, 70);
```

注－弹出式菜单必须加入到一个“父”组件中。这与将组件加入到容器中是不同的。在上面这个范例中，弹出式菜单被加入到周围的框架中。

11.5 控制外观

你可以控制在 AWT 组件中所显示的文本的前景背景颜色、背景颜色和字体。

11.6 布局管理器和布局

11.6.1 FlowLayout(流式布局)

使用 FlowLayout 布局方式的容器中组件按照加入的先后顺序按照设置的对齐方式（居中、左对齐、右对齐）从左向右排列，一行排满（即组件超过容器宽度后）到下一行开始继续排列。

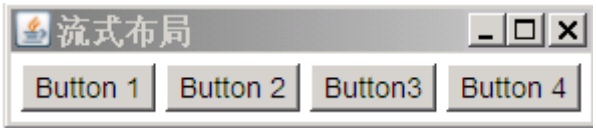
流式布局特征如下：

组件按照设置的对齐方式进行排列

不管对齐方式如何，组件均按照从左到右的方式进行排列，一行排满，转到下一行。（比如按照右对齐排列，第一个组件在第一行最右边，添加第二个组件时，第一个组件向左平移，第二个组件变成该行最右边的组件，这就是从左向右方式进行排列）

```
import java.awt.Button;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.Frame;
public class FlowLayoutDemo extends Frame {

    public FlowLayoutDemo() {
        // 设置窗体为流式布局，无参数默认为居中对齐
        setLayout(new FlowLayout());
        // 设置窗体中显示的字体样式
        setFont(new Font("Helvetica", Font.PLAIN, 14));
        // 将按钮添加到窗体中
        this.add(new Button("Button 1"));
        this.add(new Button("Button 2"));
        this.add(new Button("Button3"));
        this.add(new Button("Button 4"));
    }

    public static void main(String args[]) {
        FlowLayoutDemo frame = new FlowLayoutDemo();
        frame.setTitle("流式布局");
        // 该代码依据放置的组件设定窗口的大小使之正好能容纳你放置的所有组件
        frame.pack();
        frame.setVisible(true);
        frame.setLocationRelativeTo(null); // 让窗体居中显示

    }
}
```

原始界面：



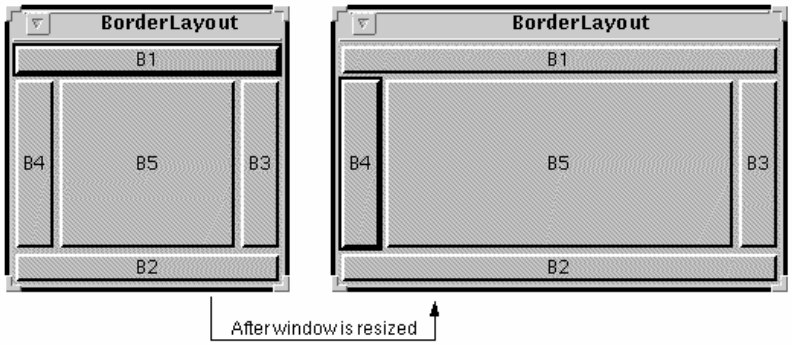
拉伸原始界面：

拉窄原始界面：

11.6.2 BorderLayout(边框布局)

Border 布局管理器为在一个 Panel 或 Window 中放置组件提供一个更复杂的方案。Border 布局管理器包括五个明显的区域：东、南、西、北、中。

北占据面板的上方，东占据面板的右侧，等等。中间区域是在东、南、西、北都填满后剩下的区域。当窗口垂直延伸时，东、西、中区域也延伸；而当窗口水平延伸时，东、西、中区域也延伸。 Border 布局管理器是用于 Dialog 和 Frame 的缺省布局管理器。



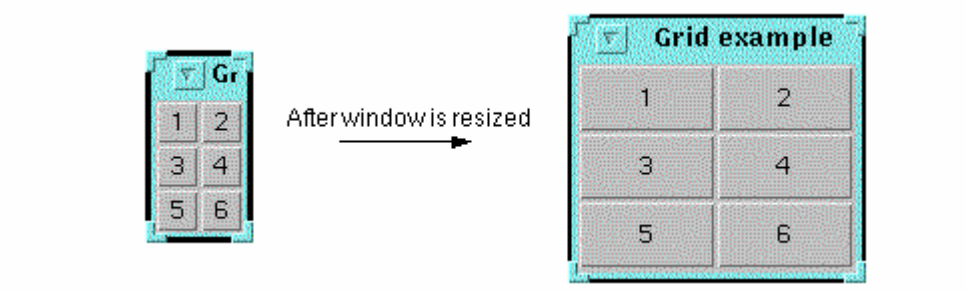
当窗口缩放时，按钮相应的位置不变化，但其大小改变。

```
import java.awt.BorderLayout;
import java.awt.Button;
import java.awt.Frame;
public class BorderLayoutDemo extends Frame{
    public BorderLayoutDemo(){
        Button N=new Button("北");
        Button S=new Button("南");
        Button W=new Button("西");
        Button E=new Button("东");
        Button C=new Button("中");

        this.add(N,BorderLayout.NORTH);
        this.add(S,BorderLayout.SOUTH);
        this.add(W,BorderLayout.WEST);
        this.add(E,BorderLayout.EAST);
        this.add(C,BorderLayout.CENTER);
    }
    public static void main(String[] args){
        BorderLayoutDemo frame = new BorderLayoutDemo();
        frame.setTitle("边框布局");
        // 该代码依据放置的组件设定窗口的大小使之正好能容纳你放置的所有组件
        frame.pack();
        frame.setVisible(true);
        frame.setLocationRelativeTo(null); // 让窗体居中显示
    }
}
```

11.6.3 GridLayout(网格布局)

Grid 布局管理器为放置组件提供了灵活性。用许多行和栏来创建管理程序。然后组件就填充到由管理程序规定的单元中。比如，由语句 new GridLayout(3,2)创建的有三行两栏的

Grid 布局能产生如下六个单元：



因为有 Border 布局管理器，组件相应的位置不随区域的缩放而改变。只是组件的大小改变。

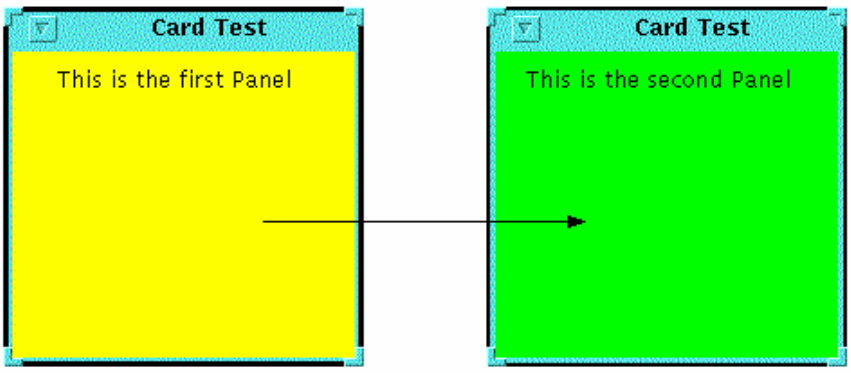
Grid 布局管理器总是忽略组件的最佳大小。所有单元的宽度是相同的，是根据单元数对可用宽 度进行平分而定的。同样地，所有单元的高度是相同的，是根据行数对可用高度进行平分而定的。 将组件添加到网格中的命令决定它们占有的单元。单元的行数是从左到右填充，就象文本一样，而页是从上到下由行填充。

```
import java.awt.BorderLayout;
import java.awt.Button;
import java.awt.Frame;
import java.awt.GridLayout;
import java.awt.Panel;
import java.awt.TextField;

public class GridLayoutDemo extends Frame {
    public GridLayoutDemo() {
        //创建第一个Panel，显示输入框
        Panel p1 = new Panel();
        p1.add(new TextField(30));
        this.add(p1,BorderLayout.NORTH);
        //创建第一个Panel，显示数字和操作符的按钮
        Panel p2 = new Panel();
        p2.setLayout(new GridLayout(3, 5, 4, 4));
        String[] name = { "0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
                "+", "-", "*", "/", "." };
        for (int i = 0; i < name.length; i++) {
            p2.add(new Button(name[i]));
        }
        this.add(p2);
    }

    public static void main(String[] args) {
        GridLayoutDemo frame = new GridLayoutDemo();
```

```
        frame.setTitle("网格布局");
        // 该代码依据放置的组件设定窗口的大小使之正好能容纳你放置的所有组件
        frame.pack();
        frame.setVisible(true);
        frame.setLocationRelativeTo(null); // 让窗体居中显示
    }
}
```

11.6.4 CardLayout(卡片布局)

Card 布局管理器能将界面看作一系列的卡，其中的一个在任何时候都可见。用 add()方法来将卡添加到 Card 布局中。Card 布局管理器的 show()方法应请求转换到一个新卡中。下例就是一个带有 5 张卡的框架。




```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class CardLayoutDemo {

    Frame f;
    Panel p1;
```

```
Panel p2;
String[] name = { "1", "2", "3", "4", "5" };
CardLayout c;

public CardLayoutDemo(){

    f = new Frame("卡片布局");
    p1 = new Panel();
    p2 = new Panel();
    c = new CardLayout();

    p1.setLayout(c);

    for (int i = 0; i < name.length; i++){
        p1.add(name[i], new Button(name[i]));
    }

    // 控制显示上一张的按钮
    Button previous = new Button("上一张");
    previous.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            c.previous(p1);
        }
    });

    // 控制显示下一张的按钮
    Button next = new Button("下一张");
    next.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            c.next(p1);
        }
    });

    // 控制显示第一张的按钮
    Button first = new Button("第一张");
    first.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            c.first(p1);
        }
    });
    // 控制显示最后一张的按钮

    Button last = new Button("最后一张");
    last.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            c.last(p1);
        }
    });
    // 根据card名显示的按钮
```

```
        Button third = new Button("第三张");
        third.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e) {
                c.show(p1, "3");
            }
        });
        p2.add(previous);
        p2.add(next);
        p2.add(first);
        p2.add(last);
        p2.add(third);

        f.add(p1);// 默认添加到中间
        f.add(p2, BorderLayout.SOUTH);
        f.pack();
        f.setVisible(true);
    }

    public static void main(String[] args){
        new CardLayoutDemo();
    }
}
```

11.6.5 GridBag 布局管理器

除了 Flow、Border、Grid 和 Card 布局管理器外，核心 Java.awt 也提供 GridBag 布局管理器。 GridBag 布局管理器在网格的基础上提供复杂的布局，但它允许单个组件在一个单元中而不是填满整个单元那样地占用它们的最佳大小。网格包布局管理器也允许单个组件扩展成不止一个单元。

12 I/O 流

12.1 File 类

File 类的对象主要用来获取文件本身的一些信息，例如文件所在的目录，文件的长度，文件读写权限等，不涉及对文件的读写操作。

创建一个 File 对象的构造方法有 3 个

- public File(String pathname)
- public File(String parentPath, String fileName)
- public File(File parentFile, String fileName)

其中，pathname 和 fileName 是文件名字，parentPath 是文件的路径，parentFile 是指定的一个目录文件。

12.1.1 文件的属性

我们在开发过程中，经常使用 File 类的下列方法获取文件本身的一些信息：

- public String getName() 获取文件的名字
- public boolean canRead() 判断文件件是否可读
- public boolean canWrite() 判断文件是否可被写入
- public boolean exits() 判断文件是否存在
- public long length() 获取文件的长度，单位是字节
- public String getAbsolutePath() 获取文件的绝对路径
- public String getParent() 获取文件的父目录
- public boolean isFile() 判断文件是否是一个目录
- public boolean isHidden() 判断文件是否是隐藏文件
- public long lastModified() 获取文件最后修改的时间，时间是毫秒数

比较两个文件是否是同一个文件：

```
File f=new File("d:/work/test/a.txt");
File f2=new File("a.txt");
// 判断是否是绝对路径
if(!f.isAbsolute()){
    f=f.getAbsoluteFile();
}
if(!f2.isAbsolute()){
    f2=f2.getAbsoluteFile();
}
if(f.equals(f2)){
    System.out.println("是同一个文件");
}
```

12.1.2 目录

12.1.2.1 创建目录

File 对象调用方法 mkdir()创建一个目录，如果创建成功返回 true，否则放回 false（如果该目录已经存在将返回 false）。

12.1.2.2 列出目录中的文件

如果 File 对象是一个目录，那么该对象可以调用下述方法列出目录下的文件和子目录：

- public String[] list() 用字符串形式返回目录下的全部文件
- public File[] listFiles() 用 File 对象形式返回目录下的全部文件

有时需要列出目录下指定类型的文件，比如.java，.txt 等扩展名的文件，可以使用 File 类的下述两个方法，列出指定类型的文件：

- public String[] list(FilenameFilter filter)
- public File listFiles(FilenameFilter filter)

FilenameFilter 是一个接口，该接口有一个方法

public boolean accept(File dir,String name)

当向 list 方法传递一个实现该接口的对象时，dir 调用 list 方法在列出文件时，将调用 accept 方法检查该文件 name 是否符合 accept 方法指定的目录和文件名字要求。

代码示例：取得一个目录下的所有 java 文件：

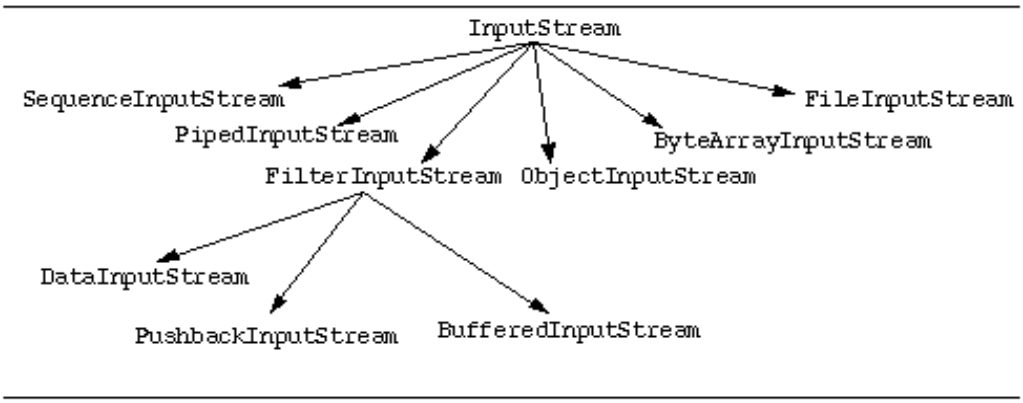
```
File f=new File("d:/work");
File[] arr=f.listFiles(new FileFilter(){
    public boolean accept(File file) {
        if(file.isFile() && file.getName().endsWith(".java")){
            return true;
        }
        return false;
    }
});
```

12.2 流的基本知识

一个流是字节的源或目的。次序是有意义的。例如，一个需要键盘输入的程序可以用流来做到这一点。

两种基本的流是：输入流和输出流。你可以从输入流读，但你不能对它写。要从输入流读取字节，必须有一个与这个流相关联的字符源。

在 java.io 包中定义了一些流类。下图表明了包中的类层次。一些更公共的类将在后面介绍。



12.3 FileInputStream 类

如果需要从本地硬盘读取文件，可以使用 FileInputStream 类，该类是从 InputStream 中派生出来的简单输入类。该类的所有方法都是从 InputStream 类继承来的。为了创建 FileInputStream 类的对象，可以调用下面两个构造方法：

- FileInputStream(String name)
- FileInputStream(File file)

第一个构造器使用给定的文件名 name 创建一个 FileInputStream 对象，第二个构造器使用 File 对象创建 FileInputStream 对象。

12.3.1 创建文件输入流

例如读取一个名为 a.txt 的文件，建立一个文件输入流对象，如下所示：

```
FileInputStream fis=new FileInputStream("a.txt");
```

或者使用 File 对象：

```
File f=new File("a.txt");
FileInputStream fis=new FileInputStream(f);
```

12.3.2 处理 IO 异常

当使用文件输入流构造器建立通往文件的输入流时，可能会出现异常。例如，试图要打开的文件可能不存在，就会出现 IO 异常。程序必须使用一个 try-catch 块检测并处理这个异常。示例如下：

```
FileInputStream fis=null;
try {
    fis=new FileInputStream("a.txt");
} catch (FileNotFoundException e) {
    //文件不存在异常
    System.out.println("File read error:"+e);
}
```

由于 IO 操作对于错误特别敏感，所以许多其他的流类构造器和方法也抛出 IO 异常，你都可以按照上述程序段的方式捕获处理这些异常。

12.3.3 从输入流中读取字节

输入流的唯一目的是提供通往数据的通道，程序可以通过这个通道读取数据。Read 方法给程序提供一个从输入流中读取数据的基本方法。

```
int  read( )
```

从输入流中读取数据的下一个字节。返回 0 到 255 范围内的 int 字节值。如果因为已经到达流末尾而没有可用的字节，则返回值 -1。

```
int  read(byte[] b)
```

从输入流中读取一定数量的字节，并将其存储在缓冲区数组 b 中。以整数形式返回实际读取的字节数。如果遇到输入流的结尾，则返回-1。

```
int  read(byte[] b,int off , int len )
```

这三个方法提供对输入管道数据的存取。简单读方法返回一个 int 值，它包含从流里读出的一个字节或者-1，其中后者表明文件结束。其它两种方法将数据读入到字节数组中，并返回所

读的字节数。第三个方法中的两个 int 参数指定了所要填入的数组的子范围。

FileInputStream 流顺序地读取文件，只要不关闭流，每次调用 read 方法就顺序地读取源其余的内容，直到流的末尾或流被关闭。

12.3.4 关闭流

虽然 Java 在程序结束时自动关闭所有打开的流，但是当我们使用完流后，显式地关闭任何打开的流仍是一个良好的习惯。一个被打开的流可能会用尽系统资源，这取决于平台和实现。如果没有关闭那些被打开的流，那么在另一个程序可能无法得到这些资源。关闭输出流的另一个原因是把该缓冲区的内容刷出，也就是把缓冲区的内容写到目的地。

```
FileInputStream fis = null;
try {
    File f = new File("a.txt");
    // 创建字节流
    fis = new FileInputStream(f);
    // 根据文件的大小，创建相同长度的字节数组
    byte[] arr = new byte[(int) f.length()];
    // 读取流中的数据到字节数组中
    fis.read(arr);
    // 将读取的数据转换成String
    String msg = new String(arr);

    System.out.println(msg);
} catch (IOException e) {
    e.printStackTrace();
} finally {
    // 判断字节流是否为空
    if (fis != null) {
        try {
            // 关闭字节流
            fis.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

12.4 FileOutputStream 类

与 FileInputStream 类相对应的类是 FileOutputStream 类。FileOutputStream 提供了基本的文件写入能力。FileOutputStream 类有两个构造方法：

- FileOutputStream(String name)

- FileOutputStream(File file)

FileOutputStream 类有三个基本的 write()方法“

- void write(int n)
- void write(byte b[])
- void write(byte b[], int off, int len)

这些方法写输出流。和输入一样，总是尝试以实际最大的块进行写操作。

void close()

当你完成写操作后，就关闭输出流。如果你有一个流所组成的栈，就关闭栈顶部的流。 这个关闭操作会关闭其余的流。

void flush()

有时一个输出流在积累了若干次之后才进行真正的写操作。flush()方法允许你强制执行写操作。

12.5 对象字节流

12.5.1 ObjectInputStream 和 ObjectOutputStream

ObjectInputStream 类和 ObjectOutputStream 类分别是 InputStream 类和 OutputStream 类的子类。ObjectInputStream 类和 ObjectOutputStream 类创建的对象被称为对象输入流和对象输出流。

对象输出流使用 writeObject(Object obj)方法将一个对象 obj 写入到一个文件，对象输入流使用 readObject()读取一个对象到程序中。

ObjectInputStream 类和 ObjectOutputStream 类的构造方法分别是：

- ObjectInputStream(InputStream in)
- ObjectOutputStream(OutputStream out)

ObjectOutputStream 的指向的是一个输出流对象，因此准备将一个对象写入到文件时，首先用 FileOutputStream 创建一个文件输出流，如下所示：

```
FileOutputStream fos = null;
```

```
ObjectOutputStream oos=null;
try {
    File f = new File("javakc.txt");
    // 创建字节流
    fos = new FileOutputStream(f);
    // 创建对象流
    oos=new ObjectOutputStream(fos);
    // 将日期对象写入到对象流中
    Date d=new Date();
    oos.writeObject(d);
} catch (IOException e) {
    e.printStackTrace();
} finally {
    //关闭流
    if (oos != null) {
        try {
            oos.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

同样 ObjectInputStream 的指向应当是一个输入流对象，因此当准备从文件中读入一个对象到程序中时，首先用 FileInputStream 创建一个文件输入流，如下列代码所以：

```
FileInputStream fis = null;
ObjectInputStream ois=null;
try {
    File f = new File("javakc.txt");
    // 创建字节流
    fis = new FileInputStream(f);
    // 创建对象流
    ois=new ObjectInputStream(fis);
    // 从对象流中读取出日期对象
    Object o=ois.readObject();
    Date d=(Date)o;
    System.out.println(o);
} catch (Exception e) {
    e.printStackTrace();
} finally {
    //关闭流
    if (ois != null) {
        try {
            ois.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

```
    }
```

12.5.2 串行化

将一个对象存放到某种类型的永久存储器上称为保持。如果一个对象可以被存放到磁盘或磁带上，或者可以发送到另外一台机器并存放到存储器或磁盘上，那么这个对象就被称为可保持的。java.io.Serializable 接口没有任何方法，它只作为一个"标记者"，用来表明实现了这个接口的类可以考虑串行化。类中没有实现 Serializable 的对象不能保存或恢复它们的状态。

当一个对象被串行化时，只有对象的数据被保存；方法和构造函数不属于串行化流。如果一个数据变量是一个对象，那么这个对象的数据成员也会被串行化。树或者对象数据的结构，包括这些子对象，构成了对象图。

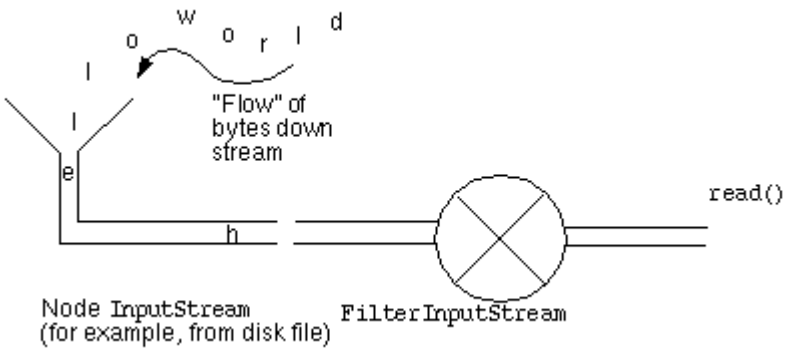
因为有些对象类所表示的数据在不断地改变 ，所以它们不会被串行化；

例 如 ， java.io.FileInputStream 、 java.io.FileOutputStream 和 java.lang.Thread 等流。如果一个可串行化对象包含对某个不可串行化元素的引用， 那么整个串行化操作就会失败，而且会抛出一个 NotSerializableException。

如果对象图包含一个不可串行化的引用，只要这个引用已经用 transient 关键字进行了标记，那么对象仍然可以被串行化。

```
public class MyClass implements Serializable {
    public transient Thread myThread;
    private String customerID;
    private int total;
}
```

12.6 缓冲字节流



BufferedInputStream 与 java.io.BufferedOutputStream 可以为 InputStream、OutputStream 类增加缓冲区功能。构建 BufferedInputStream 实例时，需要给定一个 InputStream 类型的实例，实现 BufferedInputStream 时，实际上最后是实现 InputStream 实例。同样，构建 BufferedOutputStream 时，也需要给定一个 OutputStream 实例，实现 BufferedOutputStream 时，实际上最后是实现 OutputStream 实例。

BufferedInputStream 的数据成员 buf 是一个位数组，默认为 2048 字节。当读取数据来源时，例如文件，BufferedInputStream 会尽量将 buf 填满。当使用 read()方法时，实际上是先读取 buf 中的数据，而不是直接对数据来源作读取。当 buf 中的数据不足时，BufferedInputStream 才会再实现给定的 InputStream 对象的 read()方法，从指定的装置中提取数据。

BufferedOutputStream 的数据成员 buf 也是一个位数组，默认为 512 字节。当使用 write()方法写入数据时实际上会先将数据写到 buf 中，当 buf 已满时才会实现给定的 OutputStream 对象的 write()方法，将 buf 数据写到目的地，而不是每次都对目的地作写入的动作。

下面示例中，将 javakc.jpg 复制成 temp.jpg:

```
byte[] data = new byte[1024];

File srcFile = new File("javakc.jpg");
File desFile = new File("temp.jpg");

BufferedInputStream bis = null;
BufferedOutputStream bos = null;
try {
    bis = new BufferedInputStream(new FileInputStream(srcFile));
    bos = new BufferedOutputStream(new FileOutputStream(desFile));

    System.out.println("复制文件: " + srcFile.length() + "字节");
    while (bis.read(data) != -1) {
        bos.write(data);
    }
    // 将缓冲区中的数据全部写出
    bos.flush();

} catch (Exception e) {
    e.printStackTrace();
} finally {
    // 关闭流
    try {
        if (bis != null) {
            bis.close();
        }
        if (bos != null) {
            bos.close();
        }
```

```
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    System.out.println("复制完成");
```

为了确保缓冲区中的数据一定被写出至目的地，建议最后执行 flush()将缓冲区中的数据全部写出目的流中。

12.7 数据字节流

DataInputStream 类 和 DataOutputStream 类创建的对象被称为数据输入流和数据输出流。数据流允许程序按照机器无关的风格读取操作数据，也就是当我们操作一个数据时，不必关心这个数值应当是多少个字节。

```
// 写操作
    FileOutputStream fos = null;
    DataOutputStream dos = null;
    try {
        File f = new File("javakc.txt");
        // 创建字节流
        fos = new FileOutputStream(f);
        dos = new DataOutputStream(fos);
        // 写入数据
        dos.writeInt(2013);
        dos.writeUTF("Java快车，ok");
        dos.writeFloat(100.0F);
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        // 关闭流
        if (dos != null) {
            try {
                dos.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
    // 读操作
    FileInputStream fis = null;
    DataInputStream dis = null;
    try {
        File f = new File("javakc.txt");
        // 创建字节流
        fis = new FileInputStream(f);
        dis = new DataInputStream(fis);
```

```
        // 读出数据
        int firstInt = dis.readInt();
        String str = dis.readUTF();
        float secFlt = dis.readFloat();
        System.out.println(firstInt+","+str+","+secFlt);
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        // 关闭流
        if (dis != null) {
            try {
                dis.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

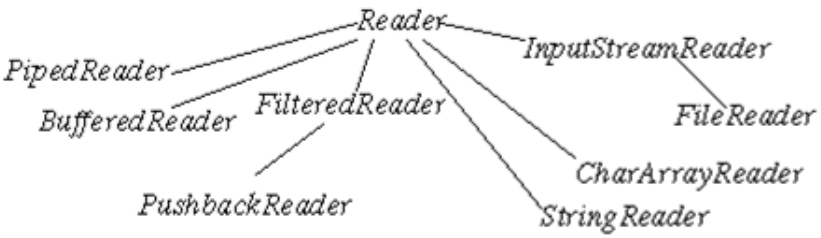
12.8 字符流

12.8.1 Reader 和 Writer

Java 技术使用 Unicode 来表示字符串和字符，而且它提供了 16 位版本的流，以便用类似的方法来处理字符。这些 16 位版本的流称为读者和作者。和流一样，它们都在 java.io 包中。 读者和作者中最重要的版本是 InputStreamReader 和 OutputStreamWriter。这些类用来作为字节流与读者和作者之间的接口。

当你构造一个 InputStreamReader 或 OutputStreamWriter 时，转换规则定义了 16 位 Unicode 和其它平台的特定表示之间的转换。

缺省情况下，如果你构造了一个连接到流的读者和作者，那么转换规则会在使用缺省平台所定义的字节编码和 Unicode 之间切换。在英语国家中，所使用的字节编码是：ISO8859-1。

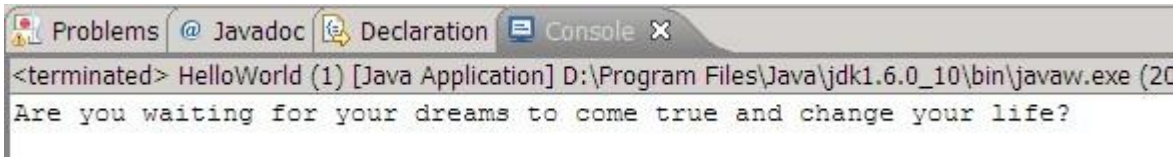


字符流在进行中文操作时会更加方便和准确，比如在读取中文文档内容时，字节流可能会读取到中文的一半，而字符流则不会出现这样的问题。

比如，我们用字节流读取大文件，需要通过字节数组，一批一批将数据读取出来，代码如下

```
FileInputStream fis = null;
try {
    File f = new File("javakc.txt");
    // 创建字节流
    fis = new FileInputStream(f);
    // 创建字节数组,每次读取 5 个字节
    byte[] arr = new byte[5];
    // 读取的字节长度
    int length=0;
    while((length=fis.read(arr))!=-1){
        // 将读取的数据转换成String
        String msg = new String(arr,0,length);
        //输出
        System.out.print(msg);
    }
} catch (IOException e) {
    // 文件不存在异常
    System.out.println("File read error:" + e);
} finally {
    if (fis != null) {
        try {
            fis.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

当读取英文文档时，读取的数据没有问题，结果如下图：

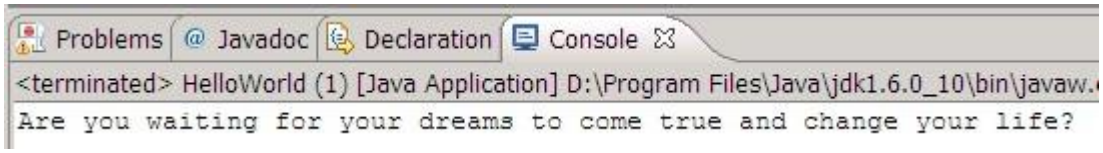


但取得中文文档时，因为将中文字符拆分，则出现了编码问题，结果如下图：

如果采用字符流读取数据，则不会出现上述问题：

```
FileInputStream fis = null;
InputStreamReader reader = null;
try {
    File f = new File("javakc.txt");
    // 创建字节流
    fis = new FileInputStream(f);
    // 创建字符流
    reader = new InputStreamReader(fis);
    // 创建字符数组,每次读取 5 个字符
    char[] arr = new char[5];
    // 读取的字符长度
    int length = 0;
    while ((length = reader.read(arr)) != -1) {
        // 将读取的数据转换成String
        String msg = new String(arr, 0, length);
        // 输出
        System.out.print(msg);
    }
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (reader != null) {
        try {
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

运行结果如下：



12.8.2 缓冲读者和作者

因为在各种格式之间进行转换和其它 I/O 操作很类似，所以在处理大块数据时效率最高。在 InputStreamReader 和 OutputStreamWriter 的结尾链接一个 BufferedReader 和

BufferedWriter 是一个好主意。记住对 BufferedWriter 使用 flush()方法。

```
FileInputStream fis = null;
InputStreamReader reader = null;
BufferedReader buf = null;
try {
    File f = new File("javakc.txt");
    // 创建字节流
    fis = new FileInputStream(f);
    // 创建字符流
    reader = new InputStreamReader(fis);
    buf = new BufferedReader(reader);
    String s = null;
    // 读取流中的数据(按行读取)
    while ((s = buf.readLine()) != null) {
        System.out.println(s);
    }
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (buf != null) {
        try {
            buf.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

12.9 随机访问文件

你经常会发现你只想读取文件的一部分数据，而不需要从头至尾读取整个文件。你可能想访问一个作为数据库的文本文件，此时你会移动到某一条记录并读取它的数据，接着移动到另一个记录，然后再 到其他记录 -- 每一条记录都位于文件的不同部分。Java 编程语言提供了一个 RandomAccessFile 类来处理这种类型的输入输出。

你可以用如下两种构造方法来打开一个随机存取文件：

- RandomAccessFile(String name, String mode);
- RandomAccessFile(File file, String mode);

mode 参数决定了你对这个文件的存取是只读(r)还是读/写(rw)。 例如，你可以打开一个打开一个数据库文件并准备更新：

```
RandomAccessFile raf;
raf = new RandomAccessFile("javakc.txt","rw");
```

RandomAccessFile 对象按照与数据输入输出对象相同的方式来读写信息。你可以访问在

DataInputStrem 和 DataOutputStream 中所有的 read()和 write()操作。

Java 编程语言提供了若干种方法，用来帮助你在文件中移动。

long getFilePointer()返回文件指针的当前位置。

void seek(long pos)设置文件指针到给定的绝对位置。这个位置是按照从文件开始的字节偏移量给出的。 位置 0 标志文件的开始。

long length()返回文件的长度。位置 length()标志文件的结束。

在文件末尾，写入字符：

```
RandomAccessFile raf = null;
try {
    File f = new File("javakc.txt");
    raf = new RandomAccessFile(f, "rw");
    raf.seek(f.length());
    raf.write("welcome to Javakc!".getBytes());
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (raf != null) {
        try {
            raf.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

12.10 学习目标

1. 理解 File 类封装的是文件的抽象路径名，并且不能操作文件的内容。
2. 熟练使用 File 类的主要方法（课堂上老师讲过的方法）。
3. 理解相对路径和绝对路径
4. File 类的 Api 中，有的返回文件路径名，有的返回 File，理解其本质是一样的。

   比如 getParent( )和 getParentFile( )，list( )和 listFiles( )
5. 理解 File 类的 Api 中，有的方法需要区分此 File 是文件还是目录。

   也就是说文件和目录要区分对待，比如 list( )，如果是文件调用此方法返回 null，如果目录调用此方法返回数组

12.11 作业

1. 递归展示目录的结构。

13 多线程

13.1 进程与线程

13.1.1 概念

几乎每种操作系统都支持进程的概念—进程就是在某种程度上相互隔离的、独立运行的程序，每一个进程都有自己独立的内存空间。比如 IE 浏览器程序，每打开一个 IE 浏览器窗口，就启动一个新的进程。在 java 中，我们执行 java.exe 程序，就启动一个独立的 Java 虚拟机进程，该进程的任务就是解析并执行 Java 程序代码。

线程是指进程中的一个执行流程，一个进程可以由多个线程组成，即一个进程中可以同时运行多个不同的线程，它们分别执行不同的任务。当进程内的多个线程同时运行时，这种运行方式成为并发运行。

线程又被称为轻量级进程，它和进程一样拥有独立的执行控制，由操作系统进行调度。

线程和进程的区别是：

- 每个进程都有独立的代码和存储空间（进程上下文），进程切换的开销大。
- 线程没有独立的存储空间，而是和所属进程中其他的线程共享代码和存储空间，但每个线程有独立的运行栈和程序计数器，因此线程切换的开销较小。
- 多进程—在操作系统中能同时运行多个任务（程序），也称多任务。
- 多线程—在同一应用程序中有多个顺序流同时执行。

许多服务器程序，如数据库服务器和 Web 服务器，都支持并发运行，这些服务器能同时响应来自不同客户的请求。

13.1.2 java 线程的运行机制

在 java 虚拟机进程中，执行程序代码的任务是由线程来完成的。每个线程都有一个独立的程序计数器和方法调用栈。

程序计数器：也称为 PC 寄存器，当线程执行一个方法时，程序计数器指向方法区中下一条要执行的字节码指令。

方法调用栈：简称方法栈，用来跟踪线程运行中一系列的方法调用过程，栈中的元素称为栈帧，每当线程调用一个方法的时候，就会向方法栈压入一个新帧。帧用来存储方法的参数、局部变量和运算过程中的临时数据。

栈帧由以下三个部分组成：

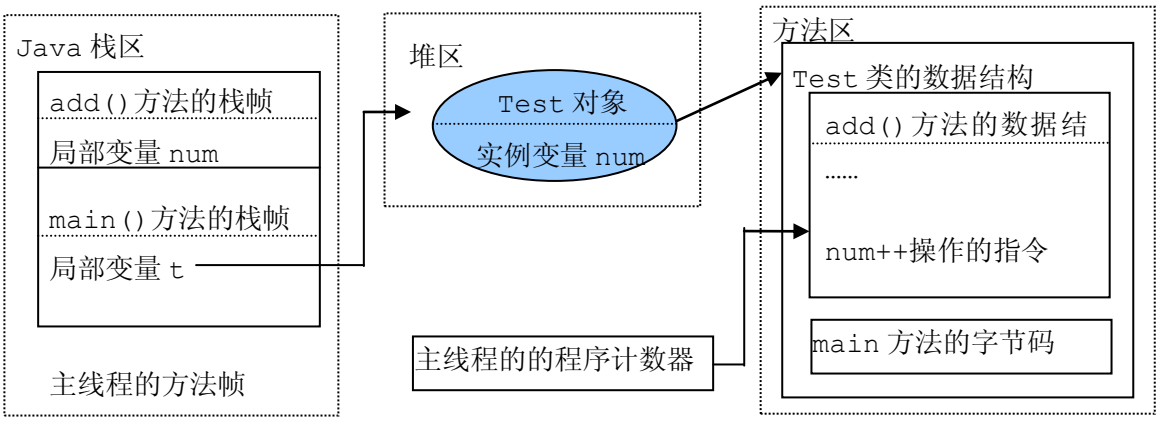
- 局部变量区：存放局部变量和方法参数。

- 操作数栈：是线程的工作区，用来存放运算过程中生成的临时数据。
- 栈数据区：为线程执行指令提供相关的信息，包括如何定位到位于堆区和方法区的特定数据，以及如何正常退出方法或者异常中断方法。

每当用 Java 命令启动一个 Java 虚拟机进程时，Java 虚拟机都会创建一个主线程，该线程从程序入口 main()方法开始执行。以下面程序为例，介绍线程的运行过程。

```
public class Test {
    private int num;          //实例变量
    public int add(){
        int b=0;             //局部变量
        num++;
        b=num;
        return b;
    }
    public static void main(String[] args) {
        Test t=new Test();    //局部变量
        int num=0;            //局部变量

        num=t.add();
        System.out.println(num);
    }
}
```

主线程从 main()方法的程序代码开始运行，当它开始执行 method()方法的“a++”操作时，运行时数据区的状态如下图所示。

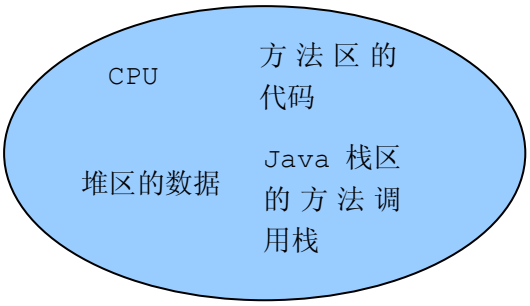


当主线程执行“a++”操作时，它能根据 method()方法的栈帧的栈数据区中的有关信息，正确地定位到堆区的 Test 对象的实例变量 num，并把它的值加 1。

当 add()方法执行完毕后，它的栈帧就会从方法栈中弹出，它的局部变量 b 结束生命周期。main()方法的栈帧就成为当前帧，主线程继续执行 main()方法。

方法区存放了线程所执行的字节码指令，堆区存放了线程所操作的数据(以对象的形式存放)，Java 栈区则是线程的工作区，保存线程的运行状态。

另外，计算机机器指令的真正执行者是 CPU，线程必须获得 CPU 的使用权，才能执行一条指令。下图中显示了线程运行中需要使用的计算机 CPU 和内存资源。



在 Java 的方法中，可以通过 Thread 类的 currentThread 方法得到正在调用该方法的线程。

```
public class Test {
    public static void main(String[] args) {
        Thread t=Thread.currentThread();
        System.out.println("调用main方法的线程名字是："+t.getName());
    }
}
```

虚拟机为调用 main 方法的线程命名为“main”。

13.2 线程的创建和启动

Java 在代码中对线程进行了支持，程序员可以创建自己的线程，它将和主线程并发运行。创建线程有两种方式：

扩展 Thread 类

实现 Runnable 接口

13.2.1 扩展 Thread 类

Thread 类代表线程类，它的最主要的两个方法是：

- run() 包含线程运行时所执行的代码。
- start() 用于启动线程

开发线程类只需要继承 Thread 类，覆盖 Thread 类的 run()方法即可。在 Thread 类中，run()方法的定义如下：

```
public void run()
```

该方法没有声明抛出任何异常，根据方法覆盖的规则，Thread 子类的 run()方法也不能声明抛出任何异常。示例如下：

```
public class MyThread extends Thread {
    public void run(){
        //线程体内的实现
        for(int i=0;i<100;i++){
            System.out.println(i);
        }
    }

    public static void main(String[] args) {
        MyThread mt=new MyThread();
        mt.start(); //启动MyThread线程
    }
}
```

当执行“java MyThread“命令时，Java 虚拟机首先创建并启动主线程。主线程的任务是执行 main 方法，main 方法创建了一个 MyThread 对象，然后调用它的 start()方法启动 MyThread 线程。MyThread 线程的任务是执行它的 run()方法。

主线程和自定义线程并发运行

在下面的例子中，main 方法创建并启动两个 MyThread 线程：

```
public class MyThread extends Thread {
    public void run(){
        //线程体内的实现
        for(int i=0;i<100;i++){
            //打印线程的名字
            System.out.println(this.getName()+":"+i);
        }
    }
    public static void main(String[] args) {
        MyThread mt=new MyThread();
        MyThread mt2=new MyThread();
        mt.start(); //启动MyThread线程
        mt2.start(); //启动MyThread线程
    }
}
```

执行结果：

```
Thread-1:7
Thread-1:8
Thread-1:9
Thread-1:10
Thread-1:11
Thread-0:1
Thread-1:12
Thread-1:13
Thread-1:14
Thread-0:2
Thread-0:3
```

```
    Thread-0:4
```

当主线程执行 main()方法时，会创建两个 MyThread 对象，然后启动两个 MyThread 线程。在 Java 虚拟机钟有两个线程并发执行 MyThread 对象的 run()方法。在两个线程各自的方法栈中都有代表 run()方法的栈帧，在这个帧中存放了局部变量 num，也就是每个线程都拥有自己的局部变量 num，它们都分别从 0 增加到 100。

因为 Thread 类中有 getName()方法，MyThread 类继承了 Thread 类，所以在代码中可以使用 this.getName()得到当前线程的名字。mt 对象启动线程的名字是 Thread-0，mt2 对象启动线程的名字是 Thread-1。

从运行结果上，我们可以看到两个线程交替运行，两个线程轮流得到 CPU 的运行时间片。

sleep 方法

Thread类中有一个sleep方法，在指定的毫秒数内让当前正在执行的线程休眠(暂停执行)，就是线程睡眠一定的时间，也就是交出 CPU 时间片，根据参数来决定暂停时间长度,让给等待序列中的下一个线程。Sleep 方法抛出 InterruptedException。

```
public class MyThread extends Thread {
    public void run(){
        for(int i=0;i<100;i++){
           System.out.println(this.getName()+":"+i);
            try {
                //让当前线程休眠 100 毫秒
                sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    public static void main(String[] args) {
        MyThread mt=new MyThread();
        MyThread mt2=new MyThread();
        mt.start();
        mt2.start();
    }
}
```

运行结果：

```
Thread-0:0
Thread-1:0
Thread-1:1
Thread-0:1
Thread-1:2
Thread-0:2
Thread-1:3
Thread-0:3
```

```
Thread-1:4
Thread-0:4
Thread-1:5
```

当 Thread-0 线程执行打印后，休眠 100 毫秒，也就失去了 CPU 的时间片，Thread-1 线程就得到了 CPU 的时间片，执行了打印操作，也休眠 100 毫秒，100 毫秒后，Thread-0 线程先恢复到可运行状态，接着运行，这样两个线程交替运行。

不要随便覆盖 Thread 类的 start()方法

创建一个线程对象后，线程并不自动开始运行，必须调用它的 start()方法才能启动线程。JDK 为 Thread 类的 start()方法提供了默认的实现，启动线程后调用 run()方法。

如果不通过 start()方法启动线程，而是直接调用 run()方法，那只是普通的方法调用，并不能启动线程。

看看如下代码：

```
public class MyThread extends Thread {
    public void run(){
        for(int i=0;i<100;i++){
            try {
                sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(this.getName()+":"+i);
        }
    }
    public void start(){
        run();
    }
    public static void main(String[] args) {
        MyThread mt=new MyThread();
        MyThread mt2=new MyThread();
        mt.start();
        mt2.start();
    }
}
```

运行结果：

```
Thread-0:0
Thread-0:1
Thread-0:2
……
Thread-0:99
Thread-1:0
Thread-1:1
Thread-1:3
Thread-1:4
```

当主线程 main 执行 start()方法时，start()方法并没有启动 MyThread 线程，而是直

接调用了 run()方法，这只是普通的方法调用。第一个 run()方法执行完成后，才开始调用第二个 run()方法，并没有出现两个线程并发运行的情况。

所以，在 Thread 子类中不要随意覆盖 start()方法，假如一定要覆盖 start()方法，那么应该先调用 super.start()方法。

```
public class MyThread extends Thread {
    public static int count=0;
    public void run(){
        for(int i=0;i<100;i++){
            try {
                sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    public void start(){
        super.start();
        System.out.println("第"+(++count)+"线程启动了");
    }
}
```

一个线程只能被启动一次

一个线程只能被启动一次，以下代码视图两次启动 MyThread 线程。

```
    MyThread mt=new MyThread();
    mt.start();
    mt.start(); //抛出IllegalThreadStateException异常
```

第二次调用 mt.start()方法时，会抛出 java.lang.IllegalThreadStateException 异常。

13.2.2 实现 Runnable 接口

Java 类不允许一个类继承多个类，因此一旦一个类继承了 Thread 类，就不能再继承其他的类，为了解决这一问题，Java 提供了 java.lang.Runnable 接口，它有一个 run()方法，定义如下：

```
    public void run()
```

示例中 MyThread 类实现了 Runnable 接口，run()方法表示线程所执行的代码。

```
public class MyThread implements Runnable {
    int count=0;
    public void run(){
        while(true){

System.out.println(Thread.currentThread().getName()+":"+count++);
         if(count>10){
```

```
            break;//当count大于 10 的时候，循环结束
          }
          try {
            Thread.sleep(100);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
        }
    }
    public static void main(String[] args) {
        MyThread mt=new MyThread();
        Thread t1=new Thread(mt);
        Thread t2=new Thread(mt);
        t1.start();
        t2.start();
    }
}
```

在 Thread 类中定义了如下形式的构造方法：

```
public Thread(Runnable runnable)
```

当线程启动时，将执行参数 runnable 所引用对象的 run()方法。其实 Thread 类也实现了 Runnable 接口。

在示例中，主线程创建了 t1 和 t2 两个线程对象。启动 t1 和 t2 线程将执行 MyThread 对象的 run()方法。t1 和 t2 共享同一个 MyThread 对象，在执行 run()方法时将操作同一个实例变量 count。打印结果如下：

```
Thread-0:0
Thread-1:1
Thread-0:2
……
Thread-1:9
Thread-0:10
Thread-1:11
```

将 main()做如下修改：

```
public static void main(String[] args) {
    MyThread mt1=new MyThread();
    MyThread mt2=new MyThread();
    Thread t1=new Thread(mt1);
    Thread t2=new Thread(mt2);
    t1.start();
    t2.start();
}
```

t1 和 t2 线程启动后，将分别执行 mt1 和 mt2 变量所引用的 MyThread 对象的 run()方法，因此 t1 和 t2 线程操作不同的 MyThread 对象的实例变量 count。运行结果如下：

```
Thread-0:0
Thread-1:0
```

```
Thread-0:1
Thread-1:1
Thread-0:2
……
Thread-0:8
Thread-1:9
Thread-0:9
Thread-0:10
Thread-1:10
```

13.3 线程的状态转换

13.3.1 新建状态(New)

用 new 语句创建的线程对象处于新建状态，此时和其他 Java 对象一样，仅仅在堆区中被分配了内存。

13.3.2 就绪状态(Runnable)

当一个线程对象创建后，其他线程调用它的 start()方法，该线程就进入了就绪状态，Java 虚拟机会为它创建方法调用栈和程序计数器。处于这个状态的线程位于可运行池中，等待获得 CPU 的使用权。

13.3.3 运行状态(Running)

处于这个状态的线程占用 CPU，执行程序代码。在并发运行环境中，如果计算机只有一个 CPU，那么任何时刻只会有一个线程处于这个状态。如果计算机有多个 CPU，那么同一时刻可以让几个线程占用不同的 CPU，使它们都处于运行状态。只有处于就绪状态的线程才有机会转到运行状态。

13.3.4 阻塞状态(Blocked)

阻塞状态是指线程因为某些原因放弃 CPU，暂时停止运行。当线程处于阻塞状态时，Java 虚拟机不会给线程分配 CPU，直到线程重新进入就绪状态，才有机会转到运行状态。

阻塞状态可以分为以下 3 种：

位于对象等待池中的阻塞状态：当线程处于运行状态时，如果执行了 wait()方法，Java 虚拟机就会把线程放到等待池中。

位于对象锁池中的阻塞状态：当线程处于运行状态时，试图获得某个对象的同步锁时，如果

该对象的同步锁已经被其他线程占用，Java 虚拟机就会把这个线程放到锁池中。

其他的阻塞状态：当前线程执行了 sleep()方法，或者调用了其他线程的 join()方法，或者发出了 I/O 请求时，就会进入这个状态。

当一个线程执行 System.out.println()或者 System.in.read()方法时，就会发出一个 I/O 请求，该线程放弃 CPU，进入阻塞状态，知道 I/O 处理完毕，该线程才会恢复运行。

```
import java.io.IOException;
class MyThread extends Thread {
    int count = 0;
    public void run() {
        while (true) {
            System.out.println(getName() + ":" + count++);
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
class YourThread extends Thread {
    public void run() {
        while (true) {
            try {
              System.out.println(" 等待用户输入");
                int data = System.in.read();
                System.out.println(getName() + " 用户输入了:"+ data);
            }catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
class Client {
    public static void main(String[] arr) {
        MyThread mt=new MyThread();
        YourThread yt=new YourThread();
        mt.start();
        yt.start();
    }
}
```

运行结果：

```
Thread-0:0
等待用户输入
Thread-0:1
```

```
Thread-0:2
a
Thread-1 用户输入了:97
等待用户输入
Thread-1 用户输入了:13
等待用户输入
Thread-1 用户输入了:10
等待用户输入
Thread-0:3
Thread-0:4
```

在上面的示例中，主线程 main 启动了 mt 和 yt 两个线程。

mt 线程启动后，运行了一次，进入休眠的阻塞状态。

yt 线程得到时间片，开始运行输出了“等待用户输入”，之后进入了 I/O 请求的阻塞状态，等待用户的输入。

mt 线程又得到时间片，执行后又休眠，这样重复了两次。

用户输入了 a 和回车，yt 由阻塞状态变为运行状态，连续打印出用户输入的值后，又进入 I/O 请求的阻塞状态。

mt 线程又得到时间片，继续运行。

13.3.5 死亡状态(Dead)

当线程退出 run(方法)时，就进入死亡状态，表示该线程结束生命周期。线程有可能是正常执行完 run()方法而退出的，也有可能是遇到异常而退出。不管线程正常结束还是异常结束，都不会对其他线程造成影响。

在下面的示例中，MyThread 线程在运行时因为抛出 RuntimeException 异常而结束，此时主线程 main 正常运行。

```
class MyThread extends Thread {
    int count = 0;
    public void run() {
        while (true) {
            System.out.println(getName() + ":" + count++);
            try {
                Thread.sleep(3000);
                if(count==3){
                    throw new RuntimeException();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

```
        }
    }
}
class Client {
    public static void main(String[] arr) throws InterruptedException {
        MyThread mt=new MyThread();
        mt.start();
        while(true){
            System.out.println(Thread.currentThread().getName()+":is
alive====="+mt.isAlive());
            Thread.sleep(3000);
        }
    }
}
```

Thread类的isAlive()方法能判断一个线程是否活着，当线程处于死亡状态活着新建状态时，该方法返回 false，在其余状态下，该方法返回 true。运行结果：

```
Thread-0:0
main:is alive====true
main:is alive====true
Thread-0:1
main:is alive====true
Thread-0:2
main:is alive====true
Exception in thread "Thread-0" java.lang.RuntimeException
    at MyThread.run(Client.java:13)
main:is alive====false
main:is alive====false
```

13.4 线程的调度

计算机通常只有一个 CPU,在任何时刻只能执行一条机器指令，每个线程只有获得 CPU 的使用权才能执行指令。所谓多线程的并发，其实是指宏观上看，各个线程轮流获得 CPU 的使用权，分别执行各自的任务。在可运行池中，会有多个处于就绪状态的线程等待 CPU，Java 虚拟机的一项任务就是负责线程的调度。线程的调度是指按照特定的机制为多个线程分配 CPU 的使用权。有两种调度模型：**分时调度模型**和**抢占式调度模型**。

分时调度模型是指让所有线程轮流获得CPU的使用权，并且平均分配每个线程占用CPU的时间片。

Java 虚拟机采用抢占式调度模型，是指优先让可运行池中优先级高的线程占用 CPU，如果可运行池中线程的优先级相同，那么就随机选择一个线程，使其占用 CPU。处于运行状态的线程会一直运行，直至它不得不放弃 CPU。一个线程会因为一下原因而放弃 CPU：

Java 虚拟机让当前线程暂时放弃 CPU，转到就绪状态，使其他的线程获得运行机会。

当前线程因为某些原因而进入阻塞状态。

线程运行结束。

值得注意的是，线程的调度不是跨平台的，它不仅取决于 Java 虚拟机，还依赖操作系统。在某些操作系统中，只要运行中的线程没有遇到阻塞，也会在运行一段时间后放弃 CPU，给其他线程运行的机会。

在 java 中，同时启动多个线程后，不能保证各个线程轮流获得均等的 CPU 时间片，从之前的例子中，大家可以体会到这一点。一个线程运行机毫秒后，就放弃的 CPU 时间片，另一个线程就得到了 CPU 时间片，各个线程交替运行。

以下程序可以证明这一点：

```
class MyThread extends Thread {
    int count = 0;
    public void run() {
        while (true) {
            System.out.print(getName() + ":" + count++ +",");
        }
    }
}
class Client {
    public static void main(String[] arr){
        MyThread mt=new MyThread();
        MyThread mt2=new MyThread();
        mt.start();
        mt2.start();
    }
}
```

两个线程抢占时间片的结果如下：

```
Thread-0:0,Thread-0:1,Thread-0:2,……,Thread-0:7,Thread-0:8,Thread
-0:9,
Thread-1:0,
Thread-0:10,Thread-1:1,Thread-0:11,Thread-0:12,Thread-0:13,Thread-1:2,
Thread-0:14,
Thread-1:3,Thread-0:15,Thread-1:4,Thread-0:16,Thread-1:5,
Thread-0:17,Thread-0:18,Thread-0:19,Thread-0:20,Thread-0:21,Thread-0:2
2,Thread-0:23,
Thread-1:6,Thread-1:7,Thread-1:8,Thread-1:9,Thread-1:10,
Thread-0:24,Thread-0:25,Thread-0:26,Thread-0:27,
Thread-1:11,Thread-1:12,Thread-1:13,……,Thread-1:16,Thread-1:17,Thr
ead-0:28,Thread-1:18,
```

如果通过代码，希望明确地让一个线程给另外一个线程运行的机会，可以采用以下方法之一：

- 调整各个线程的优先级。
- 让处于运行状态的线程调用 Thread.sleep()方法。
- 让处于运行状态的线程调用 Thread.yield()方法。

- 让处于运行状态的线程调用另一个线程的 join()方法。

13.4.1 调整各个线程的优先级

所有处于就绪状态的线程根据优先级存放在可运行池中，优先级低的线程获得较少的运行机会，优先级高的线程获得较多的运行机会。Thread 类的 setPriority(int)和 getPriority()方法分别用来设置优先级和读取优先级。优先级用整数表示，取值范围是 1~10，Thread 类有以下 3 个静态常量。

MAX_PRIORITY：取值为 10，表示最高优先级。

MIN_PRIORITY：取值为 1，表示最低优先级。

NORM_PRIORITY：取值为 5，表示默认的优先级。

修改之前的代码，分别设置两个线程的优先级。

```
class MyThread extends Thread {
    int count = 0;
    public void run() {
        while (true) {
            System.out.print(getName() + ":" + count++ +",");
        }
    }
}
class Client {
    public static void main(String[] arr){
        MyThread mt=new MyThread();
        MyThread mt2=new MyThread();
        mt.setPriority(Thread.MAX_PRIORITY);//设置最高的优先级
        mt2.setPriority(Thread.MIN_PRIORITY);//设置最低的优先级
        mt.start();
        mt2.start();
    }
}
```

由于mt(Thread-0)线程的优先级高于mt2(Thread-1)线程的优先级，因此前者优先获得 CPU 的使用权，运行结果如下：

```
Thread-0:0,Thread-1:0,Thread-0:1,
Thread-1:1,
Thread-0:2,
Thread-1:2,
Thread-0:3,Thread-0:4,Thread-0:5,Thread-0:6,Thread-0:7,Thread-0:8,Thread-0:9,
……
Thread-0:305,Thread-0:306,Thread-0:307,Thread-0:308,Thread-0:309,Thread-0:310,
```

```
Thread-1:3,
Thread-0:311,
Thread-1:4,
Thread-0:312,Thread-0:313,Thread-0:314,Thread-0:315,Thread-0:316,Thread-0:317,
……
Thread-0:995,Thread-0:996,Thread-0:997,Thread-0:998,Thread-0:999,Thread-0:1000,
Thread-1:21,Thread-1:22,Thread-1:23,Thread-1:24,Thread-1:25,Thread-1:26,
……
Thread-1:996,Thread-1:997,Thread-1:998,Thread-1:999,Thread-1:1000,
```

在两个线程同时运行的时候，每当 mt2(Thread-1)线程获得时间片后，往往只运行一次，就放弃了 CPU 的时间片，所以 mt(Thread-0)线程最先运行完毕。

如果不设置线程的优先级，线程默认的优先级为 5。

值得注意的是，尽管 Java 提供了 10 个优先级，但它与多数操作系统都不能很好地映射。比如 Windows2000 有 7 个优先级，并且不是固定的，而 Sun 公司的 Solaris 操作系统有 2 的 31 次方个优先级。如果希望程序能移植到各个操作系统中，应该确保在设置线程的优先级时，只使用 MAX_PRIORITY、MIN_PRIORITY、NORM_PRIORITY 这 3 个优先级。这样才能保证在不同的操作系统中，对同样优先级的线程采用同样的调度方式。

13.4.2 线程睡眠：Thread.sleep()方法

当一个线程在运行过程中执行了 sleep()方法时，它就会放弃 CPU，转到阻塞状态。下面示例中每执行一次循环，就睡眠 1000 毫秒。

```
class MyThread extends Thread {
    int count = 0;
    public void run() {
        while (true) {
            System.out.print(getName() + ":" + count++);
            try {
                sleep(1000);//睡眠 1 秒钟
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
class Client {
    public static void main(String[] arr) {
        MyThread mt=new MyThread();
        mt.start();
    }
}
```

Thread 类的 sleep(long millis)方法是静态的，millis 参数设定睡眠的时间，以毫秒为单位。当执行 sleep()方法时，就会放弃 CPU 开始睡眠，1 秒钟后线程结束睡眠，就会获得 CPU，继续进行下一次循环。所以会感觉程序运行很慢。

值得注意的是，当某线程结束睡眠后，首先转到就绪状态，假如其他的线程正在占用 CPU，那么该线程就在可运行池中等待获得 CPU。

线程在睡眠中如果被中断，就会收到一个 InterrupedException 异常，线程就会跳到异常处理代码块。把 InterrupedException 异常包装成一个 RuntionException，然后继续将它抛出。

在下面的示例中，主线程调用 interrupt()方法中断了睡眠中的 mt 线程，mt 线程就抛出了 InterrupedException 异常。

```
class MyThread extends Thread {
    int count = 0;
    public void run() {
        while (true) {
            System.out.println(getName() + ":" + count++);
            try {
                sleep(5000);//休眠 5 秒钟
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
class Client {
    public static void main(String[] arr) throws InterruptedException {
        MyThread mt=new MyThread();
        mt.start();
        Thread.sleep(1000);
        mt.interrupt();//中断mt线程的睡眠
    }
}
```

运行结果如下：

```
Thread-0:0
Thread-0:1java.lang.InterruptedException: sleep interrupted
        at java.lang.Thread.sleep(Native Method)
    at MyThread.run(Client.java:10)
Thread-0:2
Thread-0:3
Thread-0:4
```

可以看到主线程 main 欲中断正在睡眠中的 mt 线程，因为线程 mt 正在睡眠，所以中断失败，报出错误，线程 mt 醒来后，还可以继续运行。

interrupt()方法对于正在运行中的线程（不是睡眠中的线程）是不起作用的，只有对阻塞

中的线程有效。

13.4.3 线程让步：Thread.yield()方法

当线程在运行中执行了 Thread 类的 yield()静态方法，如果此时具有相同优先级的其他线程处于就绪状态，那么 yield()方法将把当前运行的线程放到可运行池中并使另一个线程运行。如果没有相同优先级的可运行线程，则 yield()方法什么也不做。

下面代码中，线程执行完一次循环后，就执行了 yield()方法。

```
class MyThread extends Thread {
    int count = 0;
    public void run() {
        while (true) {
            System.out.println(getName() + ":" + count++);
            yield();
        }
    }
}
class Client {
    public static void main(String[] arr) throws InterruptedException {
        MyThread mt=new MyThread();
        MyThread mt2=new MyThread();
        mt.start();
        mt2.start();
    }
}
```

从运行结果中，我们可以看到两个线程交替运行。

```
Thread-0:0
Thread-1:0
Thread-0:1
Thread-1:1
Thread-0:2
Thread-1:2
Thread-0:3
Thread-1:3
```

sleep()方法和 yield()方法都是 Thread 类的静态方法，都会使当前处于运行状态的线程放弃 CPU，把运行机会让给其他线程。两者的区别在于：

sleep()方法会给其他线程运行机会，而不考虑其他线程的优先级，因此会给较低优先级线程一个机会；yield()方法只会给相同优先级或者更高优先级线程一个运行的机会。

当线程执行了 sleep(long millis)方法后，会转到阻塞状态，参数 millis 指定睡眠的时间；当线程执行了 yield()方法后，将转到就绪状态。

sleep()方法方法抛出 InterrupedException 异常，而 yield()方法没有声明抛出任何异常。

sleep()方法比 yield()方法具有更好的可移植性。不能依靠 yield()方法来提高程序的并发性能。对于大多数程序员来说，yield()方法的唯一用途是在测试期间人为地提高程序的并发性能，以帮助发现一些隐藏的错误。

13.4.4 等待其他线程结束：join()

当前运行的线程可以调用另一个线程的 join()方法，当前运行的线程将转到阻塞状态，直至另一个线程运行结束，它才恢复运行。

```
class MyThread extends Thread {
    int count = 0;
    public void run() {
        while (count<5) {
            System.out.println(getName() + ":" + count++);
        }
    }
}
class Client {
    public static void main(String[] arr) throws InterruptedException {
        MyThread mt=new MyThread();
        mt.setName("mt");
        mt.start();
        mt.join();//主线程等待mt线程的运行结束
        System.out.println("main end");
    }
}
```

主线程调用了 mt 线程的 join()方法，主线程将等到 mt 线程运行结束后，才能恢复运行。

```
mt:0
mt:1
mt:2
mt:3
mt:4
main end
```

join()方法有两种重载形式：

```
public void join()
public void join(long timeout)
```

timeout 参数设定当前线程被阻塞的时间，以毫秒为单位。如果把示例 main()方法中的 mt.join()改为 mt.join(10)，那么当主线程被阻塞的时间超过了 10 毫秒，或者 mt 线程运行结束时，主线程就会恢复运行。

13.5 获得当前线程对象的引用

Thread 类的 currentThread()静态方法返回当前线程对象的引用。在主线程 main 中执行 currentThread()方法时，返回主线程对象的引用。

```
class Client {
    public static void main(String[] arr) {
        Thread main=Thread.currentThread();
        System.out.println(main.getName());
    }
}
```

13.6 后台线程

演员在前台演戏，许多工作人员在后台为演员提供服务，例如灯光、音效，当演出结束后，后台的服务也就停止了。有一点需要注意一下，是演员演戏先停止，后台服务再停止。

后台线程是指为其他线程提供服务的线程，也成为守护线程。如果说演员是前台线程，那么其他工作人员就是后台线程。

Java 虚拟机的垃圾回收线程就是典型的后台线程，它负责回收其他线程不再使用的内存。

后台线程的特点是：后台线程和前台线程相伴相随，只有前台线程都结束生命周期，后台线程才会结束生命周期。只要有一个前台线程还没有结束运行，后台线程就不会结束生命周期。

主线程在默认情况下是前台线程，由前台线程创建的线程在默认情况下也是前台线程。调用 Thread 类的 setDaemon(true)方法，就能把一个线程设置为后台线程。Thread 类的 isDaemon()方法用来判读一个线程是否是后台线程。

在下面的例子中没有设置后台线程，当 main 线程结束后，mt 线程继续运行。

```
class MyThread extends Thread {
    int count = 0;
    public void run() {
        while (true) {
            try {
                sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(getName() + ":" + count++);
        }
    }
}
class Client {
    public static void main(String[] arr) throws InterruptedException {
        MyThread mt=new MyThread();
```

```
        mt.setName("mt");
        mt.start();
        Thread.sleep(5000);
        System.out.println("main end");
    }
}
```

运行结果如下：

```
mt:0
mt:1
mt:2
mt:3
main end
mt:4
mt:5
.....
```

如果设置 mt 线程为后台线程：

```
public static void main(String[] arr) throws InterruptedException {
    MyThread mt=new MyThread();
    mt.setName("mt");
    mt.setDaemon(true);//设置mt线程为后台线程
    mt.start();
    Thread.sleep(5000);
    System.out.println("main end");
}
```

运行结果如下：

```
mt:0
mt:1
mt:2
mt:3
mt:4
main end
```

尽管后台线程的 run()方法执行的是无限循环，只要当 main 线程睡眠 5 秒钟结束后，mt 线程也就结束了。

在使用后台线程时，有以下注意点：

Java 虚拟机保证：当所有前台线程运行结束后，再终止后台线程，体现的是先后顺序。那么前台线程运行结束后，后台线程是一次也不运行吗？这取决于程序的实现。

只有线程启动前（即调用 start()方法以前），才能把它设置为后台线程。如果线程启动后，再调用这个线程的 setDaemon()方法，就会导致 IllegalThreadStateException 异常。

由于前台线程创建的线程在默认情况下仍然是前台线程，由后台线程创建的线程在默认情况下仍然是后台线程。

13.7 定时器 Timer

在 JDK 的 java.util 包中提供了一个实用类 Timer，它能够定时执行特定的任务。TimerTask 类表示定时器执行的一项任务。

```
class Client {
    public static void main(String[] arr) throws InterruptedException {
        Timer timer=new Timer();
        TimerTask task=new TimerTask(){
            public void run(){
                System.out.println("time is: "+new Date());
            }
        };
        timer.schedule(task,10,5000);//设置定时任务
    }
}
```

java.util.TimerTask 类是一个抽象类，它实现了 Runnable 接口。run()方法表示定时器需要完成的任务。

Timer 类的 schedule(TimerTask task,long delay,long period)方法用来设置定时器需要执行的任务。task 参数表示任务；delay 参数表示延迟执行的时间，以毫秒为单位；period 参数表示每次执行任务的间隔时间，以毫秒为单位。例如：

```
timer.schedule(task,10,5000);
```

以上代码表示定时器将在 10 毫秒后开始执行 task 任务，以后每隔 5000 毫秒重复执行一次 task 任务。

同一个定时器对象可以执行多个定时任务，例如：

```
timer.schedule(task1,0,5000);
timer.schedule(task2,0,3000);
```

以上代码表示定时器会执行两个任务，第一个任务每隔 5 秒执行一次，第二个任务每隔 3 秒执行一个。

13.8 线程的同步

线程的职责就是执行一些操作，而多数操作都涉及到处理数据。下面的线程的操作主要是处理实例变量 count。通过四个线程模拟卖票系统。

```
class TicketSell implements Runnable{
    //有 100 张票
    int count=100;
    public void run(){
        while(true){
            if(count>0){
                try{
                    Thread.sleep(10);
                }catch(Exception e){
```

```
                }
                System.out.println(Thread.currentThread().getName()
                                                    +"="+count);
                count--;
            }else{
                break;
            }
        }
    }
}
class Client {
    public static void main(String[] arr){
        TicketSell ts=new TicketSell();
        Thread t1=new Thread(ts);
        Thread t2=new Thread(ts);
        Thread t3=new Thread(ts);
        Thread t4=new Thread(ts);
        //启动四个线程卖票
        t1.start();
        t2.start();
        t3.start();
        t4.start();
    }
}
```

运行结果如下：

```
Thread-1=100
Thread-0=99
Thread-3=98
……
Thread-0=3
Thread-3=2
Thread-2=1
Thread-1=0
Thread-3=-1
Thread-0=-1
```

票的数量 count 是一个共享资源，四个线程对共享资源开始了竞争。当线程 Thread-2 卖掉最后一张票后，其他的三个线程，还认为有最后一张票，继续卖票，造成了共享资源的并发问题。

以上卖票的操作被称为原子操作，一个线程在执行原子操作期间，应该采取措施使得其他线程不能操作共享资源，否则就会出现，共享资源被重复操作的问题。

13.8.1 同步代码块

为了保证每个线程能正常执行原子操作，Java 引入了同步机制，具体做法是在代表原子操作的程序代码前加上 synchronized 标记，这样的代码被称为同步代码块。

```
class TicketSell implements Runnable{
    //有 100 张票
    int count=100;
    //同步锁
    Object o=new Object();
    public void run(){
        while(true){
            synchronized(o){
                if(count>0){
                    try{
                        Thread.sleep(10);
                    }catch(Exception e){
                    }
                    System.out.println(Thread.currentThread().getName()
                                                        +"="+count);
                    count--;
                }else{
                    break;
                }
            }
        }
    }
}
```

以上代码创建了 Object 对象 o，在同步块中，o 充当了同步锁的作用。

每个 Java 对象都有且只有一个同步锁，在任何时刻，最多只允许一个线程拥有这把锁。当第一个线程拥有了这个同步锁，执行同步块里的代码时，其他的线程因为没有拥有这把锁，就不能执行同步块里的代码。即使该线程睡眠了，其他线程也是不能执行同步块里的代码。直到该线程执行完同步块释放了 o 的同步锁，其他线程才有机会执行同步块里的代码。

当前对象也可以作为同步锁使用，所以也可以这样写同步块：

```
synchronized(this){}
```

当一个线程开始执行同步代码块时，并不意味着以不中断的方式运行。进入同步代码的线程也可以执行 Thread.sleep()或者执行 Thread.yield()方法，此时它并没有释放锁，只是把运行机会让给了其他的线程。

13.8.2 同步方法

使用 synchronized 关键字修饰的方法为同步方法，同步方法和同步块一样有线程同步的功能。

```
class TicketSell implements Runnable{
    //有 100 张票
    int count=100;
    //同步锁
    Object o=new Object();
```

```
    public void run(){
        while(true){
            if(count<1){
                break;
            }
            sale();
        }
    }
    public synchronized void sale(){
        if(count>0){
            try{
                Thread.sleep(10);
            }catch(Exception e){
            }

    System.out.println(Thread.currentThread().getName()+"="+count);
            count--;
        }
    }
}
```

同步方法中使用当前对象 this 作为同步锁，所以不需要额外声明同步锁。

synchronized 声明不会被继承。如果一个用 synchronized 修饰的方法被子类覆盖，那么子类中这个方法不再保持同步，除非也用 synchronized 修饰。

13.8.3 同步与并发

同步是解决共享资源竞争的有效手段。当一个线程已经在操纵共享资源时，其他线程只能等待，只有当已经在操纵共享资源的线程执行同步代码后，其他线程才有机会操纵共享资源。

但是，多线程的同步与并发是一对此消彼长的矛盾。假想有 10 个人同到一口井里打水，每个人都要打 10 桶水，人代表线程，井代表共享资源。一种同步方法是：所有的人依次打水，只用当前一个人打完 10 桶水后，其他人才有机会打水。当一个人在打水期间，其他人必须等待。轮到最后一个打水的人肯定怨声载道，因为他必须等到前面 9 个人打完 90 桶水后才能打水。

为了提高并发性能，应该使同步代码块中包含尽可能少的操作，使得一个线程能尽快释放锁，减少其他线程等待锁的时间。可以改为一个人打完一桶水后，就让其他人打水，大家轮流打水，直到每个人都打完 10 桶水。

13.8.4 线程安全的类

一个线程安全的类满足以下条件：

这个类的对象可以同时被多个线程安全的访问。

每个线程都能正常执行原子操作，得到正确的结果。

在每个线程的原子操作都完成后，对象处于逻辑上合理的状态。

13.9 学习目标

1. 掌握在 Java 中创建线程的两种方式（写出代码）。
2. 掌握线程状态转移的结构。
3. 掌握线程调度的方法。
4. 理解线程的同步（为什么需要线程的同步），会写同步块和同步方法。

13.10 作业

1. （选择题）下面说法中错误的一项是（ ）
   A．线程就是程序 B．线程是一个程序的单个执行流
   C．多想成用于实现并发 D．多线程是指一个程序的多个执行流

2. 下列哪些方法可以使线程从运行状态进入阻塞状态（ ）
   A. sleep B. wait C. yield D.start

3. 下列说法中错误的两项是（ ）
   A．一个线程是一个 Thread 类的实例
   B．线程从调用实现 Runnable 接口的实例的 run()方法开始执行
   C．新建的线程调用 start()方法就能立即进入运行状态
   D．在 Java 中，高优先级的可运行线程会抢占低优先级线程

4. 下列关于 Thread 类提供的线程控制方法的说法中，错误的一项是（ ）
   A．在线程 A 中执行线程 B 的 join()方法，则线程 A 等待直到 B 执行完成
   B．线程 A 通过调用 interrupt()方法来中断其阻塞状态
   C．若线程 A 调用方法 isAlive()返回值为 true，则说明 A 正在执行中
   D．currentThread（）方法返回当前线程的引用

5. 下列说法中，错误的一项是（ ）
   A．对象锁在 synchronized 块执行完成之后由持有它的线程返还
   B．对象锁在 synchronized 块中出现异常时有持有它的线程返还
   C．当持有锁的线程调用了该对象的 wait()方法时，线程将释放其持有的锁
   D．当持有锁的线程调用了该对象的构造方法时，线程将释放其持有的锁

6. 编写一个用线程实现一个数字时钟的应用程序。该线程类要采用休眠的方式，把绝对大部分时间让系统使用。

7. 创建一个线程，指定一个限定时间（如 60s），线程运行时，大约每 3s 输出 1 次当前所剩时间，直至给定的限定时间用完。（sleep 方法）。

8. 编写一个多线程程序，其中一个线程完成对某个对象的 int 成员变量的增加操作，即每次加 1，另一个线程完成对该对象的成员变量的减操作，即每次减 1，同时要保证该变量的值不会小于 0，不会大于 10，该变量的初始值为 0。

14 网络编程

14.1 基本概念

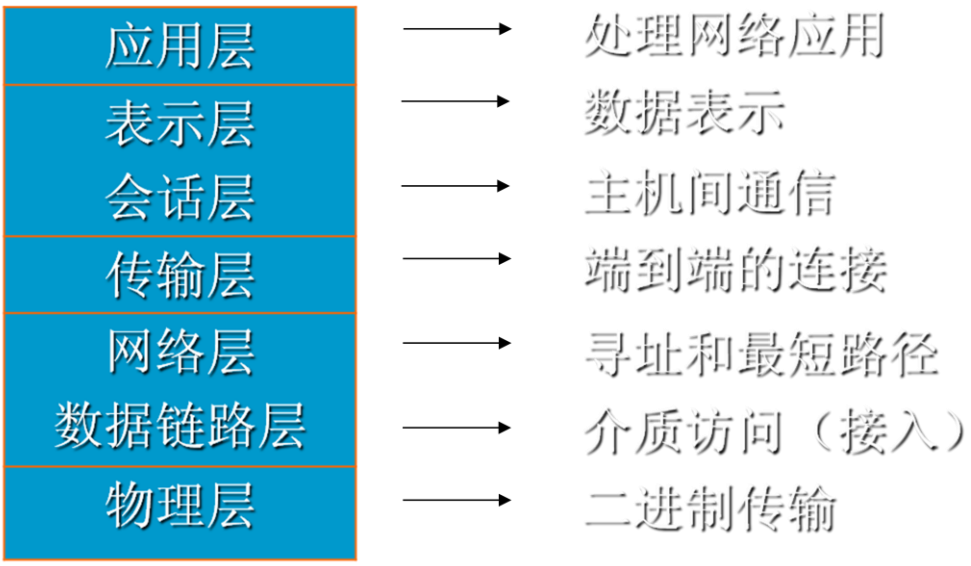
14.1.1 计算机网络

计算机网络是相互连接的独立自主的计算机集合。

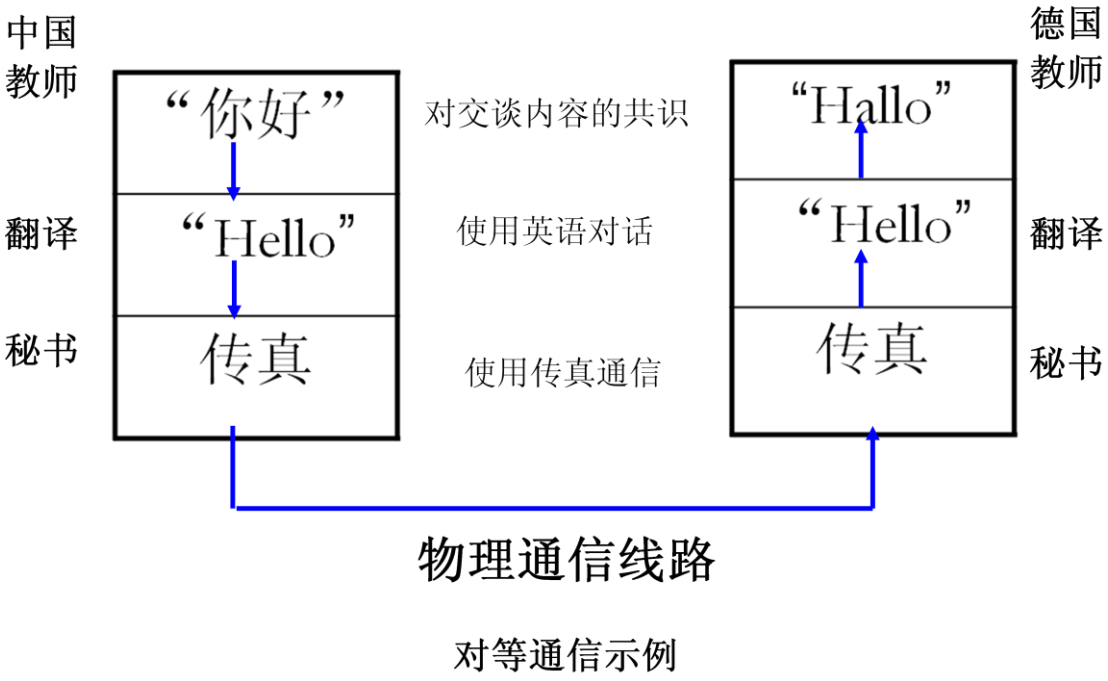
最简单的网络形式是由两台计算机组成的。

14.1.2 ISO/OSI 七层参考模型

OSI（Open System Interconnection）参考模型将网络的不同功能划分为 7 层：



- 通信实体的对等层之间不允许直接通信。
- 各层之间是严格的单向依赖关系。
- 上层使用下层提供的服务----Service user。
- 下层向上层提供服务----Service provider。



对等通信示例

OSI 各层所使用的协议：

- 应用层：远程登录协议 Telnet、文件传输协议 FTP、超文本传输协议 HTTP、域名服务 DNS、简单邮件传输协议 SMTP、邮局协议 POP3 等。
- 传输层：传输控制协议 TCP、用户数据报协议 UDP。
  - TCP：面向连接的可靠的传输协议。
  - UDP：是无连接的，不可靠地传输协议。
- 网络层：网际协议 IP、Internet 互联网控制报文协议 ICMP、Internet 组管理协议 IGMP。

14.1.3 IP 地址

- IP 网络中每台主机都必须有一个唯一的网络地址。
- IP 地址是一个逻辑地址。
- 英特网上的 IP 地址具有全球唯一性。

- 32 位，4 个字节，常用点分十进制的格式表示，例如：192.168.0.12

在 dos 中查看本机 ip 命令：ipconfig

14.1.4 TCP/IP 协议

TCP（Transmission Control Protocol）/IP（Internet Protocol）协议是当前网络数据传输的基础协议。他们可保证不同厂家生产的计算机能在共同网络环境下运行，解决异构网通信问题，TCP/IP 与底层的数据链路层和物理层无关，能广泛地支持由低两层协议构成的物理网络结构。**TCP -- 面向连接的可靠数据传输协议；TCP 重发一切没有收到的数据，进行数据内容准确性检查并保证数据分组的正确顺序。**

IP 协议是网际层的主要协议，支持网间互连的数据报通信。它提供主要功能有：无连接数据报传送、数据报路由选择和差错控制。IP 协议主要特性为，IP 协议将报文传送到目的主机后，无论传送正确与否都不进行检验、不回送确认、不保证分组的正确顺序。

为实现网络中不同计算机之间的通信，每台机器都必须有一个与众不同的标识，这就是 IP 地址， TCP/IP 使用 IP 地址来标识源地址和目的地址。IP 地址格式：数字型，32 位，由 4 个 8 位的二进制 数组成，每 8 位之间用圆点隔开，如：166.111.78.98。

14.1.5 URL

URL（统一资源定位器，Uniform Resource Locator）用于表示 Internet 上资源的地址。这里

所说的资源，可以是文件、目录或更为复杂的对象的引用。

URL 一般由协议名、资源所在的主机名和资源名等部分组成，例如下面的 URL：

所用的协议是 http（Hypertext Transfer Protocol，超文本传输协议）协议，资源所在的主

机名为 home.netscape.com，资源名为 home/welcome.html。有时资源名也可省略，这样将指向默认 的主页面，如 http://www.sun.com。URL 还可以包含端口号来指定与远端主机相连接的端口。如果 不指定端口号，则使用默认值。例如，http 协议的默认端口号是 8080。显式指定端口号的 URL 形式 如下：

http://java.cs.tsinghua.edu.cn:8888/

在 dos 中，ping 域名可以获得域名对应的 ip 地址。

14.1.6 端口号

你发起电话呼叫时，你必须知道所拨的电话号码。如果要发起网络连接，你需要知道远程机器的地址或名字。此外，每个网络连接需要一个端口号，你可以把它想象成电话的分机号码。一旦你 和一台计算机建立连接，你需要指明连接的目的。所以，就如同你可以使用一个特定的分机号码来 和财务部门对话那样，你可以使用一个特定的端口号来和会计程序通信。

TCP/IP 系统中的端口号是一个 16 位的数字，它的范围是 0~65535。实际上，小于 1024 的端口号保留给预定义的服务，而且除非要和那些服务之一进行通信(例如 telnet，SMTP 邮件和 ftp 等)， 否则你不应该使用它们。

客户和服务器必须事先约定所使用的端口。如果系统两部分所使用的端口不一致，那就不能进行通信。

在 dos 中使用 netstat -a -n 命令查看端口的使用情况。

其中，LISTENING 代表当前正处于监听状态，ESTABLISHED 代表已建立连接，正处于通信状态，TIME_WAIT 代表已结束访问，CLOSE_WAIT 代表等待从本地用户发来的连接中断请求。

14.2 使用 URL

java.net 中还有很多的类，让我们从 URL、URLConnection 开始。你不必了解任何底层套接字细节就能在 Java 代码中使用套接字提供一种途径。

14.2.1 URL

java.net 包定义了对应的 URL 类。其常用构造方法及用法举例如下：

```
public URL(String spec);
URL u1 = new URL("http://www.javakc.com/");

public URL(URL context, String spec);
URL u2 = new URL(u1, "index.html");

public URL(String protocol,String host,String file);
URL u3 = new URL("http","www.javakc.com","index.html");

public URL (String protocol,String host,int port,String file);
URL u4 = new URL("http", "www.javakc.com", 80, "index.html");
```

使用 URL 类的 openStream()方法可以建立到当前 URL 的连接并返回一个可用于从该连接读取数 据的输入流对象，方法格式为：

```
public final InputStream openStream() throws IOException
```

下面示例中演示了使用 URL 读取网络资源：

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

public class URLReader {
    public static void main(String args[]) {
        try {
            URL tirc = new URL("http://www.javakc.com");
            BufferedReader in = new BufferedReader(new InputStreamReader(
                    tirc.openStream()));
            String s;
            while ((s = in.readLine()) != null)
                System.out.println(s);
            in.close();
        } catch (MalformedURLException e) {
            System.out.println(e);
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

程序运行后将在控制台输出 www.javakc.com 主页面的源码。

14.2.2 URLConnection

URLConnection 类是所有在应用程序和 URL 之间创建通信链路的类的抽象超类。URLConnection 在获取 Web 服务器上的文档方面特别有用，但也可用于连接由 URL 标识的任何资源。该类的实例既可用于从资源中读，也可用于往资源中写。例如，您可以连接到一个 servlet 并发送一个格式良好的 XML String 到服务器上进行处理。URLConnection 的具体子类（例如 HttpURLConnection）提供特定于它们实现的额外功能。对于我们的示例，我们不想做任何特别的事情，所以我们将使用 URLConnection 本身提供的缺省行为。

连接到 URL 包括几个步骤：

- 创建 URLConnection
- 用各种 setter 方法配置它
- 连接到 URL
- 用各种 getter 方法与它交互

接着，我们将看一些演示如何用 URLConnection 来从服务器请求文档的样本代码。

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

```
public class URLClient {
    protected URLConnection connection;

    public static void main(String[] args) {
        URLClient client = new URLClient();
        String javakc = client.getDocumentAt("http://www.javakc.com");
        System.out.println(javakc);
    }

    public String getDocumentAt(String urlString) {
        StringBuffer document = new StringBuffer();
        try {
            URL url = new URL(urlString);
            URLConnection conn = url.openConnection();
            BufferedReader reader = new BufferedReader(new
InputStreamReader(
                    conn.getInputStream()));
            String line = null;
            while ((line = reader.readLine()) != null)
                document.append(line + "\n");
            reader.close();
        } catch (MalformedURLException e) {
            System.out.println("Unable to connect to URL: " + urlString);
        } catch (IOException e) {
            System.out.println("IOException when connecting to URL:
    "
                    + urlString);
        }
        return document.toString();
    }
}
```

要做的第一件事是导入 java.net 和 java.io。我们给我们的类一个实例变量以保存一个 URLConnection。我们的类有一个 main()方法，它处理浏览文档的逻辑流。我们的类还有一个 getDocumentAt()方法，该方法连接到服务器并向它请求给定文档。下面我们将分别探究这些方法的细节。

main() 方法处理浏览文档的逻辑流：创建一个新的 URLClient 并用一个有效的 URLString 调用 getDocumentAt()。当调用返回该文档时，我们把它存储在 String，然后将它打印到控制台。然而，实际的工作是在 getDocumentAt()方法中完成的。

getDocumentAt()方法处理获取 Web 上的文档的实际工作：getDocumentAt()方法有一个 String 参数，该参数包含我们想获取的文档的 URL。我们在开始时创建一个 StringBuffer 来保存文档的行。然后我们用我们传进去的 urlString 创建一个新 URL。接着创建一个 URLConnection 并打开它。

一旦有了一个 URLConnection，我们就获取它的 InputStream 并包装进 InputStreamReader，然后我们又把 InputStreamReader 包装进 BufferedReader 以使我们能够读取想从服务器上获取的文档的行。在 Java 代码中处理套接字时，我们将经常使用这种包装技术，但我们不会总是详细讨论它。

有了 BufferedReader，就使得我们能够容易地读取文档内容。我们在 while 循环中调用 reader 上的 readLine()。

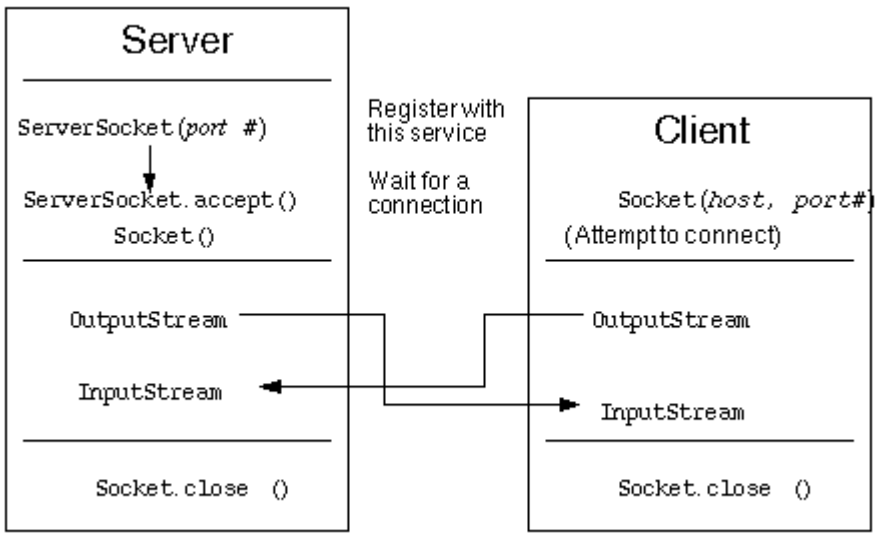
对 readLine()的调用将直至碰到一个从 InputStream 传入的行终止符（例如换行符）时才阻塞。如果没碰到，它将继续等待。只有当连接被关闭时，它才会返回 null。在这个案例中，一旦我们获取一个行（line），我们就把它连同一个换行符一起附加（append）到名为 document 的 StringBuffer 上。这保留了服务器端上读取的文档的格式。

我们在读完行之后关闭 BufferedReader：reader.close();

如果提供给 URL 构造器的 urlString 是无效的，那么将抛出 MalformedURLException。如果发生了别的错误，例如当从连接上获取 InputStream 时，那么将抛出 IOException。

14.3 socket 编程

在 Java 编程语言中，TCP/IP socket 连接是用 java.net 包中的类实现的。下图说明了服务器和客户端所发生的动作。



服务器分配一个端口号。如果客户请求一个连接，服务器使用 accept()方法打开 socket 连接。

客户在 host 的 port 端口建立连接。

服务器和客户使用 InputStream 和 OutputStream 进行通信。

14.3.1 Socket 基础

计算机以一种非常简单的方式进行相互间的操作和通信。计算机芯片是以 1 和 0 的形式存储并传输数据的开–闭转换器的集合。当计算机想共享数据时，它们所需做的全部就是以一致的速度、顺序、定时等等来回传输几百万比特和字节的数据流。每次想在两个应用程序之间进行信息通信时，您怎么会愿意担心那些细节呢？

为免除这些担心，我们需要每次都以相同方式完成该项工作的一组包协议。这将允许我们处理应用程序级的工作，而不必担心低级网络细节。这些成包协议称为协议栈（stack）。TCP/IP 是当今最常见的协议栈。多数协议栈（包括 TCP/IP）都大致对应于国际标准化组织（International Standards Organization，ISO）的开放系统互连参考模型（Open Systems Interconnect Reference Model，OSIRM）。OSIRM 认为在一个可靠的计算机组网中有七个逻辑层。各个地方的公司都对这个模型某些层的实现做了一些贡献，从生成电子信号（光脉冲、射频等等）到提供数据给应用程序。

socket 是指在一个特定编程模型下，进程间通信链路的端点。因为这个特定编程模型的流行，socket 这个名字在其他领域得到了复用，包括 Java 技术。

当进程通过网络进行通信时，Java 技术使用它的流模型。一个 socket 包括两个流：一个输入流和一个输出流。如果一个进程要通过网络向另一个进程发送数据，只需简单地写入与 socket 相关联的输出流。一个进程通过从与 socket 相关联的输入流读来读取另一个进程所写的数据。

建立网络连接之后，使用与 socket 相关联的流和使用其他流是非常相似的。 我们不想涉及层的太多细节，但您应该知道套接字位于什么地方

使用套接字的代码工作于表示层。表示层提供应用层能够使用的信息的公共表示。假设您打算把应用程序连接到只能识别 EBCDIC 的旧的银行系统。应用程序的域对象以 ASCII 格式存储信息。 在这种情况下，您得负责在表示层上编写把数据从 EBCDIC 转换成 ASCII 的代码，然后（比方说）给应用层提供域对象。应用层然后就可以用域对象来做它想做的任何事情。

套接字处理代码只存在于表示层中。您的应用层无须知道套接字如何工作的任何事情。

简言之，一台机器上的套接字与另一台机器上的套接字交谈就创建一条通信通道。程序员可以用该通道来在两台机器之间发送数据。当您发送数据时，TCP/IP 协议栈的每一层都会添加适当的报头信息来包装数据。这些报头帮助协议栈把您的数据送到目的地。好消息是 Java 语言通过"流"为您的代码提供数据，从而隐藏了所有这些细节，这也是为什么它们有时候被叫做流套接字（streaming socket）的原因。

把套接字想成两端电话上的听筒 – 我和您通过专用通道在我们的电话听筒上讲话和聆听。

直到我们决定挂断电话，对话才会结束（除非我们在使用蜂窝电话）。而且我们各自的电话线路都占线， 直到我们挂断电话。

如果想在没有更高级机制如 ORB（以及 CORBA、RMI、IIOP 等等）开销的情况下进行两台计算机之间的通信，那么套接字就适合您。套接字的低级细节相当棘手。幸运的是，Java 平台给了您一些虽然简单但却强大的更高级抽象，使您可以容易地创建和使用套接字。

14.3.2 套接字的类型

一般而言，Java 语言中的套接字有以下两种形式：

- TCP 套接字（由 Socket 类实现）
- UDP 套接字（由 DatagramSocket 类实现）

14.3.3 面向连接的 TCP 协议

“面向连接”就是在正式通信前必须要与对方建立起连接。比如你给别人打电话，必须等线路接通了、对方拿起话筒才能相互通话。

TCP（Transmission Control Protocol，传输控制协议）是基于连接的协议，也就是说，在正式收发数据前，必须和对方建立可靠的连接。一个 TCP 连接必须要经过三次“对话”才能建立起来，其中的过程非常复杂，我们这里只做简单、形象的介绍，你只要做到能够理解这个过程即可。

我们来看看这三次对话的简单过程：主机 A 向主机 B 发出连接请求数据包：“我想给你发数据，可以吗？”，这是第一次对话；主机 B 向主机 A 发送同意连接和要求同步（同步就是两台主机一个在发送，一个在接收，协调工作）的数据包：“可以，你什么时候发？”，这是第二次对话；主机 A 再发出一个数据包确认主机 B 的要求同步：“我现在就发，你接着吧！”，这是第三次对话。三次“对话”的目的是使数据包的发送和接收同步，经过三次“对话”之后，主机 A 才向主机 B 正式发送数据。

TCP 协议能为应用程序提供可靠的通信连接，使一台计算机发出的字节流无差错地发往网络上的其他计算机，对可靠性要求高的数据通信系统往往使用 TCP 协议传输数据。

我们来做一个实验，用计算机 A 从“网上邻居”上的一台计算机 B 拷贝大小为 8,644,608 字节的文件，通过状态栏右下角网卡的发送和接收指标就会发现：虽然是数据流是由计算机 B 流向计算机 A，但是计算机 A 仍发送了 3,456 个数据包，这些数据包是怎样产生的呢？因为文件传输时使用了 TCP/IP 协议，更确切地说是使用了面向连接的 TCP 协议，计算机 A 接收数据包的时候，要向计算机 B 回发数据包，所以也产生了一些通信量。

如果事先用网络监视器监视网络流量，就会发现由此产生的数据流量是 9,478,819 字节，比文件大小多出 10.96%，原因不仅在于数据包和帧本身占用了一些空间，而且也在于 TCP 协议面向连接的特性导致了一些额外的通信量的产生。

综上所述，我们可以使用 TCP 协议开发下载程序、telnet 程序。

14.3.4 面向非连接的 UDP 协议

“面向非连接”就是在正式通信前不必与对方先建立连接，不管对方状态就直接发送。与风行的手机短信非常相似：你在发短信的时候，只需要输入对方手机号就 OK 了。

UDP（User Data Protocol，用户数据报协议）是与 TCP 相对应的协议。它是面向非连接的协议，它不与对方建立连接，而是直接就把数据包发送过去！

UDP 适用于一次只传送少量数据、对可靠性要求不高的应用环境。比如，我们经常使用“ping”命令来测试两台主机之间 TCP/IP 通信是否正常，其实“ping”命令的原理就是向对方主机发送 UDP 数据包，然后对方主机确认收到数据包，如果数据包是否到达的消息及时反馈回来，那么网络就是通的。例如，在默认状态下，一次“ping”操作发送 4 个数据包（如图所示）。大家可以看到，发送的数据包数量是 4 包，收到的也是 4 包（因为对方主机收到后会发回一个确认收到的数据包）。这充分说明了 UDP 协议是面向非连接的协议，没有建立连接的过程。正因为 UDP 协议没有连接的过程，所以它的通信效率高；但也正因为如此，它的可靠性不如 TCP 协议高。QQ 就使用 UDP 发消息，因此有时会出现收不到消息的情况。

综上所述，我们可以使用 UDP 协议开发视频程序、语音聊天程序。

14.3.5 TCP 和 UDP 的区别

	TCP	UDP
是否连接	面向连接的传输控制协议	无连接的数据报服务
传输可靠性	具有高可靠性，确保传输数据的正确性，不出现丢失或乱序	在传输数据前不建立连接，不对数据报进行检查与修改，无须等待对方的应答，所以会出现分组丢失、重复、乱序
速度	工作效率较 UDP 协议低	具有较好的实时性，速度快
网络开销	“请求-确认”模式，网络开销较 UDP 大	数据包结构简单，因此网络开销小
应用场合	传输大量的数据	少量数据

14.4 Tcp 套接字编程

java.net 包中定义了两个类 Socket 和 ServerSocket，分别用来表示双向连接的客户端和服务器端。其常用的构造方法有：

```
Socket(InetAddress address, int port);

Socket(InetAddress address, int port, boolean stream);

Socket(String host, int port);

Socket(String host, int port, boolean stream);

ServerSocket(int port);

ServerSocket(int port, int count);
```

网络编程的四个基本步骤为：

- 创建 socket
- 打开连接到 socket 的输入/输出流
- 按照一定的协议对 socket 进行读/写操作
- 关闭 socket

简单的 client/server 示例：

服务器端程序：TcpServer.java

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

public class TcpServer {
    public static void main(String[] args) {
        try {
            ServerSocket s = new ServerSocket(8000);
            while (true) {
                Socket s1 = s.accept();

                InputStream is=s1.getInputStream();
                DataInputStream dis=new DataInputStream(is);
                String msg=dis.readUTF();
                System.out.println("接收到客户端发来的请求:"+msg);

                dis.close();
                s1.close();
```

```
            }
        } catch (IOException e) {
            System.out.println("程序运行出错:" + e);
            e.printStackTrace();
        }
    }
}
```

客户端程序：TestClient.java

```
import java.net.*;
import java.io.*;
import java.util.Scanner;

public class TcpClient {
    public static void main(String args[]) {
        try {
            Socket s1 = new Socket("127.0.0.1", 8000);
            OutputStream os=s1.getOutputStream();
            DataOutputStream dos=new DataOutputStream(os);
            // 向服务器端发送数据
            Scanner scan=new Scanner(System.in);
            dos.writeUTF(scan.next());

            dos.close();
            s1.close();
        } catch (ConnectException connExc) {
            System.err.println("服务器连接失败！");
        } catch (IOException e) {
        }
    }
}
```

TestServer.java 和 TestClient.java 程序运行在两台不同的机器上，首先运行 TestServer 程序，创建 ServerSocket 对象、监听所在机器（服务器）的指定 8000 号端口，等待客户端连接请求；然后运行 TestClient 程序，创建 Socket 对象，连接服务器的 8000 号端口。建立连接后，服务器端程序打开其 Socket 对象关联的输出流，并向其中写出有关连接者的信息，然后关闭输出流及 Socket 对象、程序退出；客户端程序打开 Socket 对象关联的输入流，从中读取服务器端发送来的 信息并显示到屏幕上。

14.5 UDP 套接字编程

用户数据报协议(UDP)由 Java 软件的 DatagramSocket 和 DatagramPacket 类支持。包是自包含的消息，它包括有关发送方、消息长度和消息自身。

14.5.1 DatagramPacket

DatagramPacket 有两个构造函数：一个用来接收数据，另一个用来发送数据。

DatagramPacket(byte[] recvBuf, int readLength)

用来建立一个字节数组以接收 UDP 包。byte 数组在传递给构造函数时是空的，而 int 值用来设定要读取的字节数(不能比数组的长度还大)。

DatagramPacket(byte[] sendBuf, int sendLength, InetAddress iaddr, int iport)

用来建立将要传输的 UDP 包。sendLength 不应该比 sendBuf 字节数组的长度要大。

14.5.2 DatagramSocket

DatagramSocket 用来读写 UDP 包。这个类有三个构造函数，允许你指定要绑定的端口号和 internet 地址： DatagramSocket() 一绑定本地主机的所有可用端口 DatagramSocket(int port)一绑定本地主机的指定端口

DatagramSocket(int port, InetAddress iaddr)绑定指定地址的指定端口

14.5.3 UDP 示例

UDP 服务器

```
import java.io.*;
import java.net.*;
import java.util.*;

public class UdpServer {
    // 返回server当前的时间
    public byte[] getTime() {
        Date d = new Date();
        String s="服务器端的时间: "+d.toString();
        return s.getBytes();
```

```
    }

    // Server的运行
    public void go() throws IOException {
        // 用来接收client数据的数据包
        DatagramPacket inDataPacket;
        // 用来向client发送数据的数据包
        DatagramPacket outDataPacket;

        // 创建用来发送和接收数据报包的套接字
        DatagramSocket datagramSocket = new DatagramSocket(8000);
        System.out.println("UDP 服务在 8000 端口启动了");
        while (true) {
            // 构造 DatagramPacket，用来接收长度为 length 的数据包。
            byte[] msg = new byte[12];
            inDataPacket = new DatagramPacket(msg, msg.length);

            // 接收数据包
            datagramSocket.receive(inDataPacket);
            System.out.println("msg::"+new String(msg));

        }
    }

    // 启动Server
    public static void main(String args[]) {
        UdpServer udpServer = new UdpServer();
        try {
            udpServer.go();
        } catch (IOException e) {
            System.out.println("在socket中出现了IOException");
            System.out.println(e);
            System.exit(1);
        }
    }
}
```

UDP 服务器在 8000 端口监听客户的请求。当它从客户接收到一个 DatagramPacket 时，它发送服务器上的当前时间。

UDP 客户端

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.UnknownHostException;

public class UdpClient {
```

```
    public void go() throws IOException, UnknownHostException {
        // 用来向Server发送数据的数据包
        DatagramPacket outDataPacket;

        // 创建用来发送和接收数据报包的套接字
        DatagramSocket datagramSocket = new DatagramSocket();

        //发送到服务器端的数据
        byte[] msg = "hello,javakc".getBytes();
        // 取得服务器的ip地址
        InetAddress serverAddress =
InetAddress.getByName("192.168.1.107");

        // 构造数据报包，用来将长度为 length 的包发送到指定主机上的指定端口号 8000
        outDataPacket = new DatagramPacket(msg, msg.length, serverAddress,
8000);
        // 发送请求到server.
        datagramSocket.send(outDataPacket);

        // 关闭socket
        datagramSocket.close();
    }

    public static void main(String args[]) {
        UdpClient udpClient = new UdpClient();
        try {
            udpClient.go();
        } catch (Exception e) {
            System.out.println("Exception occured with socket.");
            System.out.println(e);
            System.exit(1);
        }
    }
}
```

UDP 客户向前面创建的客户发送一个数据包并接收一个包含服务器实际时间的包。

15 设计模式基础

在面向对象的软件设计中，总是希望避免重复设计或尽可能少做重复设计。有经验的面向对象设计者的确能做出良好的设计，而新手则面对众多选择无从下手，总是求助于以前使用过的非面向对象技术。有经验的设计者显然知道一些新手所不知道的东西，这又是什么呢？ 内行的设计者知道：不是解决任何问题都要从头做起。他们更愿意复用以前使用过的解决方案。

当找到一个好的解决方案，他们会一遍又一遍地使用。这些经验是他们成为内行的部分原因。它们 帮助设计者将新的设计建立在以往工作的基础上，复用以往成功的设计方案。一个熟悉这些模式的设计者不需要再去发现它们，而能够立即将它们应用于设计问题中。

设计模式使人们可以更加简单方便地复用成功的设计和体系结构。将已证实的技术表述成设计模式也会使新系统开发者更加容易理解其设计思路。设计模式帮助你做出有利于系统复用的选择， 避免设计损害了系统复用性。通过提供一个显式类和对象作用关系以及它们之间潜在联系的说明规范，设计模式甚至能够提高已有系统的文档管理和系统维护的有效性。简而言之，设计模式可以帮助设计者更快更好地完成系统设计。

15.1 什么是设计模式

Christopher Alexander 说过：“每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的解决方案的核心。这样，你就能一次又一次地使用该方案而不必做重复劳动”。尽管Alexander所指的是城市和建筑模式，但他的思想也同样适用于面向对象设计模式，只是在面向对象的解决方案里，我们用对象和接口代替了墙壁和门窗。两类模式的核心都在于提供了相关问题的解决方案。

一般而言，一个模式有四个基本要素：

模式名称（pattern name）

一个助记名，它用一两个词来描述模式的问题、解决方案和效果。

问题(problem)

描述了应该在何时使用模式。它解释了设计问题和问题存在的前因后果，它可能描述了特定的设计问题，如怎样用对象表示算法等。也可能描述了导致不灵活设计的类或对象结构。有时候，问题部分会包括使用模式必须满足的一系列先决条件。

解决方案(solution)

描述了设计的组成成分，它们之间的相互关系及各自的职责和协作方式。因为模式就像一个模板，可应用于多种不同场合，所以解决方案并不描述一个特定而具体的设计或实现，而是提供设计问题的抽象描述和怎样用一个具有一般意义的元素组合（类或对象组合）来解决这个问题。

效果(consequences)

描述了模式应用的效果及使用模式应权衡的问题。尽管我们描述设计决策时，并不总提到模式效果，但它们对于评价设计选择和理解使用模式的代价及好处具有重要意义。软件效果大多关

注对时间和空间的衡量，它们也表述了语言和实现问题。因为复用是面向对象设计的要素之一，所以模式效果包括它对系统的灵活性、扩充性或可移植性的影响，显式地列出这些效果对理解和评价这些模式很有帮助。出发点的不同会产生对什么是模式和什么不是模式的理解不同。一个人的模式对另一个人来说可能只是基本构造部件。

15.2 什么是框架

框架(Framework)是构成一类特定软件可复用设计的一组相互协作的类。例如，一个框架能帮助建立适合不同领域的图形编辑器，像艺术绘画、音乐作曲和机械 C A D。另一个框架也许能帮助你建立针对不同程序设计语言和目标机器的编译器。而再一个也许能帮助你建立财务建模应用。你可以定义框架抽象类的应用相关的子类，从而将一个框架定制为特定应用。

框架规定了你的应用的体系结构。它定义了整体结构，类和对象的分割，各部分的主要责任，类和对象怎么协作，以及控制流程。框架预定义了这些设计参数，以便于应用设计者或实现者能集中精力于应用本身的特定细节。框架记录了其应用领域的共同的设计决策。因而框架更强调设计复用，尽管框架常包括具体的立即可用的子类。

这个层次的复用导致了应用和它所基于的软件之间的反向控制(inversion of control)。当你使用工具箱(或传统的子程序库)时，你需要写应用软件的主体并且调用你想复用的代码。而当你使用框架时，你应该复用应用的主体，写主体调用的代码。你不得不以特定的名字和调用约定来写操作地实现，但这会减少你需要做出的设计决策。

你不仅可以更快地建立应用，而且应用还具有相似的结构。它们很容易维护，且用户看来也更一致。另一方面，你也失去了一些表现创造性的自由，因为许多设计决策无须你来作出。

如果说应用程序难以设计，那么工具箱就更难了，而框架则是最难的。框架设计者必须冒险决定一个要适应于该领域的所有应用的体系结构。任何对框架设计的实质性修改都会大大降低框架所带来的好处，因为框架对应用的最主要贡献在于它所定义的体系结构。因此设计的框架必须尽可能地灵活、可扩充。

更进一步讲，因为应用的设计如此依赖于框架，所以应用对框架接口的变化是极其敏感的。当框架演化时，应用不得不随之演化。这使得松散耦合更加重要，否则框架的一个细微变化都将引起强烈反应。

刚才讨论的主要设计问题对框架设计而言最具重要性。一个使用设计模式的框架比不用设计模 式的框架更可能获得高层次的设计复用和代码复用。成熟的框架通常使用了多种设计模式。设计模 式有助于获得无须重新设计就可适用于多种应用的框架体系结构。

当框架和它所使用的设计模式一起写入文档时，我们可以得到另外一个好处。了解设计模式的人能较快地洞悉框架。甚至不了解设计模式的人也可以从产生框架文档的结构中受益。加强文档工作对于所有软件而言都是重要的，但对于框架其重要性显得尤为突出。学会使用框架常常是一个必须克服很多困难的过程。设计模式虽然无法彻底克服这些困难，但它通过对框架设计的主要元素做 更显式的说明可以降低框架学习的难度。

因为模式和框架有些类似，人们常常对它们有怎样的区别和它们是否有区别感到疑惑。它们

最主要的不同在于如下三个方面：

1） 设计模式比框架更抽象，框架是具有实现的半成品；而设计模式是一个抽象的方法

框架能够用代码表示，而设计模式只有其实例才能表示为代码。框架的威力在于它们能够使用程序设计语言写出来，它们不仅能被学习，也能被直接执行和复用。

2） 设计模式是比框架更小的体系结构元素

一个典型的框架包括了多个设计模式，而反之决非如此。

3） 框架比设计模式更加特例化

框架总是针对一个特定的应用领域。一个图形编辑器框架可能被用于一个工厂模拟，但它不会被错认为是一个模拟框架。

框架变得越来越普遍和重要。它们是面向对象系统获得最大复用的方式。较大的面向对象应用将会由多层彼此合作的框架组成。应用的大部分设计和代码将来自于它所使用的框架或受其影响。

15.3 常用的设计模式

15.3.1 单例模式

1：环境：

几乎在每个应用程序中，都需要有一个从中进行全局访问和维护某种类型数据的区域。在面向对象的(OO)系统中也有这种情况，在此类系统中，在任何给定时间只应运行一个类或某个类的一组预定义数量的实例。例如，当使用某个类来维护增量计数器时，此简单的计数器类需要跟踪在多个应用程序领域中使用的整数值。此类需要能够增加该计数器并返回当前的值。对于这种情况，所需的类行为应该仅使用一个类实例来维护该整数，而不是使用其它类实例来维护该整数。最初，人们可能会试图将计数器类实例只作为静态全局变量来创建。 这是一种通用的方法，但实际上只解决一部分问题；它解决了全局可访问性问题，但没有采取任何措施来确保在任何给定的时间只运行一个类实例。应该由类本身来负责只使用一个类实例，而不是由类用户来负责。应该始终不要让类用户来监视和控制运行的类实例的数量。

2：问题：

采用什么方法来控制创建类实例，然后确保在任何给定的时间只创建一个类实例。 这会确切地给我们提供所需的行为，并使客户端不必了解任何类细节。

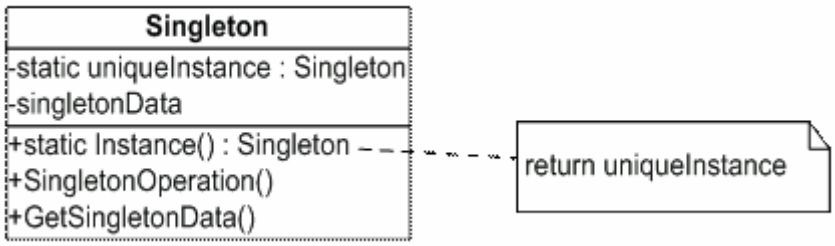
3：解决方案：

Singleton 模式主要作用是保证在 Java 应用程序中，一个类 Class 只有一个实例存在。在很多操作中，比如建立目录 数据库连接都需要这样的单线程操作。还有，singleton 能 够被状态化;这样，多个单态类在一起就可以作为一个状态仓库一样向外提供服务，比如，你要论

坛中的帖子计数器，每次浏览一次需要计数，单态类能否保持住这个计数， 并且能 synchronize 的安全自动加 1，如果你要把这个数字永久保存到数据库，你可以在不修改单态接口的情况下方便的做到。

另外方面，Singleton 也能够被无状态化。提供工具性质的功能，Singleton 模式就为我们提供了这样实现的可能。使用 Singleton 的好处还在于可以节省内存，因为它限制了实例的个数，有利于 Java 垃圾回收（garbage collection）。我们常常看到工厂模式中类装入器(class loader)中也用 Singleton 模式实现的,因为被装入的类实际也属于资源。

Singleton 的逻辑模型如下：



我们看到的是一个简单的类图表，显示有一个 Singleton 对象的私有静态属性以及返回此相同属性的公共方法 Instance ()。这实际上是 Singleton 的核心。还有其他一些属性和方法，用于说明在该类上允许执行的其他操作。为了便于此次讨论，让我们将重点放在实例属性和方法上。 客户端仅通过实例方法来访问任何 Singleton 实例。此处没有定义创建实例的方式。我们还希望能够控制如何以及何时创建实例。 在 OO 开发中，通常可以在类的构造函数中最好地处理特殊对象的创建行为。这种情况也不例外。我们可以做的是，定义我们何时以及如何构造类实例，然后禁止任何客户端直接调用该构造函数。这是在 Singleton 构造中始终使用的方法。让我们看一下 Design Patterns 中的原始示例。通常，将下面所示的 C++ Singleton 示例实现代码示例视为 Singleton 的默认实现。本示例已移植到很多其他编程语言中，通常它在任何地方的形式与此几乎相同。

4：使用示例：

一般 Singleton 模式通常有两种形式：

第一种形式：

```
public class Singleton {
    private Singleton() {
    }

    // 在自己内部定义自己一个实例，是不是很奇怪？
    // 注意这是 private 只供内部调用
    private static Singleton instance = new Singleton();

    // 这里提供了一个供外部访问本 class 的静态方法，可以直接访问
    public static Singleton getInstance() {
        return instance;
```

```
    }
}
```

第二种形式：

```
public class Singleton {
    private Singleton() {
    }
    private static Singleton instance = null;

    public static synchronized Singleton getInstance() {

        // 这个方法比上面有所改进，不用每次都进行生成对象，只是第一次
        // 使用时生成实例，提高了效率！
        if (instance == null)
            instance = new Singleton();
        return instance;
    }
}
```

使用 Singleton.getInstance( )可以访问单态类。

上面第二中形式是 lazy initialization，也就是说第一次调用时初始 Singleton，以后就不用再生成了。

注意到lazy initialization形式中的synchronized，这个synchronized很重要，如果没有synchronized， 那么使用getInstance( ) 是有可能得到多个Singleton实例 。关于lazy initialization 的 Singleton 有很多涉及 double-checked locking (DCL) 的讨论，有兴趣者进一步研究。

一般认为第一种形式要更加安全些。

使用 Singleton 注意事项：

有时在某些情况下，使用 Singleton 并不能达到 Singleton 的目的，如有多个 Singleton 对象同时被不同的类装入器装载；在 EJB 这样的分布式系统中使用也要注意这种情况，因为 EJB 是跨服务器，跨 JVM 的。我们以 SUN 公司的宠物店源码(Pet Store 1.3.1)的 ServiceLocator 为例稍微分析一下：

在 Pet Store 中 ServiceLocator 有两种，一个是 EJB 目录下；一个是 WEB 目录下，我们检查这两个 ServiceLocator 会发现内容差不多，都是提供 EJB 的查询定位服务，可是为什么要分开呢？仔细研究对这两种 ServiceLocator 才发现区别：在 WEB 中的 ServiceLocator 的采取 Singleton 模式， ServiceLocator 属于资源定位，理所当然应该使用 Singleton 模式。但是在 EJB 中，Singleton 模 式已经失去作用，所以 ServiceLocator 才分成两种，一种面向 WEB 服务的，一种是面向 EJB 服务的。Singleton 模式看起来简单，使用方法也很方便，但是真正用好，是非常不容易，需要对 Java 的类线程内存等概念有相当的了解。

15.3.2 简单工厂模式

有这样一个面试题：“请用面向对象语言实现一个计算器程序，要求输入两个数字和运算符号，得到结果。”

这个问题很简单，没有复杂的逻辑算法，编程初学者往往这样解答：

```
public class Client {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入操作数 1:");
        String s1=sc.next();
        System.out.println("请输入操作符:");
        String oper=sc.next();
        System.out.println("请输入操作数 2:");
        String s2=sc.next();

        double result;
        try{
            if("+".equals(oper) ){
                result=Double.valueOf(s1)+Double.valueOf(s2);
            }else if("-".equals(oper) ){
                result=Double.valueOf(s1)-Double.valueOf(s2);
            }else if("*".equals(oper) ){
                result=Double.valueOf(s1)*Double.valueOf(s2);
            }else if("/".equals(oper) ){
                if(Double.valueOf(s2)==0){
                    System.out.println("错误：除数不能为 0");
                    return;
                }
                result=Double.valueOf(s1)/Double.valueOf(s2);
            }else{
                System.out.println("错误：请输入正确操作符");
                return;
            }
            System.out.println("结果是："+result);
        }catch(Exception e){
            System.out.println("错误：请输入数字");
        }
    }
}
```

初学者碰到问题就直觉地用计算机能够解释的逻辑来描述和表达待解决的问题及具体的求解过程。这其实使用计算机的方式去思考，先要求输入两个数字和运算符号，然后根据运算符号判断选择如何运算，得到结果，这本身没有错，但这样的思维却使得我们的程序只能为满足实现当前的需求，程序不容易维护，不容易扩展，更不容易复用。从而达不到高质量代码的要求。

先听个故事吧。

话说三国时期，曹操带领百万大军攻打东吴，大军在长江赤壁驻扎，军船连成一片，眼看就要灭掉东吴，统一天下，曹操大悦，于是大宴众文武，在酒席间，曹操诗性大发，不觉吟道：“喝酒唱歌，人生真爽。……”。众文武齐呼：“丞相好诗！”于是一臣子速命印刷工匠刻版印刷，以便流传天下。

样张出来给曹操一看，曹操感觉不妥，说道：“喝与唱，此话过俗，应改为‘对酒当歌’较好！”，于是此臣就命工匠重新来过。工匠眼看连夜刻版之工，彻底白费，心中叫苦不迭。只得照办。

样张再次出来请曹操过目，曹操细细一品，觉得还是不好，说：“人生真爽太过直接，应改问语才够意境，因此应改为‘对酒当歌，人生几何？……’”。当臣转告工匠之时，工匠晕倒……！

三国时期活字印刷还未发明，所以要改字的时候，就必须要整个刻板全部重新刻。如果是有了活字印刷，则只需更改四个字即可，其余工作都未白做。

总结如下：第一，要改，只需更改要改之字，此为**可维护**；第二，这些字并非用完这次就无用，完全可以在后来的印刷中重复使用，此乃**可复用**；第三，此诗若要加字，只需另刻字加入即可，这是**可扩展**；第四，字的排列其实可能是竖排可能是横排，此时只需将活字移动就可做到满足排列需求，此是**灵活性**好。

而在活字印刷术出现之前，上面的四种特性都无法满足，要修改，必须重刻，要加字，必须重刻，要重新排列，必须重刻，印完这本书后，此版已无任何可再利用价值。

在程序开发过程中，有太多的类似曹操这样的客户要改变需求，更改最初想法的事件。其实客观地说，客户的要求也并不过份，不就是改几个字吗，但面对已完成的程序代码，却是需要几乎重头来过的尴尬，这实在是痛苦不堪。说白了，原因就是因为我们原先所写的程序，不容易维护，灵活性差，不容易扩展，更谈不上复用，因此面对需求变化，加班加点，对程序动大手术的那种无奈也就成了非常正常的事了。之后当我学习了面向对象的分析设计编程思想，开始考虑**通过封装、继承、多态把程序的耦合度降低**，传统印刷术的问题就在于所有的字都刻在同一版面上造成耦合度太高所致，开始**用设计模式使得程序更加的灵活，容易修改，并且易于复用**。这就是面向对象带来的好处。

有的人认为初级程序员的工作就是复制和粘贴，这其实是非常不好的编程习惯，因为程序中重复的代码多到一定程度，维护的时候，可能就是一场灾难。越大的系统，这种方式带来的问题越严重，编程有一原则，就是用尽可能的办法去避免重复。

我们使用继承和多态对上面的程序进行改造。考虑到对程序的扩展，以后可能会添加新的运算符，将基础数据封装到一个基类中。

首先写一个运算类 Operation，包括两个操作数和运算结果。

```
//运算类(父类)
public class Operation {
    private double numberA=0;
    private double numberB=0;
    private double result=0;
    //省略set、get方法
}
```

根据运算符（加减乘除），对基类进行扩展，也就是创建 Operation 类的四个子类。

```
//加法类
public class OperationAdd extends Operation{
    public double getResult() {
        return getNumberA()+getNumberB();
    }
}
//减法类
public class OperationSub extends Operation{
    public double getResult() {
        return getNumberA()-getNumberB();
    }
}
//乘法类
public class OperationMul extends Operation{
    public double getResult() {
        return getNumberA()*getNumberB();
    }
}
//除法类
public class OperationDiv extends Operation{
    public double getResult() {
```

```
            return getNumberA()/getNumberB();
        }
    }
```

客户端要解决的问题就是如何去实例化对象，采用工厂模式解决到底要实例化谁。用一个单独的类来做这个创造实例的过程，就是工厂。

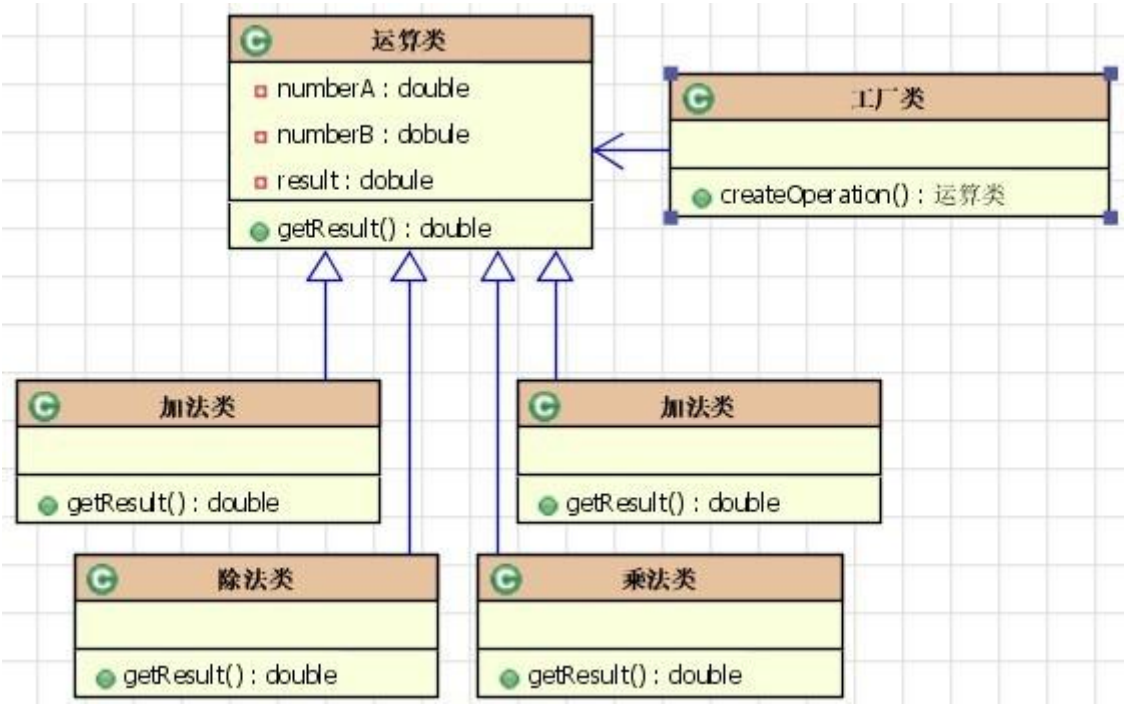
```
public class OperationFactory {
    public static Operation createOperation(String operate){
        Operation oper=null;
        if("+".equals(operate) ){
            oper=new OperationAdd();
        }else if("-".equals(operate) ){
            oper=new OperationSub();
        }else if("*".equals(operate) ){
            oper=new OperationMul();
        }else if("/".equals(operate) ){
            oper=new OperationDiv();
        }
        return oper;
    }
}
```

只要输入运算符号，工厂就实例化出合适的对象，通过多态，返回父类的方式实现了计算器的结果。客户端代码如下：

```
public class Client {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入操作数 1:");
        String s1=sc.next();
        System.out.println("请输入操作符:");
        String oper=sc.next();
        System.out.println("请输入操作数 2:");
        String s2=sc.next();

        Operation operation=OperationFactory.createOperation(oper);
        operation.setNumberA(Double.valueOf(s1));
        operation.setNumberB(Double.valueOf(s2));
        System.out.println("结果是:"+operation.getResult());
    }
}
```

这样实现后，我们将界面代码和运算逻辑代码分离，不管使用控制台程序、Web 程序、手机程序，都可以用这个运算逻辑代码实现计算器的功能，达到了复用的效果。如果有一天需要更改加法运算，只需要更改 OperationAdd 类就可以了。如果需要增加各种复杂运算，比如平方根、自然对数等，只要增加相应的运算子类，修改工厂类就可以了，和其他的子类无关。

上述代码的类图如下：



15.3.3 模板模式 TemplateMethod

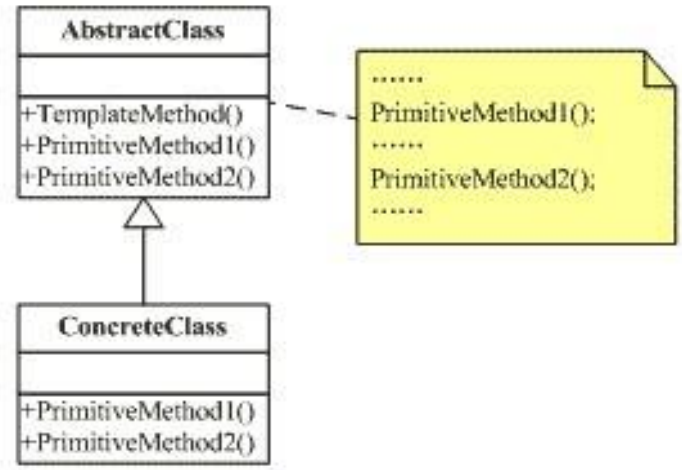
准备一个抽象类，将部分逻辑以具体方法以及具体构造子的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现。这就是模版方法模式的用意。

很多人可能没有想到，模版方法模式实际上是所有模式中最为常见的几个模式之一，而且很多人可能使用过模版方法模式而没有意识到自己已经使用了这个模式。**模版方法模式是基于继承的代码复用的基本技术，模版方法模式的结构和用法也是面向对象设计的核心**。

模版方法模式需要开发抽象类和具体子类的设计师之间的协作。一个设计师负责给出一个算法的轮廓和骨架，另一些设计师则负责给出这个算法的各个逻辑步骤。代表这些具体逻辑步骤的方法称做基本方法（primitive method）；而将这些基本法方法总汇起来的方法叫做模版方法（template method），这个设计模式的名字就是从此而来。

15.3.3.1 结构

模版方法模式的静态结构如下图所示：



这里涉及到两个角色：

抽象模版（AbstractClass）角色有如下的责任：

定义了一个或多个抽象操作，以便让子类实现。这些抽象操作叫做基本操作，它们是一个顶级逻辑的组成步骤。

定义并实现了一个模版方法。这个模版方法一般是一个具体方法，它给出了一个顶级逻辑的骨架，而逻辑的组成步骤在相应的抽象操作中，推迟到子类实现。顶级逻辑也有可能调用一些具体方法。

具体模版（ConcreteClass）角色有如下的责任：

实现父类所定义的一个或多个抽象方法，它们是一个顶级逻辑的组成步骤。

每一个抽象模版角色都可以有任意多个具体模版角色与之对应，而每一个具体模版角色都可以给出这些抽象方法（也就是顶级逻辑的组成步骤）的不同实现，从而使得顶级逻辑的实现各不相同。

15.3.3.2 代码示例

```
//抽象模版
public abstract class AbstractClass {
    public abstract void primitiveMethod1();
    public abstract void primitiveMethod2();
}
//具体模版
public class ConcreteClass extends AbstractClass {
    @Override
    public void primitiveMethod1() {
    }
    @Override
```

```
    public void primitiveMethod2() {
    }
}
//客户端程序
public class Client {
    public static void main(String[] args) {
        AbstractClass abstractClass;
        abstractClass=new ConcreteClass();
        abstractClass.primitiveMethod1();
        abstractClass.primitiveMethod2();
    }
}
```

15.3.4 代理模式

大家先看这个一个故事：

“娇娇同学，这是有人送你的礼物。”一个男生手拿着一个芭比娃娃送到她的面前。

“戴励同学.这是什么意思？”娇娇望着同班的这个男生，感觉很奇怪。

“是这样的，我的好朋友，隔壁三班的卓贾易同学，请我代他送你这个礼物的。”戴励有些脸红。

“为什么要送我礼物，我不认识他呀。”

“他说……他说……他说想和你交个朋友。”戴励脸更红了，右手抓后脑勺，说话吞吞吐吐。

“不用这样，我不需要礼物的。”娇娇显然想拒绝，

“别别别，他是我最好的朋友，他请我代他送礼物给你，也是下了很大决心的，你看在我之前时常帮你辅导数学习题的面子上，就接受一下吧。”戴励有些着急。

“那好吧，今天我对解析几何的椭圆那里还是不太懂，你再给我讲讲。”娇娇提出条件后接过礼物。

“没问题，我们到教室去讲吧。”戴励松了口气。

几天后

“娇娇，这是卓贾易送你的花。”

“娇娇，这是卓贾易送你的巧克力。”

“我不要他送的东西了，我也不想和他交朋友。我愿意……我愿意和你做朋友！”娇娇终于忍不住了，直接表白。

“啊，……我不是在做梦吧……”戴励喜从天降，不敢相信。

戴励用手抓了抓头发说，“其实我也喜欢你。不过，……不过，那我该如何向卓贾易交待呢？”

从此戴励和娇娇开始恋爱了。

在这个故事中，卓贾易是为别人在做嫁衣，他不认识娇娇，找人代理谈恋爱。 戴励是充当了卓贾易的代理。

15.3.4.1 代理模式

代理模式的作用是：为其他对象提供一种代理以控制对这个对象的访问。在某些情况下，一个客户不想或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。 代理模式在访问对象时引入一定程度的间接性，以为这种间接性，可以附加多种用途。

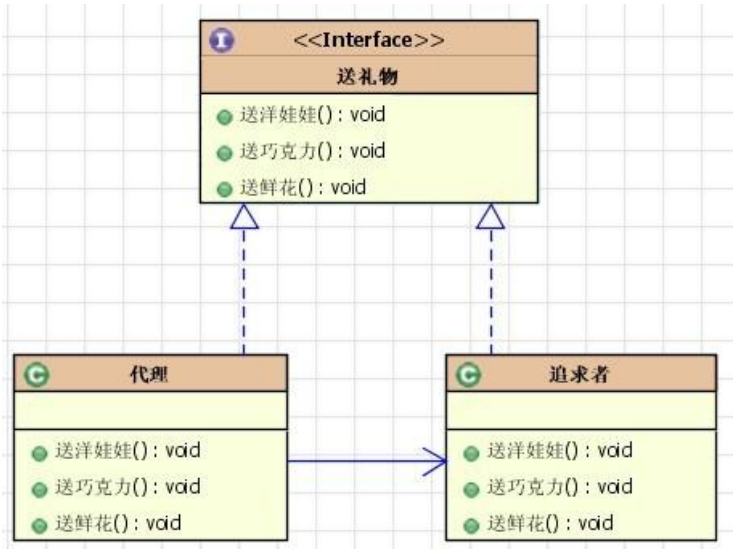
代理模式一般涉及到的角色有：

抽象角色：声明真实对象和代理对象的共同接口；

代理角色：代理对象角色内部含有对真实对象的引用，从而可以操作真实对象，同时代理对象提供与真实对象相同的接口以便在任何时刻都能代替真实对象。代理对象还可以在执行真实对象操作时，附加其他的操作，相当于对真实对象进行封装。

真实角色：代理角色所代表的真实对象，是我们最终要引用的对象。

15.3.4.2 结构



15.3.4.3 代码示例

代理接口（抽象角色），指代理的事情，就是送礼物。

```
public interface GiveGift {
    public void giveDolls();
    public void giveFlowers();
    public void giveChocolate();
}
```

被追求者类，指娇娇。

```
public class SchoolGirl {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

}
```

追求者类（真实角色），指卓贾易。具有属性 mm，代表被追求者。实现了代理接口，具有送礼物的功能。

```
public class Pursuit implements GiveGift{
    SchoolGirl mm;
    public Pursuit(SchoolGirl mm){
        this.mm=mm;
    }

    public void giveChocolate() {
        System.out.println(mm.getName()+"送你巧克力");
    }

    public void giveDolls() {
        System.out.println(mm.getName()+"送你洋娃娃");
    }

    public void giveFlowers() {
        System.out.println(mm.getName()+"送你鲜花");
    }
}
```

代理类（代理角色），代理类具有送礼物的功能，所以也实现了代理接口。属性 gg 代表了卓贾易，表示这个代理类为卓贾易做代理。 大家注意，实现的三个方法中，调用了追求者（卓贾易）的送礼物方法，表示是卓贾易送礼物给娇娇，因为卓贾易不认识娇娇，所以不能直接送礼物给娇娇。

```
public class Proxy implements GiveGift{
    Pursuit gg;
    /**
     * @param mm 被追求者
     */
    public Proxy(SchoolGirl mm){
        gg=new Pursuit(mm);
    }

    public void giveChocolate() {
        gg.giveChocolate();
    }

    public void giveDolls() {
        gg.giveDolls();
    }

    public void giveFlowers() {
        gg.giveFlowers();
    }
}
```

客户端代码：

```
public class Client {
    public static void main(String[] args) {
        SchoolGirl jiaojiao=new SchoolGirl();
        jiaojiao.setName("李娇娇");

        Proxy daili=new Proxy(jiaojiao);
        daili.giveDolls();
        daili.giveFlowers();
        daili.giveChocolate();
    }
}
```

15.4 学习目标

1. 了解什么是设计模式?
2. 了解什么是框架?
3. 了解学习设计模式的方法：四个要素、三层境界
4. 了解框架和设计模式的区别
5. 有哪些常用的的设计模式?
6. 能熟练写出单例模式的开发代码，什么场景下使用单例模式
7. 以 Swing练习为例，能够分析出哪些类可以使用单例模式，哪些类不可以使用单例模式，有什么优缺点，开发时需要注意什么?
8. 能熟练写出工厂模式的开发代码
9. 了解模板模式，知道 Java 中的抽象类使用了模板模式：抽象类是抽象模板，实现类是具体模板。
10. 了解代理模式，知道为什么使用代理模式。
11. 三层架构的分析：为什么使用三层架构，如何在三层架构模式下开发。

    逻辑层应该完成什么功能?

    数据层应该完成什么功能?

    逻辑层和数据层有什么区别?

    逻辑层和数据层如何配合完成开发任务?

15.5 作业

1. 独立写出单例模式的代码。

2. 完成 Swing 项目的三层架构的改造。