

# JAVA 学习笔记

## 编程入门

### ✧ 概述

---

计算机包括 **硬件(hardware)** 和 **软件(software)** 两部分。硬件包括计算机中可以看得见的物理部分。而软件提供看不见的指令。这些指令控制硬件并且使得硬件完成特定的任务。

#### 程序设计

定义：创建（或开发）软件。软件包含了指令，告诉计算机做什么。

应用场景：软件遍布我们周围。除了个人计算机，飞机、汽车、手机甚至烤面包机中，同样运行着软件。

#### 程序设计语言

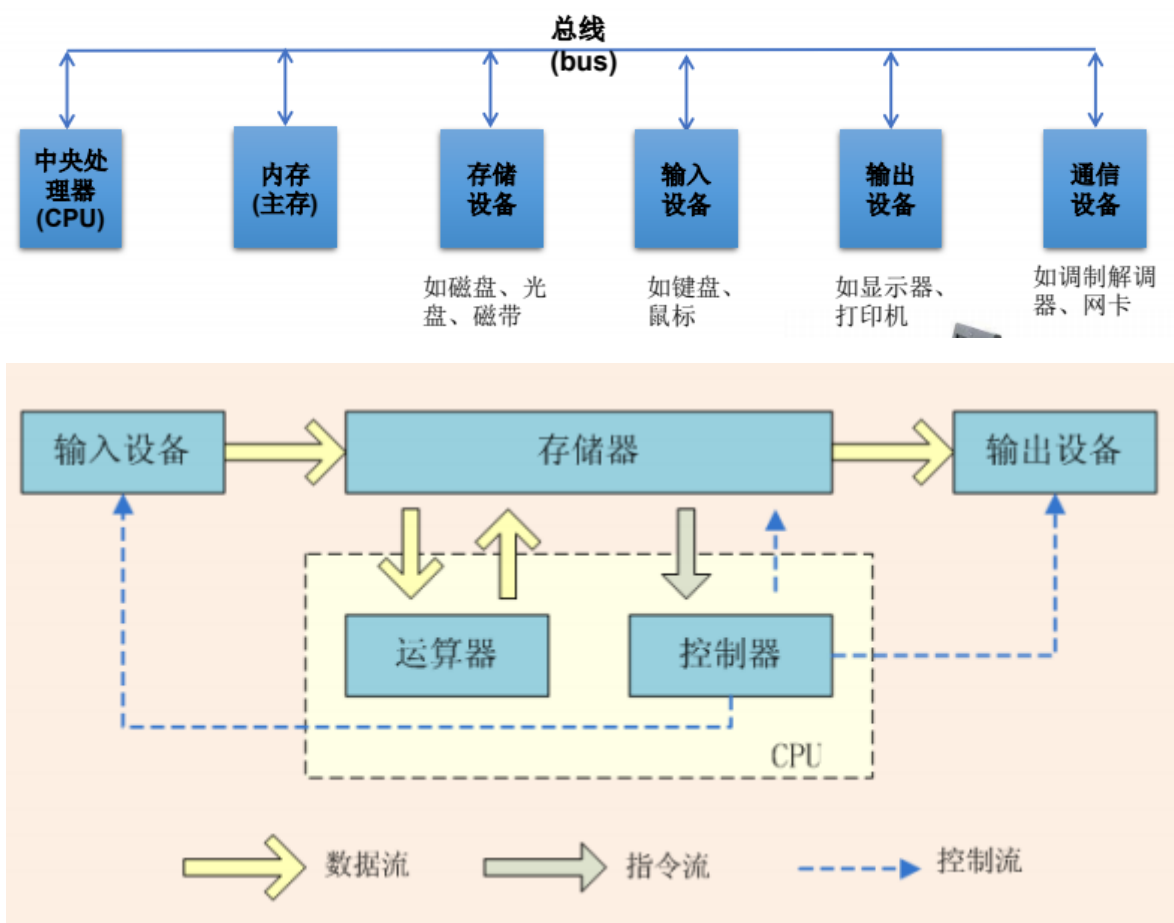
软件开发人员在称为程序设计语言的强大工具的帮助下创建软件。

### 如何选择该学习哪种程序设计语言？

- 程序设计语言有很多种，每种语言都是为了实现某个特定的目的而发明的。
- 你会困惑哪种语言是最好的。事实上，没有“最好”的语言。每种语言都有它的长处和短处。

- 经验丰富的程序员知道各种语言擅长的应用场景，因此，会尽可能的掌握各种不同的程序设计语言。
- 如果你掌握了一种编程语言，应该会更易上手其它的编程语言。关键是学习如何使用程序设计方法来解决实际问题。

## ✧ 计算机硬件介绍



**冯·诺依曼体系结构**是现代计算机的基础，现在大多计算机仍是冯·诺依曼计算机的组织结构，只是作了一些改进而已，并没有从根本上突破冯体系结构的束缚。冯·诺依曼也因此被人们称为“计算机之父”。

## 计算机硬件介绍-中央处理器

- 中央处理器(Central Processing Unit,CPU)是计算机的大脑。它从内存中获取指令，然后执行这些指令。
- 包括：控制单元(control unit)和算术/逻辑单元(arithmetic/logic unit)。
  - 控制单元：用于控制和协调其他组件的动作。

- 算术/逻辑单元：用于完成数值运算(+、-、\*、/)和逻辑运算(比较)。
- 每台计算机都有一个内部时钟，该时钟以固定速度发射电子脉冲。时钟速度越快，在给定的时间段内执行的指令就越多。速度的计量单位是赫兹(Hz)，1Hz相当于每秒1个脉冲。随着CPU速度不断提高，目前以千兆赫(GHz)来表述。
- 最初一个CPU只有一个核(core)。核是处理器中实现指令读取和执行的部分。一个多核CPU是一个具有两个或者更多独立核的组件。可提高CPU的处理能力。

## IT定律之计算机行业发展规律

- 摩尔定律(Moore's Law)
- 安迪-比尔定律(Andy and Bill's Law)
- 反摩尔定律(Reverse Moore's Law)

## 计算机硬件介绍-存储设备

- 内存中的信息在断电时会丢失。那我们可以考虑将程序和数据永久的保存在存储设备上。当计算机确实需要这些数据时，再移入内存，因为从内存中读取比从存储设备读取要快得多。
- 存储设备主要有以下三种：
  - 磁盘驱动器:每台计算机至少有一个硬盘驱动器。硬盘(hard disk)用于永久的保存数据和程序。
  - 光盘驱动器(CD和DVD)
    - CD的容量可达700MB。
    - DVD的容量可达4.7GB。
  - USB闪存驱动器
    - USB: Universal Serial Bus，通用串行总线。
    - 可以使用USB将打印机、数码相机、鼠标、外部硬盘驱动器连接到计算机上。
    - USB闪存驱动器很小，可用于存储和传输数据的设备。

## 计算机硬件介绍：内存

## 比特(bit)和字节(byte)

- 在讨论内存前，先清楚数据是如何存储在计算机中的。
- 计算机就是一系列的电路开关。每个开关存在两种状态：关(off)和开(on)。如果电路是开的，它的值是1。如果电路是关的，它的值是0。
- 一个0或者一个1存储为一个比特(bit)，是计算机中最小的存储单位。
- 计算机中是最基本的存储单元是字节(byte)。每个字节由8个比特构成。
- 计算机的存储能力是以字节和多字节来衡量的。如下：
  - 千字节(kilobyte,KB) = 1024B
  - 兆字节(megabyte,MB) = 1024KB
  - 千兆字节(gigabyte,GB) = 1024MB
  - 万亿字节(terabyte,TB) = 1024GB
- 内存(也叫 Random-Access Memory,RAM)：由一个有序的字节序列组成，用于存储程序及程序需要的数据。
- 一个程序和它的数据在被CPU执行前必须移到计算机的内存中。
- 每个字节都有一个唯一的地址。见右图。使用这个地址确定字节的位置，以便于存储和获取数据。
- 一个计算机具有的RAM越多，它的运行速度越快，但是此规律是有限制的。
- 内存与CPU一样，也构建在表面嵌有数百万晶体管的硅半导体芯片上。但内存芯片更简单、更低速、更便宜。
- 实测发现：内存存取数据的速度比硬盘的存取速度快10倍，在某些环境里，硬盘和内存之间的速度差距可能会更大。而CPU的速度比内存不知还要快多少倍。当我们把程序从硬盘放到内存以后，CPU就直接在内存运行程序，这样比CPU直接在硬盘运行程序就要快很多。
- 内存解决了一部分CPU运行过快，而硬盘数据存取太慢的问题。提高了我们的电脑的运行速度。内存就如同一条“高速车道”一般，数据由传输速度较慢的硬盘通过这条高速车道传送至CPU进行处理！
- 但内存是带电存储的(一旦断电数据就会消失)，而且容量有限，所以要长时间储存程序或数据就需要使用硬盘
- 内存在这里起了两个作用：

- ① 保存从硬盘读取的数据，提供给CPU使用
- ② 保存CPU的一些临时执行结果，以便CPU下次使用或保存到硬盘

## 计算机硬件介绍：输入和输出设备

- 常见的输入设备：键盘（keyboard）和鼠标（mouse）
- 常见的输出设备：显示器（monitor）和打印机（printer）
- 显示器屏幕分辨率：是指显示设备水平和垂直方向上显示的像素(px)数。
- 分辨率可以手工设置。
- 分辨率越高，图像越锐化、越清晰。

品牌	尺寸	像素	像素密度
华为mate 20 x	7.2	2244x1080像素	345ppi
华为mate 20	6.53	2244x1080像素	381ppi
华为mate 20 pro	6.39	3120x1440像素	538ppi
小米8 SE	5.88	2244x1080像素	423ppi
小米8	6.21	2244x1080像素	401ppi
苹果 iphone8	4.7	1334x750像素	326ppi
苹果 iphone8 plus	5.5	1920x1080像素	401ppi

计算公式：  $\text{像素密度} = \sqrt{[(\text{长度像素数})^2 + (\text{宽度像素数})^2]} / \text{屏幕尺寸}$

## 计算机硬件介绍：通信设备

- 计算机可以通过通信设备进行联网。
- 常见的设备有：
  - 拨号调制解调器：使用的是电话线，传输速度可达56 000bps(bps:每秒比特)
  - DSL（数字用户线）：使用的也是电话线，但传输速度比上面的快20倍
  - 电缆调制解调器：利用有线电视电缆进行数据传输，通常速度比DSL快。
  - 网络接口卡（NIC）：将计算机接入局域网（LAN）的设备。局域网通常用于大学、商业组织和政府组织。速度甚至可达1000Mbps

- 无线网络：在家庭、商业和学校中极其常见。计算机可通过无线适配器连接到局域网或internet上。

## ✧ 计算机发展史上的鼻祖

---

最近半个世纪以来，世界计算机科学界的重大进步，离不开图灵等人的理论奠基作用和多方面的开创性研究成果。图灵是当之无愧的计算机科学和人工智能之父。甚至认为，他在技术上的贡献及对未来世界的影响几乎可与牛顿、爱因斯坦等巨人比肩。

图灵论文中的“用有限的指令和有限的存储空间可算尽一切可算之物”理论让当时所有的科学家震惊

美国计算机学会（ACM）的年度“图灵奖”，自从1966年设立以来，一直是世界计算机科学领域的最高荣誉，相当于计算机科学界的诺贝尔奖。至今，中国人只有姚期智院士获该奖项。

20世纪最重要的数学家之一，在现代计算机、博弈论、核武器和生化武器等诸多领域内有杰出建树的最伟大的科学全才之一，被后人称为“计算机之父”和“博弈论之父”。

计算机基本工作原理是存储程序和程序控制，它是由世界著名数学家冯·诺依曼提出的。最简单的来说，冯诺依曼理论的要点是：数字计算机的数制采用二进制；计算机应该按照程序顺序执行。

同样有着“计算机之父”称号的冯·诺依曼的助手弗兰克尔在一封信中写到：“.....计算机的基本概念属于图灵。按照我的看法，冯·诺依曼的基本作用是使世界认识了由图灵引入的计算机基本概念.....”

根据冯诺依曼体系结构构成的计算机，必须具有如下功能：

- 把需要的程序和数据送至计算机中。
- 必须具有长期记忆程序、数据、中间结果及最终运算结果的能力。
- 能够完成各种算术、逻辑运算和数据传送等数据加工处理的能力。
- 能够根据需要控制程序走向，并能根据指令控制机器的各部件协调操作。
- 能够按照要求将处理结果输出给用户。

## ✧ 操作系统

---

操作系统(Operating System)是运行在计算机上的最重要的程序，它可以管理和控制计算机的活动。



- 硬件、操作系统、应用程序和用户之间的关系如右图。
- 操作系统的主要任务：
- 控制和监视系统的活动
- 分配和调配系统资源
- 调度操作

## ✧ 万维网

---

万维网（World Wide Web,www,环球信息网）常简称为Web,发明者蒂姆·伯纳斯·李。分为Web客户端和Web服务器程序。WWW可以让 Web客户端（常用浏览器）访问浏览Web服务器上的页面。是一个由许多互相链接的超文本组成的系统，通过互联网访问。在这个系统中，每个有用的事物，称为一样“资源”；并且由一个 全局“统一资源标识符”（URI）标识；这些资源通过 超文本传输协议（Hypertext Transfer Protocol）传送给用户，而后者通过点击链接来获得资源。

万维网是无数个网络站点和网页的集合，它们在一起构成了因特网Internet最主要的部分（因特网也包括电子邮件、Usenet以及新闻组）。它实际上是多媒体的集合，是由超级链接连接而成的。我们通常通过网络浏览器上网观看的，就是万维网的内容。





### 职级与职级界定——专业职级（1 / 2）

职等	职级	职级描述
Band10	资深专家	<ul style="list-style-type: none"> <li>是本技术/专业领域内公认的专家，并具有跨专业知识。</li> <li>负责解决本领域内的重大技术/专业问题，并对本专业领域内的方案进行把关制定本领域内的专业标准，并为其他相关领域提供建议与指导，负责专业模块的发展战略制定。</li> <li>能够代表公司与外部进行沟通或谈判，塑造公司形象；针对本领域发展战略性问题与内部高层沟通，沟通结果对本领域甚至整个公司均有影响。</li> <li>往往需要拥有13年及以上的相关工作经验。</li> </ul>
Band9	专家	<ul style="list-style-type: none"> <li>拥有几个相关领域的领先知识/某些学科的专家级知识，作为一些大型复杂项目/工作事项的负责人。</li> <li>能够使用创新的方法处理非常复杂的问题，制定有效解决办法，并带领团队执行。</li> <li>经常与内外部高层人员协商，协商沟通的结果可能对多个相关领域均有影响。</li> <li>往往需要拥有9 - 12年的相关工作经验。</li> </ul>
Band8	资深工程师/资深专员/资深**师	<ul style="list-style-type: none"> <li>作为某一领域的技术/专业带头人，负责具有较高挑战的项目/工作事项。</li> <li>能够运用出色的判断和分析能力，解决复杂的专业或技术问题。</li> <li>能运用娴熟的沟通技巧指导、影响和说服他人，包括内外部业务合作伙伴。</li> <li>往往需要拥有6年至8年的相关工作经验。</li> </ul>



### 职级与职级界定——专业职级（2 / 2）

职等	职级	职级描述
Band7	高级工程师/高级专员/高级**师	<ul style="list-style-type: none"> <li>有丰富的的工作经验，是项目组/团队中的重要成员，或者可以管理中、小型项目。能够为其他成员提供专业或者技术方面的指导。</li> <li>能够独立解决业务领域的比较复杂的问题，并能对复杂问题提出建议。</li> <li>能和团队成员或业务合作伙伴进行深层次的沟通，并运用沟通策略影响他人。</li> <li>往往需要拥有4 - 5年的相关工作经验。</li> </ul>
Band6	中级工程师/中级专员/中级**师	<ul style="list-style-type: none"> <li>有一定的工作经验，熟悉全部的工作环节，是某项具体业务的执行者。工作需要一般性督导，工作结果需要被检查。</li> <li>能够独立完成本领域内的专业工作，并对复杂问题能做出初步判断。</li> <li>具备一定的沟通技巧，能够有策略地传达自身观点，以帮助解决问题。</li> <li>往往拥有2 - 3年的相关工作经验。</li> </ul>
Band5	初级工程师/初级专员/初级**师	<ul style="list-style-type: none"> <li>简单的技术工作的执行者，工作有比较明确的指示和目标。需要紧密的督导，工作结果需要被检查。</li> <li>能够根据既定方法、程序或政策解决常规问题。</li> <li>能正确接收信息，并和工作伙伴交换信息，进行简单内容的沟通。</li> <li>往往拥有1年及以下的工作经验。</li> </ul>





## 职级与职级界定——管理职级

职等	管理职级	描述
Band10	资深总监	负责一个/多个区域的业务或某个大型的部门工作，推动所管辖业务或部门策略的执行，协调本业务线中的资源配置，在管理权限内自主决策。
		往往需要16年及以上的工作经验，7-8年团队管理经验。
Band9	总监	某部门的负责人或领导多个小型部门。对部门的短期和长期经营决策起着重要作用，负责制定本部门政策，指导和协调跨部门的活动，对业务部门或主要职能部门的目标的实现负有重大责任。
		往往需要13-15年相关工作经验，5-6年团队管理经验。
Band8	高级经理	某一小型部门或部门下子业务模块或子团队的负责人，制定本部门/团队的目标、政策和工作程序，负责部门/团队人员管理。协调本部门/团队内的资源调配及与其他相关部门/团队的关系，对日常工作决策和结果负责。
		往往需要拥有10-12年相关工作经验，3-4年团队管理经验。
	经理	某一小型部门或部门下子业务模块或子团队的负责人，制定本部门/团队的目标、政策和工作程序，负责部门/团队人员管理。协调本部门/团队内的资源调配及与其他相关部门/团队的关系，对日常工作决策和结果负责。
		往往需要拥有7-10年相关工作经验，1-2年团队管理经验。

## Java 概述

### ✧ Java 语言发展史

#### Java 语言

语言：人与人交流沟通的方式

计算机语言：人与计算机之间进行信息交流沟通的一种特殊语言

Java 语言是美国 Sun 公司（Stanford University Network）在 1995 年推出的计算机语言。

Java 之父：詹姆斯·高斯林（James Gosling）

#### Java 语言发展史

- 1995年，Sun公司发布Java语言
- 1996年，发布Java（1.0）
- 1997年，发布Java（1.1）
- 1998年，发布Java（1.2）
- 2000年，发布Java（1.3）
- 2002年，发布Java（1.4）
- 2004年，发布Java（5.0）
- 2006年，发布Java（6.0）
- 2009年，Oracle 甲骨文公司收购Sun公司
- 2011年，发布Java（7.0）
- 2014年，发布Java（8.0）
- 2017年9月，发布Java（9.0）
- 2018年3月，发布Java（10.0）
- 2018年9月，发布Java（11.0）

## ✧ Java 语言跨平台原理

---

### 平台

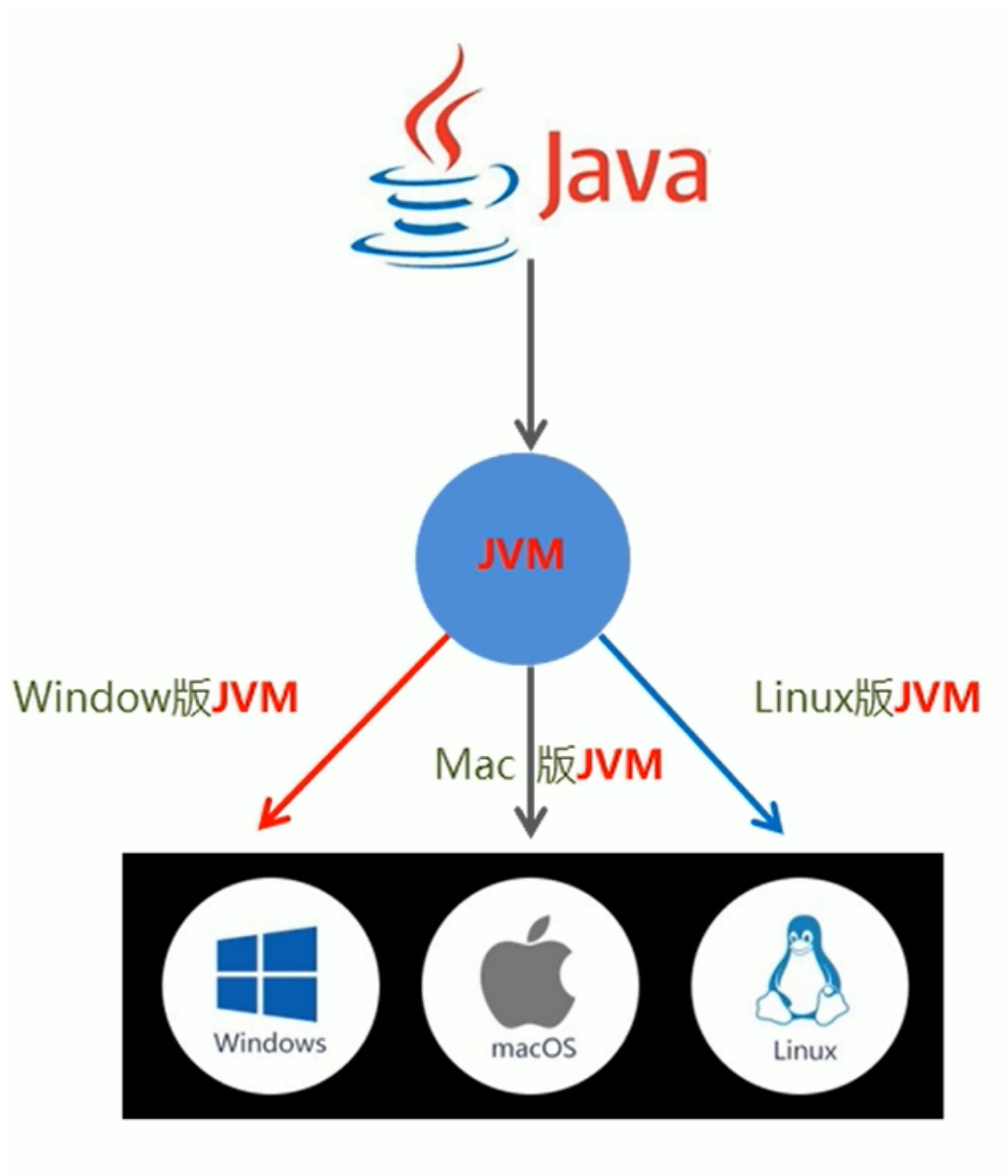
指的是操作系统

- Windows
- Mac
- Linux

### 跨平台

Java 程序可以在任意操作系统上运行

### 跨平台原理



总结：在需要运行 Java 应用程序的操作系统上，安装一个与操作系统对应的 Java 虚拟机（JVM Java Virtual Machine）即可。

## ✧ JRE 和 JDK

---

### ■ JRE（Java Runtime Environment）

是 Java 程序的运行时环境，包含 JVM 和运行时所需要的核心类库。

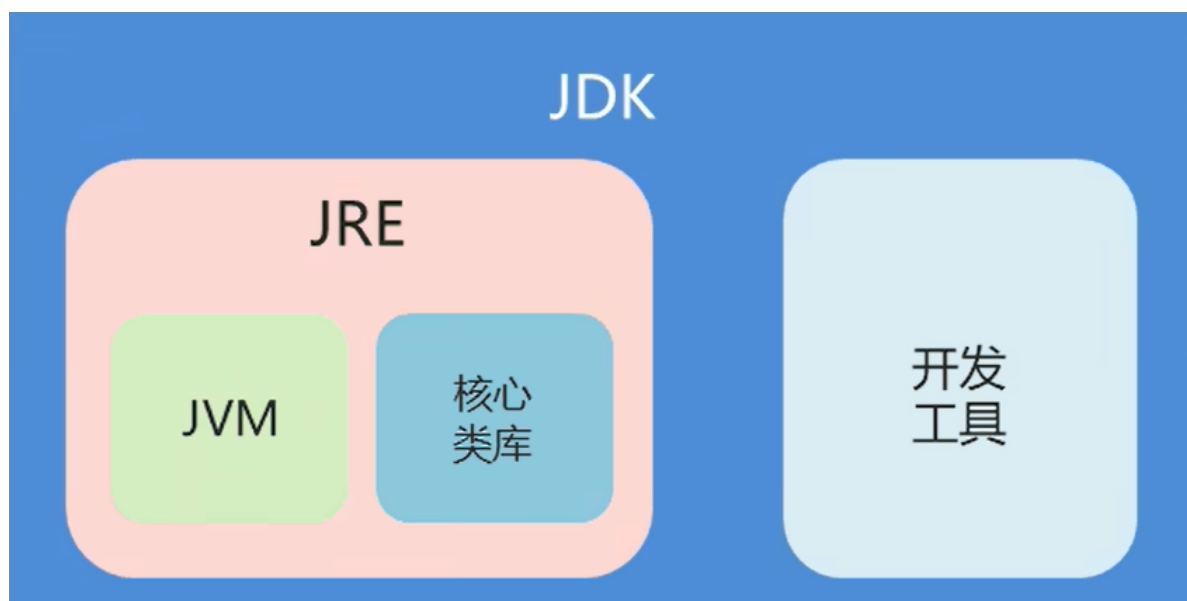
我们想要运行一个已有的 Java 程序，那么只需要安装 JRE 即可。

## JDK（Java Development Kit）

是 Java 程序开发工具包，包含 JRE 和开发人员使用的工具。

其中的开发工具：编译工具（`javac.exe`）和运行工具（`java.exe`）。

## JDK、JRE 和 JVM 的关系



## ✧ JDK 的下载和安装

---

傻瓜式安装，下一步即可。

建议：安装路径中不要包含中文和空格，所有的开发工具最好安装目录统一。

## JDK 的安装目录

目录名称	说明
bin	该路径下存放了 JDK 的各种工具命令。javac 和 java 就放在这个目录
conf	该路径下存放了 JDK 的相关配置文件
include	该路径下存放了一些平台特定的头文件
jmods	该路径下存放了 JDK 的各种模块
legal	该路径下存放了 JDK 各模块的授权文档
lib	该路径下存放了 JDK 工具的一些补充 IAR 包

其余文件为说明性文档。

## 第一个程序

### ✧ 常用 DOS 命令

---

#### 打开命令提示符窗口

- 1 按下 win+R
- 2 输入cmd
- 3 按下回车键

#### 常用命令



操作	说明
盘符名称	盘符切换。E:+回车，表示切换到E盘
dir	查看当前路径下的内容
cd 目录	进入单级目录
cd ...	回退到上一级目录
cd 目录1\目录2...	进入多级目录
cd \	回退到盘符目录
cls	清屏
exit	退出命令提示符窗口

## ✧ Path 环境变量的配置

### 为什么要配置 Path 环境变量

开发 Java 程序，需要使用 JDK 提供的开发工具，而这些开发工具在 JDK 安装目录的 bin 目录下。

为了在开发 Java 程序的时候，能够方便的使用 javac 和 java 这些命令，我们需要配置 Path 环境变量。

在系统变量中新建一个 JAVA\_HOME

1 D:\java\jdk1.8.0\_341

在 Path 中新建

1 %JAVA\_HOME%\bin

i

提示：如果命令提示符窗口是配置环境变量前打开的，需要关闭该窗口，重新打开一个窗口测试。

# ✧ HelloWorld 案例

---

## Java 程序开发运行流程

开发 Java 程序，需要三个步骤：编写程序、编译程序、运行程序

## HelloWorld 案例的编写

```
1 public class HelloWorld {  
2     public static void main (String[] args) {  
3         System.out.println("HelloWorld")  
4     }  
5 }
```

# 基础语法

## ✧ 注释

---

### 注释概述

- 注释是指在程序指定位置添加的说明性信息
- 注释不参与程序运行，仅起到说明作用

### 注释分类

- 单行注释

```
1 //注释信息
```

- 多行注释

```
1 /* 注释信息 */
```

```
1 /** 注释信息 */
```

## ✧ 关键字

---

### 关键字概述

关键字：被 Java 语言赋予了 特定含义的单词。

### 关键字的特点

- 1 关键字的字母 全部小写
- 2 常见的代码编辑器，针对关键字有特殊的颜色标记，非常直观

## ✧ 常量

---

### 常量概述

常量：在程序运行过程中，其值不可以发生改变的量

### 常量分类

常量类型	说明	举例
字符串常量	用双引号括起来的内容	“HelloWord”, "黑马程序员”
整数常量	不带小数的数字	666, -88
小数常量	带小数的数字	13.14, -5.21
字符常量	用单引号括起来的内容	‘A’, ‘0’, ‘我’
布尔常量	布尔值，表示真假	true, false
空常量	一个特殊的值，空值	值是null

## ✧ 数据类型

### 计算机存储单元

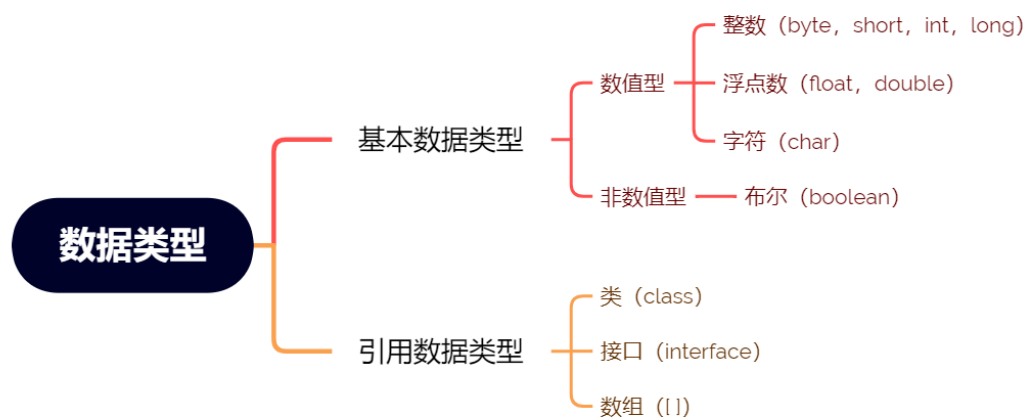
我们知道计算机是可以用来存放数据的，但是无论是内存还是硬盘，计算机存储设备的最小单元叫“**位（bit）**”，我们又称之为“**比特位**”，通常用小写字母“**b**”表示。而计算机中最小的存储单元叫“**字节（byte）**”，通常用大写字母“**B**”表示，字节是由连续的8个位组成的。

除了字节外还有一些常用的存储单位：

- 1B（字节）= 8bit
- 1KB = 1024B
- 1MB = 1024KB
- 1GB = 1024MB
- 1TB = 1024GB

### 数据类型

Java 语言是强类型语言，对于每一种数据都给出了明确的数据类型，不同的**数据类型**也分配了不同的**内存空间**，所以它们表示的**数据大小**也是不一样的。



## 数据类型内存占用和取值范围

数据类型	关键字	内存占用	取值范围
整数	byte	1	-128~127
整数	short	2	-32768~32767
整数	int (默认)	4	-2的31次方到2的31次方-1
整数	long	8	-2的63次方到2的63次方-1
浮点数	float	4	负数: -3.042823E+38到-1.401298E-45 正数: 1.401298E-45到3.042823E+38
浮点数	double (默认)	8	负数: -1.797693E+308到-4.9000000E-324 正数: 4.9000000E-324到1.797693E+308
字符	char	2	0~65535
布尔	boolean	1	true, false



## 变量概述

变量：在程序运行过程中，其值可以发生改变的量。

## 变量定义

- 格式：数据类型 变量名 = 变量值;
- 范例：

```
1 int a = 10;
```

## 变量的使用

变量的使用：取值和修改值

- 取值格式：变量名
- 范例：

```
1 a
```

- 修改值格式：变量名 = 变量值
- 范例：

```
1 a = 20;
```

## 变量使用的注意事项

- 1 名字不能重复
- 2 变量未赋值，不能使用
- 3 long类型的变量定义的时候，为了防止整数过大，后面要加L
- 4 float类型的变量定义的时候，为了防止类型不兼容，后面要加F

## ✧ 标识符

### 标识符概述

标识符：给类、方法、变量等起名字的符号。

## 标识符定义规则

- 1 由 数字、字母、下划线(\_)和美元符(\$) 组成
- 2 不能以数字开头
- 3 不能是关键字
- 4 区分大小写

## 常见命名约定

**i** 小驼峰命名法：-----> 方法、变量

- 1 标识符是一个单词的时候，首字母小写，如name
- 2 标识符由多个单词组成的时候，第一个单词首字母小写，其他单词首字母大写，如firstName

**i** 大驼峰命名法：-----> 类

- 1 标识符是一个单词的时候，首字母大写，如Student
- 2 标识符由多个单词组成的时候，每个单词首字母大写，如GoodStudent

## ✧ 类型转换

### 类型转换分类

- 自动类型转换
- 强制类型转换

### 自动类型转换

把一个表示数据 范围小的数值 或者 变量 赋值给另一个表示 数据范围大的变量。

范例：

```
1 double d = 10;
```

表示数据范围从小到大图：



## 强制类型转换

把一个表示 **数据范围大的数值** 或者 **变量** 赋值给另一个表示数据 **范围小的变量**。

- 格式：目标数据类型 变量名 = (目标数据类型)变量名;
- 范例：

```
1 int k = (int)88.88;
```

**i** 不建议使用强制类型转换，会有数据的损失。

# 运算符

## ✧ 算数运算符

### 运算符和表达式

- 运算符：对常量或者变量进行操作的 **符号**
- 表达式：用 **运算符** 把常量或者变量连接起来 **符合java语法** 的式子就可以称为表达式。
- 不同运算符连接的表达式体现的是不同类型的表达式。

## 算数运算符

符号	作用	说明
+	加	将两个数据相加求和
-	减	将两个数据相减求差
*	乘	将两个数据相乘求积
/	除	将两个数据相除求商
%	取余	将两个数据相除取余数

除法得到的是商，取余得到的是余数。



整数相除只能得到整数，要想得到小数，必须有浮点数的参与。

## 字符的“+”操作

拿字符在计算机底层 **对应的数值** 来进行计算。

- 1 'A' → 65 A-Z是连续的
- 2 'a' → 97 a-z是连续的
- 3 '0' → 48 0-9是连续的

算术表达式中 **包含多个基本数据类型** 的值的时候，整个算术表达式的 **类型** 会 **自动进行提升**。



提升规则：

- 1 byte类型，short类型和char类型将会被提升到int类型
- 2 整个表达式的类型会自动提升到表达式中最高等级操作数同样的类型
- 3 等级顺序：byte，short，char----> int ----> long ----> float ----> double

## 字符串的“+”操作

- 当‘+’操作中出现字符串时，这个‘+’是 **字符串连接符**，而不是算数运算符。

- 当‘+’操作中，如果出现了字符串，就是连接运算符，否则就是算术运算符。当连续进行‘+’操作时，从左到右逐个执行。

## ✧ 赋值运算符

符号	作用	说明
=	赋值	a=10，将10赋值给变量a
+=	加后赋值	a+=b，将a+b的值赋给a
-=	减后赋值	a-=b，将a-b的值赋给a
*=	乘后赋值	a=b，将ab的值赋给a
/=	除后赋值	a/=b，将a/b的值赋给a
%=	取余后赋值	a%=b，将a%b的值赋给a

**i** 注意：扩展的赋值运算符底层隐含了强制类型转换。

## ✧ 自增自减运算符

符号	作用	说明
++	自增	变量的值加1
--	自减	变量的值减1

```
1 // 单独使用
2 int i = 10;
3 i++; // 11
4 ++i; // 12
```

```
1 // 参与操作使用
2 int j = i++; // 先赋值再自增
3 int k = ++i; // 先自增再赋值
```



注意事项:

- ① ++和--既可以放在变量的后边,也可以放在变量的前边
- ② 单独使用的时候,++和--无论是放在变量的前边还是后边,结果都是一样的
- ③ 参与操作的时候,如果放在变量的后边,先拿变量参与操作,后拿变量做++或者--
- ④ 参与操作的时候,如果放在变量的前边,先拿变量做++或者--,后拿变量参与操作

最常见的用法: 单独使用

## ✧ 关系运算符

符号	说明
==	$a==b$ , 判断a和b的值是否相等, 成立则为true, 不成立为false
!=	$a!=b$ , 判断a和b的值是否不相等, 成立则为true, 不成立为false
>	$a>b$ , 判断a是否大于b, 成立则为true, 不成立为false
>=	$a>=b$ , 判断a是否大于等于b, 成立则为true, 不成立为false
<	$a<b$ , 判断a是否小于b, 成立则为true, 不成立为false
<=	$a<=b$ , 判断a是否小于等于b, 成立则为true, 不成立为false

注意事项:

- ① 关系运算符的结果都是布尔类型, 要么是true, 要么是false  
千万不要把“==”误写成“=”

## ✧ 逻辑运算符

## 逻辑运算符概述

逻辑运算符，是用来 **连接关系表达式** 的运算符。

逻辑运算符也可以直接 **连接布尔类型的常量或者变量**。

## 逻辑运算符

符号	作用	说明
&	逻辑与	$a \& b$ ，a和b都是true，结果为true，否则为false
	逻辑或	$a   b$ ，a和b都是false，结果为false，否则为true
^	逻辑异或	$a \wedge b$ ，a和b结果不同为true，相同为false
!	逻辑非	$!a$ ，结果和a的结果相反

## 短路逻辑运算符

符号	作用	说明
&&	短路与	$a \& \& b$ ，作用和&相同，但是有短路效果（有false则false）
	短路或	$a    b$ ，作用和 相同，但是有短路效果（有true则true）

注意事项：

逻辑与**&**，无论左边真假，右边都要执行

**i** 短路与**&&**，如果左边为真，右边执行；如果左边为假，右边不执行

逻辑或**|**，无论左边真假，右边都要执行

短路或**||**，如果左边为假，右边执行；如果左边为真，右边不执行

最常用的逻辑运算符：**&&，||，!**

## ✧ 三元运算符

- 格式：关系表达式 ? 表达式1 : 表达式2;
- 范例：

```
1 a > b ? a : b;
```

- 计算规则：
  - 首先计算 关系表达式的值
  - 如果值为true，表达式1的值就是运算结果
  - 如果值为false，表达式2的值就是运算结果

## 案例：两只老虎

需求：动物园里有两只老虎，已知两只老虎的体重分别为180kg、200kg，请用程序实现判断两只老虎的体重是否相同。

```
1 int weight1 = 180;
2 int weight2 = 200;
3 boolean b = weight1 == weight2 ? true : false;
4 System.out.println("b:" + b);
```

## 案例：三个和尚

需求：一座寺庙里住着三个和尚，已知他们的身高分别为150cm、210cm、165cm，请用程序实现获取这三个和尚的最高身高。

```
1 int height1 = 150;
2 int height2 = 210;
3 int height3 = 165;
4 int tempHeight = height1 > height2 ? height1 : height2;
5 int maxHeight = tempHeight > height3 ? tempHeight : height3;
6 System.out.println("maxHeight:" + maxHeight);
```

# 数据输入

## ✧ Scanner 使用的基本步骤

---

### ① 导包

```
1 // 导包的动作必须出现在类定义的上边
2 import java.util.Scanner;
```

### ① 创建对象

```
1 // 格式中, 只有sc是变量名, 可以变, 其他的都不允许变
2 Scanner sc = new Scanner(System.in);
```

### ① 接收数据

```
1 // 格式中, 只有i是变量名, 可以变, 其他的都不允许变
2 int i = sc.nextInt();
```

## ✧ 案例：三个和尚

---

需求：一个寺庙里住着三个和尚，他们的身高必须经过测量得出，请用程序实现获取这三个和尚的最高身高。

```
1 Scanner sc = new Scanner(System.in);
2 System.out.println("请输入第一个和尚的身高: ");
3 int height1 = sc.nextInt();
4 System.out.println("请输入第二个和尚的身高: ");
5 int height2 = sc.nextInt();
6 System.out.println("请输入第三个和尚的身高: ");
7 int height3 = sc.nextInt();
8 int tempHeight = height1 > height2 ? height1 : height2;
9 int maxHeight = tempHeight > height3 ? tempHeight : height3;
10 System.out.println("这三个和尚中身高最高的是:" + maxHeight + "cm");
```

## 分支语句

## ✧ 流程控制

---

### 流程控制语句分类

- 顺序结构
- 分支结构(if,switch)
- 循环结构(for,while,do...while)

### 顺序结构

顺序结构是程序中最基本的流程控制，没有特定的语法结构，按照代码的先后顺序，依次执行。

程序中大多数的代码都是这样执行的。

## ✧ if 语句

---

### if 语句格式1

```
1  格式：  
2  if（关系表达式） {  
3      语句体  
4  }
```

#### 执行流程：

- ① 首先计算关系表达式的值
- ② 如果关系表达式的值为true就执行语句体
- ③ 如果关系表达式的值为false就不执行语句体
- ④ 继续执行后面的语句内容

### if 语句格式2



```
1  格式:
2  if (关系表达式) {
3      语句体1;
4  } else {
5      语句体2;
6  }
```

#### 执行流程:

- 1 首先计算关系表达式的值
- 2 如果关系表达式的值为true就执行语句体1
- 3 如果关系表达式的值为false就执行语句体2
- 4 继续执行后面的语句内容

### 案例：奇偶数

需求：仍以给出一个整数，请用程序实现判断该整数是奇数还是偶数，并在控制台输出该整数是奇数还是偶数

```
1  Scanner sc = new Scanner(System.in);
2  System.out.println("请输入一个整数: ");
3  int a = sc.nextInt();
4  if (a % 2 == 0) {
5      System.out.println(a + "是偶数");
6  } else {
7      System.out.println(a + "是奇数");
8  }
```

### if 语句格式3

```
1  格式:
2  if (关系表达式1) {
3      语句体1;
4  } else if (关系表达式2) {
5      语句体2;
6  }
7  ...
8  else {
9      语句体n+1;
10 }
```

**i** 执行流程：

- 1 首先计算关系表达式1的值
- 2 如果值为true就执行语句体1；如果值为false就计算关系表达式2的值
- 3 如果值为true就执行语句体2；如果值为false就计算关系表达式3的值
- 4 ...
- 5 如果没有任何关系表达式的值为true，就执行语句体n+1

**i** 需求：键盘录入一个星期数（1，2，。。。。。。，7），输出对应的星期一，星期二，。。。。，星期日

```
1 Scanner sc = new Scanner(System.in);
2 System.out.println("请输入一个星期数 (1~7) : ");
3 int a = sc.nextInt();
4 if (a == 1) {
5     System.out.println("星期一");
6 } else if (a == 2) {
7     System.out.println("星期二");
8 } else if (a == 3) {
9     System.out.println("星期三");
10 } else if (a == 4) {
11     System.out.println("星期四");
12 } else if (a == 5) {
13     System.out.println("星期五");
14 } else if (a == 6) {
15     System.out.println("星期六");
16 } else {
17     System.out.println("星期日");
18 }
19 System.out.println("结束");
```

### 案例：考试奖励

需求：小明快要期末考试了，小明的爸爸对他说，会根据他不同的考试成绩，送他不同的礼物，假如你可以控制小明的得分，请用程序实现小明到底该获得什么样的礼物，并在控制台输出。

奖励:


95~100 山地自行车一辆

90~94 游乐场玩一次

80~89 变形金刚玩具一个

80以下 胖揍一顿

```
1 Scanner sc = new Scanner(System.in);
2 System.out.println("请输入一个分数: ");
3 int score = sc.nextInt();
4 if (score > 100 || score < 0) {
5     System.out.println("您输入的分数有误");
6 } else if (score ≥ 95 && score ≤ 100) {
7     System.out.println("山地自行车一辆");
8 } else if (score ≥ 90 && score ≤ 94) {
9     System.out.println("游乐场玩一次");
10 } else if (score ≥ 80 && score ≤ 89) {
11     System.out.println("变形金刚玩具一个");
12 } else {
13     System.out.println("胖揍一顿");
14 }
```

 数据测试: 正确数据、边界数据、错误数据

## ✧ switch 语句

### switch 语句格式

```
1 格式:
2 switch (表达式) {
3     case值1:
4         语句体1;
5         break;
6     case值2:
7         语句体2;
8         break;
9     ...
10 default:
11     语句体n+1;
12     [break;]
13 }
```

### 格式说明:

- 1 表达式: 取值为byte、short、int、char, JDK5以后可以是枚举, JDK7以后可以是string
- 2 case: 后面跟的是要和表达式进行比较的值
- 3 break: 表示中断、结束的意思, 用来结束switch语句
- 4 default: 表示所有情况都不匹配的时候, 就执行该处的内容, 和if语句的else相似

### 执行流程:

- 1 首先计算表达式的值
- 2 依次和case后面的值进行比较, 如果有对应的值, 就会执行相应的语句, 在执行的过程中, 遇到break就会结束
- 3 如果所有的case后面的值和表达式的值都不匹配, 就会执行default里面的语句体, 然后程序结束掉

```
1 //需求: 键盘录入一个星期数 (1~7), 控制台输出对应的星期一、星期二、。。。、星期日
2 System.out.println("请输入一个星期数 (1~7) : ");
3 Scanner sc = new Scanner(System.in);
4 int week = sc.nextInt();
5 switch (week) {
6     case 1:
7         System.out.println("星期一");
8         break;
9     case 2:
10        System.out.println("星期二");
11        break;
12    case 3:
13        System.out.println("星期三");
14        break;
15    case 4:
16        System.out.println("星期四");
17        break;
18    case 5:
19        System.out.println("星期五");
20        break;
21    case 6:
22        System.out.println("星期六");
```

```

23         break;
24     case 7:
25         System.out.println("星期日");
26         break;
27     default:
28         System.out.println("您输入的星期数有误");
29 }

```

### 案例：春夏秋冬

需求：一年有12个月，分属于春夏秋冬四个季节，键盘录入一个月份，请用程序实现判断该月份属于哪个季节，并输出

春：3、4、5

夏：6、7、8

秋：9、10、11

冬：12、1、2

```

1  Scanner sc = new Scanner(System.in);
2  System.out.println("请输入一个月份：");
3  int month = sc.nextInt();
4  switch (month) {
5      case 1:
6      case 2:
7      case 12:
8          System.out.println("冬季");
9          break;
10     case 3:
11     case 4:
12     case 5:
13         System.out.println("春季");
14         break;
15     case 6:
16     case 7:
17     case 8:
18         System.out.println("夏季");
19         break;
20     case 9:
21     case 10:
22     case 11:
23         System.out.println("秋季");
24         break;

```

```
25 default:
26     System.out.println("您输入的月份有误");
27 }
```

注意事项：在switch语句中，如果case控制的语句体后面不写break，将会出现穿透现象，在不判断下一个case值的情况下，向下运行，直到遇到break，或者整体switch语句结束

## 循环语句

### ✧ for 循环语句

#### 循环结构

**i** 特征：

- 重复做某件事情
- 具有明确的开始和停止标志

**i** 循环结构的组成：

- 初始化语句：用于表示循环开始时的起始状态，简单说就是循环开始的时候什么样
- 条件判断语句：用于表示循环反复执行的条件，简单说就是判断循环是否能一直执行下去
- 循环体语句：用于表示循环反复执行的内容，简单说就是循环反复执行的事情
- 条件控制语句：用于表示循环执行中每次变化的内容，简单说就是控制循环是否能执行下去

**i** 循环结构对应的语法：

- 初始化语句：这里可以是一条或者多条语句，这些语句可以完成一些初始化操作
- 条件判断语句：这里使用一个结果值为`boolean`类型的表达式，这个表达式能决定是否执行循环体。例如：`a < 3`
- 循环体语句：这里可以是任意语句，这些语句将反复执行
- 条件控制语句：这里通常是使用一条语句来改变变量的值，从而达到控制循环是否向下执行的效果。常见`i++`,`i--`这样的操作

## for 循环语句格式

```
1  格式：  
2  for (初始化语句;条件判断语句;条件控制语句) {  
3      循环体语句;  
4  }
```

### 执行流程：

- 1 执行初始化语句
- 2 执行条件判断语句，判断其结果是`true`还是`false`
  - 如果是`false`，循环结束
  - 如果是`true`，继续执行
- 3 执行循环体语句
- 4 执行条件控制语句
- 5 回到2继续

```
1  //需求：在控制台输出5次"HelloWorld"  
2  for (int i = 1; i ≤ 5; i++) {  
3      System.out.println("HelloWorld");  
4  }
```

## 案例：输出数据

需求：在控制台输出1-5和5-1的数据

```

1  for (int i = 1; i ≤ 5; i++) {
2      System.out.println(i);
3  }
4  System.out.println("-----");
5  for (int i = 5; i ≥ 1; i--) {
6      System.out.println(i);
7  }

```

## 案例：求和

需求：求1-5之间的数据和，并把求和结果在控制台输出

```

1  int sum = 0;
2  for (int i = 1; i ≤ 5; i++) {
3      sum += i; //sum = sum + i
4  }
5  System.out.println("1-5之间的数据和是: " + sum);

```

## 案例：求偶数和

需求：求1-100之间的偶数和，并把求和结果在控制台输出

```

1  int sum = 0;
2  for (int i = 1; i ≤ 100; i++) {
3      if (i % 2 == 0) {
4          sum += i;
5      }
6  }
7  System.out.println("1-100之间的偶数和为: " + sum);

```

## 案例：水仙花

需求：在控制台输出所有的水仙花数

- ① 水仙花数是一个三位数
- ② 水仙花数的个位、十位、百位的数字立方和等于原数



```

1  for (int i = 100; i ≤ 999; i++) {
2      int a = i % 10; //个位
3      int b = i / 10 % 10; //十位
4      int c = i / 10 / 10 % 10; //百位
5      if (a * a * a + b * b * b + c * c * c == i) {
6          System.out.println(i);
7      }
8  }

```

任意数字的指定位上的数值如何求？

- ❶ 先使用整除操作将要求的数字移动到个位上，再使用取余操作取出最后一位上的值。

## 案例：统计水仙花数

需求：统计水仙花数一共有多少个，并在控制台输出个数

```

1  int count = 0;
2  for (int i = 100; i ≤ 999; i++) {
3      int a = i % 10;
4      int b = i / 10 % 10;
5      int c = i / 10 / 10 % 10;
6      if (a * a * a + b * b * b + c * c * c == i) {
7          count++;
8      }
9  }
10 System.out.println("水仙花数共有: " + count + "个");

```

## ✧ while 循环语句

### while 循环语句格式

```

1  基本格式：
2  while (条件判断语句) {
3      循环体语句；
4  }

```

```
1 完整格式：
2 初始化语句；
3 while（条件判断语句） {
4     循环体语句；
5     条件控制语句；
6 }
```

### 执行流程：

- ① 执行初始化语句
- ② 执行条件判断语句，判断其结果是true还是false
  - 如果是false，循环结束
  - 如果是true，继续执行
- ③ 执行循环体语句
- ④ 执行条件控制语句
- ⑤ 回到2继续

```
1 //需求：在控制台输出5次HelloWorld
2 int i = 1;
3 while (i ≤ 5) {
4     System.out.println("HelloWorld");
5     i++;
6 }
```

## 案例：珠穆朗玛峰

需求：世界最高山峰珠穆朗玛峰（8844.43米=8844430毫米），假如我有一张足够大的纸，它的厚度是0.1毫米。请问，我折叠多少次，可以折成珠穆朗玛峰的高度？

```
1 int count = 0;
2 double paper = 0.1;
3 while (paper ≤ 8844430) {
4     paper *= 2;
5     count++;
6 }
7 System.out.println("需要折叠: " + count + "次");
```

## do...while 循环语句

## do...while 循环语句格式

```
1 基本格式:  
2  do {  
3      循环体语句;  
4  } while (条件判断语句);
```

```
1 完整格式:  
2  初始换语句;  
3  do {  
4      循环体语句;  
5      条件控制语句;  
6  } while (条件判断语句);
```

### 执行流程:

- ① 执行初始化语句
- ② 执行循环体语句
- ③ 执行条件控制语句
- ④ 执行条件判断语句，看其结果是true还是false
  - 如果是false，循环结束
  - 如果是true，继续执行
- ⑤ 回到2继续

```
1  int i = 1;  
2  do {  
3      System.out.println("HelloWorld");  
4      i++;  
5  } while (i <= 5);
```

## ✧ 三种循环的区别

### 三种循环的区别:

- for循环和while循环先判断条件是否成立，然后决定是否执行循环体（先判断后执行）

- `do...while`循环先执行一次循环体，然后判断条件是否成立，是否继续执行循环体（先执行后判断）

#### **i** `for`和`while`的区别：

- 条件控制语句所控制的自增变量，因为归属`for`循环的语法结构中，在`for`循环结束后，就不能再次被访问到了
- 条件控制语句所控制的自增变量，对于`while`循环来说不归属其语法结构中，在`while`循环结束后，该变量还可以继续使用

#### **i** 死循环格式：

```
1  for (; ;) {}
2  while (true) {}
3  do {} while (true);
```

- i** `while` 的死循环格式是最常用的  
在命令提示符窗口中可以按下`ctrl+c`强制停止死循环

## ✧ 跳转控制语句

- `continue` 跳过某循环体执行，继续下一次的执行 使用是基于条件控制的
- `break` 终止循环体执行，结束当前整个循环 使用是基于条件控制的

```
1  for (int i = 1; i ≤ 5; i++) {
2      if (i % 2 == 0) {
3          continue;
4      }
5      System.out.println(i);
6  }
7  System.out.println("-----");
8  for (int j = 1; j ≤ 5; j++) {
9      if (j % 2 == 0) {
10         break;
11     }
12     System.out.println(j);
13 }
```

## ✧ 循环嵌套

---

循环语句中包含循环语句称为循环嵌套

```
1 //需求：在控制台输出一天的小时和分钟
2 for (int hour = 0; hour < 24; hour++) {
3     for (int minute = 0; minute < 60; minute++) {
4         System.out.println(hour + "时" + minute + "分");
5     }
6     System.out.println("-----");
7 }
```

## ✧ Random

---

作用：用于产生一个随机数

**i** 使用步骤：

① 导包

```
1 import java.util.Random;
2 导包动作必须出现类定义的上
```

① 创建对象

```
1 Random r = new Random();
2 上面这个格式里面，r是变量名，可以变，其他的都不允许变
```

① 获取随机数

```
1 int number = r.nextInt(10); // 获取数据的范围[0,10)，包括0但不包括10
2 上面这个格式里面，number是变量名，可以变，数字10可以变，其他的都不允许变
```

```

1 Random r = new Random();
2 //用循环获取10个随机数
3 for (int i=0;i<10;i++) {
4     int number = r.nextInt(10);
5     System.out.println(number);
6 }
7
8 //需求: 获取一个1-100之间的随机数
9 int x = r.nextInt(100) + 1;
10 System.out.println(x);

```

## 案例：猜数字

需求：程序自动生成一个1-100之间的数字，使用程序实现猜出这个数字是多少？

**i** 当猜错的时候，根据不同情况给出相应提示：

- 如果猜的数字比真实数字大，提示你猜的数据大了
- 如果猜的数字比真实数字小，提示你猜的数据小了
- 如果猜的数字与真实数字相等，提示恭喜你猜中了

```

1 Random r = new Random();
2 int number = r.nextInt(100) + 1;
3 while(true) {
4     Scanner sc = new Scanner(System.in);
5     System.out.println("请输入你要猜的数字: ");
6     int guessNumber = sc.nextInt();
7     if (guessNumber > number) {
8         System.out.println("你猜的数字" + guessNumber + "大了");
9     } else if (guessNumber < number) {
10        System.out.println("你猜的数字" + guessNumber + "小了");
11    } else {
12        System.out.println("恭喜你猜中了!");
13        break;
14    }
15 }

```

# 数组

## ✧ 数组定义格式

---

数组是一种用于存储 **多个相同类型** 数据的存储模型

- 格式一：数据类型 [] 变量名
- 定义了一个int类型的数组，数组名是arr
- 范例：

```
1 int [] arr
```

- 格式二：数据类型 变量名 []
- 定义了一个int类型的变量，变量名是arr数组
- 范例：

```
1 int arr []
```

**i** 推荐使用格式一 定义数组。

## ✧ 数组初始化之动态初始化

---

java中的数组必须先初始化，然后才能使用

所谓初始化，就是为数组中的数组元素分配内存空间，并为每个数组元素赋值

动态初始化：初始化时只指定数组长度，由系统为数组分配初始值

- 格式：数据类型 [] 变量名 = 数据类型[数组长度];
- 范例：

```
1 int [] arr = new int[3];
```

## ✧ 数组元素访问

---

- 数组变量访问方式

- 格式： 数组名
- 数组内部保存的数据的访问方式
- 格式： 数组名[索引]
- 索引是数组中数据的编号方式
- 作用：索引用于访问数组中的数据使用，数组名[索引]等同于变量名，是一种特殊的变量名
- 特征：
  - ① 索引从0开始
  - ② 索引是连续的
  - ③ 索引逐一增加，每次加1

```
1 int[] arr = new int[3];
2 //输出数组名
3 System.out.println(arr);
4 //输出数组中的元素
5 System.out.println(arr[0]);
6 System.out.println(arr[1]);
7 System.out.println(arr[2]);
```

## ✧ 内存分配

---

### java中的内存分配

java程序在运行时，需要在内存中分配空间，为了提高运算效率，就对空间进行了不同区域的划分，因为每一片区域都有特定的处理数据方式和内存管理方式。

数组在初始化时，会为内存空间添加初始值：

- 整数：默认值0
- 浮点数：默认值0.0
- 布尔值：默认值false
- 字符：默认值是空字符
- 引用数据类型：默认值是null
- ① 栈内存：存储局部变量



- ② 局部变量：定义在方法中的变量，如arr。使用完毕，立即消失。
- ③ 堆内存：存储new出来的内容（实体，对象）
- ④ 每一个new出来的东西都有一个地址值，使用完毕，会在垃圾回收器空闲时被回收

## ✳ 数组初始化之静态初始化

---

静态初始化：初始化时指定每个数组元素的初始值，由系统决定数组长度

- 格式：数据类型[] 变量名 = new 数据类型[] {数据1, 数据2, 数据3, .....};

```
1 int[] arr = new int[] {1,2,3,.....};
```

- 简化格式：数据类型[] 变量名 = {数据1, 数据2, 数据3, .....};

```
1 int[] arr = [1,2,3,.....];
```

```
1 int[] arr = {1, 2, 3};
2 System.out.println(arr);
3 System.out.println(arr[0]);
4 System.out.println(arr[1]);
5 System.out.println(arr[2]);
```

## ✳ 数组操作的两个常见小问题

---

- 索引越界：访问了数组中不存在的索引对应的元素，造成索引越界问题
- 空指针异常：访问的数组已经不再指向堆内存的数据，造成空指针异常

```
1 int[] arr = new int[3];
2 System.out.println(arr[3]); // 索引越界
3
4 int[] arr2 = new int[3];
5 arr2 = null;
6 System.out.println(arr2[0]); // 空指针异常
```

## ✳ 数组常见操作

---

## 数组遍历

```
1 int[] arr = {11, 22, 33, 44, 55};
2 for (int x = 0; x ≤ 4; x++) {
3     System.out.println(arr[x]);
4 }
```

## 数组元素数量

- 格式: 数组名.length
- 范例:

```
1 arr.length
```

```
1 int[] arr2 = {11, 22, 33, 44, 55};
2 for (int x = 0; x < arr.length; x++) {
3     System.out.println(arr2[x]);
4 }
```

## 最值

```
1 int[] arr3 = {12, 45, 98, 73, 60};
2 int max = arr3[0];
3 int min = arr3[0];
4 for (int y = 1; y < arr.length; y++) {
5     if (arr3[y] > max) {
6         max = arr3[y];
7     };
8     if (arr3[y] < min) {
9         min = arr3[y];
10    }
11 }
12 System.out.println("最大值为: " + max);
13 System.out.println("最小值为: " + min);
```

## 方法

## ✧ 方法概论

方法是将具有独立功能的代码块组织成为一个整体，使其具有特殊功能的代码集。

**i** 注意：

- ① 方法必须先创建才可以使用，该过程称为方法定义
- ② 方法创建后并不是直接运行的，需要手动使用后才执行，该过程称为方法调用

## ✧ 方法的定义和调用

○ 方法定义格式：

```
1 public static void 方法名 () {  
2     方法体;  
3 }
```

○ 方法调用格式：

```
1 方法名();
```

```
1 //需求：定义一个方法，在方法中定义一个变量，判断该数是否是偶数  
2 //调用方法  
3 isEvenNumber();
```

```
1 public static void isEvenNumber() {  
2     //定义变量  
3     int number = 10;  
4     //判断该数据是否是偶数  
5     if (number % 2 == 0) {  
6         System.out.println(true);  
7     } else {  
8         System.out.println(false);  
9     }  
10 }
```

## 方法练习

需求：设计一个方法，用于打印两个数中的较大数。

```

1  public static void getMax() {
2      int a = 10;
3      int b = 20;
4      if (a > b) {
5          System.out.println(a);
6      } else {
7          System.out.println(b);
8      }
9  }
10
11 public static void main(String[] args) {
12     getMax();
13 }

```

## ✧ 带参数方法的定义和调用

- 带参数方法的定义格式：

```
1  public static void 方法名 (参数) {... ...}
```

- 单个参数：

```
1  public static void 方法名 (数据类型 变量名) {}
```

- 多个参数：

```
1  public static void 方法名 (数据类型 变量名, 数据类型 变量名, .....) {}
```

### 注意：

- ① 方法定义时，参数中的 **数据类型** 与 **变量名** 都不能少，缺少任意一个程序将报错
- ② 方法定义时，多个参数之间使用逗号分隔

- 带参数方法的调用格式：

```
1  方法名 (参数);
```

- 单个参数：

```
1  方法名 (变量名/常量值);
```

- 多个参数：

```
1 方法名（变量名1/常量值1，变量名2/常量值2）；
```

**i** 注意：

- ① 方法调用时，参数的数量与类型必须与方法定义中的设置相匹配，否则程序将报错

```
1 // 常量值调用
2 isEvenNumber(10);
3
4 // 变量调用
5 int number = 9;
6 isEvenNumber(number);
```

```
1 public static void isEvenNumber(int number) {
2     if (number % 2 == 0) {
3         System.out.println(true);
4     } else {
5         System.out.println(false);
6     }
7 }
```

## 形参和实参

- 形参：方法定义中的参数
- 实参：方法调用中的参数

## 带参数方法练习

需求：设计一个方法，用于打印两个数中的较大数，数据来自于方法参数

```
1 public static void main(String[] args) {
2     getMax(10, 20);
3
4     int a = 10;
5     int b = 20;
6     getMax(a, b);
7 }
8
```

```

9   public static void getMax(int a, int b) {
10      if (a > b) {
11          System.out.println(a);
12      } else {
13          System.out.println(b);
14      }
15  }

```

## ✧ 带返回值方法的定义和调用

**i** 带返回值方法的定义格式：

```

1   public static _1至75.数据类型 方法名 (参数) {
2       return 数据;
3   }

```

**i** 注意：方法定义时，return后面的返回值与方法定义上的数据类型要匹配，否则程序将报错

**i** 带返回值方法调用的格式：

○ 格式1：

```
1   方法名 (参数);
```

○ 格式2：

```
1   数据类型 变量名 = 方法名 (参数);
```

**i** 注意：方法的返回值通常会使用变量接收，否则返回值将无意义

```

1   public static void main(String[] args) {
2       isEvenNumber(10);
3
4       boolean flag = isEvenNumber(10);
5       System.out.println(flag);
6   }
7
8   public static boolean isEvenNumber(int number) {

```

```

9      if (number % 2 == 0) {
10         return true;
11     } else {
12         return false;
13     }
14 }

```

## 带返回值方法练习

需求：设计一个方法可以获取两个数的较大值，数据来自于参数

```

1  public static void main(String[] args) {
2      int max = getMax(10, 20);
3      System.out.println(max);
4
5      System.out.println(getMax(10, 20));
6  }
7
8  public static int getMax(int a, int b) {
9      if (a > b) {
10         return a;
11     } else {
12         return b;
13     }
14 }

```

## ✧ 方法的注意事项

- ① 方法不能嵌套定义
- ② void表示无返回值，可以省略return，也可以单独的书写return，后面不加数据

## 方法的通用格式

格式：

```

1  public static 返回值类型 方法名 (参数) {
2      方法体;
3      return 数据;
4  }

```

**i** 定义方法时，要做到两个明确：

- 1 明确返回值类型：主要是明确方法操作完毕之后是否有数据返回，如果没有，写void；如果有，写对应的数据类型
- 2 明确参数：主要是明确参数的类型和数量

**i** 调用方法时：

- 1 void类型的方法,直接调用即可
- 2 非void类型的方法，推荐用变量接收调用

## ✧ 方法重载

**i** 方法重载指同一个类中定义多个方法之间的关系，满足下列条件的多个方法互相构成重载：

- 1 多个方法在同一个类中
- 2 多个方法具有相同的方法名
- 3 多个方法的参数不相同，类型不同或者数量不同

**i** 方法重载的特点：

- 1 重载仅对应方法的定义，与方法调用无关，调用方式参照标准格式
- 2 重载仅针对同一个类中方法的名称与参数进行识别，与返回值无关，换句话说不能通过返回值来判定两个方法是否相互构成重载

```
1 public static void main(String[] args) {  
2     int result = sum(10, 20);  
3     System.out.println(result);  
4  
5     double result2 = sum(10.0, 20.0);  
6     System.out.println(result2);  
7  
8     int result3 = sum(10, 20, 30);  
9     System.out.println(result3);  
10 }
```



```

11
12 //需求1: 求两个int类型数据和的方法
13 public static int sum(int a, int b) {
14     return a + b;
15 }
16
17 //需求2: 求两个double类型数据和的方法
18 public static double sum(double a, double b) {
19     return a + b;
20 }
21
22 //需求3: 求三个int类型数据和的方法
23 public static int sum(int a, int b, int c) {
24     return a + b + c;
25 }

```

## 方法重载练习

需求：使用方法重载的思想，设计比较两个整数是否相同的方法，兼容全整数类型（byte, int, short, long）

```

1 public static void main(String[] args) {
2     //需求: 使用方法重载的思想，设计比较两个整数是否相同的方法，兼容全整数类
    型 (byte, int, short, long)
3     System.out.println(compare(10, 20));
4     System.out.println(compare((byte) 10, (byte) 20));
5     System.out.println(compare((short) 10, (short) 20));
6     System.out.println(compare(10L, 20L));
7 }
8
9 public static boolean compare(int a, int b) {
10     System.out.println("int");
11     return a == b;
12 }
13
14 public static boolean compare(short a, short b) {
15     System.out.println("short");
16     return a == b;
17 }
18
19 public static boolean compare(byte a, byte b) {
20     System.out.println("byte");
21     return a == b;
22 }

```

```

23
24 public static boolean compare(long a, long b) {
25     System.out.println("long");
26     return a == b;
27 }

```

## ✧ 方法的参数传递

对于基本数据类型的参数，形式参数的改变，不影响实际参数的值

```

1 public static void main(String[] args) {
2     int number = 100;
3     System.out.println("调用change方法前: " + number);
4
5     change(number);
6     System.out.println("调用change方法后: " + number);
7 }
8 public static void change(int number){
9     number = 200;
10 }

```

### 方法参数传递引用类型

对于引用类型的参数，形式参数的改变，影响实际参数的值

```

1 public static void main(String[] args) {
2     int[] arr = {10, 20, 30};
3     System.out.println("调用change方法前: " + arr[1]);
4
5     change(arr);
6     System.out.println("调用change方法后: " + arr[1]);
7 }
8
9 public static void change(int[] arr) {
10     arr[1] = 200;
11 }

```

### 案例：数组遍历

需求：设计一个方法用于数组遍历，要求遍历的结果是在一行上的。

```

1 public static void main(String[] args) {
2     int[] arr = {11, 22, 33, 44, 55};
3     arr_sort(arr);
4 }
5
6 public static void arr_sort(int[] arr) {
7     System.out.print("{");
8     for (int x = 0; x < arr.length; x++) {
9         if (x == arr.length - 1) {
10             System.out.print(arr[x]);
11         } else {
12             System.out.print(arr[x] + ", ");
13         }
14     }
15     System.out.print("}");
16 }

```

System.out.println("内容"); 输出内容并换行

**i** System.out.print("内容"); 输出内容不换行

System.out.println(); 不输出内容，起换行作用

## 案例：数组最大值

需求：设计一个方法用于获取数组中元素的最大值，调用方法并输出结果

```


1 public static void main(String[] args) {
2     int[] arr = {11, 22, 33, 44, 55};
3     getMax(arr);
4 }
5
6 public static void getMax(int[] arr) {
7     int max = arr[0];
8     for (int i = 1; i < arr.length; i++) {
9         if (arr[i] > max) {
10             max = arr[i];
11         }
12     }
13     System.out.println("数组中最大值为: " + max);
14 }

```

# Debug

Debug是供程序员使用的程序调试工具，它可以用于查看程序的执行流程，也可以用于追踪程序执行过程来调试程序。

注意事项：

-  如果数据来自于键盘输入，一定要记住输入数据，不然就不能继续往下看了。

## 基础知识练习

### ✧ 减肥计划if版

需求：输入星期数，显示今天的减肥活动

周一：跑步

周二：游泳

周三：慢走

周四：动感单车

周五：拳击

周六：爬山

周日：好好吃一顿

```
1 Scanner sc = new Scanner(System.in);
2 System.out.println("请输入一个星期数 (1-7) : ");
3 int week = sc.nextInt();
4 if (week < 1 | week > 7) {
5     System.out.println("您输入的星期数有误");
6 } else if (week == 1) {
7     System.out.println("跑步");
8 } else if (week == 2) {
9     System.out.println("游泳");
```

```

10 } else if (week == 3) {
11     System.out.println("慢走");
12 } else if (week == 4) {
13     System.out.println("动感单车");
14 } else if (week == 5) {
15     System.out.println("拳击");
16 } else if (week == 6) {
17     System.out.println("爬山");
18 } else if (week == 7) {
19     System.out.println("好好吃一顿");
20 }

```

## ✧ 减肥计划switch版

---

需求：输入星期数，显示今天的减肥活动

周一：跑步

周二：游泳

周三：慢走

周四：动感单车

周五：拳击

周六：爬山

周日：好好吃一顿

```

1 Scanner sc = new Scanner(System.in);
2 System.out.println("请输入一个星期数 (1-7) : ");
3 int week = sc.nextInt();
4 switch (week) {
5     case 1:
6         System.out.println("跑步");
7         break;
8     case 2:
9         System.out.println("游泳");
10        break;
11    case 3:
12        System.out.println("慢走");
13        break;
14    case 4:
15        System.out.println("动感单车");
16        break;

```

```

17     case 5:
18         System.out.println("拳击");
19         break;
20     case 6:
21         System.out.println("爬山");
22         break;
23     case 7:
24         System.out.println("好好吃一顿");
25         break;
26     default:
27         System.out.println("您输入的星期数有误");
28 }

```

## ✧ 逢七过

需求：朋友聚会的时候可能会玩一个游戏：\_1至75.逢七过。规则是：从任意一个数字开始报数，当你要报的数字包含7或者是7的倍数时都要说：过。为了帮助大家更好的玩这个游戏，这里我们直接在控制台打印出1-100之间满足逢七过规则的数据。

```

1  System.out.print("1-100内要喊过的数字有: ");
2  for (int guo = 1; guo <= 100; guo++) {
3      if (guo % 7 == 7 | guo / 10 % 10 == 7 | guo % 7 == 0) {
4          System.out.print(guo + " ");
5      }
6  }

```

## ✧ 不死神兔

需求：有一对兔子，从出生后第三个月起每个月都生一对兔子，小兔子长到第三个月后每个月又生一对兔子，假如兔子都不死，问第20个月的兔子对数为多少？

```

1  int[] arr = new int[20];
2  arr[0] = 1;
3  arr[1] = 1;
4  for (int x = 2; x < arr.length; x++) {
5      arr[x] = arr[x - 2] + arr[x - 1];
6  }
7  System.out.println("第二十个月兔子的对数为: " + arr[19]);

```

## ✧ 百钱百鸡

---

需求：我国古代数学家张邱建在《算经》一书中提出的数学问题：鸡翁一值钱五，鸡母一值钱三，鸡雏三值钱一。百钱买百鸡，问鸡翁、鸡母、鸡雏各几何？

```
1  for (int x = 0; x ≤ 20; x++) {
2      for (int y = 0; y ≤ 33; y++) {
3          int z = 100 - x - y;
4          if (z % 3 == 0 && 5 * x + 3 * y + z / 3 == 100) {
5              System.out.println("鸡翁: " + x + "只");
6              System.out.println("鸡母: " + y + "只");
7              System.out.println("鸡雏: " + z + "只");
8          }
9      }
10 }
```

## ✧ 数组元素求和

---

需求：有这样一个数组，元素是{68, 27, 95, 88, 171, 996, 51, 210}。求出该数组中满足要求的元素和，要求是：求和的元素个位和十位都不能是7，并且只能是偶数。

```
1  int[] arr = {68, 27, 95, 88, 171, 996, 51, 210};
2  int sum = 0;
3  for (int i = 0; i < arr.length; i++) {
4      if (arr[i] % 10 ≠ 7 && arr[i] / 10 % 10 ≠ 7 && arr[i] % 2 ==
5          0) {
6          sum += arr[i];
7      }
8  }
9  System.out.println("和为: " + sum);
```

## ✧ 数组内容相同

---

需求：设计一个方法，用于比较两个数组的内容是否相同

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

```

2    int[] arr = {11, 22, 33, 44, 55};
3    int[] arr2 = {11, 22, 33, 44, 55};
4    int[] arr3 = {11, 22, 33, 44, 5};
5
6    boolean result = arrCompare(arr, arr2);
7    System.out.println(result);
8    System.out.println("-----");
9
10   boolean flag = arrCompare(arr, arr3);
11   System.out.println(flag);
12 }
13
14 public static boolean arrCompare(int[] arr, int[] arr2) {
15     if (arr.length != arr2.length) {
16         return false;
17     }
18     for (int x = 0; x < arr.length; x++) {
19         if (arr[x] != arr2[x]) {
20             return false;
21         }
22     }
23     return true;
24 }

```

## ✧ 查找

---

需求：已知一个数组arr = {19, 28, 37, 46, 50}，键盘录入一个数据，查找该数据在数组中的索引，并在控制台输出找到的索引值。

```

1    int[] arr = {19, 28, 37, 46, 50};
2    Scanner sc = new Scanner(System.in);
3    System.out.println("请输入您要查找的值: ");
4    int a = sc.nextInt();
5    int index = -1;
6    for (int i = 0; i < arr.length; i++) {
7        if (arr[i] == a) {
8            index = i;
9            System.out.println("您查找的值在数组中的索引值为: " + index);
10           break;
11       }
12   }
13   System.out.println("您查找的值在数组中不存在");

```



## ✧ 反转

---

需求：已知一个数组arr = {19,28,37,46,50};用程序实现把数组中的元素值交换，交换后的数组arr = {50,46,37,28,19};并在控制台输出交换后的数组元素。

```
1 public static void main(String[] args) {
2     int[] arr = {19, 28, 37, 46, 50};
3     for (int start = 0, end = arr.length - 1; start ≤ end;
4         start++, end--) {
5         int temp = arr[start];
6         arr[start] = arr[end];
7         arr[end] = temp;
8     }
9     printArray(arr);
10 }
11 public static void printArray(int[] arr) {
12     System.out.print("[");
13     for (int i = 0; i < arr.length; i++) {
14         if (i == arr.length - 1) {
15             System.out.print(arr[i]);
16         } else {
17             System.out.print(arr[i] + ", ");
18         }
19     }
20     System.out.print("]");
21 }
```

## ✧ 评委打分

---

需求：在编程竞赛中，有6个评委为参赛的选手打分，分数为0-100的整数分。

选手的最后得分为：去掉一个最高分和一个最低分后的4个评委平均值（不考虑小数部分）。

```
1 public static void main(String[] args) {
2     int[] arr = new int[6];
3     Scanner sc = new Scanner(System.in);
4     for (int x = 0; x < 6; x++) {
5         System.out.println("请输入第" + (x + 1) + "位评委的打分: ");
6         arr[x] = sc.nextInt();
7     }
8 }
```

```
7     }
8     printArray(arr);
9     int max = getMax(arr);
10    int min = getMin(arr);
11    int sum = getSum(arr);
12    int score = (sum - max - min) / (arr.length - 2);
13    System.out.println();
14    System.out.println("选手的最终得分为: " + score);
15 }
16
17 public static void printArray(int[] arr) {
18     System.out.print("[");
19     for (int i = 0; i < arr.length; i++) {
20         if (i == arr.length - 1) {
21             System.out.print(arr[i]);
22         } else {
23             System.out.print(arr[i] + ", ");
24         }
25     }
26     System.out.print("]");
27 }
28
29 public static int getMax(int[] arr) {
30     int max = arr[0];
31     for (int i = 1; i < arr.length; i++) {
32         if (arr[i] > max) {
33             max = arr[i];
34         }
35     }
36     return max;
37 }
38
39 public static int getMin(int[] arr) {
40     int min = arr[0];
41     for (int i = 1; i < arr.length; i++) {
42         if (arr[i] < min) {
43             min = arr[i];
44         }
45     }
46     return min;
47 }
48
49 public static int getSum(int[] arr) {
50     int sum = 0;
51     for (int i = 0; i < arr.length; i++) {
```

```
52     sum += arr[i];
53 }
54 return sum;
55 }
```

# 面向对象基础

## ✧ 类和对象

### 什么是类

类是对现实生活中一类具有 **共同属性** 和 **行为** 的事物的抽象。

**i** 类的特点：

- 类是对象的数据类型
- 类是具有相同属性和行为的一组对象的集合

### 什么是对象的属性

属性：对象具有的各种特征，每个对象的每个 **属性** 都拥有特定的 **值**。

### 什么是对象的行为

行为：对象能够执行的操作。

### 类和对象的关系

- 类：类是对现实生活中一类具有共同属性和行为的事物的抽象
- 对象：是能够看得到摸得着的真实存在的实体

**类是对象的抽象，对象是类的实体**

## 类的定义

类的重要性：是 Java 程序的基本组成单位。

**i** 类的组成：属性 和 行为

- 属性：在类中通过 成员变量 来体现（类中方法外的变量）
- 行为：在类中通过 成员方法 来体现（和前面的方法相比去掉static关键字即可）

**i** 类的定义步骤：

- 1 定义类
- 2 编写类的成员变量
- 3 编写类的成员方法

```
1  class Phone {  
2      String brand;  
3      int price;  
4      public void call() {  
5          System.out.println("打电话");  
6      }  
7      public void sendMessage() {  
8          System.out.println("发短信");  
9      }  
10 }
```

## 对象的使用

创建对象

- 格式：

```
1  类名 对象名 = new 类名();
```

- 范例：

```
1  Phone p = new Phone();
```

使用对象

## ① 使用成员变量

- 格式:

```
1 对象名.变量名
```

- 范例:

```
1  p.brand
```

## ② 使用成员方法

- 格式:

```
1  对象名.方法名()
```

- 范例:

```
1  p.call()
```

## 案例：学生类

需求：首先定义一个学生类，然后定义一个学生测试类，在学生测试类中通过对象完成成员变量和成员方法的使用

```
1  class Student {  
2      String name;  
3      int age;  
4      public void study() {  
5          System.out.println("好好学习, 天天向上");  
6      }  
7      public void doHomework() {  
8          System.out.println("键盘敲烂, 月薪过万");  
9      }  
10 }
```

```
1  public static void main(String[] args) {  
2      Student s = new Student();  
3      System.out.println(s.name + "," + s.age);  
4      s.name = "林青霞";  
5      s.age = 30;  
6      System.out.println(s.name + "," + s.age);  
7      s.study();  
8      s.doHomework();  
9  }
```

## ✧ 成员变量和局部变量

- 成员变量：类中方法外的变量
- 局部变量：方法中的变量

成员变量和局部变量的区别：

区别	类中位置不同	内存中位置不同	生命周期不同	初始化值不同
成员变量	类中方法外	堆内存	随着对象的存在而存在，随着对象的消失而消失	有默认的初始化值
局部变量	方法内或者方法声明上	栈内存	随着方法的调用而存在，随着方法的调用完毕而消失	没有默认的初始化值，必须先定义，赋值，才能使用

## ✧ 封装

### private 关键字

#### private:

- 是一个权限修饰符
- 可以修饰成员（成员变量和成员方法）
- 作用是保护成员不被别的类使用，被 `private` 修饰的成员在本类中才能访问



针对 `private` 修饰的成员变量，如果需要被其它类使用，提供相应的操作：

- ① 提供“`get变量名()`”方法，用于获取成员变量的值，方法用 `public` 修饰
- ② 提供“`set变量名(参数)`”方法，用于设置成员变量的值，方法用 `public` 修饰

## private 关键字的使用

```
1  class Student1 {
2      private String name;
3      private int age;
4
5      public void setName(String n) {
6          name = n;
7      }
8
9      public String getName() {
10         return name;
11     }
12
13     public void setAge(int a) {
14         age = a;
15     }
16
17     public int getAge() {
18         return age;
19     }
20     public void show() {
21         System.out.println(name + "," + age);
22     }
23 }
```

```
1  public static void main(String[] args) {
2      Student1 s = new Student1();
3      s.setName("林青霞");
4      s.setAge(30);
5      s.show();
6
7      System.out.println(s.getName() + "---" + s.getAge());
8      System.out.println(s.getName() + "," + s.getAge());
9  }
```

## this 关键字

- `this` 修饰的变量用于指代成员变量

- 方法的形参如果与成员变量同名，不带 `this` 修饰的变量指的是形参，而不是成员变量
- 方法的形参没有与成员变量同名，不带 `this` 修饰的变量指的是成员变量
- 什么时候使用 `this`？—— 解决局部变量隐藏成员变量
- `this` 代表所在类的对象引用
  - 记住：方法被哪个对象调用，`this` 就代表那个对象

## 封装

### i 封装概述：

- 是面向对象三大特征之一（封装、继承、多态）
- 是面向对象编程语言对客观世界的模拟，客观世界里成员变量都是隐藏在对象内部的，外界是无法直接操作的

### i 封装原则：

- 将类的某些信息隐藏在类内部，不允许外部程序直接访问，而是通过该类提供的方法来实现对隐藏信息的操作和访问成员变量 `private`，提供对应的 `getXxx()` / `setXxx()` 方法

### i 封装好处：

- 通过方法来控制成员变量的操作，提高了代码的安全性
- 把代码用方法进行封装，提高了代码的复用性

## ✧ 构造方法

构造方法是一种特殊的方法

作用：创建对象

格式：



```
1 public class 类名{
2     修饰符 类名(参数){
3     }
4 }
```

功能：主要是完成对象数据的初始化

### 注意事项：

#### ① 构造方法的创建

- 如果没有定义构造方法，系统将给出一个 **默认** 的 **无参数构造方法**
- 如果定义了构造方法，系统将不再提供默认无参构造方法

#### ② 构造方法的重载

- 如果定义了带参构造方法，还要使用无参数构造方法，就必须再写一个无参数构造方法

#### ③ 推荐的使用方式：

- **无论是否使用，都手工书写无参数构造方法**

## 标准类制作

#### ① 成员变量：使用 **private** 修饰

#### ② 构造方法：

- 提供一个无参构造方法
- 提供一个带多个参数的构造方法

#### ③ 成员方法：

- 提供每一个成员变量对应的 **setXxx()** / **getXxx()**
- 提供一个显示对象信息的 **show()**

#### ④ 创建对象并为其成员变量赋值的两种方式

- 无参构造方法创建对象后使用 **setXxx()** 赋值
- 使用带参构造方法直接创建带有属性值的对象

```
1 public class Student {
2     //成员变量
3     private String name;
4     private int age;
```

```
5
6    //构造方法
7    public Student() {
8    }
9
10   public Student(String name, int age) {
11       this.name = name;
12       this.age = age;
13   }
14
15   //成员方法
16   public void setName(String name) {
17       this.name = name;
18   }
19
20   public void setAge(int age) {
21       this.age = age;
22   }
23
24   public String getName() {
25       return name;
26   }
27
28   public int getAge() {
29       return age;
30   }
31
32   public void show() {
33       System.out.println(name + "," + age);
34   }
35 }
```

```
1    public class test {
2        public static void main(String[] args) {
3            Student s1 = new Student();
4            s1.setName("林青霞");
5            s1.setAge(30);
6            s1.show();
7
8            Student s2 = new Student("林青霞", 30);
9            s2.show();
10        }
11    }
```

# 字符串

## ✧ API

API（Application Programming Interface）：应用程序编程接口

**i** 注意：

- 1 调用方法的时候，如果方法有明确的返回值，我们用变量接收
- 2 可以手动完成，也可以使用快捷键的方式完成（CTRL+ALT+V）

## API 使用练习

需求：按照帮助文档的使用步骤学习Scanner类的使用，并实现键盘录入一个字符串，最后输出在控制台

```
1 public class api {  
2     public static void main(String[] args) {  
3         Scanner sc = new Scanner(System.in);  
4         System.out.println("请输入你想打印的话: ");  
5         String line = sc.nextLine();  
6         System.out.println(line);  
7     }  
8 }
```

## ✧ String

### String 构造

String类在 `java.lang` 包下，所以使用时不需要导包

`String` 类代表 **字符串**，java程序中的所有字符串文字（例如："abc"）都被实现为此类的实例。也就是说，**Java程序中所有的双引号字符串，都是String类的对象**。

### **i** 字符串的特点:

- 字符串不可变，它们的值在创建后不能被更改
- 虽然String的值是不可变的，但是它们可以被共享
- 字符串效果上相当于字符数组（`char[]`），但是底层原理是字节数组（`byte[]`）（JDK8及以前是字符数组，JDK9及以后是字节数组）

### **i** 构造方法:

构造方法	描述
<code>public String()</code>	创建一个空白字符串对象，不含有任何内容
<code>public String(char[] chs)</code>	根据字符数组的内容，来创建字符串对象
<code>public String(byte[] bys)</code>	根据字节数组的内容，来创建字符串对象
<code>String s = "abc";</code>	直接赋值的方式创建字符串对象，内容就是abc

### **i** 推荐使用直接赋值的方式得到字符串对象

```
1 String s1 = new String();
2 System.out.println("s1:" + s1); // s1:
3
4 char[] chs = {'a', 'b', 'c'};
5 String s2 = new String(chs);
6 System.out.println("s2:" + s2); // s2:abc
7
8 byte[] bys = {97, 98, 99};
9 String s3 = new String(bys);
10 System.out.println("s3:" + s3); // s3:abc
11
12 String s4 = "abc";
13 System.out.println("s4:" + s4); // s4:abc
```

## String 对象特点

- ① 通过 `new` 创建的字符串对象，每一次 `new` 都会申请一个内存空间，虽然内容相同，但是地址值不同

- ② 以“”方式给出的字符串，只要 字符序列相同（顺序和大小写），无论在程序代码中出现几次，JVM都 只会建立一个String对象，并在字符串池中维护

## 字符串比较

**i** 使用 `=` 作比较：

- 基本类型：比较的是数据值是否相同
- 引用类型：比较的是地址值是否相同

**i** 使用 `equals()` 方法比较：

- 字符串是对象，它比较内容是否相同，是通过一个方法来实现的，这个方法叫 `equals()`
- `public boolean equals(Object anObject)` :将此字符串与指定对象进行比较。由于我们比较的是字符串对象，所以参数直接传递一个字符串

## 案例：用户登录

需求：已知用户名和密码，请用程序实现模拟用户登录。总共给三次机会，登录之后，给出相应的提示。

```
1  import java.util.Scanner;
2
3  public class 用户登录 {
4      public static void main(String[] args) {
5          String username = "root";
6          String password = "123456";
7
8          Scanner sc = new Scanner(System.in);
9
10         for (int i = 0; i < 3; i++) {
11
12             System.out.println("请输入用户名: ");
13             String name = sc.nextLine();
14
15             System.out.println("请输入密码: ");
16             String pasw = sc.nextLine();
17         }
```

```

18         if (name.equals(username) && pasw.equals(password)) {
19             System.out.println("登录成功! ");
20             break;
21         } else {
22             if (2 - i == 0) {
23                 System.out.println("你的账户被锁定, 请与管理员联
24 系");
25             } else {
26                 System.out.println("登录失败, 你还有" + (2 - i)
27 + "次机会");
28             }
29         }
30     }
31 }

```

## 案例：遍历字符串

需求：键盘录入一个字符串，使用程序实现在控制台遍历该字符串。

```

1  import java.util.Scanner;
2
3  public class 遍历字符串 {
4      public static void main(String[] args) {
5          Scanner sc = new Scanner(System.in);
6          System.out.println("请输入一个字符串: ");
7          String s = sc.nextLine();
8          // s.length()获取字符串长度
9          for (int i = 0; i < s.length(); i++) {
10             // s.charAt()获取指定索引处的字符
11             System.out.println(s.charAt(i));
12         }
13     }
14 }

```

## 案例：统计字符次数

需求：键盘录入一个字符串，统计该字符串中大写字母字符，小写字母字符，数字字符出现的次数（不考虑其他字符）。

```

1  import java.util.Scanner;
2

```

```

3 public class 统计字符次数 {
4     public static void main(String[] args) {
5         Scanner sc = new Scanner(System.in);
6         System.out.println("请输入一个字符串: ");
7         String line = sc.nextLine();
8
9         int bigCount = 0;
10        int smallCount = 0;
11        int numberCount = 0;
12
13        for (int i = 0; i < line.length(); i++) {
14            char ch = line.charAt(i);
15
16            if (ch ≥ 'A' && ch ≤ 'Z') {
17                bigCount++;
18            } else if (ch ≥ 'a' && ch ≤ 'z') {
19                smallCount++;
20            } else if (ch ≥ '0' && ch ≤ '9') {
21                numberCount++;
22            }
23        }
24
25        System.out.println("大写字母: " + bigCount + "个");
26        System.out.println("小写字母: " + smallCount + "个");
27        System.out.println("数字: " + numberCount + "个");
28    }
29 }

```

## 案例：字符串拼接

需求：定义一个方法，把int数组中的数据按照指定的格式拼接成一个字符串返回，调用该方法，并在控制台输出结果。例如：数组为int[] arr = {1,2,3};，执行该方法后输出的结果为：[1, 2, 3]。

```

1 public class 拼接字符串 {
2     public static void main(String[] args) {
3         int[] arr = {1, 2, 3};
4         String s = arrayToStrijng(arr);
5         System.out.println(s); // [1, 2, 3]
6     }
7
8     public static String arrayToStrijng(int[] arr) {
9         String s = "";

```

```

10         s += "[";
11
12         for (int i = 0; i < arr.length; i++) {
13             if (i == arr.length - 1) {
14                 s += arr[i];
15             } else {
16                 s += arr[i];
17                 s += ", ";
18             }
19         }
20
21         s += "]";
22         return s;
23     }
24 }

```

## 案例：字符串反转

需求：定义一个方法，实现字符串反转。键盘录入一个字符串，调用该方法后，在控制台输出结果。

例如：键盘录入abc，输出结果cba。

```

1  import java.util.Scanner;
2
3  public class 字符串反转 {
4      public static void main(String[] args) {
5          Scanner sc = new Scanner(System.in);
6          System.out.println("请输入一个字符串: ");
7          String line = sc.nextLine();
8
9          String s = stringReverse(line);
10
11         System.out.println("字符串反转结果为: " + s);
12     }
13
14     public static String stringReverse(String s) {
15         String ss = "";
16         for (int i = s.length() - 1; i ≥ 0; i--) {
17             ss += s.charAt(i);
18         }
19         return ss;
20     }
21 }

```



## 通过帮助文档查看 **String** 中的方法

方法名	描述
<code>public boolean equals(Object anObject)</code>	比较字符串的内容，严格区分大小写（用户名和密码）
<code>public char charAt(int index)</code>	返回指定索引处的 <code>char</code> 值
<code>public int length()</code>	返回此字符串的长度

## ✧ **StringBuilder**

- `StringBuilder`是一个可变的字符串类，我们可以把它看成是一个容器。
- 这里的可变指的是`StringBuilder`对象中的内容是可变的。

**i** `String` 和 `StringBuilder` 的区别：

- `String`：内容是不可变的
- `StringBuilder`：内容是可变的

**i** 构造方法：

构造方法	描述
<code>public StringBuilder()</code>	创建一个空白可变字符串对象，不含有任何内容
<code>public StringBuilder(String str)</code>	根据字符串的内容，来创建可变字符串对象

```
1  StringBuilder sb = new StringBuilder();
2  System.out.println("sb:" + sb);
3  System.out.println("sb.length(): " + sb.length());
4
5  StringBuilder sb2 = new StringBuilder("hello");
6  System.out.println("sb2:" + sb2);
7  System.out.println("sb2.length(): " + sb2.length());
```

## StringBuilder 的添加和反转方法

```
1 //创建对象
2 StringBuilder sb = new StringBuilder();
3
4 StringBuilder sb2 = sb.append("hello");
5
6 System.out.println("sb:" + sb);
7 System.out.println("sb2:" + sb2);
8 System.out.println(sb == sb2);
9
10 /*sb.append("hello");
11 sb.append("world");
12 sb.append("java");
13 sb.append(100);
14 */
15
16 //链式编程
17 sb.append("hello").append("world").append("java").append(100);
18
19 System.out.println("sb:" + sb);
20
21 sb.reverse();
22 System.out.println("sb:" + sb);
```

## StringBuilder 和 String 相互转换

### 1 StringBuilder转换为String

- `public String toString()` : 通过 `toString()` 就可以实现把StringBuilder转换为String

### 2 String转换为StringBuilder

- `public StringBuilder(String s)` : 通过构造方法就可以实现把String转换为StringBuilder

```

1  StringBuilder sb = new StringBuilder();
2  sb.append("hello");
3
4  String s = sb.toString();
5  System.out.println(s);
6
7  String str = "hello";
8  StringBuilder sb2 = new StringBuilder(str);
9  System.out.println(sb2);

```

## 案例：拼接字符串

需求：定义一个方法，把int数组中的数据按照指定的格式拼接成一个字符串返回，调用该方法，并在控制台输出结果。例如：数组为int[] arr = {1,2,3};，执行该方法后输出的结果为：[1, 2, 3]。

```

1  public static void main(String[] args) {
2      int[] arr = {1, 2, 3};
3
4      String s = arrayToString(arr);
5
6      System.out.println("s:" + s);
7  }
8
9  public static String arrayToString(int[] arr) {
10     StringBuilder sb = new StringBuilder();
11     sb.append("[");
12     for (int i = 0; i < arr.length; i++) {
13         if (i == arr.length - 1) {
14             sb.append(arr[i]);
15         } else {
16             sb.append(arr[i]).append(", ");
17         }
18     }
19     sb.append("]");
20     String s = sb.toString();
21     return s;
22 }

```

## 案例：字符串反转

需求：定义一个方法，实现字符串反转。键盘录入一个字符串，调用该方法后，在控制台输出结果。

例如：键盘录入abc，输出结果cba。

```
1 public static void main(String[] args) {
2     Scanner sc = new Scanner(System.in);
3     System.out.println("请输入一个字符串: ");
4     String line = sc.nextLine();
5
6     String s = myReverse(line);
7     System.out.println("反转后的字符串为: " + s);
8 }
9
10 public static String myReverse(String s) {
11     /*StringBuilder sb = new StringBuilder(s);
12     sb.reverse();
13     String s1 = sb.toString();
14     return s1;*/
15
16     return new StringBuilder(s).reverse().toString();
17 }
```

## 集合基础

### ✧ 集合概述

---

编程的时候如果要存储多个数据，使用长度固定的数组存储格式，不一定能满足我们的需求，更适应不了变化的需求。此时，就应该使用 **集合**。

**i** 集合的特点：

- 提供一种存储空间可变的存储模型，存储的数据容量可以发生改变

### ✧ ArrayList

---

## 1 ArrayList<E>

- 可调整大小的数组实现
- :是一种特殊的数据类型，泛型
- 使用：在出现E的地方用引用数据类型替换即可
- 范例：

## 1 ArrayList<String>, ArrayList<Student>

**i** 构造方法和添加方法：

方法名	描述
public ArrayList()	创建一个空的集合对象
public boolean add(E e)	将特定的元素追加到此集合的末尾
public void add(int index, E element)	在此集合中的特定位置插入指定的元素

```
1 //public ArrayList() 创建一个空的集合对象
2 //ArrayList<String> array = new ArrayList<>();
3
4 ArrayList<String> array = new ArrayList<String>();
5
6 //public boolean add(E e) 将特定的元素追加到此集合的末尾
7 System.out.println(array.add("hello"));
8
9 array.add("hello");
10 array.add("world");
11 array.add("java");
12
13 //public void add(int index, E element) 在此集合中的特定位置插入指定的元素
14 array.add(1, "javase");
15 array.add(3, "javase");
16
17 //输出集合
18 System.out.println("array:" + array);
```

## ✧ ArrayList 集合常用方法

方法名	描述
public boolean remove(Object o)	删除指定的元素，返回删除是否成功
public E remove(int index)	删除指定索引处的元素，返回被删除的元素
public E set(int index, E element)	修改指定索引处的元素，返回被修改的元素
public E get(int index)	返回指定索引处的元素
public int size()	返回集合中的元素个数

```

1 //创建集合
2 ArrayList<String> array = new ArrayList<String>();
3
4 //添加元素
5 array.add("hello");
6 array.add("world");
7 array.add("java");
8 //public boolean remove(Object o) 删除指定的元素，返回删除是否成功
9 System.out.println(array.remove("world"));
10
11 //public E remove(int index) 删除指定索引处的元素，返回被删除的元素
12 System.out.println(array.remove(1));
13
14 //public E set(int index, E element) 修改指定索引处的元素，返回被修改
    的元素
15 System.out.println(array.set(1, "javase"));
16
17 //public E get(int index) 返回指定索引处的元素
18 System.out.println(array.get(0));
19
20 //public int size() 返回集合中的元素个数
21 System.out.println(array.size());
22
23 //输出集合
24 System.out.println("array:" + array);

```

## ✧ 案例：存储字符串并遍历

需求：创建一个存储字符串的集合，存储3个字符串元素，使用程序实现在控制台遍历该集合。

```
1 ArrayList<String> array = new ArrayList<String>();
2
3 array.add("hello");
4 array.add("world");
5 array.add("java");
6
7 for (int i = 0; i < array.size(); i++) {
8     System.out.println(array.get(i));
9 }
```

## ✧ 案例：存储学生对象并遍历

---

需求：创建一个存储学生对象的集合，存储3个学生对象，使用程序实现在控制台遍历该集合。

```
1 public class Student {
2     private String name;
3     private int age;
4
5     public Student() {
6     }
7
8     public Student(String name, int age) {
9         this.name = name;
10        this.age = age;
11    }
12
13    public void setName(String name) {
14        this.name = name;
15    }
16
17    public String getName() {
18        return name;
19    }
20
21    public void setAge(int age) {
22        this.age = age;
23    }
24
25    public int getAge() {
26        return age;
27    }
28 }
```

```
28 }
```

```
1 public class Demo {
2     public static void main(String[] args) {
3
4         //创建集合对象
5         ArrayList<Student> array = new ArrayList<Student>();
6
7         //创建学生对象
8         Student s1 = new Student("林青霞", 30);
9         Student s2 = new Student("风清扬", 33);
10        Student s3 = new Student("张曼玉", 18);
11
12        //添加学生对象到集合中
13        array.add(s1);
14        array.add(s2);
15        array.add(s3);
16
17        //遍历集合
18        for (int i = 0; i < array.size(); i++) {
19            Student s = array.get(i);
20            System.out.println(s.getName() + "," + s.getAge());
21        }
22
23    }
24 }
```

## ✧ 案例：存储学生对象并遍历升级版

需求：创建一个存储学生对象的集合，存储3个学生对象，使用程序实现在控制台遍历该集合，学生的姓名和年龄来自键盘录入。

```
1 public class Student {
2     private String name;
3     private String age;
4
5     public Student() {
6     }
7
8     public Student(String name, String age) {
9         this.name = name;
10        this.age = age;
11    }
12 }
```



```
11     }
12
13     public void setName(String name) {
14         this.name = name;
15     }
16
17     public String getName() {
18         return name;
19     }
20
21     public void setAge(String age) {
22         this.age = age;
23     }
24
25     public String getAge() {
26         return age;
27     }
28 }
```

```
1  public class Demo {
2      public static void main(String[] args) {
3
4          ArrayList<Student> array = new ArrayList<Student>();
5
6          /*Scanner sc = new Scanner(System.in);
7          System.out.println("请输入学生姓名: ");
8          String name = sc.nextLine();
9          System.out.println("请输入学生的年龄: ");
10         String age = sc.nextLine();
11
12         Student s = new Student();
13         s.setName(name);
14         s.setAge(age);
15
16         array.add(s);*/
17
18         addStudent(array);
19         addStudent(array);
20         addStudent(array);
21
22         for (int i = 0; i < array.size(); i++) {
23             Student s = array.get(i);
24             System.out.println(s.getName() + "," + s.getAge());
25         }
26     }
```

```

27
28     public static void addStudent(ArrayList<Student> array) {
29         Scanner sc = new Scanner(System.in);
30         System.out.println("请输入学生姓名: ");
31         String name = sc.nextLine();
32         System.out.println("请输入学生的年龄: ");
33         String age = sc.nextLine();
34
35         Student s = new Student();
36         s.setName(name);
37         s.setAge(age);
38
39         array.add(s);
40     }
41 }

```

## 案例：学生管理系统

```

1  public class Student {
2      //学号
3      private String sid;
4      //姓名
5      private String name;
6      //年龄
7      private String age;
8      //居住地
9      private String address;
10
11     public Student() {
12     }
13
14     public Student(String sid, String name, String age, String
address) {
15         this.sid = sid;
16         this.name = name;
17         this.age = age;
18         this.address = address;
19     }
20
21     public String getSid() {

```

```

22         return sid;
23     }
24
25     public void setSid(String sid) {
26         this.sid = sid;
27     }
28
29     public String getName() {
30         return name;
31     }
32
33     public void setName(String name) {
34         this.name = name;
35     }
36
37     public String getAge() {
38         return age;
39     }
40
41     public void setAge(String age) {
42         this.age = age;
43     }
44
45     public String getAddress() {
46         return address;
47     }
48
49     public void setAddress(String address) {
50         this.address = address;
51     }
52 }

```

```

1  import java.util.ArrayList;
2  import java.util.Scanner;
3
4  public class StudentManager {
5      public static void main(String[] args) {
6          // 创建集合对象用于存储学生数据
7          ArrayList<Student> array = new ArrayList<Student>();
8          // 用循环完成再次回到主界面
9          while (true) {
10             // 主界面编写
11             System.out.println("-----欢迎来到学生管理系统-----
-");
12             System.out.println("1 添加学生");

```

```

13         System.out.println("2 删除学生");
14         System.out.println("3 修改学生");
15         System.out.println("4 查看所有学生");
16         System.out.println("5 退出");
17         System.out.println("请输入你的选择: ");
18
19         //用Scanner实现键盘录入数据
20         Scanner sc = new Scanner(System.in);
21         String line = sc.nextLine();
22
23         //用switch语句完成操作的选择
24         switch (line) {
25             case "1":
26                 //System.out.println("添加学生");
27                 addStudent(array);
28                 break;
29             case "2":
30                 //System.out.println("删除学生");
31                 deleteStudent(array);
32                 break;
33             case "3":
34                 //System.out.println("修改学生");
35                 updateStudent(array);
36                 break;
37             case "4":
38                 //System.out.println("查看所有学生");
39                 findAllStudent(array);
40                 break;
41             case "5":
42                 System.out.println("谢谢使用");
43                 //break;
44                 System.exit(0); //JVM退出
45         }
46     }
47 }
48
49 //定义一个方法，用于添加学生信息
50 public static void addStudent(ArrayList<Student> array) {
51     //键盘录入学生对象所需要的数据，显示提示信息，提示要输入何种信息
52     Scanner sc = new Scanner(System.in);
53
54     //为了让sid在循环外被访问到
55     String sid;
56
57     //为了让程序回到这里，用循环实现

```

```

58         while (true) {
59             System.out.println("请输入学生学号: ");
60             sid = sc.nextLine();
61             boolean flag = isUsed(array, sid);
62             if (flag) {
63                 System.out.println("该学号已被使用, 请重新输入");
64             } else {
65                 break;
66             }
67         }
68
69         System.out.println("请输入学生姓名: ");
70         String name = sc.nextLine();
71         System.out.println("请输入学生年龄: ");
72         String age = sc.nextLine();
73         System.out.println("请输入学生居住地: ");
74         String address = sc.nextLine();
75         // 创建学生对象, 把键盘录入的数据赋值给学生对象的成员变量
76         Student s = new Student();
77         s.setSid(sid);
78         s.setName(name);
79         s.setAge(age);
80         s.setAddress(address);
81         // 将学生对象添加到集合中
82         array.add(s);
83         // 给出添加成功提示
84         System.out.println("添加学生成功! ");
85     }
86
87     // 定义一个方法, 用于查看所有学生信息
88     public static void findAllStudent(ArrayList<Student> array)
89     {
90         // 判断集合中是否有数据, 如果没有显示提示信息
91         if (array.size() == 0) {
92             System.out.println("无信息, 请先添加信息再查询");
93             // 为了让程序不再往下执行
94             return;
95         }
96         // 显示表头信息
97         // \t 其实就是tab键的位置
98         System.out.println("学号\t姓名\t年龄\t居住地");
99         // 将集合数据取出来按照对应格式显示学生信息, 年龄显示补充"岁"
100        for (int i = 0; i < array.size(); i++) {


```

```
101         System.out.println(s.getSid() + "\t\t" + s.getName()
+ "\t" + s.getAge() + "岁\t" + s.getAddress());
102     }
103 }
104
105 //定义一个方法，用于删除学生信息
106 public static void deleteStudent(ArrayList<Student> array) {
107     //键盘录入要删除的学生学号
108     Scanner sc = new Scanner(System.in);
109     System.out.println("请输入要删除的学生的学号：");
110     String sid = sc.nextLine();
111     //遍历集合将对应学生对象从集合中删除
112     int index = -1;
113
114     for (int i = 0; i < array.size(); i++) {
115         Student s = array.get(i);
116         if (s.getSid().equals(sid)) {
117             //array.remove(i);
118             index = i;
119             break;
120         }
121     }
122
123     if (index == -1) {
124         System.out.println("该学号不存在，请重新输入");
125     } else {
126         array.remove(index);
127         //给出删除成功的提示
128         System.out.println("删除学生成功!");
129     }
130 }
131
132 //定义一个方法，用于修改学生信息
133 public static void updateStudent(ArrayList<Student> array) {
134     //键盘录入要修改的学生学号，显示提示信息
135     Scanner sc = new Scanner(System.in);
136     System.out.println("请输入要修改的学生的学号：");
137     String sid = sc.nextLine();
138     //键盘录入要修改的学生信息
139     System.out.println("请输入新的学生姓名：");
140     String name = sc.nextLine();
141     System.out.println("请输入新的学生年龄：");
142     String age = sc.nextLine();
143     System.out.println("请输入新的学生居住地：");
144     String address = sc.nextLine();
```

```

145         //创建学生对象
146         Student s = new Student();
147         s.setSid(sid);
148         s.setName(name);
149         s.setAge(age);
150         s.setAddress(address);
151         //遍历集合，修改对应的学生信息
152         for (int i = 0; i < array.size(); i++) {
153             Student student = array.get(i);
154             if (student.getSid().equals(sid)) {
155                 array.set(i, s);
156                 break;
157             }
158         }
159         //给出修改成功提示
160         System.out.println("修改学生成功!");
161     }
162
163     //定义一个方法，判断学号是否被使用
164     public static boolean isUsed(ArrayList<Student> array,
String sid) {
165         boolean flag = false;
166         for (int i = 0; i < array.size(); i++) {
167             Student s = array.get(i);
168             if (s.getSid().equals(sid)) {
169                 flag = true;
170                 break;
171             }
172         }
173         return flag;
174     }
175 }

```

 ALT+INSERT 根据自己的需要进行选择

## 继承

### ✧ 继承概述

继承是面向对象三大特征之一，可以使得子类具有父类的属性和方法，还可以在子类中重新定义，追加属性和方法。

### 继承的格式

- 格式：

```
1 public class 子类名 extends 父类名 {}
```

- 范例：

```
1 public class Zi extends Fu {}
```

- Fu：父类，也被称为基类、超类
- Zi：子类，也被称为派生类

### 继承中子类的特点：

- 子类可以有父类的内容
- 子类还可以有自己特有的内容

### 继承的好处和弊端

- 继承好处：
  - 提高了代码的 **复用性**（多个类相同的成员可以放到同一个类中）
  - 提高了代码的 **维护性**（如果方法的代码需要修改，修改一处即可）
- 继承弊端：
  - 继承让类与类之间产生了关系，类的耦合性增强了，当父类发生变化时子类实现也不得不跟着变化，削弱了子类的独立性

### 什么时候使用继承？

- 继承体现的关系：is a
- 假设法：我有两个类A和B，如果它们满足A是B的一种，或者B是A的一种，就说明它们存在继承关系，这个时候就可以考虑使用继承来体现，否则就不能滥用继承

### 继承中变量的访问特点



在子类方法中访问一个变量：

- 子类局部范围找
- 子类成员范围找
- 父类成员范围找
- 如果都没有就报错（不考虑父亲的父亲...）

## ✧ Super关键字

- `super` 关键字的用法和 `this` 关键字的用法类似
- `this`：代表本类对象的引用
- `super`：代表父类存储空间的标识（可以理解为父类对象引用）

**i** 三种用法：

1	<code>this.成员变量</code>	访问本类成员变量
2	<code>this(...)</code>	访问本类构造方法
3	<code>this.成员方法(...)</code>	访问本类成员方法
4		
5	<code>super.成员变量</code>	访问父类成员变量
6	<code>super(...)</code>	访问父类构造方法
7	<code>super.成员方法(...)</code>	访问父类成员方法

## ✧ 继承中构造方法的访问特点

**i** 继承中构造方法的访问特点：

- 子类中所有的构造方法默认都会访问父类中无参的构造方法
- 原因：
  - 1 因为子类会继承父类中的数据，可能还会使用父类的数据，所以，子类初始化之前，一定要先完成父类数据的初始化
  - 2 每一个子类构造方法的第一条语句默认都是：`super()`

**i** 如果父类中没有无参构造方法，只有带参构造方法：

- 通过使用 `super` 关键字去显式调用父类的带参构造方法
- 在父类中自己提供一个无参构造方法

推荐： `自己给出无参构造方法`

## ✧ 继承中成员方法的访问特点

**i** 继承中成员方法的访问特点：

- 通过子类对象访问一个方法
  - 子类成员范围找
  - 父类成员范围找
  - 如果都没有就报错（不考虑父亲的父亲...）

## ✧ 方法重写

**i** 方法重写概述：

子类中出现了和父类一模一样的方法声明

**i** 方法重写的应用：

当子类需要父类的功能，而功能主体子类有自己特有内容时，可以重写父类中的方法，这样，既沿袭了父类的功能，又定义了子类特有的内容

**i** `@Override`

- 是一个注解
- 可以帮助我们检查重写方法的方法声明的正确性

**i** 注意事项:

- 1 私有方法不能被重写（父类私有成员子类是不能继承的）
- 2 子类方法访问权限不能更低（public > 默认 > 私有）

## ✧ Java 中继承的注意事项

---

- 1 Java 中类只支持单继承，不支持多继承
- 2 Java 中类支持多层继承

# 修饰符

## ✧ 包

---

**i** 包的概述和使用:

- 包其实就是文件夹
- 作用：对类进行分类管理

**i** 包的定义格式:

- 格式： `package 包名;` (多级包用 `.` 分开)
- 范例： `package com.itheima;`

## ✧ 导包

---

```
1 cn.itcast.Teacher t = new cn.itcast.Teacher();
```

```
1 import cn.itcast.Teacher;
2
3 Teacher t = new Teacher();
```

## ✧ 修饰符

### 权限修饰符

修饰符	同一个类中	同一个包中子类无关类	不同包的子类	不同包的无关类
private	√			
默认	√	√		
protected	√	√	√	
public	√	√	√	√

### 状态修饰符

- **final** (最终态)
- **static** (静态)

#### **final**

**final** (最终态) 关键字是最终的意思，可以修饰成员方法，成员变量，类

#### **i** **final** 修饰的特点：

- 修饰方法：表明该方法是最最终方法，不能被重写
- 修饰变量：表明该变量是常量，不能再次被赋值
- 修饰类：表明该类是最最终类，不能被继承

#### **i** **final** 修饰局部变量：

- 变量是基本类型：final修饰指的是基本类型 的 数据值不能发生改变

```
1 final int age = 20;
```

- 变量是引用类型：final修饰指的是引用类型 的 地址值不能发生改变，但是地址值里面的内容是可以发生改变的

```
1 final Student s = new Student();
```

## static

static 关键字是静态的意思，可以修饰成员变量，成员方法

### static修饰的特点：

- 1 被类的所有对象共享 这也是我们判断是否使用静态关键字的条件
- 2 可以使用类名调用，也可以使用对象名调用，推荐使用类名调用

### static访问特点

- 非静态的成员方法：
  - 1 能访问静态的成员变量
  - 2 能访问非静态的成员变量
  - 3 能访问静态的成员方法
  - 4 能访问非静态的成员方法
- 静态的成员方法：
  - 1 能访问静态的成员变量
  - 2 能访问静态的成员方法

总结：静态成员方法只能访问静态成员

## 多态

## ✧ 多态概述

---

多态：同一个对象，在不同时刻表现出来的不同形态

**i** 多态的前提和体现：

- ① 有继承/实现关系
- ② 有方法重写
- ③ 有父类引用指向子类对象

## ✧ 多态中成员访问特点

---

- 成员变量：编译看左边，执行看左边
- 成员方法：编译看左边，执行看右边

**i** 为什么成员变量和成员方法的访问不一样呢？

因为成员方法有重写，而成员变量没有。

## ✧ 多态的好处和弊端

---

- 多态的好处：提高了程序的扩展性
  - 具体体现：定义方法的时候使用父类型作为参数，将来在使用的时候，使用具体的子类型参加操作
- 多态的弊端：不能使用子类的特有功能

## ✧ 多态中的转型

---

- ① 向上转型 从子到父 父类引用指向子类对象

```
1 Animal a = new Cat();
```

① 向下转型 从父到子 父类引用转为子类对象

```
1 Cat c = (Cat) a;
```

## ✧ 案例：猫和狗

---

```
1 Animal a = new Cat();
2 a.setName("加菲");
3 a.setAge(5);
4 System.out.println(a.getName() + "," + a.getAge());
5 a.eat();
6
7 a = new Cat("加菲", 5);
8 System.out.println(a.getName() + "," + a.getAge());
9 a.eat();
```

# 抽象类

## ✧ 抽象类概述

---

在Java中，一个 **没有方法体** 的方法应该定义为 **抽象方法**，而类中如果有 **抽象方法**，该类必须定义为 **抽象类**。

## ✧ 抽象类特点

---

① 抽象类和抽象方法必须使用 **abstract** 关键字修饰

- **public abstract void class 类名 {}**

- **public abstract void eat();**

- ② 抽象类中不一定有抽象方法，有抽象方法的类一定是抽象类
- ③ 抽象类不能直接实例化，但可以参照多态的方式，通过子类对象实例化，这叫抽象类多态
- ④ 抽象类的子类，要么重写抽象类中的所有抽象方法，要么写成抽象类

## ✧ 抽象类的成员特点

---

- ① 成员变量：可以是变量，也可以是常量
- ② 构造方法：有构造方法，但是不能实例化，构造方法的作用是用于子类访问父类数据的初始化
- ③ 成员方法：
  - 可以有抽象方法：限定子类必须完成某些动作
  - 也可以有非抽象方法：提高代码复用性

## ✧ 案例：猫和狗

---

```
1  Animal a = new Cat();
2  a.setName("加菲");
3  a.setAge(5);
4  System.out.println(a.getName() + "," + a.getAge());
5  a.eat();
6
7  System.out.println("-----");
8
9  a = new Cat("加菲", 5);
10 System.out.println(a.getName() + "," + a.getAge());
11 a.eat();
```

接口



## ✧ 接口概述

---

接口就是一种 **公共的规范标准**，只要符合规范标准，大家都可以通用。

Java中的接口更多的体现在 **对行为的抽象**。

## ✧ 接口特点

---

- ① 接口用关键字 `interface` 修饰 —— `public interface 接口名 {}`
- ② 类实现接口用 `implements` 表示 —— `public class 类名 implements 接口名 {}`
- ③ 接口不能直接实例化，要想实例化，需要 **参照多态的方式，通过实现类对象实例化**，这叫接口多态。
  - 多态的形式：具体类多态、抽象类多态、接口多态
  - 多态的前提：有继承或者实现关系；有方法重写；有父（类/接口）引用指向（子/实现）类对象。
- ④ 接口的实现类：
  - 要么重写接口中的所以抽象方法
  - 要么是抽象类

## ✧ 接口的成员特点

---

- ① 成员变量
  - **只能是常量**
  - 默认修饰符：`public static final`
- ② 构造方法
  - **接口没有构造方法**，因为接口主要是对行为进行抽象的，是没有具体存在。
  - **一个类如果没有父类，默认继承自Object类**

### 3 成员方法

- 只能是抽象方法
- 默认修饰符: `public abstract`

## ✧ 类和接口的关系

---

### i 类和类的关系:

继承关系，只能单继承，但是可以多层继承

```
1 public class Zi extends Fu {}
```

### i 类和接口的关系:

实现关系，可以单实现，也可以多实现，还可以在继承一个类的同时实现多个接口

```
1 public class InterImpl extends Object implements Inter1, Inter2,
  Inter3 {}
```

### i 接口和接口的关系:

继承关系，可以单继承，也可以多继承

```
1 public interface Inter3 extends Inter1, Inter2 {}
```

## ✧ 抽象类和接口的区别

---

### 1 成员区别:

- 抽象类 —— 变量，常量；有构造方法；有抽象方法；也有非抽象方法
- 接口 —— 常量；抽象方法

### 2 关系区别:

- 类与类 —— 继承，单继承
- 类与接口 —— 实现，可以单实现，也可以多实现

- 接口与接口 —— 继承，单继承，多继承

3 设计理念没区别：

- 抽象类 —— 对类抽象，包括属性、行为
- 接口 —— 对行为抽象，主要是行为

## 形参和返回值

### ✧ 类名作为形参和返回值

---

- 方法的形参是类名，实际需要的是该类的对象
- 方法的返回值是类名，其实返回的是该类的对象

### ✧ 抽象类名作为形参和返回值

---

- 方法的形参是抽象类名，实际需要的是该抽象类的子类对象
- 方法的返回值是抽象类名，其实返回的是该抽象类的子类对象

### ✧ 接口名作为形参和返回值

---

- 方法的形参是接口名，实际需要的是该接口的实现类对象
- 方法的返回值是接口名，其实返回的是该接口的实现类对象

## 内部类

## ✧ 内部类概述

内部类：就是在一个类中定义一个类。

**i** 内部类的定义格式：

```
1 public class 类名 {  
2     修饰符 class 类名 {  
3  
4     }  
5 }
```

**i** 内部类的访问特点：

- 1 内部类可以直接访问外部类的成员，包括私有
- 2 外部类要访问内部类的成员，必须创建对象

```
1 public class Demo {  
2     private int num = 10;  
3  
4     public class Inner {  
5         public void show() {  
6             System.out.println(num);  
7         }  
8     }  
9  
10    public void method() {  
11        //show();  
12  
13        Inner i = new Inner();  
14        i.show();  
15    }  
16 }
```

## ✧ 成员内部类

**i** 根据内部类在类中定义的位置不同，可以分为如下两种形式：

- ① 在类的成员位置：成员内部类
- ② 在类的局部位置：局部内部类

**i** 外部类创建对象使用内部类

- 格式： 外部类名.内部类名 对象名 = 外部类对象.内部类对象
- 范例：

```
1 Outer.Inner oi = new Outer().new Inner();
```

```
1 public class Outer {
2     private int num = 10;
3
4     /*public class Inner {
5         public void show() {
6             System.out.println(num);
7         }
8     }*/
9
10    private class Inner {
11        public void show() {
12            System.out.println(num);
13        }
14    }
15
16    public void method() {
17        Inner i = new Inner();
18        i.show();
19    }
20 }
```

```

1 public class InnerDemo {
2     public static void main(String[] args) {
3         //创建内部类对象，并调用方法
4
5         // public 修饰的内部类
6         // Outer.Inner oi = new Outer().new Inner();
7         // oi.show();
8
9         // private 修饰的内部类
10        Outer o = new Outer();
11        o.method();
12    }
13 }

```

## ✧ 局部内部类

- 局部内部类是在方法中定义的类，所以外界是无法直接使用的，需要要在方法内部创建对象并使用。
- 该类可以直接访问外部类的成员，也可以访问方法内的局部变量。

```

1 public class Outer {
2     private int num = 10;
3     public void method() {
4         int num2 = 20;
5
6         class Inner {
7             public void show() {
8                 System.out.println(num);
9                 System.out.println(num2);
10            }
11        }
12
13        Inner i = new Inner();
14        i.show();
15    }
16 }

```

```

1 public class Demo {
2     public static void main(String[] args) {
3         Outer o = new Outer();
4         o.method();
5     }
6 }

```

## ✧ 匿名内部类

- 前提：存在一个类或接口，这里的类可以是具体类，也可以是抽象类。
- 格式：

```

1 new 类名或者接口名() {
2     重写方法;
3 };

```

- 本质：是一个继承了该类或者实现了该接口的子类匿名对象。

```

1 public class Outer {
2     public void method() {
3         new Inter() {
4             @Override
5             public void show() {
6                 System.out.println("匿名内部类");
7             }
8         }.show();
9
10        new Inter() {
11            @Override
12            public void show() {
13                System.out.println("匿名内部类");
14            }
15        }.show();
16
17        System.out.println("-----");
18
19        Inter i = new Inter() {
20            @Override
21            public void show() {
22                System.out.println("匿名内部类");
23            }

```

```

24         };
25
26         i.show();
27         i.show();
28     }
29 }

```

```

1  public interface Inter {
2      void show();
3  }

```

```

1  public class Demo {
2      public static void main(String[] args) {
3          Outer o = new Outer();
4          o.method();
5      }
6  }

```

## 匿名内部类在开发中的使用

```

1  JumpingOperator jo = new JumpingOperator();
2
3  jo.method(new Jumping() {
4      @Override
5      public void jump() {
6          System.out.println("猫可以跳高了");
7      }
8  });
9
10 jo.method(new Jumping() {
11     @Override
12     public void jump() {
13         System.out.println("狗可以跳高了");
14     }
15 });

```

## 常用API



Math包含执行基本数字运算的方法。

**i** Math类的常用方法:

方法名	描述
public static int abs(int a)	返回参数的绝对值
public static double ceil(double a)	返回大于或等于参数的最小double值，等于一个整数
public static double floor(double a)	返回小于或等于参数的最大double值，等于一个整数
public static int round(float a)	按照四舍五入返回最接近参数的int
public static int max(int a, int b)	返回两个int值中的较大值
public static int min(int a, int b)	返回两个int值中的较小值
public static double pow(double a, double b)	返回a的b次幂的值
public static double random()	返回值为double的正值，[0.0,1.0)

```
1 //public static int abs(int a) 返回参数的绝对值
2 System.out.println(Math.abs(88));
3 System.out.println(Math.abs(-88));
4 System.out.println("-----");
5
6 //public static double ceil(double a) 返回大于或等于参数的最小double
  值，等于一个整数
7 System.out.println(Math.ceil(12.34));
8 System.out.println(Math.ceil(12.56));
9 System.out.println("-----");
10
11 //public static double floor(double a) 返回小于或等于参数的最大double
   值，等于一个整数
12 System.out.println(Math.floor(12.34));
13 System.out.println(Math.floor(12.56));
```

```

14 System.out.println("-----");
15
16 //public static int round(float a) 按照四舍五入返回最接近参数的int
17 System.out.println(Math.round(12.34));
18 System.out.println(Math.round(12.56));
19 System.out.println("-----");
20
21 //public static int max(int a, int b) 返回两个int值中的较大值
22 System.out.println(Math.max(66, 88));
23 System.out.println("-----");
24
25 //public static int min(int a, int b) 返回两个int值中的较小值
26 System.out.println(Math.min(66, 88));
27 System.out.println("-----");
28
29 //public static double pow(double a, double b) 返回a的b次幂的值
30 System.out.println(Math.pow(2.0, 3.0));
31 System.out.println("-----");
32
33 //public static double random() 返回值为double的正值, [0.0,1.0)
34 System.out.println(Math.random());
35 System.out.println((int) (Math.random() * 100) + 1);

```

## ✧ System

System包含几个有用的类字段和方法，它不能被实例化。

**i** System类的常用方法：

方法名	描述
public static void exit(int status)	终止当前运行的Java虚拟机，非零表示异常终止
public static long currentTimeMillis()	返回当前时间（以毫秒为单位）

```

1 System.out.println("开始");
2 //public static void exit(int status) 终止当前运行的Java虚拟机，非零表示异常终止

```

```

3 //System.exit(0);
4 System.out.println("结束");
5
6 //public static long currentTimeMillis() 返回当前时间 (以毫秒为单位)
7 System.out.println(System.currentTimeMillis());
8
9 System.out.println(System.currentTimeMillis() * 1.0 / 1000 / 60 /
10 60 / 24 / 365 + "年");
11
12 long start = System.currentTimeMillis();
13 for (int i = 0; i < 10000; i++) {
14     System.out.println(i);
15 }
16 long end = System.currentTimeMillis();
17 System.out.println("共耗时: " + (end - start) + "毫秒");

```

## ✧ Object类的toString方法

- Object是类层次结构的根，每个类都可以将Object类作为超类。所有类都直接或间接继承自该类。
- 构造方法： `public Object()`



看方法的源码，选中方法按 **CTRL+B**  
建议所有子类重写 `toString()` 方法

```

1 public class Student {
2     private String name;
3     private int age;
4
5     public Student() {
6     }
7
8     public Student(String name, int age) {
9         this.name = name;
10        this.age = age;
11    }
12
13    public String getName() {
14        return name;
15    }

```

```

16
17     public void setName(String name) {
18         this.name = name;
19     }
20
21     public int getAge() {
22         return age;
23     }
24
25     public void setAge(int age) {
26         this.age = age;
27     }
28
29     @Override
30     public String toString() {
31         return "Student{" +
32             "name='" + name + '\'' +
33             ", age=" + age +
34             '}';
35     }
36 }

```

```

1 Student s = new Student();
2 s.setName("林青霞");
3 s.setAge(30);
4 System.out.println(s);
5 System.out.println(s.toString());

```

## ✧ Object类的equals方法

**i** Object类的常用方法:

方法名	描述
public String toString()	返回对象的字符串表示形式。建议所有子类重写该方法，自动生成
public boolean equals(Object obj)	比较对象是否相等，默认比较地址，重写可以比较内容，自动生成

```
1 public class Student {
2     private String name;
3     private int age;
4
5     public Student() {
6     }
7
8     public Student(String name, int age) {
9         this.name = name;
10        this.age = age;
11    }
12
13    public String getName() {
14        return name;
15    }
16
17    public void setName(String name) {
18        this.name = name;
19    }
20
21    public int getAge() {
22        return age;
23    }
24
25    public void setAge(int age) {
26        this.age = age;
27    }
28
29    @Override
30    public String toString() {
31        return "Student{" +
32            "name='" + name + '\'' +
33            ", age=" + age +
34            '}';
35    }
36
37    @Override
38    public boolean equals(Object o) {
39        /*
40         this ---- s1
41         o ----- s2
42         */
43        //比较地址是否相同, 如果相同直接返回true
44        if (this == o) return true;
45        //判断参数是否为null 判断两个对象是否来自于同一个类
```

```

46         if (o == null || getClass() != o.getClass()) return
false;
47         //向下转型
48         Student student = (Student) o; //student = s2
49         //判断age是否相同
50         if (age != student.age) return false;
51         //比较name是否相同
52         return name != null ? name.equals(student.name) :
student.name == null;
53     }
54 }

```

```

1  Student s1 = new Student();
2  s1.setName("林青霞");
3  s1.setAge(30);
4
5  Student s2 = new Student();
6  s2.setName("林青霞");
7  s2.setAge(30);
8
9  //需求: 比较两个对象的值是否相同
10 System.out.println(s1 == s2); // false 比较的是地址值
11 System.out.println(s1.equals(s2)); // true

```

## ✧ 冒泡排序

- 排序：将一组数据按照固定的规则进行排列
- 冒泡排序：一种排序方式，对要进行排序的数据中相邻的数据进行两两比较，将较大的数据放到后面，依次对所有的数据进行操作，直至所有数据按照要求完成排序。

### 冒泡排序特点：

- 如果有n个数据进行排序，总共需要比较n-1次
- 每一次比较完毕，下一次的比较就会少一个数据参与

```

1  public class Demo {
2      public static void main(String[] args) {
3          int[] arr = {24, 69, 80, 57, 13};
4          System.out.println("排序前: " + arrayToString(arr));

```

```

5
6         for (int x = 0; x < arr.length - 1; x++) {
7             for (int i = 0; i < arr.length - 1 - x; i++) {
8                 if (arr[i] > arr[i + 1]) {
9                     int temp = arr[i];
10                    arr[i] = arr[i + 1];
11                    arr[i + 1] = temp;
12                }
13            }
14            System.out.println("第" + x + "次排序后: " +
arrayToString(arr));
15        }
16        System.out.println("排序后: " + arrayToString(arr)); //
[13, 24, 57, 69, 80]
17    }
18
19    public static String arrayToString(int[] arr) {
20        StringBuilder sb = new StringBuilder();
21        sb.append("[");
22        for (int i = 0; i < arr.length; i++) {
23            if (i == arr.length - 1) {
24                sb.append(arr[i]);
25            } else {
26                sb.append(arr[i]).append(",");
27            }
28        }
29        sb.append("]");
30        String s = sb.toString();
31        return s;
32    }
33 }

```

## ✧ Arrays 类

Arrays类包含用于操作数组的各种方法

方法名	描述
public static String toString(int[] a)	返回指定数组的内容的字符串表示形式
public static void sort(int[] a)	按照数字顺序排列指定的数组

**i** 工具类的设计思想:

- 构造方法用 `private` 修饰
- 成员用 `public static` 修饰

```
1 //定义一个数组
2 int[] arr = {24, 69, 80, 57, 13};
3 System.out.println("排序前: " + Arrays.toString(arr));
4
5 Arrays.sort(arr);
6 System.out.println("排序后: " + Arrays.toString(arr));
```

## 基本类型包装类

### ✧ 基本类型包装类

- 将基本数据类型封装成对象的好处在于可以在对象中定义更多的功能方法操作该数据
- 常用的操作之一：用于基本数据类型与字符串之间的转换

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



```
1 //需求: 判断一个数据是否在int范围内
2 System.out.println(Integer.MIN_VALUE);
3 System.out.println(Integer.MAX_VALUE);
```

## ✧ Integer

---

Integer:包装一个对象中的原始类型int的值

方法名	描述
public Integer(int value)	根据int值创建Integer对象（过时）
public Integer(String s)	根据String值创建Integer对象（过时）
public static Integer valueOf(int i)	返回表示指定的int值的Integer实例
public static Integer valueOf(String s)	返回一个保存指定值的Integer对象String

```
1 //public Integer(int value) 根据int值创建Integer对象（过时）
2 Integer i1 = new Integer(100);
3 System.out.println(i1);
4
5 //public Integer(String s) 根据String值创建Integer对象（过时）
6 Integer i2 = new Integer("100");    //字符串必须是数字字符串
7 System.out.println(i2);
8
9 //public static Integer valueOf(int i) 返回表示指定的int值的Integer
  实例
10 Integer i3 = Integer.valueOf(200);
11 System.out.println(i3);
12
13 //public static Integer valueOf(String s) 返回一个保存指定值的
  Integer对象String
14 Integer i4 = Integer.valueOf("200");
15 System.out.println(i4);
```

## ✧ int和String类型的相互转换

---

功能	方法名	描述
int转换为String	public static String valueOf(int i)	返回int参数的字符串表示形式。该方法是String类中的方法
String转换为int	public static int parseInt(String s)	将字符串解析为int类型。该方法是Integer类中的方法

```

1 //int ----> String
2 int number = 100;
3 //方式一
4 String s1 = "" + number;
5 System.out.println(s1);
6
7 //方式二
8 //public static String valueOf (int i)
9 String s2 = String.valueOf(number);
10 System.out.println(s2);
11 System.out.println("-----");
12
13 //String ----> int
14 String s = "100";
15 //方式一
16 //String ----> Integer ----> int
17 Integer i = Integer.valueOf(s);
18 //public int intValue ()
19 int x = i.intValue();
20 System.out.println(x);
21
22 //方式二
23 //public static int parseInt (String s)
24 int y = Integer.parseInt(s);
25 System.out.println(y);

```

## 案例：字符串中数据排序

需求：有一个字符串："91 27 46 38 50"，请写程序实现最终输出的结果是："27 38 46 50 91"

```

1 String s = "91 27 46 38 50";
2
3 //把字符串中的数字数据存储到一个int类型的数组中
4 String[] strArray = s.split(" ");

```

```

5   for (int i = 0; i < strArray.length; i++) {
6       System.out.println(strArray[i]);
7   }
8
9   //定义一个int数组, 把String[]数组中的每一个元素存储到int数组中
10  int[] arr = new int[strArray.length];
11  for (int i = 0; i < arr.length; i++) {
12      arr[i] = Integer.parseInt(strArray[i]);
13  }
14  for (int i = 0; i < arr.length; i++) {
15      System.out.println(arr[i]);
16  }
17
18  //对int数组排序
19  Arrays.sort(arr);
20
21  //将排序后的int数组中的元素进行拼接得到一个字符串, 这里拼接采用
    StringBuilder来实现
22  StringBuilder sb = new StringBuilder();
23  for (int i = 0; i < arr.length; i++) {
24      if (arr[i] == arr.length - 1) {
25          sb.append(arr[i]);
26      } else {
27          sb.append(arr[i]).append(" ");
28      }
29  }
30  String result = sb.toString();
31  System.out.println("result: " + result); // "27 38 46 50 91"

```

## ✧ 自动装箱和拆箱

- 装箱：把基本数据类型转换为对应的包装类类型
- 拆箱：把包装类类型转换为对应的基本数据类型



注意：在使用包装类类型的时候，如果做操作，最好先判断是否为null  
推荐只要是对象，在使用前就必须进行不为null的判断

```

1   //装箱
2   Integer i = Integer.valueOf(100);
3   //自动装箱
4   Integer ii = 100;

```

```

5
6 //拆箱
7 ii = ii.intValue() + 200;
8 System.out.println(ii);
9 //自动拆箱
10 ii += 200;
11 System.out.println(ii);
12
13 Integer iii = null;
14 if (iii != null) {
15     iii += 300; //NullPointerException 空指针
16 }

```

## ✧ 日期类

### Date

Date代表了一个特定的时间，精确到毫秒

构造方法	描述
public Date类()	分配一个Date对象，并初始化，以便它代表它被分配的时间，精确到毫秒
public Date类(long date)	分配一个Date对象，并将其初始化为表示从标准基准时间起指定的毫秒数

```

1 //public Date类() 分配一个Date对象，并初始化，以便它代表它被分配的时间，精确到毫秒
2 Date d1 = new Date();
3 System.out.println(d1);
4
5 System.out.println("-----");
6
7 //public Date类(long date) 分配一个Date对象，并将其初始化为表示从标准基准时间起指定的毫秒数
8 long date = 1000 * 60 * 60;
9 Date d2 = new Date(date);
10 System.out.println(d2);

```

## Date 类中的常用方法

方法名	描述
<code>public long getTime()</code>	获取的是日期对象从1970年1月1日00: 00: 00到现在的毫秒值
<code>public void setTime(long time)</code>	设置时间，给的是毫秒值

```
1 //创建日期对象
2 Date d = new Date();
3
4 //public long getTime() 获取的是日期对象从1970年1月1日00: 00: 00到现在的
  毫秒值
5 System.out.println(d.getTime());
6 System.out.println(d.getTime() * 1.0 / 1000 / 60 / 60 / 24 / 365
  + "年");
7 System.out.println("-----");
8
9 //public void setTime(long time) 设置时间，给的是毫秒值
10 System.out.println(d);
11
12 System.out.println("-----");
13
14 long time = 1000*60*60;
15 d.setTime(time);
16 System.out.println(d);
17
18 System.out.println("-----");
19
20 time = System.currentTimeMillis();
21 d.setTime(time);
22 System.out.println(d);
```

## SimpleDateFormat类

SimpleDateFormat类是一个具体的类，用于以区域设置敏感的方式格式化和解析日期。

日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，从‘A’到‘Z’以及从‘a’到‘z’引号的字母被解释为表示日期或时间字符串的组件的模式字母

**i** 常用的模式字母及对应关系如下：

- y —— 年
- M —— 月
- d —— 日
- H —— 时
- m —— 分
- s —— 秒

**i** 构造方法：

构造方法	描述
<code>public SimpleDateFormat()</code>	构造一个SimpleDateFormat，使用默认模式和日期格式
<code>public SimpleDateFormat(String pattern)</code>	构造一个SimpleDateFormat使用给定的模式和默认日期格式

○ 格式化日期(**Date** ---> **String**):

- `public final String format(Date date)` 将日期格式化成日期/时间字符串

○ 解析日期 (**String** ---> **Date**) :

- `public Date parse(String source)` 从给定字符串的开始解析文本以生成日期

```
1 //格式化
2 Date d = new Date();
3 SimpleDateFormat sdf = new SimpleDateFormat();
4 String s = sdf.format(d);
5 System.out.println(s);
6 System.out.println("-----");
7
8 sdf = new SimpleDateFormat("yyyy年MM月dd日 HH:mm:ss");
9 s = sdf.format(d);
10 System.out.println(s);
11
```

```

12 System.out.println("-----");
13
14 //解析
15 String ss = "2022-05-06 14:54:30";
16 SimpleDateFormat sdf2 = new SimpleDateFormat("yyyy-MM-dd
    HH:mm:ss");
17 Date d2 = sdf2.parse(ss);
18 System.out.println(d2);

```

## 案例：日期工具类

需求：定义一个日期工具类（DateUtils），包含两个方法：把日期转换为指定格式的字符串；把字符串解析为指定格式的日期。然后定义一个测试类，测试日期工具类的方法。

```

1  import java.text.ParseException;
2  import java.text.SimpleDateFormat;
3  import java.util.Date;
4
5  public class DateUtils {
6      private DateUtils() {}
7
8      public static String dateToString(Date date, String format) {
9          SimpleDateFormat sdf = new SimpleDateFormat(format);
10         String s = sdf.format(date);
11         return s;
12     }
13
14     public static Date stringToDate(String s, String format)
15     throws ParseException {
16         SimpleDateFormat sdf = new SimpleDateFormat(format);
17         Date d = sdf.parse(s);
18         return d;
19     }
20 }

```

```

1  import java.text.ParseException;
2  import java.util.Date;
3
4  public class Demo {
5      public static void main(String[] args) throws ParseException
6      {
7          Date d = new Date();
8      }
9  }

```

```

8      String s1 = DateUtils.dateToString(d, "yyyy年MM月dd日
    HH:mm:ss");
9      System.out.println(s1);
10
11     System.out.println("-----");
12
13     String s2 = DateUtils.dateToString(d, "yyyy年MM月dd日");
14     System.out.println(s2);
15     String s3 = DateUtils.dateToString(d, "HH:mm:ss");
16     System.out.println(s3);
17
18     System.out.println("-----");
19
20     String s = "2022-5-6 15:12:02";
21     Date dd = DateUtils.stringToDate(s, "yyyy-MM-dd
    HH:mm:ss");
22     System.out.println(dd);
23
24 }
25 }

```

## Calendar

Calendar为某一时刻和一组日历字段之间的转换提供了一些方法，并为操作日历字段提供了一些方法。

Calendar提供了一个类方法 `getInstance` 用于获取Calendar对象，其日历字段已使用当前日期和时间初始化：

```

1  Calendar rightNow = Calendar.getInstance();

1  //获取Calendar对象
2  Calendar c = Calendar.getInstance(); //多态的形式得到对象
3  // System.out.println(c);
4
5  //public int get(int field)
6  int year = c.get(Calendar.YEAR);
7  int month = c.get(Calendar.MONTH) + 1;
8  int day = c.get(Calendar.DATE);
9  System.out.println(year + "年" + month + "月" + day + "日");

```

**i** Calendar常用方法:



方法名	描述
public int get(int field)	返回给定日历字段的值
public abstract void add(int field, int amount)	根据日历的规则，将指定的时间量添加或减去给定的日历字段
public final void set(int year, int month, int date)	设置当前日历的年月日

```
1 //获取日历类对象
2 Calendar c = Calendar.getInstance();
3
4 //public int get(int field)
5 int year = c.get(Calendar.YEAR);
6 int month = c.get(Calendar.MONTH) + 1;
7 int day = c.get(Calendar.DATE);
8 System.out.println(year + "年" + month + "月" + day + "日");
9
10 System.out.println("-----");
11
12 //public abstract void add(int field, int amount) 根据日历的规则，将
    指定的时间量添加或减去给定的日历字段
13 //10年后的5天前
14 c.add(Calendar.YEAR, 10);
15 c.add(Calendar.DATE, -5);
16 year = c.get(Calendar.YEAR);
17 month = c.get(Calendar.MONTH) + 1;
18 day = c.get(Calendar.DATE);
19 System.out.println(year + "年" + month + "月" + day + "日");
20
21 System.out.println("-----");
22
23 //public final void set(int year, int month, int date) 设置当前日历
    的年月日
24 c.set(2048, 11, 11);
25 year = c.get(Calendar.YEAR);
26 month = c.get(Calendar.MONTH) + 1;
27 day = c.get(Calendar.DATE);
28 System.out.println(year + "年" + month + "月" + day + "日");
```

案例：二月天

需求：获取任意一年的二月份有多少天

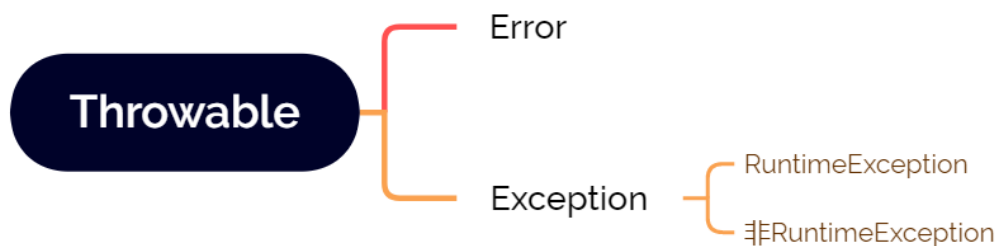
```
1 Scanner sc = new Scanner(System.in);
2 System.out.println("请输入你想查询的年份: ");
3 int year = sc.nextInt();
4
5 //设置日历对象的年月日
6 Calendar c = Calendar.getInstance();
7 c.set(year, 2, 1);
8
9 //3月1日往前推一天就是2月的最后一天
10 c.add(Calendar.DATE, -1);
11
12 //获取这一天并输出
13 int date = c.get(Calendar.DATE);
14
15 System.out.println(year + "年的2月份有" + date + "天");
```

## 异常

### ✧ 异常概述

异常：就是程序出现了不正常的情况

#### 异常体系



- Error: 严重问题，不需要处理

- Exception:称为异常类，它表示程序本身可以处理的问题
  - RuntimeException:在编译期间是不检查的，需要我们回来修改代码
  - 非RuntimeException:编译期间就必须处理，否则程序不能通过编译，就更不能正常运行了

## ✧ JVM的默认处理方案

**i** 如果程序出现了问题，我们没有做任何处理，最终JVM会做默认的处理

- 1 把异常的名称，异常原因及异常出现的位置等信息输出在了控制台
- 2 程序停止执行

## ✧ 异常处理之try...catch

格式：

```
1 try {  
2     可能出现异常的代码;  
3 } catch(异常类名 变量名) {  
4     异常的处理代码;  
5 }
```

**i** 执行流程：

- 1 程序从try里面的代码开始执行
- 2 出现异常，会自动生成一个异常类对象，该异常对象将被提交给Java时系统
- 3 当Java运行时系统接收到异常对象时，会到catch中去找匹配的异常类，找到后执行异常的处理
- 4 执行完毕之后，程序还可以继续往下执行

```
1 public static void main(String[] args) {  
2     System.out.println("开始");  
3     method();  
4     System.out.println("结束");  
}
```

```
5 }
6
7 public static void method() {
8     try {
9         int[] arr = {1, 2, 3};
10        System.out.println(arr[3]);
11    } catch (ArrayIndexOutOfBoundsException e) {
12        // System.out.println("你访问的数组索引不存在");
13        e.printStackTrace();
14    }
15 }
```

# ✧ Throwable的成员方法

方法名	描述
public String getMessage()	返回此throwable的详细消息字符串
public String toString()	返回此可抛出的简短描述
public void printStackTrace()	把异常的错误信息输出在控制台

```
1 public static void main(String[] args) {
2     System.out.println("开始");
3     method();
4     System.out.println("结束");
5 }
6
7 public static void method() {
8     try {
9         int[] arr = {1, 2, 3};
10        System.out.println(arr[3]);
11    } catch (ArrayIndexOutOfBoundsException e) {
12        // e.printStackTrace();
13
14        //public String getMessage() 返回此
15        //throwable的详细消息字符串
16        System.out.println(e.getMessage()); //Index 3
17        //out of bounds for length 3 返回出现异常的原因
18        System.out.println("-----");
19    }
```

```

19         //public String toString()                返回此可抛出的简
短描述
20         System.out.println(e.toString());
        //java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds
for length 3
21
22         System.out.println("-----");
23
24         //public void printStackTrace()            把异常的错误信息
输出在控制台
25         e.printStackTrace();
26         //java.lang.ArrayIndexOutOfBoundsException: Index 3 out
of bounds for length 3
27         // at 异常_93.Throwable的成员方法_4.method(Throwable的成员方
法_4.java:19)
28         // at 异常_93.Throwable的成员方法_4.main(Throwable的成员方法
_4.java:12)
29     }
30 }

```

## ✳ 编译时异常和运行时异常的区别

- Java中的异常分为两大类：编译时异常和运行时异常，也被称为受检异常和非受检异常。
- 所有的RuntimeException类及其子类被称为运行时异常，其他的异常都是编译时异常。
- 编译时异常：必须显示处理，否则程序就会发生错误，无法通过编译
- 运行时异常：无需显示处理，也可以和编译时异常一样处理

```

1  public static void main(String[] args) {
2      method();
3  }
4
5  //编译时异常
6  public static void method2() {
7      try {
8          String s = "2048-08-09";
9          SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-
dd");
10         Date d = sdf.parse(s);

```

```

11         System.out.println(d);
12     } catch (ParseException e) {
13         e.printStackTrace();
14     }
15 }
16
17 //运行时异常
18 public static void method() {
19     int[] arr = {1, 2, 3};
20     try {
21         System.out.println(arr[3]);
22     } catch (ArrayIndexOutOfBoundsException e) {
23         e.printStackTrace();
24     }
25 }

```

## ✧ 异常处理之throws

格式:

```
1 throws 异常类名;
```

**i** 注意: 这个格式是跟在方法的括号后面的

- 编译时异常必须要进行处理, 两种处理方案: try...catch...或者throws, 如果采用throws这种方案, 将来谁调用谁处理
- 运行时异常可以不处理, 出现问题后, 需要我们回来修改代码

```

1 public static void main(String[] args) {
2     System.out.println("开始");
3     method();
4     try {
5         method2();
6     } catch (ParseException e) {
7         throw new RuntimeException(e);
8     }
9     System.out.println("结束");
10 }
11
12 //运行时异常

```

```

13 public static void method() throws ArrayIndexOutOfBoundsException
14 {
15     int[] arr = {1, 2, 3};
16     System.out.println(arr[3]);
17 }
18 //编译时异常
19 public static void method2() throws ParseException {
20     String s = "2048-08-09";
21     SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
22     Date d = sdf.parse(s);
23     System.out.println(d);
24 }

```

## ✧ 自定义异常

格式:

```

1 public class 异常类名 extends Exception {
2     无参构造
3     带参构造
4 }

```

throws	throw
用在方法声明后面，跟的是异常类名	用在方法体内，跟的是异常对象名
表示抛出异常，由该方法的调用者来处理	表示抛出异常，由该方法体内的语句处理
表示出现异常的一种可能性，并不一定会发生这些异常	执行throw一定抛出了某种异常

```

1 public class ScoreException extends Exception {
2     public ScoreException() {};
3     public ScoreException(String message) {
4         super(message);
5     }
6 }

```

```
1 public class Teacher {
2     public void checkScore(int score) throws ScoreException {
3         if (score < 0 || score > 100) {
4             // throw new ScoreException();
5             throw new ScoreException("你给的分数有误, 分数应该在0-100
        之间");
6         } else {
7             System.out.println("分数正常");
8         }
9     }
10 }
```

```
1 public class Demo {
2     public static void main(String[] args) {
3         Scanner sc = new Scanner(System.in);
4         System.out.println("请输入分数: ");
5         int score = sc.nextInt();
6
7         Teacher t = new Teacher();
8         try {
9             t.checkScore(score);
10        } catch (ScoreException e) {
11            e.printStackTrace();
12        }
13    }
14 }
```

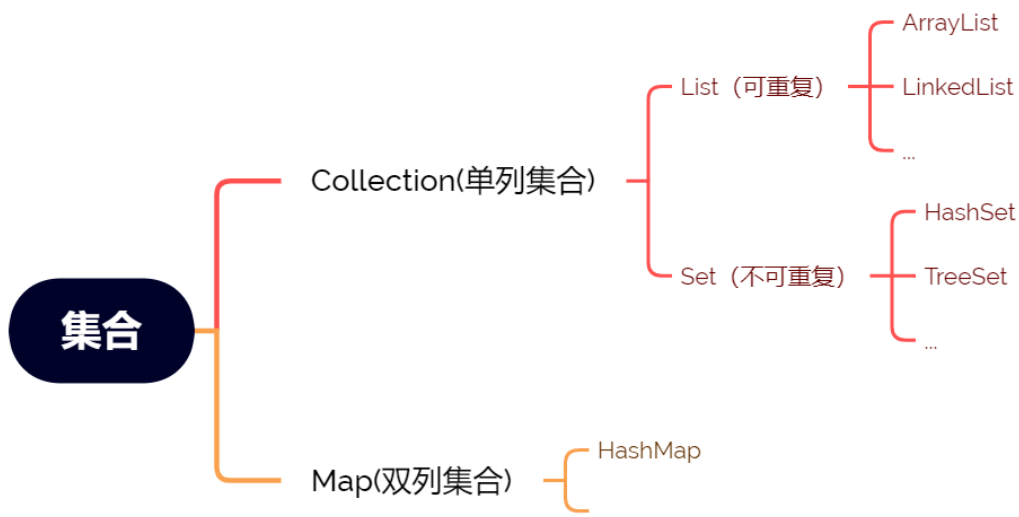
## 集合进阶

### Collection

---

集合类的特点：提供一种存储空间可变的存储模型，存储的数据容量可以随时发生改变





## Collection集合概述和使用

- 是单列集合的顶层接口，他表示一组对象，这些对象也称为Collection的元素
- JDK不提供此接口的任何直接实现，它提供更具体的子接口（如Set和List）实现

### **i** 创建Collection集合的对象：

- 多态的方式
- 具体的实现类ArrayList

```
1 //创建Collection集合的对象
2 Collection<String> c = new ArrayList<String>();
3
4 //添加元素: boolean add(E e)
5 c.add("hello");
6 c.add("world");
7 c.add("java");
8
9 //输出集合对象
10 System.out.println(c);
```

## Collection集合常用方法

方法名	描述
boolean add(E e)	添加元素
boolean remove(Object o)	从集合中移除指定的元素
void clear()	清空集合中的元素
boolean contains(Object o)	判断集合中是否存在指定的元素
boolean isEmpty()	判断集合是否为空
int size()	集合的长度，也就是集合中元素的个数

```

1 //创建Collection集合对象
2 Collection<String> c = new ArrayList<String>();
3
4 //boolean add(E e) 添加元素
5 //System.out.println(c.add("hello"));
6 //System.out.println(c.add("world"));
7 //System.out.println(c.add("world"));
8 c.add("hello");
9 c.add("world");
10 c.add("java");
11
12
13 //boolean remove(Object o) 从集合中移除指定的元素
14 System.out.println(c.remove("world"));
15 System.out.println(c.remove("javaee"));
16
17 //void clear() 清空集合中的元素
18 //c.clear();
19
20 //boolean contains(Object o) 判断集合中是否存在指定的元素
21 System.out.println(c.contains("world"));
22 System.out.println(c.contains("javaee"));
23
24 //boolean isEmpty() 判断集合是否为空
25 System.out.println(c.isEmpty());
26
27 //int size() 集合的长度，也就是集合中元素的个数
28 System.out.println(c.size());
29
30 //输出集合对象
31 System.out.println(c);

```

## Collection集合的遍历

- Iterator:迭代器，集合的专用遍历方式
- `Iterator<E> iterator()` : 返回此合集中元素的迭代器，通过集合的 `iterator()` 方法得到
- 迭代器是通过集合的 `iterator()` 方法得到的，所以我们说它是依赖于集合而存在的

### Iterator中的常用方法:

方法名	描述
<code>E next()</code>	返回迭代中的下一个元素
<code>boolean hasNext()</code>	如果迭代具有更多元素，则返回true

```
1  Collection<String> c = new ArrayList<String>();
2
3  c.add("hello");
4  c.add("world");
5  c.add("java");
6
7  //Iterator<E> iterator():返回此合集中元素的迭代器，通过集合的iterator()
  方法得到
8  Iterator<String> it = c.iterator();
9
10 //E next() 返回迭代中的下一个元素
11 /*System.out.println(it.next());
12 System.out.println(it.next());
13 System.out.println(it.next());*/
14
15 //boolean hasNext() 如果迭代具有更多元素，则返回true
16 /*if (it.hasNext()) {
17     System.out.println(it.next());
18 }
19 if (it.hasNext()) {
20     System.out.println(it.next());
21 }
22 if (it.hasNext()) {
23     System.out.println(it.next());
24 }
25 if (it.hasNext()) {
```

```

26     System.out.println(it.next());
27 }*/
28
29 //用while循环改进判断
30 while (it.hasNext()) {
31     // System.out.println(it.next());
32     String s = it.next();
33     System.out.println(s);
34 }

```

## Collection集合存储学生对象并遍历

需求：创建一个存储学生对象的集合，存储3个学生对象，使用程序实现在控制台遍历该集合。

```

1  import java.util.ArrayList;
2  import java.util.Collection;
3  import java.util.Iterator;
4
5  public class Demo {
6      public static void main(String[] args) {
7          //创建Collection集合对象
8          Collection<Student> c = new ArrayList<Student>();
9
10         //创建学生对象
11         Student s1 = new Student("林青霞", 30);
12         Student s2 = new Student("张曼玉", 35);
13         Student s3 = new Student("王祖贤", 33);
14
15         //把学生添加到集合
16         c.add(s1);
17         c.add(s2);
18         c.add(s3);
19
20         //遍历集合（迭代器方式）
21         Iterator<Student> it = c.iterator();
22         while (it.hasNext()) {
23             Student s = it.next();
24             System.out.println(s.getName() + "," + s.getAge());
25         }
26     }
27 }

```

## List集合概述

- 有序集合（也称为序列），用户可以精确控制列表中每个元素的插入位置。用户可以通过整数索引访问元素，并搜索列表中的元素。
- 与Set集合不同，列表通常允许重复的元素

### List集合特点:

- 有序：存储和取出的元素顺序一致
- 可重复：存储的元素可以重复

```
1 //创建集合对象
2 List<String> list = new ArrayList<String>();
3
4 //添加元素
5 list.add("hello");
6 list.add("world");
7 list.add("java");
8 list.add("world");
9
10 //输出集合对象
11 //System.out.println(list);
12
13 //遍历集合(迭代器)
14 Iterator<String> it = list.iterator();
15 while (it.hasNext()) {
16     String s = it.next();
17     System.out.println(s);
18 }
```

## List集合的特有方法

方法名	描述
<code>void add(int index, E element)</code>	在此集合中的指定位置插入指定的元素
<code>E remove(int index)</code>	删除指定索引处的元素，返回被删除的元素
<code>E set(int index, E element)</code>	修改指定索引处的元素，返回被修改的元素
<code>E get(int index)</code>	返回指定索引处的元素

```

1 List<String> list = new ArrayList<String>();
2
3 list.add("hello");
4 list.add("world");
5 list.add("java");
6
7 //void add(int index, E element) 在此集合中的指定位置插入指定的元素
8 list.add(1, "javaee");
9 //list.add(11, "javaee"); //IndexOutOfBoundsException 索引越界
10
11 //E remove(int index) 删除指定索引处的元素，返回被删除的元素
12 System.out.println(list.remove(1));
13
14 //E set(int index, E element) 修改指定索引处的元素，返回被修改的元素
15 System.out.println(list.set(1, "javaee"));
16
17 //E get(int index) 返回指定索引处的元素
18 System.out.println(list.get(1));
19
20 System.out.println(list);
21
22 //用for循环遍历集合
23 for (int i = 0; i < list.size(); i++) {
24     String s = list.get(i);
25     System.out.println(s);
26 }

```

## 案例：List集合存储学生对象并遍历

```

1 List<Student> list = new ArrayList<Student>();
2
3 Student s1 = new Student("林青霞", 30);
4 Student s2 = new Student("张曼玉", 35);
5 Student s3 = new Student("王祖贤", 33);

```

```

6
7 list.add(s1);
8 list.add(s2);
9 list.add(s3);
10
11
12 //for循环的方式遍历
13 for (int i = 0; i < list.size(); i++) {
14     Student s = list.get(i);
15     System.out.println(s.getName() + "," + s.getAge());
16 }
17
18 System.out.println("-----");
19
20 //迭代器的方式遍历
21 Iterator<Student> it = list.iterator();
22 while (it.hasNext()) {
23     Student s = it.next();
24     System.out.println(s.getName() + "," + s.getAge());
25 }

```

## 并发修改异常

并发修改异常 —— `ConcurrentModificationException`

**i** 产生原因:

迭代器遍历的过程中，通过集合对象修改了集合中元素的长度，造成了迭代器获取元素中判断预期修改值与实际修改值不一致

**i** 解决方案

用for循环遍历，然后用集合对象做对应的操作即可

```

1 List<String> list = new ArrayList<String>();
2
3 list.add("hello");
4 list.add("world");
5 list.add("java");
6
7 /*Iterator<String> it = list.iterator();
8 while (it.hasNext()) {
9     String s = it.next();

```

```

10     if (s.equals("world")) {
11         list.add("javaee");
12     }
13 }*/ //ConcurrentModificationException 当不允许这样的修改时，可以通过
    检测到对象的并发修改的方法来抛出此异常
14
15 for (int i = 0; i < list.size(); i++) {
16     String s = list.get(i);
17     if (s.equals("world")) {
18         list.add("javaee");
19     }
20 }
21
22 System.out.println(list);

```

## 列表迭代器

- ListIterator —— 列表迭代器
- 通过List集合的 `listIterator()` 方法得到，所以说它是List集合特有的迭代器
- 用于允许程序员沿任一方向遍历列表的列表的迭代器，在迭代期间修改列表，并获取列表中迭代器的当前位置

### 常用方法：

方法名	描述
E next()	返回列表中的下一个元素，并且前进光标位置
boolean hasNext()	如果此列表迭代器在向前方向遍历列表时具有更多元素，则返回 true
E previous()	返回列表中的上一个元素，并向后移动光标位置
boolean hasPrevious()	如果此列表迭代器在相反方向遍历列表时具有更多元素，则返回 true
void add(E e)	将指定的元素插入列表（可选操作）

```

1 List<String> list = new ArrayList<>();
2
3 list.add("hello");

```



```

4 list.add("world");
5 list.add("java");
6
7 ListIterator<String> lit = list.listIterator();
8 while (lit.hasNext()) {
9     String s = lit.next();
10    System.out.println(s);
11 }
12
13 System.out.println("-----");
14
15 //逆向遍历
16 while (lit.hasPrevious()) {
17     String s = lit.previous();
18     System.out.println(s);
19 }
20
21 System.out.println("-----");
22
23 //获取列表迭代器
24 while (lit.hasNext()) {
25     String s = lit.next();
26     if (s.equals("world")) {
27         lit.add("javaee");
28     }
29 }
30
31 System.out.println(list);

```

## 增强for循环

- 增强for：简化数组和Collection集合的遍历
- 实现此接口允许对象成为增强型 for语句的目标
- 它是JDK5之后出现的，其内部原理是一个Iterator迭代器

格式：

```

1 for (元素数据类型 变量名 : 数组或者Collection集合) {
2     //在此处使用变量即可，该变量就是元素
3 }

```

```

1 int[] arr = {1, 2, 3, 4, 5};
2 for (int i : arr) {

```

```

3      System.out.println(i);
4  }
5
6  System.out.println("-----");
7
8  String[] str = {"hello", "world", "java"};
9  for (String s : str) {
10     System.out.println(s);
11 }
12
13 System.out.println("-----");
14
15 List<String> list = new ArrayList<String>();
16 list.add("hello");
17 list.add("world");
18 list.add("java");
19
20 for (String s : list) {
21     System.out.println(s);
22 }
23
24 System.out.println("-----");
25
26 //内部原理是一个Iterator迭代器
27 for (String s : list) {
28     if (s.equals("world")) {
29         list.add("javaee"); //ConcurrentModificationException
30     }
31 }

```

## 案例：List集合存储学生对象用三种方式遍历

```

1  import java.util.ArrayList;
2  import java.util.Iterator;
3  import java.util.List;
4
5  public class Demo {
6      public static void main(String[] args) {
7          List<Student> list = new ArrayList<Student>();
8          Student s1 = new Student("林青霞", 30);
9          Student s2 = new Student("张曼玉", 35);
10         Student s3 = new Student("王祖贤", 33);
11
12         list.add(s1);

```

```

13         list.add(s2);
14         list.add(s3);
15
16         // 迭代器方式, 集合特有的遍历方式
17         Iterator<Student> it = list.iterator();
18         while (it.hasNext()) {
19             Student s = it.next();
20             System.out.println(s.getName() + "," + s.getAge());
21         }
22
23         System.out.println("-----");
24
25         // 普通for循环方式
26         for (int i = 0; i < list.size(); i++) {
27             Student s = list.get(i);
28             System.out.println(s.getName() + "," + s.getAge());
29         }
30
31         System.out.println("-----");
32
33         // 增强for循环
34         for (Student s : list) {
35             System.out.println(s.getName() + "," + s.getAge());
36         }
37     }
38 }

```

## 数据结构

### 栈和队列

- 数据结构是计算机存储、组织数据的方式，是指相互之间存在一种或多种特定关系的数组元素的集合。
- 通常情况下，精心选择的数据结构可以带来更高的运行或者存储效率。

#### **i** 栈:

- 数据进入栈模型的过程称为：压/进栈
- 数据离开栈模型的过程称为：弹/出栈
- 栈是一种数据先进后出的模型

### **i** 队列:

- 数据从后端进入队列模型的过程称为: 入队列
- 数据从前端进出队列模型的过程称为: 出队列
- 队列是一种数据先进先出的模型

## 数组和链表

### **i** 数组:

- 查询数据通过索引定位, 查询任意数据耗时相同, 查询速度快
- 删除数据时, 要将原始数据删除, 同时后面每个数据前移, 删除效率低
- 添加数据时, 添加位置后的每个数据后移, 再添加元素, 添加效率极低

### **i** 链表:

- 链表的每个元素称为结点, 一个结点包含: 结点的存数位置(地址)、存储具体的数据、下一个结点的地址
- 链表是一种 增删快 的模型(对比数组)
- 链表是一种 查询慢 的模型(对比数组)

## List集合子类特点

- List集合的常用子类: ArrayList、LinkedList
- ArrayList: 底层数据结构是数组, 查询快, 增删慢
- LinkedList: 底层数据结构是链表, 查询慢, 增删快

```
1 //需求: 分别使用ArrayList和LinkedList完成存储字符串并遍历
2 //创建集合对象
3 ArrayList<String> array = new ArrayList<String>();
4
5 array.add("hello");
6 array.add("world");
7 array.add("java");
8
9 //遍历
```

```

10  for (String s : array) {
11      System.out.println(s);
12  }
13
14  System.out.println("-----");
15
16  LinkedList<String> linkedlist = new LinkedList<String>();
17
18  linkedlist.add("hello");
19  linkedlist.add("world");
20  linkedlist.add("java");
21
22  for (String s : linkedlist) {
23      System.out.println(s);
24  }

```

## 案例：ArrayList集合存储学生对象用三种方式遍历

```

1  import java.util.ArrayList;
2  import java.util.Iterator;
3
4  public class Demo {
5      public static void main(String[] args) {
6          ArrayList<Student> array = new ArrayList<Student>();
7
8          Student s1 = new Student("林青霞", 30);
9          Student s2 = new Student("张曼玉", 35);
10         Student s3 = new Student("王祖贤", 33);
11
12         array.add(s1);
13         array.add(s2);
14         array.add(s3);
15
16         //迭代器
17         Iterator<Student> it = array.iterator();
18         while (it.hasNext()) {
19             Student s = it.next();
20             System.out.println(s.getName() + "," + s.getAge());
21         }
22
23         System.out.println("-----");
24
25         //for
26         for (int i = 0; i < array.size(); i++) {

```

```

27         Student s = array.get(i);
28         System.out.println(s.getName() + "," + s.getAge());
29     }
30
31     System.out.println("-----");
32
33     //增强for
34     for (Student s : array) {
35         System.out.println(s.getName() + "," + s.getAge());
36     }
37 }
38 }

```

## LinkedList集合的特有功能

方法名	描述
public void addFirst(E e)	在该列表开头插入指定的元素
public void addLast(E e)	将指定的元素追加到此列表的末尾
public E getFirst()	返回此列表中的第一个元素
public E getLast()	返回此列表中的最后一个元素
public E removeFirst()	从此列表中删除并返回第一个元素
public E removeLast()	从此列表中删除并返回最后一个元素

```

1  LinkedList<String> linkedList = new LinkedList<String>();
2
3  linkedList.add("hello");
4  linkedList.add("world");
5  linkedList.add("java");
6
7  linkedList.addFirst("javaee");
8  linkedList.addLast("javaee");
9
10 System.out.println("-----");
11
12 System.out.println(linkedList.getFirst());
13 System.out.println(linkedList.getLast());
14 System.out.println(linkedList);
15

```

```
16 System.out.println("-----");
17
18 System.out.println(linkedList.removeFirst());
19 System.out.println(linkedList.removeLast());
20 System.out.println(linkedList);
```

## ✧ Set

### Set集合概述和特点

- 不包含重复元素 的集合
- 没有带索引的方法，所以不能使用普通for循环遍历
- HashSet对集合的迭代顺序不做任何保证

```
1 //用Set集合存储字符串并遍历
2 Set<String> s = new HashSet<String>();
3
4 s.add("hello");
5 s.add("world");
6 s.add("java");
7
8 Iterator<String> it = s.iterator();
9 while (it.hasNext()) {
10     String s1 = it.next();
11     System.out.println(s1);
12 }
13
14 System.out.println("-----");
15
16 for (String s1 : s) {
17     System.out.println(s1);
18 }
```

### 哈希值

- 哈希值是JDK根据对象的 地址 或者 字符串 或者 数字 算出来的 int类型的数值
- Object类中有一个方法可以获取 对象的哈希值
  - public int hashCode() : 返回对象的哈希码值

### 对象哈希值的特点:

- 同一个对象多次调用 `hashCode()` 方法返回第哈希值是相同的
- 默认情况下, 不同对象的哈希值是不同的, 而重写 `hashCode()` 方法可以实现让不同对象的哈希值相同

```
1 Student s1 = new Student("林青霞", 30);
2
3 //同一对象多次调用hashCode()方法返回第哈希值是相同的
4 System.out.println(s1.hashCode());
5 System.out.println(s1.hashCode());
6
7 System.out.println("-----");
8
9 //默认情况下, 不同对象的哈希值是不相同的
10 //通过方法重写可以实现不同对象的哈希值是相同的
11 Student s2 = new Student("林青霞", 30);
12 System.out.println(s2.hashCode());
13
14 System.out.println("-----");
15
16 System.out.println("hello".hashCode()); //99162322
17 System.out.println("world".hashCode()); //113318802
18 System.out.println("java".hashCode()); //3254818
19
20 System.out.println("-----");
21
22 // 字符串中重写了hashCode()方法
23 System.out.println("重地".hashCode()); //1179395
24 System.out.println("通话".hashCode()); //1179395
```

## HashSet

### HashSet集合概述和特点

#### HashSet集合特点

- 底层数据结构是哈希表



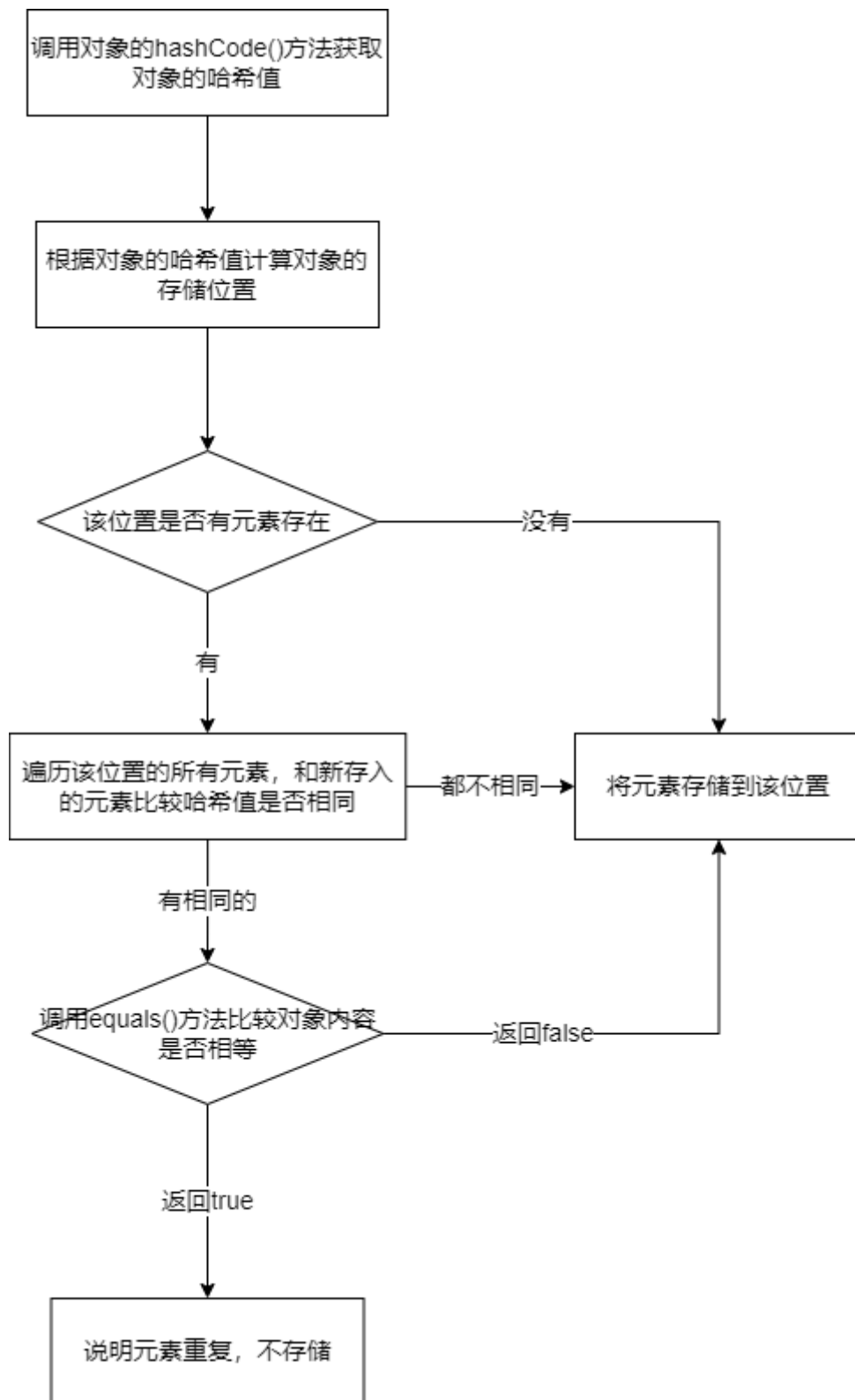
- 对集合的迭代顺序不作任何保证，也就是说不保证存储和取出的元素顺序一致。
- 没有带索引的方法，索引不能使用普通for循环遍历
- 由于是Set集合，索引是不包含重复元素的

```
1 //创建集合对象
2 HashSet<String> hs = new HashSet<String>();
3
4 // 添加元素
5 hs.add("hello");
6 hs.add("world");
7 hs.add("java");
8
9 // 遍历
10 for (String s : hs) {
11     System.out.println(s);
12 }
```

## HashSet集合保证元素唯一性源码分析

HashSet集合存储元素，要保证元素唯一性，需要重写 `hashCode()` 和 `equals()` 方法。

**i** HashSet 存储一个元素的过程：



```
1  import java.util.HashSet;
2  import java.util.Iterator;
3  import java.util.Set;
4
5  public class Demo {
6      public static void main(String[] args) {
7          //用Set集合存储字符串并遍历
8          Set<String> s = new HashSet<String>();
9
10         s.add("hello");
```

```

11         s.add("world");
12         s.add("java");
13
14         Iterator<String> it = s.iterator();
15         while (it.hasNext()) {
16             String s1 = it.next();
17             System.out.println(s1);
18         }
19
20         System.out.println("-----");
21
22         for (String s1 : s) {
23             System.out.println(s1);
24         }
25     }
26 }

```

```

1 //用Set集合存储字符串并遍历
2 Set<String> s = new HashSet<String>();
3
4 s.add("hello");
5 s.add("world");
6 s.add("java");
7 -----
8
9 public boolean add(E e) {
10     return map.put(e, PRESENT) == null;
11 }
12
13 static final int hash(Object key) {
14     int h;
15     return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
16 }
17
18 public V put(K key, V value) {
19     return putVal(hash(key), key, value, false, true);
20 }
21
22 //哈希值和元素的hashCode()方法相关
23 final V putVal(int hash, K key, V value, boolean
onlyIfAbsent, boolean evict) {
24     Node<K,V>[] tab; Node<K,V> p; int n, i;
25
26 //如果哈希表未进行初始化就对其进行初始化
27     if ((tab = table) == null || (n = tab.length) == 0)

```

```

28         n = (tab = resize()).length;
29
30 //根据对象的哈希值计算对象的存储位置, 如果该位置没有元素, 就存储元素
31         if ((p = tab[i = (n - 1) & hash]) == null)
32             tab[i] = newNode(hash, key, value, null);
33         else {
34             Node<K,V> e; K k;
35
36             /*
37              存入的元素和以前的元素比较哈希值
38              如果哈希值不同, 会继续向下执行, 把元素添加到集合
39              如果哈希值相同, 会调用对象的equals()方法比较
40              如果返回false, 会继续向下执行, 把元素添加到集合
41              如果返回true, 说明元素重复, 不存储
42             */
43             if (p.hash == hash &&
44                 ((k = p.key) == key || (key != null &&
45                     key.equals(k))))
46                 e = p;
47             else if (p instanceof TreeNode)
48                 e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash,
49                     key, value);
50             else {
51                 for (int binCount = 0; ; ++binCount) {
52                     if ((e = p.next) == null) {
53                         p.next = newNode(hash, key, value, null);
54                         if (binCount ≥ TREEIFY_THRESHOLD - 1) // -1
55                             for 1st
56                                 treeifyBin(tab, hash);
57                         break;
58                     }
59                     if (e.hash == hash &&
60                         ((k = e.key) == key || (key != null &&
61                             key.equals(k))))
62                         break;
63                     p = e;
64                 }
65             }
66             if (e != null) { // existing mapping for key
67                 V oldValue = e.value;
68                 if (!onlyIfAbsent || oldValue == null)
69                     e.value = value;
70                 afterNodeAccess(e);
71                 return oldValue;
72             }

```

```

69     }
70     ++modCount;
71     if (++size > threshold)
72         resize();
73     afterNodeInsertion(evict);
74     return null;
75 }

```

## 常见数据结构之哈希表

### 哈希表

- JDK8之前，底层采用 **数组+链表** 实现，可以说是一个元素为链表的数组
- JDK8之后，在长度比较长的时候，底层实现了优化

### 案例：HashSet集合存储学生对象并遍历

```

1  import java.util.HashSet;
2
3  public class Demo {
4      public static void main(String[] args) {
5          Student s1 = new Student("林青霞", 30);
6          Student s2 = new Student("张曼玉", 35);
7          Student s3 = new Student("王祖贤", 33);
8
9          Student s4 = new Student("王祖贤", 33);
10
11         HashSet<Student> hs = new HashSet<Student>();
12
13         hs.add(s1);
14         hs.add(s2);
15         hs.add(s3);
16
17         for (Student s : hs) {
18             System.out.println(s.getName() + ", " + s.getAge());
19         }
20     }
21 }

```

## LinkedHashSet

## LinkedHashSet集合概述和特点

### LinkedHashSet集合特点

- 哈希表和链表实现的Set接口，具有可预测的迭代次序
- 由链表保证元素有序，也就是说元素的存储和取出顺序是一致的
- 由哈希表保证元素唯一，也就是说没有重复的元素

```
1  LinkedHashSet<String> linkedHsahSet = new LinkedHashSet<String>();
2
3  linkedHsahSet.add("hello");
4  linkedHsahSet.add("world");
5  linkedHsahSet.add("java");
6
7  for (String s : linkedHsahSet) {
8      System.out.println(s);
9  }
```

## TreeSet

## TreeSet集合概述和特点

### TreeSet集合特点:

- 元素有序，这里的顺序不是指存储和取出的顺序，而是按照一定的规律进行排序，具体排序方法取决于构造方法。
- 没有带索引的方法，所以不能使用普通for循环遍历
- 由于是Set集合，所以不包含重复元素

### 构造方法

构造方法	描述
<code>TreeSet()</code>	根据元素的自然排序进行排序
<code>TreeSet(Comparator comparator)</code>	根据指定的比较器进行排序


### TreeSet集合存储整数并遍历

```
1  TreeSet<Integer> ts = new TreeSet<Integer>();
2  ts.add(10);
3  ts.add(40);
4  ts.add(30);
5  ts.add(50);
6  ts.add(20);
7
8  ts.add(30);
9
10 for (Integer i : ts) {
11     System.out.println(i); // 10 20 30 40 50
12 }
```

## 自然排序Comparable的使用

- 用TreeSet集合存储自定义对象，无参构造方法使用的是 **自然排序** 对元素进行排序的
- 自然排序，就是 **让元素所属的类实现Comparable接口**，重写compareTo(To)方法
- 重写方法时，一定要注意排序规则必须按照要求的主要条件和次要条件来写

需求：存储学生对象并遍历，创建TreeSet集合使用无参构造方法

-  要求：按照年龄从小到大进行排序，年龄相同时，按照姓名的字母顺序排序

```
1  public class Student implements Comparable<Student> {
2      private String name;
3      private int age;
4
5      public Student() {
6      }
7  }
```

```

8      public Student(String name, int age) {
9          this.name = name;
10         this.age = age;
11     }
12
13     public String getName() {
14         return name;
15     }
16
17     public void setName(String name) {
18         this.name = name;
19     }
20
21     public int getAge() {
22         return age;
23     }
24
25     public void setAge(int age) {
26         this.age = age;
27     }
28
29     @Override
30     public int compareTo(Student s) {
31         //      return 0; 只添加一个
32         //      return 1; 升序添加
33         //      return -1; 降序添加
34         //按照年龄从小到大进行排序
35         int num = this.age - s.age;
36         //年龄相同时, 按照姓名的字母顺序排序
37         int num2 = num == 0 ? this.name.compareTo(s.name) : num;
38         return num2;
39     }
40 }

```

```

1  TreeSet<Student> ts = new TreeSet<Student>();
2  Student s1 = new Student("xishi", 29);
3  Student s2 = new Student("wangzhaojun", 28);
4  Student s3 = new Student("diaochan", 30);
5  Student s4 = new Student("yangyuhuan", 33);
6
7  Student s5 = new Student("linqingxia", 33);
8  Student s6 = new Student("linqingxia", 33);
9
10 ts.add(s1);
11 ts.add(s2);

```



```

12 ts.add(s3);
13 ts.add(s4);
14 ts.add(s5);
15 ts.add(s6);
16
17 for (Student s : ts) {
18     System.out.println(s.getName() + "," + s.getAge());
19 }

```

## 比较器排序Comparator的使用

- 用TreeSet集合存储自定义对象，带参构造方法使用的是比较器排序对元素进行排序的
- 比较器排序，就是让集合构造方法接收Comparator的实现类对象，重写compare(To 1, To 2)方法
- 重写方法时，一定要注意排序规则必须按照要求的主要条件和次要条件来写

存储学生对象并遍历，创建TreeSet集合使用带参构造方法

- i** 要求：按照年龄从小到大进行排序，年龄相同时，按照姓名字母的字母顺序排序

```

1 import java.util.Comparator;
2 import java.util.TreeSet;
3
4 public class Demo {
5     public static void main(String[] args) {
6         TreeSet<Student> ts = new TreeSet<Student>(new
7         Comparator<Student>() {
8             @Override
9             public int compare(Student s1, Student s2) {
10                 int num = s1.getAge() - s2.getAge();
11                 int num2 = num == 0 ?
12                 s1.getName().compareTo(s2.getName()) : num;
13                 return num2;
14             }
15         });
16
17         Student s1 = new Student("xishi", 29);
18         Student s2 = new Student("wangzhaojun", 28);
19         Student s3 = new Student("diaochan", 30);
20         Student s4 = new Student("yangyuhuan", 33);

```

```

19
20     Student s5 = new Student("linqingxia", 33);
21     Student s6 = new Student("linqingxia", 33);
22
23     ts.add(s1);
24     ts.add(s2);
25     ts.add(s3);
26     ts.add(s4);
27     ts.add(s5);
28     ts.add(s6);
29
30     for (Student s : ts) {
31         System.out.println(s.getName() + "," + s.getAge());
32     }
33 }
34 }

```

## 案例：成绩排序

需求：用TreeSet集合存储多个学生信息（姓名、语文成绩、数学成绩），并遍历该集合

要求：按照总分从高到低出现

```

1  public class Student {
2      private String name;
3      private int chinese;
4      private int math;
5
6      public Student() {
7      }
8
9      public Student(String name, int chinese, int math) {
10         this.name = name;
11         this.chinese = chinese;
12         this.math = math;
13     }
14
15     public String getName() {
16         return name;
17     }
18
19     public void setName(String name) {
20         this.name = name;

```

```

21     }
22
23     public int getChinese() {
24         return chinese;
25     }
26
27     public void setChinese(int chinese) {
28         this.chinese = chinese;
29     }
30
31     public int getMath() {
32         return math;
33     }
34
35     public void setMath(int math) {
36         this.math = math;
37     }
38
39     public int getSum() {
40         return this.chinese + this.math;
41     }
42 }

```

```

1  import java.util.Comparator;
2  import java.util.TreeSet;
3
4  public class Demo {
5      public static void main(String[] args) {
6          TreeSet<Student> ts = new TreeSet<Student>(new
7  Comparator<Student>() {
8              @Override
9              public int compare(Student s1, Student s2) {
10                 //主要条件
11                 int num = s2.getSum() - s1.getSum();
12                 //次要条件
13                 int num2 = num == 0 ? s1.getChinese() -
14 s2.getChinese() : num;
15                 int num3 = num2 == 0 ?
16 s1.getName().compareTo(s2.getName()) : num2;
17                 return num3;
18             }
19         });
20
21         Student s1 = new Student("林青霞", 98, 100);
22         Student s2 = new Student("张曼玉", 95, 95);

```

```

20         Student s3 = new Student("王祖贤", 100, 93);
21         Student s4 = new Student("柳岩", 100, 97);
22         Student s5 = new Student("风清扬", 98, 98);
23
24         Student s6 = new Student("左冷禅", 97, 99);
25         Student s7 = new Student("赵云", 97, 99);
26
27         ts.add(s1);
28         ts.add(s2);
29         ts.add(s3);
30         ts.add(s4);
31         ts.add(s5);
32         ts.add(s6);
33         ts.add(s7);
34
35         for (Student s : ts) {
36             System.out.println(s.getName() + "," + s.getChinese()
+ "," + s.getMath() + "," + s.getSum());
37         }
38     }
39 }

```

### 案例：不重复的随机数

要求：编写一个程序，获取10个1-20之间的随机数，要求随机数不能重复，并在控制台输出。

```

1  import java.util.Random;
2  import java.util.Set;
3  import java.util.TreeSet;
4
5  public class Demo {
6      public static void main(String[] args) {
7          // Set<Integer> set = new HashSet<Integer>();
8          Set<Integer> set = new TreeSet<Integer>();
9
10         Random r = new Random();
11
12         while (set.size() < 10) {
13             int number = r.nextInt(20) + 1;
14             set.add(number);
15         }
16     }

```

```
17         for (Integer i : set) {
18             System.out.println(i);
19         }
20     }
21 }
```

## ✧ 泛型

### 泛型概述

- 泛型：是JDK中引入的特性，它提供了编译时类型安全检测机制，该机制允许在编译时检测到非法的类型。
- 它的本质是 **参数化类型**，也就是说所操作的数据类型被指定为一个参数
- 一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。
- 参数化类型，顾名思义，就是 **将参数类型由原来的具体的类型参数化，然后在使用/调用时传入具体的类型**。
- 这种参数类型可以用在类、方法和接口中，分别被称为泛型类、泛型方法、泛型接口

#### **i** 泛型定义格式：

- **<类型>**：指定一种类型的格式。这里的类型可以看成是形参
- **<类型1, 类型2, ... >**：指定多种类型的格式，多种类型之间用逗号隔开。这里的类型可以看成是形参
- 将来具体调用的时候给定的类型可以看成是实参，并且实参的数据类型只能是引用数据类型

#### **i** 泛型的好处：

- 把运行时的问题提前到了编译期间
- 避免了强制类型转换

### 泛型类

### 泛型类的定义格式:

- 格式: `修饰符 class 类名<类型> { }`
- 范例: `public class Generic<T> { }`
- 此处的 `T` 可以随便写为任意标识, 常见的如 `T`、`E`、`K`、`V` 等形式的参数常用于表示泛型

```
1 public class Generic<T> {  
2     private T t;  
3  
4     public T getT() {  
5         return t;  
6     }  
7  
8     public void setT(T t) {  
9         this.t = t;  
10    }  
11 }
```

```
1 Generic<String> g1 = new Generic<String>();  
2 g1.setT("林青霞");  
3 System.out.println(g1.getT());  
4  
5 Generic<Integer> g2 = new Generic<Integer>();  
6 g2.setT(30);  
7 System.out.println(g2.getT());
```

## 泛型方法

### 泛型方法的定义格式:

- 格式:

```
1 修饰符 <类型> 返回值类型 方法名(类型 变量名) {}
```

- 范例:

```
1 public <T> void sjow(T t) {}
```

```

1 public class Generic {
2     public <T> void show(T t) {
3         System.out.println(t);
4     }
5 }

```

```

1 Generic g = new Generic();
2 g.show("林青霞");
3 g.show(30);
4 g.show(true);
5 g.show(12.34);

```

## 泛型接口

**i** 泛型接口的定义格式:

○ 格式:

```

1 修饰符 interface 接口名<类型> {}

```

○ 范例:

```

1 public interface Generic<T> {}

```

```

1 public interface Generic<t> {
2     void show(T t);
3 }

```

```

1 public class GenericImpl<T> implements Generic<T> {
2     @Override
3     public void show(T t) {
4         System.out.println(t);
5     }
6 }

```

```

1 Generic<String> g1 = new GenericImpl<String>();
2 g1.show("林青霞");
3
4 Generic<Integer> g2 = new GenericImpl<Integer>();
5 g2.show(30);

```

## 类型通配符

为了表示各种泛型List的父类，可以使用类型通配符。

- 类型通配符： `<?>`
- `List<?>`：表示元素类型未知的List，它的元素可以匹配 任何的类型
- 这种带通配符的List仅表示它是各种泛型List的父类，并不能把元素添加到其中

#### 类型通配符上限

- 如果不希望List<?>是任何泛型List的父类，只希望它代表某一类泛型的父类，可以使用类型通配符的上限
- 类型通配符上限： `<? extends 类型>`
- `List<? extends Number>`：它表示的类型是 Number或者其子类型

#### 类型通配符下限

- 类型通配符下限： `<? super 类型>`
- `List<? super Number>`：它表示的是 Number或者其父类型

## 可变参数

可变参数又称参数个数可变，用作方法形参出现，那么方法参数个数就是可变的了。

- 格式：

```
1 修饰符 返回值类型 方法名(数据类型... 变量名) {}
```

- 范例：

```
1  public static int sum(int... a) {}
```

```
1  public static int sum(int... a) {
2      int sum = 0;
3      for (int i : a) {
4          sum += i;
5      }
6      return sum;
7  }
```



```
1 System.out.println(sum(10, 20));
2 System.out.println(sum(10, 20, 30));
3 System.out.println(sum(10, 20, 30, 40));
```

**i** 可变参数注意事项:

- 这里的变量其实是一个数组
- 如果一个方法有多个参数，包含可变参数，**可变参数要放在最后**

```
1 public static int sum(int b, int... a) {}
```

## 可变参数的使用

Arrays工具类中有一个静态方法:

- `public static <T> List<T> asList(T... a)` : 返回由指定数组支持的固定大小的列表
- 返回的集合不能做增删操作，可以做修改操作

List接口中有一个静态方法:

- `public static <E> List<E> of(E... elements)` : 返回包含任意数量元素的**不可变列表**
- 返回的集合不能做增删改操作

Set接口中有一个静态方法:

- `public static <E> Set<E> of(E... elements)` : 返回一个包含任意数量元素的**不可变集合**
- 在给元素的时候，不能给重复的元素
- 返回的集合不能做增删操作，没有修改的方法

## ✧ Map

### Map集合概述和特点

## Map结合概述

- `Interface Map<K, V>` K: 键的类型; V: 值的类型
- 将键映射到值的对象; 不能包含重复的键; 每个键可以映射到最多一个值

## 创建Map集合的对象

- 多态的方式
- 具体的实现类HashMap

```
1  import java.util.HashMap;
2  import java.util.Map;
3
4  public class MapDemo {
5      public static void main(String[] args) {
6          // 创建集合对象
7          Map<String, String> map = new HashMap<String, String>();
8
9          // V put (K key, V value) 将指定的值与该映射中的指定键相关联
10         map.put("itheima001", "林青霞");
11         map.put("itheima002", "张曼玉");
12         map.put("itheima003", "王祖贤");
13         map.put("itheima003", "柳岩"); // 替换了王祖贤
14
15         // 输出集合对象
16         System.out.println(map);
17     }
18 }
```

## Map集合的基本功能

方法名	描述
V put(K key, V value)	添加元素
V remove(Object key)	根据键删除键值对元素
void clear()	移除所有的键值对元素
boolean containsKey(Object key)	判断集合是否包含指定的键
boolean containsValue(Object value)	判断集合是否包含指定的值
boolean isEmpty()	判断集合是否为空
int size()	集合的长度，也就是集合中键值对的个数

## Map集合的获取功能

方法名	描述
V get(Object key)	根据键获取值
Set keySet()	获取所有键的集合
Collection values()	获取所有值的集合
Set<Map.Entry<K, V>> entrySet()	获取所有键值对对象的集合

## Map集合的遍历（方式一）

**i** 遍历思路：

- 获取所有键的集合，用 `keySet()` 方法实现
- 遍历键的集合，获取到每一个键，用 `增强for` 实现
- 根据键去找值，用 `get(Object key)` 方法实现

```

1 // 创建集合对象
2 Map<String, String> map = new HashMap<String, String>();
3
4 // 添加元素
5 map.put("张无忌", "赵敏");

```

```

6  map.put("郭靖", "黄蓉");
7  map.put("杨过", "小龙女");
8
9  // 获取所有键的集合
10 Set<String> keySet = map.keySet();
11
12 // 遍历键的集合, 获取到每一个键
13 for (String key : keySet) {
14     // 根据键去找值
15     String value = map.get(key);
16     System.out.println(key + "," + value);
17 }

```

## Map集合的遍历（方式二）

**i** 遍历思路:

- 获取所有键值对对象的集合, 用 `Set<Map.Entry<K, V>> entrySet()` 方法实现
- 遍历键值对对象的集合, 得到每一个键值对对象集合, 用 `增强for` 实现
- 根据键值对对象获取键和值, 用 `getKey()` 和 `getValue()` 方法实现

```

1  // 创建集合对象
2  Map<String, String> map = new HashMap<String, String>();
3
4  // 添加元素
5  map.put("张无忌", "赵敏");
6  map.put("郭靖", "黄蓉");
7  map.put("杨过", "小龙女");
8
9  // 获取所有键值对对象的集合
10 Set<Map.Entry<String, String>> entrySet = map.entrySet();
11
12 // 遍历键值对对象的集合, 得到每一个键值对对象集合
13 for (Map.Entry<String, String> me : entrySet) {
14     // 根据键值对对象获取键和值
15     String key = me.getKey();
16     String value = me.getValue();
17     System.out.println(key + "," + value);
18 }

```

## 案例：HashMap集合存储学生对象并遍历

```
1 // 创建集合对象
2 Map<String, Student> hm = new HashMap<String, Student>();
3
4 // 创建学生对象
5 Student s1 = new Student("林青霞", 30);
6 Student s2 = new Student("张曼玉", 35);
7 Student s3 = new Student("王祖贤", 33);
8
9 // 添加元素
10 hm.put("itheima001", s1);
11 hm.put("itheima002", s2);
12 hm.put("itheima003", s3);
13
14 // 方式一：键找值
15 Set<String> keySet = hm.keySet();
16 for (String key : keySet) {
17     Student value = hm.get(key);
18     System.out.println(key + "," + value.getName() + "," +
19         value.getAge());
20 }
21 System.out.println("-----");
22
23 // 方式二：键值对对象找键和值
24 Set<Map.Entry<String, Student>> entrySet = hm.entrySet();
25 for (Map.Entry<String, Student> me : entrySet) {
26     String key = me.getKey();
27     Student value = me.getValue();
28     System.out.println(key + "," + value.getName() + "," +
29         value.getAge());
30 }
```

## 案例：HashMap集合存储学生对象并遍历

需求：创建一个HashMap集合，键是学生对象(Student)，值是居住地(String)。存储多个键值对元素，并遍历。

要求保证键的唯一性：如果学生对象的成员变量相同，我们就认为是同一个对象。

```
1 import java.util.Objects;
2
3 public class Student {
```

```

4     private String name;
5     private int age;
6
7     public Student() {
8     }
9
10    public Student(String name, int age) {
11        this.name = name;
12        this.age = age;
13    }
14
15    public String getName() {
16        return name;
17    }
18
19    public void setName(String name) {
20        this.name = name;
21    }
22
23    public int getAge() {
24        return age;
25    }
26
27    public void setAge(int age) {
28        this.age = age;
29    }
30
31    @Override
32    public boolean equals(Object o) {
33        if (this == o) return true;
34        if (o == null || getClass() != o.getClass()) return
false;
35
36        Student student = (Student) o;
37
38        if (age != student.age) return false;
39        return Objects.equals(name, student.name);
40    }
41
42    @Override
43    public int hashCode() {
44        int result = name != null ? name.hashCode() : 0;
45        result = 31 * result + age;
46        return result;
47    }

```

```
1  import java.util.HashMap;
2  import java.util.Map;
3  import java.util.Set;
4
5  public class MapDemo {
6      public static void main(String[] args) {
7          // 创建集合对象
8          Map<Student, String> hm = new HashMap<Student, String>();
9
10         // 创建学生对象
11         Student s1 = new Student("林青霞", 30);
12         Student s2 = new Student("张曼玉", 35);
13         Student s3 = new Student("王祖贤", 33);
14         Student s4 = new Student("王祖贤", 33);
15
16         // 添加元素
17         hm.put(s1, "西安");
18         hm.put(s2, "武汉");
19         hm.put(s3, "郑州");
20         hm.put(s4, "北京");
21
22         // 方式一：键找值
23         Set<Student> keySet = hm.keySet();
24         for (Student key : keySet) {
25             String value = hm.get(key);
26             System.out.println(key.getName() + "," + key.getAge()
+ "," + value);
27         }
28
29         System.out.println("-----");
30
31         // 方式二：键值对对象找键和值
32         Set<Map.Entry<Student, String>> entrySet = hm.entrySet();
33         for (Map.Entry<Student, String> me : entrySet) {
34             Student key = me.getKey();
35             String value = me.getValue();
36             System.out.println(key.getName() + "," + key.getAge()
+ "," + value);
37         }
38     }
39 }
```

## 案例：ArrayList集合存储HashMap元素并遍历

需求：创建一个ArrayList集合，存储三个元素，每一个元素都是HashMap，每一个HashMap的键和值都是String，并遍历。

```
1 // 创建ArrayList集合
2 ArrayList<HashMap<String, String>> array = new
  ArrayList<HashMap<String, String>>();
3 // 创建HashMap集合并添加键值对元素
4 HashMap<String, String> hm1 = new HashMap<String, String>();
5 hm1.put("孙策", "大乔");
6 hm1.put("周瑜", "小乔");
7 // 把HashMap作为元素添加到ArrayList集合
8 array.add(hm1);
9
10 // 创建HashMap集合并添加键值对元素
11 HashMap<String, String> hm2 = new HashMap<String, String>();
12 hm1.put("郭靖", "黄蓉");
13 hm1.put("杨过", "小龙女");
14 // 把HashMap作为元素添加到ArrayList集合
15 array.add(hm2);
16
17 // 创建HashMap集合并添加键值对元素
18 HashMap<String, String> hm3 = new HashMap<String, String>();
19 hm1.put("令狐冲", "任盈盈");
20 hm1.put("林平之", "岳灵珊");
21 // 把HashMap作为元素添加到ArrayList集合
22 array.add(hm3);
23
24 // 遍历ArrayList集合
25 for (HashMap<String, String> hm : array) {
26     Set<String> keySet = hm.keySet();
27     for (String key : keySet) {
28         String value = hm.get(key);
29         System.out.println(key + "," + value);
30     }
31 }
```

## 案例：HashMap集合存储ArrayList元素并遍历

需求：创建一个HashMap集合，存储三个键值对元素，每一个键值对元素的键是String，值是ArrayList，每一个ArrayList的元素是String，并遍历。



```

1 // 创建HashMap集合
2 HashMap<String, ArrayList<String>> hm = new HashMap<String,
  ArrayList<String>>();
3
4 // 创建ArrayList集合, 并添加元素
5 ArrayList<String> sgyy = new ArrayList<String>();
6 sgyy.add("诸葛亮");
7 sgyy.add("赵云");
8
9 // 把ArrayList作为元素添加到HashMap集合
10 hm.put("三国演义", sgyy);
11
12 // 创建ArrayList集合, 并添加元素
13 ArrayList<String> xyj = new ArrayList<String>();
14 sgyy.add("唐僧");
15 sgyy.add("孙悟空");
16
17 // 把ArrayList作为元素添加到HashMap集合
18 hm.put("西游记", xyj);
19
20 // 创建ArrayList集合, 并添加元素
21 ArrayList<String> shz = new ArrayList<String>();
22 sgyy.add("武松");
23 sgyy.add("鲁智深");
24
25 // 把ArrayList作为元素添加到HashMap集合
26 hm.put("水浒传", shz);
27
28 // 遍历HashMap集合
29 Set<String> keySet = hm.keySet();
30 for (String key : keySet) {
31     System.out.println(key);
32     ArrayList<String> value = hm.get(key);
33     for (String s : value) {
34         System.out.println("\t" + s);
35     }
36 }

```

## 案例：统计字符串中每个字符出现的次数

需求：键盘录入一个字符串，要求统计字符串中每个字符出现的次数。

```

1 Scanner sc = new Scanner(System.in);
2 System.out.println("请输入一个字符串: ");

```

```

3 String line = sc.nextLine();
4
5 // HashMap<Character, Integer> hm = new HashMap<Character,
// Integer>(); // 无序
6 TreeMap<Character, Integer> hm = new TreeMap<Character, Integer>
(); // 自然顺序
7 for (int i = 0; i < line.length(); i++) {
8     char key = line.charAt(i);
9     Integer value = hm.get(key);
10
11     if (value == null) {
12         hm.put(key, 1);
13     } else {
14         value++;
15         hm.put(key, value);
16     }
17 }
18
19 StringBuilder sb = new StringBuilder();
20 Set<Character> keySet = hm.keySet();
21 for (Character key : keySet) {
22     Integer value = hm.get(key);
23     sb.append(key).append("(").append(value).append(")");
24 }
25
26 String result = sb.toString();
27 System.out.println(result);

```

## ✧ Collections

### Collections概述和使用

#### Collections类的概述

- 是针对集合操作的工具类

#### Collections类的常用方法

方法名	描述
<pre>public static &lt;T extends Comparable&lt;? super T&gt;&gt; void sort(List list)</pre>	将指定的列表按升序排序
<pre>public static void reverse(List&lt;?&gt; list)</pre>	反转指定列表中元素的顺序
<pre>public static void shuffle(List&lt;?&gt; list)</pre>	使用默认的随机源随机排列指定的列表

```

1  // 创建集合对象
2      List<Integer> list = new ArrayList<Integer>();
3
4      // 添加元素
5      list.add(30);
6      list.add(20);
7      list.add(50);
8      list.add(10);
9      list.add(40);
10
11     Collections.sort(list);
12     System.out.println(list); // [10, 20, 30, 40, 50]
13
14     Collections.reverse(list);
15     System.out.println(list); // [50, 40, 30, 20, 10]
16
17     Collections.shuffle(list);
18     System.out.println(list); // [30, 50, 40, 10, 20]
```

## 案例：Arraylist集合存储学生对象并排序

需求：ArrayList存储学生对象，使用Collections对ArrayList进行排序

要求：按照年龄从小到大进行排序，年龄相同时，按照姓名的字母顺序排序

```

1  ArrayList<Student> array = new ArrayList<Student>();
2
3  Student s1 = new Student("lingxia", 30);
4  Student s2 = new Student("zhangmanyu", 35);
5  Student s3 = new Student("wangzuxian", 33);
6  Student s4 = new Student("liuyan", 33);
7
8  array.add(s1);
9  array.add(s2);
```

```

10 array.add(s3);
11 array.add(s4);
12
13 Collections.sort(array, new Comparator<Student>() {
14     @Override
15     public int compare(Student s1, Student s2) {
16         int num = s1.getAge() - s2.getAge();
17         int num2 = num == 0 ?
18             s1.getName().compareTo(s2.getName()) : num;
19         return num2;
20     }
21 });
22
23 for (Student s : array) {
24     System.out.println(s.getName() + "," + s.getAge());
25 }

```

## 案例：模拟斗地主

需求：通过程序实现斗地主过程中的洗牌，发牌和看牌。

```

1 public static void main(String[] args) {
2     // 创建牌盒
3     ArrayList<String> array = new ArrayList<String>();
4
5     // 定义花色数组
6     String[] colors = {"♦", "♣", "♥", "♠"};
7     // 定义点数数组
8     String[] numbers = {"2", "3", "4", "5", "6", "7", "8", "9",
9         "10", "J", "Q", "K", "A"};
10    for (String color : colors) {
11        for (String number : numbers) {
12            array.add(color + number);
13        }
14    }
15    array.add("小王");
16    array.add("大王");
17
18    // 洗牌
19    Collections.shuffle(array);
20
21    // 发牌
22    ArrayList<String> lqxArray = new ArrayList<String>();
23    ArrayList<String> lyArray = new ArrayList<String>();

```

```

23     ArrayList<String> fqyArray = new ArrayList<String>();
24     ArrayList<String> dpArray = new ArrayList<String>();
25
26     for (int i = 0; i < array.size(); i++) {
27         String poker = array.get(i);
28
29         if (i ≥ array.size() - 3) {
30             dpArray.add(poker);
31         } else if (i % 3 == 0) {
32             lqxArray.add(poker);
33         } else if (i % 3 == 1) {
34             lyArray.add(poker);
35         } else if (i % 3 == 2) {
36             fqyArray.add(poker);
37         }
38     }
39
40     // 看牌
41     lookPoker("林青霞", lqxArray);
42     lookPoker("柳岩", lyArray);
43     lookPoker("风清扬", fqyArray);
44     lookPoker("底牌", dpArray);
45 }
46
47 // 看牌的方法
48 public static void lookPoker(String name, ArrayList<String>
array) {
49     System.out.print(name + "的牌是: ");
50     for (String poker : array) {
51         System.out.print(poker + " ");
52     }
53     System.out.println();
54 }

```

## 案例：模拟斗地主升级版

需求：通过程序实现斗地主过程中的洗牌，发牌和看牌。

要求：对牌进行排序。

```

1     public static void main(String[] args) {
2         HashMap<Integer, String> hm = new HashMap<Integer, String>();
3
4         ArrayList<Integer> array = new ArrayList<Integer>();
5

```

```
6     String[] colors = {"♦", "♣", "♥", "♠"};
7     String[] numbers = {"3", "4", "5", "6", "7", "8", "9", "10",
8         "J", "Q", "K", "A", "2"};
9
10    int index = 0;
11    for (String number : numbers) {
12        for (String color : colors) {
13            hm.put(index, color + number);
14            array.add(index);
15            index++;
16        }
17    }
18    hm.put(index, "小王");
19    array.add(index);
20    index++;
21
22    hm.put(index, "大王");
23    array.add(index);
24
25    Collections.shuffle(array);
26
27    TreeSet<Integer> lqxSet = new TreeSet<Integer>();
28    TreeSet<Integer> lySet = new TreeSet<Integer>();
29    TreeSet<Integer> fqySet = new TreeSet<Integer>();
30    TreeSet<Integer> dpSet = new TreeSet<Integer>();
31    for (int i = 0; i < array.size(); i++) {
32        int x = array.get(i);
33
34        if (i ≥ array.size() - 3) {
35            dpSet.add(x);
36        } else if (i % 3 == 0) {
37            lqxSet.add(x);
38        } else if (i % 3 == 1) {
39            lySet.add(x);
40        } else if (i % 3 == 2) {
41            fqySet.add(x);
42        }
43    }
44
45    lookPoker("林青霞", lqxSet, hm);
46    lookPoker("柳岩", lySet, hm);
47    lookPoker("风清扬", fqySet, hm);
48    lookPoker("底牌", dpSet, hm);
49 }
```

```
50
51 public static void lookPoker(String name, TreeSet<Integer> ts,
    HashMap<Integer, String> hm) {
52     System.out.print(name + "的牌是: ");
53     for (Integer key : ts) {
54         String poker = hm.get(key);
55         System.out.print(poker + " ");
56     }
57     System.out.println();
58 }
```

# IO流

## ✧ File

---

### File类概述和构造方法

- File: 它是文件和目录路径名的抽象表示。
- 文件和目录是可以通过File封装成对象的
- 对于File而言，其封装的并不是一个真正存在的文件，仅仅是一个路径名而已，它可以是存在的，也可以是不存在的，将来是要通过具体的操作把这个路径的内容转换为具体存在的。

#### File类构造方法

方法名	描述
<code>File(String pathname)</code>	通过将给定的路径名字符串转换为抽象路径名来创建新的File实例
<code>File(String parent, String child)</code>	从父路径名字符串和子路径名字符串创建新的File实例
<code>File(File parent, String child)</code>	从父抽象路径名和子路径名字符串创建新的File实例

```
1 File f1 = new File("E:\\itcast\\java.txt");
2 System.out.println(f1); // E:\itcast\java.txt
3
4 File f2 = new File("E:\\itcast", "java.txt");
5 System.out.println(f2); // E:\itcast\java.txt
6
7 File f3 = new File("E:\\itcast");
8 File f4 = new File(f3, "java.txt");
9 System.out.println(f4); // E:\itcast\java.txt
```

## File类创建功能

方法名	描述
<code>public boolean creatNewFile()</code>	当具有该名称的文件不存在时，创建一个由该抽象路径名命名的空文件
<code>public boolean mkdir()</code>	创建由此抽象路径名命名的目录
<code>public boolean mkdirs()</code>	创建由此抽象路径名命名的目录，包括任何必需但不存在的父目录



```

1 File f1 = new File("E:\\itcast\\java.txt");
2 System.out.println(f1.createNewFile()); // 文件不存在, 就创建文件, 返回
   true; 文件存在, 就不创建文件, 返回false
3
4 File f2 = new File("E:\\itcast\\JavaSE");
5 System.out.println(f2.mkdir()); // 目录不存在, 就创建目录, 返回true;
   目录存在, 就不创建目录, 返回false
6
7 File f3 = new File("E:\\itcast\\JavaWEB\\HTML");
8 System.out.println(f3.mkdirs()); // 创建多级目录
9
10 File f4 = new File("E:\\itcast\\javase.txt");
11 System.out.println(f4.createNewFile());

```

## File类判断和获取功能

方法名	描述
public boolean isDirectory()	测试此抽象路径名表示的File是否为目录
public boolean isFile()	测试此抽象路径名表示的File是否为文件
public boolean exists()	测试此抽象路径名表示的File是否存在
public String getAbsolutePath()	返回此抽象路径名的绝对路径名字符串
public String getPath()	将此抽象路径名转换为路径名字符串
public String getName()	返回由此抽象路径名表示的文件或目录的名称
public String[] list()	返回此抽象路径名表示的目录中的文件和目录的名称 字符串数组
public File[] listFiles()	返回此抽象路径名表示的目录中的文件和目录的File 对象数组

## File类删除功能

方法名	描述
<code>public boolean delete()</code>	删除由此抽象路径名表示的文件或目录

### 绝对路径和相对路径的区别

- 绝对路径：**完整的路径名**，不需要任何其他信息就可以定位它所表示的文件。例如：`E:\itcast\java.txt`
- 相对路径：必须使用取自其他路径名的信息进行解释。例如：`myFile\java.txt`

### 删除目录时的注意事项

- 如果一个**目录中有内容**（目录，文件），**不能直接删除**。应该先删除目录中的内容，最后才能删除目录

## 递归

递归概述：以编程的角度来看，递归指的是方法定义中调用方法本身的现象。

### 递归解决问题的思路

- 把一个复杂的问题层层转化为一个**与原问题相似的规模较小**的问题来求解
- 递归策略只需**少量的程序**就可以描述出解题过程所需要的多次重复计算

### 递归解决问题要找到两个内容：

- 递归出口：否则会出现内存溢出
- 递归规则：与原问题相似的规模较小的问题

### 不死神兔

```

1 public static int f(int n) {
2     if (n == 1 || n == 2) {
3         return 1;
4     } else {
5         return f(n-1) + f(n - 2);
6     }
7 }
8
9 System.out.println(f(20)); // 6765

```

### 案例：递归求阶乘

需求：用递归求5的阶乘，并把结果在控制台输出。

```

1 public static void main(String[] args) {
2     int result = jc(5);
3     System.out.println("5的阶乘是: " + result); // 120
4 }
5
6 public static int jc(int n) {
7     if (n == 1) {
8         return 1;
9     } else {
10        return n * jc(n - 1);
11    }
12 }

```

### 案例：遍历目录

需求：给定一个路径(E:\itcast)，请通过递归完成遍历该目录下的所有内容，并把所有文件的绝对路径输出在控制台。

```

1 public static void main(String[] args) {
2     File srcFile = new File("E:\\itcast");
3
4     getAllFilePath(srcFile);
5 }
6
7 public static void getAllFilePath(File srcFile) {
8     File[] fileArray = srcFile.listFiles();
9     if (fileArray != null) {
10        for (File file : fileArray) {

```

```
11         if (file.isDirectory()) {
12             getAllFilePath(file);
13         } else {
14             System.out.println(file.getAbsolutePath());
15         }
16     }
17 }
18 }
```

## ✧ 字节流

### IO流概述和分类

#### i IO流概述

- IO: 输入/输出(Input/Output)
- 流: 是一种抽象的概念, 是对数据传输的总称。也就是说数据在设备间的传输成为流, 流的本质是数据传输。
- IO流就是用来处理设备间数据传输问题的
  - 常见的应用: 文件复制、文件上传、文件下载

#### i IO流分类

- 按照数据的流向
  - 输入流: 读数据
  - 输出流: 写数据
- 按照数据类型来分
  - 字节流
    - 字节输入流, 字节输出流
  - 字符流
    - 字符输入流, 字符输出流

i 一般来说, 我们说IO流的分类是按照 **数据类型** 来分的

**i** 那么两种流都在什么情况下使用呢？

- 如果数据通过Windows自带的记事本软件打开，我们还 **可以读懂里面的内容，就使用字符流，否则使用字节流**。如果不知道该使用哪种类型的流，就使用字节流。

## 字节流写数据

**i** 字节流抽象基类

- **InputStream**：这个抽象类是表示字节输入流的所有类的超类
- **OutputStream**：这个抽象类是表示字节输出流的所有类的超类
- 子类名特点：子类名称都是以其父类名作为子类名的后缀

**i** **FileOutputStream**：文件输出流用于将数据写入File

- **FileOutputStream(String name)**：创建文件输出流以指定的名称写入文件

**i** 使用字节输出流写数据的步骤

- 创建字节输出流对象（调用系统功能创建了文件，创建字节输出流对象，让字节输出流对象指向文件）
- 调用字节输出流对象的写数据方法
- 释放资源（关闭此文件输出流并释放与此流相关联的任何系统资源）

```
1 // 创建字节输出流对象
2 FileOutputStream fos = new
  FileOutputStream("myByteStream\\fos.txt");
3 // 将指定的字节写入文件输出流
4 fos.write(97);
5 // 释放资源
6 fos.close();
```

## 字节流写数据的3种方式

方法名	描述
<code>void write(int b)</code>	将指定的字节写入此文件输出流，一次写一个字节数据
<code>void write(byte[] b)</code>	将 <b>b.length</b> 字节从指定的字节数组写入此文件输出流，一次写一个字节数组数据
<code>void write(byte[] b, int off, int len)</code>	将 <b>len</b> 字节从指定的字节数组开始，从偏移量 <b>off</b> 开始写入此文件输出流，一次写一个字节数组的部分数据

## 字节流写数据的两个小问题

### 字节流写数据如何实现换行？

- 写完数据后，加换行符
  - Windows: `\r\n`
  - Linux: `\n`
  - mac: `\r`

### 字节流写数据如何实现追加写入呢？

- `public FileOutputStream(String name, boolean append)`
- 创建文件输出流以指定的名称写入文件。如果第二个参数为 `true`，则字节将写入文件的末尾而不是开头

## 字节流写数据加异常处理

- `finally`：在异常处理时提供`finally`块来执行所有清除操作。比如说IO流中的释放资源。
- 特点：被`finally`控制的语句一定会执行，除非JVM退出

```

1  try {
2      可能出现异常的代码;
3  } catch (异常类名 变量名) {
4      异常的处理代码;
5  } finally {
6      执行所有清除操作;
7  }

```

```

1  FileOutputStream fos = null;
2
3  try {
4      fos = new FileOutputStream("myByteStream\\fos.txt");
5      fos.write("hello".getBytes());
6  } catch (IOException e) {
7      e.printStackTrace();
8  } finally {
9      if (fos != null) {
10         try {
11             fos.close();
12         } catch (IOException e) {
13             e.printStackTrace();
14         }
15     }
16 }

```

## 字节流读数据（一次读一个字节数据）

**i** FileInputStream: 从文件系统中的文件获取输入字节

- **FileInputStream(String name)**：通过打开与实际文件的连接来创建一个 **FileInputStream**，该文件由文件系统中的路径名 **name** 命名。

**i** 需求：把文件 **fos.txt** 中的内容读取出来在控制台输出。

```

1  FileInputStream fis = new
    FileInputStream("maByteStream\\fos.txt");
2  int by = fis.read();
3  System.out.println((char) by);
4  fis.close();

```

```

1  FileInputStream fis = new
   FileInputStream("myByteStream\\fos.txt");
2  int by;
3  while ((by = fis.read()) != -1) {
4      System.out.println((char) by);
5  }
6  fis.close();

```

## 案例：复制文本文件

需求：把“E:\itcast\窗里窗外.txt”复制到模块目录下的“窗里窗外.txt”

```

1  FileInputStream fis = new FileInputStream("E:\\itcast\\窗里窗
   外.txt");
2  FileOutputStream fos = new FileOutputStream("myByteStream\\窗里窗
   外.txt");
3
4  int by;
5  while ((by = fis.read()) != -1) {
6      fos.write(by);
7  }
8
9  fos.close();
10 fis.close();

```

## 字节流读数据（一次读一个字节数组数据）

需求：把文件fos.txt中的内容读取出来在控制台输出。

```

1  FileInputStream fis = new
   FileInputStream("myByteStream\\fos.txt");
2
3  byte[] bys = new byte[1024]; // 一般给1024及其整数倍
4  int len;
5  while ((len = fis.read(bys)) != -1) {
6      System.out.println(new String(bys, 0, len));
7  }
8
9  fis.close();

```

## 案例：复制图片



需求：把“E:\itcast\mn.jpg”复制到模块目录下的“mn.jpg”

```
1  FileInputStream fis = new
   FileInputStream("E:\\\\itcast\\\\mn.jpg");
2  FileOutputStream fos = new
   FileOutputStream("myByteStream\\mn.jpg");
3
4  byte[] bys = new byte[1024];
5  int len;
6  while ((len = fis.read(bys)) != -1) {
7      fos.write(bys, 0, len);
8  }
9
10 fis.close();
11 fos.close();
```

## 字节缓冲流

### i 字节缓冲流

- **BufferedOutputStream**：该类实现缓冲输出流。通过设置这样的输出流，应用程序可以向底层输出流写入字节，而不必为写入的每个字节导致底层系统的调用。
- **BufferedInputStream**：创建BufferedInputStream将创建一个内部缓冲区数组。当从流中读取或跳过字节时，内部缓冲区将根据需要从所包含的输入流中重新填充，一次很多字节。

### i 构造方法

- 字节缓冲输出流：**BufferedOutputStream(OutputStream out)**
- 字节缓冲输入流：**BufferedInputStream(InputStream in)**

### i 为什么构造方法需要的是字节流，而不是具体的文件或者路径呢？

- 字节缓冲流 **仅提供缓冲区**，而真正的读写数据还得依靠基本的字节流对象进行操作。

## 案例：复制视频

需求：把“E:\itcast\字节流复制图片.avi”复制到模块目录下的“字节流复制图片.avi”

```
1  public static void main(String[] args) throws IOException {
2      long startTime = System.currentTimeMillis();
3
4      method1(); // 共耗时64565毫秒
5      method2(); // 共耗时107毫秒
6      method3(); // 共耗时405毫秒
7      method4(); // 共耗时60毫秒
8
9      long endTime = System.currentTimeMillis();
10     System.out.println("共耗时: " + (endTime - startTime) + "毫
秒");
11 }
12
13 // 基本字节流一次读写一个字节
14 public static void method1() throws IOException {
15     FileInputStream fis = new FileInputStream("E:\\itcast\\字节流
复制图片.avi");
16     FileOutputStream fos = new FileOutputStream("muByteStream\\字
节流复制图片.avi");
17
18     int by;
19     while ((by = fis.read()) != -1) {
20         fos.write(by);
21     }
22
23     fos.close();
24     fis.close();
25 }
26
27 // 基本字节流一次读写一个字节数组
28 public static void method2() throws IOException {
29     FileInputStream fis = new FileInputStream("E:\\itcast\\字节流
复制图片.avi");
30     FileOutputStream fos = new FileOutputStream("muByteStream\\字
节流复制图片.avi");
31
32     byte[] bys = new byte[1024];
33     int len;
34     while ((len = fis.read(bys)) != -1) {
35         fos.write(bys, 0, len);
36     }
37
38     fos.close();
```

```

39     fis.close();
40 }
41
42 // 字节缓冲流一次读写一个字节
43 public static void method3() throws IOException {
44     BufferedInputStream bis = new BufferedInputStream(new
45     FileInputStream("E:\\itcast\\字节流复制图片.avi"));
46     BufferedOutputStream bos = new BufferedOutputStream(new
47     FileOutputStream("muByteStream\\字节流复制图片.avi"));
48
49     int by;
50     while ((by = bis.read()) != -1) {
51         bos.write(by);
52     }
53
54     bos.close();
55     bis.close();
56 }
57
58 // 字节缓冲流一次读写一个字节数组
59 public static void method4() throws IOException {
60     BufferedInputStream bis = new BufferedInputStream(new
61     FileInputStream("E:\\itcast\\字节流复制图片.avi"));
62     BufferedOutputStream bos = new BufferedOutputStream(new
63     FileOutputStream("muByteStream\\字节流复制图片.avi"));
64
65     byte[] bys = new byte[1024];
66     int len;
67     while ((len = bis.read(bys)) != -1) {
68         bos.write(bys, 0, len);
69     }
70
71     bos.close();
72     bis.close();
73 }

```

## ✧ 字符流

### 为什么出现字符流

由于字节流操作中文不是特别方便，所以java就提供了字符流

- 字符流 = 字节流 + 编码表



用字节流复制文本文件时，文本文件也会有中文，但是没有问题，原因是最终底层操作会自动将字节拼接成中文，如何识别是中文的呢？

- 汉字在存储的时候，无论选择哪种编码存储，第一个字节都是负数

## 编码表



### 基础知识

- 计算机中储存的信息都是用 二进制 数表示的，我们在屏幕上看到的英文、汉字等字符是二进制数转换之后的结果
- 按照某种规则，将字符存储到计算机中，成为 编码 。反之，将在存储在计算机中的二进制数按照某种规则解析显示出来，称为 解码 。这里强调一下：按照A编码存储，必须按照A编码解析，这样才能显示正确的文本符号，否则就会导致乱码现象。
  - 字符编码：就是一套自然语言的字符与二进制数之间的对应规则（A —— 65）

1	HEX — 十六进制
2	DEC — 十进制
3	OCT — 八进制
4	BIN — 二进制



### 字符集

- 是一个系统支持的所有字符的集合，包括各国家文字、标点符号、图形符号、数字等
- 计算机要准确的存储和识别各种字符集符号，就需要进行字符编码，一套字符集必然至少有一套字符编码。常见的字符集有ASCII字符集、GBXXX字符集、Unicode字符集等



### ASCII字符集

- ASCII （American Standard Code for Information Interchange，美国信息交换标准代码）：是基于拉丁字母的一套电脑编码系统，用于显示现代英语，主要包

括控制字符（回车键、退格、换行键等）和可显示字符（英文大小写字符、阿拉伯数字和西文符号）

- 基本的ASCII字符集，使用7位表示一个字符，共128字符。ASCII的扩展字符集使用8位表示一个字符，共258字符，方便支持欧洲常用字符。

### GBXXX字符集

- GB2312：简体中文码表。一个小于127的字符的意义与原来相同，但两个大于127的字符连在一起时，就表示一个汉字，这样大约可以组合了包含7000多个简体汉字，此外数学符号、罗马希腊的字母、日文的假名等都编进去了，连在ASCII里面本来就有的数字、标点、字母通通重新编了两个字节长的编码，这就是常说的“全角”字符，而原来在127号以下的那些就叫“半角”字符了
- **GBK**：最常用的中文码表。实在GB2312标准基础上的扩展规范，使用了双字节编码方案，共收录了21003个汉字，完全兼容GB2312标准，同时支持繁体汉字以及日韩汉字等
- GB18030：最新的中文码表。收录汉字70244个，采用多字节编码，每个字可以由1个、2个或4个字节组成。支持中国国内少数民族的文字，同时支持繁体汉字以及日韩汉字等

### Unicode字符集

- 为表达任意语言的任意字符而设计，是业界的一种标准，也称为统一码、标准万国码。它最多使用4个字节的数字来表达每个字母、符号，或者文字。有三种编码方案，UTF-8、UTF-16和UTF32.最为常用的是UTF-8编码
- **UTF-8** 编码：可以用来表示Unicode标准中任意字符，它是电子邮件、网页及其他存储或传送文字的应用中，优先采用的编码。互联网工作小组（IETF）要求所有互联网协议都必须支持UTF-8编码。它使用1-4个字节为每个字符编码
- 编码规则：
  - 128个US-ASCII字符，只需1个字节编码
  - 拉丁文等字符，需要2个字节编码
  - 大部分常用字（含中文），使用3个字符编码
  - 其他极少使用的Unicode辅助字符，使用4个字节编码

 小结：采用何种规则编码，就要采用对应规则解码，否则就会出现乱码

## 字符串中的编码解码问题

### i 编码

- `byte[] getBytes()`：使用平台的默认字符集将该String编码为一系列字节，将结果存储到新的字节数组中
- `byte[] getBytes(String charsetName)`：使用指定的字符集将该String编码为一系列字节，将结果存储到新的字节数组中

### i 解码

- `String(byte[] bytes)`：通过使用平台的默认字符集解码指定的字节数组来构造新的String
- `String(byte[] bytes, String charsetName)`：通过指定的字符集解码指定的字节数组来构造新的String

## 字符流中的编码解码问题

### i 字符流抽象基类

- **Reader**：字符输入流的抽象类
- **Writer**：字符输出流的抽象类

### i 字符流中和编码解码问题相关的两个类

- `InputStreamReader`
- `OutputStreamWriter`

## 字符流写数据的5中方式

方法名	描述
<code>void write(int c)</code>	写一个字符
<code>void write(char[] cbuf)</code>	写入一个字符数组
<code>void write(char[] cbuf, int off, int len)</code>	写入字符数组的一部分
<code>void write(String str)</code>	写一个字符串
<code>void write(String str, int off, int len)</code>	写一个字符串的一部分

方法名	描述
<code>flush()</code>	刷新流，还可以继续写数据
<code>close()</code>	关闭流，释放资源，但是在关闭之前会先刷新流。一旦关闭，就不能再写数据

## ✧ 字符流读数据的2种方式

方法名	描述
<code>int read()</code>	一次读一个字符数据
<code>int read(char[] cbuf)</code>	一次读一个字符数组数据

## ✧ 字符流复制Java文件

需求：把模块目录下的“ConversionStreamDemo.java”复制到模块目录下的“Copy.java”

```

1  InputStreamReader isr = new InputStreamReader(new
   FileInputStream("myByteStream\\ConversionStreamDemo.java"));
2  OutputStreamWriter osw = new OutputStreamWriter(new
   FileOutputStream("myByteStream\\Copy.java"));
3
4  // 一次读写一个字符数据

```

```

5   int ch;
6   while ((ch = isr.read()) != -1) {
7       osw.write(ch);
8   }
9   osw.close();
10  isr.close();
11
12  // 一次读写一个字符数组
13  char[] chs = new char[1024];
14  int len;
15  while ((len = isr.read()) != -1) {
16      osw.write(chs, 0, len);
17  }
18  osw.close();
19  isr.close();

```

## 案例：复制Java文件（改进版）

需求：把模块目录下的“ConversionStreamDemo.java”复制到模块目录下的“Copy.java”

```

1   FileReader fr = new
    FileReader("myByteStream\\ConversionStreamDemo.java");
2   FileWriter fw = new FileWriter("myByteStream\\Copy.java");
3
4   // 一次读写一个字符数据
5   int ch;
6   while ((ch = fr.read()) != -1) {
7       fw.write(ch);
8   }
9   fw.close();
10  fr.close();
11
12  // 一次读写一个字符数组
13  char[] chs = new char[1024];
14  int len;
15  while ((len = fr.read()) != -1) {
16      fw.write(chs, 0, len);
17  }
18  fw.close();
19  fr.close();

```

## 字符缓冲流



- **BufferWriter**：将文本写入字符数输出流，缓冲字符，以提供单个字符，数组和字符串的高效写入，可以指定缓冲区大小，或者可以接受默认大小，默认值足够大，可用于大多数用途
- **BufferReader**：从字符输入流读取文本，缓冲字符，以提供字符，数组和行的高效读取，可以指定缓冲区大小，或者可以使用默认大小，默认值足够大，可用于大多数用途

#### **i** 构造方法

- **BufferWriter(Writer out)**
- **BufferReader(Reader in)**

## 案例：复制Java文件（字符缓冲流改进版）

需求：把模块目录下的“ConversionStreamDemo.java”复制到模块目录下的“Copy.java”

```
1  BufferedReader br = new BufferedReader(new
   FileReader("myByteStream\\ConversionStreamDemo.java"));
2  BufferedWriter bw = new BufferedWriter(new
   FileWriter("myByteStream\\Copy.java"));
3
4  // 一次读写一个字符数据
5  int ch;
6  while ((ch = br.read()) != -1) {
7      bw.write(ch);
8  }
9  bw.close();
10 br.close();
11
12 // 一次读写一个字符数组
13 char[] chs = new char[1024];
14 int len;
15 while ((len = br.read()) != -1) {
16     bw.write(chs, 0, len);
17 }
18 bw.close();
19 br.close();
```

## 字符缓冲流特有功能

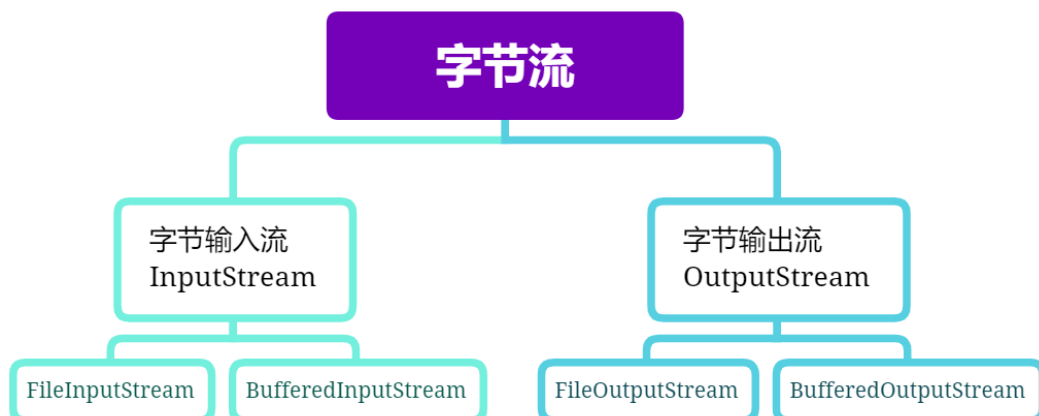
- `BufferedReader`
  - `void newLine()`：写一行行分隔符，行分隔符字符串由系统属性定义
- `BufferWriter`
  - `public String readLine()`：读一行文字。结果包含行的内容的字符串，不包括任何终止字符，如果流的结尾已经到达，则为`null`

## 案例：字符缓冲流复制Java文件

需求：把模块目录下的“ConversionStreamDemo.java”复制到模块目录下的“Copy.java”

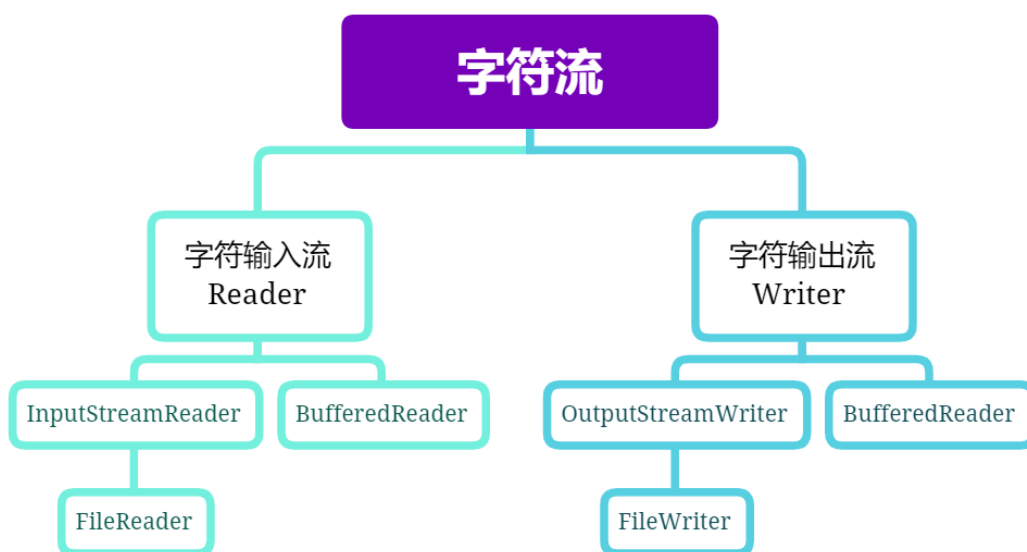
```
1  BufferedReader br = new BufferedReader(new
   FileReader("myByteStream\\ConversionStreamDemo.java"));
2  BufferedWriter bw = new BufferedWriter(new
   FileWriter("myByteStream\\Copy.java"));
3
4  String line;
5  while ((line = br.readLine()) != null) {
6      bw.write(line);
7      bw.newLine();
8      bw.flush();
9  }
10
11 bw.close();
12 br.close();
```

## ✧ IO流小结



i

小结：字节流可以复制任意文件数据，有4种方式，一般采用字节缓冲流一次读写一个字节数组的方式



i

小结：字符流只能复制文本数据，有5种方式，一般采用字符缓冲流的特有方式

## ✧ 案例：集合到文件

需求：把ArrayList集合中的字符串数据写入到文本文件。

要求：每一个字符串元素做为文本中的一行数据。

```
1 ArrayList<String> array = new ArrayList<String>();
2
3 array.add("hello");
4 array.add("world");
5 array.add("java");
6
7 BufferedWriter bw = new BufferedWriter(new
  FileWriter("myByteStream\\array.txt"));
8
9 for (String s : array) {
10     bw.write(s);
11     bw.newLine();
12     bw.flush();
13 }
```

```
13 }  
14  
15 bw.close();
```

## 案例：文件到集合

需求：把文本文件的数据读取到集合中，并遍历集合。

要求：文件中每一行数据就是一个集合元素。

```
1  BufferedReader br = new BufferedReader(new  
   FileReader("myByteStream\\array.txt"));  
2  
3  ArrayList<String> array = new ArrayList<String>();  
4  
5  String line;  
6  while ((line = br.readLine()) != null) {  
7      array.add(line);  
8  }  
9  
10 br.close();  
11  
12 for (String s : array) {  
13     System.out.println(s);  
14 }
```

## 案例：点名器

需求：我有一个文件里面存满了班级同学的姓名，每一个姓名占一行，要求通过程序实现随机点名器。

```
1  BufferedReader br = new BufferedReader(new  
   FileReader("myCharStream\\names.txt"));  
2  
3  ArrayList<String> array = new ArrayList<String>();  
4  String line;  
5  while ((line = br.readLine()) != null) {  
6      array.add(line);  
7  }  
8  
9  br.close();  
10  
11 Random r = new Random();
```

```

12     int index = r.nextInt(array.size());
13
14     String name = array.get(index);
15
16     System.out.println("幸运者是: " + name);

```

## 案例：集合到文件（改进版）

需求：把ArrayList集合中的学生数据写入到文本文件。

要求：每一个学生对象的数据作为文件中的一行数据。

```

1     ArrayList<Student> array = new ArrayList<Student>();
2
3     Student s1 = new Student("itheima001", "林青霞", 30, "西安");
4     Student s2 = new Student("itheima002", "张曼玉", 35, "武汉");
5     Student s3 = new Student("itheima003", "王祖贤", 33, "郑州");
6
7     array.add(s1);
8     array.add(s2);
9     array.add(s3);
10
11    BufferedWriter bw = new BufferedWriter(new
12        FileWriter("myCharStream\\student"));
13
14    for (Student s : array) {
15        StringBuilder sb = new StringBuilder();
16
17        sb.append(s.getSid).append(",").append(s.getName()).append(",").
18        append(s.getAge()).append(",").append(s.getAddress);
19
20        bw.write(sb.toString());
21        bw.newLine();
22        bw.flush();
23    }
24
25    bw.close();

```

## 案例：文件到集合（改进版）

需求：把文本文件中的数据读取到集合中，并遍历集合。

要求：文件中每一行数据就是一个学生对象的成员变量值。

```

1  BufferedReader br = new BufferedReader(new
    FileReader("myCharStream\\students.txt"));
2  ArrayList<Student> array = new ArrayList<Student>();
3
4  String line;
5  while ((line = br.readLine()) != null) {
6      String[] strArray = line.split(",");
7      Student s = new Student();
8
9      s.setSid(strArray[0]);
10     s.setName(strArray[1]);
11     s.setAge(Integer.parseInt(strArray[2]));
12     s.setAddress(strArray[3]);
13
14     array.add(s);
15 }
16
17 br.close();
18
19 for (Student s : array) {
20     System.out.println(s.getSid() + "." + s.getName() + "." +
        s.getAge() + "," + s.getAddress());
21 }

```

案例：集合到文件（数据排序改进版）

需求：键盘录入5个学生信息（姓名、语文成绩、数学成绩、英语成绩）。

要求：按照成绩总分从高到低写入文本文件。

```

1  TreeSet<Student> ts = new TreeSet<Student>(new
    Comparator<Student>() {
2      @Override
3      public int compare(Student s1, Student s2) {
4          int num = s2.getSum() - s1.getSum();
5          int num2 = num == 0 ? s1.getChinese() - s2.getChinese() :
            num;
6          int num3 = num2 == 0 ? s1.getMath() - s2.getMath() :
            num2;
7          int num4 = num3 == 0 ?
            s1.getName().compareTo(s2.getName()) : num3;
8          return num4;
9      }
10 });
11
12 for (int i = 0; i < 5; i++) {

```

```

13     Scanner sc = new Scanner(System.in);
14     System.out.println("请输入第" + (i + 1) + "个学生的信息: ");
15     System.out.println("姓名: ");
16     String name = sc.nextLine();
17     System.out.println("语文成绩: ");
18     int chinese = sc.nextInt();
19     System.out.println("数学成绩: ");
20     int math = sc.nextInt();
21     System.out.println("英语成绩: ");
22     int english = sc.nextInt();
23
24     Student s = new Student();
25     s.setName(name);
26     s.setChinese(chinese);
27     s.setMath(math);
28     s.setEnglish(english);
29
30     ts.add(s);
31 }
32
33 BufferedWriter bw = new BufferedWriter(new
    FileWriter("myCharStream\\ts.txt"));
34
35 for (Student s : ts) {
36     StringBuilder sb = new StringBuilder();
37
38     sb.append(s.getName()).append(", ").append(s.getChinese()).append
39     (" ").append(s.getMath()).append(", ").append(s.getEnglish()).appe
40     nd(", ").append(s.getSum());
41
42     bw.write(sb.toString());
43     bw.newLine();
44     bw.flush();
45 }
46 bw.close();

```

## 案例：复制单级文件夹

需求：把“E:\itcast”这个文件夹复制到模块目录下。

```

1 public static void main(String[] args) throws IOException {
2     File srcFolder = new File("E:\\itcast");
3

```

```

4      String srcFolderName = srcFolder.getName();
5
6      File destFolder = new File("myCharStream", srcFolderName);
7
8      if (!destFolder.exists()) {
9          destFolder.mkdir();
10     }
11
12     File[] listFiles = srcFolder.listFiles();
13
14     for (File srcfile : listFiles) {
15         String srcfileName = srcfile.getName();
16         File destFile = new File(destFolder, srcfileName);
17         copyFile(srcfile, destFile);
18     }
19 }
20
21 public static void copyFile(File srcFile, File destFile) throws
IOException {
22     BufferedInputStream bis = new BufferedInputStream(new
FileInputStream(srcFile));
23     BufferedOutputStream bos = new BufferedOutputStream(new
FileOutputStream(destFile));
24
25     byte[] bys = new byte[1024];
26     int len;
27     while ((len = bis.read()) != -1) {
28         bos.write(bys, 0, len);
29     }
30
31     bos.close();
32     bis.close();
33 }

```

## 案例：复制多级文件夹

需求：把“E:\itcast”复制到F盘目录下。

```

1  public static void main(String[] args) throws IOException {
2      File srcFile = new File("E:\\itcast");
3      File destFile = new File("F:\\");
4
5      copyFolder(srcFile, destFile);
6  }

```



```

7
8 private static void copyFolder(File srcFile, File destFile)
  throws IOException {
9     if (srcFile.isDirectory()) {
10         String srcFileName = srcFile.getName();
11         File newFolder = new File(destFile, srcFileName);
12         if (!newFolder.exists()) {
13             newFolder.mkdir();
14         }
15
16         File[] listFiles = srcFile.listFiles();
17         for (File file : listFiles) {
18             copyFolder(file, newFolder);
19         }
20     } else {
21         File newFile = new File(destFile, srcFile.getName());
22         copyFile(srcFile, newFile);
23     }
24 }
25
26 public static void copyFile(File srcFile, File destFile) throws
  IOException {
27     BufferedInputStream bis = new BufferedInputStream(new
  FileInputStream(srcFile));
28     BufferedOutputStream bos = new BufferedOutputStream(new
  FileOutputStream(destFile));
29
30     byte[] bys = new byte[1024];
31     int len;
32     while ((len = bis.read()) != -1) {
33         bos.write(bys, 0, len);
34     }
35
36     bos.close();
37     bis.close();
38 }

```

## 复制文件的异常处理

**i** try...catch...finally 的做法

```

1  try {
2      可能出现异常的代码;
3  } catch (异常类名 变量名) {
4      异常的处理代码;
5  } finally {
6      执行所有清除操作;
7  }

```

```

1  public static void copyFile(File srcFile, File destFile) {
2      BufferedInputStream bis = null;
3      BufferedOutputStream bos = null;
4
5      try {
6          bis = new BufferedInputStream(new
FileInputStream(srcFile));
7          bos = new BufferedOutputStream(new
FileOutputStream(destFile));
8
9          byte[] bys = new byte[1024];
10         int len;
11         while ((len = bis.read()) != -1) {
12             bos.write(bys, 0, len);
13         }
14     } catch (IOException e) {
15         e.printStackTrace();
16     } finally {
17         if (bos != null) {
18             try {
19                 bos.close();
20             } catch (IOException e) {
21                 e.printStackTrace();
22             }
23         }
24
25         if (bis != null) {
26             try {
27                 bis.close();
28             } catch (IOException e) {
29                 e.printStackTrace();
30             }
31         }
32     }
33 }

```

```
1 try (定义流对象) {  
2     可能出现异常的代码;  
3 } catch (异常类名 变量名) {  
4     异常的处理代码;  
5 }
```

#### 自动释放资源

```
1 public static void copyFile(File srcFile, File destFile) {  
2     try (BufferedInputStream bis = new BufferedInputStream(new  
3         FileInputStream(srcFile));  
4         BufferedOutputStream bos = new BufferedOutputStream(new  
5             FileOutputStream(destFile));) {  
6         byte[] bys = new byte[1024];  
7         int len;  
8         while ((len = bis.read()) != -1) {  
9             bos.write(bys, 0, len);  
10        }  
11    } catch (IOException e) {  
12        e.printStackTrace();  
13    }
```

#### JDK9改进方案

```
1 定义输入流对象;  
2 定义输出流对象;  
3 try (输入流对象; 输出流对象) {  
4     可能出现异常的代码;  
5 } catch (异常类名 变量名) {  
6     异常的处理代码;  
7 }
```

#### 自动释放资源

```

1 public static void copyFile(File srcFile, File destFile) throws
  IOException {
2     BufferedInputStream bis = new BufferedInputStream(new
  FileInputStream(srcFile));
3     BufferedOutputStream bos = new BufferedOutputStream(new
  FileOutputStream(destFile));
4     try (bis; bos) {
5         byte[] bys = new byte[1024];
6         int len;
7         while ((len = bis.read()) != -1) {
8             bos.write(bys, 0, len);
9         }
10    } catch (IOException e) {
11        e.printStackTrace();
12    }
13 }

```

## ✳ 特殊操作流

### 标准输入输出流

System类中有两个静态的成员变量：

- `public static final InputStream in`：标准输入流。通常该流对应于键盘输入或由主机环境或用户指定的另一个输入源
- `public static final PrintStream out`：标准输出流。通常该流对应于显示输出或由主机环境或用户指定的另一个输出目标

#### 标准输入流

**i** 自己实现键盘录入数据

- `BufferedReader br = new BufferedReader(new InputStreamReader(System.in));`

**i** 写起来太麻烦了，java就提供了一个类实现键盘录入

- `Scanner sc = new Scanner(System.in);`

## 标准输出流

输出语句的本质：是一个标准的输出流

- `PrintStream ps = System.out,`
- `PrintStream` 类有的方法，`System.out` 都可以使用

## 字节打印流

### i 打印流分类

- 字节打印流：PrintStream
- 字符打印流：PrintWriter

### i 打印流的特点

- 只负责输出数据，不负责读取数据
- 有自己的特有方法

### i 字节打印流

- `PrintStream(String fileName)`：使用指定的文件名创建新的打印流
- 使用继承父类的方法写数据，查看的时候会转码；使用自己的特有方法写数据，查看的数据原样输出

```
1 PrintString ps = new PrintString("myOtherStream\\ps.txt");
2
3 ps.print(97);
4 ps.println();
5 ps.print(98);
6
7 ps.println(97);
8 ps.println(98);
```

## 字符打印流

## 构造方法

方法名	描述
<code>PrintWriter(String fileName)</code>	使用指定的文件名创建一个新的 <code>PrintWriter</code> ，而不需要自动执行行刷新
<code>PrintWriter(Writer out, boolean autoFlush)</code>	创建一个新的 <code>PrintWriter</code> out: 字符输出流 autoFlush: 一个布尔值，如果为真，则 <code>println</code> ， <code>printf</code> 或 <code>format</code> 方法将刷新输出缓冲区

```
1  PrintWriter pw = new PrintWriter("myOtherStream\\pw.txt");
2
3  pw.write("hello");
4  pw.write("\r\n");
5  pw.flush();
6  pw.write("world");
7  pw.write("\r\n");
8  pw.flush();
9  pw.close();
```

```
1  PrintWriter pw = new PrintWriter(new
   FileWriter("myOtherStream\\pw.txt"), true);
2
3  pw.println("hello");
4  pw.println("world");
5  pw.close();
```

## 复制**Java**文件打印流改进版

需求：把模块目录下的“PrintStreamDemo.java”复制到模块目录下的“Copy.java”

```

1  BufferedReader br = new BufferedReader(new
    FileReader("myOtherStream\\PrintStreamDemo.java"));
2  PrintWriter pw = new PrintWriter(new
    FileWriter("myOtherStream\\Copy.java"), true);
3
4  String line;
5  while ((line = br.readLine()) != null) {
6      pw.println(line);
7  }
8
9  pw.close();
10 br.close();

```

## 对象序列化流

- 对象序列化：就是将对象保存到磁盘中，或者在网络中传输对象
- 这种机制就是使用一个字节序列表示一个对象，该字节序列包含：对象的类型、对象的数据和对象中存储的属性等信息
- 字节序列写到文件之后，相当于文件中持久保存了一个对象的信息
- 反之，该字节序列还可以从文件中读取回来，重构对象，对它进行反序列化

**i** 要实现序列化和反序列化就要使用对象序列化流和对象反序列化流

- 对象序列化流： `ObjectOutputStream`
- 对象反序列化流： `ObjectInputStream`

### 对象序列化流

对象序列化流： `ObjectOutputStream`

- 将 Java 对象的原始数据类型和图形写入 `OutputStream`。可以使用 `ObjectInputStream` 读取（重构）对象。可以通过使用流的文件来实现对象的持久存储。如果流是网络套接字流，则可以在另一个主机上或另一个进程中重构对象。

**i** 构造方法

- `ObjectOutputStream(OutputStream out)` : 创建一个写入指定的 `ObjectStream` 的 `ObjectOutputStream`

#### **i** 序列化对象的方法

- `void writeString(Object obj)` : 将制定的对象写入 `ObjectOutputStream`

#### **i** 注意

- 一个对象要想被序列化, 该对象所属的类必须实现 `Serializable` 接口
- `Serializable` 是一个 标记接口, 实现该接口, 不需要重写任何方法

```
1 ObjectOutputStream oos = new ObjectOutputStream(new
2   FileOutputStream("myOthersStream\\oos.txt"));
3 Student s = new Student("林青霞", 30);
4
5 oos.writeObject(s);
6
7 oos.close();
```

## 对象反序列化流

对象反序列化流: `ObjectInputStream`

- `ObjectInputStream` 反序列化先前使用 `ObjectOutputStream` 编写的原始数据和对象

#### **i** 构造方法

- `ObjectInputStream(InputStream in)` : 创建从指定的 `InputStream` 读取的 `ObjectInputStream`

#### **i** 反序列化对象的方法

- `Object readObject()` : 从 `ObjectInputStream` 读取一个对象



```
1 ObjectOutputStream ois = new ObjectOutputStream(new
  FileInputStream("myOthersStream\\oos.txt"));
2
3 Object obj = new ois.readObject();
4
5 Student s = (Student) obj;
6 System.out.println(s.getName() + "," + s.getAge());
7
8 ois.close();
```

## 对象序列化流问题



用对象序列化流序列化了一个对象后，假如修改了对象所属的类文件，读取数据会不会出问题？

- 会出问题，抛出 `InvalidClassException` 异常



如果出问题了，如何解决？

- 给对象所属的类加一个 `serialVersionUID`
  - `private static final serialVersionUID = 421;`



如果一个对象中的某个成员变量的值不想被序列化，又该如何实现？

- 给该成员变量加一个 `transient` 关键字，该关键字标记的成员变量不参与序列化过程

## Properties



Properties概述

- 是一个Map体系的集合类
- Properties可以保存到流中或从流中加载



练习：Properties作为Map集合的使用

```

1 Properties prop = new Properties();
2
3 // 存储元素
4 prop.put("itheima001", "林青霞");
5 prop.put("itheima002", "张曼玉");
6 prop.put("itheima003", "王祖贤");
7
8 // 遍历集合
9 Set<Object> keySet = prop.keySet();
10 for (Object key : keySet) {
11     Object value = prop.get(key);
12     System.out.println(key + "," + value);
13 }

```

### Properties作为集合的特有方法

方法名	描述
Object setProperty(String key, String value)	设置集合的键和值，都是String类型，底层调用 HashTable方法 put
String getProperty(String key)	使用此属性列表中指定的键搜索属性
Set stringPropertyNames()	从该属性列表中返回一个不可修改的键集，其中 键及其对应的值是字符串

```

1 Properties prop = new Properties();
2
3 // 存储元素
4 prop.setProperty("itheima001", "林青霞");
5 prop.setProperty("itheima002", "张曼玉");
6 prop.setProperty("itheima003", "王祖贤");
7
8 System.out.println(prop.getProperty("itheima001"));
9
10 Set<String> names = prop.stringPropertyNames();
11 for (String key : names) {
12     String value = prop.getProperty(key);
13     System.out.println(key + "," + value);
14 }

```

### Properties和IO流结合的方法

方法名	描述
<code>void load(Stream inStream)</code>	从输入字节流读取数组列表（键和元素对）
<code>void load(Reader reader)</code>	从输入字符流读取属性列表（键和元素对）
<code>void store(OutputStream out, String comments)</code>	将此属性列表（键和元素对）写入此Properties表中，以适合于使用load(InputStream)方法的格式写入输出字节流
<code>void store(Writer writer, String comments)</code>	将此属性列表（键和元素对）写入此Properties表中，以适合使用load(Reader)方法的格式写入输出字符流

## 案例：游戏次数

需求：请写程序实现猜数字小游戏只能试玩3次，如果还想玩，提示：游戏试玩已结束，想玩请充值。

```

1 Properties prop = new Properties();
2
3 FileReader fr = new FileReader("myOtherStream\\game.txt");
4 prop.load(fr);
5 fr.close();
6
7 String count = prop.getProperty("count");
8 int number = Integer.parseInt(count);
9
10 if (number >= 3) {
11     System.out.println("游戏试玩已结束，想玩请充值");
12 } else {
13     GuessNumber.start();
14
15     number++;
16     prop.setProperty("count", String.valueOf(number));
17     FileWriter fw = new FileWriter("myOtherStream\\game.txt");
18     prop.store(fw, null);
19     fw.close();
20 }

```

# 多线程

## ✧ 实现多线程

---

### 进程

进程：是正在运行的程序

- 是系统进行资源分配和调用的独立单位
- 每一个进程都有它自己的内存空间和系统资源

### 线程

线程：是进程中的单个顺序控制流，是一条执行路径

- 单线程：一个进程如果只有一条执行路径，则称为单线程程序
- 多线程：一个进程如果有多条执行路径，则成为多线程程序

### 多线程的实现方式1

#### 继承Thread类

- 定义一个MyThread类继承Thread类
- 在MyThread类中重写run()方法
- 创建MyThread类的对象
- 启动线程

#### 两个小问题

- 为什么要重写run()方法？
  - 因为run()是用来封装被线程执行的代码
- run()方法和start()方法的区别？

- `run()`: 封装线程执行的代码，直接调用，相当于普通方法的调用
- `start()`: 启动线程，然后由JVM调用此线程的`run()`方法

```
1 public class MyThread extends Thread {
2     @Override
3     public void run() {
4         for (int i = 0; i < 100; i++) {
5             System.out.println(i);
6         }
7     }
8 }
```

```
1 public class MyThreadDemo {
2     public static void main(String[] args) {
3         MyThread my1 = new MyThread();
4         MyThread my2 = new MyThread();
5
6         // 启动线程
7         my1.start();
8         my2.start();
9     }
10 }
```

## 设置和获取线程名称

**i** Thread类中设置和获取线程名称的方法

- `void setName(String name)` : 将此线程的名称更改为等于参数`name`
- `String getName()` : 返回此线程的名称

## 线程调度

**i** 线程有两种调度模型

- 分时调度模型: 所有线程轮流使用CPU的使用权，平均分配每个线程占用CPU的时间片
- 抢占式调度模型: 优先让优先级高的线程使用CPU，如果线程的优先级相同，`name`会随机选择一个，优先级高的线程获取的CPU时间片相对多一些

**i** Java使用的是抢占式调度模型。

- 加入计算机只有一个CPU，那么CPU在某一时刻只能执行一条指令，线程只有得到CPU的时间片，也就是使用权，才可以执行指令。所以说多线程程序的执行有 **随机性**，因为谁抢到CPU的使用权是不一定的。

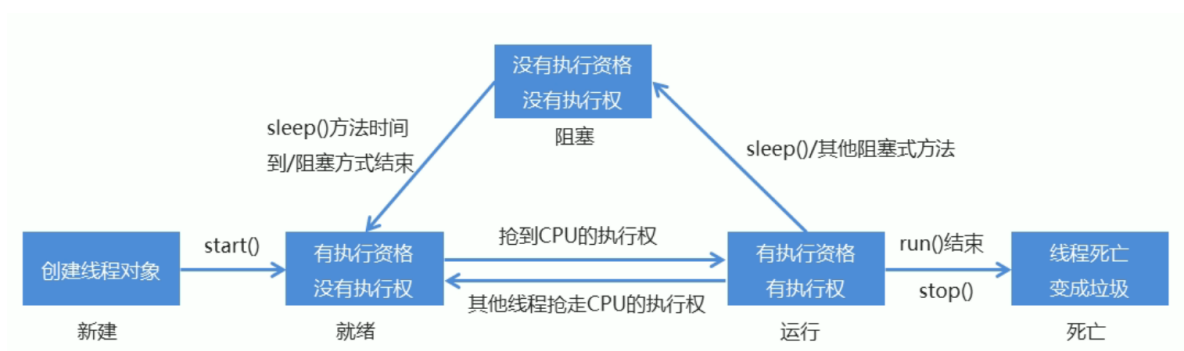
**i** Thread类中设置和获取线程优先级的方法

- `public final int getPriority()`：返回此线程的优先级
- `public final void setPriority(int newPriority)`：更改此线程的优先级
  - 线程默认优先级是 **5**，线程优先级的范围是 **1-10**
  - 线程优先级高仅仅表示获取CPU时间片的几率高，但是要在次数比较多或者多次运行的时候才能看到你想要的结果

## 线程控制

方法名	说明
<code>static void sleep(long millis)</code>	使当前正在执行的线程停留（暂停执行）指定的毫秒数
<code>void join()</code>	等待这个线程死亡
<code>void setDaemon(boolean on)</code>	将此线程标记为守护线程，当运行的线程都是守护线程时，Java虚拟机将退出

## 线程生命周期



## 多线程的实现方式2

### i 实现Runnable接口

- 定义一个MyRunnable实现Runnable接口
- 在MyRunnable类中重写run()方法
- 创建MyRunnable类的对象
- 创建Thread类的对象，把MyRunnable对象作为构造方法的参数
  - `Thread(Runnable target, String name)`
- 启动线程

### i 多线程的实现方案有2中

- 继承Thread类
- 实现Runnable接口

### i 相比继承Thread类，实现Runnable接口的好处

- 避免了Java单继承的局限性
- 适合多个相同程序的代码去处理同一个资源的情况，把线程和程序的代码、数据有效分离，较好地体现了面向对象的设计思想

```
1 MyRunnable my = new MyRunnable();
2
3 Thread t1 = new Thread(my, "高铁");
4 Thread t2 = new Thread(my, "飞机");
5
6 // 启动线程
7 t1.start();
8 t2.start();
```

## ✧ 线程同步

### 案例：卖票

需求：某电影院正在上映国产大片，共有100张票，而它有3个窗口售卖，请设计一个程序实现模拟该电影院卖票。

```
1 public class SellTicket implements Runnable {
2     private int tickets = 100;
3
4     @Override
5     public void run() {
6         while (true) {
7             if (tickets > 0) {
8
9                 System.out.println(Thread.currentThread().getName() + "正在出售第"
10                + tickets + "张票");
11                 tickets--;
12             }
13         }
14     }
15 }
```

```
1 public class SellTicketDemo {
2     public static void main(String[] args) {
3         SellTicket st = new SellTicket();
4
5         Thread t1 = new Thread(st, "窗口一");
6         Thread t2 = new Thread(st, "窗口二");
7         Thread t3 = new Thread(st, "窗口三");
8
9         // 启动线程
10        t1.start();
11        t2.start();
12        t3.start();
13    }
14 }
```

## 卖票案例的思考

要求：每次出票时间100毫秒，用sleep()方法实现。

```
1 public class SellTicket implements Runnable {
2     private int tickets = 100;
3     private Object obj = new Object();
4
5     @Override
6     public void run() {
```



```

7         while (true) {
8             synchronized (obj) {
9                 if (tickets > 0) {
10                     try {
11                         Thread.sleep(100);
12                     } catch (InterruptedException e) {
13                         throw new RuntimeException(e);
14                     }
15
16                     System.out.println(Thread.currentThread().getName() + "正在出售第"
17                     + tickets + "张票");
18                     tickets--;
19                 }
20             }
21 }

```

### 为什么会出现安全问题？

- 是否是多线程环境
- 是否有数据共享
- 是否有多条语句操作共享数据

### 如何解决多线程安全问题？

- 基本思想：让程序没有安全问题的环境

### 如何实现？

- 把多条语句操作共享数据的代码给锁起来，让任意时刻只能有一个线程执行即可
- Java提供了同步代码块的方式来解决

## 同步代码块

锁多条语句操作共享数据，可以使用同步代码块实现

- 格式

```
1 synchronized(任意对象) {  
2     多条语句操作共享数据的代码;  
3 }
```

- `synchronized(任意对象)`: 就相当于给代码加锁了, 任意对象就可以看成是一把锁

### 同步的好处和弊端

- 好处: 解决了多线程的数据安全问题
- 弊端: 当线程很多时, 因为每个线程都会去判断同步上的锁, 这是很耗费资源的, 无形中会降低程序的运行效率

## 同步方法

- 同步方法: 把 `synchronized` 关键字加到方法上
- 格式:

```
1 修饰符 synchronized 返回值类型 方法名(方法参数) {}
```

### 同步方法的锁对象是什么?

- `this`
- 同步静态方法: 把 `synchronized` 关键字加到静态方法上
- 格式:

```
1 修饰符 static synchronized 返回值类型 方法名(方法参数) {}
```

### 同步静态方法的锁对象是什么?

- `类名.this`

## 线程安全的类

### **StringBuffer**

- 线程安全, 可变的字符序列

- 从版本JDK5开始，被StringBuilder替代。通常应该使用StringBuilder类，因为它支持所有相同的操作，但它更快，因为它不执行同步

## Vector

- 从Java平台v1.2开始，该类改进了List接口，使其成为Java Collections Framework的成员。与新的集合实现不同，Vector被同步，如果不需要线程安全的实现，建议使用ArrayList代替Vector

## HashTable

- 该类实现了一个哈希表，它将键映射到值。任何非null对象都可以用作键或者值
- 从Java平台v1.2开始，该类进行了改进，实现了Map接口，使其成为Java Collections Framework的成员。与新的集合实现不同，HashTable被同步。如果不需要线程安全的实现，建议使用HashMap代替Hashtable

## Lock锁

Lock实现提供比使用synchronized方法和语句可以获得更广泛的锁定操作

**i** Lock中提供了获得锁和释放锁的方法

- `void lock()` : 获得锁
- `void unlock()` : 释放锁

Lock是接口不能直接实例化，这里采用它的实现类ReentrantLock来实例化

**i** ReentrantLock的构造方法

- `ReentrantLock()` : 创建一个ReentrantLock的实例

```
1 public class SellTicket implements Runnable {
2     private int tickets = 100;
3     private Lock lock = new ReentrantLock();
4
5     @Override
6     public void run() {
7         while (true) {
```

```

8         try {
9             lock.lock();
10            if (tickets > 0) {
11                try {
12                    Thread.sleep(100);
13                } catch (InterruptedException e) {
14                    throw new RuntimeException(e);
15                }
16
17                System.out.println(Thread.currentThread().getName() + "正在出售第"
18                + tickets + "张票");
19                tickets--;
20            }
21        } finally {
22            lock.unlock();
23        }
24    }

```

## ✧ 生产者消费者

### 生产者消费者模式概述

生产者消费者模式是一个十分经典的多线程协作的模式，弄懂生产者消费者问题能够让我们对多线程编程的理解更加深刻。

所谓的生产者消费者问题，实际上主要是包含了两类线程：

- 一类是生产者线程用余生产数据
- 一类是消费者线程用于消费数据

为了解偶生产者和消费者的关系，通常会采用共享的数据区域，就像是一个仓库

- 生产者生产数据之后直接放置在公共数据区中，并不需要关心消费者的行为
- 消费者只需要从共享数据区中去取数据，并不需要关心生产者的行为

**i** Object类中的等待和唤醒方法：

方法名	描述
void wait()	导致当前线程等待，直到另一个线程调用该对象的notify()方法或notifyAll()方法
void notify()	唤醒正在等待对象监视器的单个线程
void notifyAll()	唤醒正在等待对象监视器的所有线程

## 生产者消费者案例

```

1  public class Box {
2      private int milk;
3      private boolean state = false;
4
5      public synchronized void put(int milk) {
6          // 如果有牛奶，等待消费
7          if (state) {
8              try {
9                  wait();
10             } catch (InterruptedException e) {
11                 throw new RuntimeException(e);
12             }
13         }
14
15         // 如果没有牛奶，就生产牛奶
16         this.milk = milk;
17         System.out.println("送奶工将第" + this.milk + "瓶奶放入奶
18         箱");
19
20         // 修改完毕之后，修改奶箱状态
21         state = true;
22
23         // 唤醒其他等待的线程
24         notifyAll();
25     }
26
27     public synchronized void get() {
28         // 如果没有牛奶，应该等待生产
29         if (!state) {
30             try {

```

```

30         wait();
31     } catch (InterruptedException e) {
32         throw new RuntimeException(e);
33     }
34 }
35
36 // 如果有牛奶，就消费牛奶
37 System.out.println("用户拿到第" + this.milk + "瓶奶");
38
39 // 消费完毕之后，修改奶箱状态
40 state = false;
41
42 // 唤醒其他等待的线程
43 notifyAll();
44 }
45 }

```

```

1  public class Producer implements Runnable {
2      private Box b;
3
4      public Producer(Box b) {
5          this.b = b;
6      }
7
8      @Override
9      public void run() {
10         for (int i = 1; i ≤ 5; i++) {
11             b.put(i);
12         }
13     }
14 }

```

```

1  public class Customer implements Runnable {
2      private Box b;
3
4      public Customer(Box b) {
5          this.b = b;
6      }
7
8      @Override
9      public void run() {
10         while (true) {
11             b.get();
12         }
13     }

```

```
14 }

```

```
1 public class BoxDemo {
2     public static void main(String[] args) {
3         Box b= new Box();
4
5         Producer p =new Producer(b);
6
7         Customer c = new Customer(b);
8
9         Thread t1 = new Thread(p);
10        Thread t2 = new Thread(c);
11
12        t1.start();
13        t2.start();
14    }
15 }
```

# 网络编程

## ✧ 网络编程入门

### 网络编程概述

#### 计算机网络

- 是指将地理位置不同的具有独立功能的多台计算机及其外部设备，通过通信线路连接起来，在网络操作系统，网络管理软件及网络通信协议的管理和协调下，实现资源共享和信息传递的计算机系统

#### 网络编程

- 在网络通信协议下，实现网络互联的不同计算机上运行的程序间可以进行数据交换

## 网络编程三要素

### ○ IP地址

- 要想让网络中的计算机能够互相通信，必须为每台计算机指定一个标识号，通过这个标识号来指定要接收数据的计算机和识别发送的计算机，而IP地址就是这个标识号。也就是设备的标识

### ○ 端口

- 网络的通信，本质上是两个应用程序的通信。每台计算机都有很多的应用程序，那么在网络通信时，如何区分这些应用程序呢？如果说IP地址可以唯一标识网络中的设备，那么端口号就可以唯一标识设备中的应用程序了。也就是应用程序的标识

### ○ 协议

- 通过计算机网络，可以使多台计算机实现连接，位于同一个网络中的计算机在进行连接和通信时需要遵守一定的规则。在计算机网络中，这些连接和通信的规则被称为网络通信协议，它对数据的传输格式、传输速率、传输步骤等做了统一的规定，通信双方必须同时遵守才能完成数据交换。常见的协议有UDP协议和TCP协议

## IP地址

IP地址：是网络中设备的唯一标识

### IP地址分为两大类

- **IPV4**：是给每个连接在网络上的主机分配一个32bit地址。按照TCP/IP规定，IP地址用二进制来表示，每个IP地址长32bit，也就是4个字节。例如一个采用二进制形式的IP地址是“11000000 10101000 00000001 01000010”，这么长的地址，处理起来太费劲了。为了方便使用，IP地址经常被写成十进制的形式，中间使用符号“.”分隔不同的字节。于是，上面的IP地址可以表示为“192.168.1.66”。IP地址的这种表示法叫做“点分十进制表示法”，这显然比1和0容易记忆很多
- **IPV6**：由于互联网的蓬勃发展，Ip地址的需求量愈来愈大，但是网络地址资源有限，使得IP的分配越发紧张。为了扩大地址空间，通过IPV6重新定义地址空间，采用128位地址长度，每16个字节一组，分成8组十六进制数，这样就解决了网络地址资源数量不够的问题



## 常用命令

- `ipconfig` : 查看本机IP地址
- `ping IP地址` : 检查网络是否连通

## 特殊IP地址

- 127.0.0.1: 是回送地址，可以代表本机地址，一般用来测试使用

## InetAddress的使用

**InetAddress**: 此类表示Internet协议（IP）地址

方法名	描述
<code>static InetAddress getByName(String host)</code>	确定主机名称的IP地址。主机名称可以使机器名称，也可以是IP地址
<code>String getHostName()</code>	获取此IP地址的主机名
<code>String.getHostAddress()</code>	返回文本显示中的IP地址字符串

```
1 // InetAddress address = InetAddress.getByName("DESKTOP-VJICL7M");
2 InetAddress address = InetAddress.getByName("192.168.1.5");
3
4 String name = address.getHostName();
5 String ip = address.getHostAddress();
6
7 System.out.println("主机名" + name);
8 System.out.println("IP地址" + ip);
```

## 端口

**端口**: 设备上应用程序的唯一标识

**端口号**: 用两个字节表示的整数，它的取值范围是0~65535.其中，0~1023之间的端口号用于一些知名的网络服务和应用，普通的应用程序需要使用1024以上的端口号。如果端口号被另一个服务或应用所占用，会导致当前程序启动失败。

# 协议

协议：计算机网络中，连接和通信的规则被称为网络通信协议。

## i UDP协议

- 用户数据服务协议（User Datagram Protocol）
- UDP是无连接通信协议。即在传输数据时，数据的发送端和接收端不建立逻辑连接。简单来说，当一台计算机向另一台计算机发送数据时，发送端不会确认接收端是否存在，就会发出数据，同样接收端在收到数据时，也不会向发送端反馈是否收到数据。
- 由于UDP协议消耗资源小，通信效率高，所以通常都会用于音频、视频和普通数据的传输
- 例如视频会议通常采用UDP协议，因为这种情况即使偶尔丢失一两个数据包，也不会对接收结果产生太大影响。但是使用UDP协议传送数据时，由于UDP的面向无连接性，不能保证数据的完整性，因此在传输重要数据时不建议使用UDP协议

## i TCP协议

- 传输控制协议（Transmission Control Protocol）
- TCP协议是 **面向连接** 的通信协议，即传输数据之前，在发送端和接收端建立逻辑连接，然后再传输数据，它提供了两台计算机之间 **可靠无差错** 的数据传输。在TCP连接中必须要明确客户端与服务器端，由客户端向服务端发出连接请求，每次连接的创建都需要经过“三次握手”
- “三次握手”：TCP协议中，在发送数据的准备阶段，客户端与服务器之间的三次交互，以保证连接的可靠
  - 第一次握手，客户端向服务器端发出连接请求，等待服务器确认
  - 第二次握手，服务器端向客户端会送一个响应，通知客户端收到了连接请求
  - 第三次握手，客户端再次向服务器端发送确认信息，确认连接
- 完成三次握手，连接建立后，客户端和服务端就可以开始进行数据传输了。由于这种面向连接的特性，TCP洗衣可以保证传输数据的安全，所以应用十分广泛。例如上传文件、下载文件、浏览网页等

## UDP发送数据

### 发送数据的步骤

- 1 创建发送端的Socket对象（DatagramSocket）
- 2 创建数据，并把数据打包
- 3 调用DatagramSocket对象的方法发送数据
- 4 关闭发送端

```
1  DatagramSocket ds = new DatagramSocket();
2
3  byte[] bys = "hello,udp,我来了".getBytes();
4  /*int length = bys.length;
5   InetAddress address = InetAddress.getByName("192.168.1.66");
6   int port = 10086;
7   DatagramPacket dp = new DatagramPacket(bys, length, address,
8   port);*/
9  DatagramPacket dp = new DatagramPacket(bys, bys.length,
10  InetAddress.getByName("192.168.1.66"), 10086);
11
12 ds.send(dp);
13
14 ds.close();
```

## UDP接收数据

### 接收数据的步骤

- 1 创建接收端的Socket对象（DatagramSocket）
- 2 创建一个数据包，用于接收数据
- 3 调用DatagramSocket对象的方法接收数据
- 4 解析数据包，并把数据在控制台显示
- 5 关闭接收端

```

1 DatagramSocket ds = new DatagramSocket(10086);
2
3 byte[] bys = new byte[1024];
4 DatagramPacket dp = new DatagramPacket(bys, bys.length);
5
6 ds.receive(dp);
7
8 byte[] datas = dp.getData();
9 int len = dp.getLength();
10 String dataString = new String(datas, 0, len);
11 System.out.println("数据是: " + dataString);
12
13 ds.close();

```

## UDP通信程序练习

按照下面的要求实现程序：

- UDP发送数据：数据来自于键盘录入，直到输入的数据是886，发送数据结束
- UDP接收数据：因为接收端不知道发送端什么时候停止发送，故采用死循环接收

```

1 import java.io.BufferedReader;
2 import java.io.IOException;
3 import java.io.InputStreamReader;
4 import java.net.DatagramPacket;
5 import java.net.DatagramSocket;
6 import java.net.InetAddress;
7
8 public class SendDemo {
9     public static void main(String[] args) throws IOException {
10         DatagramSocket ds = new DatagramSocket();
11
12         BufferedReader br = new BufferedReader(new
13 InputStreamReader(System.in));
14         String line;
15         while ((line = br.readLine()) != null) {
16             if ("886".equals(line)) {
17                 break;
18             }
19
20             byte[] bys = line.getBytes();

```

```

20         DatagramPacket dp = new DatagramPacket(bys,
21         bys.length, InetAddress.getByName("192.168.1.66"), 12345);
22
23         ds.send(dp);
24     }
25
26     ds.close();
27 }

```

```

1  import java.io.IOException;
2  import java.net.DatagramPacket;
3  import java.net.DatagramSocket;
4
5  public class ReceiveDemo {
6      public static void main(String[] args) throws IOException {
7          DatagramSocket ds = new DatagramSocket(12345);
8
9          while (true) {
10             byte[] bys = new byte[1024];
11             DatagramPacket dp = new DatagramPacket(bys,
12             bys.length);
13
14             ds.receive(dp);
15
16             System.out.println("数据是: " + new
17             String(dp.getData(), 0, dp.getLength()));
18         }
19     }
20 }

```

## ✧ TCP通信程序

---

### TCP通信原理

TCP通信协议是一种可靠的网络协议，它在通信的两端各建立一个Socket对象，从而在通信的两端形成网络虚拟链路，一旦建立了虚拟的网络链路，两端的程序就可以通过虚拟链路进行通信。

Java对基于TCP协议的网络提供了良好的封装，使用Socket对象；来代表两端的通信端口，并通过Socket产生的IO流来进行网络通信。

Java为客户端提供了Socket类，为服务器端提供了ServerSocket类。

## TCP发送数据

### 发送数据的步骤

- 1 创建客户端的Socket对象（Socket）
- 2 获取输出流，写数据
- 3 释放资源

```
1 // Socket s = new Socket(InetAddress.getByName("192.168.1.66"),  
2 // 10086);  
2 Socket s = new Socket("192.168.1.66", 10086);  
3  
4 OutputStream os = s.getOutputStream();  
5  
6 os.write("hello java,我来了".getBytes());  
7  
8 s.close();
```

## TCP接收数据

### 接收数据的步骤

- 1 创建服务器端的Socket对象（ServerSocket）
- 2 获取输入流，读数据，并把数据显示在控制台
- 3 释放资源

```

1  ServerSocket ss = new ServerSocket(10086);
2
3  Socket s = ss.accept();
4  InputStream is = s.getInputStream();
5
6  byte[] bys = new byte[1024];
7  int len = is.read(bys);
8  String data = new String(bys, 0, len);
9  System.out.println("数据是: " + data);
10
11 s.close();
12 ss.close();

```

## TCP通信程序练习1

- 客户端：发送数据，接收服务器反馈
- 服务端：接收数据，给出反馈

```

1  import java.io.IOException;
2  import java.io.InputStream;
3  import java.io.OutputStream;
4  import java.net.Socket;
5
6  public class ClientDemo {
7      public static void main(String[] args) throws IOException {
8          Socket s = new Socket("192.168.1.66", 10086);
9
10         OutputStream os = s.getOutputStream();
11         os.write("hello java,我来了".getBytes());
12
13         InputStream is = s.getInputStream();
14         byte[] bys = new byte[1024];
15         int len = is.read();
16         String data = new String(bys, 0, len);
17         System.out.println("客户端: " + data);
18
19         s.close();
20     }
21 }

```

```

1  import java.io.IOException;
2  import java.io.InputStream;
3  import java.io.OutputStream;
4  import java.net.ServerSocket;

```

```

5  import java.net.Socket;
6
7  public class ServerDemo {
8      public static void main(String[] args) throws IOException {
9          ServerSocket ss = new ServerSocket(10086);
10
11         Socket s = ss.accept();
12         InputStream is = s.getInputStream();
13         byte[] bys = new byte[1024];
14         int len = is.read(bys);
15         String data = new String(bys, 0, len);
16         System.out.println("服务器: " + data);
17
18         // 给出反馈
19         OutputStream os = s.getOutputStream();
20         os.write("数据已经收到".getBytes());
21
22         ss.close();
23     }
24 }

```

## TCP通信程序练习2

- 客户端：数据来自于键盘录入，直到输入的数据是886，发送数据结束
- 服务器：接收到的数据在控制台输出

```

1  import java.io.*;
2  import java.net.Socket;
3
4  public class ClientDemo {
5      public static void main(String[] args) throws IOException {
6          Socket s = new Socket("192.168.1.66", 10086);
7
8          BufferedReader br = new BufferedReader(new
9              InputStreamReader(System.in));
10         // 封装输出流对象
11         BufferedWriter bw = new BufferedWriter(new
12             OutputStreamWriter(s.getOutputStream()));
13
14         String line;
15         while ((line = br.readLine()) != null) {
16             if ("886".equals(line)) {
17                 break;
18             }
19         }
20     }
21 }

```



```

16         }
17
18         // 获取输出流对象
19         bw.write(line);
20         bw.newLine();
21         bw.flush();
22     }
23
24     s.close();
25 }
26 }

```

```

1  import java.io.BufferedReader;
2  import java.io.IOException;
3  import java.io.InputStreamReader;
4  import java.net.ServerSocket;
5  import java.net.Socket;
6
7  public class ServerDemo {
8      public static void main(String[] args) throws IOException {
9          ServerSocket ss = new ServerSocket(10086);
10
11         // 监听客户端的连接, 返回一个对应的Socket对象
12         Socket s = ss.accept();
13
14         // 获取输入流
15         /*InputStream is = s.getInputStream();
16         InputStreamReader isr = new InputStreamReader(is);
17         BufferedReader br = new BufferedReader(isr);*/
18         BufferedReader br = new BufferedReader(new
19             InputStreamReader(s.getInputStream()));
20         String line;
21         while ((line = br.readLine()) != null) {
22             System.out.println(line);
23         }
24
25         // 释放资源
26         ss.close();
27     }
28 }

```

## TCP通信程序练习3

- 客户端：数据来自于键盘录入，直到输入的数据是886，发送数据结束

- 服务器：接收到的数据写入文本文件

```
1  import java.io.*;
2  import java.net.ServerSocket;
3  import java.net.Socket;
4
5  public class ServerDemo {
6      public static void main(String[] args) throws IOException {
7          ServerSocket ss = new ServerSocket(10086);
8
9          // 监听客户端的连接, 返回一个对应的Socket对象
10         Socket s = ss.accept();
11
12         // 获取输入流
13         BufferedReader br = new BufferedReader(new
InputStreamReader(s.getInputStream()));
14         // 把数据写入文本文件
15         BufferedWriter bw = new BufferedWriter(new
FileWriter("myNet\\s.txt"));
16         String line;
17         while ((line = br.readLine()) != null) {
18             bw.write(line);
19             bw.newLine();
20             bw.flush();
21         }
22
23         // 释放资源
24         bw.close();
25         ss.close();
26     }
27 }
```

## TCP通信程序练习4

- 客户端：数据来自于文本文件
- 服务器：接收到的数据写入文本文件

```
1  import java.io.*;
2  import java.net.Socket;
3
4  public class ClientDemo {
5      public static void main(String[] args) throws IOException {
6          Socket s = new Socket("192.168.1.66", 10086);
```

```

7
8         BufferedReader br = new BufferedReader(new
FileReader("myNet\\InetAddressDemo.txt"));
9         // 封装输出流对象
10        BufferedWriter bw = new BufferedWriter(new
OutputStreamWriter(s.getOutputStream()));
11
12        String line;
13        while ((line = br.readLine()) != null) {
14            bw.write(line);
15            bw.newLine();
16            bw.flush();
17        }
18
19        // 释放资源
20        br.close();
21        s.close();
22    }
23 }

```

## TCP通信程序练习5

- 客户端：数据来自于文本文件，接收服务器反馈
  - 服务器：接收到的数据写入文本文件，给出反馈
- 
- 出现问题：程序一直等待
  - 原因：读数据的方法是阻塞式的
  - 解决办法：自定义结束标记，或使用 `shutdownOutput()` 方法（推荐）

```

1  import java.io.*;
2  import java.net.Socket;
3
4  public class ClientDemo {
5      public static void main(String[] args) throws IOException {
6          Socket s = new Socket("192.168.1.66", 10086);
7
8          BufferedReader br = new BufferedReader(new
FileReader("myNet\\InetAddressDemo.txt"));
9          // 封装输出流对象
10         BufferedWriter bw = new BufferedWriter(new
OutputStreamWriter(s.getOutputStream()));
11

```

```

12     String line;
13     while ((line = br.readLine()) != null) {
14         bw.write(line);
15         bw.newLine();
16         bw.flush();
17     }
18
19     // 自定义结束标记
20     /*bw.write("886");
21     bw.newLine();
22     bw.flush();*/
23
24     s.shutdownOutput();
25
26     // 给出反馈
27     BufferedWriter bwServer = new BufferedWriter(new
OutputStreamWriter(s.getOutputStream()));
28     bwServer.write("文件上传成功");
29     bwServer.newLine();
30     bwServer.flush();
31
32     // 释放资源
33     br.close();
34     s.close();
35 }
36 }

```

```

1  import java.io.*;
2  import java.net.ServerSocket;
3  import java.net.Socket;
4
5  public class ServerDemo {
6      public static void main(String[] args) throws IOException {
7          ServerSocket ss = new ServerSocket(10086);
8
9          // 监听客户端的连接, 返回一个对应的Socket对象
10         Socket s = ss.accept();
11
12         // 获取输入流
13         BufferedReader br = new BufferedReader(new
InputStreamReader(s.getInputStream()));
14         // 把数据写入文本文件
15         BufferedWriter bw = new BufferedWriter(new
FileWriter("myNet\\s.txt"));
16         String line;

```

```

17         while ((line = br.readLine()) != null) {
18             /*if ("886".equals(line)) {
19                 break;
20             }*/
21
22             bw.write(line);
23             bw.newLine();
24             bw.flush();
25         }
26
27         // 接收反馈
28         BufferedReader brClient = new BufferedReader(new
InputStreamReader(s.getInputStream()));
29         String data = brClient.readLine();
30         System.out.println("服务器的反馈: " + data);
31
32         // 释放资源
33         bw.close();
34         ss.close();
35     }
36 }

```

## TCP通信程序练习6

- 客户端：数据来自于文本文件，接收服务器反馈
- 服务器：接收到的数据写入文本文件，给出反馈，代码用线程进行封装，为每一个客户端开启一个线程

```

1  import java.io.*;
2  import java.net.Socket;
3
4  public class serverThread implements Runnable {
5      private Socket s;
6
7      public serverThread(Socket s) {
8          this.s = s;
9      }
10
11     @Override
12     public void run() {
13         // 接收数据, 写入文本文件
14         try {

```

```

15         BufferedReader br = new BufferedReader(new
InputStreamReader(s.getInputStream()));
16
17         // 解决名称冲突问题
18         int count = 0;
19         File file = new File("myNet\\Copy[" + count +
"].java");
20
21         while (file.exists()) {
22             count++;
23             file = new File("myNet\\Copy[" + count +
"].java");
24         }
25
26         BufferedWriter bw = new BufferedWriter(new
FileWriter(file));
27
28         String line;
29         while ((line = br.readLine()) != null) {
30             bw.write(line);
31             bw.newLine();
32             bw.flush();
33         }
34
35         // 给出反馈
36         BufferedWriter bwServer = new BufferedWriter(new
OutputStreamWriter(s.getOutputStream()));
37         bwServer.write("文件上传成功");
38         bwServer.newLine();
39         bwServer.flush();
40
41         // 释放资源
42         s.close();
43     } catch (IOException e) {
44         e.printStackTrace();
45     }
46 }
47 }

```

```

1 import java.io.*;
2 import java.net.ServerSocket;
3 import java.net.Socket;
4
5 public class ServerDemo {
6     public static void main(String[] args) throws IOException {

```

```

7      ServerSocket ss = new ServerSocket(10086);
8
9      while (true) {
10         // 监听客户端的连接, 返回一个对应的Socket对象
11         Socket s = ss.accept();
12
13         // 为每一个客户端开启一个线程
14         new Thread(new serverThread(s)).start();
15     }
16 }
17 }

```

## Lambda表达式

### ✧ 体验Lambda表达式

---

需求：启动一个线程，在控制台输出一句话：多线程启动了

#### 方式1

- ① 定义一个类MyRunnable实现Runnable接口，重写run()方法
- ② 创建MyRunnable类的对象
- ③ 创建Thread类的对象，把MyRunnable类的对象作为构造参数传递
- ④ 启动线程

#### 方式2

- 匿名内部类的方式改进

#### 方式3

- Lambda表达式的方式改进

```

1  public class LambdaDemo {
2      public static void main(String[] args) {

```

```

3      /*MyRunnable my = new MyRunnable();
4      Thread t = new Thread(my);
5      t.start();*/
6
7      // 使用匿名内部类的方式改进
8      // new Thread(new Runnable() {
9      //     @Override
10     //     public void run() {
11     //         System.out.println("多线程徐启动了");
12     //     }
13     // }).start();
14
15     // Lambda表达式改进
16     new Thread( () → {
17         System.out.println("多线程徐启动了");
18     } ).start();
19 }
20 }

```

## Lambda表达式的标准格式

### 匿名内部类中重写run()方法的代码分析

```

1  new Thread(new Runnable() {
2      @Override
3      public void run() {
4          System.out.println("多线程徐启动了");
5      }
6  }).start();

```

- 方法形式参数为空，说明调用方法时不需要传递参数
- 方法返回值类型为void，说明方法执行没有结果返回
- 方法体中的内容，是我们具体要做的事情

### Lambda表达式的代码分析

```

1  new Thread( () → {
2      System.out.println("多线程徐启动了");
3  } ).start();

```

- `()`: 里面没有内容，可以看成是方法形式参数为空



- ->: 用箭头指向后面要做的事情
- {}: 包含一段代码, 我们称之为代码块, 可以看成是方法体中的内容

组成Lambda表达式的三要素: 形式参数、箭头、代码块

### Lambda表达式的格式

- 格式:

1 (形式参数) → {代码块}

- 形式参数: 如果有多个参数, 参数之间用逗号隔开; 如果没有参数, 留空即可
- ->: 由英文中划线和大于符号组成, 固定写法, 代表指向动作
- 代码块: 是我们具体要做的事情, 也就是以前我们写的方法体内容

## Lambda表达式的练习

### Lambda表达式的使用前提

- 有一个接口
- 接口中有且仅有一个抽象方法

### 练习1

- 定义一个接口(Eatable), 里面定义一个抽象方法:void eat()
- 定义一个测试类(EatableDemo), 在测试类中提供两个方法:
  - 一个方法是: useEatable(Eatable e)
  - 一个方法是主方法: 在主方法中调用useEatable()方法

```
1 public interface Eatable {  
2     void eat();  
3 }
```

```
1 public class EatableImpl implements Eatable {  
2     @Override  
3     public void eat() {  
4         System.out.println("一天一苹果, 医生远离我");  
5     }  
6 }
```

```

1 public class EatableDemo {
2     public static void main(String[] args) {
3         Eatable e = new EatableImpl();
4         useEatable(e);
5
6         // 匿名内部类
7         useEatable(new EatableImpl() {
8             @Override
9             public void eat() {
10                 System.out.println("一天一苹果, 医生远离我");
11             }
12         });
13
14         // Lambda表达式
15         useEatable(() -> {
16             System.out.println("一天一苹果, 医生远离我");
17         });
18     }
19
20     private static void useEatable(Eatable e) {
21         e.eat();
22     }
23 }

```

## Lambda表达式的使用

### 练习2

- 定义一个接口(Flyable)，里面定义一个抽象方法: void fly(String s)
- 定义一个测试类 (FlyableDemo)，在测试类中提供两个方法：
  - 一个方法是： useFlyable(Flyable f)
  - 一个方法是主方法：在主方法中调用useFlyable()方法

```

1 public interface Flyable {
2     void fly(String s);
3 }

```

```

1 public class FlyableDemo {
2     public static void main(String[] args) {
3         useFlyable((String s) → {
4             System.out.println(s);
5             System.out.println("飞机自驾游");
6         });
7     }
8
9     private static void useFlyable(Flyable f) {
10        f.fly("风和日丽, 晴空万里");
11    }
12 }

```

## Lambda表达式的使用

### 练习3

- 定义一个接口(Addable), 里面定义一个抽象方法: void add(int x, int y)
- 定义一个测试类 (AddableDemo), 在测试类中提供两个方法:
  - 一个方法是: useAddable(Addable a)
  - 一个方法是主方法: 在主方法中调用useAddable()方法

```

1 public interface Addable {
2     int add(int x, int y);
3 }

```

```

1 public class AddableDemio {
2     public static void main(String[] args) {
3         useAddable((int x, int y) → {
4             return x + y;
5         });
6     }
7
8     public static void useAddable(Addable a) {
9         int sum = a.add(10, 20);
10        System.out.println(sum);
11    }
12 }

```

## Lambda表达式的省略模式

### 省略规则

- 参数类型可以省略，但多个参数的情况下，不能只省略一个
- 如果参数有且仅有一个，那么小括号可以省略
- 如果代码块的语句只有一条，可以省略大括号和分号，如果有return，可以省略return

## Lambda表达式的注意事项

### 注意事项

- 使用Lambda必须要有接口，并且要求接口中有且仅有一个抽象方法
- 必须要有上下文环境，才能推导出Lambda对应的接口
  - 根据 局部变量的赋值 得知Lambda对应的接口：

```
Runnable r = () -> System.out.println("Lambda表达式");
```
  - 根据 调用方法的参数 得知Lambda对应的接口：

```
new Thread() -> System.out.println("Lambda表达式").start();
```

## Lambda表达式和匿名内部类的区别

- 所需类型不同
  - 匿名内部类：可以是接口，也可以是抽象类，还可以是具体类
  - Lambda表达式：只能是接口
- 使用限制不同
  - 如果接口中有且仅有一个抽象方法，可以使用匿名内部类，也可以使用Lambda表达式
  - 如果接口中有多于一个抽象方法，只能使用匿名内部类，而不能使用Lambda表达式
- 实现原理不同
  - 匿名内部类：编译之后，产生一个单独的class字节码文件
  - Lambda表达式：编译之后，没有一个单独的class字节码文件。对应的字节码会在运行的时候动态生成

# 接口组成更新

## ✧ 接口组成更新描述

### 接口的组成

- 常量
  - `public static final`
- 抽象方法
  - `public abstract`
- 默认方法（Java8）
- 静态方法（Java8）
- 私有方法（Java9）

## ✧ 接口中默认方法

### 默认方法的定义格式

- 格式：

```
1 public default 返回值类型 方法名(参数列表) {}
```

### 接口中默认方法注意事项

- 默认方法不是抽象方法，所以不强制被重写，但是可以被重写，重写的时候去掉`default`关键字
- `public`可以省略，`default`不能省略

## ✧ 接口中静态方法

### 接口中静态方法定义格式

- 格式:

```
1 public static 返回值类型 方法名(参数列表) {}
```

### 接口中静态方法的注意事项

- 静态方法只能通过接口名调用，不能通过实现类名或者对象名调用
- `public` 可以省略，`static` 不能省略

## ✧ 接口中私有方法

---

### 接口中私有方法定义格式

- 格式1:

```
1 private 返回值类型 方法名(参数列表) {}
```

- 格式2:

```
1 private static 返回值类型 方法名(参数列表) {}
```

### 接口中私有方法注意事项

- 默认方法可以调用私有的静态方法和非静态方法
- 静态方法只能调用私有的静态方法

## 方法引用

## ✧ 体验方法引用

---

```
1 public interface Printable {
2     void printString(String s);
3 }
```

```
1 public class PrintableDemo {
2     public static void main(String[] args) {
3         usePrintable(s → System.out.println(s));
4
5         // 方法引用符: ::
6         usePrintable(System.out::println);
7     }
8
9     public static void usePrintable(Printable p) {
10        p.printString("爱生活爱Java");
11    }
12 }
```

## ✧ 方法引用符

---

- `::` 该符号为引用运算符，而它所在的表达式被称为方法引用

### 推导与省略

- 如果使用Lambda，那么根据“可推导就是可省略”的原则，无需指定参数类型，也无需指定重载形式，它们都将被自动推导
- 如果使用方法引用，也是同样可以根据上下文进行推导
- 方法引用是Lambda的孪生兄弟

## ✧ Lambda表达式支持的方法引用

---

### 常见的引用方式

- 引用类方法
- 引用对象的实例方法
- 引用类的实例方法

- 引用构造器

## 引用类方法

- 引用类方法，其实就是引用类的静态方法
- 格式：

```
1 类名::静态方法
```

- 范例：

```
1 Integer::parseInt
```

- Integer类的方法：`public static parseInt(String s)` 将此String转换为int类型数据

## 引用对象的实例方法

- 引用对象的实例方法，其实就是引用类中的成员方法
- 格式：

```
1 对象::成员方法
```

- 范例：

```
1 "HelloWorld"::toUpperCase
```

- String类中的方法：`public String toUpperCase()` 将此String所有字符转换为大写

## 引用类的实例对象

- 引用类的实例对象，其实就是引用类中的成员方法
- 格式：

```
1 类名::成员方法
```

- 范例：

```
1 String::substring
```



- String 类中的方法：`public String substring(int beginIndex, int endIndex)` 从beginIndex开始到endIndex结束，截取字符串，返回一个子串，子串的长度为endIndex - beginIndex

## 引用构造器

- 引用构造器，其实就是引用构造方法
- 格式：

```
1 类名::new
```

- 范例：

```
1  Student::new
```

# 函数式接口

## ✧ 函数式接口概述

- 函数式接口：有且仅有一个抽象方法的接口
- Java中的函数式编程体现就是Lambda表达式，所以函数式接口就是可以适用于Lambda使用的接口
- 只有确保接口中有且仅有一个抽象方法，Java中的Lambda才能顺利地进行推导

**i** 如何检测一个接口是否是函数式接口？

- `@FunctionalInterface`
- 放在定义接口的上方，如果接口是函数式接口，编译通过；如果不是，编译失败

**i** 注意：

- 我们自己定义函数式接口的时候，`@FunctionalInterface` 是可选的，就算不写这个注解，只要保证满足函数式接口的定义条件，也照样是函数式接口。但是，`建议加上该注解`

## ✧ 函数式接口作为方法的参数

---

- 如果方法的参数是一个函数式接口，可以使用Lambda表达式作为参数传递

```
1 start() → System.out.println(Thread.currentThread().getName() +  
  "线程启动了");
```

## ✧ 函数式接口作为方法的返回值

---

- 如果方法的返回值是一个函数式接口，可以使用Lambda表达式作为结果返回

```
1 private static Comparator<String> getComparator() {  
2     return (s1, s2) → s1.length() - s2.length();  
3 }
```

## ✧ 常用的函数式接口

---

- Supplier接口
- Consumer接口
- Predicate接口
- Function接口

### Supplier接口

- `Supplier<T>`：包含一个无参的方法
- `T get()`：获得结果

- 该方法不需要参数，它会按照某种实现逻辑由Lambda表达式（实现）返回一个数据
- `Supplier<T>` 接口也被称为生产型接口，如果我们指定了接口的泛型是什么类型，那么接口中的 `get()` 方法就会产生什么类型的数据供我们使用

## Supplier接口练习之获取最大值

```
1 public class SupplierTest {
2     public static void main(String[] args) {
3         // 定义一个int数组
4         int[] arr = {19, 50, 28, 46};
5
6         int maxValue = getMax(() -> {
7             int max = arr[0];
8
9             for (int i = 1; i < arr.length; i++) {
10                 if (arr[i] > max) {
11                     max = arr[i];
12                 }
13             }
14
15             return max;
16         });
17
18         System.out.println(maxValue);
19     }
20
21     // 返回一个int数组中的最大值
22     public static int getMax(Supplier<Integer> sup) {
23         return sup.get();
24     }
25 }
```

## Consumer接口

- `Consumer<T>`：包含两个方法
- `void accept(T t)`：对给定的参数执行此操作
- `defaultConsumer<T> andThen(Consumer after)`：返回一个组合的Consumer，依次执行此操作，然后执行Then操作

- `Consumer<T>` 接口也被称为消费型接口，他消费的数据的数据类型由泛型指定

## Consumer接口之按照要求打印信息

```
1 public class ConsumerTest {
2     public static void main(String[] args) {
3         String[] strArray = {"林青霞,30", "张曼玉,35", "王祖贤,33"};
4
5         printInfo(strArray, str → {
6             String name = str.split(",")[0];
7             System.out.print("姓名: " + name);
8         }, str → {
9             int age = Integer.parseInt(str.split(",")[1]);
10            System.out.println(",年龄: " + age);
11        });
12    }
13
14    private static void printInfo(String[] array,
15    Consumer<String> con1, Consumer<String> con2) {
16        for (String s : array) {
17            con1.andThen(con2).accept(s);
18        }
19    }
```

## Predicate接口

- `Predicate<T>` : 常用的四个方法
- `boolean test(T t)` : 对给定的参数进行判断（判断逻辑由Lambda表达式实现），返回一个布尔值
- `default Predicate<T> negate()` : 返回一个逻辑的否定，对应逻辑非
- `default Predicate<T> and(Predicate other)` : 返回一个组合判断，对应短路与
- `default Predicate<T> or(Predicate other)` : 返回一个组合判断，对应短路或
- `Predicate<T>` 接口通常用于判断参数是否满足指定的条件

## Predicate接口之筛选满足条件的数据

- `String[] strArray = {"林青霞,30", "柳岩,34", "张曼玉,35", "貂蝉,31", "王祖贤,33"};`
- 字符串数组中有多条信息，请通过Predicate接口的拼装将符合要求的字符串筛选到集合ArrayList中，并遍历ArrayList集合
- 同时满足如下要求：姓名长度大于2，年龄大于33

```
1  import java.util.ArrayList;
2  import java.util.function.Predicate;
3
4  public class PreducateTest {
5      public static void main(String[] args) {
6          String[] strArray = {"林青霞,30", "柳岩,34", "张曼玉,35",
7                               "貂蝉,31", "王祖贤,33"};
8
9          ArrayList<String> array = myFilter(strArray, s →
10 s.split(",")[0].length() > 2, s → Integer.parseInt(s.split(",")
11 [1]) > 33);
12
13         for (String str : array) {
14             System.out.println(str);
15         }
16     }
17
18     private static ArrayList<String> myFilter(String[] strArray,
19 Predicate<String> pre1, Predicate<String> pre2) {
20         ArrayList<String> array = new ArrayList<String>();
21
22         // 遍历数组
23         for (String str : strArray) {
24             if (pre1.and(pre2).test(str)) {
25                 array.add(str);
26             }
27         }
28
29         return array;
30     }
31 }
```

## Function接口

- `Function<T, R>` : 常用的两个方法
- `R apply(T t)` : 将此函数应用于给定的参数
- `default <V> Function andThen(Function after)` : 返回一个组合函数，首先将该函数应用于输入，然后将after函数应用于结果
- `Function<T, R>` 接口通常用于对参数进行处理，转换（处理逻辑由Lambda表达式实现），然后返回一个新的值

### Function接口之按照指定要求操作数据

```
1  import java.util.function.Function;
2
3  public class FunctionTest {
4      public static void main(String[] args) {
5          String s = "林青霞,30";
6
7          // convert(s, ss → ss.split(",")[1], ss →
Integer.parseInt(ss), i → i + 70);
8          convert(s, ss → ss.split(",")[1], Integer::parseInt, i -
> i + 70);
9      }
10
11     private static void convert(String s, Function<String,
String> fun1, Function<String, Integer> fun2, Function<Integer,
Integer> fun3) {
12         int i = fun1.andThen(fun2).andThen(fun3).apply(s);
13         System.out.println(i);
14     }
15 }
```

## Stream流

### ✧ Stream流的生成方式



- 生成流
  - 通过数据源（集合、数组等）生成流
  - `list.stream()`
- 中间操作
  - 一个流后面可以跟零个或多个中间操作，其目的主要是打开流，做出某种程度的数据过滤/映射，然后返回一个新的流，交给下一个操作使用
  - `filter()`
- 终结操作
  - 一个流只能有一个终结操作，当这个操作执行后，流就被使用“光”了，无法再被操作，所以这必定是流的最后一个操作
  - `forEach()`

#### Stream流常见生成方式

- Collection体系的集合可以使用默认方法 `stream()` 生成流
  - `default Stream<E> stream()`
- Map体系的集合间接的生成流
- 数组可以通过Stream接口的静态方法 `of(T... values)` 生成流

## ✧ Stream流中常见的中间操作

- `Stream<T> filter(Predicate predicate)`：用于对流中的数据进行过滤
- `Stream<T> limit(long maxSize)`：返回此流中的元素组成的流，截取前指定参数个数的数据
- `Stream<T> skip(long n)`：跳过指定参数个数的数据，返回由该流的剩余参数组成的流
- `static <T> Stream<T> concat(Stream a, Stream b)`：合并a和b两个流为一个流
- `Stream<T> distinct()`：返回由该流的不同元素（根据 `Object.equals(Object)`）组成的流

- `Stream<T> sorted()` : 返回由此流的元素组成的流，根据自然顺序进行排序
- `Stream<T> sorted(Comparator)` : 返回由该流的元素组成的流，根据提供的Comparator进行排序
- `<R> Stream<R> map(Function mapper)` : 返回由给定的函数应用于此流的元素的结果组成的流
- `IntStream mapToInt(ToIntFunction mapper)` : 返回一个IntStream其中包含将给定函数应用于此流的元素的结果

## ✳ Stream流的常见终结操作方法

---

- `void forEach(Consumer action)` : 对此流的每个元素执行操作
- `long count()` : 返回此流中的元素数

## ✳ Stream流的收集操作

---

- `R collect(Collector collector)`
- 但是这个收集方法的参数是一个Collector接口

**i** 工具类Collectors提供了具体的收集方法

- `public static <T> Collector toList()` : 把元素收集到List集合中
- `public static <T> Collector toSet()` : 把元素收集到Set集合中
- `public static Collector toMap(Function keyMapper, Function valueMapper)` : 把元素收集到Map集合中

# 反射



## 类加载

当程序要使用某个类时，如果该类还未被加载到内存中，则系统会通过类的加载、类的连接、类的初始化这三个步骤来对类进行初始化。如果不出现意外情况，JVM会连续完成这三个步骤，所以有时也把这三个步骤统称为类加载或类初始化。

### i 类的加载

- 就是将class文件读入内存，并为之创建一个java.lang.Class对象
- 任何类被使用时，系统都会为之建立一个java.lang.Class对象

### i 类的连接

- 验证阶段：用于检验被加载的类是否有正确的内部结构，并和其他类协调一致
- 准备阶段：负责为类的类变量分配内存，并设置默认初始化值
- 解析阶段：将类的二进制数据中的符号引用替换为直接引用

### i 类的初始化

- 在该阶段，主要就是对类变量进行初始化

### i 类的初始化步骤

- 假如类还未被加载和连接，则程序先加载并连接该类
- 假如类的直接父类还未被初始化，则先初始化其直接父类
- 假如类中有初始化语句，则系统依次执行这些初始化语句

i 注意：在执行第二个步骤的时候，系统对直接父类的初始化步骤也是初始化步骤1-3

### i 类的初始化时机

- 创建类的实例

- 调用类的方法
- 访问类或者接口的类变量，或者为该类变量赋值
- 使用反射方式来强制创建某个类或接口对应的`java.lang.Class`对象
- 初始化某个类的子类
- 直接使用`java.exe`命令来运行整个主类

## 类加载器

### 类加载器的作用

- 负责将.class文件加载到内存中，并为之生成对应的`java.lang.Class`对象
- 虽然不用过分关心类加载机制，但了解这个机制就能更好地理解程序的运行

### JVM的类加载机制

- 全盘负责：就是当一个类加载器负责加载某个Class时，该Class所依赖的和引用的其他Class也将由该类加载器负责载入，除非显示使用另一个类加载器来载入
- 父类委托：就是当一个类加载器负责加载某个Class时，先让父类加载器试图加载该Class，只有在父类加载器无法加载该类时才尝试从自己的类路径中加载该类
- 缓存机制：保证所有被加载过的Class都会被缓存，当程序需要使用某个Class对象时，类加载器先从缓存区中搜索该Class，只有当缓存区中不存在该Class对象时，系统才会读取该类对应的二进制数据，并将其转换成Class对象，存储到缓存区

### Java运行时具有以下内置类加载器

- Bootstrap class loader：它是虚拟机的内置类加载器，通常表示为`null`，并且没有父`null`
- Platform class loader：平台类加载器可以看到所有平台类，平台类包括由平台类加载器或其祖先定义的JavaSE平台API，其实现类和JDK特有的运行时类
- System class loader：它也被称为应用程序类加载器，与平台类加载器不同，系统类加载器通常用于定义应用程序类路径、模块路径和JDK特定工具上的类

- 类加载器的继承关系：`System`的父加载器为`Platform`，而`Platform`的父加载器为`Bootstrap`

#### `ClassLoader` 中的两个方法

- `static ClassLoader getClassLoader()`：返回用于委派的系统类加载器
- `ClassLoader getParent()`：返回父类加载器进行委派

## ✧ 反射

### 反射概述

**Java反射机制**：是指在运行时去获取一个类的变量和方法信息，然后通过获取到的信息来创建对象，调用方法的一种机制。由于这种动态性，可以极大地增强程序的灵活性，程序不用在编译期就完成确定，在运行期仍然可以扩展。

### 获取Class类的对象

- 实用类的 `class` 属性来获取该类对应的Class对象
  - 举例：`Student.class` 将会返回`Student`类对应的Class对象
  - 基本数据类型也可以通过`.class`得到对应的Class属性
- 调用对象的 `getClass()` 方法，返回对象所属类对应的Class对象
  - 该方法是`Object`类中的方法，所有的Java对象都可以调用该方法
- 使用Class类中的静态方法 `forName(String className)`，该方法需要传入字符串参数，该字符串参数的值是某个类的全路径，也就是完整包名的路径

### 反射获取构造方法并使用

#### Class类中用于获取构造方法的方法

- `Constructor<?>[] getConstructors()` : 返回所有公共构造方法对象的数组
- `Constructor<?>[] getDeclaredConstructors()` : 返回所有构造方法对象的数组
- `Constructor<T> getConstructor(Class<?>... parameterTypes)` : 返回单个公共构造方法对象
- `Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes)` : 返回单个构造方法对象

**i** Constructor类中创建对象的方法

- `T newInstance(Object... initargs)` : 根据指定的构造方法创建对象

## 反射获取成员变量并使用

- `Field[] getFields()` : 返回所有公共成员变量对象的数组
- `Field[] getDeclaredFields()` : 返回所有成员变量对象的数组
- `Field getField(String name)` : 返回单个公共成员变量对象
- `Field getDeclaredField(String name)` : 返回单个成员变量对象

**i** Field类中用于给成员变量赋值的方法

- `void set(Object obj, Object value)` : 给Object对象的成员变量赋值为value

## 反射获取成员方法并使用

- `Method[] getMethods()` : 返回所有公共成员方法对象的数组，不包括继承的
- `Method[] getDeclaredMethods()` : 返回所有成员方法对象的数组，不包括继承的
- `Method getMethod(String name, Class<?>... parameterTypes)` : 返回单个公共成员方法对象

- `Method getDeclaredMethod(String name, Class<?>... parameterTypes)` : 返回单个成员方法对象

**i** Method类中用于调用成员方法的对象

- `Object invoke(Object obj, Object... args)` : 调用obj对象的成员方法，参数是args，返回值是Object类型

## 反射练习

**i** 练习1: 有一个ArrayList集合，现在想在这个集合中添加一个字符串数据，如何实现？

```
1 import java.lang.reflect.InvocationTargetException;
2 import java.lang.reflect.Method;
3 import java.util.ArrayList;
4
5 public class ReflectTest {
6     public static void main(String[] args) throws
        NoSuchMethodException, InvocationTargetException,
        IllegalAccessException {
7         // 创建集合
8         ArrayList<Integer> array = new ArrayList<Integer>();
9
10        Class<? extends ArrayList> c = array.getClass();
11        Method m = c.getMethod("add", Object.class);
12        m.invoke(array, "hello");
13        m.invoke(array, "world");
14        m.invoke(array, "java");
15
16        System.out.println(array);
17    }
18 }
```


**i** 练习2: 通过配置文件运行类中的方法

```
1 import java.io.FileReader;
2 import java.io.IOException;
3 import java.lang.reflect.Constructor;
4 import java.lang.reflect.InvocationTargetException;
5 import java.lang.reflect.Method;
```

```

6  import java.util.Properties;
7
8  public class ReflectTest {
9      public static void main(String[] args) throws IOException,
        ClassNotFoundException, NoSuchMethodException,
        InvocationTargetException, InstantiationException,
        IllegalAccessException {
10         // 加载数据
11         Properties prop = new Properties();
12         FileReader fr = new FileReader("myReflect\\class.txt");
13         prop.load(fr);
14         fr.close();
15
16         String className = prop.getProperty("className");
17         String methodMame = prop.getProperty("methodMame");
18
19         // 通过反射来使用
20         Class<?> c = Class.forName(className);
21
22         Constructor<?> con = c.getConstructor();
23         Object obj = con.newInstance();
24
25         Method m = c.getMethod(methodMame);
26         m.invoke(obj);
27     }
28 }

```

 class.txt


```

1  className=com.itheima_06.Teacher
2  methodName=teach

```

## 模块化

### ✧ 模块的基本使用

 模块的基本使用步骤

- 创建模块（按照以前讲解的方式创建模块，创建包、创建类、定义方法）
- 在模块的src目录下新建一个名为 `module-info.java` 的描述性文件，该文件专门定义模块名、访问权限、模块依赖等信息
  - 描述性文件中使用模块导出和模块依赖来进行配置并使用
- 模块中所有未导出的包都是模块私有的，它们是不能在模块之外被访问的
  - 模块导出格式： `export 包名;`
- 一个模块要访问其他模块，必须明确指定依赖哪些模块，未明确指定依赖的模块不能访问
  - 模块依赖格式： `requires 模块名;`
  - 注意： 写模块名报错，需要按下ALT+ENTER提示，然后选择模块依赖

## ✧ 模块服务的使用

---

- 模块导出： `export com.itheima_03;`
- 服务提供： `provides MyService with Itheima;` 指定MyService的服务实现类是Itheima
- 声明服务接口： `uses MyService;`
- 使用接口提供的服务
  - `ServiceLoader`：一种加载服务实现的工具
  - `ServiceLoader<MyService> myServices = ServiceLoader.load(MyService.class);`