

第一章 Java概述

1.1 Java历史

Java诞生于SUN（Stanford University Network），09年SUN被Oracle（甲骨文）收购。

Java之父是詹姆斯.高斯林(James Gosling)。

1996年发布JDK1.0版。

目前最新的版本是Java12。我们学习的Java8。

1.2 Java语言最主要的特点

- 特点一：面向对象

两个基本概念：类、对象

三大特性：封装、继承、多态

- 特点二：健壮性

吸收了C/C++语言的优点，但去掉了其影响程序健壮性的部分（如指针、内存的申请与释放等），提供了一个相对安全的内存管理和访问机制

- 特点三：跨平台性

跨平台性：通过Java语言编写的应用程序在不同的系统平台上都可以运行。“Write once , Run Anywhere”一次编写，处处运行。

原理：只要在需要运行 java 应用程序的操作系统上，先安装一个Java虚拟机 (JVM Java Virtual Machine) 即可。由JVM来负责Java程序在该系统中的运行。因为有了JVM，同一个Java 程序在三个不同的操作系统中都可以执行。这样就实现了Java 程序的跨平台性。

1.3 Java环境搭建

1.3.1 JDK、JRE、JVM

Java开发人员需要安装JDK。如果仅仅是运行Java程序，那么只需要按照JRE。

JDK（Java Development kits）：Java开发工具包。

JRE（Java Runtime Environment）：Java运行环境。

JVM（Java Virtual Machine）：Java虚拟机。

JDK = JRE + 开发工具（javac.exe,java.exe,javadoc.exe等）

JRE = JVM + 核心类库（常用类：String、日期时间、数学、集合、IO、网络、多线程等）

1.3.2 Java环境搭建

1、安装JDK

2、配置JDK的开发工具目录到path环境变量中

例如：D:\ProgramFiles\Java\jdk1.8.0_51\bin;

注意：这个安装目录以你自己的安装目录为准

(1) 为什么配置path?

希望在命令行使用javac.exe等工具时，任意目录下都可以找到这个工具所在的目录。

(2) 如何配置环境变量?

【计算机】右键【属性】，选择【高级系统设置】，选择【高级】，选择【环境变量】，选择【系统环境变量】，编辑path，在【path原有值】的前面加入D:\ProgramFiles\Java\jdk1.8.0_51\bin;

1.4 第一个Java应用程序

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

1.4.1 Java程序的开发步骤

三步：

1、编辑/编写源代码

要求：源文件必须是.java文件

2、编译

目的：把源文件编译为.class字节码文件（因为JVM只认识字节码）

工具：javac.exe

格式：

```
javac 源文件名.java
```

3、运行

工具：java.exe

格式：

```
java 类名
java 字节码文件名
```

要求：可以被运行的类，必须包含main方法

1.4.2 Java程序的结构与格式

结构：

```
类{
    方法{
        语句;
    }
}
```

格式：

- (1) 每一级缩进一个Tab键
- (2) {}的左半部分在行尾，右半部分单独一行，与和它成对的"{"的行首对齐

1.4.3 Java程序的入口

Java程序的入口是main方法

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

}
```

1.4.4 Java注释

1、单行注释

```
//注释内容
```

2、多行注释

```
/*
注释内容
*/
```

3、文档注释

```
/**
文档注释（后面注解部分讲解）
*/
```

1.5 编写Java程序时应该注意的问题

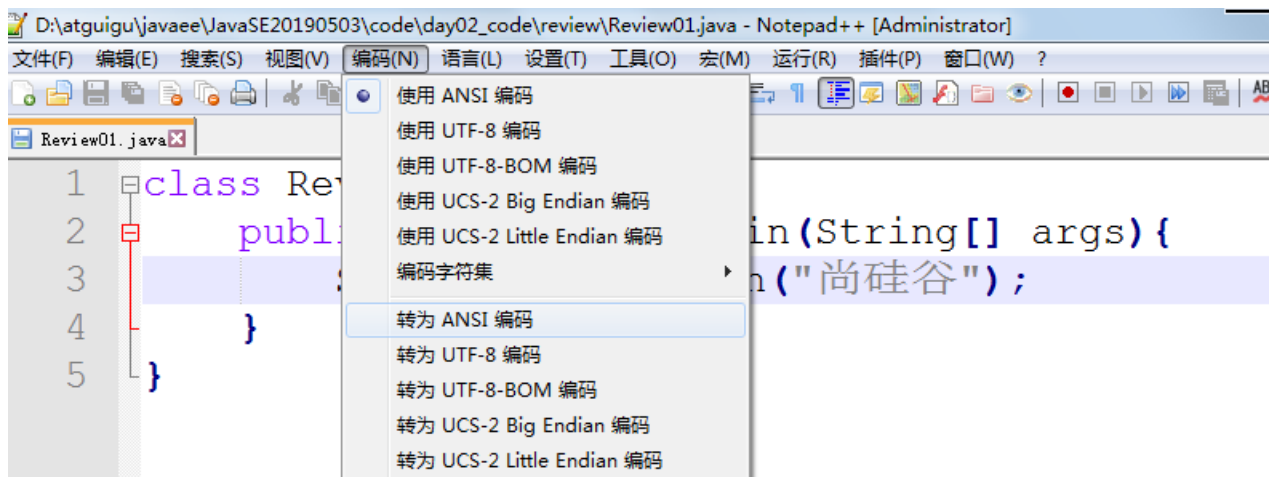
1、字符编码问题

当cmd命令窗口的字符编码与.java源文件的字符编码不一致，如何解决？

```
D:\atguigu\javaee\JavaSE20190503\code\day02_code\review>javac Review01.java
Review01.java:3: 错误: 编码GBK的不可映射字符
    System.out.println("灏氮 璋?");
                        ^
1 个错误
```

解决方案一：

在Notepad++等编辑器中，修改源文件的字符编码



解决方案二：

在使用javac命令式，可以指定源文件的字符编码

```
javac -encoding utf-8 Review01.java
```

2、大小写问题

(1) 源文件名：

不区分大小写，我们建议大家还是区分

(2) 字节码文件名与类名

区分大小写

(3) 代码中

区分大小写

3、源文件名与类名一致问题？

(1) 源文件名是否必须与类名一致？ public呢？

如果这个类不是public，那么源文件名可以和类名不一致。

如果这个类是public，那么要求源文件名必须与类名一致。

我们建议大家，不管是否是public，都与源文件名保持一致，而且一个源文件尽量只写一个类，目的是为了好维护。

(2) 一个源文件中是否可以有多个类？ public呢？

一个源文件中可以有多个类，编译后会生成多个.class字节码文件。

但是一个源文件只能有一个public的类。

(3) main必须在public的类中吗？

不是。

但是后面写代码时，基本上main习惯上都在public类中。

第二章 Java的基础语法

2.1 标识符

简单的说，凡是程序员自己命名的部分都可以称为标识符。

即给类、变量、方法、包等命名的字符序列，称为标识符。

1、标识符的命名规则

(1) Java的标识符只能使用26个英文字母大小写，0-9的数字，下划线_，美元符号\$

(2) 不能使用Java的关键字（包含保留字）和特殊值

- (3) 数字不能开头
- (4) 不能包含空格
- (5) 严格区分大小写

2、标识符的命名规范

- (1) 见名知意

- (2) 类名、接口名等：每个单词的首字母都大写，形式：XxxYyyZzz，

例如：HelloWorld, String, System等

- (3) 变量、方法名等：从第二个单词开始首字母大写，其余字母小写，形式：xxxYyyZzz，

例如：age,name,bookName,main

- (4) 包名等：每一个单词都小写，单词之间使用点.分割，形式：xxx.yyy.zzz，

例如：java.lang

- (5) 常量名等：每一个单词都大写，单词之间使用下划线_分割，形式：XXX_YYY_ZZZ，

例如：MAX_VALUE,PI

2.2 变量

2.2.1 变量的概念

变量的作用：用来存储数据，代表内存的一块存储区域，变量中的值是可以改变的。

2.2.2 变量的三要素

- 1、数据类型
- 2、变量名
- 3、值

2.2.3 变量的使用应该注意什么？

- 1、先声明后使用

如果没有声明，会报“找不到符号”错误

- 2、在使用之前必须初始化

如果没有初始化，会报“未初始化”错误

- 3、变量有作用域

如果超过作用域，也会报“找不到符号”错误

4、在同一个作用域中不能重名

2.2.4 变量的声明和赋值、使用的语法格式？

1、变量的声明的语法格式：

```
数据类型  变量名;  
例如:  
int age;  
String name;  
double weight;  
char gender;  
boolean isMarry;
```

2、变量的赋值的语法格式：

```
变量名 = 值;  
例如:  
age = 18;  
name = "柴林燕"; //字符串的值必须用"  
weight = 44.4;  
gender = '女'; //单字符的值必须使用'  
isMarry = true;
```

3、变量的使用的语法格式：

通过变量名直接引用

例如：

(1) 输出变量的值

```
System.out.print(name);  
System.out.print("姓名: " + name); //""中的内容会原样显示  
System.out.print("name = " + name);
```

(2) 计算

```
age = age + 1;
```

2.3 数据类型

2.3.1 Java数据类型的分类

1、基本数据类型

8种：整型系列（byte,short,int,long）、浮点型(float,double)、单字符型（char）、布尔型（boolean）

2、引用数据类型

类、接口、数组、枚举.....

2.3.2 Java的基本数据类型

1、整型系列

(1) byte: 字节类型

占内存: 1个字节

存储范围: -128~127

(2) short: 短整型类型

占内存: 2个字节

存储范围: -32768~32767

(3) int: 整型

占内存: 4个字节

存储范围: -2^{31} ~ $2^{31}-1$

(4) long: 长整型

占内存: 8个字节

存储范围: -2^{63} ~ $2^{63}-1$

注意: 如果要表示某个常量数字它是long类型, 那么需要在数字后面加L

2、浮点型系列 (小数)

(1) float: 单精度浮点型

占内存: 4个字节

精度: 科学记数法的小数点后6~7位

注意: 如果要表示某个常量数字是float类型, 那么需要在数字后面加F或f

(2) double: 双精度浮点型

占内存: 8个字节

精度: 科学记数法的小数点后15~16位

3、单字符类型

char: 字符类型

占内存: 2个字节

Java中使用的字符集: Unicode编码集

字符的三种表示方式:

(1) '一个字符'

例如: 'A', '0', '尚'

(2) 转义字符

```
\n: 换行
\r: 回车
\t: Tab键
\\: \
\: "
\' :
\b: 删除键Backspace
```

(3) \u字符的Unicode编码值的十六进制型

例如: \u5c1a代表'尚'

4、布尔类型

boolean: 只能存储true或false

2.3.3 进制（了解，可以暂时忽略）

1、进制的分类:

(1) 十进制

数字组成: 0-9

进位规则: 逢十进一

(2) 二进制

数字组成: 0-1

进位规则: 逢二进一

(3) 八进制

数字组成: 0-7

进位规则: 逢八进一

(4) 十六进制

数字组成: 0-9, a~f (或A~F)

进位规则: 逢十六进一

2、请分别用四种类型的进制来表示10，并输出它的结果: （了解）

(1) 十进制: 正常表示

```
System.out.println(10);
```

(2) 二进制: 0b或0B开头

```
System.out.println(0B10);
```

(3) 八进制: 0开头

```
System.out.println(010);
```

(4) 十六进制: 0x或0X开头

```
System.out.println(0X10);
```

3、为什么byte是-128~127? (理解)

1个字节: 8位

0000 0001 ~ 0111 111 ==> 1~127

1000 0001 ~ 1111 1111 ==> -127 ~ -1

0000 0000 ==> 0

1000 0000 ==> -128 (特殊规定)

解释: 计算机数据的存储 (了解)

计算机数据的存储使用二进制补码形式存储, 并且最高位是符号位, 1是负数, 0是正数。

规定: 正数的补码与反码、原码一样, 称为三码合一;

负数的补码与反码、原码不一样:

负数的原码: 把十进制转为二进制, 然后最高位设置为1

负数的反码: 在原码的基础上, 最高位不变, 其余位取反 (0变1, 1变0)

负数的补码: 反码+1

例如: byte类型 (1个字节, 8位)

25 ==> 原码 0001 1001 ==> 反码 0001 1001 --> 补码 0001 1001

-25 ==> 原码 1001 1001 ==> 反码 1110 0110 ==> 补码 1110 0111

底层是用加法代替减法: $-128 == \gg -127 - 1 == \gg -127 + (-1)$

$-127 - -1 ==> -127 + 1$

4、学生疑惑解答?

(1) 为什么float (4个字节) 比long (8个字节) 的存储范围大?

(2) 为什么double (8个字节) 比float (4个字节) 精度范围大?

因为float、double底层也是二进制，先把小数转为二进制，然后把二进制表示为科学记数法，然后只保存：

- (1) 符号位 (2) 指数位 (3) 尾数位

详见《float型和double型数据的存储方式.docx》

2.3.4 基本数据类型的转换

1、自动类型转换

- (1) 当把存储范围小的值（常量值、变量的值、表达式计算的结果值）赋值给了存储范围大的变量时，

byte->short->int->long->float->double

char->

```
int i = 'A'; //char自动升级为int
double d = 10; //int自动升级为double
```

- (2) 当存储范围小的数据类型与存储范围大的数据类型一起混合运算时，会按照其中最大的类型运算

```
int i = 1;
byte b = 1;
double d = 1.0;

double sum = i + b + d; //混合运算，升级为double
```

- (3) 当byte,short,char数据类型进行算术运算时，按照int类型处理

```
byte b1 = 1;
byte b2 = 2;
byte b3 = (byte)(b1 + b2); //b1 + b2自动升级为int

char c1 = '0';
char c2 = 'A';
System.out.println(c1 + c2); //113
```

- (4) boolean类型不参与

2、强制类型转换

- (1) 当把存储范围大的值（常量值、变量的值、表达式计算的结果值）赋值给了存储范围小的变量时，需要强制类型转换

double->float->long->int->short->byte

->char

提示：有风险，可能会损失精度或溢出

```
double d = 1.2;
int num = (int)d; //损失精度

int i = 200;
byte b = (byte)i; //溢出
```

(2) boolean类型不参与

(3) 当某个值想要提升数据类型时，也可以使用强制类型转换

```
int i = 1;
int j = 2;
double shang = (double)i/j;
```

提示：这个情况的强制类型转换是没有风险的。

2.3.5 特殊的数据类型转换

1、任意数据类型的数据与String类型进行“+”运算时，结果一定是String类型

```
System.out.println("" + 1 + 2); //12
```

2、但是String类型不能通过强制类型()转换，转为其他的类型

```
String str = "123";
int num = (int)str; //错误的
```

2.4 运算符

1、按照操作数个数的分类：

(1) 一元运算符：操作数只有一个

例如：正号 (+)，负号 (-)，自增 (++)，自减 (--)，逻辑非 (!)，按位取反 (~)

(2) 二元运算符：操作数有两个

例如：加 (+)，减 (-)，乘 (*)，除 (/)，模 (%)

大于 (>)，小于 (<)，大于等于 (>=)，小于等于 (<=)，等于 (==)，不等于 (!=)

赋值 (=, +=, -=, *=, /=, %=, >>=, <<=。。。)

逻辑与 (&)，逻辑或 (|)，逻辑异或 (^)，短路与 (&&)，短路或 (||)

左移 (<<)，右移 (>>)，无符号右移 (>>>)，按位与 (&)，按位或 (|)，按位异或 (^)

(3) 三元运算符：操作数三个

例如：？：

2、Java基本数据类型的运算符：

(1) 算术运算符

(2) 赋值运算符

(3) 比较运算符

(4) 逻辑运算符

(5) 条件运算符

(6) 位运算符 (难)

2.4.1 算术运算符

加法：+

减法：-

乘法：*

除法：/

注意：整数与整数相除，只保留整数部分

取模：% 取余

注意：取模结果的正负号只看被模数

正号：+

负号：-

自增：++

自减：--

原则：自增与自减

++/--在前的，就先自增/自减，后取值

++/--在后的，就先取值，后自增/自减

整个表达式的扫描，是从左往右扫描，如果后面的先计算的，那么前面的就暂时先放到“操作数栈”中

代码示例：

```

int i = 1;
i++; // i=2

int j = 1;
++j; // j=2

int a = 1;
int b = a++; // (1) 先取a的值"1"放操作数栈 (2) a再自增, a=2 (3) 再把操作数栈中的"1"赋值给b, b=1

int m = 1;
int n = ++m; // (1) m先自增, m=2 (2) 再取m的值"2"放操作数栈 (3) 再把操作数栈中的"2"赋值给n, n=2

int i = 1;
int j = i++ + ++i * i++;
/*
从左往右加载
(1) 先算i++
①取i的值"1"放操作数栈
②i再自增 i=2
(2) 再算++i
①i先自增 i=3
②再取i的值"3"放操作数栈
(3) 再算i++
①取i的值"3"放操作数栈
②i再自增 i=4
(4) 先算乘法
用操作数栈中 3 * 3 = 9, 并把9压入操作数栈
(5) 再算求和
用操作数栈中的 1 + 9 = 10
(6) 最后算赋值
j = 10
*/

```

2.4.2 赋值运算符

基本赋值运算符：=

扩展赋值运算符：+=, -=, *=, /=, %=...

注意：所有的赋值运算符的=左边一定是一个变量

扩展赋值运算符=右边的计算结果的类型如果比左边的大的话会强制类型转换，所以结果可能有风险。

扩展赋值运算符的计算：（1）赋值最后算（2）加载数据的顺序是把左边的变量的值先加载，再去与右边的表达式进行计算

```
int i = 1;
```

```
int j = 5;
j *= i++ + j++; // j = j * (i++ + j++);
/*
(1)先加载j的值“5”
(2)在计算i++
①先加载i的值“1”
②再i自增, i=2
(3)再计算j++
①先加载j的值“5”
②再j自增, j=6
(4)算 加法
i + 5 = 6
(5)算乘法
5 * 6 = 30
(6)赋值
j = 30
*/
```

2.4.3 比较运算符

大于: >

小于: <

大于等于: >=

小于等于: <=

等于: == 注意区分赋值运算符的=

不等于: !=

注意: 比较表达式的运算结果一定只有true/false

比较表达式可以作为 (1) 条件 (2) 逻辑运算符的操作数

2.4.4 逻辑运算符

逻辑运算符的操作数必须是布尔值, 结果也是布尔值

逻辑与: &

运算规则: 只有左右两边都为true, 结果才为true。

例如: true & true 结果为true

false & true 结果为false

true & false 结果为false

false & false 结果为false

逻辑或: |

运算规则: 只要左右两边有一个为true, 结果就为true。

例如：true | true 结果为true

false | true 结果为true

true | false 结果为true

false | false 结果为false

逻辑异或：^

运算规则：只有左右两边不同，结果才为true。

例如：true ^ true 结果为false

false ^ true 结果为true

true ^ false 结果为true

false ^ false 结果为false

逻辑非：!

运算规则：布尔值取反

例如：!true 为false

!false 为true

短路与：&&

运算规则：只有左右两边都为true，结果才为true。

例如：true & true 结果为true

true & false 结果为false

false & ? 结果就为false

它和逻辑与不同的是当&&左边为false时，右边就不看了。

短路或：||

运算规则：只要左右两边有一个为true，结果就为true。

例如：true | ? 结果为true

false | true 结果为true

false | false 结果为false

它和逻辑或不同的是当||左边为true时，右边就不看了。

开发中一般用短路与和短路或比较多

面试题：&& 和 &的区别？

&&当左边为false，右边不计算

&不管左边是true还是false，右边都要计算

2.4.5 条件运算符

?:

语法格式：

条件表达式 ? 结果表达式1 : 结果表达式2

运算规则：

整个表达式的结果：当条件表达式为true时，就取结果表达式1的值，否则就取结果表达式2的值

代码示例：

(1) boolean类型

```
boolean marry = true;  
System.out.println(marry? "已婚" : "未婚");
```

(2) 求最值

```
int i = 3;  
int j = 5;  
int max = i >= j ? i : j;  
//当i >= j时, max就赋值为i的值, 否则就赋值为j的值
```

2.4.6 位运算符

左移: <<

运算规则: 左移几位就相当于乘以2的几次方

右移: >>

运算规则: 右移几位就相当于除以2的几次方

无符号右移: >>>

运算规则: 往右移动后, 左边空出来的位直接补0, 不看符号位

按位与: &

运算规则:

1 & 1 结果为1

1 & 0 结果为0

0 & 1 结果为0

0 & 0 结果为0

按位或: |

运算规则:

1 | 1 结果为1

1 | 0 结果为1

0 | 1 结果为1

0 & 0 结果为0

按位异或: ^

运算规则:

1 ^ 1 结果为0

1 ^ 0 结果为1

$0 \wedge 1$ 结果为1

$0 \wedge 0$ 结果为0

按位取反：~


运算规则：~0就是1

~1就是0

如何区分&，|，^是逻辑运算符还是位运算符？

如果操作数是boolean类型，就是逻辑运算符，如果操作数是整数，那么就位运算符。

2.4.7 运算符优先级

. ()	<div>高</div>  <div>低</div> <div>学的技</div>
++ -- ~ !	
* / %	
+ -	
<< >> >>>	
< > <= >= instanceof	
== !=	
&	
^	
&&	
? :	
= *= /= %=	
+= -= <<= >>=	
>>>= &= ^= =	

提示说明：

- (1) 表达式不要太复杂
- (2) 先算的使用()

2.4.8 运算符操作数类型说明

1、算术运算符

数字和单个字符可以使用算术运算符。

其中+，当用于字符串时，表示拼接。

2、赋值运算符

右边的常量值、表达式的值、变量的值的类型必须与左边的变量一致或兼容（可以实现自动类型转换）或使用强制类型转换可以成功。

3、比较运算符

其他的比较运算符都是只能用于8种基本数据类型。

其中的==和!=可以用于引用数据类型的比较，用于比较对象的地址。（后面讲）

```
int i = 10;
int j = 10;
System.out.println(i==j);//true

char c1 = '帅';
char c2 = '帅';
System.out.println(c1 == c2);//true
```

4、逻辑运算符

逻辑运算符的操作数必须是boolean值

5、条件运算符

?前面必须是条件，必须是boolean值

结果表达式1和结果表达式2要保持类型一致或兼容

6、位运算符

一般用于整数系列

以上运算符都是针对基本数据类型设计的。

能够用于引用数据类型只有基本的赋值运算符=，和比较运算符中的==和!=。其他运算符都不能用于引用数据类型。

其中字符串类型还有一个+，表示拼接。

第三章 流程控制语句结构

流程控制语句结构分为：

- 1、顺序结构：从上到下依次执行
- 2、分支结构：多个分支选择其中一个分支执行
- 3、循环结构：重复执行某些代码

3.1 顺序结构

执行过程：从上到下顺序执行

3.1.1 输出语句

- 1、System.out.print(输出内容); #输出内容后不换行
- 2、System.out.println(输出内容); #输出内容后换行

```
#输出常量
System.out.print(1);
System.out.print('尚');
System.out.print(44.4);
System.out.print(true);
System.out.print("尚硅谷");

#输出变量
int a = 1;
char c = '尚';
double d = 44.4;
boolean b = true;
String school = "尚硅谷";
System.out.print(a);
System.out.print(c);
System.out.print(d);
System.out.print(b);
System.out.print(school);

#输出拼接结果
System.out.print("a = " + a);
System.out.print("c = " + c);
System.out.print("d = " + d);
System.out.print("b = " + b);
System.out.print("school = " + school);
```

3.1.2 输入语句

键盘输入代码的三个步骤：

1、准备Scanner类型的变量

2、提示输入xx

3、接收输入内容

示例代码：

```
//1、准备Scanner类型的变量
java.util.Scanner input = new java.util.Scanner(System.in); //System.in默认代表键盘输入

//2、提示输入xx
System.out.print("请输入一个整数：");

//3、接收输入内容
int num = input.nextInt();

//列出各种数据类型的输入
int num = input.nextInt();
long bigNum = input.nextLong();
double d = input.nextDouble();
boolean b = input.nextBoolean();
String s = input.next();
char c = input.next().charAt(0); //先按照字符串接收，然后再取字符串的第一个字符（下标为0）
```

3.2 分支结构

分支结构：根据条件选择性的执行某些代码

分为：

1、条件判断：if...else系列

2、选择结构：switch...case系列

3.2.1 条件判断

1、单分支结构

语法格式：

```
if(条件表达式){
    当条件表达式成立(true)时需要执行的语句块;
}
```

执行过程：

条件成立，就执行{}其中的语句块，不成立就不执行。

注意：

- (1) if(条件表达式)中的条件表达式的结果必须是boolean类型
- (2) 当{}中的语句只有一个语句（简单的语句，也可以是一个复合语句）时，可以省略{}，但是我们不建议省略

```
//省略{}的情况
if(score<0 || score>100)
    System.out.println("输入有误!"); //简单的语句
else
    //复合语句
    if(score==100){
        System.out.println("满分");
    }else if(score>=80){
        System.out.println("优秀");
    }else if(score>=60){
        System.out.println("及格");
    }else{
        System.out.println("不及格");
    }
}
```

示例代码：

```
int year = 2019;
int days = 28;
if(year%4==0 && year%100!=0 || year%400==0){
    days= 29;
}
```

2、双分支结构

语法格式：

```
if(条件表达式){
    当条件表达式成立(true)时需要执行的语句块1;
}else{
    当条件表达式不成立(false)时需要执行的语句块2;
}
```

执行过程：

当条件表达式成立(true)时执行语句块1，否则执行语句块2

注意：

- (1) if(条件表达式)中的条件表达式的结果必须是boolean类型

(2) 当{}中的语句只有一个语句（简单的语句，也可以是一个复合语句）时，可以省略{}，但是我们不建议

示例代码：

```
int num = 10;
if(num%2==0){
    System.out.println(num + "是偶数");
}else{
    System.out.println(num + "是奇数");
}
```

3、多分支结构

语法格式：

```
if(条件表达式1){
    当条件表达式1成立的时候，执行的语句块1；
}else if(条件表达式2){
    当条件表达式1不成立，
    条件表达式2成立的时候，执行的语句块2；
}else if(条件表达式3){
    当条件表达式1不成立，
    条件表达式2也不成立，
    条件表达式3成立的时候，执行的语句块3；
}
...
【else{
    当以上所有的条件表达式都不成立，需要执行的语句块n+1；
}】
```

执行过程：

- (1) 多个条件顺序往下判断，如果上面有一个条件成立了，下面的条件就不看了
- (2) 多个分支也只会执行其中的一个

注意：

- (1) 每一个条件表达式都必须是boolean值
- (2) 当{}中只有一个语句时，也可以省略{}，但不建议省略
- (3) 当多个条件是“互斥”关系（没有重叠部分），顺序可以随意；

当多个条件是“包含”关系（有重叠部分），顺序不能随意，小的在上，大的在下面

示例代码：

```
int score = 78;
if(score==100){
    System.out.println("满分");
}else if(score>=80){
    System.out.println("优秀");
}else if(score>=60){
    System.out.println("及格");
}else{
    System.out.println("不及格");
}
```

4、嵌套

执行过程：

当嵌套在if中，就是当外面的if成立时，才会看里面的条件判断；

当嵌套在else中，就当外面的else满足时，才会看里面的条件判断；

3.2.2 选择结构

语法格式：

```
switch(表达式){
    case 常量值1:
        语句块1;
        【break;】
    case 常量值2:
        语句块2;
        【break;】
    ...
    【default:
        语句块n+1;
        【break;】
    】
}
```

执行过程：

(1) 入口

①当switch(表达式)的值与case后面的某个常量值匹配，就从这个case进入；

②当switch(表达式)的值与case后面的所有常量值都不匹配，寻找default分支进入；

(2) 一旦从“入口”进入switch，就会顺序往下执行，直到遇到“出口”

(3) 出口

①自然出口：遇到了switch的结束}

②中断出口：遇到了break等

注意：

(1) switch(表达式)的值的类型，只能是：4种基本数据类型 (byte,short,int,char) ， 两种引用数据类型 (枚举、String)

(2) case后面必须是常量值， 而且不能重复

示例代码：

```
int month = 4;
switch(month){
    case 3:
    case 4:
    case 5:
        System.out.println("春季");
        break;
    case 6:
    case 7:
    case 8:
        System.out.println("夏季");
        break;
    case 9:
    case 10:
    case 11:
        System.out.println("秋季");
        break;
    case 12:
    case 1:
    case 2:
        System.out.println("冬季");
        break;
    default:
        System.out.println("输入有误! ");
}
```

3.3 循环结构

循环结构：

“重复”执行某些代码

循环结构的分类：

- 1、for循环
- 2、while循环
- 3、do...while循环

3.3.1 for循环

语法格式：

```
for(;;){  
    循环体语句块;  
    if(条件表达式){  
        break;  
    }  
}  
for(初始化表达式; 循环条件; 迭代表达式){  
    循环体语句块; (需要重复执行的代码)  
}
```

执行过程：

- (1) 初始化表达式;
- (2) 判断循环条件;
- (3) 如果循环条件成立，先执行循环体语句块；然后执行迭代表达式，再回到 (2) ...
- (4) 如果循环条件不成立，会结束for；

或者在当前循环中遇到break语句，也会结束当前for循环;

注意：

- (1) for(;;)中的两个；是不能多也不能少
- (2) 循环条件必须是boolean类型

示例代码：

```
//遍历1-100之间的偶数  
for(int i=1; i<=100; i++){//每次循环的步幅是1  
    if(i%2==0){  
        System.out.println(i);  
    }  
}  
  
//遍历1-100之间的偶数  
for(int i=2; i<=100; i+=2){//每次循环的步幅是2  
    System.out.println(i);  
}
```

3.3.2 while循环

语法格式：

```
while(循环条件){  
    循环体语句块;  
}
```

经典的形式：

```
while(true){  
    循环体语句块;  
    if(条件表达式){  
        break;  
    }  
}
```

执行过程：

- (1) 先判断循环条件
- (2) 如果循环条件成立，就执行循环体语句块；然后回到 (1)
- (3) 如果循环条件不成立，就结束while循环；

如果在循环体语句块中，遇到break，也会结束while循环；

注意：

- (1) while(循环条件)中循环条件必须是boolean类型

示例代码：

```
//遍历1-100之间的偶数  
int num = 2;  
while(num<=100){  
    System.out.println(num);  
    num+=2;  
}
```

3.3.3 do...while循环

语法格式：

```
do{
    循环体语句块;
}while(循环条件);
```

执行过程：

- (1) 先执行一次循环体语句块；
- (2) 判断循环条件
- (3) 如果循环条件成立，再次执行循环体语句块；然后回到 (2) ...
- (4) 如果循环条件不成立，就结束do...while循环；

如果在循环体语句块中，遇到break，也会结束do...while循环；

注意：

- (1) while(循环条件)中循环条件必须是boolean类型
- (2) do{}while();最后有一个分号
- (3) do...while结构的循环体语句是至少会执行一次，这个和for和while是不一样的

示例代码：

```
//从键盘输入整数，统计正数、负数的个数，输入0结束
java.util.Scanner input = new java.util.Scanner(System.in);

int num;
int positive = 0;
int negative = 0;
do{
    System.out.print("请输入整数（0结束）：");
    num = input.nextInt();

    if(num > 0){
        positive++;
    }else if(num < 0){
        negativie++;
    }
}while(num!=0);

System.out.println("正数的个数：" + positive);
System.out.println("负数的个数：" + negativie);
```

3.3.4 三种循环的选择

原则：三种循环之间是可以互相转换的，都能实现循环的功能

建议（习惯上）：当我们次数比较明显的时候，或者说从几循环到几的时候，一般先考虑for；

当循环体语句块至少要执行一次的时候，一般先考虑do...while；

当循环条件比较明显，但是次数不明显，循环体语句块也不是至少执行一次，那么可以考虑while结构；

三种循环结构都具有四要素：

- (1) 循环变量的初始化表达式
- (2) 循环条件
- (3) 循环变量的修改的迭代表达式
- (4) 循环体语句块

3.3.5 跳转语句

1、break

用于：

- (1) switch结构

作用：结束switch结构

- (2) 循环结构

作用：结束当前循环

2、continue

用于：

只能用于循环结构

作用：提前结束本次循环，继续下一次循环

3、return（后面讲）

第四章 数组

4.1 数组的相关概念和名词（了解）

1、数组(array)：

一组具有相同数据类型的数据的按照一定顺序排列的集合。

把有限的几个相同类型的变量使用一个名称来进行统一管理。

2、数组名：

(1) 这个数组名，代表的是一组数

(2) 这个数组名中存储的整个数组的“首地址”

3、下标(index):

我们使用编号、索引、下标来区别表示一组数当中某一个。

范围：[0,数组长度-1]

例如：for(int i = 0; i<arr.length; i++){

4、元素(element):

这一组中的的每一个数据都是元素。

如何表示数组元素？ 数组名[下标]

5、数组的长度(length)

数组中元素的总个数。

如何获取数组长度？ 数组名.length

注意：名称是为了沟通的方便，概念不用一字不落背下来

4.2 数组的相关语法

4.2.1 数组的声明

语法格式：

//推荐

元素的数据类型[] 数组名；

//也对，但是不推荐

元素的数据类型 数组名[]；

示例：

//要存储一组整数

```
int[] array;
```

//要存储一组单字符

```
char[] array;
```

//要存储一组字符串

```
String[] array;
```

4.2.2 数组的初始化

初始化的目的：（1）确定数组的长度（2）为元素赋值

两种初始化方式：

1、动态初始化

语法格式：

```
//指定数组长度
数组名 = new 元素的数据类型[长度];

//为元素赋值
数组名[下标] = 值; //这个值可以是常量值，也可以是个表达式的计算结果，也可以是键盘输入的

//如果每个元素的赋值比较有规律，通常使用for循环赋值
for(int i=0; i<长度; i++){
    数组名[下标] = 值;
}
```

问：如果只指定数组长度，没有为元素手动赋值，那么元素有值吗？

有默认值

(1) 基本数据类型

byte,short,int,long: 0

float,double: 0.0

char: \u0000

boolean: false

(2) 引用数据类型

统统都是null

2、静态初始化

语法格式：

```
数组名 = new 元素的数据类型[] {值列表};

//int[] arr = new int[5]{1,2,3,4,5}; //错误的

//更简洁
//当声明与静态初始化一起完成时，可以简化
元素的数据类型[] 数组名 = {值列表};
```

适用场合：

当数组的元素是已知的有限个时，可以使用静态初始化。

示例代码：

```
String[] weeks =
{"monday", "tuesday", "wednesday", "thursday", "friday", "saturday", "sunday"};

int[] daysOfMonths = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

char[] letters =
{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'};
```

4.2.3 数组的遍历

for循环遍历数组：

```
for(int i=0; i<数组名.length; i++){
    //或赋值
    数组名[i] = 值;
    //或显示
    System.out.println(数组名[i]);
    //或其他操作
    //例如：判断是否是偶数
    if(数组名[i]%2==0){
        //...
    }
}
```

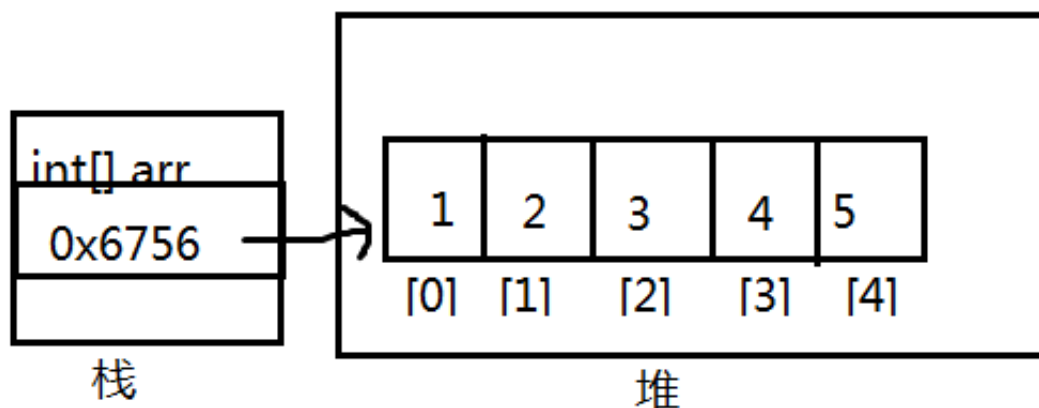
4.2.4 数组的内存分析

元素是基本数据类型的一维数组内存分析：

```
int[] arr = {1,2,3,4,5};
```



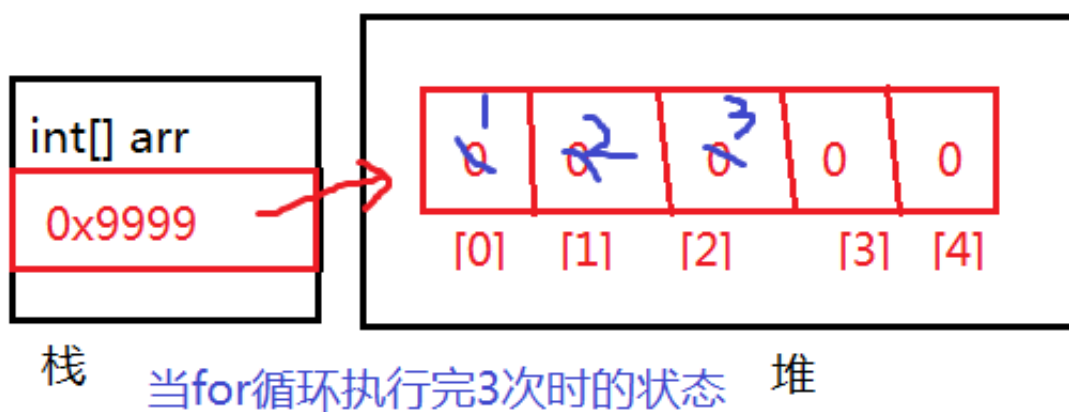
```
int[] arr = {1,2,3,4,5};
```



```
int[] arr = new int[5];  
for(int i=0; i<arr.length; i++){  
    arr[i] = i+1;  
}
```

```
int[] arr = new int[5];
```

```
for(int i=0; i<arr.length; i++){  
    arr[i] = i+1;  
}
```



4.3 数组的相关算法

4.3.1 数组找最值

1、数组中找最值

思路：

- (1) 先假设第一个元素最大/最小
- (2) 然后用max/min与后面的元素一一比较

示例代码：

```
int[] arr = {4,5,6,1,9};  
//找最大值  
int max = arr[0];  
for(int i=1; i<arr.length; i++){  
    if(arr[i] > max){  
        max = arr[i];  
    }  
}
```

2、数组中找最值及其下标

情况一：找最值及其第一次出现的下标

思路：

- (1) 先假设第一个元素最大/最小
- (2) 然后用max/min与后面的元素一一比较

示例代码：

```
int[] arr = {4,5,6,1,9};  
//找最大值  
int max = arr[0];  
int index = 0;  
for(int i=1; i<arr.length; i++){  
    if(arr[i] > max){  
        max = arr[i];  
        index = i;  
    }  
}
```

或

```
int[] arr = {4,5,6,1,9};  
//找最大值  
int maxIndex = 0;  
for(int i=1; i<arr.length; i++){  
    if(arr[i] > arr[maxIndex]){  
        maxIndex = i;  
    }  
}  
System.out.println("最大值: " + arr[maxIndex]);
```

情况二：找最值及其所有最值的下标（即可能最大值重复）

思路：

（1）先找最大值

①假设第一个元素最大

②用max与后面的元素一一比较

（2）遍历数组，看哪些元素和最大值是一样的

示例代码：

```
int[] arr = {4,5,6,1,9};
//找最大值
int max = arr[0];
for(int i=1; i<arr.length; i++){
    if(arr[i] > max){
        max = arr[i];
    }
}

//遍历数组，看哪些元素和最大值是一样的
for(int i=0; i<arr.length; i++){
    if(max == arr[i]){
        System.out.print(i+"\t");
    }
}
```

4.3.2 数组统计：求总和、均值、统计偶数个数等

思路：遍历数组，挨个的累加，判断每一个元素

示例代码：

```
int[] arr = {4,5,6,1,9};
//求总和、均值
int sum = 0; //因为0加上任何数都不影响结果
for(int i=0; i<arr.length; i++){
    sum += arr[i];
}
double avg = (double)sum/arr.length;
```

示例代码2：

```
int[] arr = {4,5,6,1,9};

//求总乘积
long result = 1;//因为1乘以任何数都不影响结果
for(int i=0; i<arr.length; i++){
    result *= arr[i];
}
```

示例代码3:

```
int[] arr = {4,5,6,1,9};
//统计偶数个数
int even = 0;
for(int i=0; i<arr.length; i++){
    if(arr[i]%2==0){
        even++;
    }
}
```

4.3.3 反转

方法有两种:

- 1、借助一个新数组
- 2、首尾对应位置交换

第一种方式示例代码:

```
int[] arr = {1,2,3,4,5,6,7,8,9};

//(1)先创建一个新数组
int[] newArr = new int[arr.length];

//(2)复制元素
int len = arr.length;
for(int i=0; i<newArr.length; i++){
    newArr[i] = arr[len - 1 - i];
}

//(3)舍弃旧的, 让arr指向新数组
arr = newArr;//这里把新数组的首地址赋值给了arr

//(4)遍历显示
for(int i=0; i<arr.length; i++){
    System.out.println(arr[i]);
}
```

第二种方式示例代码：

```
int[] arr = {1,2,3,4,5,6,7,8,9};

//(1)计算要交换的次数:  次数 = arr.length/2
//(2)首尾交换
for(int i=0; i<arr.length/2; i++){//循环的次数就是交换的次数
    //首 与 尾交换
    int temp = arr[i];
    arr[i] = arr[arr.length-1-i];
    arr[arr.length-1-i] = temp;
}

// (3) 遍历显示
for(int i=0; i<arr.length; i++){
    System.out.println(arr[i]);
}
```

4.3.4 复制

应用场景：

- 1、扩容
- 2、备份
- 3、截取

示例代码：扩容

```
int[] arr = {1,2,3,4,5,6,7,8,9};

//如果要把arr数组扩容，增加1个位置
//(1)先创建一个新数组，它的长度 = 旧数组的长度+1
int[] newArr = new int[arr.length + 1];

//(2)复制元素
//注意: i<arr.length  因位arr比newArr短，避免下标越界
for(int i=0; i<arr.length; i++){
    newArr[i] = arr[i];
}

//(3)把新元素添加到newArr的最后
newArr[newArr.length-1] = 新值;

//(4)如果下面继续使用arr，可以让arr指向新数组
arr = newArr;

//(4)遍历显示
for(int i=0; i<arr.length; i++){
```

```
        System.out.println(arr[i]);
    }
}
```

示例代码：备份

```
int[] arr = {1,2,3,4,5,6,7,8,9};

//1、创建一个长度和原来的数组一样的新数组
int[] newArr = new int[arr.length];

//2、复制元素
for(int i=0; i<arr.length; i++){
    newArr[i] = arr[i];
}

//3、遍历显示
for(int i=0; i<arr.length; i++){
    System.out.println(arr[i]);
}
```

示例代码：截取

```
int[] arr = {1,2,3,4,5,6,7,8,9};

int start = 2;
int end = 5;

//1、创建一个新数组，新数组的长度 = end-start + 1;
int[] newArr = new int[end-start+1];

//2、赋值元素
for(int i=0; i<newArr.length; i++){
    newArr[i] = arr[start + i];
}

//3、遍历显示
for(int i=0; i<newArr.length; i++){
    System.out.println(newArr[i]);
}
```

4.3.5 查找

查找分为两种：

1、顺序查找：挨个看

对数组没要求

2、二分查找：对折对折再对折

对数组有要求，元素必须有大小顺序的

顺序查找示例代码：

```
int[] arr = {4,5,6,1,9};
int value = 1;
int index = -1;

for(int i=0; i<arr.length; i++){
    if(arr[i] == value){
        index = i;
        break;
    }
}

if(index== -1){
    System.out.println(value + "不存在");
}else{
    System.out.println(value + "的下标是" + index);
}
```

二分查找示例代码：

```
/*
2、编写代码，使用二分查找法在数组中查找 int value = 2;是否存在，如果存在显示下标，不存在显示不存在。
已知数组: int[] arr = {1,2,3,4,5,6,7,8,9,10};
*/
class Exam2{
    public static void main(String[] args){
        int[] arr = {1,2,3,4,5,6,7,8,9}; //数组是有序的
        int value = 2;

        int index = -1;
        int left = 0;
        int right = arr.length - 1;
        int mid = (left + right)/2;
        while(left<=right){
            //找到结束
            if(value == arr[mid]){
                index = mid;
                break;
            } //没找到
            else if(value > arr[mid]){ //往右继续查找
                //移动左边界，使得mid往右移动
                left = mid + 1;
            } else if(value < arr[mid]){ //往左边继续查找
                right = mid - 1;
            }
        }
    }
}
```

```

        mid = (left + right)/2;
    }

    if(index== -1){
        System.out.println(value + "不存在");
    }else{
        System.out.println(value + "的下标是" + index);
    }

}
}

```

使用for

```

class Exam2{
    public static void main(String[] args){
        int[] arr = {1,2,3,4,5,6,7,8,9}; //数组是有序的
        int value = 2;

        int index = -1;

        for(int left=0, right=arr.length-1, mid = (left+right)/2; left<=right;
mid = (left + right)/2){
            //找到结束
            if(value == arr[mid]){
                index = mid;
                break;
            } //没找到
            else if(value > arr[mid]){ //往右继续查找
                //移动左边界, 使得mid往右移动
                left = mid + 1;
            } else if(value < arr[mid]){ //往左边继续查找
                right = mid - 1;
            }
        }

        if(index== -1){
            System.out.println(value + "不存在");
        }else{
            System.out.println(value + "的下标是" + index);
        }

    }
}

```


4.3.6 排序

数组的排序算法有千万种，我们只讲了两中：

- 1、冒泡排序
- 2、简单的直接排序

示例代码：冒泡：从小到大，从左到右两两比较

```
int[] arr = {5,4,6,3,1};
for(int i=1; i<arr.length; i++){//外循环的次数 = 轮数 = 数组的长度-1
    /*
    第1轮, i=1,从左到右两两比较, arr[0]与arr[1]。。。。arr[3]与arr[4]
    第2轮, i=2,从左到右两两比较, arr[0]与arr[1]。。。。arr[2]与arr[3]
    ...
        arr[j]与arr[j+1]比较
    找两个关键点：（1）j的起始值：0（2）找j的终止值，依次是3,2,1,0，得出j<arr.length-i
    */
    for(int j=0; j<arr.length-i; j++){
        //两两比较
        //从小到大，说明前面的比后面的大，就交换
        if(arr[j] > arr[j+1]){
            int temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
        }
    }
}
```

示例代码：从大到小，从右到左

```
char[] arr = {'h','e','l','l','o','j','a','v','a'};
for(int i=1; i<arr.length; i++){//外循环的次数 = 轮数 = 数组的长度-1
    /*
    第1轮, i=1, 从右到左两两比较, arr[8]与arr[7], arr[7]与arr[6]....arr[1]与arr[0]
    第2轮, i=2, 从右到左两两比较, arr[8]与arr[7], arr[7]与arr[6]....arr[2]与arr[1]
    ...
    第8轮, i=8, 从右到左两两比较, arr[8]与arr[7]
        arr[j]与arr[j-1]
    找两个关键点：（1）j的起始值：8（2）找j的终止值，依次是1,2,3,...8，得出j>=i
    */
    for(int j=8; j>=i; j--){
        //从大到小，后面的元素 > 前面的元素，就交换
        if(arr[j]>arr[j-1]){
            int temp = arr[j];
            arr[j] = arr[j-1];
            arr[j-1] = temp;
        }
    }
}
```

```
}  
}
```

示例代码：简单的直接选择排序

```
int[] arr = {3,2,6,1,8};  
  
for(int i=1; i<arr.length; i++){//外循环的次数 = 轮数 = 数组的长度-1  
    // (1) 找出本轮未排序元素中的最大值  
    /*  
    未排序元素：  
    第1轮：i=1,未排序, [0,4]  
    第2轮：i=2,未排序, [1,4]  
    ...  
  
    每一轮未排序元素的起始下标：0,1,2,3, 正好是i-1的  
    未排序的后面的元素依次：  
    第1轮：[1,4]   j=1,2,3,4  
    第2轮：[2,4]   j=2,3,4  
    第3轮：[3,4]   j=3,4  
    第4轮：[4,4]   j=4  
    j的起点是i, 终点都是4  
    */  
    int max = arr[i-1];  
    int index = i-1;  
    for(int j=i; j<arr.length; j++){  
        if(arr[j] > max){  
            max = arr[j];  
            index = j;  
        }  
    }  
  
    // (2) 如果这个最大值没有在它应该在的位置, 就与这个位置的元素交换  
    /*  
    第1轮, 最大值应该在[0]  
    第2轮, 最大值应该在[1]  
    第3轮, 最大值应该在[2]  
    第4轮, 最大值应该在[3]  
    正好是i-1的值  
    */  
    if(index != i-1){  
        //交换arr[i-1]与arr[index]  
        int temp = arr[i-1];  
        arr[i-1] = arr[index];  
        arr[index] = temp;  
    }  
}
```

```
//显示结果
for(int i=0; i<arr.length; i++){
    System.out.print(arr[i]);
}
```

4.4 二维数组

二维数组的标记：[][]

4.4.1 相关的表示方式

(1) 二维数组的长度/行数：

二维数组名.length

(2) 二维数组的其中一行：

二维数组名[行下标]

行下标的范围：[0, 二维数组名.length-1]

(3) 每一行的列数：

二维数组名[行下标].length

因为二维数组的每一行是一个一维数组

(4) 每一个元素

二维数组名[行下标][列下标]

4.4.2 二维数组的声明和初始化

1、二维数组的声明

```
//推荐
元素的数据类型[] [] 二维数组的名称;
```

```
//不推荐
元素的数据类型 二维数组名[][];
```

```
//不推荐
元素的数据类型[] 二维数组名[];
```

面试：

```
int[] x, y[];
//x是一维数组, y是二维数组
```

2、二维数组的初始化

(1) 静态初始化

```
二维数组名 = new 元素的数据类型[][]{  
    {第一行的值列表},  
    {第二行的值列表},  
    ...  
    {第n行的值列表}  
};
```

//如果声明与静态初始化一起完成

```
元素的数据类型[][] 二维数组的名称 = {  
    {第一行的值列表},  
    {第二行的值列表},  
    ...  
    {第n行的值列表}  
};
```

(2) 动态初始化（不规则：每一行的列数可能不一样）

// (1) 先确定总行数

```
二维数组名 = new 元素的数据类型[总行数][];
```

// (2) 再确定每一行的列数

```
二维数组名[行下标] = new 元素的数据类型[该行的总列数];
```

// (3) 再为元素赋值

```
二维数组名[行下标][列下标] = 值;
```

(3) 动态初始化（规则：每一行的列数是相同的）

// (1) 确定行数和列数

```
二维数组名 = new 元素的数据类型[总行数][每一行的列数];
```

// (2) 再为元素赋值

```
二维数组名[行下标][列下标] = 值;
```

4.4.3 二维数组的遍历

```
for(int i=0; i<二维数组名.length; i++){
    for(int j=0; j<二维数组名[i].length; j++){
        System.out.print(二维数组名[i][j]);
    }
    System.out.println();
}
```

第五章 面向对象基础

5.1 类与对象

1、类：一类具有相同特性的事物的抽象描述。

对象：类的一个个体，实例，具体的存在。

类是对象的设计模板。

2、如何声明类？

```
【修饰符】 class 类名{
    成员列表：属性、方法、构造器、代码块、内部类
}
```

3、如何创建对象？

```
new 类名(); //匿名对象

类名 对象名 = new 类名(); //有名对象
```

5.2 类的成员之一：属性

1、如何声明属性？

```
【修饰符】 class 类名{
    【修饰符】 数据类型 属性名; //属性有默认值
    【修饰符】 数据类型 属性名 = 值; //属性有初始值
}
```

说明：属性的类型可以是Java的任意类型，包括基本数据类型、引用数据类型（类、接口、数组等）

总结：Java的数据类型

(1) 基本数据类型

byte,short,int,long,float,double,char,boolean

(2) 引用数据类型

①类：

例如：String、Student、Circle、System、Scanner、Math...

②接口：后面讲

③数组：

例如：int[], String[], char[], int[][]

```
int[] arr = new int[5];
```

这里把int[]看成数组类型，是一种引用数据类型，右边赋值的是一个数组的对象

元素的数据类型：int

数组的数据类型：int[]

2、如何为属性赋值？

(1) 在声明属性时显式赋值

```
【修饰符】 class 类名{  
    【修饰符】 数据类型 属性名 = 值; //属性有初始值  
}
```

代码示例：

```
class Student{  
    String name;  
    char gender = '男';//显式赋值  
}  
  
class TestStudent{  
    public static void main(String[] args){  
        Student s1 = new Student();  
        System.out.println("姓名：" + s1.name); //null  
        System.out.println("性别：" + s1.gender); //男  
  
        s1.name = "小薇"; //修改属性的默认值  
        s1.gender = '女'; //修改属性的初始值  
        System.out.println("姓名：" + s1.name); //小薇  
        System.out.println("性别：" + s1.gender); //女  
  
        Student s2 = new Student();  
        System.out.println("姓名：" + s2.name); //null  
        System.out.println("性别：" + s2.gender); //男  
    }  
}
```

(2) 创建对象之后赋值

```
【修饰符】 class 类名{
    【修饰符】 数据类型 属性名; //属性有默认值
}

//创建对象
类名 对象名 = new 类名();

//为对象的属性赋值
对象名.属性名 = 值;
```

3、如何访问属性的值？

(1) 在本类的方法中访问

示例代码：

```
class Circle{
    double radius;

    double getArea(){
        return 3.14 * radius * radius; //直接访问
    }
}
```

(2) 在其他类的方法中访问

```
class Circle{
    double radius;
}

class TestCircle{
    public static void main(String[] args){
        Circle c1 = new Circle();
        double area = 3.14 * c1.radius * c1.radius; //对象名.属性名
    }
}
```

4、属性的特点

(1) 属性有默认值

基本数据类型：

byte, short, int, long: 0

float, double: 0.0

char: \u0000

boolean: false

引用数据类型:

null

(2) 每一个对象的属性是独立, 互不干扰

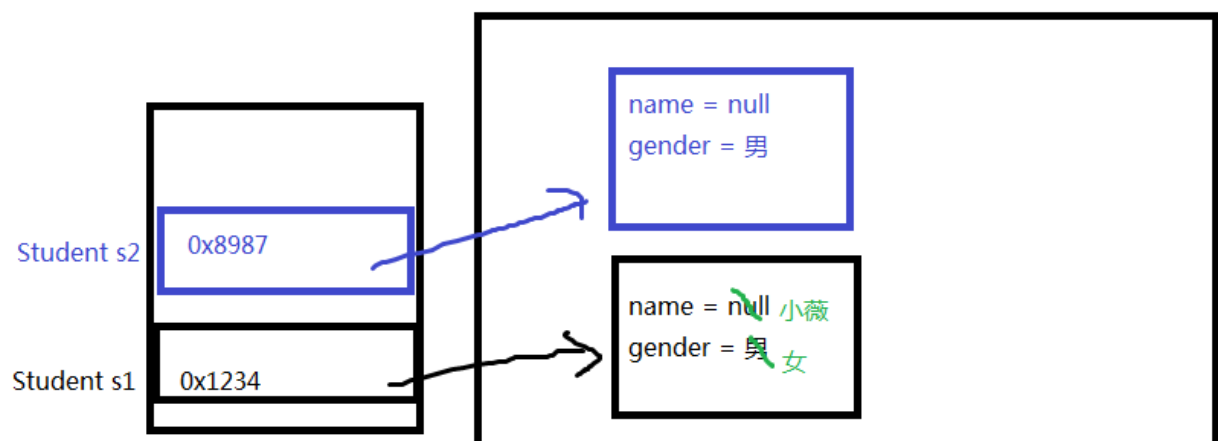
5、对象属性的内存图

```
class Student{
    String name;
    char gender = '男';//显式赋值
}

class TestStudent{
    public static void main(String[] args){
        Student s1 = new Student();
        System.out.println("姓名: " + s1.name);//null
        System.out.println("性别: " + s1.gender);//男

        s1.name = "小薇";
        s1.gender = '女';
        System.out.println("姓名: " + s1.name);//小薇
        System.out.println("性别: " + s1.gender);//女

        Student s2 = new Student();
        System.out.println("姓名: " + s2.name);//null
        System.out.println("性别: " + s2.gender);//男
    }
}
```



5.4 类的成员之二：方法

5.4.1 方法的概念

方法（method）：代表一个独立的可复用的功能

目的/好处：

- (1) 复用
- (2) 简化代码

5.4.2 方法的语法

1、方法的声明格式：

```
【修饰符】 class 类名{  
    【修饰符】 返回值类型 方法名(【形参列表】){  
        方法体：实现功能的代码  
    }  
}
```

说明：

- (1) 【修饰符】：待讲
- (2) 返回值类型：

①void：表示无返回值

②非void：所有的Java数据类型都可以

- (3) 方法名：能很好的体现方法的功能

命名的规范：①见名知意②从第二个单词开始首字母大写

- (4) 【形参列表】：

在完成这个方法的功能时，需要一些数据，这些数据要由“调用者”来决定，那我们就可以设计形参。

语法格式：

()：无参，空参

(数据类型 形参名)：一个形参

(数据类型1 形参名1,, 数据类型n 形参名n)：n个形参

- (5) 方法体：实现方法的功能，最好一个方法就完成一个独立的功能。

2、方法的调用格式：

```
//本类同级别方法调用：直接调用
【变量 = 】 方法名(【实参列表】);
```

```
//在其他类的方法中调用
【变量 = 】 对象名.方法名(【实参列表】);
```

(1) 是否传实参

看被调用的方法是否有形参

(2) 是否接收返回值

看被调用的方法是否是void，如果是void，就不需要也不能接收，如果不是void，就可以接收。

3、方法的声明与调用的代码示例

(1) 无参无返回值方法

```
//本类
class Circle{
    double radius;
    void printInfo(){
        System.out.println("半径: " + radius);
    }

    void test(){
        printInfo();//本类中调用无参无返回值方法
    }
}
```

```
//其他类
class Circle{
    double radius;
    void printInfo(){
        System.out.println("半径: " + radius);
    }
}

class TestCircle{
    public static void main(String[] args){
        Circle c1 = new Circle();
        c1.printInfo(); //其他类中调用无参无返回值方法
    }
}
```

(2) 无参有返回值方法

```
//本类
class Circle{
    double radius;

    double getArea(){
        return 3.14 * radius * radius();
    }

    void printInfo(){
        // System.out.println("半径: " + radius + ", 面积: " + getArea());//本类中调用无参有返回值
        double area = getArea();//本类中调用无参有返回值
        System.out.println("半径: " + radius + ", 面积: " + area);
    }
}
```

```
//其他类
class Circle{
    double radius;

    double getArea(){
        return 3.14 * radius * radius();
    }
}

class TestCircle{
    public static void main(String[] args){
        Circle c1 = new Circle();
        double area = c1.getArea();
        System.out.println("面积: " + area);
        //或
        System.out.println("面积: " + c1.getArea());
    }
}
```

(3) 有参无返回值方法

```
//本类
class GraphicTools{
    void printRectangle(int line, int column, char sign){
        for(int i=1; i<=line; i++){
            for(int j=1; j<=column; j++){
                Sytem.out.print(sign);
            }
        }
    }
}
```

```

        System.out.println();
    }
}

void test(){
    printRectangle(5,10,'%');//本类中调用有参无返回值方法
}
}

```

```

//其他类
class GraphicTools{
    void printRectangle(int line, int column, char sign){
        for(int i=1; i<=line; i++){
            for(int j=1; j<=column; j++){
                Sytem.out.print(sign);
            }
            System.out.println();
        }
    }
}

class Test{
    public static void main(String[] args){
        GraphicTools tools = new GraphicTools();
        tools.printRectangle(5,10,'%');
    }
}

```

(4) 有参有返回值方法

```

//本类
class MyMath{
    int sum(int a,int b){
        return a+b;
    }

    void print(){
        int x = 4;
        int y = 7;
        System.out.println(x + "+" + y + "=" + sum(x,y));//本类中调用有参有返回值的
方法
    }
}

```

```

//其他类
class MyMath{
    int sum(int a,int b){

```

```

        return a+b;
    }
}
class Test{
    public static void main(String[] args){
        MyMath my = new MyMath();
        int x = 4;
        int y = 7;

        System.out.println(x + "+" + y + "=" + my.sum(x,y));
    }
}

```

4、方法声明与调用的原则

(1) 方法必须先声明后调用

如果调用方法时，如果方法名写错或调用一个不存在的方法，编译会报错

(2) 方法声明的位置必须在类中方法外

正确示例：

```

类{
    方法1(){

    }
    方法2(){

    }
}

```

错误示例：

```

类{
    方法1(){
        方法2(){ //错误
        }
    }
}

```

(3) 方法的调用的位置有要求

正确示例：

```
类{
    方法1(){
        调用方法
    }
}
```

错误示例：

```
类{
    方法1(){

    }

    调用方法 //错误位置
}
```

(4) 方法的调用格式要与方法的声明格式对应

①是否要加“对象.”：看是否在本类中，还是其他类中

②是否要接收返回值：看被调用方法是否是void

③是否要传实参：看被调用方法是有形参列表

5.4.3 方法的重载Overload

概念：在同一个类中，出现了两个或多个的方法，它们的方法名称相同，形参列表不同，这样的形式称为方法的重载。和返回值类型无关。

示例代码：

```
//求两个整数的最大值
public int max(int a,int b){
    return a>b?a:b;
}

//求三个整数的最大值
public int max(int a, int b, int c){
    return max(max(a,b),c);
}

//求两个小数的最大值
public double max(double a, double b){
    return a>b?a:b;
}
```

5.4.4 方法的参数传递机制

Java中方法的参数传递机制：值传递

- (1) 形参是基本数据类型时，实参给形参传递数据值，是copy的形式，形参对值的修改不影响实参。
- (2) 形参是引用数据类型时，实参给形参传递地址值，形参对对象的属性的修改，会影响实参对象的属性值，因为此时形参和实参就是指向同一个对象。

示例代码：

```
class Test{
    public static void swap(int a, int b){
        int temp = a;
        a = b;
        b = temp;
    }

    public static void main (String[] args){
        int x = 1;
        int y = 2;
        swap(x,y); //调用完之后，x与y的值不变
    }
}
```

示例代码：

```
class Test{
    public static void change(MyData my){
        my.num *= 2;
    }

    public static void main(String[] args){
        MyData m = new MyData();
        m.num = 1;

        change(m); //调用完之后，m对象的num属性值就变为2
    }
}

class MyData{
    int num;
}
```

陷阱1：

```
/*
陷阱1：在方法中，形参 = 新new对象，那么就与实参无关了
*/
```

```

class Test{
    public static void change(MyData my){
        my = new MyData();//形参指向了新对象
        my.num *= 2;
    }

    public static void main(String[] args){
        MyData m = new MyData();
        m.num = 1;

        change(m);//调用完之后，m对象的num属性值仍然为1
    }
}

class MyData{
    int num;
}

```

陷阱2：见字符串和包装类部分

5.3 对象数组

一维数组：

- 1、元素是基本数据类型
- 2、元素是引用数据类型，也称为对象数组，即数组的元素是对象

注意：对象数组，首先要创建数组对象本身，即确定数组的长度，然后再创建每一个元素对象，如果不创建，数组的元素的默认值就是null，所以很容易出现空指针异常NullPointerException。

示例代码：

```

class MyDate{
    int year;
    int month;
    int day;
}

class Test{
    public static void main(String[] args){
        MyDate[] arr = new MyDate[3];//创建数组对象本身，指定数组的长度

        for(int i=0; i<arr.length; i++){
            arr[i] = new MyDate();//每一个元素要创建对象
            arr[i].year = 1990 + i;
            arr[i].month = 1 + i;
            arr[i].day = 1 + i;
        }
    }
}

```

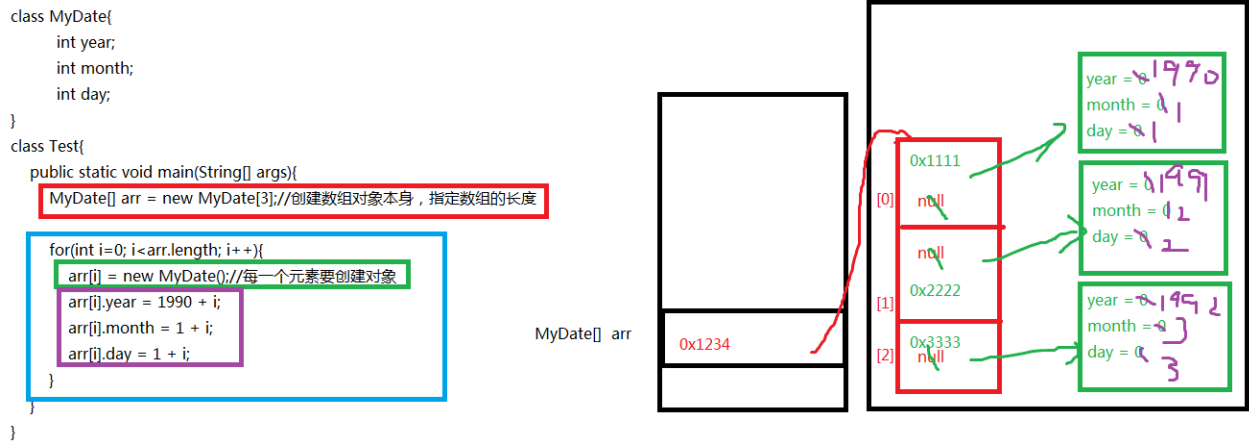


```

    }
}
}

```

对象数组的内存图：



第六章 面向对象的基本特征

面向对象的基本特征：

- 1、封装
- 2、继承
- 3、多态

6.1 封装

1、好处：

- (1) 隐藏实现细节，方便使用者使用
- (2) 安全，可以控制可见范围

2、如何实现封装？

通过权限修饰符

面试题：请按照可见范围从小到大（从大到小）列出权限修饰符？

修饰符	本类	本包	其他包的子类	任意位置
private	√	×	×	×
缺省	√	√	×	×
protected	√	√	√	×
public	√	√	√	√

权限修饰符可以修饰什么？

类（类、接口等）、属性、方法、构造器、内部类

类（外部类）：public和缺省

属性：4种

方法：4种

构造器：4种

内部类：4种

3、通常属性的封装是什么样的？

当然属性的权限修饰符可以是private、缺省、protected、public。但是我们大多数时候，见到的都是private，然后给它们配上get/set方法。

示例代码：标准Javabean的写法

```
public class Student{
    //属性私有化
    private String name;
    private int age;
    private boolean marry;

    //公共的get/set
    public void setName(String n){
        name = n; //这里因为还没有学习this等，可能还会优化
    }
    public String getName(){
        return name;
    }
    public void setAge(int a){
        age = a;
    }
    public int getAge(){
        return age;
    }
}
```

```

    public void setMarry(boolean m){
        marry = m;
    }
    public boolean isMarry(){//boolean类型的属性的get方法，习惯使用把get换成is
        return marry;
    }
}

```

6.2 构造器

1、构造器的作用：

(1) 和new一起使用创建对象

```

//调用无参构造创建对象
类名 对象名 = new 类名();

//调用有参构造创建对象
类名 对象名 = new 类名(实参列表);

```

(2) 可以在创建对象的同时为属性赋值

```

public class Circle{
    private double radius;

    public Circle(){

    }
    public Circle(double r){
        radius = r;//为radius赋值
    }
}

```

2、声明构造器的语法格式：

```

【修饰符】 class 类名{
    【修饰符】 类名(){//无参构造

    }
    【修饰符】 类名(形参列表){//有参构造

    }
}

```

3、构造器的特点：

- (1) 所有的类都有构造器
- (2) 如果一个类没有显式/明确的声明一个构造器，那么编译器将会自动添加一个默认的空参构造
- (3) 如果一个类显式/明确的声明了构造器，那么编译器将不再自动添加默认的空参构造，如果需要，那么就需要手动添加
- (4) 构造器的名称必须与类名相同
- (5) 构造器没有返回值类型
- (6) 构造器可以重载

示例代码：

```
public class Circle{  
    private double radius;  
  
    public Circle(){  
  
    }  
    public Circle(double r){  
        radius = r; //为radius赋值  
    }  
}
```

6.3 关键字this

1、this关键字：

意思：当前对象

- (1) 如果出现在构造器中：表示正在创建的对象
- (2) 如果出现在成员方法中：表示正在调用这个方法的对象

2、this的用法：

- (1) this.属性

当局部变量与成员变量同名时，那么可以在成员变量的前面加“this.”用于区别

- (2) this.方法

调用当前对象的成员方法，完全可以省略“this.”

- (3) this()或this(实参列表)

this()表示调用本类的无参构造

this(实参列表)表示调用本类的有参构造

this()或this(实参列表)要么没有，要么必须出现在构造器的首行

示例代码：

```
public class Student{
    private String name;
    private int score;

    public Student(){

    }
    public Student(String name){
        this.name = name;
    }
    public Student(String name, int score){
        this(name);
    }
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
    public void setScore(int score){
        this.score = score;
    }
    public int getScore(){
        return score;
    }
}
```

3、成员变量与局部变量的区别？

这里只讨论实例变量（关于类变量见static部分）

（1）声明的位置不同

成员变量：类中方法外

局部变量：方法中或代码中

①方法的形参列表

②方法体中局部变量

③代码块中的局部变量

（2）运行时在内存中的存储位置不同

成员变量：堆

局部变量：栈

基本数据类型的变量在栈中，引用数据类型的变量在堆中：不准确

(3) 修饰符

成员变量：有很多修饰符，例如：权限修饰符

局部变量：不能加权限修饰符，唯一的能加的是final

(4) 初始化

成员变量：有默认值

局部变量：没有默认值，必须手动初始化

(5) 生命周期

成员变量：随着对象的创建而创建，随着对象被回收而消亡，即与对象同生共死。每一个对象都是独立的。

局部变量：方法调用时才分配，方法运行结束就没有了。每一次方法调用，都是独立的

6.4 包

1、包的作用：

(1) 可以避免类重名

有了包之后，类的全名称就变为：包.类名

(2) 分类组织管理众多的类

例如：java.lang包，java.util包，java.io包.....

(3) 可以控制某些类型或成员的可见范围

如果某个类型或者成员的权限修饰缺省的话，那么就仅限于本包使用

2、声明包的语法格式：

```
package 包名;
```

注意：

(1)必须在源文件的代码首行

(2)一个源文件只能有一个

3、包的命名规范和习惯：

(1) 所有单词都小写，每一个单词之间使用.分割

(2) 习惯用公司的域名倒置

例如：com.atguigu.xxx;

建议大家取包名时不要使用"java.xx"包

4、使用其他包的类：

前提：被使用的类或成员的权限修饰符是>缺省的

(1) 使用类型的全名称

例如：java.util.Scanner input = new java.util.Scanner(System.in);

(2) 使用import 语句之后，代码中使用简名称

5、import语句

```
import 包.类名;  
import 包.*;
```

注意：当使用两个不同包的同名类时，例如：java.util.Date和java.sql.Date。

一个使用全名称，一个使用简名称

示例代码：

```
package com.atguigu.test;  
  
import java.util.Scanner;  
  
public class Test{  
    public static void main(String[] args){  
        Scanner input = new Scanner(System.in);  
    }  
}
```

6.5 eclipse的使用

1、eclipse管理项目和代码的结构

workspace --> project --> 包-->类...

一个工作空间可以有多个项目。

2、快捷键

常规快捷键：

Ctrl + S：保存

Ctrl + C：复制

Ctrl + V：粘贴

Ctrl + X：剪切

Ctrl + Y：反撤销

Ctrl + Z：撤销

Ctrl + A：全选

eclipse中默认的快捷键：

Ctrl + 1：快速修复

Alt + /：代码提示

Alt + ?： Alt + Shift + / 方法的形参列表提示

Ctrl + D：删除选中行

Ctrl + Alt + ↓：向下复制行

Ctrl + Alt + ↑：向上复制行

Alt + ↓：与下面的行交换位置

Alt + ↑：与上面的行交换位置

Ctrl + Shift + F：快速格式

Ctrl + /：单行注释，再按一次取消

Ctrl + Shift + /：多行注释

Ctrl + Shift + \：取消多行注释

Shift + 回车：在光标下一行插入新航开始编辑

Ctrl + Shift + 回车：在光标上一行插入新航开始编辑

Alt + Shift + A：多行编辑 再按一次退出多行编辑模式

Alt + Shift + S：弹出自动生成代码的菜单选择，包括自动生成构造器、get/set、equals.....

Ctrl + Shift + O：快速导包

Ctrl + Shift + T：打开某个类的源文件

Ctrl + O：打开某个类型的摘要outline

3、快速开发的代码模板

代码模板 + Alt + /

(1) main

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

(2) sysout

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

(3) for

```
for(int i=0; i<数组名.length; i++){  
}
```

其他详细使用见《JavaSE柴林燕相关工具.docx》

6.6 面向对象的基本特征之二：继承

1、为什么要继承？继承的好处？（理解）

(1) 代码的复用

(2) 代码的扩展

2、如何实现继承？

语法格式：

```
【修饰符】 class 子类 extends 父类{  
  
}
```

3、继承的特点

(1) 子类会继承父类的所有特征（属性、方法）

但是，私有的在子类中是不能直接使用的

(2) 子类不会继承父类的构造器

因为，父类的构造器是用于创建父类的对象的

(3) 子类的构造器中又必须去调用父类的构造器

在创建子类对象的同时，为从父类继承的属性进行初始化用，可以借助父类的构造器中的代码为属性赋值。

(4) Java只支持单继承：一个子类只能有一个“直接”父类

(5) Java又支持多层继承：父类还可以有父类，特征会代代相传

(6) 一个父类可以同时拥有很多个子类

6.7 关键字super

super关键字：引用父类的，找父类的xx

用法：

(1) super.属性

当子类声明了和父类同名的成员变量时，那么如果要表示某个成员变量是父类的，那么可以加“super.”

(2) super.方法

当子类重写了父类的方法，又需要在子类中调用父类被重写的方法，可以使用“super.”

(3) super()或super(实参列表)

super()：表示调用父类的无参构造

super(实参列表)：表示调用父类的有参构造

注意：

(1) 如果要写super()或super(实参列表)，必须写在子类构造器的首行

(2) 如果子类的构造器中没有写：super()或super(实参列表)，那么默认会有super()

(3) 如果父类没有无参构造，那么在子类的构造器的首行“必须”写super(实参列表)

6.8 方法的重写

1、方法的重写 (Override)

当子类继承了父类的方法时，又觉得父类的方法体的实现不适合于子类，那么子类可以选择进行重写。

2、方法的重写的要求

(1) 方法名：必须相同

(2) 形参列表：必须相同

(3) 修饰符

权限修饰符：>=

(4) 返回值类型

如果是基本数据类型和void：必须相同

如果是引用数据类型：<=

在Java中我们认为，在概念范围上：子类 < 父类

3、重载（Overload）与重写（Override）的区别

重载（Overload）：在同一个类中，方法名相同，形参列表不同，和返回值类型无关的两个或多个方法。

重写（Override）：在父子类之间。对方法签名的要求见上面。

特殊的重载：

```
public class TestOverload {
    public static void main(String[] args) {
        B b = new B();
        //b对象可以调用几个a方法
        b.a();
        b.a(""); //从b对象同时拥有两个方法名相同，形参不同的角度来说，算是重载
    }
}

class A{
    public void a(){
        //...
    }
}

class B extends A{
    public void a(String str){

    }
}
```

6.9 非静态代码块

1、语法格式

```
【修饰符】 class 类名{
    {
        非静态代码块
    }
}
```

2、作用

目的：在创建的过程中，为对象属性赋值，协助完成实例初始化的过程

3、什么时候执行？

（1）每次创建对象时都会执行

(2) 优先于构造器执行

6.10 实例初始化过程

1、概念描述

- 实例初始化过程：实例对象创建的过程
- 实例初始化方法：实例对象创建时要执行的方法
- 实例初始化方法的由来：它是有编译器编译生成的
- 实例初始化方法的形式：()或(形参列表)
- 实例初始化方法的构成：
 - ①属性的显式赋值代码
 - ②非静态代码块的代码
 - ③构造器的代码

其中

①和②按顺序执行，从上往下

③在①和②的后面

因此一个类有几个构造器，就有几个实例初始化方法。

2、单个类实例初始化方法

示例代码：

```
class Demo{
{
    System.out.println("非静态代码块1");
}

private String str = assign();//调用方法，来为str进行显式赋值

public Demo(){
    System.out.println("无参构造");
}
public Demo(String str){
    this.str = str;
    System.out.println("有参构造");
}

{
    System.out.println("非静态代码块2");
}

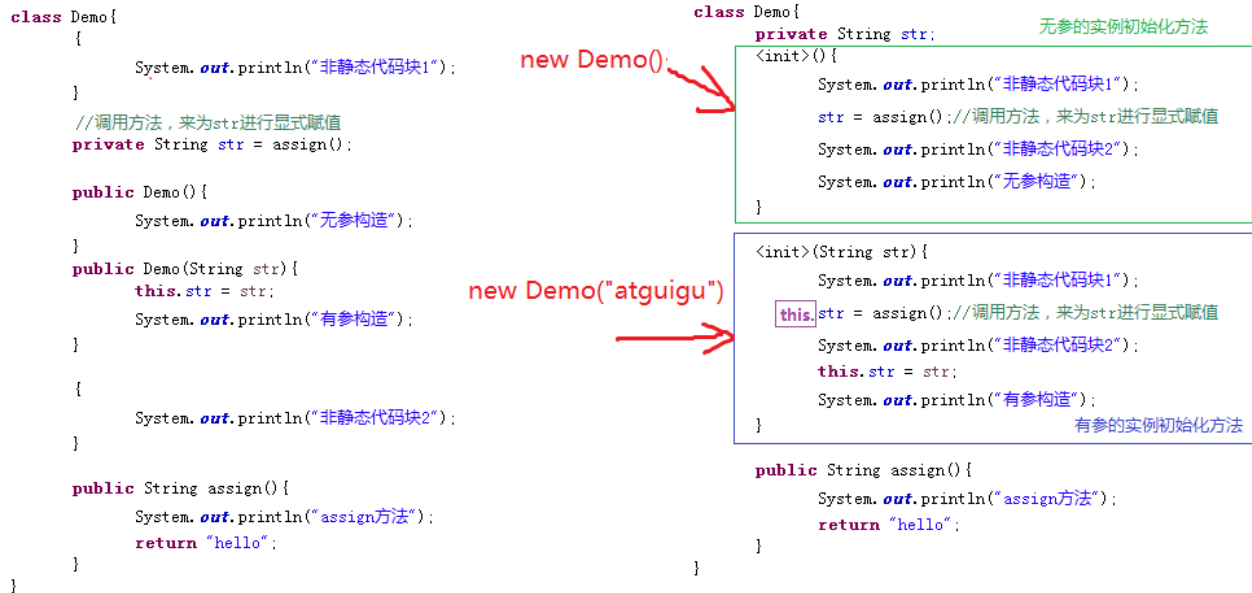
public String assign(){
```

```

        System.out.println("assign方法");
        return "hello";
    }
}

```

图解：



3、父子类的实例初始化

注意：

- (1) 原先`super()`和`super(实参列表)`说是调用父类的构造器，现在就要纠正为调用父类的实例初始化方法了
- (2) 原先`super()`和`super(实参列表)`说是必须在子类构造器的首行，现在要纠正为必须在子类实例初始化方法的首行

结论：

- (1) 执行顺序是先父类实例初始化方法，再子类实例初始化方法
- (2) 如果子类重写了方法，通过子类对象调用，一定是执行重写过的方法

示例代码：

```

class Ba{
    private String str = assign();
    {
        System.out.println("(1)父类的非静态代码块");
    }
    public Ba(){
        System.out.println("(2)父类的无参构造");
    }
    public String assign(){
        System.out.println("(3)父类的assign()");
        return "ba";
    }
}

```

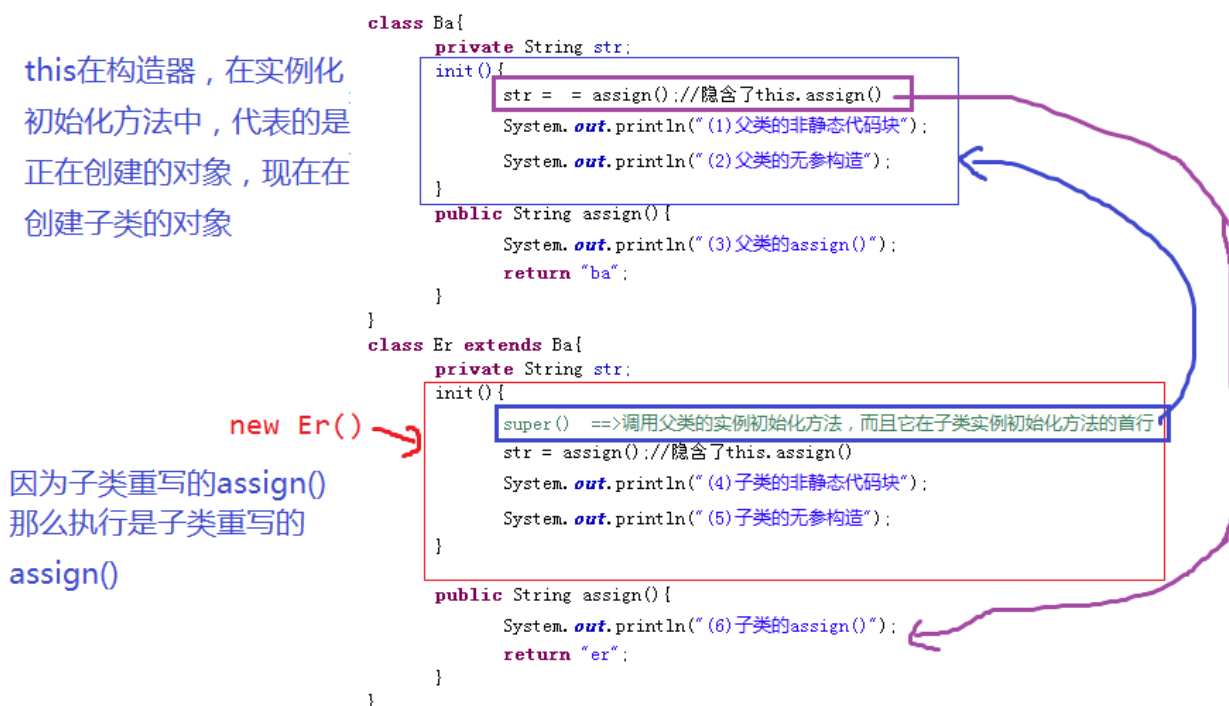
```

    }
}
class Er extends Ba{
    private String str = assign();
    {
        System.out.println("(4)子类的非静态代码块");
    }
    public Er(){
        //super() ==>调用父类的实例初始化方法，而且它在子类实例初始化方法的首行
        System.out.println("(5)子类的无参构造");
    }

    public String assign(){
        System.out.println("(6)子类的assign()");
        return "er";
    }
}
class Test{
    public static void main(String[] args){
        new Er();//612645
    }
}

```

图解：



6.11 面向对象的基本特征之三：多态

1、多态：

语法格式：

父类 引用/变量 = 子类的对象;

2、前提:

- (1) 继承
- (2) 方法的重写
- (3) 多态引用

3、现象:

编译时看左边/"父类", 运行时看右边/"子类".

编译时, 因为按父类编译, 那么只能父类有的方法, 子类扩展的方法是无法调用的;

执行时一定是运行子类重写的过的方法体。

示例代码:

```
class Person{
    public void eat(){
        System.out.println("吃饭");
    }
    public void walk(){
        System.out.println("走路");
    }
}
class Woman extends Person{
    public void eat(){
        System.out.println("细嚼慢咽的吃饭");
    }
    public void walk(){
        System.out.println("婀娜多姿走路");
    }
    public void shop(){
        System.out.println("买买买...");
    }
}
class Man extends Person{
    public void eat(){
        System.out.println("狼吞虎咽的吃饭");
    }
    public void walk(){
        System.out.println("大摇大摆的走路");
    }
    public void smoke(){
        System.out.println("吞云吐雾");
    }
}
```

```

class Test{
    public static void main(String[] args){
        Person p = new Woman();//多态引用
        p.eat();//执行子类重写
        p.walk();//执行子类重写
        //p.shop();//无法调用
    }
}

```

4、应用：

- (1) 多态参数：形参是父类，实参是子类对象
- (2) 多态数组：数组元素类型是父类，元素存储的是子类对象

示例代码：多态参数

```

class Test{
    public static void main(String[] args){
        test(new Woman());//实参是子类对象
        test(new Man());//实参是子类对象
    }
    public static void test(Person p){//形参是父类类型
        p.eat();
        p.walk();
    }
}

```

示例代码：多态数组

```

class Test{
    public static void main(String[] args){
        Person[] arr = new Person[2];//多态数组
        arr[0] = new Woman();
        arr[1] = new Man();

        for(int i=0; i<arr.length; i++){
            all[i].eat();
            all[i].walk();
        }
    }
}

```

5、向上转型与向下转型：父子类之间的转换

- (1) 向上转型：自动类型转换

当把子类的对象赋值给父类的变量时（即多态引用时），在编译时，这个对象就向上转型为父类。此时就看不见子类“特有、扩展”的方法。

(2) 向下转型：强制转换。有风险，可能会报ClassCastException异常。

当需要把父类的变量赋值给一个子类的变量时，就需要向下转型。

要想转型成功，必须保证该变量中保存的对象的运行时类型是<=强转的类型

示例代码：

```
class Person{
    //方法代码省略...
}
class Woman extends Person{
    //方法代码省略...
}
class ChineseWoman extends Woman{
    //方法代码省略...
}
```

```
public class Test{
    public static void main(String[] args){
        //向上转型
        Person p1 = new Woman();
        //向下转型
        Woman m = (Woman)p1;
        //p1变量中实际存储的对象就是Woman类型，和强转的Woman类型一样

        //向上转型
        Person p2 = new ChineseWoman();
        //向下转型
        Woman w2 = (Woman) p2;
        //p2变量中实际存储的对象是ChineseWoman类型，强制的类型是Woman，ChineseWoman<Woman
        类型
    }
}
```

6、instanceof

表达式语法格式：

```
对象/变量 instanceof 类型
```

运算结果：true 或 false

作用：

用来判断这个对象是否属于这个类型，或者说，是否是这个类型的对象或这个类型子类的对象

示例代码：

```
class Person{
    //方法代码省略...
}
class Woman extends Person{
    //方法代码省略...
}
class ChineseWoman extends Woman{
    //方法代码省略...
}
```

```
public class Test{
    public static void main(String[] args){
        Person p = new Person();
        Woman w = new Woman();
        ChineseWoman c = new ChineseWoman();

        if(p instanceof Woman){//false
        }
        if(w instanceof Woman){//true
        }
        if(c instanceof Woman){//true
        }
    }
}
```

第七章 面向对象的高级特性

修饰符的学习围绕三个问题：

- (1) 单词的意思
- (2) 可以修饰什么？
- (3) 用它修饰后有什么不同？

7.1 关键字：final

final：最终的

用法：

- (1) 修饰类（包括外部类、内部类类）

表示这个类不能被继承，没有子类

(2) 修饰方法

表示这个方法不能被重写

(3) 修饰变量（成员变量（类变量、实例变量），局部变量）

表示这个变量的值不能被修改

注意：如果某个成员变量用final修饰后，也得手动赋值，而且这个值一旦赋完，就不能修改了，即没有set方法

7.2 关键字：native

native：本地的，原生的

用法：

只能修饰方法

表示这个方法的方法体代码不是用Java语言实现的。

但是对于Java程序员来说，可以当做Java的方法一样去正常调用它，或者子类重写它。

JVM内存的管理：



方法区：类的信息、常量、静态变量、动态编译生成的字节码信息

虚拟机栈：Java语言实现的方法的局部变量

本地方法栈：非Java语言实现的方法的局部变量，即native方法执行时的内存区域

堆：new出来的对象

程序计数器：记录每一个线程目前执行到哪一句指令

7.3 关键字：static

static：静态的

用法：

1、成员方法：我们一般称为静态方法或类方法

(1) 不能被重写

(2) 被使用

本类中：其他方法中可以直接使用它

其他类中：可以使用“类名.方法”进行调用，也可以使用“对象名.方法”，推荐使用“类名.方法”

(3) 在静态方法中，我们不能出现：this，super，非静态的成员

2、成员变量：我们一般称为静态变量或类变量

(1) 静态变量的值是该类所有对象共享的

(2) 静态变量存储在方法区

(3) 静态变量对应的get/set也是静态的

(4) 静态变量与局部变量同名时，就可以使用“类名.静态变量”进行区分

3、内部类：后面讲

4、代码块：静态代码块

5、静态导入（JDK1.5引入）

没有静态导入

```
package com.atguigu.utils;

public class Utils{
    public static final int MAX_VALUE = 1000;
    public static void test(){
        //...
    }
}
```

```

package com.atguigu.test;

import com.atguigu.utils;

public class Test{
    public static void main(String[] args){
        System.out.println(Utils.MAX_VALUE);
        Utils.test();
    }
}

```

使用静态导入

```

package com.atguigu.utils;

public class Utils{
    public static final int MAX_VALUE = 1000;
    public static void test(){
        //...
    }
}

```

```

package com.atguigu.test;

import static com.atguigu.utils.Utils.*;

public class Test{
    public static void main(String[] args){
        System.out.println(MAX_VALUE);
        test();
    }
}

```

7.4 静态代码块

1、语法格式：

```

【修饰符】 class 类名{
    static{
        静态代码块;
    }
}

```

2、作用：

协助完成类初始化，可以为类变量赋值。

3、类初始化()

类的初始化有：

①静态变量的显式赋值代码

②静态代码块中代码

其中①和②按顺序执行

注意：类初始化方法，一个类只有一个

4、类的初始化的执行特点：

(1) 每一个类的()只执行一次

(2) 如果一个子类在初始化时，发现父类也没有初始化，会先初始化父类

(3) 如果既要类初始化又要实例化初始化，那么一定是先完成类初始化的

7.5 变量的分类与区别

1、变量按照数据类型分：

(1) 基本数据类型的变量，里面存储数据值

(2) 引用数据类型的变量，里面存储对象的地址值

```
int a = 10; //a中存储的是数据值
```

```
Student stu = new Student(); //stu存储的是对象的地址值
```

```
int[] arr = new int[5]; //arr存储的是数组对象的地址值
```

```
String str = "hello"; //str存储的是"hello"对象的地址值
```

2、变量按照声明的位置不同：

(1) 成员变量

(2) 局部变量

3、成员变量与局部变量的区别

(1) 声明的位置不同

成员变量：类中方法外

局部变量：（1）方法的{}中，即形参（2）方法体的{}的局部变量（3）代码块{}中

（2）存储的位置不同

成员变量：

如果是静态变量（类变量），在方法区中

如果是非静态的变量（实例变量），在堆中

局部变量：栈

（3）修饰符不同

成员变量：4种权限修饰符、static、final。。。。

局部变量：只有final

（4）生命周期

成员变量：

如果是静态变量（类变量），和类相同

如果是非静态的变量（实例变量），和所属的对象相同，每一个对象是独立

局部变量：每次执行都是新的

（5）作用域

成员变量：

如果是静态变量（类变量），在本类中随便用，在其他类中使用“类名.静态变量”

如果是非静态的变量（实例变量），在本类中只能在非静态成员中使用，在其他类中使用“对象名.非静态的变量”

局部变量：有作用域

7.7 根父类

1、java.lang.Object类是类层次结构的根父类。包括数组对象。

（1）Object类中声明的所有的方法都会被继承到子类中，那么即所有的对象，都拥有Object类中的方法

（2）每一个对象的创建，最终都会调用到Object实例初始化方法()

（3）Object类型变量、形参、数组，可以存储任意类型的对象

2、Object类的常用方法

（1）public String toString():

①默认情况下，返回的是“对象的运行时类型 @ 对象的hashCode值的十六进制形式”

②通常是建议重写，如果在eclipse中，可以用Alt +Shift + S-->Generate toString()

③如果我们直接System.out.println(对象), 默认会自动调用这个对象的toString()

(2) public final Class<?> getClass(): 获取对象的运行时类型

(3) protected void finalize(): 当对象被GC确定为要被回收的垃圾, 在回收之前由GC帮你调用这个方法。而且这个方法只会被调用一次。子类可以选择重写。

(4) public int hashCode(): 返回每个对象的hash值。

规定: ①如果两个对象的hash值是不同的, 那么这两个对象一定不相等;

②如果两个对象的hash值是相同的, 那么这两个对象不一定相等。

主要用于后面当对象存储到哈希表等容器中时, 为了提高性能用的。

(5) public boolean equals(Object obj): 用于判断当前对象this与指定对象obj是否“相等”

①默认情况下, equals方法的实现等价于与“==”, 比较的是对象的地址值

②我们可以选择重写, 重写有些要求:

A: 如果重写equals, 那么一定要一起重写hashCode()方法, 因为规定:

a: 如果两个对象调用equals返回true, 那么要求这两个对象的hashCode值一定是相等的;

b: 如果两个对象的hashCode值不同的, 那么要求这个两个对象调用equals方法一定是false;

c: 如果两个对象的hashCode值相同的, 那么这个两个对象调用equals可能是true, 也可能是false

B: 如果重写equals, 那么一定要遵循如下几个原则:

a: 自反性: x.equals(x)返回true

b: 传递性: x.equals(y)为true, y.equals(z)为true, 然后x.equals(z)也应该为true

c: 一致性: 只要参与equals比较的属性值没有修改, 那么无论何时调用结果应该一致

d: 对称性: x.equals(y)与y.equals(x)结果应该一样

e: 非空对象与null的equals一定是false

7.8 关键字: abstract

1、什么时候会用到抽象方法和抽象类?

当声明父类的时候, 在父类中某些方法的方法体的实现不能确定, 只能由子类决定。但是父类中又要体现子类的共同的特征, 即它要包含这个方法, 为了统一管理各种子类的对象, 即为了多态的应用。

那么此时, 就可以选择把这样的方法声明为抽象方法。如果一个类包含了抽象方法, 那么这个类就必须是个抽象类。

2、抽象类的语法格式


```
【权限修饰符】 abstract class 类名{  
  
}  
【权限修饰符】 abstract class 类名 extends 父类{  
  
}
```

3、抽象方法的语法格式

```
【其他修饰符】 abstract 返回值类型 方法名(【形参列表】);
```

抽象方法没有方法体

4、抽象类的特点

- (1) 抽象类不能直接实例化，即不能直接new对象
- (2) 抽象类就是用来被继承的，那么子类继承了抽象类后，必须重写所有的抽象方法，否则这个子类也得是抽象类
- (3) 抽象类也有构造器，这个构造的作用不是创建抽象类自己的对象用的，给子类在实例化过程中调用；
- (4) 抽象类也可以没有抽象方法，那么目的是不让你创建对象，让你创建它子类的对象
- (5) 抽象类的变量与它子类的对象也构成多态引用

5、不能和abstract一起使用的修饰符？

- (1) final：和final不能一起修饰方法和类
- (2) static：和static不能一起修饰方法
- (3) native：和native不能一起修饰方法
- (4) private：和private不能一起修饰方法

7.9 接口

1、接口的概念

接口是一种标准。注意关注行为标准（即方法）。

面向对象的开发原则中有一条：面向接口编程。

2、接口的声明格式

```
【修饰符】 interface 接口名{  
    接口的成员列表;  
}
```

3、类实现接口的格式

```
【修饰符】 class 实现类 implements 父接口们{  
  
}  
  
【修饰符】 class 实现类 extends 父类 implements 父接口们{  
  
}
```

4、接口继承接口的格式

```
【修饰符】 interface 接口名 extends 父接口们{  
    接口的成员列表;  
}
```

5、接口的特点

- (1) 接口不能直接实例化，即不能直接new对象
- (2) 只能创建接的实现类对象，那么接口与它的实现类对象之间可以构成多态引用。
- (3) 实现类在实现接口时，必须重写所有抽象的方法，否则这个实现类也得是抽象类。
- (4) Java规定类与类之间，只能是单继承，但是Java的类与接口之间是多实现的关系，即一个类可以同时实现多个接口
- (5) Java还支持接口与接口之间的多继承。

6、接口的成员

JDK1.8之前：

- (1) 全局的静态的常量：public static final，这些修饰符可以省略
- (2) 公共的抽象方法：public abstract，这些修饰符也可以省略

JDK1.8之后：

- (3) 公共的静态的方法：public static，这个就不能省略了
- (4) 公共的默认的方法：public default，这个就不能省略了

7、默认方法冲突问题

- (1) 当一个实现类同时实现了两个或多个接口，这个多个接口的默认方法的签名相同。

解决方案：

方案一：选择保留其中一个

```
接口名.super.方法名(【实参列表】);
```

方案二：完全重写

(2) 当一个实现类同时继承父类，又实现接口，父类中有一个方法与接口的默认方法签名相同

解决方案：

方案一：默认方案，保留父类的

方案二：选择保留接口的

```
接口名.super.方法名(【实参列表】);
```

方案三：完全重写

8、示例代码

```
public interface Flyable{
    long MAX_SPEED = 7900000;
    void fly();
}
```

```
public class Bird implements Flyable{
    public void fly(){
        //....
    }
}
```

9、常用的接口

(1) java.lang.Comparable接口：自然排序

抽象方法：int compareTo(Object obj)

(2) java.util.Comparator接口：定制排序

抽象方法：int compare(Object obj1 ,Object obj2)

(3) 示例代码

如果员工类型，默认顺序，自然顺序是按照编号升序排列，那么就实现Comparable接口

```

class Employee implements Comparable{
    private int id;
    private String name;
    private double salary;

    //省略了构造器, get/set, toString

    @Override
    public int compareTo(Object obj){
        return id - ((Employee)obj).id;
    }
}

```

如果在后面又发现有新的需求，想要按照薪资排序，那么只能选择用定制排序，实现Comparator接口

```

class SalaryComparator implements Comparator{
    public int compare(Object o1, Object o2){
        Employee e1 = (Employee)o1;
        Employee e2 = (Employee)o2;
        if(e1.getSalary() > e2.getSalary()){
            return 1;
        }else if(e1.getSalary() < e2.getSalary()){
            return -1;
        }
        return 0;
    }
}

```

7.10 内部类

1、内部类的概念

声明在另外一个类里面的类就是内部类。

2、内部类的4种形式

- (1) 静态内部类
- (2) 非静态成员内部类
- (3) 有名字的局部内部类
- (4) 匿名内部类

7.10.1 匿名内部类

1、语法格式：

```
//在匿名子类中调用父类的无参构造
new 父类(){
    内部类的成员列表
}

//在匿名子类中调用父类的有参构造
new 父类(实参列表){
    内部类的成员列表
}

//接口没有构造器，那么这里表示匿名子类调用自己的无参构造，调用默认父类Object的无参构造
new 父接口名(){

}
```

2、匿名内部类、匿名对象的区别？

```
System.out.println(new Student("张三")); //匿名对象

Student stu = new Student("张三"); //这个对象有名字，stu

//既有匿名内部类，又是一个匿名的对象
new Object(){
    public void test(){
        .....
    }
}.test();

//这个匿名内部类的对象，使用obj这个名字引用它，既对象有名字，但是这个Object的子类没有名字
Object obj = new Object(){
    public void test(){
        .....
    }
};
```

3、使用的形式

(1) 示例代码：继承式

```
abstract class Father{
    public abstract void test();
}

class Test{
    public static void main(String[] args){
        //用父类与匿名内部类的对象构成多态引用
        Father f = new Father(){
```

```

        public void test(){
            System.out.println("用匿名内部类继承了Father这个抽象类，重写了test抽象方法")
        }
    };
    f.test();
}
}

```

(2) 示例代码：实现式

```

interface Flyable{
    void fly();
}
class Test{
    public static void main(String[] args){
        //用父接口与匿名内部类的对象构成了多态引用
        Flyable f = new Flyable(){
            public void fly(){
                System.out.println("用匿名内部类实现了Flyable这个接口，重写了抽象方法");
            }
        };
        f.fly();
    }
}

```

(3) 示例代码：用匿名内部类的匿名对象直接调用方法

```

new Object(){
    public void test(){
        System.out.println("用匿名内部类的匿名对象直接调用方法")
    }
}.test();

```

(4) 示例代码：用匿名内部类的匿名对象直接作为实参

```

Student[] all = new Student[3];
all[0] = new Student("张三",23);
all[1] = new Student("李四",22);
all[2] = new Student("王五",20);

//用匿名内部类的匿名对象直接作为实参
//这个匿名内部类实现了Comparator接口
//这个匿名内部类的对象，是定制比较器的对象
Arrays.sort(all, new Comparator(){
    public int compare(Object o1, Object o2){
        Student s1 = (Student)o1;

```

```

        Student s2 = (Student)o2;
        return s1.getAge() - s2.getAge();
    }
});

```

7.10.2 静态内部类

1、语法格式

```

【修饰符】 class 外部类名 【extends 外部类的父类】 【implements 外部类的父接口们】 {
    【其他修饰符】 static class 静态内部类 【extends 静态内部类自己的父类】 【implements
静态内部类的父接口们】 {
        静态内部类的成员列表;
    }

    外部类的其他成员列表;
}

```

2、使用注意事项

(1) 包含成员是否有要求：

可以包含类的所有成员

(2) 修饰符要求：

- 权限修饰符：4种
- 其他修饰符：abstract、final

(3) 使用外部类的成员上是否有要求

- 只能使用外部类的静态成员

(4) 在外部类中使用静态内部类是否有要求

- 正常使用

(5) 在外部类的外面使用静态内部类是否有要求

(1) 如果使用的是静态内部类的静态成员

外部类名.静态内部类名.静态成员

(2) 如果使用的是静态内部类的非静态成员

①先创建静态内部类的对象

外部类名.静态内部类名 对象名 = new 外部类名.静态内部类名(【实参列表】);

②通过对象调用非静态成员

对象名.xxx

(6) 字节码文件形式：外部类名\$静态内部类名.class

3、示例代码

```
class Outer{
    private static int i = 10;
    static class Inner{
        public void method(){
            //...
            System.out.println(i); //可以
        }
        public static void test(){
            //...
            System.out.println(i); //可以
        }
    }

    public void outMethod(){
        Inner in = new Inner();
        in.method();
    }

    public static void outTest(){
        Inner in = new Inner();
        in.method();
    }
}

class Test{
    public static void main(String[] args){
        Outer.Inner.test();

        Outer.Inner in = new Outer.Inner();
        in.method();
    }
}
```

7.10.3 非静态内部类

1、语法格式

```
【修饰符】 class 外部类名 【extends 外部类的父类】 【implements 外部类的父接口们】 {
    【修饰符】 class 非静态内部类 【extends 非静态内部类自己的父类】 【implements 非静态内部类的父接口们】 {
        非静态内部类的成员列表;
    }

    外部类的其他成员列表;
}
```


2、使用注意事项

(1) 包含成员是否有要求：

不允许出现静态的成员

(2) 修饰符要求

权限修饰符：4种

其他修饰符：abstract, final

(3) 使用外部类的成员上是否有要求

都可以使用

(4) 在外部类中使用非静态内部类是否有要求

在外部类的静态成员中不能使用非静态内部类

(5) 在外部类的外面使用非静态内部类是否有要求

//使用非静态内部类的非静态成员

//(1)创建外部类的对象

外部类名 对象名1 = new 外部类名(【实参列表】);

//(2)通过外部类的对象去创建或获取非静态内部类的对象

//创建

外部类名.非静态内部类名 对象名2 = 对象名1.new 非静态内部类名(【实参列表】);

//获取

外部类名.非静态内部类名 对象名2 = 对象名1.get非静态内部类对象的方法(【实参列表】);

//(3)通过非静态内部类调用它的非静态成员

对象名2.xxx

(6) 字节码文件形式：外部类名\$非静态内部类名.class

3、示例代码

```
class Outer{
    private static int i = 10;
    private int j = 20;
    class Inner{
        public void method(){
            //...
            System.out.println(i);//可以
            System.out.println(j);//可以
        }
    }
}
```

```

    public void outMethod(){
        Inner in = new Inner();
        in.method();
    }
    public static void outTest(){
        // Inner in = new Inner();//不可以
    }

    public Inner getInner(){
        return new Inner();
    }
}
class Test{
    public static void main(String[] args){
        Outer out = new Outer();

        Outer.Inner in1 = out.new Inner();    //创建
        in1.method();

        Outer.Inner in2 = out.getInner(); //获取
        in2.method();
    }
}

```

7.10.4 局部内部类

1、语法格式

```

【修饰符】 class 外部类名    【extends 外部类的父类】    【implements 外部类的父接口们】 {
    【修饰符】 返回值类型    方法名(【形参列表】){
        【修饰符】 class 局部内部类    【extends 局部内部类自己的父类】    【implements 局部
内部类的父接口们】 {
            局部内部类的成员列表;
        }
    }
    外部类的其他成员列表;
}

```

2、使用注意事项

(1) 包含成员是否有要求

不允许出现静态的成员

(2) 修饰符要求

权限修饰符：不能

其他修饰符：abstract、final

(3) 使用外部类的成员等上是否有要求

①使用外部类的静态成员：随使用

②使用外部类的非静态成员：能不能用要看所在的方法是否是静态的

③使用所在方法的局部变量：必须 final修饰的

(4) 在外部类中使用局部内部类是否有要求

有作用域

(5) 在外部类的外面使用局部内部类是否有要求

没法使用

(6) 字节码文件形式：外部类名\$编号局部内部类名.class

3、示例代码

```
class Outer{
    private static int i = 10;
    private int j = 20;

    public void outMethod(){
        class Inner{
            public void method(){
                //...
                System.out.println(i); //可以
                System.out.println(j); //可以
            }
        }
        Inner in = new Inner();
        in.method();
    }
    public static void outTest(){
        final int k = 30;
        class Inner{
            public void method(){
                //...
                System.out.println(i); //可以
                System.out.println(j); //不可以
                System.out.println(k); //可以
            }
        }
        Inner in = new Inner();
        in.method();
    }
}
```

```
}  
}
```

第八章 枚举与注解

8.1 枚举

1、枚举（JDK1.5引入的）

枚举类型的对象是有限、固定的几个常量对象。

2、语法格式

```
//形式一：枚举类型中只有常量对象列表  
【修饰符】 enum 枚举类型名 {  
    常量对象列表  
}  
  
//形式二：枚举类型中只有常量对象列表  
【修饰符】 enum 枚举类型名 {  
    常量对象列表;  
  
    其他成员列表;  
}
```

说明：常量对象列表必须在枚举类型的首行

回忆：首行

- (1) `super()`或`super(实参列表)`：必须在子类构造器的首行
- (2) `this()`或`this(实参列表)`：必须在本类构造器的首行
- (3) `package 包;` 声明包的语句必须在源文件.java的代码首行
- (4) 枚举常量对象列表必须在枚举类型的首行

3、在其他类中如何获取枚举的常量对象

```
//获取一个常量对象
枚举类型名.常量对象名

//获取一个常量对象
枚举类型名.valueOf("常量对象名")

//获取所有常量对象
枚举类型名[] all = 枚举类型名.values();
```

4、枚举类型的特点

- (1) 枚举类型有一个公共的基本的父类，是java.lang.Enum类型，所以不能再继承别的类型
- (2) 枚举类型的构造器必须是私有的
- (3) 枚举类型可以实现接口

```
interface MyRunnable{
    void run();
}
enum Gender implements MyRunnable{
    NAN,NV;
    public void run(){
        //...
    }
}
//或
enum Gender implements MyRunnable{
    NAN{
        public void run(){
            //...
        }
    },NV{
        public void run(){
            //...
        }
    };
}
```

5、父类java.lang.Enum类型

- (1) 构造器

protected Enum(String name, int ordinal): 由编译器自动调用

- (2) String name(): 常量对象名

- (3) `int ordinal()`: 返回常量对象的序号, 第一个的序号是0
- (4) `String toString()`: 返回常量对象名, 如果子类想重写, 需要手动
- (5) `int compareTo(Object obj)`: 按照常量对象的顺序比较

8.2 注解

1、注解

它是代码级别的注释

2、标记符号: @

3、系统预定义的三个最基本的注解:

- (1) `@Override`: 表示某个方法是重写的方法

它只能用在方法上面, 会让编译器对这个方法进行格式检查, 是否满足重写的要求

- (2) `@SuppressWarnings(xx)`: 抑制警告

- (3) `@Deprecated`: 表示xx已过时

4、和文档注释相关的注解

- (1) 文档注释

```
/**
 文档注释
 */
```

- (2) 常见的文档注释

`@author`: 作者

`@since`: 从xx版本加入的

`@see`: 另请参考

`@param`: 形参

`@return`: 返回值

`@throws`或`@exception`: 异常

5、JUnit相关的几个注解

(1) @Test: 表示它是一个单元测试方法

这个方法需要是: `public void xxx(){}`

(2) @Before: 表示在每一个单元测试方法之前执行

这个方法需要是: `public void xxx(){}`

(3) @After: 表示在每一个单元测试方法之后执行

这个方法需要是: `public void xxx(){}`

(4) @BeforeClass: 表示在类初始化阶段执行, 而且只执行一次

这个方法需要是: `public static void xxx(){}`

(3) @AfterClass: 表示在类的“卸载”阶段执行, 而且只执行一次

这个方法需要是: `public static void xxx(){}`

6、元注解

(1) @Target(xx): 用它标记的注解能够用在xx位置

(xx): 由ElementType枚举类型的10个常量对象指定, 例如: TYPE, METHOD, FIELD等

例如:

```
@Target(ElementType.TYPE)
```

```
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})
```

```
import static java.lang.annotation.ElementType.*;
```

```
@Target({TYPE, METHOD, FIELD})
```

(2) @Retention (xx) : 用它标记的注解可以滞留到xx阶段

(xx): 由RetentionPolicy枚举类型的3个常量对象指定, 分别是: SOURCE, CLASS, RUNTIME

唯有RUNTIME阶段的注解才能被反射读取到

例如:

```
@Retention(RetentionPolicy.RUNTIME)
```

(3) @Documentd: 用它标记的注解可以读取到API中

(4) @Inherited: 用它标记的注解可以被子类继承

7、自定义注解

```
@元注解
【修饰符】 @interface 注解名{

}

@元注解
【修饰符】 @interface 注解名{
    配置参数列表
}
```

配置参数的语法格式：

```
数据类型 配置参数名();

或

数据类型 配置参数名() default 默认值;
```

关于配置参数：

(1) 配置参数的类型有要求：

八种基本数据类型、String、枚举、Class类型、注解、它们的数组。

(2) 如果自定义注解声明了配置参数，那么在使用这个注解时必须为配置参数赋值，除非它有默认值

```
@自定义注解名(配置参数名1=值, 配置参数名2=值。。。)

//如果配置参数类型是数组，那么赋值时，可以用{}表示数组
@自定义注解名(配置参数名1={值}, 配置参数名2=值。。。)
```

(3) 如果配置参数只有一个，并且名称是value，那么赋值时可以省略value=

(4) 如果读取这个注解时，要获取配置参数的值的话，可以当成方法一样来访问

```
自定义注解对象.配置参数();
```

第九章 异常

9.1 异常的类型体系结构

1、异常系列的超父类：java.lang.Throwable

(1) 只有它或它子类的对象，才能被JVM或throw语句“抛”出

(2) 也只有它或它子类的对象，才能被catch“捕获”

2、Throwable分为两大派别

(1) Error：严重的错误，需要停下来重新设计、升级解决这个问题

(2) Exception：一般的异常，可以通过判断、检验进行避免，或者使用try...catch进行处理

3、Exception又分为两大类

(1) 运行时异常：

它是RuntimeException或它子类的对象。

这种类型的异常，编译器不会提醒你，要进行throws或try...catch进行处理，但是运行时可能导致崩溃。

(2) 编译时异常：

异常除了运行时异常以外的都是编译时异常。

这种类型的异常，编译器是强制要求你，throws或try...catch进行处理，否则编译不通过。

4、列出常见的异常类型

(1) 运行时异常

RuntimeException、NullPointerException（空指针异常），ClassCastException（类型转换异常），ArithmeticException（算术异常），NubmerFormatException（数字格式化异常），IndexOutOfBoundsException（下标越界异常）（ArrayIndexOutOfBoundsException（数组下标越界异常）、StringIndexOutOfBoundsException（字符串下标越界异常））、InputMismatchException（输入类型不匹配异常）。。。。

(2) 编译时异常

FileNotFoundException（文件找不到异常）、IOException（输入输出异常）、SQLException（数据库sql语句执行异常）。。。

9.2 异常的处理

1、在当前方法中处理：try...catch...finally

```
//形式一： try...catch
try{
    可能发生异常的代码
```

```

}catch(异常类型 异常名e){
    处理异常的代码（一般都是打印异常的信息的语句）
}catch(异常类型 异常名e){
    处理异常的代码（一般都是打印异常的信息的语句）
}。。。

//形式二： try...finally
try{
    可能发生异常的代码
}finally{
    无论try中是否有异常，也不管是不是有return，都要执行的部分
}

//形式三： try..catch..finally
try{
    可能发生异常的代码
}catch(异常类型 异常名e){
    处理异常的代码（一般都是打印异常的信息的语句）
}catch(异常类型 异常名e){
    处理异常的代码（一般都是打印异常的信息的语句）
}。。。
finally{
    无论try中是否有异常，也不管catch是否可以捕获异常，也不管try和catch中是不是有return，
    都要执行的部分
}

```

执行特点：

- (1) 如果try中的代码没有异常，那么try中的代码会正常执行，catch部分就不执行，finally中会执行
- (2) 如果try中的代码有异常，那么try中发生异常的代码的后面就不执行了，找对应的匹配的catch分支执行，finally中会执行



2、finally与return混合使用时

(1) 如果finally中有return，一定从finally中的return返回。

此时try和catch中的return语句，执行了一半，执行了第一个动作。所以，finally中的return语句会覆盖刚刚的返回值

return 返回值; 语句有两个动作：（1）把返回值放到“操作数栈”中，等当前方法结束后，这个“操作数栈”中的值会返回给调用处（2）结束当前方法的执行

(2) 如果finally中没有return，finally中的语句会执行，但是不影响最终的返回值

即try和catch中的return语句两步拆开来走，先把（1）把返回值放到“操作数栈”中，（2）然后走finally中的语句（3）再执行return后半动作，结束当前方法

3、在当前方法中不处理异常，明确要抛给调用者处理，使用throws

语法格式：

```
【修饰符】 返回值类型 方法名(【形参列表】) throws 异常列表{  
  
}
```

此时调用者，就知道需要处理哪些异常。

方法的重写的要求：

(1) 方法名：相同

(2) 形参列表：相同

(3) 返回值类型：

基本数据类型和void：相同

引用数据类型：<=

(4) 修饰符：

权限修饰符：>=

其他修饰符：static, final, private不能被重写

(5) throws：<=

方法的重载：

- (1) 方法名：相同
- (2) 形参列表：必须不同
- (3) 返回值类型：无关
- (4) 修饰符：无关
- (5) throws：无关

9.3 手动抛出异常：throw

```
throw 异常对象;  
  
//例如:  
throw new AccountException("xxx");
```

throw抛出来的异常对象，和VM抛出来的异常对象一样，也要用try..catch处理或者throws。

如果是运行时异常，编译器不会强制要求你处理，如果是编译时异常，那么编译器会强制要求你处理。

9.4 自定义异常

1、必须继承Throwable或它的子类

我们见到比较多的是继承RuntimeException和Exception.

如果你继承RuntimeException或它的子类，那么你自定义的这个异常就是运行时异常。编译器就不会提醒你处理。

如果你继承Exception，那么它属于编译时异常，编译器会强制你处理。

2、建议大家保留两个构造器

```
//无参构造  
public 自定义异常名(){  
  
}  
  
//有参构造  
public 自定义异常名(String message){  
    super(message);  
}
```

3、自定义异常对象，必须手动抛出，用throw抛出

9.5 关于异常的几个方法

(1) `e.printStackTrace()`: 打印异常对象的详细信息, 包括异常类型, `message`, 堆栈跟踪信息。这个对于调试, 或者日志跟踪是非常有用的

(2) `e.getMessage()`: 只是获取异常的`message`信息

关于异常信息的打印:

用`System.err`打印和用`e.printStackTrace()`都是会标记红色的突出。

用`System.out`打印, 当成普通信息打印。

这两个打印是两个独立的线程, 顺序是不能精确控制的。

第十章 多线程

10.1 相关的概念

1、程序 (Program)

为了实现一个功能, 完成一个任务而选择一种编程语言编写的一组指令的集合。

2、进程 (Process)

程序的一次运行。操作系统会给这个进程分配资源 (内存)。

进程是操作系统分配资源的最小单位。

进程与进程之间的内存是独立, 无法直接共享。

最早的DOS操作系统是单任务的, 同一时间只能运行一个进程。后来现在的操作系统都是支持多任务的, 可以同时运行多个进程。进程之间来回切换。成本比较高。

3、线程 (Thread)

线程是进程中的其中一条执行路径。一个进程中至少有一个线程, 也可以有多个线程。有的时候也把线程称为轻量级的进程。

同一个进程的多个线程之间有些内存是可以共享的 (方法区、堆), 也有些内存是独立的 (栈 (包括虚拟机栈和本地方法栈)、程序计数器)。

线程之间的切换相对进程来说成本比较低。

4、并行: 多个处理器同时可以执行多条执行路径。

5、并发: 多个任务同时执行, 但是可能存在先后关系。

10.2 两种实现多线程的方式

1、继承Thread类

步骤：

- (1) 编写线程类，去继承Thread类
- (2) 重写public void run(){}
- (3) 创建线程对象
- (4) 调用start()

```
class MyThread extends Thread {
    public void run(){
        //...
    }
}

class Test{
    public static void main(String[] args){
        MyThread my = new MyThread();
        my.start();//有名字的线程对象启动

        new MyThread().start();//匿名线程对象启动

        //匿名内部类的匿名对象启动
        new Thread(){
            public void run(){
                //...
            }
        }.start();

        //匿名内部类，但是通过父类的变量多态引用，启动线程
        Thread t = new Thread(){
            public void run(){
                //...
            }
        };
        t.start();
    }
}
```

2、实现Runnable接口

步骤：

- (1) 编写线程类，实现Runnable接口
- (2) 重写public void run(){}
- (3) 创建线程对象

(4) 借助Thread类的对象启动线程

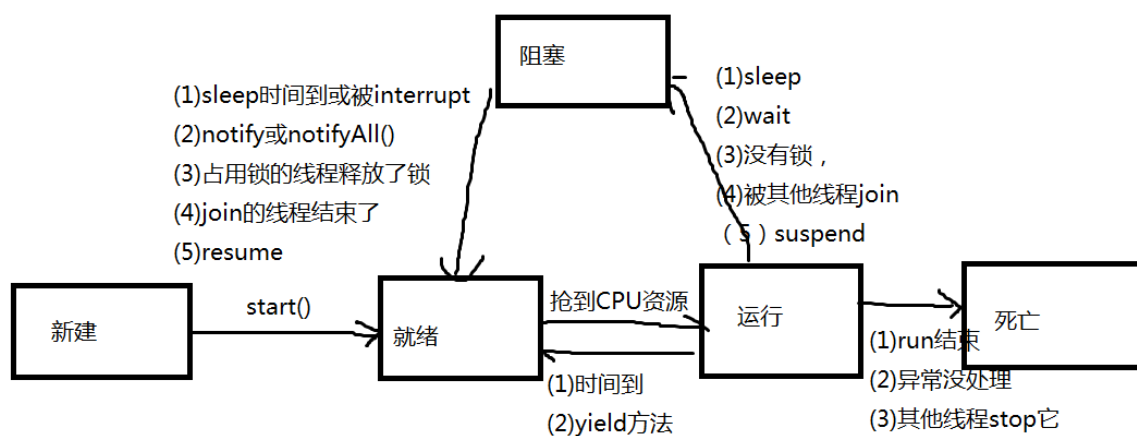
```
class MyRunnable implements Runnable{
    public void run(){
        //...
    }
}

class Test {
    public static void main(String[] args){
        MyRunnable my = new MyRunnable();
        Thread t1 = new Thread(my);
        Thread t2 = new Thread(my);
        t1.start();
        t2.start();

        //两个匿名对象
        new Thread(new MyRunnable()).start();

        //匿名内部类的匿名对象作为实参直接传给Thread的构造器
        new Thread(new Runnable(){
            public void run(){
                //...
            }
        }).start();
    }
}
```

10.3 线程的生命周期



10.4 Thread的相关API

1、构造器

- Thread()
- Thread(String name)
- Thread(Runnable target)
- Thread(Runnable target, String name)

2、其他方法

- (1) public void run()
- (2) public void start()
- (3) 获取当前线程对象：Thread.currentThread()
- (4) 获取当前线程的名称：getName()
- (5) 设置或获取线程的优先级：set/getPriority()

优先级的范围：[1,10]，Thread类中有三个常量：MAX_PRIORITY(10)，MIN_PRIORITY(1)，NORM_PRIORITY(5)

优先级只是影响概率。

- (6) 线程休眠：Thread.sleep(毫秒)
- (7) 打断线程：interrupt()
- (8) 暂停当前线程：Thread.yield()
- (9) 线程要加塞：join()

xx.join()这句代码写在哪个线程体中，哪个线程被加塞，和其他线程无关。

- (10) 判断线程是否已启动但未终止：isAlive()

10.5 关键字：volatile

volatile：易变，不稳定，不一定什么时候会变

修饰：成员变量

作用：当多个线程同时去访问的某个成员变量时，而且是频繁的访问，再多次访问时，发现它的值没有修改，Java执行引擎就会对这个成员变量的值进行缓存。一旦缓存之后，这个时候如果有一个线程把这个成员变量的值修改了，Java执行引擎还是从缓存中读取，导致这个值不是最新的。如果不希望Java执行引擎把这个成员变的值缓存起来，那么就可以在成员变量的前面加volatile，每次用到这个成员变量时，都是从主存中读取。

10.6 关键字：synchronized（同步）

1、什么情况下会发生线程安全问题？

- (1) 多个线程

(2) 共享数据

(3) 多个线程的线程体中，多条语句再操作这个共享数据时

2、如何解决线程安全问题？同步锁

形式一：同步代码块

形式二：同步方法

3、同步代码块

```
synchronized(锁对象){  
    //一次任务代码，这其中的代码，在执行过程中，不希望其他线程插一脚  
}
```

锁对象：

(1) 任意类型的对象

(2) 确保使用共享数据的这几个线程，使用同一个锁对象

4、同步方法

```
synchronized 【修饰符】 返回值类型 方法名(【形参列表】) throws 异常列表{  
    //同一时间，只能有一个线程能进来运行  
}
```

锁对象：

(1) 非静态方法：this（谨慎）

(2) 静态方法：当前类的Class对象

10.7 线程通信

1、为了解决“生产者与消费者问题”。

当一些线程负责往“数据缓冲区”放数据，另一个线程负责从“数据缓冲区”取数据。

问题1：生产者线程与消费者线程使用同一个数据缓冲区，就是共享数据，那么要考虑同步

问题2：当数据缓冲区满的时候，生产者线程需要wait()，当消费者消费了数据后，需要notify或notifyAll

当数据缓冲区空的时候，消费者线程需要wait()，当生产者生产了数据后，需要notify或notifyAll

2、java.lang.Object类中声明了：

(1) wait()：必须由“同步锁”对象调用

(2) notify()和notifyAll()：必须由“同步锁”对象调用

3、面试题：sleep()和wait的区别

- (1) sleep()不释放锁，wait()释放锁
- (2) sleep()在Thread类中声明的，wait()在Object类中声明
- (3) sleep()是静态方法，是Thread.sleep()
wait()是非静态方法，必须由“同步锁”对象调用
- (4) sleep()方法导致当前线程进入阻塞状态后，当时间到或interrupt()醒来
wait()方法导致当前线程进入阻塞状态后，由notify或notifyAll()

4、哪些操作会释放锁？

- (1) 同步代码块或同步方法正常执行完一次自动释放锁
- (2) 同步代码块或同步方法遇到return等提前结束
- (3) wait()

5、不释放锁

- (1) sleep()
- (2) yield()
- (3) suspend()

第十一章 常用类

11.1 包装类

11.1.1 包装类

当要使用只针对对象设计的API或新特性（例如泛型），那么基本数据类型的数据就需要用包装类来包装。

序号	基本数据类型	包装类
1	byte	Byte
2	short	Short
3	int	Integer
4	long	Long
5	float	Float
6	double	Double
7	char	Character
8	boolean	Boolean
9	void	Void

11.1.2 装箱与拆箱

JDK1.5之后，可以自动装箱与拆箱。

注意：只能与自己对应的类型之间才能实现自动装箱与拆箱。

```
Integer i = 1;
Double d = 1; //错误的, 1是int类型
```

装箱：把基本数据类型转为包装类对象。

转为包装类的对象，是为了使用专门为对象设计的API和特性

拆箱：把包装类对象拆为基本数据类型。

转为基本数据类型，一般是因为需要运算，Java中的大多数运算符是为基本数据类型设计的。比较、算术等

总结：对象（引用数据类型）能用的运算符有哪些？

- (1) instanceof
- (2) =: 赋值运算符
- (3) ==和!=: 用于比较地址，但是要求左右两边对象的类型一致或者是有父子类继承关系。
- (4) 对于字符串这一种特殊的对象，支持“+”，表示拼接。

11.1.3 包装类的一些API

1、基本数据类型和字符串之间的转换

- (1) 把基本数据类型转为字符串

```
int a = 10;
//String str = a;//错误的
//方式一:
String str = a + "";
//方式二:
String str = String.valueOf(a);
```

(2) 把字符串转为基本数据类型

```
int a = Integer.parseInt("整数的字符串");
double a = Double.parseDouble("小数的字符串");
boolean b = Boolean.parseBoolean("true或false");
```

2、数据类型的最大最小值

```
Integer.MAX_VALUE和Integer.MIN_VALUE
Long.MAX_VALUE和Long.MIN_VALUE
Double.MAX_VALUE和Double.MIN_VALUE
```

3、转大小写

```
Character.toUpperCase('x');
Character.toLowerCase('X');
```

4、转进制

```
Integer.toBinaryString(int i)
Integer.toHexString(int i)
Integer.toOctalString(int i)
```

11.1.4 包装类对象的缓存问题

包装类	缓存对象
Byte	-128~127
Short	-128~127
Integer	-128~127
Long	-128~127
Float	没有
Double	没有
Character	0~127
Boolean	true和false

```
Integer i = 1;
Integer j = 1;
System.out.println(i == j); //true

Integer i = 128;
Integer j = 128;
System.out.println(i == j); //false

Integer i = new Integer(1); //新new的在堆中
Integer j = 1; //这个用的是缓冲的常量对象，在方法区
System.out.println(i == j); //false

Integer i = new Integer(1); //新new的在堆中
Integer j = new Integer(1); //另一个新new的在堆中
System.out.println(i == j); //false

Integer i = new Integer(1);
int j = 1;
System.out.println(i == j); //true, 凡是和基本数据类型比较，都会先拆箱，按照基本数据类型的规则比较
```

11.2 字符串

11.2.1 字符串的特点

- 1、字符串String类型本身是final声明的，意味着我们不能继承String。
- 2、字符串的对象也是不可变对象，意味着一旦进行修改，就会产生新对象

我们修改了字符串后，如果想要获得新的内容，必须重新接受。

如果程序中涉及到大量的字符串的修改操作，那么此时的时空消耗比较高。可能需要考虑使用StringBuilder或StringBuffer。

3、String对象内部是用字符数组进行保存的

JDK1.9之前有一个char[] value数组，JDK1.9之后byte[]数组

4、String类中这个char[] values数组也是final修饰的，意味着这个数组不可变，然后它是private修饰，外部不能直接操作它，String类型提供的所有的都是用新对象来表示修改后内容的，所以保证了String对象的不可变。

5、就因为字符串对象设计为不可变，那么所以字符串有常量池来保存很多常量对象

常量池在方法区。

如果细致的划分：

(1) JDK1.6及其之前：方法区

(2) JDK1.7：堆

(3) JDK1.8：元空间

11.2.2 字符串对象的比较

1、==：比较是对象的地址

只有两个字符串变量都是指向字符串的常量对象时，才会返回true

```
String str1 = "hello";
String str2 = "hello";
str1 == str2 //true
```

2、equals：比较是对象的内容，因为String类型重写equals，区分大小写

只要两个字符串的字符内容相同，就会返回true

```
String str1 = new String("hello");
String str2 = new String("hello");
str1.equals(strs) //true
```

3、equalsIgnoreCase：比较的是对象的内容，不区分大小写

```
String str1 = new String("hello");
String str2 = new String("HELLO");
str1.equalsIgnoreCase(strs) //true
```

4、compareTo：String类型重写了Comparable接口的抽象方法，自然排序，按照字符的Unicode编码值进行比较大小的，严格区分大小写

```
String str1 = "hello";
String str2 = "world";
str1.compareTo(str2) //小于0的值
```

5、compareToIgnoreCase: 不区分大小写, 其他按照字符的Unicode编码值进行比较大小

```
String str1 = new String("hello");
String str2 = new String("HELLO");
str1.compareToIgnoreCase(str2) //等于0
```

11.2.3 空字符的比较

1、哪些是空字符串

```
String str1 = "";
String str2 = new String();
String str3 = new String("");
```

空字符串: 长度为0

2、如何判断某个字符串是否是空字符串

```
if("").equals(str)

if(str!=null && str.isEmpty())

if(str!=null && str.equals(""))

if(str!=null && str.length()==0)
```

11.2.4 字符串的对象的个数

1、字符串常量对象

```
String str1 = "hello";//1个, 在常量池中
```

2、字符串的普通对象

```
String str2 = new String();
String str22 = new String("");
//两个对象, 一个是常量池中的空字符串对象, 一个是堆中的空字符串对象
```

3、字符串的普通对象和常量对象一起

```
String str3 = new String("hello");
//str3首先指向堆中的一个字符串对象，然后堆中字符串的value数组指向常量池中常量对象的value数组
```

11.2.5 字符串拼接结果

原则：

- (1) 常量+常量：结果是常量池
- (2) 常量与变量 或 变量与变量：结果是堆
- (3) 拼接后调用intern方法：结果在常量池

```
@Test
public void test06(){
    String s1 = "hello";
    String s2 = "world";
    String s3 = "helloworld";

    String s4 = (s1 + "world").intern();//把拼接的结果放到常量池中
    String s5 = (s1 + s2).intern();

    System.out.println(s3 == s4);//true
    System.out.println(s3 == s5);//true
}

@Test
public void test05(){
    final String s1 = "hello";
    final String s2 = "world";
    String s3 = "helloworld";

    String s4 = s1 + "world";//s4字符串内容也helloworld, s1是常量, "world"常量, 常量
+ 常量 结果在常量池中
    String s5 = s1 + s2;//s5字符串内容也helloworld, s1和s2都是常量, 常量+ 常量 结果在
常量池中
    String s6 = "hello" + "world";//常量+ 常量 结果在常量池中, 因为编译期间就可以确定
结果

    System.out.println(s3 == s4);//true
    System.out.println(s3 == s5);//true
    System.out.println(s3 == s6);//true
}

@Test
public void test04(){
    String s1 = "hello";
    String s2 = "world";
```



```
String s3 = "helloworld";

String s4 = s1 + "world";//s4字符串内容也helloworld, s1是变量, "world"常量, 变量
+ 常量的结果在堆中
String s5 = s1 + s2;//s5字符串内容也helloworld, s1和s2都是变量, 变量 + 变量的结果
在堆中
String s6 = "hello" + "world";//常量+ 常量 结果在常量池中, 因为编译期间就可以确定
结果

System.out.println(s3 == s4);//false
System.out.println(s3 == s5);//false
System.out.println(s3 == s6);//true
}
```

11.2.6 字符串的API

- (1) boolean isEmpty()
- (2) int length()
- (3) String concat(xx): 拼接, 等价于+
- (4) boolean contains(xx)
- (5) int indexOf(): 从前往后找, 要是没有返回-1
- (6) int lastIndexOf(): 从后往前找, 要是没有返回-1
- (7) char charAt(index)
- (8) new String(char[]) 或 new String(char[] ,int, int)
- (9) char[] toCharArray()
- (10) byte[] getBytes(): 编码, 把字符串变为字节数组, 按照平台默认的字符编码进行编码
byte[] getBytes(字符编码方式): 按照指定的编码方式进行编码
- (11) new String(byte[]) 或 new String(byte[], int, int): 解码, 按照平台默认的字符编码进行解码
new String(byte[], 字符编码方式) 或 new String(byte[], int, int, 字符编码方式): 解码, 按照指定的编码方式进行解码
- (12) String substring(int begin): 从[begin]开始到最后
String substring(int begin,int end): 从[begin, end)
- (13) boolean matches(正则表达式)
- (14) String replace(xx,xx): 不支持正则
String replaceFirst(正则, value): 替换第一个匹配部分
String replaceAll(正则, value): 替换所有匹配部分

(15) `String[] split(正则)`: 按照某种规则进行拆分

(16) `boolean startsWith(xx)`: 是否以xx开头

`boolean endsWith(xx)`: 是否以xx结尾

(17) `String trim()`: 去掉前后空白符, 字符串中间的空白符不会去掉

(18) `String toUpperCase()`: 转大写

(19) `String toLowerCase()`: 转小写

面试题: 字符串的length和数组的length有什么不同?

字符串的length(), 数组的length属性

11.3 可变字符序列

1、可变字符序列: `StringBuilder`和`StringBuffer`

`StringBuffer`: 老的, 线程安全的 (因为它的方法有synchronized修饰)

`StringBuilder`: 线程不安全的

2、面试题: `String`和`StringBuilder`、`StringBuffer`的区别?

`String`: 不可变对象, 不可变字符序列

`StringBuilder`、`StringBuffer`: 可变字符序列

3、常用的API, `StringBuilder`、`StringBuffer`的API是完全一致的

(1) `append(xx)`: 拼接, 追加

(2) `insert(int index, xx)`: 插入

(3) `delete(int start, int end)`

`deleteCharAt(int index)`

(4) `set(int index, xx)`

(5) `reverse()`: 反转

.... 替换、截取、查找...

11.4 和数学相关的

1、`java.lang.Math`类

(1) `sqrt()`: 求平方根

(2) `pow(x,y)`: 求x的y次方

(3) random(): 返回[0,1)范围的小数

(4) max(x,y): 找x,y最大值

min(x,y): 找最小值

(5) round(x): 四舍五入

ceil(x): 进一

floor(x): 退一

.....

2、java.math包

BigInteger: 大整数

BigDecimal: 大小数

运算通过方法完成: add(),subtract(),multiply(),divide()....

11.5 日期时间API

11.5.1 JDK1.8之前

1、java.util.Date

new Date(): 当前系统时间

long getTime(): 返回该日期时间对象距离1970-1-1 0.0.0 0毫秒之间的毫秒值

new Date(long 毫秒): 把该毫秒值换算成日期时间对象

2、java.util.Calendar:

(1) getInstance(): 得到Calendar的镀锡

(2) get(常量)

3、java.text.SimpleDateFormat: 日期时间的格式化

y: 表示年

M: 月

d: 天

H: 小时, 24小时制

h: 小时, 12小时制

m: 分

s: 秒

S: 毫秒

E: 星期

D: 年当中的天数

```
@Test
public void test10() throws ParseException{
    String str = "2019年06月06日 16时03分14秒 545毫秒 星期四 +0800";
    SimpleDateFormat sf = new SimpleDateFormat("yyyy年MM月dd日 HH时mm分ss秒 SSS毫
秒 E Z");
    Date d = sf.parse(str);
    System.out.println(d);
}

@Test
public void test9(){
    Date d = new Date();

    SimpleDateFormat sf = new SimpleDateFormat("yyyy年MM月dd日 HH时mm分ss秒 SSS毫
秒 E Z");
    //把Date日期转成字符串, 按照指定的格式转
    String str = sf.format(d);
    System.out.println(str);
}

@Test
public void test8(){
    String[] all = TimeZone.getAvailableIDs();
    for (int i = 0; i < all.length; i++) {
        System.out.println(all[i]);
    }
}

@Test
public void test7(){
    TimeZone t = TimeZone.getTimeZone("America/Los_Angeles");

    //getInstance(TimeZone zone)
    Calendar c = Calendar.getInstance(t);
    System.out.println(c);
}

@Test
public void test6(){
    Calendar c = Calendar.getInstance();
    System.out.println(c);

    int year = c.get(Calendar.YEAR);
    System.out.println(year);
}
```

```

    int month = c.get(Calendar.MONTH)+1;
    System.out.println(month);

    //...
}

@Test
public void test5(){
    long time = Long.MAX_VALUE;
    Date d = new Date(time);
    System.out.println(d);
}

@Test
public void test4(){
    long time = 1559807047979L;
    Date d = new Date(time);
    System.out.println(d);
}

@Test
public void test3(){
    Date d = new Date();
    long time = d.getTime();
    System.out.println(time);//1559807047979
}

@Test
public void test2(){
    long time = System.currentTimeMillis();
    System.out.println(time);//1559806982971
    //当前系统时间距离1970-1-1 0:0:0 0毫秒的时间差，毫秒为单位
}

@Test
public void test1(){
    Date d = new Date();
    System.out.println(d);
}

```

11.5.2 JDK1.8之后

java.time及其子包中。

1、LocalDate、LocalTime、LocalDateTime

(1) now(): 获取系统日期或时间

(2) of(xxx): 或者指定的日期或时间

(3) 运算: 运算后得到新对象, 需要重新接受

plusXxx(): 在当前日期或时间对象上加xx

minusXxx(): 在当前日期或时间对象上减xx

方法	描述
now() / now(Zoned zone)	静态方法, 根据当前时间创建对象/指定时区的对象
of()	静态方法, 根据指定日期/时间创建对象
getDayOfMonth()/getDayOfYear()	获得月份天数(1-31) /获得年份天数(1-366)
getDayOfWeek()	获得星期几(返回一个 DayOfWeek 枚举值)
getMonth()	获得月份, 返回一个 Month 枚举值
getMonthValue() / getYear()	获得月份(1-12) /获得年份
getHours()/getMinute()/getSecond()	获得当前对象对应的小时、分钟、秒
withDayOfMonth()/withDayOfYear()/withMonth()/withYear()	将月份天数、年份天数、月份、年份修改为指定的值并返回新的对象
with(TemporalAdjuster t)	将当前日期时间设置为校对器指定的日期时间
plusDays(), plusWeeks(), plusMonths(), plusYears(),plusHours()	向当前对象添加几天、几周、几个月、几年、几小时
minusMonths() / minusWeeks()/minusDays()/minusYears()/minusHours()	从当前对象减去几月、几周、几天、几年、几小时
plus(TemporalAmount t)/minus(TemporalAmount t)	添加或减少一个 Duration 或 Period
isBefore()/isAfter()	比较两个 LocalDate
isLeapYear()	判断是否是闰年 (在LocalDate类中声明)
format(DateTimeFormatter t)	格式化本地日期、时间, 返回一个字符串
parse(CharSequence text)	将指定格式的字符串解析为日期、时间

2、DateTimeFormatter: 日期时间格式化

该类提供了三种格式化方法:

预定义的标准格式。如: ISO_DATE_TIME;ISO_DATE

本地化相关的格式。如: ofLocalizedDate(FormatStyle.MEDIUM)

自定义的格式。如: ofPattern("yyyy-MM-dd hh:mm:ss")

```
@Test
public void test10(){
    LocalDateTime now = LocalDateTime.now();
```

```

//    DateTimeFormatter df =
DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG);//2019年6月6日 下午04时
40分03秒

    DateTimeFormatter df =
DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT);//19-6-6 下午4:40
    String str = df.format(now);
    System.out.println(str);
}
@Test
public void test9(){
    LocalDateTime now = LocalDateTime.now();

    DateTimeFormatter df = DateTimeFormatter.ISO_DATE_TIME;//2019-06-
06T16:38:23.756
    String str = df.format(now);
    System.out.println(str);
}

@Test
public void test8(){
    LocalDateTime now = LocalDateTime.now();

    DateTimeFormatter df = DateTimeFormatter.ofPattern("yyyy年MM月dd日 HH时mm分
ss秒 SSS毫秒 E 是这一年的D天");
    String str = df.format(now);
    System.out.println(str);
}

@Test
public void test7(){
    LocalDate now = LocalDate.now();
    LocalDate before = now.minusDays(100);
    System.out.println(before);//2019-02-26
}

@Test
public void test06(){
    LocalDate lai = LocalDate.of(2019, 5, 13);
    LocalDate go = lai.plusDays(160);
    System.out.println(go);//2019-10-20
}

@Test
public void test05(){
    LocalDate lai = LocalDate.of(2019, 5, 13);
    System.out.println(lai.getDayOfYear());
}

```

```
@Test
public void test04(){
    LocalDate lai = LocalDate.of(2019, 5, 13);
    System.out.println(lai);
}

@Test
public void test03(){
    LocalDateTime now = LocalDateTime.now();
    System.out.println(now);
}

@Test
public void test02(){
    LocalTime now = LocalTime.now();
    System.out.println(now);
}

@Test
public void test01(){
    LocalDate now = LocalDate.now();
    System.out.println(now);
}
```

第十二章 集合

12.1 概念

数据结构：存储数据的某种结构

(1) 底层的物理结构

①数组：开辟连续的存储空间，每一个元素使用[下标]进行区别

②链式：不需要开辟连续的存储空间，但是需要“结点”来包装要存储的数据，结点包含两部分内容：

A、数据

B、记录其他结点的地址，例如：next, pre, left, right, parent等

(2) 表现出来的逻辑结构：动态数组、单向链表、双向链表、队列、栈、二叉树、哈希表、图等

12.2 手动实现一些逻辑结构

1、动态数组

包含：

(1) 内部使用一个数组，用来存储数据

(2) 内部使用一个total，记录实际存储的元素的个数


```

public class MyArrayList {
    //为什么使用Object, 因为只是说这个容器是用来装对象的, 但是不知道用来装什么对象。
    private Object[] data;
    private int total;

    public MyArrayList(){
        data = new Object[5];
    }

    //添加一个元素
    public void add(Object obj){
        //检查是否需要扩容
        checkCapacity();
        data[total++] = obj;
    }

    private void checkCapacity() {
        //如果data满了, 就扩容为原来的2倍
        if(total >= data.length){
            data = Arrays.copyOf(data, data.length*2);
        }
    }

    //返回实际元素的个数
    public int size(){
        return total;
    }

    //返回数组的实际容量
    public int capacity(){
        return data.length;
    }

    //获取[index]位置的元素
    public Object get(int index){
        //校验index的合理性范围
        checkIndex(index);
        return data[index];
    }

    private void checkIndex(int index) {
        if(index<0 || index>=total){
            throw new RuntimeException(index+"对应位置的元素不存在");
        }
    }

    //替换[index]位置的元素
    public void set(int index, Object value){

```

```

//校验index的合理性范围
checkIndex(index);

data[index] = value;
}

//在[index]位置插入一个元素value
public void insert(int index, Object value){
    /*
     * (1)考虑下标的合理性
     * (2)总长度是否够
     * (3)[index]以及后面的元素往后移动, 把[index]位置腾出来
     * (4)data[index]=value 放入新元素
     * (5)total++ 有效元素的个数增加
     */

    //(1)考虑下标的合理性: 校验index的合理性范围
    checkIndex(index);

    //(2)总长度是否够: 检查是否需要扩容
    checkCapacity();

    //(3)[index]以及后面的元素往后移动, 把[index]位置腾出来
    /*
     * 假设total = 5, data.length= 10, index= 1
     * 有效元素的下标[0,4]
     * 移动: [1]->[2],[2]->[3],[3]->[4],[4]->[5]
     * 移动元素的个数: total-index
     */
    System.arraycopy(data, index, data, index+1, total-index);

    //(4)data[index]=value 放入新元素
    data[index] = value;

    //(5)total++ 有效元素的个数增加
    total++;
}

//返回所有实际存储的元素
public Object[] getAll(){
    //返回total个
    return Arrays.copyOf(data, total);
}

//删除[index]位置的元素
public void remove(int index){
    /*
     * (1)校验index的合理性范围
     * (2)移动元素, 把[index+1]以及后面的元素往前移动

```

```

    * (3)把data[total-1]=null 让垃圾回收器尽快回收
    * (4)总元素个数减少 total--
    */

// (1)考虑下标的合理性：校验index的合理性范围
checkIndex(index);

// (2)移动元素，把[index+1]以及后面的元素往前移动
/*
    * 假设total=8, data.length=10, index = 3
    * 有效元素的范围[0,7]
    * 移动: [4]->[3],[5]->[4],[6]->[5],[7]->[6]
    * 移动了4个: total-index-1
    */
System.arraycopy(data, index+1, data, index, total-index-1);

// (3)把data[total-1]=null 让垃圾回收器尽快回收
data[total-1] = null;

// (4)总元素个数减少 total--
total--;
}

// 查询某个元素的下标
public int indexOf(Object obj){
    if(obj == null){
        for (int i = 0; i < total; i++) {
            if(data[i] == null){//等价于 if(data[i] == obj)
                return i;
            }
        }
    }else{
        for (int i = 0; i < data.length; i++) {
            if(obj.equals(data[i])){
                return i;
            }
        }
    }
    return -1;
}

// 删除数组中的某个元素
// 如果有重复的，只删除第一个
public void remove(Object obj){
    /*
    * (1)先查询obj的[index]
    * (2)如果存在，就调用remove(index)删除就可以
    */

```

```

        //(1)先查询obj的[index]
        int index = indexOf(obj);

        if(index != -1){
            remove(index);
        }
        //不存在, 可以什么也不做
        //不存在, 也可以抛异常
        //throw new RuntimeException(obj + "不存在");
    }

    public void set(Object old, Object value){
        /*
         * (1)查询old的[index]
         * (2)如果存在, 就调用set(index, value)
         */

        //(1)查询old的[index]
        int index = indexOf(old);
        if(index!=-1){
            set(index, value);
        }

        //不存在, 可以什么也不做
    }
}

```

2、单向链表

包含：

(1) 包含一个Node类型的成员变量first：用来记录第一个结点的地址

如果这个链表是空的，还没有任何结点，那么first是null。

最后一个结点的特征：就是它的next是null

(2) 内部使用一个total，记录实际存储的元素的个数

(3) 使用了一个内部类Node

```

private class Node{
    Object data;
    Node next;
}

```

```

public class SingleLinkedList {
    //这里不需要数组，不需要其他的复杂的结构，我只要记录单向链表的“头”结点
    private Node first;//first中记录的是第一个结点的地址
}

```

`private int total;` //这里我记录total是为了后面处理的方便，例如：当用户获取链表有效元素的个数时，不用现数，而是直接返回total等

/*
* 内部类，因为这种Node结点的类型，在别的地方没有用，只在单向链表中，用于存储和表示它的结点关系。

* 因为我这里涉及为内部类型。

*/

```
private class Node{  
    Object data; //因为数据可以是任意类型的对象，所以设计为Object  
    Node next; //因为next中记录的下一个结点的地址，因此类型是结点类型  
    //这里data,next没有私有化，是希望在外部类中可以不需要get/set，而是直接“结点对象.data”，“结点对象.next”使用
```

```
    Node(Object data, Node next){  
        this.data = data;  
        this.next = next;  
    }  
}
```

```
public void add(Object obj){
```

```
    /*
```

```
    * (1)把obj的数据，包装成一个Node类型结点对象
```

```
    * (2)把新结点“链接”当前链表的最后
```

```
    * ①当前新结点是第一个结点
```

```
    * 如何判断是否是第一个    if(first==null)说明暂时还没有第一个
```

```
    * ②先找到目前的最后一个，把新结点链接到它的next中
```

```
    * 如何判断是否是最后一个    if(某个结点.next == null)说明这个结点是最后一个
```

```
    */
```

```
//    (1)把obj的数据，包装成一个Node类型结点对象
```

```
//这里新结点的next赋值为null，表示新结点是最后一个结点
```

```
Node newNode = new Node(obj, null);
```

```
//①当前新结点是第一个结点
```

```
if(first == null){
```

```
    //说明newNode是第一个
```

```
    first = newNode;
```

```
}else{
```

```
    //②先找到目前的最后一个，把新结点链接到它的next中
```

```
    Node node = first;
```

```
    while(node.next != null){
```

```
        node = node.next;
```

```
    }
```

```
    //退出循环时node指向最后一个结点
```

```
    //把新结点链接到它的next中
```

```
    node.next = newNode;
```

```
}
```

```
total++;
```

```

}

public int size(){
    return total;
}

public Object[] getAll(){
    //(1)创建一个数组，长度为total
    Object[] all = new Object[total];

    //(2)把单向链表的每一个结点中的data，拿过来放到all数组中
    Node node = first;
    for (int i = 0; i < total; i++) {
//        all[i] = 结点.data;
        all[i] = node.data;
        //然后node指向下一个
        node = node.next;
    }

    //(3)返回数组
    return all;
}

public void remove(Object obj){
    if(obj == null){
        //(1)先考虑是否是第一个
        if(first!=null){//链表非空

            //要删除的结点正好是第一个结点
            if(first.data == null){
                //让第一个结点指向它的下一个
                first = first.next;
                total--;
                return;
            }

            //要删除的不是第一个结点
            Node node = first.next;//第二个结点
            Node last = first;
            while(node.next!=null){//这里不包括最后一个，因为node.next==null，不进入循环，而node.next==null是最后一个
                if(node.data == null){
                    last.next = node.next;
                    total--;
                    return;
                }
                last = node;
                node = node.next;
            }
        }
    }
}

```

```

        //单独判断最后一个是否是要删除的结点
        if(node.data == null){
            //要删除的是最后一个结点
            last.next = null;
            total--;
            return;
        }
    }
}
}else{
    //(1)先考虑是否是第一个
    if(first!=null){//链表非空

        //要删除的结点正好是第一个结点
        if(obj.equals(first.data)){
            //让第一个结点指向它的下一个
            first = first.next;
            total--;
            return;
        }

        //要删除的不是第一个结点
        Node node = first.next;//第二个结点
        Node last = first;
        while(node.next!=null){//这里不包括最后一个，因为node.next==null，不进入循环，而node.next==null是最后一个
            if(obj.equals(node.data)){
                last.next = node.next;
                total--;
                return;
            }
            last = node;
            node = node.next;
        }

        //单独判断最后一个是否是要删除的结点
        if(obj.equals(node.data)){
            //要删除的是最后一个结点
            last.next = null;
            total--;
            return;
        }
    }
}
}

public int indexOf(Object obj){
    if(obj == null){
        Node node = first;

```

```

        for (int i = 0; i < total; i++) {
            if(node.data == null){
                return i;
            }
            node = node.next;
        }
    }else{
        Node node = first;
        for (int i = 0; i < total; i++) {
            if(obj.equals(node.data)){
                return i;
            }
            node = node.next;
        }
    }
    return -1;
}
}

```

12.3 Collection

因为集合的类型很多，那么我们把它们称为集合框架。

集合框架分为两个家族：Collection（一组对象）和Map（一组映射关系、一组键值对）

12.3.1 Collection

Collection是代表一种对象的集合。它是Collection系列的根接口。

它们虽然：有些可能是有序的，有些可能是无序的，有些可能可以重复的，有些不能重复的，但是它们有共同的操作规范，因此这些操作的规范就抽象为了Collection接口。

常用方法：

- (1) boolean add(Object obj)：添加一个
- (2) boolean addAll (Collection c) ：添加多个
- (3) boolean remove(Object obj)：删除一个
- (4) boolean removeAll(Collection c)：删除多个
- (5) boolean contains(Object c)：是否包含某个
- (6) boolean containsAll(Collection c)：是否包含所有
- (7) boolean isEmpty()：是否为空
- (8) int size()：获取元素个数
- (9) void clear()：清空集合

(10) Object[] toArray(): 获取所有元素

(11) Iterator iterator(): 获取遍历当前集合的迭代器对象

(12) retainAll(Collection c): 求当前集合与c集合的交集

12.3.2 Collection系列的集合的遍历

1、明确使用Iterator迭代器

```
Collection c = ....;

Iterator iter = c.iterator();
while(iter.hasNext()){
    Object obj = iter.next();
    //...
}
```

Iterator 接口的方法:

(1) boolean hasNext()

(2) Object next()

(3) void remove()

2、foreach

```
Collection c = ....;

for(Object obj : c){
    //...
}
```

什么样的集合（容器）能够使用foreach遍历？

(1) 数组:

(2) 实现了java.lang.Iterable接口

这个接口有一个抽象方法: Iterator iterator()

Iterator也是一个接口，它的实现类，通常在集合（容器）类中用内部类实现。并在iterator()的方法中创建它的对象。

```
public class MyArrayList implements Iterable{
    //为什么使用Object, 因为只是说这个容器是用来装对象的, 但是不知道用来装什么对象。
    private Object[] data;
    private int total;

    //其他代码省略....
}
```

```

@Override
public Iterator iterator() {
    return new MyItr();
}

private class MyItr implements Iterator{
    private int cursor;//游标

    @Override
    public boolean hasNext() {
        return cursor!=total;
    }

    @Override
    public Object next() {
        return data[cursor++];
    }
}
}

```

思考：如果遍历数组，什么情况下选用foreach，什么情况下选用for循环？

当如果你的操作中涉及到[下标]操作时，用for最好。

当你只是查看元素的内容，那么选foreach更简洁一些。

思考：如果遍历Collection系列集合，什么情况下选用foreach，是否能选用for循环？

首先考虑使用foreach，如果该集合也有索引信息的话，也可以通过for来操作，如果没有下标的信息，就不要用for。即，如果该集合的物理结构是数组的，那么可以用for，如果物理结构是链式，那么使用下标操作效率很低。

思考：如果遍历Collection系列集合，什么情况下选用foreach，什么情况下使用Iterator？

如果只是查看集合的元素，使用foreach，代码会更简洁。

但是如果要涉及到在遍历集合的同时根据某种条件要删除元素等操作，那么选用Iterator。

12.4 List

12.4.1 List概述

List：是Collection的子接口。

List系列的集合：有序的、可重复的

List系列的常用集合：ArrayList、Vector、LinkedList、Stack

12.4.2 List的API

常用方法：

- (1) boolean add(Object obj): 添加一个
- (2) boolean addAll (Collection c) : 添加多个
- (3) void add(int index, Object obj): 添加一个, 指定位置添加
- (4) void addAll(int index, Collection c) : 添加多个
- (5) boolean remove(Object obj): 删除一个
- (6) Object remove(int index): 删除指定位置的元素, 并返回刚刚删除的元素
- (7) boolean removeAll(Collection c) : 删除多个
- (8) boolean contains(Object c): 是否包含某个
- (9) boolean containsAll(Collection c): 是否包含所有
- (10) boolean isEmpty(): 是否为空
- (11) int size(): 获取元素个数
- (12) void clear(): 清空集合
- (13) Object[] toArray(): 获取所有元素
- (14) Iterator iterator(): 获取遍历当前集合的迭代器对象
- (15) retainAll(Collection c): 求当前集合与c集合的交集
- (16) ListIterator listIterator(): 获取遍历当前集合的迭代器对象, 这个迭代器可以往前、往后遍历
- (17) ListIterator listIterator(int index): 从[index]位置开始, 往前或往后遍历
- (18) Object get(int index): 返回index位置的元素
- (19) List subList(int start, int end): 截取[start,end)部分的子列表

12.4.3 ListIterator 接口

Iterator 接口的方法：

- (1) boolean hasNext()
- (2) Object next()
- (3) void remove()

ListIterator 是 Iterator子接口：增加了如下方法

- (4) void add(Object obj)

- (5) void set(Object obj)
- (6) boolean hasPrevious()
- (7) Object previous()
- (8) int nextIndex()
- (9) int previousIndex()

12.4.4 List的实现类们的区别

ArrayList、Vector、LinkedList、Stack

- (1) ArrayList、Vector：都是动态数组

Vector是最早版本的动态数组，线程安全的，默认扩容机制是2倍，支持旧版的迭代器Enumeration

ArrayList是后增的动态数组，线程不安全的，默认扩容机制是1.5倍

- (2) 动态数组与LinkedList的区别

动态数组：底层物理结构是数组

优点：根据[下标]访问的速度很快

缺点：需要开辟连续的存储空间，而且需要扩容，移动元素等操作

LinkedList：底层物理结构是双向链表

优点：在增加、删除元素时，不需要移动元素，只需要修改前后元素的引用关系

缺点：我们查找元素时，只能从first或last开始查找

- (3) Stack：栈

是Vector的子类。比Vector多了几个方法，能够表现出“先进后出或后进先出”的特点。

①Object peek()：访问栈顶元素

②Object pop()：弹出栈顶元素

③push()：把元素压入栈顶

- (4) LinkedList可以作为很多种数据结构使用

单链表：只关注next就可以

队列：先进先出，找对应的方法

双端队列(JDK1.6加入)：两头都可以进出，找对应的方法

栈：先进后出，找对应的方法

建议：虽然LinkedList是支持对索引进行操作，因为它实现List接口的所有方法，但是我们不太建议调用类似这样的方法，因为效率比较低。

(1) Vector

```
//synchronized意味着线程安全的
public synchronized boolean add(E e) {
    modCount++;
    //看是否需要扩容
    ensureCapacityHelper(elementCount + 1);
    //把新的元素存入[elementCount], 存入后, elementCount元素的个数增1
    elementData[elementCount++] = e;
    return true;
}

private void ensureCapacityHelper(int minCapacity) {
    // overflow-conscious code
    //看是否超过了当前数组的容量
    if (minCapacity - elementData.length > 0)
        grow(minCapacity); //扩容
}

private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length; //获取目前数组的长度
    //如果capacityIncrement增量是0, 新容量 = oldCapacity的2倍
    //如果capacityIncrement增量是不是0, 新容量 = oldCapacity +
    capacityIncrement增量;
    int newCapacity = oldCapacity + ((capacityIncrement > 0) ?
        capacityIncrement : oldCapacity);
}
```

```

//如果按照上面计算的新容量还不够，就按照你指定的需要最小容量来扩容minCapacity
if (newCapacity - minCapacity < 0)
    newCapacity = minCapacity;

//如果新容量超过了最大数组限制，那么单独处理
if (newCapacity - MAX_ARRAY_SIZE > 0)
    newCapacity = hugeCapacity(minCapacity);

//把旧数组中的数据复制到新数组中，新数组的长度为newCapacity
elementData = Arrays.copyOf(elementData, newCapacity);
}

```

```

public boolean remove(Object o) {
    return removeElement(o);
}

public synchronized boolean removeElement(Object obj) {
    modCount++;
    //查找obj在当前Vector中的下标
    int i = indexOf(obj);
    //如果i>=0，说明存在，删除[i]位置的元素
    if (i >= 0) {
        removeElementAt(i);
        return true;
    }
    return false;
}

public int indexOf(Object o) {
    return indexOf(o, 0);
}

public synchronized int indexOf(Object o, int index) {
    if (o == null) { //要查找的元素是null值
        for (int i = index ; i < elementCount ; i++)
            if (elementData[i]==null) //如果是null值，用==null判断
                return i;
    } else { //要查找的元素是非null值
        for (int i = index ; i < elementCount ; i++)
            if (o.equals(elementData[i])) //如果是非null值，用equals判断
                return i;
    }
    return -1;
}

public synchronized void removeElementAt(int index) {
    modCount++;
    //判断下标的合法性
    if (index >= elementCount) {
        throw new ArrayIndexOutOfBoundsException(index + " >= " +
            elementCount);
    }
    else if (index < 0) {

```

```

        throw new ArrayIndexOutOfBoundsException(index);
    }

    //j是要移动的元素个数
    int j = elementCount - index - 1;
    //如果需要移动元素，就调用System.arraycopy进行移动
    if (j > 0) {
        //把index+1位置以及后面的元素往前移动
        //index+1的位置的元素移动到index位置，依次类推
        //一共移动j个
        System.arraycopy(elementData, index + 1, elementData, index, j);
    }
    //元素的总个数减少
    elementCount--;
    //将elementData[elementCount]这个位置置空，用来添加新元素，位置的元素等着被GC回收
    elementData[elementCount] = null; /* to let gc do its work */
}

```

(2) ArrayList源码分析

JDK1.6:

```

public ArrayList() {
    this(10); //指定初始容量为10
}
public ArrayList(int initialCapacity) {
    super();
    //检查初始容量的合法性
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal Capacity: "+
                                           initialCapacity);
    //数组初始化为长度为initialCapacity的数组
    this.elementData = new Object[initialCapacity];
}

```

JDK1.7

```

private static final int DEFAULT_CAPACITY = 10; //默认初始容量10
private static final Object[] EMPTY_ELEMENTDATA = {};
public ArrayList() {
    super();
    this.elementData = EMPTY_ELEMENTDATA; //数组初始化为一个空数组
}
public boolean add(E e) {
    //查看当前数组是否够多存一个元素
    ensureCapacityInternal(size + 1); // Increments modCount!!
    elementData[size++] = e;
}

```

```

        return true;
    }
    private void ensureCapacityInternal(int minCapacity) {
        if (elementData == EMPTY_ELEMENTDATA) { //如果当前数组还是空数组
            //minCapacity按照 默认初始容量和minCapacity中的最大值处理
            minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
        }
        //看是否需要扩容处理
        ensureExplicitCapacity(minCapacity);
    }
    //...

```

JDK1.8

```

private static final int DEFAULT_CAPACITY = 10;
private static final Object[] EMPTY_ELEMENTDATA = {};
private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

public ArrayList() {
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA; //初始化为空数组
}
public boolean add(E e) {
    //查看当前数组是否够多存一个元素
    ensureCapacityInternal(size + 1); // Increments modCount!!

    //存入新元素到[size]位置, 然后size自增1
    elementData[size++] = e;
    return true;
}
private void ensureCapacityInternal(int minCapacity) {
    //如果当前数组还是空数组
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        //那么minCapacity取DEFAULT_CAPACITY与minCapacity的最大值
        minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
    }
    //查看是否需要扩容
    ensureExplicitCapacity(minCapacity);
}
private void ensureExplicitCapacity(int minCapacity) {
    modCount++; //修改次数加1

    // 如果需要的最小容量 比 当前数组的长度 大, 即当前数组不够存, 就扩容
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length; //当前数组容量

```



```

    int newCapacity = oldCapacity + (oldCapacity >> 1); //新数组容量是旧数组容
量的1.5倍
    //看旧数组的1.5倍是否够
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    //看旧数组的1.5倍是否超过最大数组限制
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);

    //复制一个新数组
    elementData = Arrays.copyOf(elementData, newCapacity);
}

```

```

public boolean remove(Object o) {
    //先找到o在当前ArrayList的数组中的下标
    //分o是否为空两种情况讨论
    if (o == null) {
        for (int index = 0; index < size; index++)
            if (elementData[index] == null) { //null值用==比较
                fastRemove(index);
                return true;
            }
    } else {
        for (int index = 0; index < size; index++)
            if (o.equals(elementData[index])) { //非null值用equals比较
                fastRemove(index);
                return true;
            }
    }
    return false;
}

private void fastRemove(int index) {
    modCount++; //修改次数加1
    //需要移动的元素个数
    int numMoved = size - index - 1;

    //如果需要移动元素，就用System.arraycopy移动元素
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index,
                           numMoved);

    //将elementData[size-1]位置置空，让GC回收空间，元素个数减少
    elementData[--size] = null; // clear to let GC do its work
}

```

```

public E remove(int index) {
    rangeCheck(index); //检验index是否合法

```

```

        modCount++; //修改次数加1

        //取出[index]位置的元素, [index]位置的元素就是要被删除的元素, 用于最后返回被删除
        的元素
        E oldValue = elementData(index);

        //需要移动的元素个数
        int numMoved = size - index - 1;

        //如果需要移动元素, 就用System.arraycopy移动元素
        if (numMoved > 0)
            System.arraycopy(elementData, index+1, elementData, index,
                             numMoved);
        //将elementData[size-1]位置置空, 让GC回收空间, 元素个数减少
        elementData[--size] = null; // clear to let GC do its work

        return oldValue;
    }

```

```

    public E set(int index, E element) {
        rangeCheck(index); //检验index是否合法

        //取出[index]位置的元素, [index]位置的元素就是要被替换的元素, 用于最后返回被替换
        的元素
        E oldValue = elementData(index);
        //用element替换[index]位置的元素
        elementData[index] = element;
        return oldValue;
    }

    public E get(int index) {
        rangeCheck(index); //检验index是否合法

        return elementData(index); //返回[index]位置的元素
    }

```

```

    public int indexOf(Object o) {
        //分为o是否为空两种情况
        if (o == null) {
            //从前往后找
            for (int i = 0; i < size; i++)
                if (elementData[i] == null)
                    return i;
        } else {
            for (int i = 0; i < size; i++)
                if (o.equals(elementData[i]))
                    return i;
        }
        return -1;
    }

```

```

    }
    public int lastIndexOf(Object o) {
        //分为o是否为空两种情况
        if (o == null) {
            //从后往前找
            for (int i = size-1; i >= 0; i--)
                if (elementData[i]==null)
                    return i;
        } else {
            for (int i = size-1; i >= 0; i--)
                if (o.equals(elementData[i]))
                    return i;
        }
        return -1;
    }
}

```

(3) LinkedList源码分析

```

int size = 0;
Node<E> first;//记录第一个结点的位置
Node<E> last;//记录最后一个结点的位置

private static class Node<E> {
    E item;//元素数据
    Node<E> next;//下一个结点
    Node<E> prev;//前一个结点

    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}

```

```

public boolean add(E e) {
    linkLast(e);//默认把新元素链接到链表尾部
    return true;
}

void linkLast(E e) {
    final Node<E> l = last;//用l 记录原来的最后一个结点

    //创建新结点
    final Node<E> newNode = new Node<>(l, e, null);
    //现在的新结点是最后一个结点了
    last = newNode;

    //如果l==null, 说明原来的链表是空的
    if (l == null)

```

```

        //那么新结点同时也是第一个结点
        first = newNode;
    else
        //否则把新结点链接到原来的最后一个结点的next中
        l.next = newNode;
    //元素个数增加
    size++;
    //修改次数增加
    modCount++;
}

```

```

public boolean remove(Object o) {
    //分o是否为空两种情况
    if (o == null) {
        //找到o对应的结点x
        for (Node<E> x = first; x != null; x = x.next) {
            if (x.item == null) {
                unlink(x); //删除x结点
                return true;
            }
        }
    } else {
        //找到o对应的结点x
        for (Node<E> x = first; x != null; x = x.next) {
            if (o.equals(x.item)) {
                unlink(x); //删除x结点
                return true;
            }
        }
    }
    return false;
}

E unlink(Node<E> x) { //x是要被删除的结点
    // assert x != null;
    final E element = x.item; //被删除结点的数据
    final Node<E> next = x.next; //被删除结点的下一个结点
    final Node<E> prev = x.prev; //被删除结点的上一个结点

    //如果被删除结点的前面没有结点，说明被删除结点是第一个结点
    if (prev == null) {
        //那么被删除结点的下一个结点变为第一个结点
        first = next;
    } else { //被删除结点不是第一个结点
        //被删除结点的上一个结点的next指向被删除结点的下一个结点
        prev.next = next;
        //断开被删除结点与上一个结点的链接
        x.prev = null; //使得GC回收
    }
}

```

```

//如果被删除结点的后面没有结点，说明被删除结点是最后一个结点
if (next == null) {
    //那么被删除结点的上一个结点变为最后一个结点
    last = prev;
} else { //被删除结点不是最后一个结点
    //被删除结点的下一个结点的prev执行被删除结点的上一个结点
    next.prev = prev;
    //断开被删除结点与下一个结点的连接
    x.next = null; //使得GC回收
}
//把被删除结点的数据也置空，使得GC回收
x.item = null;
//元素个数减少
size--;
//修改次数增加
modCount++;
//返回被删除结点的数据
return element;
}

```

12.5 Set

12.5.1 Set概述

Set系列的集合：不可重复的

Set系列的集合，有有序的也有无序的。HashSet无序的，TreeSet按照元素的大小顺序遍历，LinkedHashSet按照元素的添加顺序遍历。

12.5.2 实现类的特点

(1) HashSet:

底层是HashMap实现。添加到HashSet的元素是作为HashMap的key，value是一个Object类型的常量对象PRESENT。

依赖于元素的hashCode()和equals()保证元素的不可重复，存储位置和hashCode()值有关，根据hashCode()来算出它在底层table数组中的[index]

(2) TreeSet

底层是TreeMap实现。添加到TreeSet的元素是作为TreeMap的key，value是一个Object类型的常量对象PRESENT。

依赖于元素的大小，要么是java.lang.Comparable接口compareTo(Object obj)，要么是java.util.Comparator接口的compare(Object o1, Object o2)来比较元素的大小。认为大小相等的两个元素就是重复元素。

(3) LinkedHashSet

底层是LinkedHashMap。添加到LinkedHashSet的元素是作为LinkedHashMap的key，value是一个Object类型的常量对象PRESENT。

LinkedHashSet是HashSet的子类，比父类多维护了元素的添加顺序。

当且仅当，你既想要元素不可重复，又要保证元素的添加顺序时，再使用它。

12.6 Map

12.6.1 Map概述

用来存储键值对，映射关系的集合。所有的Map的key都不能重复。

键值对、映射关系的类型：Entry类型

Entry接口是Map接口的内部接口。所有的Map的键值对的类型都实现了这个接口。
HashMap中的映射关系，是有一个内部类来实现Entry的接口，JDK1.7是一个叫做Entry的内部类实现Entry接口。
JDK1.8是一个叫做Node的内部类实现Entry接口。
TreeMap中的映射关系，是有一个内部类Entry来实现Entry的接口

12.6.2 API

- (1) put(Object key, Object value)：添加一对映射关系
- (2) putAll(Map m)：添加多对映射关系
- (3) clear()：清空map
- (4) remove(Object key)：根据key删除一对
- (5) int size()：获取有效元素的对数
- (6) containsKey(Object key)：是否包含某个key
- (7) containsValue(Object value)：是否包含某个value
- (8) Object get(Object key)：根据key获取value
- (9) 遍历相关的几个方法

Collection values()：获取所有的value进行遍历

Set keySet()：获取所有key进行遍历

Set entrySet()：获取所有映射关系进行遍历

12.6.3 Map的实现类们的区别

- (1) HashMap：

依据key的hashCode()和equals()来保证key是否重复。

key如果重复，新的value会替换旧的value。

hashCode()决定了映射关系在table数组中的存储的位置， $\text{index} = \text{hash}(\text{key.hashCode()}) \& \text{table.length}-1$

HashMap的底层实现：JDK1.7是数组+链表；JDK1.8是数组+链表/红黑树

(2) TreeMap

依据key的大小来保证key是否重复。key如果重复，新的value会替换旧的value。

key的大小依赖于，java.lang.Comparable或java.util.Comparator。

(3) LinkedHashMap

依据key的hashCode()和equals()来保证key是否重复。key如果重复，新的value会替换旧的value。

LinkedHashMap是HashMap的子类，比HashMap多了添加顺序

12.6.4 HashMap源码分析

JDK1.6源码：

```
public HashMap() {
    //this.loadFactor加载因子，影响扩容的频率
    //DEFAULT_LOAD_FACTOR: 默认加载因子0.75
    this.loadFactor = DEFAULT_LOAD_FACTOR;
    //threshold阈值 = 容量 * 加载因子
    //threshold阈值，当size达到threshold时，考虑扩容
    //扩容需要两个条件同时满足： (1) size >= threshold (2) table[index] != null,
    即新映射关系要存入的位置非空
    threshold = (int)(DEFAULT_INITIAL_CAPACITY * DEFAULT_LOAD_FACTOR);
    //table是数组,
    //DEFAULT_INITIAL_CAPACITY: 默认是16
    table = new Entry[DEFAULT_INITIAL_CAPACITY];
    init();
}
```

JDK1.7源码：

```
public HashMap() {
    //DEFAULT_INITIAL_CAPACITY: 默认初始容量16
    //DEFAULT_LOAD_FACTOR: 默认加载因子0.75
    this(DEFAULT_INITIAL_CAPACITY, DEFAULT_LOAD_FACTOR);
}
public HashMap(int initialCapacity, float loadFactor) {
    //校验initialCapacity合法性
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal initial capacity: " +
            //校验initialCapacity合法性
            initialCapacity);
}
```

```

        if (initialCapacity > MAXIMUM_CAPACITY)
            initialCapacity = MAXIMUM_CAPACITY;
        //校验loadFactor合法性
        if (loadFactor <= 0 || Float.isNaN(loadFactor))
            throw new IllegalArgumentException("Illegal load factor: " +
                                                loadFactor);

        //加载因子，初始化为0.75
        this.loadFactor = loadFactor;
        // threshold 初始为初始容量
        threshold = initialCapacity;
        init();
    }

```

```

public V put(K key, V value) {
    //如果table数组是空的，那么先创建数组
    if (table == EMPTY_TABLE) {
        //threshold一开始是初始容量的值
        inflateTable(threshold);
    }
    //如果key是null，单独处理
    if (key == null)
        return putForNullKey(value);

    //对key的hashCode进行干扰，算出一个hash值
    int hash = hash(key);

    //计算新的映射关系应该存到table[i]位置，
    //i = hash & table.length-1，可以保证i在[0,table.length-1]范围内
    int i = indexFor(hash, table.length);

    //检查table[i]下面有没有key与我新的映射关系的key重复，如果重复替换value
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }

    modCount++;
    //添加新的映射关系
    addEntry(hash, key, value, i);
    return null;
}

private void inflateTable(int toSize) {
    // Find a power of 2 >= toSize

```



```

        int capacity = roundUpToPowerOf2(toSize); //容量是等于toSize值的最接近的2的
n次方
        //计算阈值 = 容量 * 加载因子
        threshold = (int) Math.min(capacity * loadFactor, MAXIMUM_CAPACITY +
1);

        //创建Entry[]数组, 长度为capacity
        table = new Entry[capacity];
        initHashSeedAsNeeded(capacity);
    }
    //如果key是null, 直接存入[0]的位置
    private V putForNullKey(V value) {
        //判断是否有重复的key, 如果有重复的, 就替换value
        for (Entry<K,V> e = table[0]; e != null; e = e.next) {
            if (e.key == null) {
                V oldValue = e.value;
                e.value = value;
                e.recordAccess(this);
                return oldValue;
            }
        }
        modCount++;
        //把新的映射关系存入[0]的位置, 而且key的hash值用0表示
        addEntry(0, null, value, 0);
        return null;
    }
    void addEntry(int hash, K key, V value, int bucketIndex) {
        //判断是否需要扩容
        //扩容: (1) size达到阈值 (2) table[i]正好非空
        if ((size >= threshold) && (null != table[bucketIndex])) {
            //table扩容为原来的2倍, 并且扩容后, 会重新调整所有映射关系的存储位置
            resize(2 * table.length);
            //新的映射关系的hash和index也会重新计算
            hash = (null != key) ? hash(key) : 0;
            bucketIndex = indexFor(hash, table.length);
        }
        //存入table中
        createEntry(hash, key, value, bucketIndex);
    }
    void createEntry(int hash, K key, V value, int bucketIndex) {
        Entry<K,V> e = table[bucketIndex];
        //原来table[i]下面的映射关系作为新的映射关系next
        table[bucketIndex] = new Entry<>(hash, key, value, e);
        size++; //个数增加
    }
}

```

1、put(key,value)

(1) 当第一次添加映射关系时, 数组初始化为一个长度为**16**的**HashMap\$Entry**的数组, 这个**HashMap\$Entry**类型是实现了java.util.**Map.Entry**接口

(2) 特殊考虑：如果key为null，index直接是[0]

(3) 在计算index之前，会对key的hashCode()值，做一个hash(key)再次哈希的运算，这样可以使得Entry对象更加散列的存储到table中

(4) 计算index = table.length-1 & hash;

(5) 如果table[index]下面，已经有映射关系的key与我要添加的新的映射关系的key相同了，会用新的value替换旧的value。

(6) 如果没有相同的，会把新的映射关系添加到链表的头，原来table[index]下面的Entry对象连接到新的映射关系的next中。

(7) 添加之前先判断if(size >= threshold && table[index]!=null)如果该条件为true，会扩容

```
if(size >= threshold && table[index]!=null){
```

①会扩容

②会重新计算key的hash

③会重新计算index

```
}
```

2、get(key)

(1) 计算key的hash值，用这个方法hash(key)

(2) 找index = table.length-1 & hash;

(3) 如果table[index]不为空，那么就挨个比较哪个Entry的key与它相同，就返回它的value

3、remove(key)

(1) 计算key的hash值，用这个方法hash(key)

(2) 找index = table.length-1 & hash;

(3) 如果table[index]不为空，那么就挨个比较哪个Entry的key与它相同，就删除它，把它前面的Entry的next的值修改为被删除Entry的next

JDK1.8源码

几个常量和变量：

- (1) DEFAULT_INITIAL_CAPACITY: 默认的初始容量 16
- (2) MAXIMUM_CAPACITY: 最大容量 $1 \ll 30$
- (3) DEFAULT_LOAD_FACTOR: 默认加载因子 0.75
- (4) TREEIFY_THRESHOLD: 默认树化阈值8, 当链表的长度达到这个值后, 要考虑树化
- (5) UNTREEIFY_THRESHOLD: 默认反树化阈值6, 当树中的结点的个数达到这个阈值后, 要考虑变为链表
- (6) MIN_TREEIFY_CAPACITY: 最小树化容量64
当单个的链表的结点个数达到8, 并且table的长度达到64, 才会树化。
当单个的链表的结点个数达到8, 但是table的长度未达到64, 会先扩容
- (7) Node<K,V>[] table: 数组
- (8) size: 记录有效映射关系的对数, 也是Entry对象的个数
- (9) int threshold: 阈值, 当size达到阈值时, 考虑扩容
- (10) double loadFactor: 加载因子, 影响扩容的频率

```
public HashMap() {  
    this.loadFactor = DEFAULT_LOAD_FACTOR;  
    // all other fields defaulted, 其他字段都是默认值  
}
```

```
public V put(K key, V value) {  
    return putVal(hash(key), key, value, false, true);  
}  
//目的: 干扰hashCode值  
static final int hash(Object key) {  
    int h;  
    //如果key是null, hash是0  
    //如果key非null, 用key的hashCode值 与 key的hashCode值高16进行异或  
    //    即就是用key的hashCode值高16位与低16位进行了异或的干扰运算  
  
    /*  
    index = hash & table.length-1  
    如果用key的原始的hashCode值 与 table.length-1 进行按位与, 那么基本上高16没机会用上。  
    这样就会增加冲突的概率, 为了降低冲突的概率, 把高16位加入到hash信息中。  
    */  
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
}  
final V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean  
evict) {  
    Node<K,V>[] tab; //数组  
    Node<K,V> p; //一个结点  
    int n, i; //n是数组的长度 i是下标  
  
    //tab和table等价  
    //如果table是空的  
    if ((tab = table) == null || (n = tab.length) == 0){  
        n = (tab = resize()).length;  
    }
```

```

        /*
        tab = resize();
        n = tab.length;*/
        /*
        如果table是空的, resize()完成了①创建了一个长度为16的数组②threshold = 12
        n = 16
        */
    }
    //i = (n - 1) & hash , 下标 = 数组长度-1 & hash
    //p = tab[i] 第1个结点
    //if(p==null) 条件满足的话说明 table[i]还没有元素
    if ((p = tab[i = (n - 1) & hash]) == null){
        //把新的映射关系直接放入table[i]
        tab[i] = newNode(hash, key, value, null);
        //newNode () 方法就创建了一个Node类型的新结点, 新结点的next是null
    }else {
        Node<K,V> e;
        K k;
        //p是table[i]中第一个结点
        //if(table[i]的第一个结点与新的映射关系的key重复)
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k)))){
            e = p; //用e记录这个table[i]的第一个结点
        }else if (p instanceof TreeNode){ //如果table[i]第一个结点是一个树结点
            //单独处理树结点
            //如果树结点中, 有key重复的, 就返回那个重复的结点用e接收, 即e!=null
            //如果树结点中, 没有key重复的, 就把新结点放到树中, 并且返回null, 即
            e=null
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        }else {
            //table[i]的第一个结点不是树结点, 也与新的映射关系的key不重复
            //binCount记录了table[i]下面的结点的个数
            for (int binCount = 0; ; ++binCount) {
                //如果p的下一个结点是空的, 说明当前的p是最后一个结点
                if ((e = p.next) == null) {
                    //把新的结点连接到table[i]的最后
                    p.next = newNode(hash, key, value, null);

                    //如果binCount>=8-1, 达到7个时
                    if (binCount >= TREEIFY_THRESHOLD - 1){ // -1 for 1st
                        //要么扩容, 要么树化
                        treeifyBin(tab, hash);
                    }
                }
                break;
            }
            //如果key重复了, 就跳出for循环, 此时e结点记录的就是那个key重复的结点
            if (e.hash == hash && ((k = e.key) == key || (key != null &&
                key.equals(k)))){
                break;
            }
        }
    }

```

```

    }

    p = e; //下一次循环, e=p.next, 就类似于e=e.next, 往链表下移动
    }
}

//如果这个e不是null, 说明有key重复, 就考虑替换原来的value
if (e != null) { // existing mapping for key
    V oldValue = e.value;
    if (!onlyIfAbsent || oldValue == null){
        e.value = value;
    }

    afterNodeAccess(e); //什么也没干
    return oldValue;
}

}

}

++modCount;

//元素个数增加
//size达到阈值
if (++size > threshold){
    resize(); //一旦扩容, 重新调整所有映射关系的位置
}

afterNodeInsertion(evict); //什么也没干
return null;
}

final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table; //oldTab原来的table
    //oldCap: 原来数组的长度
    int oldCap = (oldTab == null) ? 0 : oldTab.length;

    //oldThr: 原来的阈值
    int oldThr = threshold; //最开始threshold是0

    //newCap, 新容量
    //newThr: 新阈值
    int newCap, newThr = 0;
    if (oldCap > 0) { //说明原来不是空数组
        if (oldCap >= MAXIMUM_CAPACITY) { //是否达到数组最大限制
            threshold = Integer.MAX_VALUE;
            return oldTab;
        } else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
            oldCap >= DEFAULT_INITIAL_CAPACITY) {
            //newCap = 旧的容量*2, 新容量<最大数组容量限制
            //新容量: 32, 64, ...
            //oldCap >= 初始容量16
            //新阈值重新算 = 24, 48 ....
            newThr = oldThr << 1; // double threshold
        }
    } else if (oldThr > 0) { // initial capacity was placed in threshold

```

```

        newCap = oldThr;
    }else { // zero initial threshold signifies using
defaults
        newCap = DEFAULT_INITIAL_CAPACITY; //新容量是默认初始化容量16
        //新阈值= 默认的加载因子 * 默认的初始化容量 = 0.75*16 = 12
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
    }
    if (newThr == 0) {
        float ft = (float)newCap * loadFactor;
        newThr = (newCap < MAXIMUM_CAPACITY && ft <
(float)MAXIMUM_CAPACITY ?
            (int)ft : Integer.MAX_VALUE);
    }
    threshold = newThr; //阈值赋值为新阈值12, 24。。。

//创建了一个新数组，长度为newCap, 16, 32, 64。。。
@SuppressWarnings({"rawtypes", "unchecked"})
    Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
    table = newTab;

    if (oldTab != null) { //原来不是空数组
//把原来的table中映射关系，倒腾到新的table中
        for (int j = 0; j < oldCap; ++j) {
            Node<K,V> e;
            if ((e = oldTab[j]) != null) { //e是table下面的结点
                oldTab[j] = null; //把旧的table[j]位置清空
                if (e.next == null) //如果是最后一个结点
                    newTab[e.hash & (newCap - 1)] = e; //重新计算e的在新table
中的存储位置，然后放入
                else if (e instanceof TreeNode) //如果e是树结点
//把原来的树拆解，放到新的table
                    ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                else { // preserve order
                    Node<K,V> loHead = null, loTail = null;
                    Node<K,V> hiHead = null, hiTail = null;
                    Node<K,V> next;

                    /*
把原来table[i]下面的整个链表，重新挪到了新的table中
*/

                    do {
                        next = e.next;
                        if ((e.hash & oldCap) == 0) {
                            if (loTail == null)
                                loHead = e;
                            else
                                loTail.next = e;
                            loTail = e;
                        }
                    }

```

```

        else {
            if (hiTail == null)
                hiHead = e;
            else
                hiTail.next = e;
            hiTail = e;
        }
    } while ((e = next) != null);
    if (loTail != null) {
        loTail.next = null;
        newTab[j] = loHead;
    }
    if (hiTail != null) {
        hiTail.next = null;
        newTab[j + oldCap] = hiHead;
    }
}

}

}

return newTab;
}

Node<K,V> newNode(int hash, K key, V value, Node<K,V> next) {
    //创建一个新结点
    return new Node<>(hash, key, value, next);
}

final void treeifyBin(Node<K,V>[] tab, int hash) {
    int n, index;
    Node<K,V> e;
    //MIN_TREEIFY_CAPACITY: 最小树化容量64
    //如果table是空的, 或者 table的长度没有达到64
    if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
        resize();//先扩容
    else if ((e = tab[index = (n - 1) & hash]) != null) {
        //用e记录table[index]的结点的地址
        TreeNode<K,V> hd = null, tl = null;
        /*
        do...while, 把table[index]链表的Node结点变为TreeNode类型的结点
        */
        do {
            TreeNode<K,V> p = replacementTreeNode(e, null);
            if (tl == null)
                hd = p;//hd记录根结点
            else {
                p.prev = tl;
                tl.next = p;
            }

```

```

        tl = p;
    } while ((e = e.next) != null);

    //如果table[index]下面不是空
    if ((tab[index] = hd) != null)
        hd.treeify(tab); //将table[index]下面的链表进行树化
    }
}

```

1、添加过程

(1) 当第一次添加映射关系时，数组初始化为一个长度为**16**的**HashMap\$Node**的数组，这个**HashMap\$Node**类型是实现了**java.util.Map.Entry**接口

(2) 在计算index之前，会对key的hashCode()值，做一个hash(key)再次哈希的运算，这样可以使得Entry对象更加散列的存储到table中

JDK1.8关于hash(key)方法的实现比JDK1.7要简洁。key.hashCode() ^ key.Code()>>>16;

(3) 计算index = table.length-1 & hash;

(4) 如果table[index]下面，已经有映射关系的key与我要添加的新的映射关系的key相同了，会用新的value替换旧的value。

(5) 如果没有相同的，

①table[index]链表的长度没有达到8个，会把新的映射关系添加到链表的尾

②table[index]链表的长度达到8个，但是table.length没有达到64，会先对table进行扩容，然后再添加

③table[index]链表的长度达到8个，并且table.length达到64，会先把该分支进行树化，结点的类型变为TreeNode，然后把链表转为一棵红黑树

④table[index]本来就已经是红黑树了，那么直接连接到树中，可能还会考虑考虑左旋右旋以保证树的平衡问题

(6) 添加完成后判断if(size > threshold){

①会扩容

②会重新计算key的hash

③会重新计算index

}

2、remove(key)

(1) 计算key的hash值，用这个方法hash(key)

(2) 找index = table.length-1 & hash;

(3) 如果table[index]不为空，那么就挨个比较哪个Entry的key与它相同，就删除它，把它前面的Entry的next的值修改为被删除Entry的next

(4) 如果table[index]下面原来是红黑树，结点删除后，个数小于等于6，会把红黑树变为链表

12.6.5 关于HashMap的面试问题

1、HashMap的底层实现

答：JDK1.7是数组+链表，JDK1.8是数组+链表/红黑树

2、HashMap的数组的元素类型

答：java.util.Map\$Entry接口类型。

JDK1.7的HashMap中有内部类Entry实现Entry接口

JDK1.8的HashMap中有内部类Node和TreeNode类型实现Entry接口

3、为什么要使用数组？

答：因为数组的访问的效率高

4、为什么数组还需要链表？或问如何解决hash或[index]冲突问题？

答：为了解决hash和[index]冲突问题

(1) 两个不相同的key的hashCode值本身可能相同

(2) 两个hashCode不相同的key，通过hash(key)以及 hash & table.length-1 运算得到的[index]可能相同

那么意味着table[index]下可能需要存储多个Entry的映射关系对象，所以需要链表

5、HashMap的数组的初始化长度

答：默认的初始容量值是16

6、HashMap的映射关系的存储索引index如何计算

答：hash & table.length-1

7、为什么要使用hashCode()？空间换时间

答：因为hashCode()是一个整数值，可以用来直接计算index，效率比较高，用数组这种结构虽然会浪费一些空间，但是可以提高查询效率。

8、hash()函数的作用是什么

答：在计算index之前，会对key的hashCode()值，做一个hash(key)再次哈希的运算，这样可以使得Entry对象更加散列的存储到table中

JDK1.8关于hash(key)方法的实现比JDK1.7要简洁。key.hashCode() ^ key.Code()>>>16; 因为这样可以使得hashCode的高16位信息也能参与到运算中来

9、HashMap的数组长度为什么一定要是2的幂次方

答：因为2的n次方-1的二进制值是前面都0，后面几位都是1，这样的话，与hash进行&运算的结果就能保证在[0,table.length-1]范围内，而且是均匀的。

10、HashMap 为什么使用 &按位与运算代替%模运算？

答：因为&效率高

11、HashMap的数组什么时候扩容？

答：JDK1.7版：当要添加新Entry对象时发现（1）size达到threshold（2）table[index]!=null时，两个条件同时满足会扩容

JDK1.8版：当要添加新Entry对象时发现（1）size达到threshold（2）当table[index]下的结点个数达到8个但是table.length又没有达到64。两种情况满足其一都会导致数组扩容

而且数组一旦扩容，不管哪个版本，都会导致所有映射关系重新调整存储位置。

12、如何计算扩容阈值(临界值)？

答：threshold = capacity * loadfactor

13、loadFactor为什么是0.75，如果是1或者0.1呢有什么不同？

答：1的话，会导致某个table[index]下面的结点个数可能很长

0.1的话，会导致数组扩容的频率太高

14、JDK1.8的HashMap什么时候树化？

答：当table[index]下的结点个数达到8个但是table.length已经达到64

15、JDK1.8的HashMap什么时候反树化？

答：当table[index]下的树结点个数少于6个

16、JDK1.8的HashMap为什么要树化？

答：因为当table[index]下的结点个数超过8个后，查询效率就低下了，修改为红黑树的话，可以提高查询效率

17、JDK1.8的HashMap为什么要反树化？

答：因为当table[index]下树的结点个数少于6个后，使用红黑树反而过于复杂了，此时使用链表既简洁又效率也不错

18、作为HashMap的key类型重写equals和hashCode方法有什么要求

（1）equals与hashCode一起重写

（2）重写equals()方法，但是有一些注意事项；

- 自反性：x.equals(x)必须返回true。
 - 对称性：x.equals(y)与y.equals(x)的返回值必须相等。
 - 传递性：x.equals(y)为true，y.equals(z)也为true，那么x.equals(z)必须为true。
 - 一致性：如果对象x和y在equals()中使用的信息都没有改变，那么x.equals(y)值始终不变。
 - 非null：x不是null，y为null，则x.equals(y)必须为false。
- （3）重写hashCode（）的注意事项
- 如果equals返回true的两个对象，那么hashCode值一定相同，并且只要参与equals判断属性没有修改，hashCode值也不能修改；
 - 如果equals返回false的两个对象，那么hashCode值可以相同也可以不同；
 - 如果hashCode值不同的，equals一定要返回false；

hashCode不宜过简单，太简单会导致冲突严重，hashCode也不宜过于复杂，会导致性能低下；

19、为什么大部分 hashCode 方法使用 31？

答：因为31是一个不大不小的素数

20、请问已经存储到HashMap中的key的对象属性是否可以修改？为什么？

答：如果该属性参与hashCode的计算，那么不要修改。因为一旦修改hashCode()已经不是原来的值。

而存储到HashMap中时，key的hashCode()-->hash()-->hash已经确定了，不会重新计算。用新的hashCode值再查询get(key)/删除remove(key)时，算的hash值与原来不一样就不找不到原来的映射关系了。

21、所以为什么，我们实际开发中，key的类型一般用String和Integer

答：因为他们不可变。

22、为什么HashMap中的Node或Entry类型的hash变量与key变量加final声明？

答：因为不希望修改hash和key值

23、为什么HashMap中的Node或Entry类型要单独存储hash？

答：为了在添加、删除、查找过程中，比较hash效率更高，不用每次重新计算key的hash值

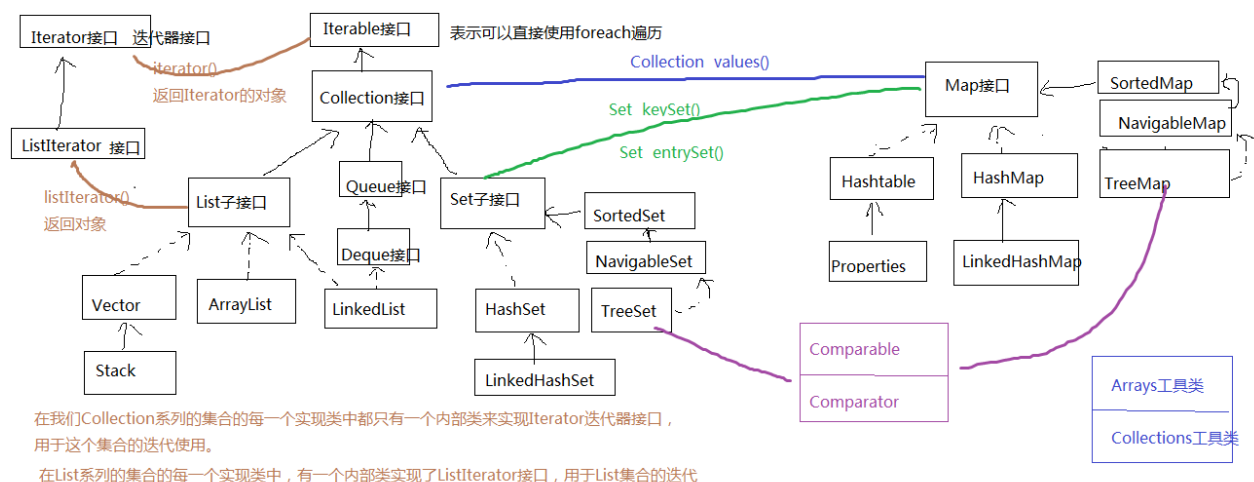
24、请问已经存储到HashMap中的value的对象属性是否可以修改？为什么？

答：可以。因为我们存储、删除等都是根据key，和value无关。

25、如果key是null是如何存储的？

答：会存在table[0]中

12.7 集合框架图



第13章 泛型

13.1 泛型的概述

泛型：参数化类型

类型形参： ， ， ， ， ，

类型实参：必须是引用数据类型，不能是基本数据类型

__, __, __, <ArrayList>. . .

13.2 形式一：泛型类与泛型接口

1、声明语法格式：

【修饰符】 class 类名/接口<类型形参列表>{

}

【修饰符】 class 类名/接口<类型形参1 extends 父类上限>{

}

【修饰符】 class 类名/接口<类型形参1 extends 父类上限 & 父接口上限>{

}

在类名或接口名后面声明的泛型形参类型，可以在当前类或接口中使用，用作声明成员变量、方法的形参、方法的返回值。

但是不能用于静态成员上

2、使用语法格式

在 (1) 创建泛型类、泛型接口的对象时，为泛型形参指定具体类型

(2) 在继承泛型类或实现泛型接口时，为泛型形参指定具体类型

示例代码

```
ArrayList<String> list = new ArrayList<String>();
```

```
ArrayList<String> list = new ArrayList<>();//JDK1.7之后可以省略
```

```
class MyStringArrayList extends ArrayList<String>{
```

}

```
class Employee implements Comparable<Employee>{
```

```
public int compareTo(Employee e){
```

_____ }

}

```
Arrays.sort(数组, new Comparator<泛型实参>(){
    public int compare(泛型实参类型 o1, 泛型实参类型 o2){
        _____
    }
});
```

3、泛型如果没有指定，会被擦除，按照最左边的上限处理，如果没有指定上限，按照Object处理

-

13.3 形式二：泛型方法

1、声明的语法格式

```
【修饰符】 <泛型形参列表> 返回值类型 方法名(【数据形参列表】)【throws 异常列表】{}
【修饰符】 <泛型形参 extends 父类上限 & 父接口上限> 返回值类型 方法名(【数据形参列表】)
【throws 异常列表】{};
```

(1) 在方法返回值类型前面声明的泛型形参类型，只能在当前方法中使用，用于表示形参的类型或返回值类型，或方法局部变量的类型，和别的方法无关。

(2) 泛型方法可以是静态方法，也可以是非静态方法

2、使用

当调用方法，会根据具体的数据的实参的类型，来确定泛型实参的类型。

-

13.4 通配符？

(1) ?：代表任意引用数据类型

(2) ? extends 上限：代表上限本身或它的子类

(3) ? super 下限：代表下限本身或它的父类

例如：

ArrayList<?>：表示可以接受任意类型

```
ArrayList<?> list = new ArrayList<String>();
ArrayList<?> list = new ArrayList<Integer>();
ArrayList<?> list = new ArrayList<Animal>();
```

ArrayList<? extends 上限>：

```
ArrayList<? extends Person> list = new ArrayList<Person>();  
ArrayList<? extends Person> list = new ArrayList<Animal>();//Animal不行, 因为  
Animal是父类  
ArrayList<? extends Person> list = new ArrayList<Student>();  
ArrayList<? extends Person> list = new ArrayList<Dog>();//Dog也不行
```

ArrayList<? super 下限>:

```
ArrayList<? super Person> list = new ArrayList<Person>();  
ArrayList<? super Person> list = new ArrayList<Animal>();  
ArrayList<? super Person> list = new ArrayList<Student>();//Student, 因为  
Student是子类  
ArrayList<? super Person> list = new ArrayList<Dog>();//Dog也不行
```

ArrayList<?>: 不能添加元素, 除了null

ArrayList<? extends 上限>: 不能添加元素, 除了null

ArrayList<? super 下限>: 可以添加下限或下限子类的对象

13.5 Collections工具类

java.util.Collections: 工具类, 操作集合

(1) public static boolean addAll(Collection<? super T> c, T... elements)

添加elements的几个对象到c集合中。T是elements对象的类型, 要求Collection集合的元素类型必须是T或T的父类

(2) public static int binarySearch(List<? extends Comparable<? super T>> list, T key)

在list集合中用二分查找key的下标, 如果存在返回的是合理的下标, 如果不存在返回的是一个负数下标
T是元素的类型,

<? extends Comparable<? super T>>, 要求集合的元素必须实现Comparable接口

<? super T>, 在实现Comparable接口, 可以指定Comparable<类型实参>为T或T的父类。

(3) public static boolean disjoint(Collection c1, Collection c2)

判断c1和c2没有交集就为true

(4) public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)

求coll集合中最大元素

<T extends Object & Comparable<? super T>>: 要求T或T的父类实现Comparable接口

因为找最大值需要比较大小

(5) public static <T extends Comparable<? super T>> void sort(List list) 给list集合排序

<T extends Comparable<? super T>>: 要求T或T的父类实现Comparable接口

_(6) public static Collection synchronizedCollection(Collection c)

以synchronizedXX开头的方法, 表示把某种非线程安全集合转为一个线程安全的集合。

_(7) public static List unmodifiableList(List<? extends T> list)

以unmodifiableXx开头的方法, 表示返回一个“只读”的集合。

第14章 IO流

14.1 java.io.File类

它是文件和目录路径名的抽象描述。

API:

_(1) getName(): 获取文件或目录的名称

_(2) getPath(): 获取文件或目录的路径

_(3) getAbsolutePath(): 获取文件或目录的绝对路径

_(4) getCanonicalPath(): 获取文件或目录的规范路径

_(5) long length(): 获取文件的长度, 单位字节

_(6) long lastModified(): 最后修改时间, 单位毫秒

_(7) String getParent(): 获取上级或父目录

File getParentFile(): 获取上级或父目录

_(8) isFile(): 判断是否是文件, 当且仅当是文件并且是存在的, 才会true

_(9) isDirectory(): 判断是否是目录, 当且仅当是目录并且存在的, 才会true

_(10) exists(): 是否存在

_(11) isHidden(): 是否隐藏

_(12) canWrite(): 是否可写

_(13) canRead(): 是否可读

_(14) String[] list(): 获取下一级

File[] listFiles(): 获取下一级

File[] listFiles(FileFilter f): 获取下一级, 但是会过滤掉一下文件和目录

_(15) createNewFile(): 创建文件

createTempFile(String prefix, String suffix)

_(16) mkdir(): 创建一级目录, 如果父目录不存在, 就失败, 但是不报异常

mkdirs(): 创建多级目录

(17) delete(): 删除文件或空目录

(18) renameTo(File f): 重命名文件或目录

14.2 IO流的四大抽象基类

1、四大超类

(1) InputStream: 字节输入流

(2) OutputStream: 字节输出流

(3) Reader: 字符输入流

(4) Writer: 字符输出流

2、分类

(1) 按照方向分: 输入流和输出流

(2) 按照处理的方式不同: 字节流和字符流

字符流只能处理纯文本的数据 (使用范围很窄)

(3) 按照角色不同: 节点流和处理流

处理流是建立在节点流的基础上的, 处理流需要包装一个节点流的对象。

处理流也可以包装另外一个处理流。

其实JDK中IO体系是用到了一个装饰者设计模式

3、API

(1) InputStream:

int read(): 一次读取一个字节, 返回的是本次读取的字节的值, 如果流末尾返回-1

int read(byte[] data): 一次读取多个字节, 读取的数据存储到data字节数组中, 最多读取data.length个字节, 返回的是实际本次读取的字节的个数, 如果流末尾返回-1。从data[0]开始存储。

int read(byte[] data, int offset, int len): 一次读取多个字节, 读取的数据存储到data字节数组中, 最多读取len个字节, 返回的是实际本次读取的字节的个数, 如果流末尾返回-1。从data[offset]开始存储。

void close(): 关闭

(2) OutputStream:

void write(int data): 一次写一个字节

void write(byte[] data): 一次写整个字节数组

void write(byte[] data, int offset, int len): 一次字节数组的一部分, 从[offset]开始, 一共len个

void close(): 关闭

void flush(): 刷新

-

_(3) Reader:

int read(): 一次读取一个字符, 返回的是本次读取的字符的Unicode值, 如果流末尾返回-1

int read(char[] data): 一次读取多个字符, 读取的数据存储到data字符数组中, 最多读取data.length个字符, 返回的是实际本次读取的字符的个数, 如果流末尾返回-1。从data[0]开始存储。

int read(char[] data, int offset, int len): 一次读取多个字符, 读取的数据存储到data字符数组中, 最多读取len个字符, 返回的是实际本次读取的字符的个数, 如果流末尾返回-1。从data[offset]开始存储。

void close(): 关闭

-

_(4) Writer

void write(int data): 一次写一个字符

void write(char[] data): 一次写整个字符数组

void write(char[] data, int offset, int len): 一次字符数组的一部分, 从[offset]开始, 一共len个

void write(String str): 一次写整个字符串

void write(String str, int offset, int count): 一次写字符串的一部分, 从[offset]开始, 一共count个

void close(): 关闭

void flush(): 刷新

-

14.3 文件IO流

1、类型

FileInputStream: 文件字节输入流, 可以读取任意类型的文件

FileOutputStream: 文件字节输出流, 可以把字节数据输出到任意类型的文件

FileReader: 文件字符输入流, 只能读取纯文本的文件。按照平台默认的字符编码进行解码。

FileWriter: 文件字符输出流, 只能把字符数据输出到纯文本文件。按照平台默认的字符编码进行编码。

-

2、读写文件的代码

示例代码：

```
public void copy(File src, File dest)throws IOException{
    //选择IO流
    FileInputStream fis = new FileInputStream(src);
    FileOutputStream fos = new FileOutputStream(dest);

    //读写
    byte[] data = new byte[1024];
    while(true){
        int len = fis.read(data);
        if(len==-1){
            break;
        }
        fos.write(data,0,len);
    }

    //关闭
    fis.close();
    fos.close();
}
```

14.4 缓冲IO流

1、分为

BufferedInputStream：字节输入缓冲流，给InputStream系列IO流增加缓冲效果

BufferedOutputStream：字节输出缓冲流，给OutputStream系列IO流增加缓冲效果

BufferedReader：字符输入缓冲流，给Reader系列IO流增加缓冲效果

String readLine(): 按行读取

BufferedWriter：字符输出缓冲流，给Writer系列IO流增加缓冲效果

void newLine(): 输出换行符

-

2、默认的缓冲区的大小是 $8192 = 1024 * 8$ （字节/字符）

3、可以给读写的过程提高效率

不仅仅是可以给文件IO流增加缓冲效果，可以给任意符合对应类型的IO流增加缓冲效果。

示例代码：

```
public void copyBuffer(File src, File dest)throws IOException{
    //选择IO流
    FileInputStream fis = new FileInputStream(src);
    FileOutputStream fos = new FileOutputStream(dest);

    }
```

```

//BufferedInputStream只能给FileInputStream增加缓冲效果，读的过程加快了
BufferedInputStream bis = new BufferedInputStream(fis); //fis在里面，bis在外面，fis比喻成内衣，bis比喻成外套

//BufferedOutputStream只能FileOutputStream增加缓冲效果，写的过程加快了
BufferedOutputStream bos = new BufferedOutputStream(fos);

//数据流向：src-->fis-->bis (从fis先缓冲到bis) -->data-->bos (从data缓冲到bos中) -->fos-->dest

//读写
byte[] data = new byte[1024];
while(true){
    int len = bis.read(data);
    if(len== -1){
        break;
    }
    bos.write(data,0,len);
}

//关闭
//关闭比喻成脱衣服
bos.close();
fos.close();

bis.close(); //先脱外套，再脱内衣
fis.close();
}

```

14.5 编码与解码的IO流（转换流）

1、编码：OutputStreamWriter

可以把字符流转为字节流输出，并且在转换的过程中，可以指定字符编码。

2、解码：InputStreamReader

可以把字节输入流转为字符输入流，并且在转换的过程中，可以指定字符编码。

3、示例代码：解码（文件是GBK，当前平台是UTF-8）

```

@Test
public void test4() throws IOException{
    //因为这里想要用在程序中按照“指定”的编码方式进行解码，而不是按照平台“默认”的编码方式进行解码
    //所以，我这里仍然用FileInputStream字节流，把文件编码后的数据，原样读取
    //从文件到FileInputStream fis内存的过程中，先不解码
    //因为如果选择FileReader，从文件到FileReader的过程中，就已经按照平台“默认”的编码方式解码好了，我们无法控制
}

```

```

FileInputStream fis = new FileInputStream("d:/File类概述.txt");
//我要使用InputStreamReader, 把FileInputStream转为字符流
//    InputStreamReader isr = new InputStreamReader(fis); //如果没有指定, 还是按照
默认的编码方式
InputStreamReader isr = new InputStreamReader(fis, "GBK"); //如果指定, 就按照指
定的编码方式解码

//文件-->fis (字节流) -->解码-->isr (字符流) -->br ->读取的是字符
//字符流, 要么按照char[]读取, 要么可以用BufferedReader包装按行读取
BufferedReader br = new BufferedReader(isr);
String line;
while((line = br.readLine()) != null){
    System.out.println(line);
}

br.close();
isr.close();
fis.close();
}

```

4、示例代码：编码（文件是GBK，当前平台是UTF-8）

```

@Test
public void test3() throws IOException{
    String hua = "File类可以使用文件路径字符串来创建File实例";

    //因为想要用在程序中进行编码, 所以这里选用FileOutputStream
    FileOutputStream fos = new FileOutputStream("d:/File类概述.txt", true);

    //这里xx, 想要直接操作字符串, 那么必须是字符流, 而fos是字节流, 无法直接操作字符串
    //    xx.write("\r\n");
    //    xx.write(hua);

    //数据流向: 内存 --> osw (字符流) -->在写入fos过程中进行编码-->fos (字节流) -->文件

    OutputStreamWriter osw = new OutputStreamWriter(fos, "GBK");
    osw.write("\r\n");
    osw.write(hua);

    osw.close();
    fos.close();
}

```

14.6 数据IO流

1、类型

DataInputStream：允许应用程序以与机器无关方式从底层输入流中读取基本 Java 数据类型。

DataOutputStream：允许应用程序以适当方式将基本 Java 数据类型写入输出流中。

- 它俩必须配对使用
- 读的顺序要与写的顺序一致

2、API

| DataOutputStream | DataInputStream |
|-----------------------------|-----------------------|
| writeInt(xx)：输出int类型整数 | int readInt() |
| writeDouble(xx)：输出double类型 | double readDouble() |
| writeBoolean(xx) | boolean readBoolean() |
| writeLong(xx) | long readLong() |
| writeChar(xx) | char readChar() |
| writeByte(xx) | byte readByte() |
| writeShort(xx) | short readShort() |
| writeFloat(xx) | float readFloat() |
| writeUTF(String str)：输出字符串的 | String readUTF() |

14.7 对象IO流

1、类型

ObjectOutputStream：对象序列化，输出对象，把对象转为字节序列输出

ObjectInputStream：对象反序列化，读取对象，把字节序列重构成Java对象

2、API

ObjectOutputStream：writeObject(对象)

ObjectInputStream：Object readObject()

3、序列化需要注意哪些内容？

（1）所有要序列化的对象的类型都必须实现java.io.Serializable接口

如果对象的属性类型也是引用数据类型，那么也要实现java.io.Serializable接口

（2）希望类的修改对象反序列化不产生影响，那么我们最后能够增加一个序列化版本ID

private static final long serialVersionUID = 1L;

（3）如果有些属性不想要序列化，可以加transient

_(4) 如果某个属性前面有static修饰, 也不参与序列化

4、除了Serializable接口之外, 还可以实现java.io.Externalizable接口, 但是要求重写:

void readExternal(ObjectInput in)

void writeExternal(ObjectOutput out)

关于哪些属性序列化和反序列化, 由程序员自己定。

14.8 其他的IO流相关内容

1、如果要实现按行读取, 可选择什么类型?

BufferedReader: String readLine()

Scanner: String nextLine()

2、如果要按行输出, 可以选择什么类型?

_(1) 自己处理\r\n

_(2) BufferedWriter: newLine()

_(3) PrintStream和PrintWriter: println()

-

14.9 JDK1.7之后引入新try..catch

语法格式:

```
try(需要关闭的资源对象的声明){
    业务逻辑代码
}catch(异常类型 e){
    处理异常代码
}catch(异常类型 e){
    处理异常代码
}
....
```

它没有finally, 也不需要程序员去关闭资源对象, 无论是否发生异常, 都会关闭资源对象

示例代码:

```
@Test
public void test03(){
    //从d:/1.txt(GBK)文件中, 读取内容, 写到项目根目录下1.txt(UTF-8)文件中
    try(
        FileInputStream fis = new FileInputStream("d:/1.txt");
        InputStreamReader isr = new InputStreamReader(fis, "GBK");
        BufferedReader br = new BufferedReader(isr);
    )
```

```

        FileOutputStream fos = new FileOutputStream("1.txt");
        OutputStreamWriter osw = new OutputStreamWriter(fos, "UTF-8");
        BufferedWriter bw = new BufferedWriter(osw);
    ){
        String str;
        while((str = br.readLine()) != null){
            bw.write(str);
            bw.newLine();
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

第十八章 设计模式

18.1 模板设计模式（了解）

1、当解决某个问题，或者完成某个功能时，主体的算法结构（步骤）是确定的，只是其中的一个或者几个小的步骤不确定，要有使用者（子类）来确定时，就可以使用模板设计模式

2、示例代码：计算任意一段代码的运行时间

```

//模板类
public abstract class CalTime{
    public long getTime(){
        //1、获取开始时间
        long start = System.currentTimeMillis();

        //2、运行xx代码：这个是不确定的
        doWork();

        //3、获取结束时间
        long end = System.currentTimeMillis();

        //4、计算时间差
        return end - start;
    }

    protected abstract void doWork();
}

```

使用模板类：

```

public class MyCalTime extends CalTime{
    _____protected void doWork(){
    _____//....需要计算运行时间的代码
    _____}
}

```

测试类

```

public class Test{
    _____public static void main(String[] args){
    _____MyCalTime my = new MyCalTime();
    _____System.out.println("运行时间：" + my.getTime());
    _____}
}

```

18.2 单例设计模式

单例：整个系统中，某个类型的对象只有一个。

1、饿汉式

(1) 枚举式

```

public enum Single{
    _____INSTANCE
}

```

(2) 形式二

```

public class Single{
    _____public static final Single INSTANCE = new Single();
    _____private Single(){
    _____
    _____}
}

```

(3) 形式三

```

public class Single{
    _____private static final Single INSTANCE = new Single();
    _____private Single(){
    _____
    _____}
    _____public static Single getInstance(){
    _____return INSTANCE;
    _____}
}

```


2、懒汉式

(1) 内部类形式

```
public class Single{  
    private Single(){}  
  
    private static class Inner{  
        static final Single INSTANCE = new Single();  
    }  
  
    public static Single getInstance(){  
        return Inner.INSTANCE;  
    }  
  
}
```

(2) 形式二

```
public class Single{  
    private static Single instance;  
    private Single(){}  
  
    public static Single getInstance(){  
        if(instance == null){  
            synchronized(Single.class){  
                if(instance == null){  
                    instance = new Single();  
                }  
            }  
        }  
        return instance;  
    }  
  
}
```