# 2019java临潼火箭班知识点总结

向阳

目录

[2019java临潼火箭班知识点总结 2](#_Toc9002334)

[1.JavaSE的数据类型和运算符 2](#_Toc9002335)

[1.1数据类型 2](#_Toc9002336)

[2.数据类型转换 2](#_Toc9002337)

[2.1自动类型转换（隐式） 2](#_Toc9002338)

[2.2强制类型转换（显式）： 2](#_Toc9002339)

[3.java运算符 2](#_Toc9002340)

[3.1算数运算符 2](#_Toc9002341)

[3.2赋值运算符 2](#_Toc9002342)

[3.3比较运算符 2](#_Toc9002343)

[3.4逻辑运算符（重点） 2](#_Toc9002344)

[3.5三目运算符 2](#_Toc9002345)

[4.数据控制语句 2](#_Toc9002346)

[4.1判断语句 2](#_Toc9002347)

[4.2选择语句 2](#_Toc9002348)

[4.3.循环语句 2](#_Toc9002349)

[4.4 跳出循环 2](#_Toc9002350)

[5.方法 2](#_Toc9002351)

[5.1 方法使用 2](#_Toc9002352)

[5.2方法重载： 2](#_Toc9002353)

[5.3方法递归 2](#_Toc9002354)

[6.数组 2](#_Toc9002355)

[6.1一维数组 2](#_Toc9002356)

[6.2二维数组 2](#_Toc9002357)

[6.3数组拷贝： 2](#_Toc9002358)

[6.类与对象 2](#_Toc9002359)

[7封装 2](#_Toc9002360)

[7.1代码块 2](#_Toc9002361)

[7.2内部类： 2](#_Toc9002362)

[8.继承 2](#_Toc9002363)

[9.多态 2](#_Toc9002364)

[9.1引用类型转换： 2](#_Toc9002365)

[9.2抽象类 2](#_Toc9002366)

[10.接口 2](#_Toc9002367)

[10.1抽象方法的使用： 2](#_Toc9002368)

[10.2模板设计模式 2](#_Toc9002369)

[10.3简单工厂模式 2](#_Toc9002370)

[10.4工厂方法模式 2](#_Toc9002371)

[10.5抽象工厂模式 2](#_Toc9002372)

[11.三大特殊类 2](#_Toc9002373)

[11.1.String类 2](#_Toc9002374)

[11.2.Object类 2](#_Toc9002375)

[11.3.包装类 2](#_Toc9002376)

[12设计模式 2](#_Toc9002377)

[12.1. 单例设计模式 2](#_Toc9002378)

[12.2. 多例设计模式 2](#_Toc9002379)

[13.异常与捕获 2](#_Toc9002380)

[13.1java常见异常： 2](#_Toc9002381)

[13.2捕获异常 2](#_Toc9002382)

[数据结构 2](#_Toc9002383)

[1.时间复杂度和空间复杂度 2](#_Toc9002384)

[1.1时间复杂度 2](#_Toc9002385)

[1.2.空间复杂度 2](#_Toc9002386)

[2.顺序表 2](#_Toc9002387)

[2.1顺序表 2](#_Toc9002388)

[3.链表 2](#_Toc9002389)

[3.1无头单向非循环链表 2](#_Toc9002390)

[3.2带头节点循环单链表 2](#_Toc9002391)

[3.3不带头节点双向链表 2](#_Toc9002392)

[4.栈和队列 2](#_Toc9002393)

[4.1栈 2](#_Toc9002394)

[4.2队列 2](#_Toc9002395)

[5.排序 2](#_Toc9002396)

[5.1插入排序 2](#_Toc9002397)

[5.2选择排序 2](#_Toc9002398)

[5.5快速排序 2](#_Toc9002399)

[5.6归并排序 2](#_Toc9002400)

[5.7排序算法的复杂度及稳定性 2](#_Toc9002401)

# 2019java临潼火箭班知识点总结

## 1.JavaSE的数据类型和运算符

### 1.1数据类型

1.java数据类型分为两种

（1）基本数据类型：

基本数据类型包括boolean（布尔型）、float（单精度浮点型）、char（字符型）、byte（字节型）、short（短整型）、int（整型）、long（长整型）和double（双精度浮点型）共8种

|  |  |  |  |
| --- | --- | --- | --- |
| 类型名称 | 关键字 | 占用内存 | 取值范围 |
| 字节型 | byte | 1字节 | -128~127 |
| 短整型 | short | 2字节 | -32768~32767 |
| 整型 | int | 4字节 | -2147483648~2147483647 |
| 长整型 | long | 8字节 | -9223372036854775808L~9223372036854775807L |
| 单精度浮点型 | float | 4字节 | +/-3.4E+38F（6~7个有效位） |
| 双精度浮点型 | double | 8字节 | +/-1.8E+308(15个有效位） |
| 字符型 | char | 2字节 | ISO单一字符集 |
| 布尔型 | boolean | 1字节 | true或false |

（2）引用数据类型

数组，类（class），接口（interface）

## 2.数据类型转换

当数据类型不一样时，将会发生数据类型转换。

### 2.1自动类型转换（隐式）

1.特点：代码不需要进行特殊处理，自动完成。

2.规则：数据范围从小到大。

* 数值型数据的转换：byte→short→int→long→float→double。
* 字符型转换为整型：char→int。

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

**int a=10;**

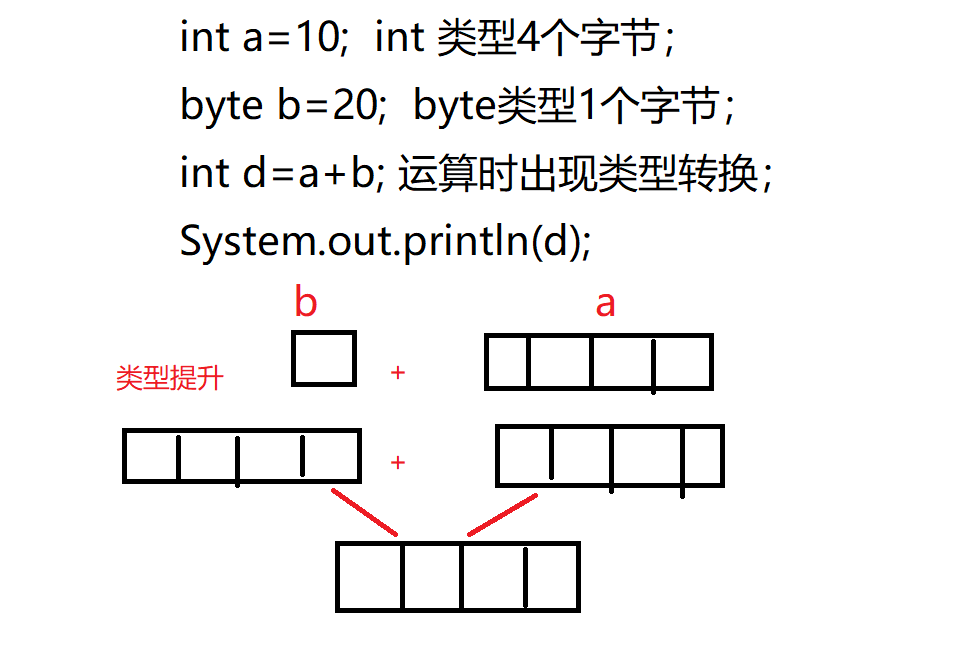
**byte b=20;**

**//byte c=a+b; //报错**

**int d=a+b;**

**System.out.println(d);**

**}**



### 2.2强制类型转换（显式）：

当两种数据类型不兼容，或目标类型的取值范围小于源类型时，自动转换将无法进行，这时就需要进行强制类型转换。

规则：

* 范围小的类型向范围大的类型提升，byte、short、char运算时直接提升为int。
* byte🡪short🡪char🡪int🡪long🡪float🡪double.

**转换格式**：数据类型变量名=（数据类型）被转数据值

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

**int a=10;**

**double b=20;**

**a=(int)b;**

**System.out.println(a);**

**}**

**注意：**

* 浮点转成整数，直接取消小数点，可能造成数据损失精度。
* int强制转成short砍掉2个字节，可能造成数据丢失

## 3.java运算符

### 3.1算数运算符

|  |  |
| --- | --- |
| + | 加法运算，字符串连接运算 |
| - | 减法运算 |
| \* | 乘法运算 |
| / | 除法运算 |
| % | 取模运算，两个数字相除取余数 |
| ++-- | 自增自减运算 |

其中，++和--既可以出现在操作数的左边，也可以出现在右边，但结果不同。

Int a=++1; a的结果为2 intb= --1;b的结果为0

Int a=1++;a的结果为1 intb=1--;b的结果为1

进行算术运算时应注意以下两点：

* 求余（％）运算要求参与运算的两个操作数均为整型，不能为其他类型。
* 两个整数进行除法运算，其结果仍为整数。如果整数与实数进行除法运算，则结果为实数。

### 3.2赋值运算符

赋值运算符包括：

|  |  |
| --- | --- |
| = | 等于号 |
| += | 加等于 |
| -= | 减等于 |
| \*= | 乘等于 |
| /= | 除等于 |
| %= | 取模等 |

赋值运算符，就是将符号右边的值，赋给左边的变量。

### 3.3比较运算符

|  |  |
| --- | --- |
| 比较运算符包括： |  |
| == | 比较符号两边数据是否相等，相等结果是true。 |
| < | 比较符号左边的数据是否小于右边的数据，如果小于结果是true。 |
| > | 比较符号左边的数据是否大于右边的数据，如果大于结果是true。 |
| <= | 比较符号左边的数据是否小于或者等于右边的数据，如果小于结果是true。 |
| >= | 比较符号左边的数据是否大于或者等于右边的数据，如果小于结果是true。 |
| ！= | 不等于符号，如果符号两边的数据不相等，结果是true |

* 比较运算符，是两个数据之间进行比较的运算，运算结果都是布尔值true或者false。

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

**System.out.println(1 == 1);//true**

**System.out.println(1 < 2);//true**

**System.out.println(3 > 4);//false**

**System.out.println(3 <= 4);//true**

**System.out.println(3 >= 4);//false**

**System.out.println(3 != 4);//true**

**}**

### 3.4逻辑运算符（重点）

|  |  |
| --- | --- |
| 逻辑运算符包括 | 特点 |
| &&短路与 | 1.两边都是true，结果是true  2.一边是false，结果是false  短路特点：符号左边是false，右边不再运算 |
| ||短路或 | 1.两边都是false，结果是false  2.一边是true，结果是true  短路特点：符号左边是true，右边不再运算 |
| ！取反 | 1.!true结果是false2.!false结果是true |

* 逻辑运算符，是用来连接两个布尔类型结果的运算符，运算结果都是布尔值 true 或者 false

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

**System.out.println(true && true);//true**

**System.out.println(true && false);//false**

**System.out.println(false && true);//false,右边不计算**

**System.out.println(false||false);//false**

**System.out.println(false||false);//true**

**System.out.println(true||false);//true,右边不计算**

**System.out.println(!false);//true**

**}**

### 3.5三目运算符

格式：

* 数据类型 变量名 = 布尔类型表达式？结果1：结果2

三元运算符计算方式：

* 布尔类型表达式结果是true，三元运算符整体结果为结果1，赋值给变量。
* 布尔类型表达式结果是false，三元运算符整体结果为结果2，赋值给变量。

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

**int i = (1 == 2 ? 100 : 200);**

**System.out.println(i);//200**

**int j = (3 <= 4 ? 500 : 600);**

**System.out.println(j);//500**

**}**

## 4.数据控制语句

### 4.1判断语句

#### 4.1.1 if语句

**if语句：**

**格式：**

**if(布尔表达式){**

**//条件满足时执行代码**

**}else{**

**//条件不满足时执行代码**

**}**

**If else语句：**

**if(布尔表达式){**

**//条件满⾜时执⾏代码**

**}else if(布尔表达式){**

**//条件满足时执行代码**

**} ...**

**else{**

**//条件不满足时执行代码**

**}**

### 4.2选择语句

#### switch语句

**格式**

**switch(表达式){**

**case常量值1:**

**语句体 1;**

**break;**

**case常量值2:**

**语句体2;**

**break;**

**...**

**default:**

**语句体n+1;**

**break;**

**}**

**流程：**

* 首先计算出表达式的值
* 其次，和 case依次比较，一旦有对应的值，就会执行相应的语句，在执行的过程中，遇到break就会结束。
* 最后，如果所有的 case都和表达式的值不匹配，就会执行default语句体部分，然后程序结束掉。

### 4.3.循环语句

#### 4.3.1 for循环：

**格式：**

**for(初始化表达式;循环结束判断;步进表达式){**

**循环体**

**}**

**循环结构的基本组成部分，一般可以分成四部分：**

1. 初始化语句：在循环开始最初执行，而且只做唯一一次。

2. 条件判断：如果成立，则循环继续；如果不成立，则循环退出。

3. 循环体：重复要做的事情内容，若干行语句。

4. 步进语句：每次循环之后都要进行的扫尾工作，每次循环结束之后都要执行一次。

#### 3.2while循环

**格式：**

**初始化表达式**

**while(循环结束条件判断){**

**循环语句**

**修改循环结束判断**

}

#### 3.3 do-while循环

**do{**

**循环语句句;**

**修改循环结束判断;**

**}while(循环结束条件判断);**

**特点：**一定会先执行一次循环。

**循环使用规则：**

1.优先使用for和while ，do-while一般不使用。

2.知道明确循环次数用for，不知道循环次数用while。

### 4.4 跳出循环

**4.4.1 break**

使用场景：终止 switch或者循环

* 在选择结构 switch语句中
* 在循环语句中
* 离开使用场景的存在是没有意义的

**4.4.2 continue**

使用场景：结束本次循环，继续下一次的循环

* 执⾏行行到此语句句时会跳过当前循环的剩余部分，返回循环判断

## 5.方法

### 5.1 方法使用

**格式：**

**修饰符 返回值类型 方法名(参数列表){**

**方法体**

**return 结果;**

**}**

* **修饰符：** public static 固定写法
* **返回值类型：** 当方法以void关键字声明，那么此方法没有返回值；若有返回值，返回值可以为基本类型和引用类型。
* **参数列表：**方法在运算过程中的未知数据，调用者调用方法时传递
* **return ：**将方法执行后的结果带给调用者，方法执行到 return ，整体方法运行结束。

**public class TestDemo02 {**

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

**// 单独调用**

**add(10, 20);**

**System.out.println("===========");**

**// 打印调用**

**System.out.println(add(10, 20)); // 30**

**System.out.println("===========");**

**// 赋值调用**

**int number = add(15, 25);**

**number += 100;**

**System.out.println("变量的值：" + number); // 140**

**}**

**public static int add(int a, int b) {**

**System.out.println("执行方法");**

**int sum = a + b;**

**return sum;**

**}**

**}**

### 5.2方法重载：

**定义：**函数名相同，参数列表不同，返回值不做要求。

**实现代码：**

**public class TestDemo02 {**

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

**//此时将根据参数的类型和个数的不同执行不同的方法体**

**System.out.println(add(10, 20));**

**System.out.println(add(30, 40, 50));**

**System.out.println(add(1.1, 2.1));**

**}**

**public static int add(int x, int y) {**

**return x + y;**

**}**

**//此时方法名称相同，参数的个数不同**

**public static int add(int x, int y, int z) {**

**return x + y + z;**

**}**

**public static double add(double x, double y) {**

**return x + y;**

**}**

### 5.3方法递归

**定义：是一个方法自己调用自己的形式。**

**特点：**

* **方法必须有一个递归的结束条件；**
* **方法在每次递归处理的时候一定要做出一些变更；**

**实现1到100叠加：**

**public class TestDemo02 {**

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

**System.out.println(sum(100));**

**}**

**public static int sum(int num) {**

**if (num == 1)**

**return 1;**

**return num + sum(--num);**

**}**

**}**

**第1次执行sum()方法 return 100 + sum(99)；**

**第2次执行sum()方法 return 99 + sum(98)；**

**第3次执行sum()方法 return 98 + sum(97)；**

**…**

**倒数第2次执行sum()方法 return 2 + sum(1)；**

**倒数第1次执行sum()方法 return 1；**

## 6.数组

**数组概念**： 数组就是一种数据结构，用来储存同一类型值得集合。

**数组的特点：**

* 数组是一种引用数据类型；
* 数组当中的多个数据，类型必须统一；
* 数组的长度在程序运行期间不可改变；

数组的存储方式为： 数组名在stack（栈）上存储数组第一个元素的地址，数组元素存储在heap（堆）上。

### 6.1一维数组

两种常见的初始化方式：

1.动态初始化（指定长度）在创建数组的时候，直接指定数组当中的数据元素个数。

**int[] arrayA = new int[10];**

**// 格式：数据类型[] 数组名称 = new 数据类型[数组长度];**

2. 2.静态初始化（指定内容）在创建数组的时候，不直接指定数据个数多少，而是直接将具体的数据内容进行指定。

**int[] arrayB = new int[1,2,3,4,5,6,7,8,9,10]**

**// 数据类型[] 数组名称 = new 数据类型[] { 元素1, 元素2, ... };**

使用静态初始化数组的时候，格式还可以省略一下。

**省略格式：**

**数据类型[ ] 数组名称 = { 元素1, 元素2, … };**

**注意事项：**

1. 静态初始化没有直接指定长度，但是仍然会自动推算得到长度。
2. 静态初始化标准格式可以拆分成为两个步骤。
3. 动态初始化也可以拆分成为两个步骤。
4. 静态初始化一旦使用省略格式，就不能拆分成为两个步骤了。

### 6.2二维数组

* 格式 ： 数据类型[ ][ ] 数组名 = new 数据类型[m][n];

**二维数组初始化：**

二维数组可以初始化，和一维数组一样，可以通过3种方式来指定元素的初始值。

* 格式一：使用大括号直接赋值，适合已经确定知道数组元素的情况

**int[][] arr1 = new int[][]{{1,2}, {2, 3}, {4, 5}};**

* 格式二：给定二维数组的大小

**int[][] arr2 = new int[4][3];**

* 格式三：数组第二维的长度可变化，未改变

**int[][] arr3 = new int[4][];**

**使用建议：**

如果不确定数组当中的具体内容，用动态初始化；否则，已经确定了具体的内容，用静态初始化。

**数组索引**

* 索引：

每一个存储到数组的元素，都会自动的拥有一个编号，从0开始，这个自动编号称为数组索引可以通过数组的索引访问到数组中的元素。

* 数组的长度属性：

每个数组都具有长度，而且是固定的，Java中赋予了数组的一个属性，可以获取到数组的长度，语句为：数组名.length ，属性length的执行结果是数组的长度，int类型结果。由次可以推断出数组的最大索引值为数组名.length-1。

* 索引访问数组中的元素：

**数组名[索引]=数值，为数组中的元素赋值**

**变量=数组名[索引]，获取出数组中的元素**

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

**int[] arr = {1,2,3,4,5}; //定义存储int类型数组，赋值元素1，2，3，4，5**

**arr[0] = 6; //为0索引元素赋值为6**

**int i = arr[0]; //获取数组0索引上的元素**

**System.out.println(i); //直接输出数组0索引元素**

**System.out.println(arr[0]);**

**}**

* 注意事项

1.数组的索引编号从0开始，一直到“数组的长度-1”为止。

2.如果访问数组元素的时候，索引编号并不存在，那么将会发生

数组索引越界异常

ArrayIndexOutOfBoundsException

3.数组属于引用类型，因此在使用前一定要开辟空间（实例化），否则会产生 NullPoninterException

### 6.3数组拷贝：

**1.for**

**public class TestDame1 {**

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

**int[] array = {10, 20, 30, 40, 50};**

**int[] array2 = new int[array.length];**

**System.out.println(Arrays.toString(array));**

**System.out.println("=======================");**

**//第一种方法 for**

**for (int i = 0; i < array.length; i++) {**

**array2[i] = array[i];**

**}**

**System.out.println(Arrays.toString(array));**

**System.out.println(Arrays.toString(array2));**

**2.clone（要克隆的数组.clone）**

**public class TestDame1 {**

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

**int[] array = {10, 20, 30, 40, 50};**

**int[] array3 = new int[array.length];**

**System.out.println(Arrays.toString(array));**

**System.out.println("=======================");**

**//第二种方法 alone**

**array3 = array.clone();**

**System.out.println(Arrays.toString(array));**

**System.out.println(Arrays.toString(array3));**

**}**

**3.System.arraycopy(源数组名，源数组拷贝起点，目标数组名，目标数组拷贝起点，拷贝长度)**

**public class TestDame1 {**

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

**int[] array = {10, 20, 30, 40, 50};**

**int[] array4 = new int[array.length];**

**System.out.println(Arrays.toString(array));**

**System.out.println("=======================");**

**System.arraycopy(array,0,array4,0,array4.length);**

**System.out.println(Arrays.toString(array));**

**System.out.println(Arrays.toString(array4));**

**}**

**4.Arrays.copyOf（数组名，数组长度）**

**public class TestDame1 {**

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

**int[] array = {10, 20, 30, 40, 50};**

**int[] array5 = new int[array.length];**

**System.out.println(Arrays.toString(array));**

**System.out.println("=======================");**

**array5 = Arrays.copyOf(array, array.length);**

**System.out.println(Arrays.toString(array));**

**System.out.println(Arrays.toString(array4));**

**}**

**注意事项：**

1. for:拷贝对于基本类型来说是深拷贝。
2. clone对于基本类型来说是深拷贝。
3. System.arraycopy()对于基本类型来说是深拷贝。
4. native System.arraycopy()拷贝最快
5. 数组元素是引用类型是浅拷贝。native：底层代码由C/C++
6. 四种拷贝方式对于引用类型来说都是浅拷贝。

## 6.类与对象

**类：**是一组相关属性和行为的集合。可以看成是一类事物的模板，使用事物的属性特征和行为特征来描述该类事物。（理解为做月饼的模子）

现实中描述一类事物：

* **属性** ：就是该事物的状态信息。
* **行为** ：就是该事物能够做什么。

**对象：**是一类事物的具体体现。对象是类的一个实例，必然具备该类事物的属性和行为。（相当于你用模子做出的月饼）。

**定义一个类：**

Class 类名称{

属性1

属性2

属性n

方法1（）{ }

方法2（）{ }

方法n（）{ }

}

**定义一个学生类：**

**class person {  
 public String name; //属性  
 public int age;  
  
 public person(String name, int age) {  
 this.name = name;   
 this.age = age;  
 }**

**public void eat(){  
 System.*out*.println("打游戏！"); //行为  
 }  
 public void sleep() {  
 System.*out*.println("睡觉觉！");  
 }**

**}**

**注意：**

1. 成员变量是直接定义在类当中的，在方法外边。

2. 成员方法不要写static关键字。

**产生对象：**

**格式：**

类名称 对象名 = new 类名称();

Person person = new Person();

**使用：**

使用，分为两种情况：

使用成员变量：对象名.成员变量名

使用成员方法：对象名.成员方法名(参数)

（也就是，想用谁，就用对象名点儿谁。）

**注意：**1.只要出现了了关键字 new ，就开辟了了内存。

2. Java 中，所谓的性能调优，调整的就是内存问题。

**局部变量和成员变量区别：**

1. 定义的位置不一样

局部变量：在方法的内部

成员变量：在方法的外部，直接写在类当中

2. 作用范围不一样

局部变量：只有方法当中才可以使用，出了方法就不能再用

成员变量：整个类全都可以通用。

3. 默认值不一样

局部变量：没有默认值，如果要想使用，必须手动进行赋值

成员变量：如果没有赋值，会有默认值，规则和数组一样

4. 内存的位置不一样

局部变量：位于栈内存

成员变量：位于堆内存

5. 生命周期不一样

局部变量：随着方法进栈而诞生，随着方法出栈而消失

成员变量：随着对象创建而诞生，随着对象被垃圾回收而消失

**对象属性的初始化方式？**

1、提供一些列的get和set方法(public)：

2、通过构造函数。

**对象的产生需要以下2步：**

1、为对象开辟内存

2、调用合适的构造函数-（构造函数不止一个。）

构造函数可以发生重载。

3、静态代码块 实例代码块

## 7封装

**封装性：**将类的某些信息隐藏在类内部，不允许外部程序直接访问，而是通过该类提供的方法来实现对隐藏信息的操作和访问。

**好处：**

* 只能通过规定的方法访问数据。
* 隐藏类的实例细节，方便修改和实现。

步骤：

1. 使用 private 关键字来修饰成员变量。

**Public class Student{**

**Private String name;**

**private int age;**

**}**

2. 对需要访问的成员变量，提供对应的一对 get 方法 、set方法。

**public class Student {**

**private String name;**

**private int age;**

**public void setName(String n) {**

**name = n;**

**}**

**public String getName() {**

**return name;**

**}**

**public void setAge(int a) {**

**age = a;**

**}**

**public int getAge() {**

**return age;**

**}**

**}**

set方法：主要用于进行属性内容的设置与修改

get方法：主要用于属性内容的取得

**构造方法：**当一个对象被创建时候，构造方法用来初始化该对象，给对象的成员变量赋初始值

**格式**：

**修饰符 构造方法名 (参数列表){**

**方法体**

**}**

分为有参构造方法和无参构造方法。

**public class Student {**

**private String name;**

**private int age;**

**// 无参数构造方法**

**public Student() {}**

**// 有参数构造方法**

**public Student(String name,int age) {**

**this.name = name;**

**this.age = age;**

**}**

**}**

**注意事项**

1. 如果你不提供构造方法，系统会给出无参数构造方法。

2. 如果你提供了构造方法，系统将不再提供无参数构造方法。

3. 构造方法是可以重载的，既可以定义参数，也可以不定义参数

**This关键字：表示当前类对象的引用：**

this.属性 操作当前对象的属性

this.方法 调用当前对象的方法。

**使⽤用this调⽤用构造⽅方法时请注意：**

1. this调用构造方法的语句必须在构造方法首行

2. 使⽤用this调用构造方法时，请留有出口

**static关键字**：

一旦使用static修饰成员方法，那么这就成为了静态方法。静态方法不属于对象，而是属于类的。

如果没有static关键字，那么必须首先创建对象，然后通过对象才能使用它。

如果有了static关键字，那么不需要创建对象，直接就能通过类名称来使用它。

无论是成员变量，还是成员方法。如果有了static，都推荐使用类名称进行调用。

静态变量：类名称.静态变量

静态方法：类名称.静态方法()

注意事项：

1. 静态不能直接访问非静态。

原因：因为在内存当中是【先】有的静态内容，【后】有的非静态内容。

“先人不知道后人，但是后人知道先人。”

2. 静态方法当中不能用this。

原因：this代表当前对象，通过谁调用的方法，谁就是当前对象。

权限修饰符：

在Java中提供了四种访问权限，使用不同的访问权限修饰符修饰时，被修饰的内容会有不同的访问权限，

* public ：公共的。
* protected ：受保护的
* default ：默认的
* private ：私有的

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | public | protected | default（空的） | private |
| 同一类中 | √ | √ | √ | √ |
| 同一包中 | √ | √ | √ |  |
| 不同包的子类 | √ | √ |  |  |
| 不同包中的无关类 | √ |  |  |  |

### 7.1代码块

**代码块：在java中用{}括起来的称为代码块**

**1.普通代码块：**定义在方法中的代码块 （使用较少**）**

**public void sayHello(){**

**{**

**System.out.println("普通代码块");**

**}**

**}**

**2.** **构造代码块：**在java类中使用{}声明的代码块（和静态代码块的区别是少了static关键字）

**public class CodeBlock {**

**static{**

**System.out.println("静态代码块");**

**}**

**{**

**System.out.println("构造代码块");**

**}**

**}**

构造块优先于构造方法执行，每产生一个新的对象就调用一次构造块，构造块

可以进行简单的逻辑操作（在调用构造方法前）

**3.** **静态代码块：**使⽤用static定义的代码块

**public class CodeBlock {**

**static{**

**System.out.println("静态代码块");**

**}**

**}**

**1. 静态块优先于构造块执行。**

**2. 无论产⽣多少实例化对象，静态块都只执行一次**

**3.** **静态块的主要作用是为 static 属性进行初始化**

**代码块执行顺序：**

* **静态代码块>构造代码块>构造函数>普通代码块**

**public class CodeBlock {**

**static{**

**System.out.println("静态代码块");**

**}**

**{**

**System.out.println("构造代码块");**

**}**

**public CodeBlock(){**

**System.out.println("无参构造函数");**

**}**

**public void sayHello(){**

**{**

**System.out.println("普通代码块");**

**}**

**}**

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

**System.out.println("执行了main方法");**

**new CodeBlock().sayHello();;**

**System.out.println("---------------");**

**new CodeBlock().sayHello();;**

**}**

**}**

### 7.2内部类：

**定义：**将一个类A定义在另一个类B里面，里面的那个类A就称为内部类，B则称为外部类。

#### 7.2.1实例内部类(成员)

在描述事物时，若一个事物内部还包含其他事物，就可以使用内部类这种结构。比如，学生student

包含类 eat，这时eat就可以使用内部类来描述，定义在成员位置。

**class Student { //外部类**

**class Eat { //内部类**

**}**

**}**

**访问特点**

内部类可以直接访问外部类的成员，包括私有成员。

外部类要访问内部类的成员，必须要建立内部类的对象。

**创建内部类对象格式：**

**外部类名.内部类名 对象名 = new 外部类型().new 内部类型()；**

**注意：内用外，随意访问；外用内，需要内部类对象。**

如何使用成员内部类？有两种方式：

1. 间接方式：在外部类的方法当中，使用内部类；然后main只是调用外部类的方法。

2. 直接方式，公式：

类名称 对象名 = new 类名称();

外部类名称.内部类名称 对象名 = new 外部类名称().new 内部类名称();

**举例**

**public class InnerClass {**

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

**Body body = new Body(); // 外部类的对象**

**// 通过外部类的对象，调用外部类的方法，里面间接在使用内部类Heart**

**body.methodBody();**

**System.out.println("=====================");**

**// 按照公式写：**

**Body.Heart heart = new Body().new Heart();**

**heart.beat();**

**}**

**}**

#### 7.2.2匿名内部类（重点）

**定义：**是内部类的简化写法。它的本质是一个 带具体实现的 父类或者父接口的 匿名的 子类对象。

**前提：**

匿名内部类必须继承一个父类或者实现一个父接口。

**格式：**

**new 父类名或者接口名(){**

**// 方法重写**

**@Override**

**public void method() {**

**// 执行语句**

**}**

**};**

#### 7.2.3 静态内部类

在内部类添加修饰符static，这个内部类就变成了静态内部类，一个静态内部类钟表可以声明static成员，但是在非静态内部类中不可以声明静态成员。静态内部类有一个最大的特点，就是不可以使用外部类的非静态成员。即普通的内部类对象隐式的在外部保存了一个引用，指向创建它的外部类对象，但如果内部类被定义为static，就会有更多的限制，静态内部类具有以下两个特点：

（1）如果创建静态内部类的对象，不需要其他外部类对象；

（2）不能从静态内部类的对象中访问非静态外部类的对象。

**public class StaticinnerClass{**

**int x = 100;**

**static class innner{**

**void doitinner(){**

**}**

**}**

**}**

#### 7.2.4方法内部类

访问仅限于方法内或该作用域内局部内部类就像是方法里面的一个局部变量一样，无修饰符局部内部类只能访问方法中定义的final 类型的局部变量。

**class Outer{**

**private int num =5;**

**public void dispaly(final int temp)**

**{**

**//方法内部类即嵌套在方法里面**

**public class Inner{**

**}**

**}**

**}**

**public class Test{**

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

**{}**

**}**

## 8.继承

**继承性**：就是子类继承父类的属性和行为，使得子类对象具有与父类相同的属性、相同的行为。子类可以直接访问父类中的非私有的属性和行为。

**好处**

1. 提高代码的复用性。

2. 类与类之间产生了关系，是多态的前提。

**1.2 继承的格式**

通过 extends 关键字，可以声明一个子类继承另外一个父类，定义格式如下：

**class 父类 {**

**...**

**}**

**class 子类 extends 父类 {**

**...**

**}**

具体实例：

**/\***

**\* 定义员工类Employee，做为父类**

**\*/**

**class Employee {**

**String name; // 定义name属性**

**// 定义员工的工作方法**

**public void work() {**

**System.out.println("尽心尽力地工作");**

**}**

**}**

**/\***

**\* 定义讲师类Teacher 继承 员工类Employee**

**\*/**

**class Teacher extends Employee {**

**// 定义一个打印name的方法**

**public void printName() {**

**System.out.println("name=" + name);**

**}**

**}**

**/\***

**\* 定义测试类**

**\*/**

**public class ExtendDemo01 {**

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

**// 创建一个讲师类对象**

**Teacher t = new Teacher();**

**// 为该员工类的name属性进行赋值**

**t.name = "小明";**

**// 调用该员工的printName()方法**

**t.printName(); // name = 小明**

**// 调用Teacher类继承来的work()方法**

**t.work(); // 尽心尽力地工作**

**}**

**}**

在父子类的继承关系当中，如果成员变量重名，则创建子类对象时，访问有两种方式：

直接通过子类对象访问成员变量：

* 等号左边是谁，就优先用谁，没有则向上找。

间接通过成员方法访问成员变量：

* 该方法属于谁，就优先用谁，没有则向上找。

局部变量： 直接写成员变量名

本类的成员变量： this.成员变量名

父类的成员变量： super.成员变量名

在父子类的继承关系当中，创建子类对象，访问成员方法的规则：

* 创建的对象是谁，就优先用谁，如果没有则向上找。

**注意事项：**

无论是成员方法还是成员变量，如果没有都是向上找父类，绝对不会向下找子类的。

#### 8.1重写（Override）：

**概念**：在继承关系当中，方法的名称一样，参数列表也一样。

**方法重写** ：子类中出现与父类一模一样的方法时（返回值类型，方法名和参数列表都相同），会出现覆盖效果，也称为重写或者复写。声明不变，重新实现。

重写和重载区别

重写（Override）：方法的名称一样，参数列表【也一样】。覆盖、覆写。

重载（Overload）：方法的名称一样，参数列表【不一样】。.

**class Fu {**

**public void show() {**

**System.out.println("Fu show");**

**}**

**}**

**class Zi extends Fu {**

**//子类重写了父类的show方法**

**public void show() {**

**System.out.println("Zi show");**

**}**

**}**

**public class ExtendsDemo05{**

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

**Zi z = new Zi();**

**// 子类中有show方法，只执行重写后的show方法**

**z.show(); // Zi show**

**}**

方法的覆盖重写特点：创建的是子类对象，则优先用子类方法。

继承关系中，父子类构造方法的访问特点：

#### 8.2super关键字

1. 子类构造方法当中有一个默认隐含的“super()”调用，所以一定是先调用的父类构造，后执行的子类构造。

2. 子类构造可以通过super关键字来调用父类重载构造。

3. super的父类构造调用，必须是子类构造方法的第一个语句。不能一个子类构造调用多次super构造。

**总结**：

子类必须调用父类构造方法，不写则赠送super()；写了则用写的指定的super调用，super只能有一个，还必须是第一个。

**super和this的含义**

super ：代表父类的存储空间标识(可以理解为父亲的引用)。

this ：代表当前对象的引用(谁调用就代表谁)。

**super和this的用法**

1. 访问成员

**this.成员变量 ‐‐ 本类的**

**super.成员变量 ‐‐ 父类的**

**this.成员方法名() ‐‐ 本类的**

**super.成员方法名() ‐‐ 父类的**

2. 访问构造方法

**this(...) ‐‐ 本类的构造方法**

**super(...) ‐‐ 父类的构造方法**

#### 8.3final关键字

**final** ： 不可改变。可以用于修饰类、方法和变量。

类：被修饰的类，不能被继承。

方法：被修饰的方法，不能被重写。

变量：被修饰的变量，不能被重新赋值

**使用方法**

##### 8.3.1.修饰类

**final class 类名 {**

**}**

##### 8.3.2.修饰方法

**修饰符 final 返回值类型 方法名(参数列表){**

**//方法体**

**}**

重写被 final 修饰的方法，编译时就会报错

##### 8.3.3.修饰变量

基本类型的局部变量，被final修饰后，只能赋值一次，不能再更改。

## 9.多态

**多态性：**多态是同一个行为具有多个不同表现形式或形态的能力

**多态存在的三个必要条件**

继承

重写

父类引用指向子类对象

多态体现的格式：

父类类型 变量名 = new 子类对象；

变量名.方法名();

例子：

**public class Test {**

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

**show(new Cat()); // 以 Cat 对象调用 show 方法**

**show(new Dog()); // 以 Dog 对象调用 show 方法**

**Animal a = new Cat(); // 向上转型**

**a.eat(); // 调用的是 Cat 的 eat**

**Cat c = (Cat)a; // 向下转型**

**c.work(); // 调用的是 Cat 的 work**

**}**

**public static void show(Animal a) {**

**a.eat();**

**// 类型判断**

**if (a instanceof Cat) { // 猫做的事情**

**Cat c = (Cat)a;**

**c.work();**

**} else if (a instanceof Dog) { // 狗做的事情**

**Dog c = (Dog)a;**

**c.work();**

**}**

**}**

**}**

**abstract class Animal {**

**abstract void eat();**

**}**

**class Cat extends Animal {**

**public void eat() {**

**System.out.println("吃鱼");**

**}**

**public void work() {**

**System.out.println("抓老鼠");**

**}**

**}**

**class Dog extends Animal {**

**public void eat() {**

**System.out.println("吃骨头");**

**}**

**public void work() {**

**System.out.println("看家");**

**}**

**}**

### 9.1引用类型转换：

多态的转型分为向上转型与向下转型两种：

* 向上转型 ：多态本身是子类类型向父类类型向上转换的过程，这个过程是默认的。当父类引用指向一个子类对象时，便是向上转型。

**使用格式：**

父类类型 变量名 = new 子类类型();

如：Animal a = new Cat();

* 向下转型 ：父类类型向子类类型向下转换的过程，这个过程是强制的。

一个已经向上转型的子类对象，将父类引用转为子类引用，可以使用强制类型转换的格式，便是向下转型。

**使用格式：**

子类类型 变量名 = (子类类型) 父类变量名;

如:Cat c =(Cat) a;

**多态性总结：**

* 对象多态性的核心在于方法的覆写。
* 通过对象的向上转型可以实现接收参数的统一，向下转型可以实现⼦类扩充方法的调用（一般不操作向下转型，有安全隐患）。
* 两个没有关系的类对象是不能够进行转型的，一定会产生ClassCastException。

### 9.2抽象类

**抽象方法**：就是加上abstract关键字，然后去掉大括号，直接分号结束。

格式：

**修饰符 abstract 返回值类型 方法名 (参数列表)；**

**抽象类：**抽象方法所在的类，必须是抽象类才行。在class之前写上abstract即可。

格式

**abstract class 类名字 {**

**}**

**抽象的使用**

**public class Cat extends Animal {**

**public void run (){**

**System.out.println("小猫在墙头走~~~")；**

**}**

**}**

**public class CatTest {**

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

**// 创建子类对象**

**Cat c = new Cat();**

**// 调用run方法**

**c.run();**

**}**

**}**

继承抽象类的子类必须重写父类所有的抽象方法。否则，该子类也必须声明为抽象类。最终，必须有子类实现该父类的抽象方法。

**如何使用抽象类和抽象方法：**

1. 不能直接创建new抽象类对象。

2. 必须用一个子类来继承抽象父类。

3. 子类必须覆盖重写抽象父类当中所有的抽象方法。

覆盖重写（实现）：子类去掉抽象方法的abstract关键字，然后补上方法体大括号。

4. 创建子类对象进行使用。

5. private与abstract不能同时使用。

## 10.接口

**定义：**接口就是抽象方法和全局常量的集合，在Java中接口使用interface关键字定义。

public interface 接口名称 {

// 抽象方法

// 默认方法

// 静态方法

// 私有方法

}

**1.含有抽象方法**

抽象方法：使用 abstract 关键字修饰，可以省略，没有方法体。该方法供子类实现使用。

**2.含有默认方法和静态方法**

* 默认方法：使用 default 修饰，不可省略，供子类调用或者子类重写。
* 静态方法：使用 static 修饰，供接口直接调用。

**3. 含有私有方法和私有静态方法**

私有方法：使用 private 修饰，供接口中的默认方法或者静态方法调用。

**实现方法：**

类与接口的关系为实现关系，即类实现接口，该类可以称为接口的实现类，也可以称为接口的子类。实现的动作类似继承，格式相仿，只是关键字不同，实现使用 implements 关键字。

**非抽象子类实现接口：**

1. 必须重写接口中所有抽象方法。

2. 继承了接口的默认方法，即可以直接调用，也可以重写。

### 10.1抽象方法的使用：

1. 继承默认方法，代码如下：

**定义接口：**

**public interface LiveAble {**

**// 定义抽象方法**

**public abstract void eat();**

**public abstract void sleep();**

**}**

**定义实现类：**

**public class Animal implements LiveAble {**

**@Override**

**public void eat() {**

**System.out.println("吃东西");**

**}**

**@Override**

**public void sleep() {**

**System.out.println("晚上睡");**

**}**

**}**

**定义测试类：**

**public class InterfaceDemo {**

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

**// 创建子类对象**

**Animal a = new Animal();**

**// 调用实现后的方法**

**a.eat();**

**a.sleep();**

**}**

**}**

接口的使用，它不能创建对象，但是可以被实现（ implements ，类似于被继承）。一个实现接口的类（可以看做是接口的子类），需要实现接口中所有的抽象方法，创建该类对象，就可以调用方法了，否则它必须是一个抽象类。.

1、定义接口使用关键字interface 接口的名称使用 I 来表示

2、接口中的数据成员默认为 public static final

接口中的方法默认为 public abstract

3、接口内不能有已经实现的方法。

4、接口不能进行实例化。IMessage imessage = new IMessage();

5、接口可以继承多个接口，此时的接口拥有了继承的接口

里面的抽象方法。

6、如果一个抽象类继承了接口，那么抽象类当中可以不实现接口的方法。但是，如果再有一个普通类继承了此抽象类，那么普通类一定要实现接口和抽象类里面的抽象方法。

### 10.2模板设计模式

**开闭原则(OCP): 一个软件实体如类、模块和函数应该对扩展开放、对修改关闭。**

**模板方法模式包含以下主要角色。**

**(1) 抽象类：**负责给出一个算法的轮廓和骨架。它由一个模板方法和若干个基本方法构成。这些方法的定义如下。

**① 模板方法：定义了算法的骨架，按某种顺序调用其包含的基本方法。**

**② 基本方法：是整个算法中的一个步骤，包含以下几种类型。**

* **抽象方法**：在抽象类中申明，由具体子类实现。
* **具体方法：**在抽象类中已经实现，在具体子类中可以继承或重写它。
* **钩子方法：**在抽象类中已经实现，包括用于判断的逻辑方法和需要子类重写的空方法两种。

**(2) 具体子类**：实现抽象类中所定义的抽象方法和钩子方法，它们是一个顶级逻辑的一个组成步骤。

**例子：**

**public abstract class GoBuying {**

**public final void browseGoods(){**

**System.out.println("1.选择商品");**

**}**

**public final void selectGoods(){**

**System.out.println("2.选择商品");**

**}**

**public final void callService(){**

**System.out.println("3.客服咨询");**

**}**

**public final void computPrice(){**

**System.out.println("4.结算");**

**}**

**public abstract void orderPrice();**

**public abstract void sendService();**

**public final void showOrder(){**

**System.out.println("7.查看订单");**

**}**

**public final void process(){**

**browseGoods();**

**selectGoods();**

**callService();**

**computPrice();**

**orderPrice();**

**sendService();**

**showOrder();**

**}**

**public boolean isCallService(){**

**return true;**

**}**

**}**

**class Jd extends GoBuying{**

**@Override**

**public void orderPrice() {**

**System.out.println("5.您好，欢迎使用支付宝。");**

**}**

**@Override**

**public void sendService() {**

**System.out.println("6.您好，京东为您配送");**

**}**

**public boolean isCallService(){**

**return false;**

**}**

**}**

**class TestDemo02{**

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

**GoBuying goBuying=new Jd();**

**goBuying.process();**

**}**

**}**

### 10.3简单工厂模式

实例化对象的时候不再使用 new Object()形式，可以根据用户的选择条件来实例化相关的类。对于客户端来说，去除了具体的类的依赖。只需要给出具体实例的描述给工厂，工厂就会自动返回具体的实例对象。

代码实现：

**//定义水果 接口**

**public interface Fruit {**

**void plantFruit();**

**void eatFruit();**

**}**

**//苹果实现水果接口**

**public class Apple implements Fruit {**

**@Override**

**public void plantFruit() {**

**System.out.println("种苹果。");**

**}**

**@Override**

**public void eatFruit() {**

**System.out.println("吃苹果。");**

**}**

**}**

**//橘子实现水果接口**

**public class Orange implements Fruit {**

**@Override**

**public void plantFruit() {**

**System.out.println("种橘子。");**

**}**

**@Override**

**public void eatFruit() {**

**System.out.println("吃橘子。");**

**}**

**}**

**//水果工厂**

**public class FruitFactory {**

**public static Fruit getFurit(String fruitName) {**

**//简单工厂模式**

**if (fruitName.equalsIgnoreCase("Apple")) { // 如果是苹果，则返回苹果实例**

**return new Apple();**

**} else if (fruitName.equalsIgnoreCase("Orange")) { // 如果是橘子，则返回橘子实例**

**return new Orange();**

**} else {**

**return null;**

**}**

**}**

**}**

**//测试类**

**public class Test {**

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

**//调用简单工厂模式**

**FruitFactory.getFurit("Orange").plantFruit();**

**}**

**}**

总结

1：我要购买苹果，只需向工厂角色（FruitFactory）请求即可。而工厂角色在接到请求后，会自行判断创建和提供哪一个产品。

2：但是对于工厂角色（FruitFactory ）来说，增加新的产品（比如说增加草莓）就是一个痛苦的过程。工厂角色必须知道每一种产品，如何创建它们，以及何时向客户端提供它们。换言之，接纳新的产品意味着修改这个工厂。

### 10.4工厂方法模式

将对象的创建交由父类中定义的一个标准方法来完成，而不是其构造函数，究竟应该创建何种对象由具体的子类负责决定

**代码实现**

**/定义水果 接口**

**public interface Fruit {**

**void plantFruit();**

**void eatFruit();**

**}**

**//苹果实现水果接口**

**public class Apple implements Fruit {**

**@Override**

**public void plantFruit() {**

**System.out.println("种苹果。");**

**}**

**@Override**

**public void eatFruit() {**

**System.out.println("吃苹果。");**

**}**

**}**

**//橘子实现水果接口**

**public class Orange implements Fruit {**

**@Override**

**public void plantFruit() {**

**System.out.println("种橘子。");**

**}**

**@Override**

**public void eatFruit() {**

**System.out.println("吃橘子。");**

**}**

**}**

**//水果工厂（[b]注意：此工厂声明为一个接口，具有了良好的扩展性[/b]）**

**public interface FactoryMethod {**

**/\*\***

**\* 工厂方法**

**\*/**

**Fruit getFruit(); //定义获取水果这一过程**

**}**

**//苹果对水果工厂的实现**

**public class getApple implements FactoryMethod{**

**@Override**

**public Fruit getFruit() {**

**// TODO Auto-generated method stub**

**return new Apple();**

**}**

**}**

**//橘子对水果工厂的实现**

**public class getOrange implements FactoryMethod {**

**@Override**

**public Fruit getFruit() {**

**// TODO Auto-generated method stub**

**return new Orange();**

**}**

**}**

**//测试类**

**public class TestFactoryMethod {**

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

**getApple apple = new getApple();**

**apple.getFruit().eatFruit();**

**}**

**}**

**输出：吃苹果。**

**总结：**

A：工厂方法模式和简单工厂模式在结构上的不同是很明显的。工厂方法模式的核心是一个抽象工厂类，而简单工厂模式把核心放在一个具体类上。工厂方法模式可以允许很多具体工厂类从抽象工厂类中将创建行为继承下来，从而可以成为多个简单工厂模式的综合，进而推广了简单工厂模式。

B：工厂方法模式退化后可以变得很像简单工厂模式。设想如果非常确定一个系统只需要一个具体工厂类，那么就不妨把抽象工厂类合并到具体的工厂类中去。由于反正只有一个具体工厂类，所以不妨将工厂方法改成为静态方法，这时候就得到了简单工厂模式。

C：如果需要加入一个新的水果，那么只需要加入一个新的水果类以及它所对应的工厂类。没有必要修改客户端，也没有必要修改抽象工厂角色或者其他已有的具体工厂角色。对于增加新的水果类而言，这个系统完全支持"开-闭"原则。

### 10.5抽象工厂模式

抽象工厂模式可以说是简单工厂模式的扩展，它们主要的区别在于需要创建对象的复杂程度上。

在抽象工厂模式中，抽象产品可能是一个或多个，从而构成一个或多个产品族。 在只有一个产品族的情况下，抽象工厂模式实际上退化到工厂方法模式。

举例：

**//定义水果 接口**

**public interface Fruit {**

**void plantFruit();**

**void eatFruit();**

**}**

**//苹果实现水果接口**

**public class Apple implements Fruit {**

**@Override**

**public void plantFruit() {**

**System.out.println("种苹果。");**

**}**

**@Override**

**public void eatFruit() {**

**System.out.println("吃苹果。");**

**}**

**}**

**//橘子实现水果接口**

**public class Orange implements Fruit {**

**@Override**

**public void plantFruit() {**

**System.out.println("种橘子。");**

**}**

**@Override**

**public void eatFruit() {**

**System.out.println("吃橘子。");**

**}**

**}**

**//定义 肉 的接口**

**public interface Meat {**

**void buyMeat();**

**void eatMeat();**

**}**

**//猪肉实现接口**

**public class BigMeat implements Meat {**

**@Override**

**public void buyMeat() {**

**System.out.println("买猪肉。");**

**}**

**@Override**

**public void eatMeat() {**

**System.out.println("吃猪肉。");**

**}**

**}**

**//牛肉实现接口**

**public class CowMeat implements Meat{**

**@Override**

**public void buyMeat() {**

**System.out.println("买牛肉。");**

**}**

**@Override**

**public void eatMeat() {**

**System.out.println("吃牛肉。");**

**}**

**}**

**//定义消费者的接口**

**public interface UserFactory {**

**/\*\***

**\* 水果工厂方法**

**\*/**

**public Fruit getFruit(Fruit whichFruit);**

**/\*\***

**\* 肉的工厂方法**

**\*/**

**public Meat getMeat(Meat whichMeat);**

**}**

**//实现消费者的接口**

**public class User implements UserFactory {**

**@Override**

**public Fruit getFruit(Fruit whichFruit) {**

**return whichFruit;**

**}**

**@Override**

**public Meat getMeat(Meat whichMeat) {**

**return whichMeat;**

**}**

**}**

**//测试类**

**public class TestUserFactory {**

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

**Fruit apple=new Apple();**

**Meat bigMeat=new BigMeat();**

**//消费者实例**

**User me=new User();**

**me.getFruit(apple).eatFruit();**

**me.getMeat(bigMeat).buyMeat();**

**}**

**}**

**总结：**

A：抽象工厂模式可以向客户端提供一个接口，使得客户端在不必指定产品的具体类型的情况下，创建多个产品族中的产品对象。这就是抽象工厂模式的用意。

B：抽象工厂模式是所有形态的工厂模式中最为抽象和最具一般性的一种形态。

抽象类与接⼝口的区别

|  |  |  |  |
| --- | --- | --- | --- |
|  | 区别 | 抽象类 | 接口 |
| 1 | 结构组成 | 普通类+抽象方法 | 抽象方法+全局变量 |
| 2 | 权限 | 各种权限 | Public |
| 3 | 子类使用 | 使用extends关键字 | 使用implements关键字实现接口 |
| 4 | 关系 | 一个抽象类可以实现若干接口 | 接口不能继承抽象类，但接口可以使extends关键字继承多个父接口 |
| 5 | 子类限制 | 一个子类只能继承一个抽象类 | 一个子类可以实现多个接口 |

## 11.三大特殊类

### 11.1.String类

**常用方法：**

**1.判断功能的方法**

**public boolean equals (Object anObject) ：**将此字符串与指定对象进行比较。

**public boolean equalsIgnoreCase (String anotherString) ：**将此字符串与指定对象进行比较，忽略大小写

**2.获取功能的方法**

**public int length () ：**返回此字符串的长度。

**public String concat (String str) ：**将指定的字符串连接到该字符串的末尾。

**public char charAt (int index) ：**返回指定索引处的 char值。

**public int indexOf (String str) ：**返回指定子字符串第一次出现在该字符串内的索引。

**public String substring (int beginIndex) ：**返回一个子字符串，从beginIndex开始截取字符串到字符

串结尾。

**public String substring (int beginIndex, int endIndex)** ：返回一个子字符串，从beginIndex到

endIndex截取字符串。含beginIndex，不含endIndex。

**3.转换功能的方法**

**public char[] toCharArray () ：**将此字符串转换为新的字符数组。

**public byte[] getBytes () ：**使用平台的默认字符集将该 String编码转换为新的字节数组。

**public String replace (CharSequence target, CharSequence replacement) ：**将与target匹配的字符串使用replacement字符串替换。

4.分割功能的方法

**public String[] split(String regex) ：**将此字符串按照给定的regex（规则）拆分为字符串数组。

String StringBuffer StringBuilder的区别和联系

StringBuffer 和 StringBuilder 也是操作字符串的两个类

1、StringBuffer===》synchronized 但是StringBuilder没有

StringBuffer多线程情况下使用，synchronized线程安全的关键字

StringBuilder没有 String 没有 单线程情况下 线程不安全

2、拼接上：String每次都会产生新的空间

而StringBuffer StringBuilder ==》append() ==>不会产生新的空间

3、String的拼接底层被优化为StringBuilder

append进行拼接 结果将会调用StringBuilder的toString()

String str = "hello";

str = str+"bit";

System.out.println(str);

**String类 == 与 equals 的区别**

1. ”==”：进行的数值比较，比较的是两个字符串对象的内存地址数值。

2. “equals（）”：可以进行字符串内容的比较

### 11.2.Object类

Object类是一个特殊的类，是所有类的父类，如果一个类没有用extends明确指出继承于某个类，那么它默认继承Object类。Object类中的三个常用方法：toString()、equals()、hashCode()

1.取得对象信息的方法：toString()

class Student

**{**

**String name = "Mary";**

**int age = 21;**

**public String toString()**

**{**

**return "姓名："+name+"，年龄："+age;**

**}**

**}**

2.对象相等判断方法：equals()

该方法用于比较对象是否相等，而且此方法必须被重写。

**class Student**

**{**

**String name;**

**int age;**

**public Student(String name,int age)**

**{**

**this.name=name;**

**this.age=age;**

**}**

**}**

**public class Text{**

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

**{**

**Student s1 = new Student("Mary",21);**

**Student s2 = new Student("Mary",21);**

**System.out.println(s1.equals(s2));//输出一个boolean值**

**System.out.println(s1.equals(s2)?"s1和s2是同一个人":"s1和s2不是同一个人");//?:条件运算符**

**}**

**}**

3.对象签名:hashCode()

该方法用来返回其所在对象的物理地址（哈希码值），常会和equals方法同时重写，确保相等的两个对象拥有相等的.hashCode。

**class Student**

**{**

**String name;**

**int age;**

**//重写父类（Object类）中的equals方法**

**public boolean equals()**

**{**

**boolean temp;**

**Student s1 = new Student();**

**s1.name="张三";s1.age=12;**

**Student s2 = new Student();**

**s2.name="张三";s2.age=12;**

**System.out.println("s1的哈希码："+s1.hashCode());**

**System.out.println("s2的哈希码："+s2.hashCode());**

**if((s1.name.equals(s2.name))&&(s1.age==s2.age))**

**{**

**temp = true;**

**}**

**else**

**{**

**temp = false;**

**}**

**return temp;**

**}**

**//重写hashCode()方法**

**public int hashCode()**

**{**

**return age\*(name.hashCode());**

**}**

**}**

**public class Text{**

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

**{**

**Student s3 = new Student();**

**System.out.println(s3.equals()?"s1和s2是同一人":"s1和s2不是同一人");**

**}**

**}**

### 11.3.包装类

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

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

基本类型和对应的包装类可以相互装换：

* 由基本类型向对应的包装类转换称为装箱，例如把 int 包装成 Integer 类的对象；
* 包装类向对应的基本类型转换称为拆箱，例如把 Integer 类的对象重新简化为 int。

手动装箱：

// 手动装箱

Integer a = new Integer(10);

// 手动拆箱

int b = a.intValue();

自动装箱：

// 自动装箱

Integer a = 10;

// 自动拆箱

int b = a;

Integer与int的区别(包装类型和基本数据类型的区别):

1):默认值:

int的默认值是0.

Integer的默认值为null.

推论:Integer既可以表示null,又可以表示0.

## 12设计模式

### 12.1. 单例设计模式

定义：是一个类只允许产生一个实例化对象。

**基本的实现思路**

单例模式要求类能够有返回对象一个引用(永远是同一个)和一个获得该实例的方法（必须是静态方法，通常使用getInstance这个名称）。

单例的实现主要是通过以下两个步骤：

* 将该类的构造方法定义为私有方法，这样其他处的代码就无法通过调用该类的构造方法来实例化该类的对象，只有通过该类提供的静态方法来得到该类的唯一实例；
* 在该类内提供一个静态方法，当我们调用这个方法时，如果类持有的引用不为空就返回这个引用，如果类保持的引用为空就创建该类的实例并将实例的引用赋予该类保持的引用。

#### 1.1懒汉式单例模式

特点：当第一次去使用Singleton对象的时候才会为其产生实例化对象的操作.

**class Singleton{**

**private static Singleton instance ;**

**private Singleton() { // private声明构造**

**}**

**public static Singleton getInstance() {**

**if (instance==null) { // 表示此时还没有实例例化**

**instance = new Singleton() ;**

**}**

**return instance ;**

**}**

**public void print() {**

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

**}**

**}**

**public class SingletonTest {**

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

**Singleton singleton = null ; // 声明对象**

**singleton = Singleton.getInstance() ;**

**singleton.print();**

**}**

**}**

#### 1.2、饿汉式（静态常量）

**public class Singleton {**

**private final static Singleton INSTANCE = new Singleton();**

**private Singleton(){}**

**public static Singleton getInstance(){**

**return INSTANCE;**

**}**

**}**

#### 1.3饿汉式（静态代码块）

**public class Singleton {**

**private static Singleton instance;**

**static {**

**instance = new Singleton();**

**}**

**private Singleton() {}**

**public static Singleton getInstance() {**

**return instance;**

**}**

**}**

### 12.2. 多例设计模式

**多例模式的定义**：

作为对象的创建模式，多例模式中的多例类可以有多个实例，而且多例类必须自己创建、管理自己的实例，并向外界提供自己的实例。

**多例模式的特点：**

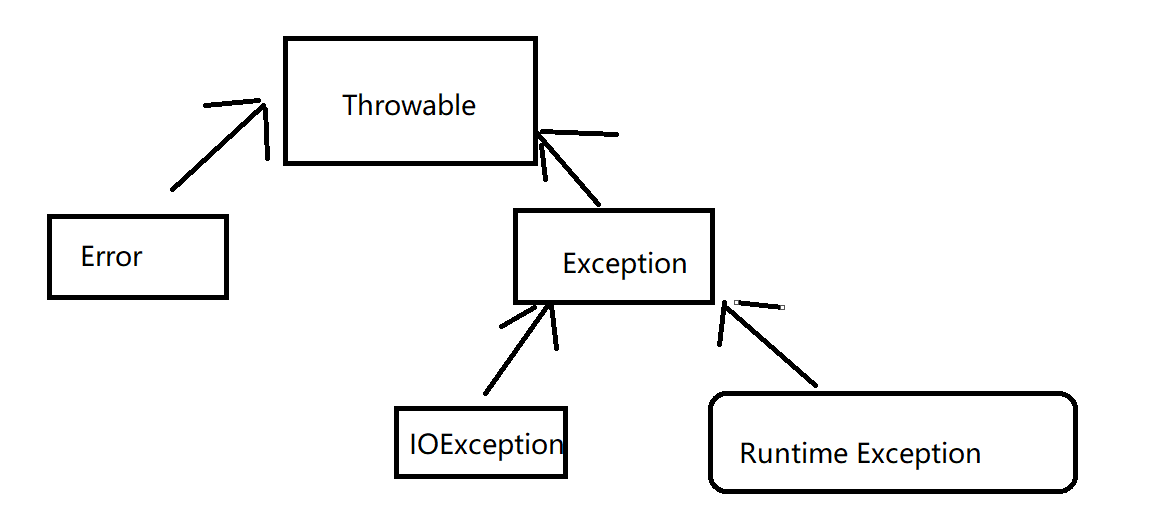
所谓的多例模式（Multiton Pattern），实际上就是单例模式的自然推广。作为对象的创建模式，多例模式或多例类有如下的特点：

（1）多例类可有多个实例

（2）多例类必须自己创建、管理自己的实例，并向外界提供自己的实例。

（3）根据是否有实例上限分为：有上限多例类和无上限多例类。

## 13.异常与捕获



### 13.1java常见异常：

|  |  |
| --- | --- |
| 异常 | 描述 |
| ArithmeticException | 当出现异常的运算条件时，抛出此异常。例如，一个整数"除以零"时，抛出此类的一个实例。 |
| ArrayIndexOutOfBoundsException | 用非法索引访问数组时抛出的异常。如果索引为负或大于等于数组大小，则该索引为非法索引。 |
| ArrayStoreException | 试图将错误类型的对象存储到一个对象数组时抛出的异常。 |
| ClassCastException | 当试图将对象强制转换为不是实例的子类时，抛出该异常。 |
| IllegalArgumentException | 抛出的异常表明向方法传递了一个不合法或不正确的参数。 |
| IllegalMonitorStateException | 抛出的异常表明某一线程已经试图等待对象的监视器，或者试图通知其他正在等待对象的监视器而本身没有指定监视器的线程。 |
| IllegalStateException | 在非法或不适当的时间调用方法时产生的信号。换句话说，即 Java 环境或 Java 应用程序没有处于请求操作所要求的适当状态下。 |
| IllegalThreadStateException | 线程没有处于请求操作所要求的适当状态时抛出的异常。 |
| IndexOutOfBoundsException | 指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。 |
| NegativeArraySizeException | 如果应用程序试图创建大小为负的数组，则抛出该异常。 |
| NullPointerException | 当应用程序试图在需要对象的地方使用 null 时，抛出该异常 |
| NumberFormatException | 当应用程序试图将字符串转换成一种数值类型，但该字符串不能转换为适当格式时，抛出该异常。 |
| SecurityException | 由安全管理器抛出的异常，指示存在安全侵犯。 |
| StringIndexOutOfBoundsException | 此异常由 String 方法抛出，指示索引或者为负，或者超出字符串的大小。 |
| UnsupportedOperationException | 当不支持请求的操作时，抛出该异常。 |

### 13.2捕获异常

使用 try 和 catch 关键字可以捕获异常。try/catch 代码块放在异常可能发生的地方。

try/catch代码块中的代码称为保护代码，使用 try/catch 的语法如下：

try

{

// 程序代码

}catch(ExceptionName e1)

{

//Catch 块

}

**多重捕获块**

一个 try 代码块后面跟随多个 catch 代码块的情况就叫多重捕获。

多重捕获块的语法如下所示：

try{

// 程序代码

}catch(异常类型1 异常的变量名1){

// 程序代码

}catch(异常类型2 异常的变量名2){

// 程序代码

}catch(异常类型2 异常的变量名2){

// 程序代码

}

#### throws/throw 关键字：

如果一个方法没有捕获到一个检查性异常，那么该方法必须使用 throws 关键字来声明。throws 关键字放在方法签名的尾部。

也可以使用 throw 关键字抛出一个异常，无论它是新实例化的还是刚捕获到的。

**public class className**

**{**

**public void deposit(double amount) throws RemoteException**

**{**

**// Method implementation**

**throw new RemoteException();**

**}**

**//Remainder of class definition**

**}**

**解释throw和throws的区别**

1. throw⽤用于⽅方法内部，主要表示⼿手⼯工异常抛出。

2. throws主要在方法声明上使用，明确告诉用户本方法可能产生的异常，同时该方法可能不处

理此异常。

#### finally关键字

finally 关键字用来创建在 try 代码块后面执行的代码块。

无论是否发生异常，finally 代码块中的代码总会被执行。

在 finally 代码块中，可以运行清理类型等收尾善后性质的语句。

finally 代码块出现在 catch 代码块最后，语法如下：

try{

// 程序代码

}catch(异常类型1 异常的变量名1){

// 程序代码

}catch(异常类型2 异常的变量名2){

// 程序代码

}finally{

// 程序代码

}

例子：

**public class ExcepTest{**

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

**int a[] = new int[2];**

**try{**

**System.out.println("Access element three :" + a[3]);**

**}catch(ArrayIndexOutOfBoundsException e){**

**System.out.println("Exception thrown :" + e);**

**}**

**finally{**

**a[0] = 6;**

**System.out.println("First element value: " +a[0]);**

**System.out.println("The finally statement is executed");**

**}**

**}**

**}**

# 数据结构

## 1.时间复杂度和空间复杂度

### 1.1时间复杂度

算法中的基本操作的执行次数，为算法的时间复杂度。

计算时间复杂度时，我们其实并不一定要计算精确的执行次数，而只需要大概执行次数，那么这

里我们使用大O的渐进表示法。

**大O符号（Big O notation）：是用于描述函数渐进行为的数学符号。**

1、用常数1取代运行时间中的所有加法常数。

2、在修改后的运行次数函数中，只保留最高阶项。

3、如果最高阶项存在且不是1，则去除与这个项目相乘的常数。得到的结果就是大O阶。

**1.常数阶O(1)**

无论代码执行了多少行，只要是没有循环等复杂结构，那这个代码的时间复杂度就都是O(1)，

代码在执行的时候，它消耗的时候并不随着某个变量的增长而增长，那么无论这类代码有多长，即使有几万几十万行，都可以用O(1)来表示它的时间复杂度。

**2. 线性阶O(n)**

for循环里面的代码会执行n遍，因此它消耗的时间是随着n的变化而变化的，因此这类代码都可以用O(n)来表示它的时间复杂度。

**3.对数阶O(logN)**

**int i = 1;**

**while(i<n)**

**{**

**i = i \* 2;**

**}**

在while循环里面，每次都将 i 乘以 2，乘完之后，i 距离 n 就越来越近了。我们试着求解一下，假设循环x次之后，i 就大于 2 了，此时这个循环就退出了，也就是说 2 的 x 次方等于 n，那么 x = log2^n

也就是说当循环 log2^n 次以后，这个代码就结束了。因此这个代码的时间复杂度为：O(logn)

**4. 线性对数阶O(nlogN)**

**for(m=1; m<n; m++)**

**{**

**i = 1;**

**while(i<n)**

**{**

**i = i \* 2;**

**}**

**}**

线性对数阶O(nlogN) 其实非常容易理解，将时间复杂度为O(logn)的代码循环N遍的话，那么它的时间复杂度就是 n \* O(logN)，也就是了O(nlogN)。

**5. 平方阶O(n²)**

看有几个for循环

### 1.2.空间复杂度

空间复杂度是对一个算法在运行过程中临时占用存储空间大小的量度 。空间复杂度不是程序占用了多少bytes的空间，因为这个也没太大意义，所以空间复杂度算的是变量的个数。空间复杂度计算规则基本跟实践复杂度类似，也使用大O渐进表示法。

**1.空间复杂度 O(1)**

如果算法执行所需要的临时空间不随着某个变量n的大小而变化，即此算法空间复杂度为一个常量，可表示为 O(1)

**int i = 1;**

**int j = 2;**

**++i;**

**j++;**

**int m = i + j;**

代码中的 i、j、m 所分配的空间都不随着处理数据量变化，因此它的空间复杂度 S(n) = O(1)

**2. 空间复杂度 O(n)**

**int[] m = new int[n]**

**for(i=1; i<=n; ++i)**

**{**

**j = i;**

**j++;**

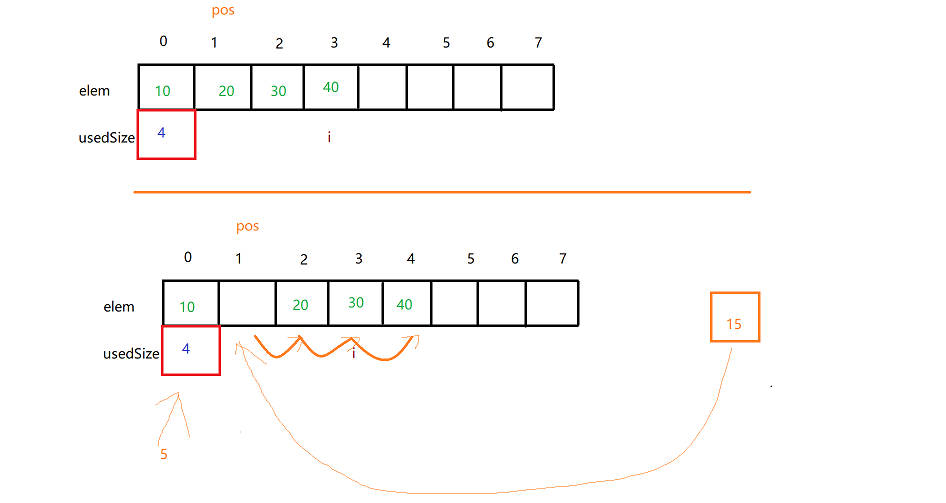
**}**

这段代码中，第一行new了一个数组出来，这个数据占用的大小为n，这段代码的2-6行，虽然有循环，但没有再分配新的空间，因此，这段代码的空间复杂度主要看第一行即可，即 S(n) = O(n)

## 2.顺序表

### 2.1顺序表

线性表的顺序存储是指用一组地址连续的存储单元依次存储线性表中的各个元素，使得线性表在逻辑结构上相邻的元素存储在连续的物理存储单元中，即：通过数据元素物理存储的连续性来反应元素之间逻辑上的相邻关系。采用顺序存储结构存储的线性表通常简称为顺序表。



代码实现：

//顺序表

**public interface ISequence {**

**//在pos位置插入data值**

**boolean add(int pos, Object data);**

**//查找关键字key 找到返回key的下标，没有返回-1;**

**int search(Object key);**

**//查找是否包含关键字key是否在顺序表当中(这个和search有点冲突)**

**boolean contains(Object key);**

**//得到pos位置的值**

**Object getPos(int pos);**

**// 删除第一次出现的关键字key**

**Object remove(Object key);**

**//得到顺序表的长度**

**int size();**

**// 打印顺序表**

**void display();**

**// 清空顺序表以防内存泄漏**

**void clear();**

**}**

**实现过程：**

**public class MySequence implements ISequence {**

**//顺序表应该有什么属性，1.数组2.usedSize**

**private Object[] elem;//放data的都是Object**

**private int useSize;**

**//默认的顺序表的容量**

**private static final int DEFAULT\_SIZE = 10;**

**//提供一个构造方法**

**public MySequence() {**

**this.elem = new Object[DEFAULT\_SIZE];**

**this.useSize = 0;**

**}**

**//判断pos是否是满的**

**public boolean isFull() {**

**if (this.useSize == this.elem.length - 1) {**

**return true;**

**}**

**return false;**

**}**

**@Override**

**//往pos位置放data**

**public boolean add(int pos, Object data) {**

**//1.pos的合法性 2.是否是满的顺序表**

**if (pos < 0 || pos > this.useSize) {**

**return false;**

**}**

**//如果是满的，把数组长度扩大为原来的二倍。**

**//Arrays.copyOf(this.elem,this.elem.length\*2)前面原来数组，后面是数组长度\*2再给elem**

**if (isFull()) {**

**this.elem = Arrays.copyOf(this.elem, this.elem.length \* 2);**

**}**

**//必须从后面挪数据。定义一个i，让i+1=i i相当于pos位置。**

**for (int i = this.useSize - 1; i >= pos; i--) {**

**this.elem[i + 1] = this.elem[i];**

**}**

**//放入data数据到pos**

**this.elem[pos] = data;**

**useSize++;**

**return true;**

**}**

**//判断是否为空**

**/\*public boolean isEmpty() {**

**if (this.useSize==0){**

**return true;**

**}**

**return false;**

**}\*/**

**// 还可以这样写**

**public boolean isEmpty() {**

**return this.useSize == 0;//判断usedSize是否为0，是返回ture，不是返回false。**

**}**

**@Override**

**//查找关键字找到后返回下标**

**public int search(Object key) {**

**//找不到返回-1**

**if (key == null) {**

**return -1;**

**}**

**if (isEmpty())**

**//抛一个异常**

**throw new UnsupportedOperationException("顺序表为空");**

**//遍历顺序表。**

**for (int i = 0; i < this.useSize; i++) {**

**//引用类型必须用equals不能==。**

**if (this.elem[i].equals(key)) {**

**return i;**

**}**

**}**

**return -1;**

**}**

**@Override**

**//查找是否包含关键字key是否在顺序表当中**

**public boolean contains(Object key) {**

**if (key == null) {**

**return false;**

**}**

**if (isEmpty()) {**

**throw new UnsupportedOperationException("顺序表为空");**

**}**

**for (int i = 0; i < this.useSize; i++) {**

**if (this.elem[i].equals(key)) {**

**return true;**

**}**

**}**

**return false;**

**}**

**@Override**

**//得到pos位置的值,一定要严谨**

**public Object getPos(int pos) {**

**if (pos < 0 || pos >= this.useSize || isEmpty()) {**

**return null;**

**}**

**return this.elem[pos];**

**}**

**// 删除第一次出现的关键字key**

**//删除之前，保存需要删除的值作为返回值。**

**@Override**

**public Object remove(Object key) {**

**//删除之前先找有没有这个值。**

**int index = search(key);**

**if (index == -1) {**

**return null;**

**}**

**Object oldData=this.elem[index];**

**int i = 0;**

**for (i = index; i < useSize - 1; i++) {**

**this.elem[i] = this.elem[i + 1];**

**}**

**this.elem[i + 1] = null;**

**this.useSize--;**

**return oldData;**

**}**

**@Override**

**//得到顺序表的长度**

**public int size() {**

**return this.useSize;**

**}**

**@Override**

**// 打印顺序表**

**public void display() {**

**for (int i = 0; i < this.useSize; i++) {**

**System.out.print(this.elem[i] + " ");**

**}**

**System.out.println();**

**}**

**@Override**

**public void clear() {**

**for (int i = 0; i < this.useSize; i++) {**

**this.elem[i] = null;**

**}**

**//引用类型置为空**

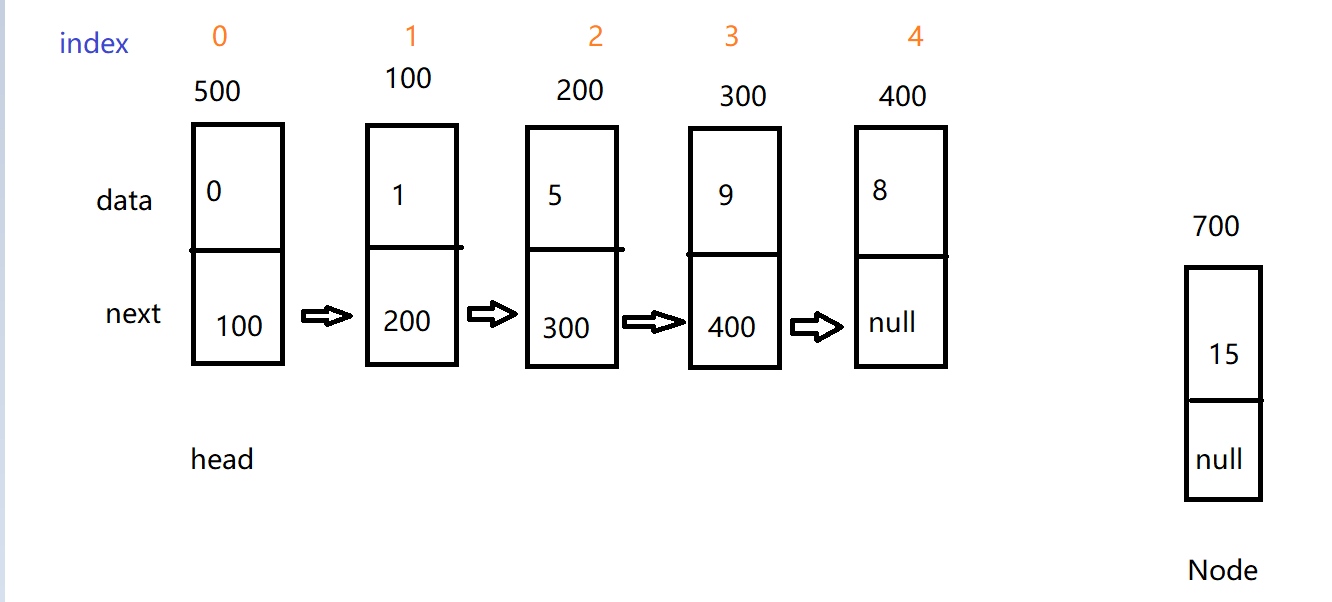
**this.useSize=0;**

**}**

**}**

## 3.链表

### 3.1无头单向非循环链表



代码实现：

**需求代码：**

**public interface ILinked {**

**//头插法**

**void addFirst(int data);**

**//尾插法**

**void addLast(int data);**

**//任意位置插入,第一个数据节点为0号下标**

**boolean addIndex(int index, int data);**

**//查找是否包含关键字key是否在单链表当中**

**boolean contains(int key);**

**//删除第一次出现关键字为key的节点**

**int remove(int key);**

**//删除所有值为key的节点**

**void removeAllKey(int key);**

**//得到单链表的长度**

**int getLength();**

**//打印数组**

**void display();**

**//清空数组**

**void clear();**

**}**

**实现功能代码**

**public class MySingListImpl implements ILinked {**

**class Node {**

**private int data;**

**private Node next;**

**public Node(int data) {**

**this.data = data;**

**this.next = null;**

**}**

**public Node getNext() {**

**return next;**

**}**

**}**

**private Node head;**

**public MySingListImpl() {**

**this.head = null;**

**}**

**@Override**

**//头插法**

**public void addFirst(int data) {**

**Node node = new Node(data);**

**if (this.head == null) {**

**this.head = node;**

**} else {**

**node.next = this.head;**

**this.head = node;**

**}**

**}**

**@Override**

**//尾插法**

**public void addLast(int data) {**

**Node node = new Node(data);**

**if (this.head == null) {**

**this.head = node;**

**} else {**

**Node cur = this.head;**

**while (cur.next != null) {**

**cur = cur.next;**

**}**

**cur.next = node;**

**}**

**}**

**//检查index合法性**

**private void checkIndex(int index) {**

**if (index < 0 || index > getLength()) {**

**throw new UnsupportedOperationException("index不合法");**

**}**

**}**

**private Node searchIndex(int index) {**

**checkIndex(index);**

**Node cur = this.head;**

**for (int i = 0; i < index - 1; i++) {**

**cur = cur.next;**

**}**

**return cur;**

**}**

**@Override**

**public boolean addIndex(int index, int data) {**

**if (index == 0) {**

**addFirst(data);**

**return true;**

**}**

**Node node = new Node(data);**

**Node cur = searchIndex(index);**

**node.next = cur.next;**

**return true;**

**}**

**@Override**

**public boolean contains(int key) {**

**Node cur = this.head;**

**while (cur != null) {**

**if (cur.data == key) {**

**return true;**

**}**

**cur = cur.next;**

**}**

**return false;**

**}**

**@Override**

**public int remove(int key) {**

**int oldData = 0;**

**Node pre = searchIndex(key);**

**if (pre == null) {**

**throw new UnsupportedOperationException("不存在key节点");**

**}**

**if (pre == head && pre.data == key) {**

**oldData = this.head.data;**

**this.head = this.head.next;**

**return oldData;**

**}**

**Node del = pre.next;**

**oldData = del.data;//先保存**

**pre.next = del.next;//删除节点**

**return oldData;**

**}**

**@Override**

**public void removeAllKey(int key) {**

**if (this.head == null) {**

**return;**

**}**

**Node pre = this.head;**

**Node cur = this.head.next;**

**while (cur != null) {**

**if (cur.data == key) {**

**pre.next = cur.next;**

**cur = cur.next;**

**} else {**

**pre = cur;**

**cur = cur.next;**

**}**

**}**

**if (this.head.data == key) {**

**this.head = this.head.next;**

**}**

**}**

**@Override**

**public int getLength() {**

**int count = 0;**

**Node cur = this.head;**

**while (cur != null) {**

**count++;**

**cur = cur.next;**

**}**

**return count;**

**}**

**@Override**

**public void display() {**

**Node cur = this.head;**

**while (cur != null) {**

**System.out.print(cur.data + " ");**

**cur = cur.next;**

**}**

**System.out.println();**

**} }**

**@Override**

**public void removeAllKey(int key) {**

**if (this.head == null) {**

**return;**

**}**

**Node pre = this.head;**

**Node cur = this.head.next;**

**while (cur != null) {**

**if (cur.data == key) {**

**pre.next = cur.next;**

**cur = cur.next;**

**} else {**

**pre = cur;**

**cur = cur.next;**

**}**

**}**

**if (this.head.data == key) {**

**this.head = this.head.next;**

**}**

**} @Override**

**public int getLength() {**

**int count = 0;**

**Node cur = this.heat;**

**while (cur != null) {**

**count++;**

**cur = cur.next;**

**}**

**return count;**

**}**

**@Override**

**public void display() {**

**Node cur = this.head;**

**while (cur != null) {**

**System.out.print(cur.data + " ");**

**cur = cur.next;**

**}**

**System.out.println();**

**}**

**@Override**

**public void clear() {**

**Node cur = this.head;**

**while (this.head != null) {**

**this.head.next = null;**

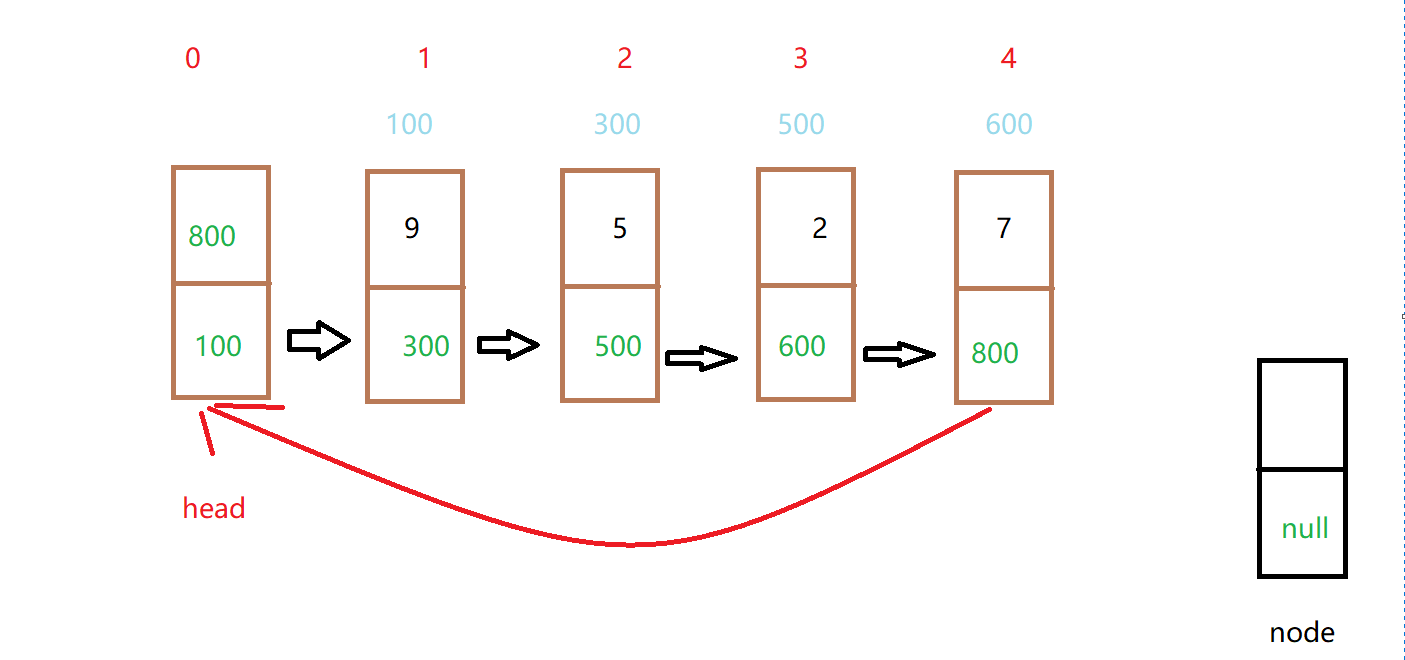
**this.head = cur;**

**cur = cur.next;**

**}**

**}**

### 3.2带头节点循环单链表



**需求代码：**

**public interface ICLinked {**

**//头插法**

**void addFirst(int data);**

**//尾插法**

**void addLast(int data);**

**//任意位置插入,第一个数据节点为0号下标**

**boolean addIndex(int index,int data);**

**//查找是否包含关键字key是否在单链表当中**

**boolean contains(int key);**

**//删除第一次出现关键字为key的节点**

**int remove(int key);**

**//删除所有值为key的节点**

**void removeAllKey(int key);**

**//得到单链表的长度**

**int getLength();**

**void display();**

**void clear();**

**}**

**实现功能代码**

**public class HeadSingleListImpl implements ICLinked{**

**//节点类**

**class Node {**

**private int data;**

**private Node next;**

**//头结点**

**public Node() {**

**this.data = -1;**

**}**

**//数据节点**

**public Node(int data) {**

**this.data = data;**

**}**

**}**

**private Node head;**

**public HeadSingleListImpl() {**

**this.head = new Node();**

**this.head.next = this.head;**

**}**

**@Override**

**public void addFirst(int data) {**

**Node node = new Node(data);**

**node.next = this.head.next;**

**this.head.next = node;**

**}**

**@Override**

**public void addLast(int data) {**

**Node cur = this.head;**

**while (cur.next != this.head) {**

**cur = cur.next;**

**}**

**Node node = new Node(data);**

**node.next = cur.next;**

**cur.next = node;**

**}**

**//index-1的位置**

**private Node searchIndex(int index) {**

**checkIndex(index);**

**Node cur = this.head;**

**for (int i = 0; i < index; i++) {**

**cur = cur.next;**

**}**

**return cur;**

**}**

**private void checkIndex(int index) {**

**if(index < 0 || index > getLength()) {**

**throw new UnsupportedOperationException("index不合法");**

**}**

**}**

**@Override**

**public boolean addIndex(int index,int data) {**

**Node pre = searchIndex(index);**

**Node node = new Node(data);**

**node.next = pre.next;**

**pre.next = node;**

**return true;**

**}**

**@Override**

**public boolean contains(int key) {**

**Node cur = this.head.next;**

**while(cur != this.head) {**

**if(cur.data == key) {**

**return true;**

**}**

**cur = cur.next;**

**}**

**return false;**

**}**

**//如果找不到返回空**

**private Node searchPre(int key) {**

**Node pre = this.head;**

**//......**

**while(pre.next != this.head) {**

**if(pre.next.data == key) {**

**return pre;**

**}**

**pre = pre.next;**

**}**

**return null;**

**}**

**@Override**

**public int remove(int key) {**

**Node pre = searchPre(key);**

**if(pre == null) {**

**throw new UnsupportedOperationException("没有" +**

**"key这个关键字");**

**}**

**Node del = pre.next;**

**int oldData = del.data;**

**pre.next = del.next;**

**return oldData;**

**}**

**@Override**

**public void removeAllKey(int key) {**

**Node pre = this.head;**

**Node cur = this.head.next;**

**while(cur != this.head) {**

**if(cur.data == key) {**

**pre.next = cur.next;**

**cur = pre.next;**

**}else{**

**pre = cur;**

**cur = cur.next;**

**}**

**}**

**}**

**@Override**

**public int getLength() {**

**int count = 0;**

**//cur指向第一个数据节点**

**Node cur = this.head.next;**

**while(cur != this.head) {**

**count++;**

**cur = cur.next;**

**}**

**return count;**

**}**

**@Override**

**public void display() {**

**Node cur = this.head.next;**

**while(cur != this.head) {**

**System.out.print(cur.data+" ");**

**cur = cur.next;**

**}**

**System.out.println();**

**}**

**@Override**

**public void clear() {**

**//防止内存泄漏**

**while(this.head.next != this.head) {**

**Node cur = this.head.next;**

**this.head.next = cur.next;**

**}**

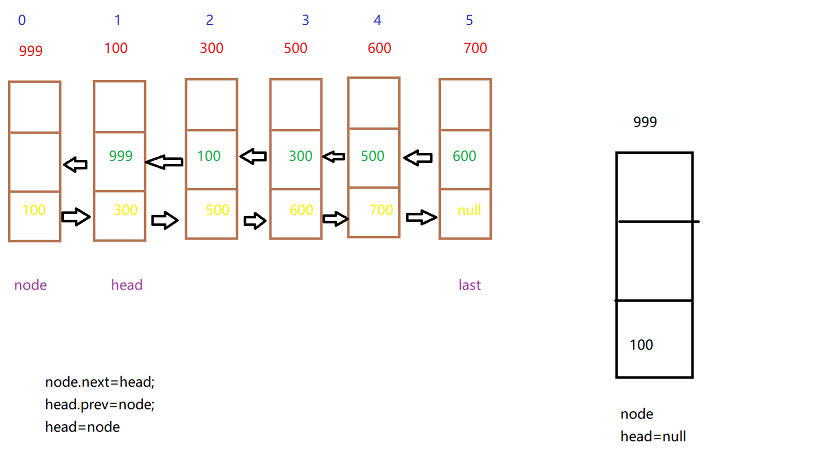
**//头结点置为空**

**this.head = null;**

**}**

**}**

### 3.3不带头节点双向链表



**需求代码：**

**public interface IDoubleLinked {**

**//头插法**

**void addFirst(int data);**

**//尾插法**

**void addLast(int data);**

**//任意位置插入,第一个数据节点为0号下标**

**boolean addIndex(int index,int data);**

**//查找是否包含关键字key是否在单链表当中**

**boolean contains(int key);**

**//删除第一次出现关键字为key的节点**

**int remove(int key);**

**//删除所有值为key的节点**

**void removeAllKey(int key);**

**//得到单链表的长度**

**int getLength();**

**void display();**

**void clear();**

**}**

**实现功能代码**

**public class DoubleLinkedListImpl implements IDoubleLinked{**

**class Node {**

**private int data;**

**private Node next;//后继**

**private Node prev;//前驱**

**public Node(int data) {**

**this.data = data;**

**this.next = null;**

**this.prev = null;**

**}**

**}**

**private Node head;**

**private Node last;**

**public DoubleLinkedListImpl() {**

**this.head = null;**

**this.last = null;**

**}**

**@Override**

**public void addFirst(int data) {**

**Node node = new Node(data);**

**if(this.head == null) {**

**this.head = node;**

**this.last = node;**

**}else {**

**node.next = this.head;**

**this.head.prev = node;//**

**this.head = node;**

**}**

**}**

**@Override**

**public void addLast(int data) {**

**Node node = new Node(data);**

**if(this.head == null) {**

**this.head = node;**

**this.last = node;**

**}else {**

**this.last.next = node;**

**node.prev = this.last;**

**this.last = node;**

**}**

**}**

**private Node searchIndex(int index){**

**checkIndex(index);**

**int count = 0;**

**Node cur = this.head;**

**while(count < index) {**

**cur = cur.next;**

**count++;**

**}**

**return cur;**

**}**

**private void checkIndex(int index) {**

**if(index < 0 || index > getLength()) {**

**throw new UnsupportedOperationException("Index不合法");**

**}**

**}**

**@Override**

**public boolean addIndex(int index, int data) {**

**if(index == 0) {**

**addFirst(data);**

**return true;**

**}**

**if(index == getLength()) {**

**addLast(data);**

**return true;**

**}**

**Node node = new Node(data);**

**Node cur = searchIndex(index);**

**node.next = cur;**

**node.prev = cur.prev;**

**node.prev.next = node;**

**node.next.prev = node;**

**return false;**

**}**

**@Override**

**public boolean contains(int key) {**

**Node cur = this.head;**

**while(cur != null) {**

**if(cur.data == key){**

**return true;**

**}**

**cur = cur.next;**

**}**

**return false;**

**}**

**@Override**

**public int remove(int key) {**

**Node cur = this.head;**

**while(cur != null) {**

**if(cur.data == key) {**

**int oldData = cur.data;**

**//要删除的节点是头结点**

**if(cur == this.head) {**

**this.head = this.head.next;**

**this.head.prev = null;**

**}else {**

**cur.prev.next = cur.next;**

**//cur.next != null表示删除的不是**

**//尾节点**

**if(cur.next != null) {**

**cur.next.prev = cur.prev;**

**}else {**

**//尾节点删除，last需要移动**

**this.last = cur.prev;**

**}**

**}**

**return oldData;**

**}**

**cur = cur.next;**

**}**

**return -1;**

**}**

**@Override**

**public void removeAllKey(int key) {**

**Node cur = this.head;**

**while(cur != null) {**

**if(cur.data == key) {**

**//要删除的节点是头结点**

**if(cur == this.head) {**

**this.head = this.head.next;**

**this.head.prev = null;**

**}else {**

**cur.prev.next = cur.next;**

**//cur.next != null表示删除的不是**

**//尾节点**

**if(cur.next != null) {**

**cur.next.prev = cur.prev;**

**}else {**

**//尾节点删除，last需要移动**

**this.last = cur.prev;**

**}**

**}**

**}**

**cur = cur.next;**

**}**

**}**

**@Override**

**public int getLength() {**

**int count = 0;**

**Node cur = this.head;**

**while(cur != null) {**

**count++;**

**cur =cur.next;**

**}**

**return count;**

**}**

**@Override**

**public void display() {**

**Node cur = this.head;**

**while(cur != null) {**

**System.out.print(cur.data+" ");**

**cur = cur.next;**

**}**

**System.out.println();**

**}**

**@Override**

**public void clear() {**

**while(this.head.next != null) {**

**Node cur = this.head.next;**

**this.head.next = cur.next;**

**cur.prev = null;**

**}**

**this.head = null;**

**this.last = null;**

**}**

**}**

## 4.栈和队列

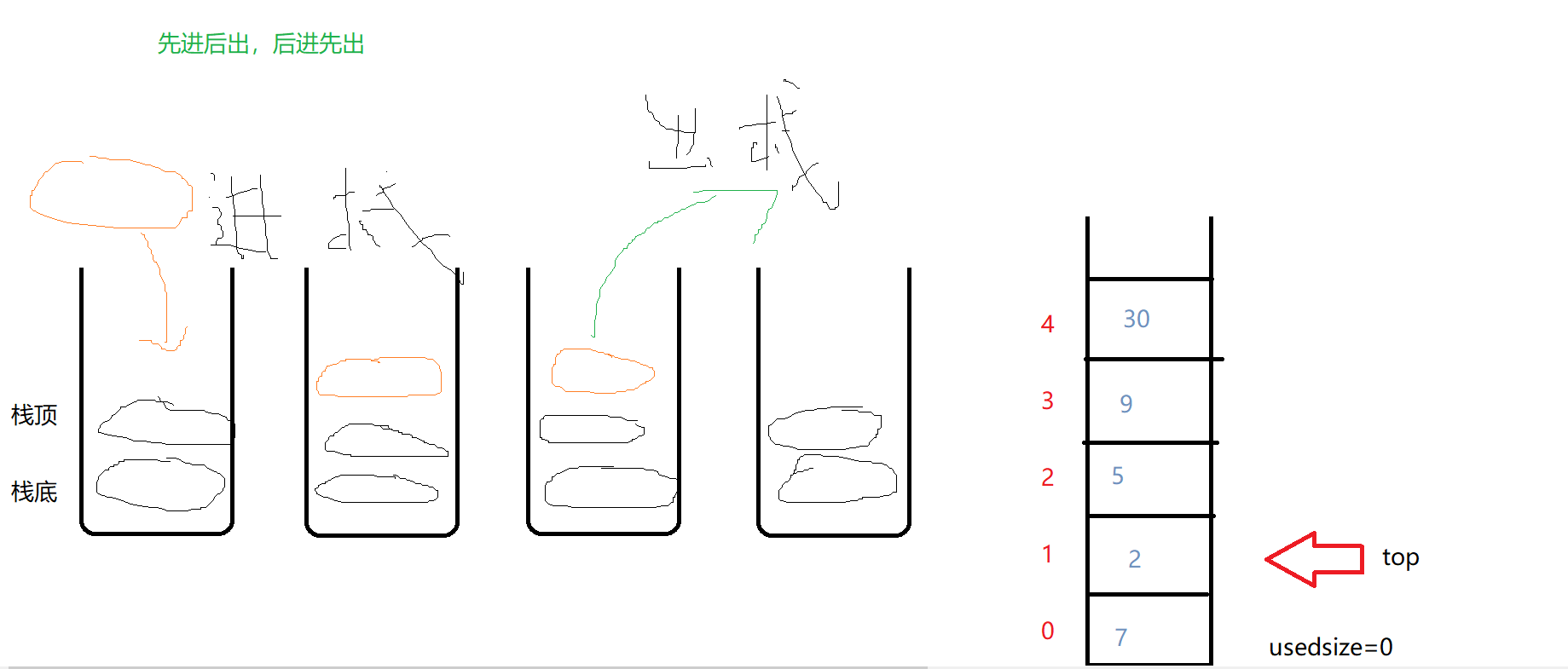
### 4.1栈

栈是一种特殊的线性表，仅能在线性表的一端操作，栈顶允许操作，栈底不允许操作。

**栈的特性：**栈的先进后出，后进先出。

**栈操作数据元素的方法**

1. 入栈 数据元素用栈的数据结构存储起来，也叫压栈
2. 出栈 将数据元素从栈中取出来，也叫弹栈



**需求代码**

**public interface IMyStack {**

**// 将 item 压入栈中**

**void push(int item);**

**// 返回栈顶元素，并且出栈**

**int pop();**

**// 返回栈顶元素，但不出栈**

**int peek();**

**// 判断这个栈是否为空栈**

**boolean empty();**

**// 返回元素个数**

**int size();**

**}**

**实现代码**

**public class MyStackImpl implements IMyStack {**

**private int top;//永远指向可以存放数据元素的下标**

**private int[] elem;//数组**

**private int usedSize;//数据元素的个数**

**public MyStackImpl(int size) {**

**this.top = 0;**

**this.elem = new int[size];**

**this.usedSize = 0;**

**/\*this(10);\*/**

**}**

**private boolean isFull() {**

**return this.usedSize == this.elem.length;**

**}**

**@Override**

**public void push(int item) {**

**if(isFull()){**

**throw new UnsupportedOperationException("" +**

**"栈为满");**

**}**

**this.elem[this.top++] = item;**

**this.usedSize++;**

**}**

**@Override**

**public int pop() {**

**if(empty()){**

**throw new UnsupportedOperationException("栈为空");**

**}**

**int data = this.elem[this.top-1];**

**this.top--;**

**this.usedSize--;**

**return data;**

**}**

**@Override**

**public int peek() {**

**if(empty()){**

**throw new UnsupportedOperationException("栈为空");**

**}**

**return this.elem[this.top-1];**

**}**

**@Override**

**public boolean empty() {**

**return this.top == 0;**

**}**

**@Override**

**public int size() {**

**return this.usedSize;**

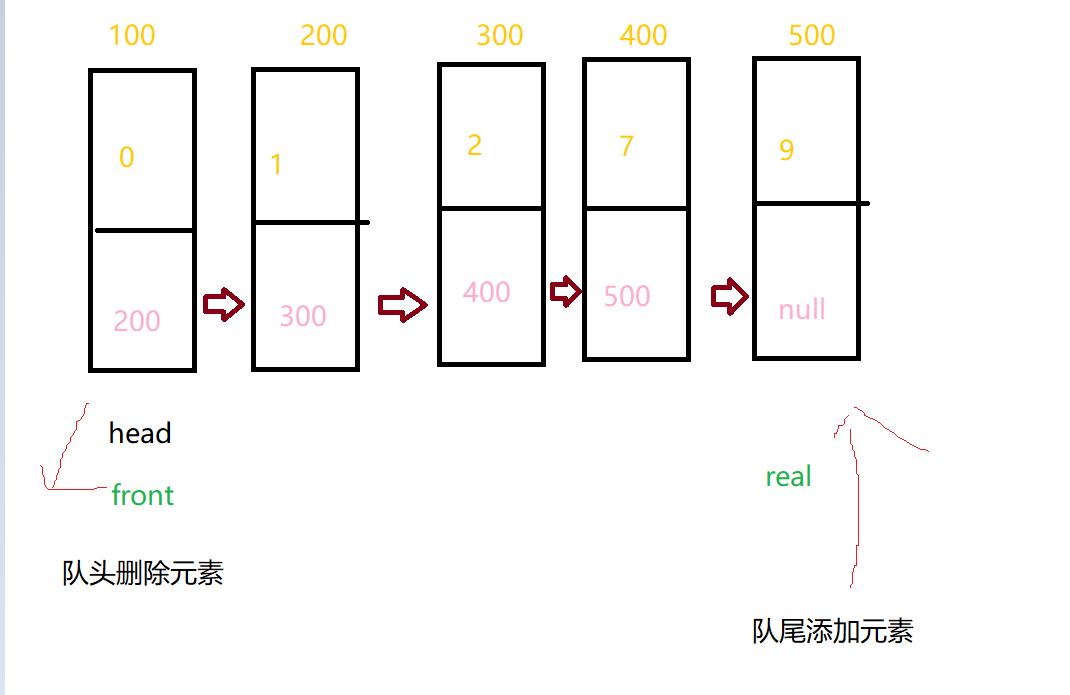
**}**

**}**

### 4.2队列

一种只允许在一端进行插入，在另一端进行删除的线性表结构。运行插入的一端叫队尾（rear），允许删除的一端叫队头（font）。

先进先出，后进后出。



**需求代码**

**public interface IMyQueue {**

**// 判断这个队列是否为空**

**boolean empty();**

**// 返回队首元素，但不出队列**

**int peek();**

**// 返回队首元素，并且出队列**

**int poll();**

**// 将 item 放入队列中**

**void add(int item);**

**// 返回元素个数**

**int size();**

**}**

**实现代码**

**public class MyQueueImpl implements IMyQueue{**

**class Node {**

**private int data;**

**private Node next;**

**public Node(int data) {**

**this.data = data;**

**}**

**}**

**private Node front;**

**private Node rear;**

**private int usedSize;**

**public MyQueueImpl (){**

**this.front = null;**

**this.rear = null;**

**this.usedSize = 0;**

**}**

**@Override**

**public boolean empty() {**

**return this.usedSize == 0;**

**}**

**@Override**

**public int peek() {**

**if(empty()){**

**return -1;**

**}**

**return this.front.data;**

**}**

**@Override**

**public int poll() {**

**if(empty()){**

**return -1;**

**}**

**int data = this.front.data;**

**if(this.usedSize == 1) {**

**this.front = null;**

**this.rear = null;**

**}else {**

**this.front = this.front.next;**

**}**

**this.usedSize--;**

**return data;**

**}**

**@Override**

**public void add(int item) {**

**Node node = new Node(item);**

**if(empty()) {**

**this.front = node;**

**this.rear = node;**

**}else {**

**this.rear.next = node;**

**this.rear = node;**

**}**

**this.usedSize++;**

**}**

**@Override**

**public int size() {**

**return this.usedSize;**

**}**

**}**

## 5.排序

### 5.1插入排序

#### 5.1.1直接插入排序

**直接插入排序基本思想是**：每一步将一个待排序的记录，插入到前面已经排好序的有序序列中去，直到插完所有元素为止。

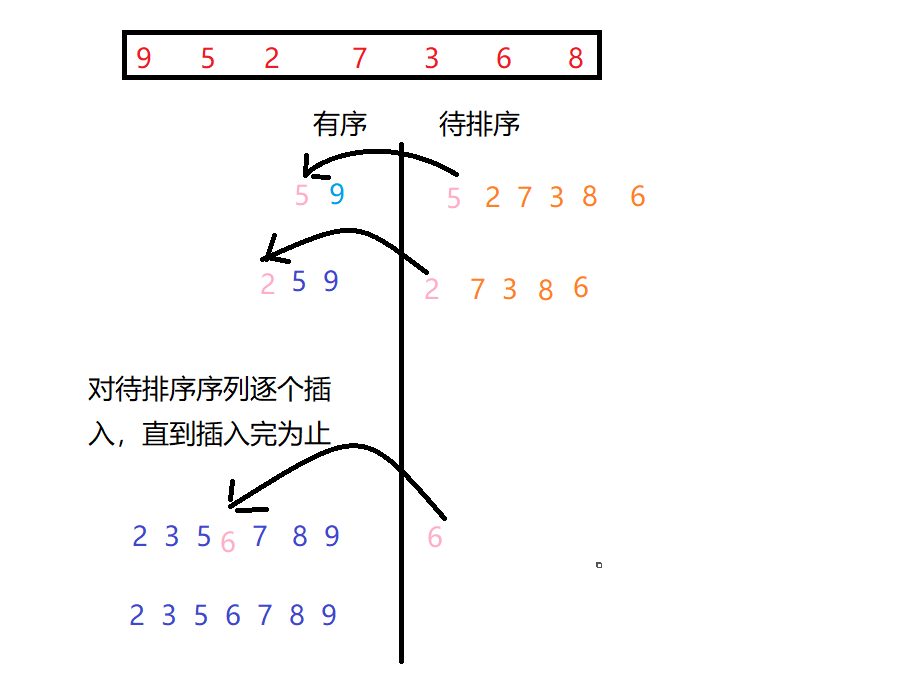
**直接插入排序的特性总结：**

1. 元素集合越接近有序，直接插入排序算法的时间效率越高

2. 时间复杂度：O(N^2)

3. 空间复杂度：O(1)，它是一种稳定的排序算法

4. 稳定性：稳定



**public static void insertSort(int[] array) {**

**//直接插入排序，越有序越快。**

**int temp = 0;**

**for (int i = 0; i < array.length; i++) {**

**temp = array[i];**

**int j = 0;**

**for (j = i - 1; j >= 0; j--) {**

**if (array[j] > temp) {**

**array[j + 1] = array[j];**

**} else {**

**//array[j+1]=temp;可要可不要。**

**break;**

**}**

**}**

**array[j + 1] = temp;**

**}**

**}**

#### 5.1.2希尔排序

**希尔排序的特性总结：**

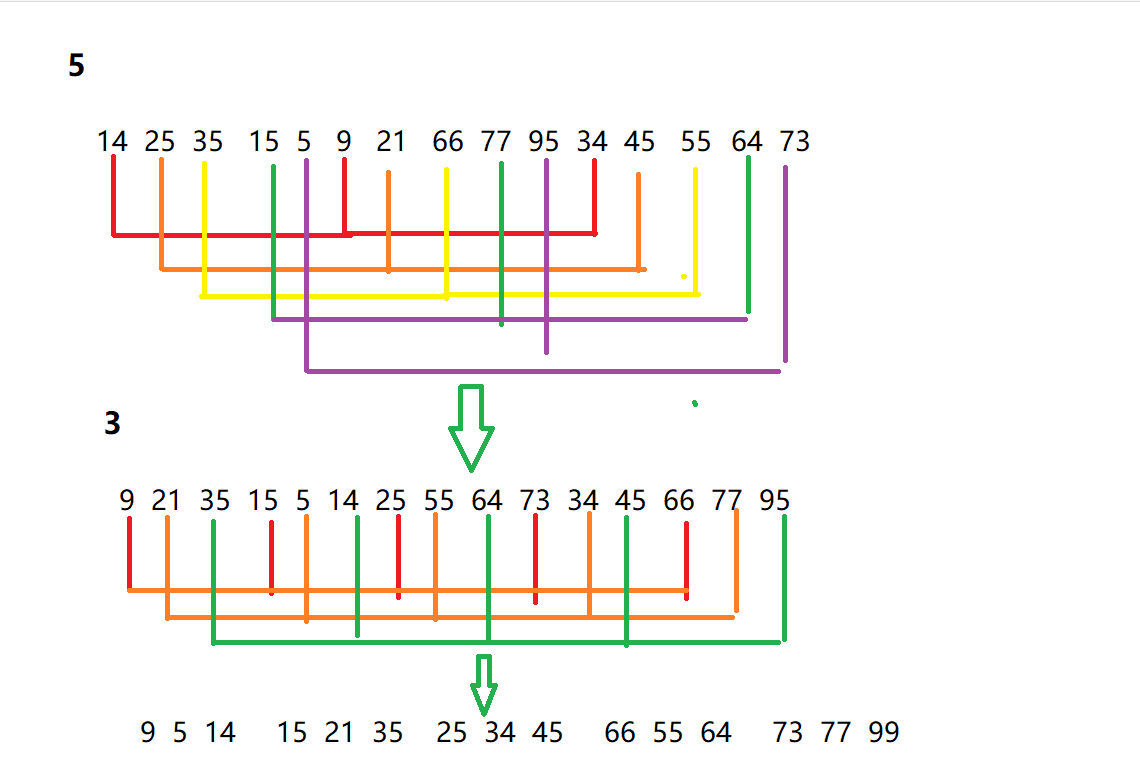
1. 希尔排序是对直接插入排序的优化。

2. 当gap > 1时都是预排序，目的是让数组更接近于有序。当gap == 1时，数组已经接近有序的了，这样就

会很快。这样整体而言，可以达到优化的效果。我们实现后可以进行性能测试的对比。

3. 希尔排序的时间复杂度不好计算，需要进行推导，推导出来平均时间复杂度： O(N^1.3—N^2）

4. 稳定性：不稳定



**public static void shell(int[] array, int gap) {**

**int temp = 0;**

**for (int i = gap; i < array.length; i++) {**

**int j = 0;**

**temp=array[i];**

**for (j = i - gap; j >= 0; j -= gap) {**

**if (array[j] > temp) {**

**array[j + gap] = array[j];**

**} else {**

**break;**

**}**

**}**

**array[j + gap] = temp;**

**}**

### 5.2选择排序

#### 5.2.1直接选择排序

**基本思想：**

每一次从待排序的数据元素中选出最小（或最大）的一个元素，存放在序列的起始位置，直到全部待排序的

数据元素排完

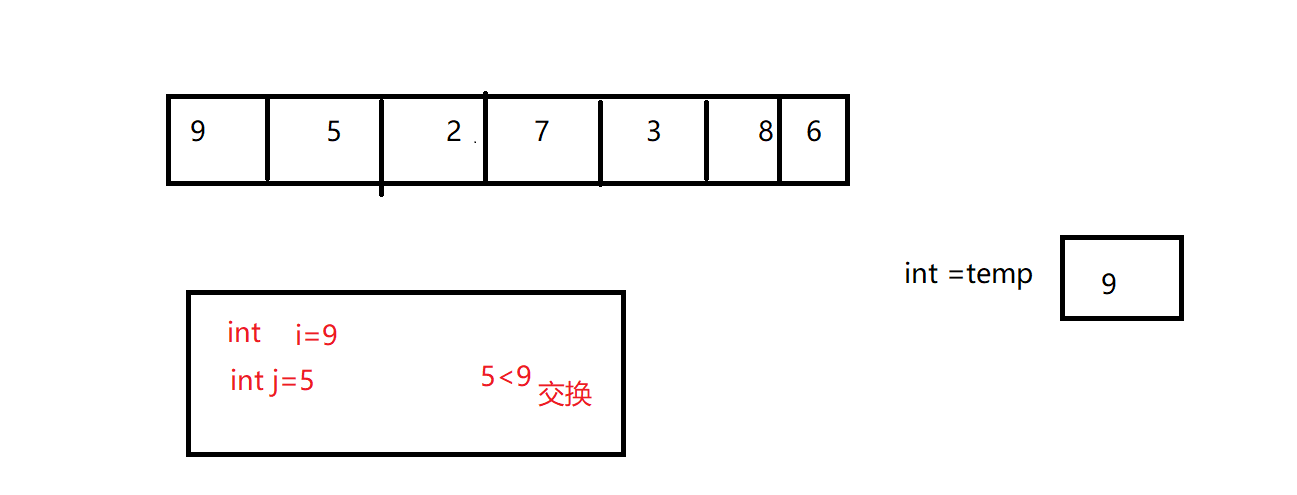
**直接选择排序的特性总结：**

1. 直接选择排序思考非常好理解，但是效率不是很好。实际中很少使用

2. 时间复杂度：O(N^2)

3. 空间复杂度：O(1)

4. 稳定性：不稳定



**public static void selsctSort(int[] array) {**

**for (int i = 0; i < array.length; i++) {**

**for (int j = i + 1; j < array.length; j++) {**

**int temp = 0;**

**if (array[j] < array[i]) {**

**temp = array[j];**

**array[j] = array[i];**

**array[i] = temp;**

**}**

**}**

**}**

**}**

#### 5.2.2堆排序

堆排序是利用堆这种数据结构而设计的一种排序算法，堆排序是一种选择排序，它的最坏，最好，平均时间复杂度均为O(nlogn)，它也是不稳定排序。

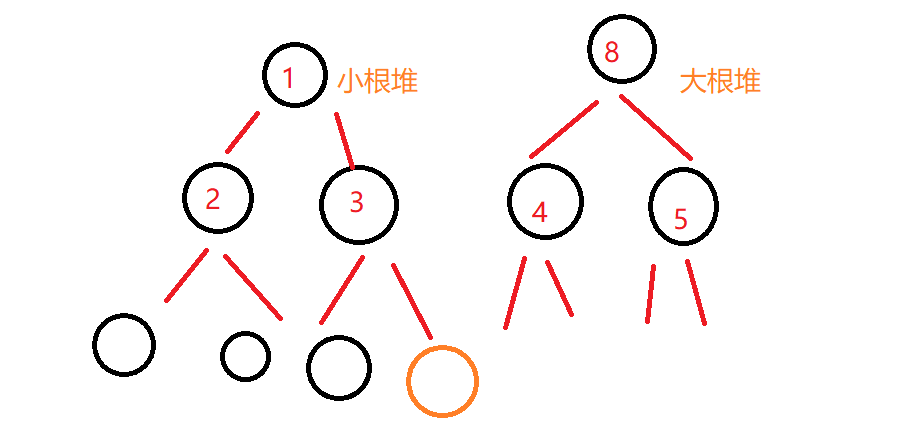
**直接选择排序的特性总结：**

1. 堆排序使用堆来选数，效率就高了很多。

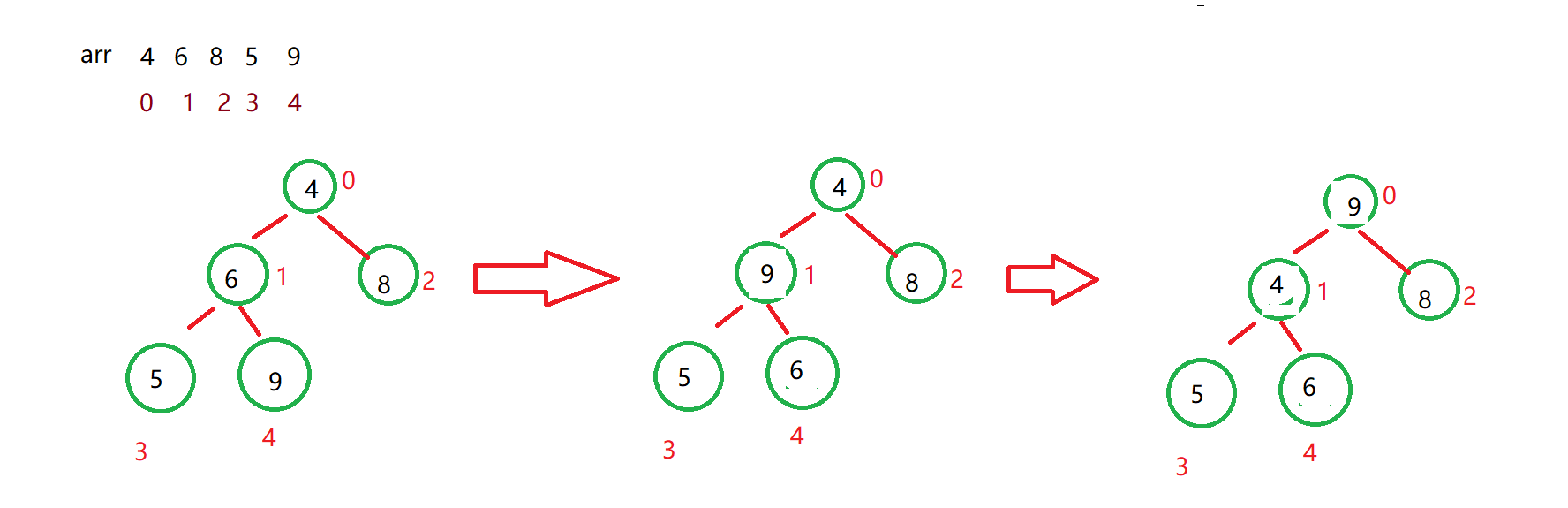
2. 时间复杂度：O(N\*logN)

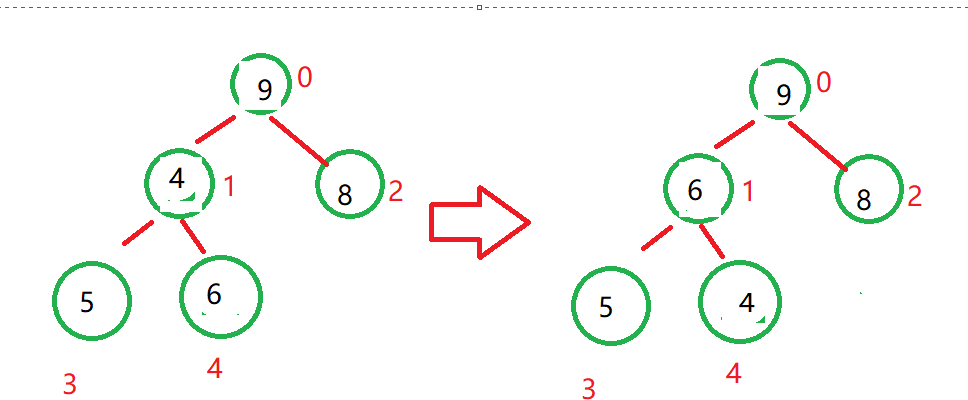
3. 空间复杂度：O(1)

4. 稳定性：不稳定

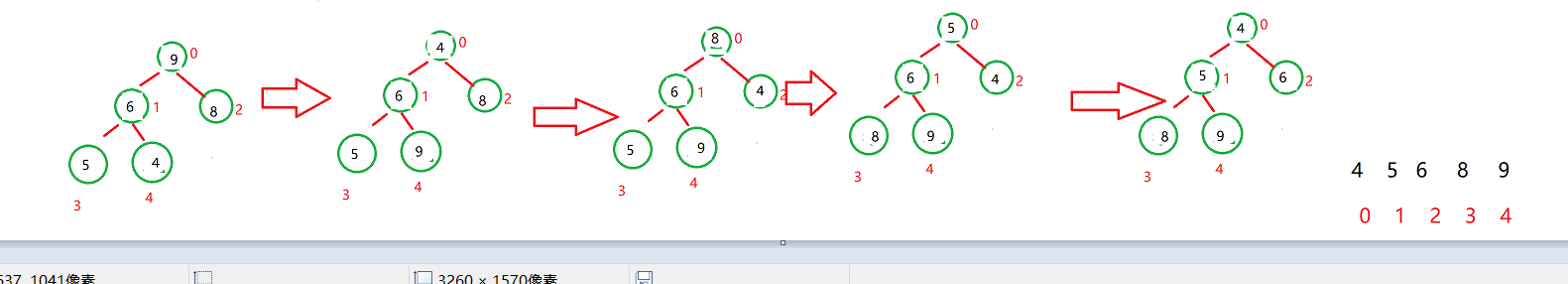


**步骤一 构造初始堆。将给定无序序列构造成一个大顶堆（一般升序采用大顶堆，降序采用小顶堆)。**





**步骤二 将堆顶元素与末尾元素进行交换，使末尾元素最大。然后继续调整堆，再将堆顶元素与末尾元素交换，得到第二大元素。如此反复进行交换、重建、交换。**



**public static void adjust(int[] array, int start, int end) {**

**int temp = array[start];**

**for (int i = 2 \* start + 1; i <= end; i = 2 \* i + 1) {**

**if (i < end && array[i] < array[i + 1]) {**

**i++;**

**}**

**if (array[i] > temp) {**

**array[start] = array[i];**

**start = i;**

**} else {**

**break;**

**}**

**array[start] = temp;**

**}**

**public static void heapSort(int[] array) {**

**for (int i = (array.length - 1 - 1) / 2; i >= 0; i--) {//n/2**

**adjust(array, i, array.length - 1);//log2n**

**}**

**//整棵树调整为大根堆的时间复杂度？nlog2n**

**//交换**

**for (int j = 0; j < array.length - 1; j++) {**

**int tmp = array[array.length - 1 - j];**

**array[array.length - 1 - j] = array[0];**

**array[0] = tmp;**

**//array.length-1-j -1:上面已经交换说明有序了**

**//调整的时候，不需要调整有序的数据。**

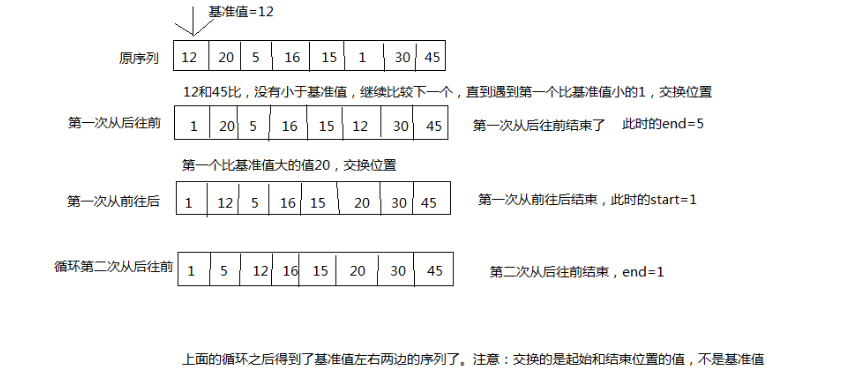
**adjust(array, 0, array.length - 1 - j - 1);**

**}**

**}**

### 5.5快速排序

**快速排序的原理：**选择一个关键值作为基准值。比基准值小的都在左边序列（一般是无序的），比基准值大的都在右边（一般是无序的）。一般选择序列的第一个元素。



**public class TestFastSort {**

**public static int partion(int[] array, int low, int high) {**

**int tmp = array[low];**

**while (low < high) {**

**while (low < high && array[high] >= tmp) {**

**high--;**

**}**

**if (low >= high) {**

**break;**

**} else {**

**array[low] = array[high];**

**}**

**while ((low < high) && array[low] <= tmp) {**

**low++;**

**}**

**if (low >= high) {**

**break;**

**} else {**

**array[high] = array[low];**

**}**

**}**

**array[low] = tmp;**

**return low;**

**}**

**public static void insertSort1(int[] array,**

**int start, int end) {**

**int tmp = 0;**

**//5 21**

**for (int i = start + 1; i <= end; i++) {**

**tmp = array[i];**

**int j = 0;**

**for (j = i - 1; j >= start; j--) {**

**if (array[j] > tmp) {**

**array[j + 1] = array[j];**

**} else {**

**break;**

**}**

**}**

**array[j + 1] = tmp;**

**}**

**}**

**public static void swap(int[] array, int start, int end) {**

**int temp = array[start];**

**array[start] = array[end];**

**array[end] = temp;**

**}**

**public static void medianOfThree(int[] array, int low, int high) {**

**int mid = (low + high) >> 1;**

**if (array[mid] > array[high]) {**

**swap(array, array[mid], array[high]);**

**}**

**if (array[low] > array[high]) {**

**swap(array, array[low], array[high]);**

**}**

**if (array[mid] > array[low]) {**

**swap(array, array[mid], array[low]);**

**}**

**}**

**public static void quick(int[] array, int start, int end) {**

**//1 2 3 4 5**

**if (end - start + 1 <= 16) {**

**insertSort1(array, start, end);**

**return;**

**}**

**int par = partion(array, start, end);**

**//找左边是否有两个数据以上**

**if (par > start + 1) {**

**quick(array, start, par - 1);**

**}**

**//右边是否有两个数据以上**

**if (par < end - 1) {**

**quick(array, par + 1, end);**

**}**

**}**

### 5.6归并排序

**归并排序** 是利用归并的思想实现的排序方法，该算法采用经典的分治（divide-and-conquer）策略（分治法将问题分成一些小的问题然后递归求解，而治的阶段则将分的阶段得到的各答案"修补"在一起，即分而治之)。

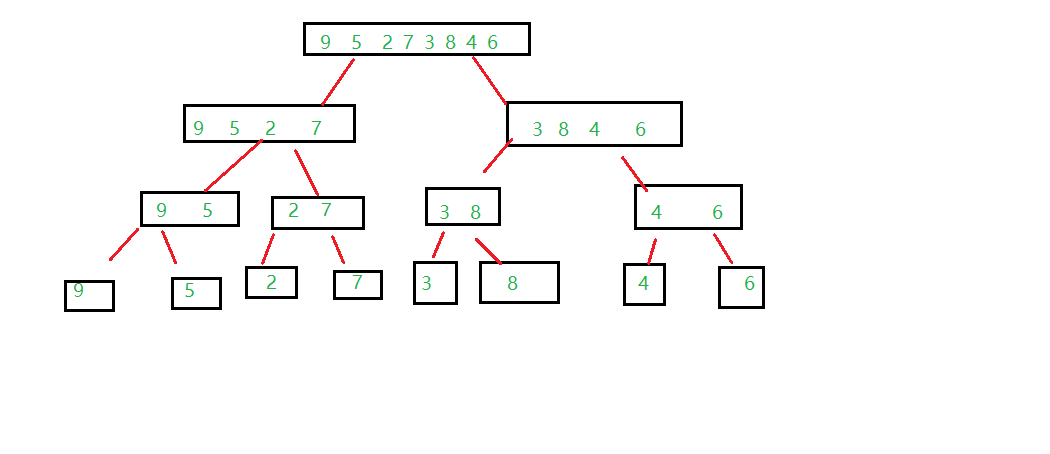
**归并排序的特性总结：**

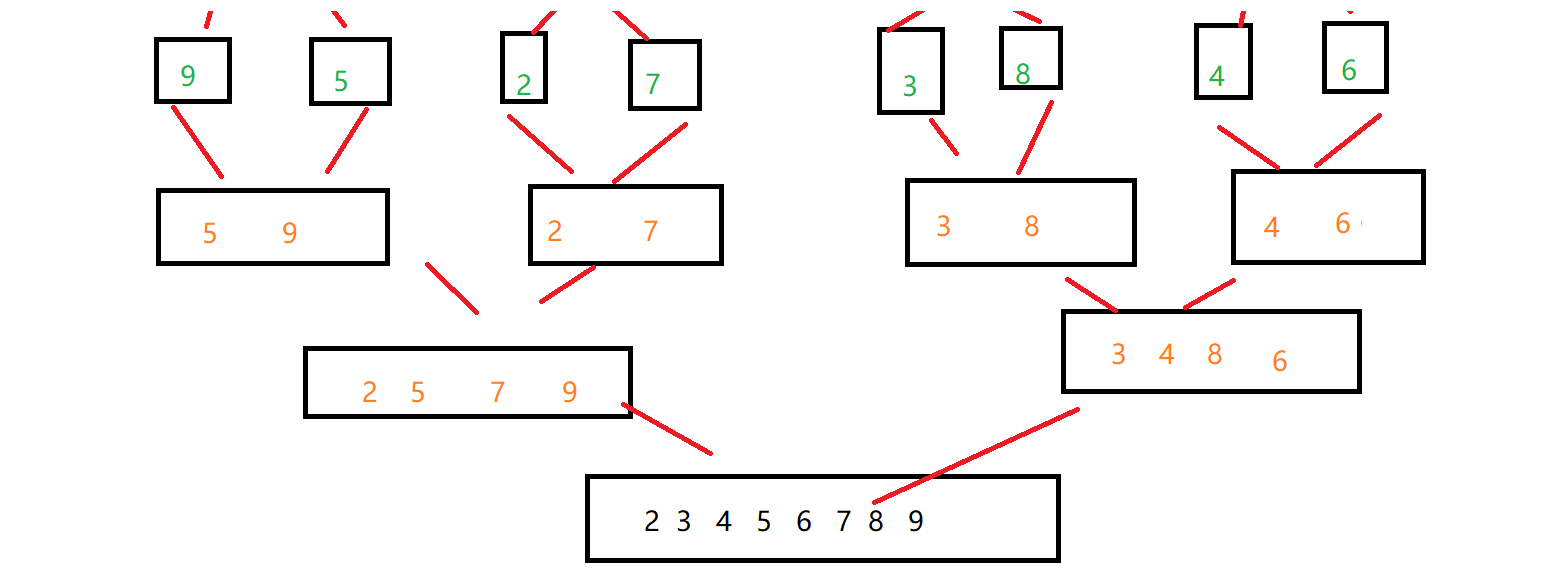
1. 归并的缺点在于需要O(N)的空间复杂度，归并排序的思考更多的是解决在磁盘中的外排序问题。

2. 时间复杂度：O(N\*logN)

3. 空间复杂度：O(N)

4. 稳定性：稳定





**public class TestMergeSort {**

**public static void merge(int[] array, int start, int mid, int end) {**

**//在另一个数组中归并。**

**int[] tempArray = new int[array.length];**

**int tempIndex = start;**

**int start2 = mid + 1;**

**//两个归并段都有数据**

**int i=start;**

**while (start <= mid && start2 <= end) {**

**if (array[start] <= array[start2]) {**

**tempArray[tempIndex++] = array[start++];**

**start = start + 1;**

**} else {**

**tempArray[tempIndex++] = array[start2++];**

**}**

**}**

**while (start <= mid) {**

**tempArray[tempIndex++] = array[start++];**

**}**

**while (start2<=end){**

**tempArray[tempIndex++] = array[start2++];**

**}**

**while (i<=end){**

**array[i]=tempArray[i];**

**i++;**

**}**

**}**

**public static void mergeSort(int[] array, int start, int end) {**

**if (start >= end) {//递归中止条件。**

**return;**

**}**

**int mid = (start + end) / 2;**

**mergeSort(array, start, mid);**

**mergeSort(array, mid + 1, end);//分解**

**//肯定是一个有序的序列。**

**//递归左，递归右**

**//合并的过程**

**merge(array, start, mid, end);**

**}**

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

**int[] array = {9, 5, 2, 7, 3, 95, 78, 165, 984, 1546, 1, 66, 55, 44, 88, 77, 88};**

**mergeSort(array,0,array.length-1);**

**for (int i = 0; i < array.length; i++) {**

**System.out.print(array[i] + " ");**

**}**

**}**

### 5.7排序算法的复杂度及稳定性

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **排序方法** | **平均情况** | **最好情况** | **最坏情况** | **辅助空间** | **稳定性** |
| **冒泡排序** | **O(n2)** | **O(n)** | **O(n2)** | **O(1)** | **稳定** |
| **简单选择排序** | **O(n2)** | **O(n2)** | **O(n2)** | **O(1)** | **稳定** |
| **直接插入排序** | **O(n2)** | **O(n)** | **O(n2)** | **O(1)** | **稳定** |
| **希尔排序** | **O(nlogn)-O(n)** | **O(n1.3)** | **O(n2)** | **O(1)** | **不稳定** |
| **堆排序** | **O(nlogn)** | **O(nlogn)** | **O(nlogn)** | **O(1)** | **不稳定** |
| **归并排序** | **O(nlogn)** | **O(nlogn)** | **O(nlogn)** | **O(n)** | **稳定** |
| **快速排序** | **O(nlogn)** | **O(nlogn)** | **O(n2)** | **O(logn)-O(n)** | **不稳定** |