# 1、三元运算：

1、语法格式：

布尔表达式？结果1：结果2；

2、如果表达式结果为true，返回结果1，如果结果为false，返回结果2

# 2、continue、break、return的区别：

* continue :只适用于循环中，用于跳过当前循环，进入下一次循环。一旦执行了continue，那么之后的语句就不再执行，直接开始下一次的循环。
* break：适用于 switch或循环语句中。用于结束当前循环语句或switch语句。多层循环语句中，如果配合标识符使用，可以结束外层循环语句。
* return：终止当前方法的运行，用于结束当前方法。也可以用来返回当前函数返回值类型的值。

# 3、switch语句

语法格式：

switch(变量):  
 case 变量值1：  
 break；  
 case 变量值2：  
 break；  
 ......  
 default:  
 break;

* 变量 可以是byte、short、int、char、enum类型,jdk1.7之后,也可以使用String类型。
* case 变量值 可以有多个，case之间使用break结束语句。jvm在扫描case变量值时，会从上到下进行扫描，直到扫描到合适的变量值时，会执行这个case下的代码，如果执行完当前case，没有遇到break，那么会继续执行下一个case语句，直到遇到break或执行完switch全部语句。如果没有扫描到合适的变量值，那么会执行default下的代码，如果没有default，那么结束switch代码块的执行。
* default：其他的，当switch中case都不满足变量要求时，会执行default语句。default语句可以不存在。

# 4、方法重载与方法重写

## 1、方法重载

1.1 什么是方法重载？

* 在同一个类中有多个（两个及以上）同名函数，被称作方法重载。

1.2 方法重载的要求

* 函数名一致
* 参数列表不同（参数个数不同，或者相同位置对应的参数类型不同）
* 与返回值类型无关，与权限修饰符无关

|  |
| --- |
| **public class** Dome1 {  // 方法重载：同类同函数名，形参列表不能相同，与返回值无关，与权限修饰符无关。  **private** **short** test() {  **short** aa = 4;  System.***out***.println(**"无参test函数，返回值类型short"**);  **return** aa;  }   **protected** String test(String s) {  System.***out***.println(**"String形参类型test函数,返回值类型String"**);  **return** s;  }   **default int** test(**int** count) {  System.***out***.println(**"int 形参类型test函数，返回值类型 int"**);  **return** count;  }   **public void** test(String s, **int** count) {  System.***out***.println(**"String + int 多形参类型test函数"**);  } } |

## 2、方法重写

2.1 什么是方法重写？

* 在继承中，子父类出现了同名的函数，叫做方法重写。

2.2 方法重写的前提

* 在继承关系中，父类的方法不能满足子类的需求，这时就需要使用方法重写。

2.3 方法重写的要求

* 子类和父类的函数名及形参列表必须一致
* 子类的权限修饰符必须要大于或等于父类的权限修饰符
* 子类的返回值类型必须要小于或等于父类的返回值类型
* 子类抛出的异常类型必须要小于或等于父类抛出的异常类型

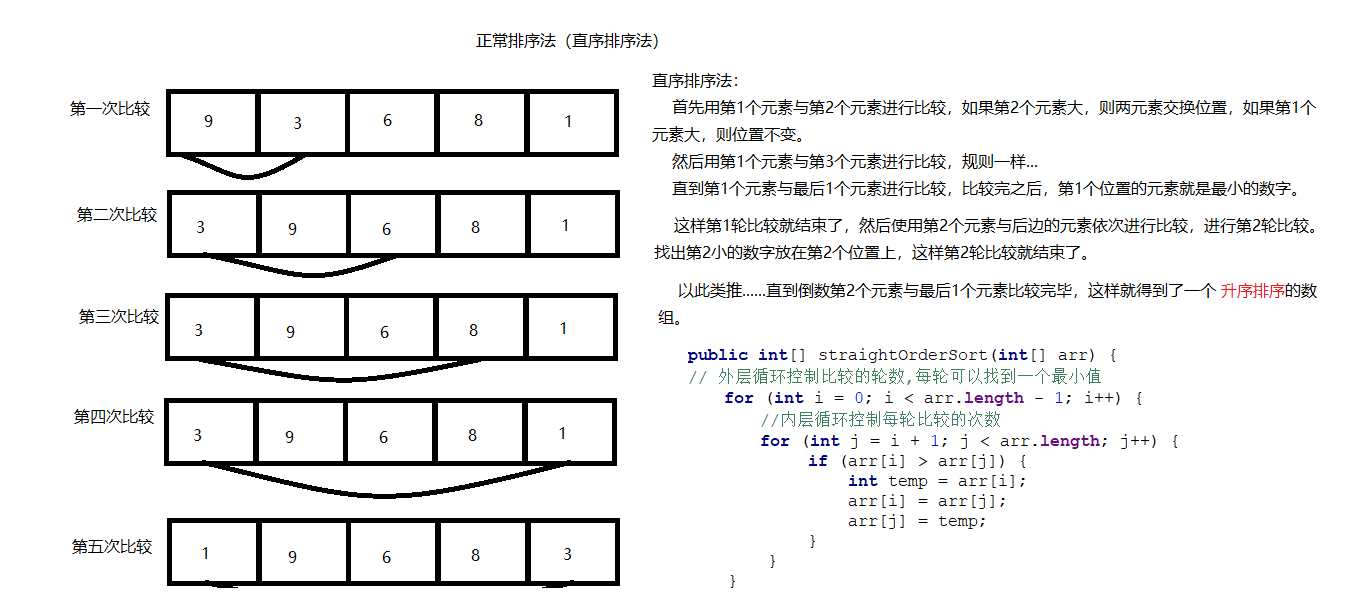
|  |
| --- |
| **class** Father {  // test 重载方法：同类同函数名，形参列表不能相同，与返回值无关。  **private static short** test() {  **short** aa = 4;  System.***out***.println(**"无参test函数，返回值类型short"**);  **return** aa;  }   **protected** String test(String s) {  System.***out***.println(**"String形参类型test函数,返回值类型String"**);  **return** s;  }   **public int** test(**int** count) {  System.***out***.println(**"int 形参类型test函数，返回值类型 int"**);  **return** count;  }   **public static void** test(String s, **int** count) {  System.***out***.println(**"String + int 多形参类型test函数"**);  } }  // 子类继承父类，当父类函数不满足子类的需求时，重写父类的方法 **class** Son **extends** Father {  // 子类的 函数名与形参列表 必须与父类一致。  // 权限修饰符 必须大于或等于父类权限修饰符。  // 返回值类型 必须大于或等于父类返回值类型。  // 抛出的异常类型 必须大于或等于父类抛出的异常类型。  @Override  **public** String test(String s) {  **return "父类权限修饰符是 protected，子类权限小于父类时报错"**;  }   @Override  **public int** test(**int** count) {  System.***out***.println(**"当返回值类型与父类返回值类型不同时，编译报错"**);  **return super**.test(count);  } } |

# 5、正常排序、冒泡排序、二分法排序

## 1、正常排序（直序排序法）

直序排序规则：

* 每个元素都与后边的所有元素比较一次，得到一个最大值放在最后1个位置上
* 然后进行第二轮的比较，每轮都比之前少比较一次，也就是上一轮得到的最大值不再参与比较。
* 当所有元素比较完后，就能得到一个升序排列的数组。



|  |
| --- |
| **public int**[] straightOrderSort(**int**[] arr) { // 外层循环控制比较的轮数,每轮可以找到一个最小值  **for** (**int** i = 0; i < arr.**length** - 1; i++) { //内层循环控制每轮比较的次数  **for** (**int** j = i + 1; j < arr.**length**; j++) {  **if** (arr[i] > arr[j]) { //小元素放前边，大元素放后边，得到升序排序后的数组  **int** temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;  }  }  }  **return** arr; // 返回升序排序后的数组 } |

## 2、冒泡排序法

冒泡排序规则：

* 将相邻两个元素进行比较，符合条件的进行位置交换，每轮找出一个最大值（最小值），且这个元素不再参与下一轮的比较，直到完成排序

|  |
| --- |
| // 冒泡排序法 **public int**[] bubbleSort(**int**[] arr) {  //外层循环控制比较的轮数，每次得到一个最大数字  **for** (**int** i = 0; i < arr.**length** - 1; i++) {  //内层循环控制的是每轮相邻元素比较的次数  **for** (**int** j = 0; j < arr.**length** - 1 - i; j++) {  **if** (arr[j] < arr[j + 1]) { // 大的放到前边  **int** temp = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  }  }  }  **return** arr; //返回排序后的数组 } |

## 折半查找法（二分查找法）

1、前提

* 数组是有序且没有重复元素的

2、查找原理

* 定义三个变量，分别表示数组索引的开始(start)、中间(mid)、结尾(end)。
* 首先使用被查找的值target与mid表示的元素进行比较，

如果相等，则是需要查找的值，直接返回mid。

如果mid表示的元素值小于target，则将start的指针 移到mid+1位置，并 重新计算mid索引值，然后进行下一次查找。

如果mid表示的元素值大于target，则将end的指针 移到mid-1位置，并重新计算mid索引值，然后进入下一次查找。

* 每次将mid表示的元素与被查找值target进行比较厚，都要重新计算mid的指针位置。
* 如果查到，返回索引值，如果查不到，返回-1。

|  |
| --- |
| // 折半查找法（二分查找法） 前提是被查找的数组必须是有序的，并且没有重复元素 **public int** compromisedSearch(**int**[] arr, **int** target) {  **int** min = 0;  **int** max = arr.**length** - 1;  **int** mid = (max + min) / 2;  **while** (**true**) {  **if** (target > arr[mid]) {  min = mid + 1;  } **else if** (target < arr[mid]) {  max = mid - 1;  } **else** {  **return** mid;  }  mid = (max + min) / 2;//重新计算中间值  **if** (max < min) {  **return** -1;//找不到  }  } } |

# 6、面向对象

## 1、封装

* 封装能够隐藏具体的实现，只对外提供一个接口供调用者使用。提高了数据的安全性
* 将 实体类的成员属性私有化，并根据需求提供相应的setter/getter方法。

## 2、构造代码块和静态代码块

2.1 构造代码块：用来初始化对象

* 当创建任意对象都需要进行同一个操作时（比如调用某个特定的方法），就可以使用构造代码块来初始化该对象
* 构造代码块的代码其实是在构造函数中执行的，但是它比构造函数中的代码先执行。
* 构造代码块的格式

{

构造代码块

}

2.2 静态代码块

* 静态代码块中的代码，是在其所属的类被加载到内存中时执行的。
* 静态代码块主要用于对一个项目进行初始化工作的。（比如从配置文件中读取数据库的用户名和密码）
* 静态代码块的格式

static{

静态代码块

}

## 3、instanceof

* 用于判断对象是否属于某种类别
  1. instanceof使用前提
* 用于比较的对象与类别，必须存在继承或实现关系。
  1. instanceof使用格式：
* 对象 instanceof 类别

3.3 应用场景

* 只有存在多态才可能使用到instanceof，一般用于数据类型强转前的判断。

## 4、匿名对象

* 匿名对象没有引用类型的变量指向它，如果一个对象只被使用一次就不再使用，那么就可以使用匿名对象。
* 匿名对象一般都是作为参数使用的。

## 5、抽象类abstract

5.1 抽象类的应用场景

* 在现实开发中，描述一类事务的时候，如果该类确实存在某种行为，但是却不能确定具体是什么的时候，我们可以使用抽象类

5.2 抽象类的好处

* 强制要求子类必须要实现指定的方法（抽象方法）
* 抽象函数没有方法体，所以必须子类必须要继承

## 6、继承

* 子类可以继承父类的所有属性和方法，但是对私有的属性和方法不具有直接使用权。父类的属性可以通过get/set方法来调用。这就是隐式继承。

## 7、线程：

多线程的存在的意义： 解决了一个进程允许多个任务可以同时执行。

多线程的创建方式：

### 7.1方式一: 继承Thread。

1. 自定义一个类继承Thread.

2. 重写Thread的run方法，把自定义线程的任务代码放在run方法上。

3. 创建Thread类的子类对象，并且调用start方法开启线程。

### 7.2 方式二： 实现Runnable接口。。

1. 自定义一个类实现Runnable接口.

2. 实现Runnable的run方法。把自定义线程的任务代码放在run方法上。

3. 创建Runnable实现类的对象。

4. 创建Thread的对象，然后把Runnable实现类的对象作为参数传递。

5. 调用Thread对象的start方法开启线程。

### 7.3 java中的同步机制：

7.3.1 出现线程安全问题的根本原因：

1. 存在两个或者两个以上的线程共享着资源。

2. 操作资源的代码块必须有语句。

7.3.2 同步代码块

同步代码块的格式：

synchronized(锁对象){

需要被同步的代码块...

}

7.3.3 同步代码块要注意的细节：

1.锁对象可以是任意的对象。

2. 锁对象必须 是多线程共享的资源。否则锁不住。

3. 没有线程安全问题的时候不要使用锁，因为会导致效率降低。

4. 调用sleep方法并不会释放锁对象，但是调用wait方法的线程就会释放锁对象。

7.4 同步函数

修饰符 synchronized 返回值类型 函数名(形参列表..){

}

7.4.1注意：

1. 同步函数的锁对象是不能任意的，非静态同步函数的锁对象是this对象，静态函数的锁对象是当前字节码对象。

2. 同步函数的锁不能由你指定，是固定的。