## Java基础面试题

### JAVA三大开发平台分别是什么?

JavaSE标准版: 做电脑上运行的软件。

JavaEE企业版: 用来做网站的。

JavaME微型版: 做手机软件的。

### 简述JAVA语言的特点?

**简单性:** Java是由C++简化设计而来，剔除了C++的操作符重载（overload）和多继承等特征，Java的垃圾回收机制，使用户不必为存储管理问题烦恼，能更多的时间和精力花在研发上。

**面向对象**: Java是一个完全面向对象的语言。对程序员来说，这意味着要注意应用中的数据和操纵数据的方法（method），而不是严格地用过程来思考。

**编译和解释性**: Java编译程序生成字节码（byte-code），而不是通常的机器码。Java字节码提供对体系结构中性的目标文件格式，代码设计成可有效地传送程序到多个平台。Java程序可以在任何实现了Java解释程序和运行系统（run-time system）的系统上运行。

**安全性:** Java没有指针，所以程序员不能得到隐蔽起来的内幕和伪造指针去指向存储器。编译的Java代码中的存储引用在运行时由Java解释程序决定实际存储地址。

**可移植性(跨平台性/结构中立)**: 因为JVM的存在,可以实现一次编译,多处运行的机制。

**多线程性**: Java是多线程语言，它提供支持多线程的执行，能处理不同任务，使具有线程的程序设计很容易。

### 面向对象和面向过程的区别?

二者都是一种思想，面向对象是相对于面向过程而言的。

面向过程，强调的是功能行为，以函数为最小单位，考虑怎么做。

面向对象，将功能封装进对象，强调具备了功能的对象，以类/对象为最小单位，考虑谁来做。

### 什么是JDK、JRE、JVM以及它们的关系（\*\*\*）?

**JDK: Java Development Kit(JAVA软件开发包)**，JDK是整个Java开发的核心，它包括了JRE（Java运行环境）、一堆Java工具（javac/java/jdb等）和Java基础的类库。

**JRE: Java Runtime Environment(Java运行时环境)**，运行JAVA程序所必须的环境的集合，包含JVM（Java虚拟机）及Java核心类库。

**JVM: Java Virtual Machine(Java虚拟机)**，它是整个Java实现跨平台的最核心的部分，所有的Java程序会首先被编译为.class的类文件，这种类文件可以在虚拟机上执行。

JDK = JRE + 开发工具集；

JRE = JVM + Java标准类库；

### 一个".java"源文件中是否可以包括多个类（不是内部类）?有什么限制?

一个Java源文件可以包含多个类，但是只能有一个public修饰的类；

### Java的两大核心机制是什么（\*\*\*）?

**Java虚拟机（Java Virtual Machine）**：Java虚拟机可以理解成一个以字节码为机器指令的CPU，对于不同的运行平台，有不同的虚拟机, Java虚拟机机制屏蔽了底层运行平台的差别，能够很好的实现跨平台。

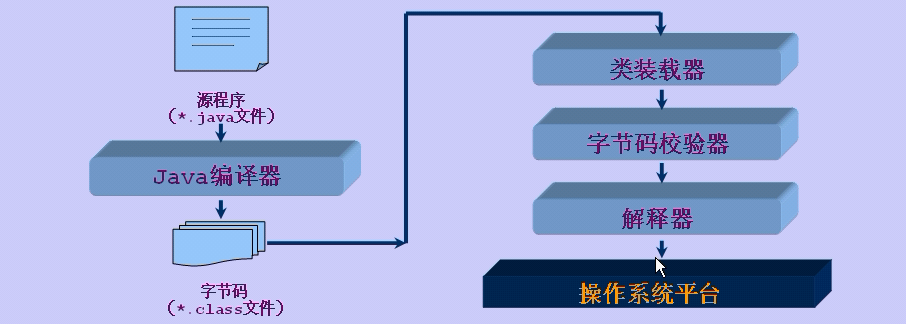
**垃圾收集机制（Garbage Collection）**： Java的垃圾回收机制是Java虚拟机提供的能力，用于在空闲时间以不定时的方式动态回收无任何引用的对象占据的内存空间。

### JVM的三项主要功能是什么（\*）?

加载代码: 由类加载器完成；

校验代码: 由字节码校验器完成；

执行代码: 由运行时解释器完成；



### 什么是垃圾回收(GC)（\*）?

垃圾回收是一种动态存储管理技术，它自动地释放不再被程序引用的对象，按照特定的垃圾收集算法来实现资源自动回收的功能。当一个对象不再被引用的时候，内存回收它占领的空间，以便空间被后来的新对象使用，以免造成内存泄露。

### 对象无用了,就会立马被回收吗（\*\*\*）?

垃圾回收器如发现一个对象不能被任何活线程访问时，他将认为该对象符合删除条件，就将其加入回收队列，但不是立即销毁对象，**何时销毁并释放内存是无法预知的**。

其他问法：System.gc()能立即回收无用内存吗/程序员可以精准干预垃圾回收吗？

### goto是不是Java关键字（\*）?

goto不是关键字,是Java的保留字;

注: const和goto是java的保留字。

### 列举常用关键字（\*）?

关键字的特征: 都是小写字母

用来定义数据类型的关键字:

class 用来定义类 interface 用来定义接口

enum 用来定义枚举 byte 字节类型

short 短整型 int 整数类型

long 长整型 float 单精度浮点类型

double 双精度浮点类型 char 字符类型

boolean 布尔类型 void 空类型

注: 类和接口都是数据类型,它是引用数据类型

用来定义数据类型值的关键字:

true 真值 false 假值 null 空值

用来流程控制的关键字:

if 定义单分支语句 if else 定义双分支

if else if 定义多分支 switch 定义开关语句

case 定义开关语句的每一个分支

default 定义开关语句默认选项

while 用来定义while循环

do while 用来定义do whie循环

for 用来定义for循环

break 打断,破坏,用来结束整个循环体或打断switch的贯穿性

continue 用来跳过本次循环,执行后续循环

return 定义方法返回值

用于定义访问控制修饰符的关键字:

private 私有的 protected 受保护的 public 公共的

用来定义类、函数、变量修饰符的关键字:

abstract 用来定义抽象类或抽象方法

final 用来定义常量、常量方法、常量类;

static 静态的

synchronized 定义同步的

用来定义类和类关系的关键字:

extends 定义继承关系

implements 定义实现关系

实例相关关键字:

new 实例化对象

this 当前对象的指针

super 父对象的指针

instanceof 判断对象是否属于某个类的实例

用于异常的关键字:

try 用于捕获异常

catch 用于处理异常

finally 无论是否有异常,都执行

throw 用来抛出异常

throws 用来声明异常

用于包的关键字:

package 定义包

import 导包

### 请简述标识符的概念及命名规范?

Java 对各种变量、方法和类、接口等要素命名时使用的字符序列称为标识符（在Java里面,所有可以自己起名字的地方都叫标识符）。

（1）标识符由字母、数字、下划线“\_”、美元符“$”组成

（2）标识符不能以数值开头。

（3）不可以是Java的关键字/保留字。

（4）Java 标识符大小写敏感，且长度无限制。

（5）见名之意。

（6）驼峰原则：首字母小写,多个单词组成的话,首个单词的首字母小写，之后每个单词的首字母都大写。

类名/接口首字母大写，如果包含多个单词，每个单词首字母都大写。

变量/方法首字母小写，如果包含多个单词，之后的每个单词首字母都大写。

（7）常量全部大写，多个单词组成,则每个单词由\_拼接。

（8）包名全部小写。我们在项目中，包名一律使用这种方式。

[cn.]com.公司名.项目名.模块名

### 变量的分类及区别（局部变量、成员变量、静态变量的区别）（\*\*\*）?

局部变量定义在方法内部或语句块内部; 成员变量定义在方法外部,类内部; 静态变量定义在方法外部,类内部,由static修饰;

局部变量没有初始值,必须先赋值才能使用; 成员变量和静态变量都由默认的初始值;

局部变量通过变量名调用; 成员变量通过对象调用; 静态变量既可以通过对象调用,也可以通过类名调用;

局部变量在栈内存; 成员变量在堆内存; 静态变量在方法区;

局部变量的生命周期: 与方法共存亡; 成员变量的生命周期: 与对象共存亡; 静态变量的生命周期: 与类共存亡;

局部变量不能被访问控制修饰符及 static 所修饰; 成员变量/静态变量可以被 public,private,static 等修饰符所修饰。成员变量/静态变量和局部变量都能被 final 所修饰；

### Java数据类型分类（\*）?

基本数据类型

整数类型: byte、short、int、long

浮点类型: float、double

布尔类型: boolean

字符类型: char

注: 整形数据默认为int数据类型，浮点型默认为double数据类型；如果要表示long型数据或float型数据，要在相应的数值后面加上l、L或f、F，否则会出现编译问题。

引用数据类型：类、接口、数组等

### 八种基本数据类型的取值范围（\*）?

整数类型的取值范围算法: -2位数-1 ~ 2位数-1 - 1

1byte = 8bit

数据类型 占几个字节 取值范围

byte 1 -27 ~ 27 -1 -128 ~ 127

short 2 -215 ~ 215 -1 -32,768 ~ 32767

int 4 -231 ~ 231 -1

long 8 -263 ~ 263 -1

float 4

double 8

char 2 0 ~ 65535

boolean 1 只有两个值,true和false

### Java的基本数据类型有哪些? String是不是基本数据类型（\*\*\*）?

八种基本数据类型:byte、short、int、long、float、double、boolean、char

String不是基本数据类型,它是引用类型

### char类型的变量能存放汉字吗（\*）?

可以，因为java采用Unicode编码。

### Java数据类型转换（\*）?

自动类型转换: 容量小的类型自动转换为容量大的数据类型。它们的顺序是: byte、short(char)、int、long、float、double

强制类型转换: 将容量大的数据类型转换为容量小的数据类型。使用时要加上强制转换符（()），但可能造成精度降低或溢出,格外要注意。

byte、short、char类型参与运算，直接转换为int类型参与运算。

大类型和小类型进行运算，得到的肯定是大类型。

### 计算机中常用的进制?

进制 范围 前缀

二进制 0-1 0b

八进制 0-7 0

十进制 0-9

十六进制 0-9, A-F 0x

### 进制转换规则?

二进制、八进制、十进制、十六进制

基数: X进制的基数是X

系数: 每一位上面的数值

权: 每一位上基数的幂(乘方)

注: 任何数值的0次幂都是1

**二进制和十进制相互转换:**

十进制转二进制: 十进制数除2取余法，即十进制数除2，余数为权位上的数，得到的商值继续除2，依此步骤继续向下运算直到商为0为止。从最后一个余数为读到第一个;

二进制转十进制: 二进制转换成十进制，需要二进制的每一位的数值乘以相应的2的幂次方，然后相加。

**八进制、十六进制和十进制相互转换:**

十进制转八进制和十六进制: 把十进制转八进制或者十六进制按照除8或者16取余，直到商为0为止。

八进制和十六进制转十进制: 把八进制、十六进制数按权展开、相加即得十进制数。

**二进制与八进制之间的转换:**

间接转换法: 把二进制转换为十进制,在转换为八进制; 把八进制转换为十进制,在转换为二进制;

二进制转八进制: 3位二进制数按权展开相加得到1位八进制数。（注意事项，3位二进制转成八进制是从右到左开始转换，不足时补0）。

八进制转成二进制: 八进制数通过除2取余法，得到二进制数，对每个八进制为3个二进制，不足时在最左边补零。

**二进制与十六进制之间的转换:**

二进制转十六进制: 与二进制转八进制方法近似，八进制是取三合一，十六进制是取四合一。（注意事项，4位二进制转成十六进制是从右到左开始转换，不足时补0）。

十六进制转二进制: 十六进制数通过除2取余法，得到二进制数，对每个十六进制为4个二进制，不足时在最左边补零。

**十六进制与八进制之间的转换:**

第一种：他们之间的转换可以先转成二进制然后再相互转换。

第二种：他们之间的转换可以先转成十进制然后再相互转换。

### 什么是原码、反码、补码（\*）?

正数的原码、反码、补码都是二进制表现形式;

负数的原码是负数的绝对值转换成二进制位然后在高位补1;

负数的反码是在其原码的基础上,符号位不变，其余各个位取反;

负数的补码是在反码的基础上补1;

数值在计算机中是以**补码形式**存在的;

### 为何要使用原码, 反码和补码（\*）?

为了让计算机运算的设计更简单了;

### Java运算符的分类有哪些?

算术运算符 + - \* / % ++ --

比较/关系运算符 > >= < <= != ==

逻辑运算符 && || !

赋值运算符 = += -+ \*= %= /=

三目运算符 条件表达式?值1:值2; 如果表达式true,则得到值1,否则值2

位运算符 & | ^ ~ >> << >>>

instanceof运算符

### 除法运算的陷阱有哪些（\*）?

**两个整数值进行除法运算,得到的肯定是整数值,不会保留小数位;**

0和整数类型进行除法运算,不能作为除数; 0和浮点类型进行除法运算,可以作为除数;

当两个数的绝对值均为0.0时候，商等于NaN。

### 0可以不可以作为除数（\*\*\*）?

0和整数类型进行除法运算,不能作为除数,否则抛出异常;

0和浮点类型进行除法运算,可以作为除数,得到无穷大的值;

两个绝对值为0.0的数进行除法运算，得到的是NaN;

### 代码阅读题

int x = 10;

int z = x++ + 3; // 10 + 3 = 13 z = 13 x = 11

int k = ++x + z + 6; // 12 + 13 + 6 = 31 x = 12

// z、k、x的值分别是多少?

System.out.println(z); // 13

System.out.println(k); // 31

System.out.println(x); // 12

解题思路: 参照下题;

### ++/--在前和在后的区别（\*）?

++ 递增运算,在原有值的基础上+1

-- 递减运算,在原有值的基础上-1

如果++/--是一行单独的运算,那么在前和在后没有区别;

如果++/--参与运算或调用,那么在前是先递增/递减运算再调用值,在后是先调用值再递增/递减运算。

### 2 << 2的结果?

8

### 用最有效的方式计算2乘以8（\*\*\*）?

2 << 3;

### & 和 && 的区别（\*\*\*）?

& 按位与运算符,&& 逻辑与运算符;两者都可以进行逻辑与运算;

& 不具备短路功能;&&具备短路功能,一旦表达式出现了false,则之后的表达式都不再运行;

### | 和 || 的区别（\*）?

| 按位或运算符,|| 逻辑或运算符;两者都可以进行逻辑或运算;

| 不具备短路功能,||具备短路功能,一旦表达式出现了true,则之后的表达式都不再运行;

### short s = 10; s = s + 1的结果是什么? s += 1结果是什么,为什么（\*）?

s = s + 1的结果是报错,short在进行运算的时候,转换成int类型运算,int类型给short类型赋值,需要强转;

s = s + 1写法应该改为 s = (short)(s + 1);

s += 1正常运行,s = 11;因为+=不会改变数据类型;

### 赋值运算符代码阅读题（\*）?

|  |
| --- |
| // 思考1  int i = 1;  i \*= 0.1;  System.out.println(i);//0  i++;  System.out.println(i);//1  // 思考2  int m = 2;  int n = 3;  n \*= m++; //n = n \* m++;  System.out.println("m=" + m);//3  System.out.println("n=" + n);//6  // 思考3  int n1 = 10;  n1 += (n1++) + (++n1);//n1 = n1 + (n1++) + (++n1);  System.out.println(n1);//32 |
| public class Test01 {  public static void main(String args[]) {  int x = 1;  int y = 1;  if (x++ == 2 & ++y == 2) {  x = 7;  }  System.out.println("x=" + x + ",y=" + y); // x=2,y=2  }  }  public class Test02 {  public static void main(String args[]) {  int x = 1, y = 1;  if (x++ == 2 && ++y == 2) {  x = 7;  }  System.out.println("x=" + x + ",y=" + y); // x=2,y=1  }  }  public class Test03 {  public static void main(String args[]) {  int x = 1, y = 1;  if (x++ == 1 | ++y == 1) {  x = 7;  }  System.out.println("x=" + x + ",y=" + y);//x=7,y=2  int x = 1, y = 1;  if (x++ == 1 || ++y == 1) {  x = 7;  }  System.out.println("x=" + x + ",y=" + y);//x=7,y=1  }  }  public class Test04 {  public static void main(String args[]) {  boolean x = true;  boolean y = false;  short z = 42;  //if(y == true)  if ((z++ == 42) && (y = true)) z++;  if ((x = false) || (++z == 45)) z++;  System.out.println("z =" + z);//z =46  }  } |

### instanceof 运算符的作用?

instanceof 是 Java 的保留关键字。

作用是：测试它左边的对象是否是它右边的类的实例，返回 boolean 的数据类型。 instanceof 是Java 中的二元运算符，左边是对象，右边是类；当对象是右边类或子类所创建对象时，返回true；否则，返回false。

### Java的switch语句可以定义什么类型的值（\*\*\*）?

byte,short,char,int,String,枚举enum其中的一种。

注:

1. 枚举是jdk1.5之后支持的,String是jdk1.7之后才支持的;
2. 因为有自动拆箱，switch也可作用于byte、short、char、int 的包装类上。

### switch语句和多重if语句的区别（\*）?

相同点：都是用来处理多分支条件的结构;

不同点

switch选择结构:只能处理等值条件判断的情况，而且条件必须是整型变量或字符串型变量;

多重if选择结构:不仅可以做等值比较也可以做区间值比较，特别适合某个变量处于某个连续区间时的情况;而且条件没有类型限制;

if 在应用中更为常见,有时switch 结构更清晰.

### 如下代码的执行结果（\*）?

0.3 \* 1 == 0.3 true

0.1 + 0.2 == 0.3 false

0.1 \* 3 == 0.3 false

### 为什么0.1 + 0.2 = 0.30000000000000004（\*）?

说法一:

十进制数在进行运算的时候,会转换为二进制进行运算,运算完成之后在转换为十进制;在小数运算转换过程中,可以出现误差;

说法二:

和浮点类型的底层存储有关系;

### 结束循环的两种方式（\*）?

1、循环条件不成立,结束循环

2、在循环体内部,满足某种条件,使用break强制结束循环

### while 和 do while的区别（\*\*\*）?

两者都是用来实现循环结构的;

while是先判断后执行; do while是先执行一次后判断;

如果初始条件成立,那么while和do while的执行结果一致;

如果初始条件不成立,那么while什么都不执行,do while至少执行一次;

### for循环 和 while循环的区别（\*）?

两者都是用来实现循环结构的;

语法定义不同:

while循环,循环三要素定义在不同的地方;初始部分定义在循环外,条件定义在while后面的(),迭代定义在循环体;

for循环,循环三要素都定义在for循环后面的(),使用;分割

使用场景不同:

while 用于不确定循环次数的场合;

for 用于确定循环次数的场合;

初始变量的作用域不同:

while的初始变量在循环外都可以使用;

for循环的初始变量只能在for的循环体中;

### for循环、while循环、do while循环的使用场景（\*\*\*）?

while 用于不确定循环次数的场合;

do while 用于至少执行一次的场合;

for 用于确定循环次数的场合;

### break和continue的区别（\*\*\*）?

使用场合不同：

break可用于switch结构和循环结构中；

continue只能用于循环结构中；

return 只能用于方法中；

作用不同

break语句终止某个循环，程序跳转到循环块外的下一条语句。

continue跳出本次循环，进入下一次循环

return 用来定义方法返回值或结束方法；

### 如果有循环多层嵌套,怎么结束指定循环（\*）?

break写在哪个循环里面,就是结束哪个循环;

**如果想要结束指定循环,我们可以给循环打标记,使用break 标记名; 结束指定循环**

f1:for(int i = 1;i <= 10;i++){

for(int j = 1;j <= 10;j++){

if(i + j == 15){

// break; // break是结束内层循环

break f1; // 结束指定循环

}

System.out.println(i+" "+j);

}

}

### 下面程序的输出结果?

|  |
| --- |
| public class OperatorTest {  public static void main(String[] args) {  boolean x = true;  boolean y = false;  short z = 40;  if ((z++ == 40) && (y = true)) {  z++;  }  if ((x = false) || (++z == 43)) {  z++;  }  System.out.println("z = " + z);  }  } |

运行结果：z = 44

### 输入两个正整数m和n，求其最大公约数和最小公倍数。

|  |
| --- |
| */\*输入两个正整数m和n，求其最大公约数和最小公倍数。 比如：12和20的最大公约数是4，最小公倍数是60。 说明：break关键字的使用：一旦在循环中执行到break，就跳出循环\*/* Scanner scanner = new Scanner(System.*in*); System.*out*.print("请输入第一个正整数:"); int m = scanner.nextInt(); System.*out*.print("请输入第二个正整数:"); int n = scanner.nextInt();  *// 求其最大公约数,最大公约数一定不会超过这两个数的最小值 // 最大公约数的查询范围: 1 - 12,无论是正向查找还是反向查找都可以,但是为了节省时间,考虑反向查找* int maxCommonDivisor = -1; int min = m <= n ? m : n; for (int i = min; i >= 1; i--) {  if (m % i == 0 && n % i == 0) {  maxCommonDivisor = i;  break; *// 如果第一次走进来了,那么就找到了最大公约数,结束循环* } }  *// 求其最小公倍数* int max = m >= n ? m : n; int minCommonMultiple = -1; for (int i = max; i <= m \* n; i++) {  if (i % m == 0 && i % n == 0) {  minCommonMultiple = i;  break;  } }  *// Variable 'maxCommonDivisor' might not have been initialized* System.*out*.println(m + "和" + n + "的最大公约数是:" + maxCommonDivisor + ",最小公倍数:" + minCommonMultiple); |

### 判断101-200之间有多少个素数，并输出所有素数。(只能被自己整除)

|  |
| --- |
| */\*  \* 判断101-200之间有多少个素数，并输出所有素数。(只能被自己整除)  \* 质数又称素数。一个大于1的自然数，除了1和它自身外，不能被其他自然数整除的数叫做质数；否则称为合数。  \* 判断素数的方法：用一个数分别去除2到 Math.sqrt(这个数)，如果能被整除，则表明此数不是素数，反之是素数。  \* Math.sqrt(数值) 获取这个数值的平方根整数  \* \*/* int counter = 0; for (int i = 101; i <= 200; i++) {  *// 循环变量i,就是我们需要进行判断处理的每个数值  // i到底是不是素数,需要内层循环都执行完毕才可以得出结论* boolean isFlag = true; *// 先假定每个值(i)都是素数* for (int j = 2; j <= Math.*sqrt*(i); j++) {  if (i % j == 0) {  isFlag = false; *// 如果可以被2 - 其平方根整除,那么假定不成立* break;  }  }   if (isFlag) {  counter++;  System.*out*.println(i);  } } System.*out*.println("素数个数是:" + counter); |

### 打印出所有的"水仙花数"，所谓"水仙花数"是指一个三位数，其各位数字立方和等于该数本身。

|  |
| --- |
| */\*  \* 打印出所有的"水仙花数"，所谓"水仙花数"是指一个三位数，其各位数字立方和等于该数本身。  \* \*/* for (int i = 100; i <= 999; i++) {  *// 153  // 百位* int hundreds = i / 100;  *// 十位* int decade = i / 10 % 10;  *// 各位* int unit = i % 10;   boolean isFlag = (hundreds \* hundreds \* hundreds + decade \* decade \* decade + unit \* unit \* unit == i) ? true : false;   if(isFlag){  System.*out*.println(i);  } } |

### 输入一个正整数,将其分解质因数。例如：输入90,打印出`90=2\*3\*3\*5`。

|  |
| --- |
| */\*输入一个正整数,将其分解质因数。例如：输入90,打印出90=2\*3\*3\*5。 质因数:就是一个数的约数，并且是质数。 \*/* Scanner scanner = new Scanner(System.*in*); System.*out*.print("请输入一个整数:"); *// 1、输入一个数整数n；* int n = scanner.nextInt();  System.*out*.print(n + " = ");  *// 2、用for循环从k=2开始一直遍历到k小于等于n；* for (int k = 2; k <= n; k++) {  *// 如果n%k为0,那么表示接下来拿到商,继续从2开始分解* if(n % k == 0 && n != k){  System.*out*.print(k + " \* ");  n /= k;  k = 1; *// 重置k,表示又从2开始除* }else if(k == n){  System.*out*.print(k);  } } |

### 分析如下代码结果?

|  |
| --- |
| int k = 10; *// 初始值* do {  k += 1; *// i++; 递增* if (k > 20) {  break;  } } while (k > 0); System.*out*.println(k); *// 21* int j = 1; do {  System.*out*.println(++j); *// 2 3 4 5* } while (j < 5); |

### 为什么要使用数组?

当我们定义一批有关联的数据的时候,Java变量的局限性导致无论存还是取都比较麻烦;

1、非常浪费内存

2、代码复用性低

3、变量之间的关联是松散的,不便于集中管理

### 什么是数组以及Java数组的特点（\*）?

数组(Array)，是多个相同类型数据按一定顺序排列的集合，并使用一个名字命名，并通过编号的方式对这些数据进行统一管理。

1.数组是有序排列的集合。

2.数组本身是引用数据类型，而数组中的元素可以是任何数据类型，包括基本数据类型和引用数据类型。

3.数组的元素的类型，必须一样。

4.创建数组对象会在内存中开辟一整块连续的空间，而数组名中引用的是这块连续空间的首地址。

5.数组的长度是固定的，长度一旦确定，就不能修改。

6.我们可以直接通过下标(或索引)的方式调用指定位置的元素，速度很快。

### 数组在内存中如何分配（\*\*\*）?

当一个对象使用 new 关键字创建的时候，会在堆上分配内存空间，然后才返回到对象的引用。这对数组来说也是一样的，因为数组也是一个对象。

### 数组元素/成员变量的默认值（\*）?

| **数组元素类型** | **元素默认初始值** |
| --- | --- |
| byte | 0 |
| short | 0 |
| int | 0 |
| long | 0L |
| float | 0.0F |
| double | 0.0 |
| char | 0 或写为:’\u0000’(表现为空) |
| boolean | false |
| 引用类型 | null |

### 数组有没有length()这个方法? String有没有length()这个方法（\*）？

数组没有length()这个方法，有length的属性。

String有length()这个方法。

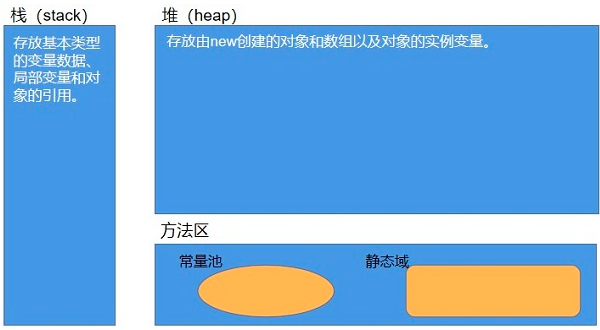
### Java中的任何数据类型都可以使用System.out.pritln方法显示吗（\*）?

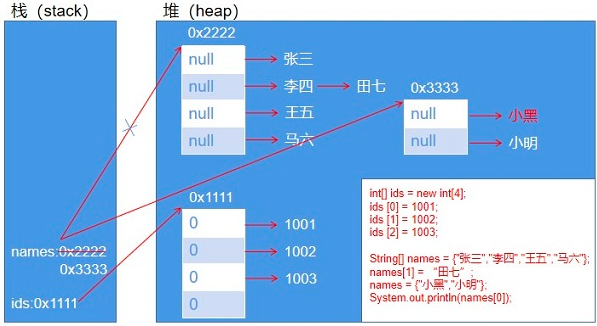
对基本数据类型而言，输出的往往是变量的值;

对于像数组这一类复杂的数据类型，输出的是其堆空间中存储位置的hashCode值;

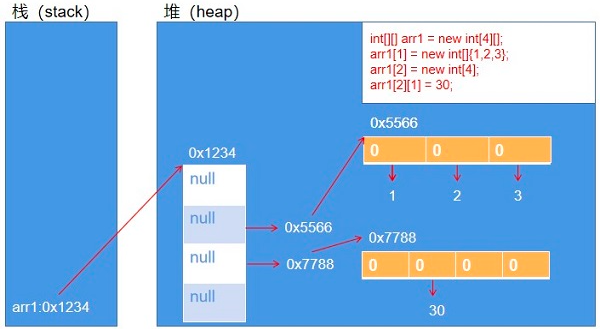
### 一维数组和二维数组内存分析（\*\*\*）?

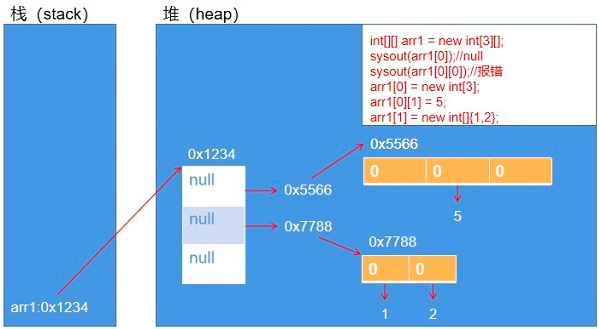
**一维数组图示：**





**二维数组图示：**





### 通过代码实现数组反转?

|  |
| --- |
| String arr[] = new String[]{"AA", "BB", "CC", "DD", "EE", "FF", "GG"};  System.*out*.println("反转之前:"); for (String str : arr) {  System.*out*.print(str + "\t"); } System.*out*.println("\n-------------------------------------------");  */\*for(int i = 0;i < arr.length / 2;i++){  // 定义临时变量  String temp = arr[i]; // 临时变量保存的是哪个,就先给哪个赋值  arr[i] = arr[arr.length - 1 - i];  arr[arr.length - 1 - i] = temp; }\*/* for(int i = 0,j = arr.length - 1;i < j;i++,j--){ *// i = 3 j = 3  // 定义临时变量* String temp = arr[i]; *// 临时变量保存的是哪个,就先给哪个赋值* arr[i] = arr[j];  arr[j] = temp; }  System.*out*.println("反转之后:"); for (String str : arr) {  System.*out*.print(str + "\t"); } |

### 使用冒泡排序实现数组元素排序（\*\*\*）?

原理：冒泡排序（Bubble Sort）也是一种简单直观的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。 这个算法的名字由来是因为越小的元素会经由交换慢慢"浮"到数列的顶端。

|  |
| --- |
| public class BubbleSort {  public static void main(String[] args) {  */\*  \* 它重复地走访过要排序的元素列，依次比较两个相邻的元素，  \* 如果顺序（如从大到小、首字母从Z到A）错误就把他们交换过来。  \* 走访元素的工作是重复地进行直到没有相邻元素需要交换，也就是说该元素列已经排序完成。  \* \*/* int[] arr = {28, 76, 154, 9, 2, 0, 200};  System.*out*.println("冒泡排序之前:");  for (int n : arr) {  System.*out*.print(n + "\t");  }   *// 从小到大排列  /\*for (int i = 0; i < arr.length - 1; i++) {  for (int j = 0; j < arr.length - 1 - i; j++) {  if (arr[j] > arr[j + 1]) {  int temp = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  }  }  }\*/   // 从大到小* for (int i = 0; i < arr.length - 1; i++) {  for (int j = 0; j < arr.length - 1 - i; j++) {  if (arr[j] < arr[j + 1]) {  int temp = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  }  }  }   System.*out*.println("\n--------------------------------");  System.*out*.println("冒泡排序之后:");  for (int n : arr) {  System.*out*.print(n + "\t");  }   } } |

### 使用选择排序实现数组排序（\*\*\*）?

|  |
| --- |
| public class SelectionSort {  public static void main(String[] args) {  */\*  \* 选择排序：  \* 第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，  \* 然后再从剩余的未排序元素中寻找到最小（大）元素，然后放到已排序的序列的末尾。  \* 以此类推，直到全部待排序的数据元素的个数为零。  \*   \* \*/* int arr[] = {120, 89, 32, 18, 10};  System.*out*.println("选择排序之前:");  for (int n : arr) {  System.*out*.print(n + "\t");  }   for (int i = 0; i < arr.length - 1; i++) { *// 0 1 2 3* for (int j = i; j < arr.length - 1; j++) {  if (arr[i] > arr[j + 1]) {  int temp = arr[i];  arr[i] = arr[j + 1];  arr[j + 1] = temp;  }  }  }   System.*out*.println("\n--------------------------------");  System.*out*.println("选择排序之后:");  for (int n : arr) {  System.*out*.print(n + "\t");  }  } } |

### 二分查找的原理和代码实现（\*\*\*）?

|  |
| --- |
| public class ArrayBinarySearch {  public static void main(String[] args) {  */\*  \* 二分查找的前提：数据必须是有序的，如果是一个无序的数组,那么必须先排序，后查找；  \* 二分查找的基本思想是：  \* 在有序表中，取中间元素作为比较对象，若给定值与中间元素相等，则查找成功；  \* 若给定值小于中间元素，则在中间元素的左半区继续查找；  \* 若给定值大于中间元素，则在中间元素的右半区继续查找。  \* 不断重复上述过程，直到找到为止。  \* \*/  // 有序的数组* int[] arr = new int[]{1, 4, 7, 10, 13, 15, 21, 25};   *// 要查找的元素* int findNum = 12;  findNum = 21;   *// 定义变量,来存储找到的元素的下标* int index = -1;   *// 初始化指针* int min = 0;  int max = arr.length - 1;   *// 循环操作* while (min <= max) { *// 最小指针 小于等于 最大指针,一致循环  // 得到中间指针* int mid = (max + min) / 2;   *// 判定找到* if (arr[mid] == findNum) {  index = mid;  break;  } else if (findNum > arr[mid]) {  min = mid + 1;  } else {  max = mid - 1;  }  }   System.out.println(findNum + "在arr数组的下标是:" + index);  } } |

### 谈谈你对面向对象中类和对象的理解，并指出二者的关系?

类是对一类事物的描述，是抽象的、概念上的定义。

对象是实际存在的该类事物的每个个体，因而也称为实例(instance)。

类从一系列相同对象中抽取出来相同特征的抽象的概念。对象是类的具体化，即是该类事物实实在在存在的个体。

### 类的组成部分(类的成员)（\*\*\*）?

成员变量/属性/实例变量/域/字段

成员方法/方法/实例方法/函数

构造方法/构造器/构造函数

代码块

内部类

### 方法存在的意义?

1. 提高代码的复用性。

2. 增强代码的可维护性。

### 方法组成的三要素（\*）?

返回值类型、方法名、参数列表。

### 类的方法内是否可以定义变量？是否可以调用属性？是否可以定义方法？是否可以调用方法？

是 是 否 是

### 如何理解万事万物皆对象?

1.在Java语言范畴中，我们都将功能、结构等封装到类中，通过类的实例化，来调用具体的功能结构

>Scanner,String等

>文件：File

>网络资源：URL

2.涉及到Java语言与前端Html、后端的数据库交互时，前后端的结构在Java层面交互时，都体现为类、对象。

### 什么是方法重载以及规则（\*\*\*）?

在同一个类中,允许存在一个以上的同名方法,只要它们的参数个数、参数类型、参数顺序不同即可。

与返回值类型/修饰符/方法体/形参名称无关,在调用时,jvm虚拟机通过参数列表的不同来区分同名方法。

方法重载有分为: 成员方法重载 和 构造方法重载;

注：

重载存在于父类和子类，当前类中。

重载可以抛出不同的异常，可以有不同的修饰符。

### 形参和实参的区别（\*）?

形参：形式参数，定义在方法的声明，相当于变量的声明。

实现：实际参数，定义在方法的调用，相当于给形参赋值。

### 值传递和引用传递的区别（\*\*\*）?

参数是基本数据类型时，将实参基本数据类型变量的“数据值”副本传递给形参，简称值传递。因此，就算是改变了对象副本，也不会影响源对象的值。

参数是引用数据类型时，将实参引用数据类型变量的“地址值”(对象的引用)传递给形参，简称引用传递。因此，对引用对象所做的改变会反映到所有的对象上。

### 谈谈return关键字的使用（\*）？

① 结束方法 ② 针对于有返回值的方法，return + 返回数据

### new关键字的作用（\*）?

1、在堆内存中给对象分配空间;

2、给成员变量进行默认初始化;

### 构造函数的特点（\*）?

1、构造函数的方法名与类名相同

2、构造函数没有返回值类型

3、构造函数可以不写,系统会默认赋予一个无参构造函数;如果定义了构造函数，那么系统不再提供默认的无参构造函数。

4、构造函数的主要作用是完成对对象成员的初始化工作

### 构造函数和成员方法的区别（\*\*\*）?

1、构造函数名要和类名一致,成员方法是自定义名称,最好是动词;

2、构造函数不需要定义返回值类型,成员方法必须定义返回值类型;

3、构造函数是创建对象以及完成对对象成员的初始化工作,成员方法代表对象的某一个行为;

4、构造函数是当使用new关键字的时候,由jvm来调用; 成员方法是由实例化出来的对象来调用;

### this关键字的理解（\*）?

在Java中，this关键字比较难理解，它的作用和其词义很接近。

它在方法内部使用，即这个方法所属对象的引用；

它在构造器内部使用，表示该构造器正在初始化的对象。

当在方法内需要用到调用该方法的对象时，就用this。

我们还可以用this来区分属性和局部变量。

this可以调用成员变量、成员方法、构造器；

### 请分别阐述面向对象三大特征（\*\*\*）?

三大特性是：封装,继承,多态

所谓封装，就是隐藏对象内部的复杂性，只对外公开简单的接口。便于外界调用，从而提高系统的可扩展性、可维护性。

所谓继承，就是子类继承父类‘所有’的特征和行为，使得子类对象（实例）具有父类的成员变量和方法，或类从父类继承方法，使得子类具有父类相同的行为。

所谓多态，就是程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编译期并不确定，而是在程序运行期间才确定，即一个引用变量倒底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。

### 继承的规则（\*）?

1、一个类可以被多个子类继承。

2、**Java中类的单继承性**：**一个类只能有一个父类（直接父类）**。

3、子父类是相对的概念。

4、子类直接继承的父类，称为：直接父类。间接继承的父类称为：间接父类。

5、子类继承父类以后，就获取了直接父类以及所有间接父类中声明的属性和方法。

### 四种访问权限修饰符是什么？它们的访问范围分别是什么（\*）？

Java的四种访问权限修饰符public、protected、缺省、 private；

public：同一个类、同一包中、不同包中的子类、不同包中的非子类都可以访问。

protected：同一个类、同一包中、不同包中的子类可以访问。

缺省：同一个类、同一包中可以访问。

private：只能在当前类中访问。

### 什么是方法的重写以及重写的规则（\*\*\*）?

从父类继承过来的方法，不满足子类的需求，那么就需要对继承而来的方法进行改造，也称为方法的重写、覆盖。

重写的规则：

1、子类重写的方法必须和父类被重写的方法具有相同的方法名称、参数列表；

2、子类重写的方法的返回值类型不能大于父类被重写的方法的返回值类型；

如果父类被重写方法的返回值类型是void或基本数据类型，那么子类重写的方法的返回值类型必须和父类保持一致。

如果父类被重写的方法的返回值类型是A，那么子类重写的方法的返回值类型可以是A也可以是A的子类。

3、子类重写的方法使用的访问权限不能小于父类被重写的方法的访问权限；

4、子类方法抛出的异常不能大于父类被重写方法的异常；

5、声明为final的方法不能被重写；声明为static的方法不存在重写；

### 如果现在父类的一个方法定义成private访问权限，在子类中将此方法声明为default访问权限，那么这样还叫重写吗？

NO；私有的方法不存在重写，就算定义了，只能子类的扩展。

### 成员变量会发生重写吗？

不会；

### 方法重写和方法重载的区别（\*\*\*）?

方法的重写Overriding 和重载Overloading 是Java 多态性的不同表现。

重写 Overriding是父类与子类之间多态性的一种表现，重载 Overloading 是一个类中多态性的一种表现。如果在子类中定义某方法与其父类有相同的名称和参数， 我们说该方法被重写 (Overriding)。子类的对象使用这个方法时，将调用子类中的定义，对它而言，父类中的定义如同被“屏蔽”了。如果在一个类中定义了多个同名的方法，它们或有不同的参数个数或有不同的参数类型，则称为方法的重载(Overloading)。

### super关键字的作用（\*）?

super可用于访问父类中定义的属性；

super可用于调用父类中定义的成员方法；

super可用于在子类构造器中调用父类的构造器；

### super和this的区别（\*\*\*）?

super(参数)：调用基类中的某一个构造函数（应该为构造函数中的第一条语句）；this(参数)：调用本类中另一种形成的构造函数（应该为构造函数中的第一条语句）；

super：它引用当前对象的父类中的成员（用来访问父类中成员数据或函数，基类与派生类中有相同成员定义时如：super.变量名 super.成员函数据名（实参）；this：它代表当前对象名（在程序中易产生二义性之处，应使用 this 来指明当前对象；如果函数的形参与类中的成员数据同名，这时需用 this 来指明成员变量名）；

调用super()必须写在子类构造方法的第一行，否则编译不通过。每个子类构造方法的第一条语句，都是隐含地调用 super()，如果父类没有这种形式的构造函数，那么在编译的时候就会报错。

super() 和 this() 类似，区别是，super() 从子类中调用父类的构造方法，this() 在同一类内调用其它方法。

super() 和 this() 均需放在构造方法内第一行。

尽管可以用this调用一个构造器，但却不能调用两个。

this 和 super 不能同时出现在一个构造函数里面，因为this必然会调用其它的构造函数，其它的构造函数必然也会有 super 语句的存在，所以在同一个构造函数里面有相同的语句，就失去了语句的意义，编译器也不会通过。

this() 和 super() 都指的是对象，所以，均不可以在 static 环境中使用。包括：static 变量,static 方法，static 语句块。

从本质上讲，this 是一个指向本对象的指针, 然而 super 是一个 Java 关键字。

### 非继承关系的初始化顺序（\*）?

静态代码块和静态成员

构造代码块和普通成员

构造函数

### 继承关系的初始化顺序（\*）?

父类静态代码块和父类静态成员

子类静态代码块和子类静态成员

父类构造代码块和普通成员

父类构造函数

子类构造代码块和普通成员

子类构造函数

### 什么是向上转型(上溯造型)和向下转型（\*\*\*）?

Employee是父类,SalariedEmployee是子类;

向上转型: 父类引用变量指向子类实例对象;

Employee salariedEmployee = new SalariedEmployee();

向下转型: 子类引用变量指向父类实例对象;

SalariedEmployee salariedEmployee = (SalariedEmployee)new Employee();

### 向上转型的特点（\*\*\*）?

1、既可以使用父类的共性,也可以使用子类强大的功能（重写的方法）

2、不能使用子类的特性(对于子类的扩展,不能调用)

### Java实现多态有三个必要条件（\*）?

继承: 在多态中必须存在有继承关系的子类和父类。

重写: 子类对父类中某些方法进行重新定义，在调用这些方法时就会调用子类的方法。

向上转型: 在多态中需要将父类引用指向子类对象，只有这样该引用才能够具备既能调用父类的方法和子类重写的方法。

### Java多态的实现形式与表现在哪里（\*\*\*）?

实现形式：

1、继承类

2、实现接口

多态表现：

多态要有动态绑定，否则就不是多态，方法重载也不是多态（因为方法重载是编译期决定好的，没有后期也就是运行期的动态绑定）当满足这三个条件:1、有继承 2、有重写 3、要有父类引用指向子类对象。

### Java多态的优点（\*）?

可替换性：多态对已存在代码具有可替换性。例如，多态对圆Circle类工作，对其他任何圆形几何体，如圆环，也同样工作。

可扩充性：多态对代码具有可扩充性。增加新的子类不影响已存在类的多态性、继承性，以及其他特性的运行和操作。实际上新加子类更容易获得多态功能。

### Java类的属性和成员变量（\*）?

对象属性是指get或者set方法名，去掉get或者set后,把剩余的部分首字母改为小写后,即为这个类的属性。

成员变量是指在类中方法外定义的变量;

### 对象类型转换规则（\*）？

从子类到父类的类型转换可以自动进行；

从父类到子类的类型转换必须通过造型(强制类型转换)实现；

无继承关系的引用类型间的转换是非法的；

在造型前可以使用instanceof操作符测试一个对象的类型；

### 代码块分类、执行机制、执行顺序（\*）?

分类：

普通代码块：在方法或语句中出现的{}就称为普通代码块。

构造块：直接在类中定义且没有加static关键字的代码块称为{}构造代码块。

静态代码块：在java类中使用static关键字声明的代码块。

执行机制：

普通代码块和一般的语句执行顺序由他们在代码中出现的次序决定--“先出现先执行”

构造代码块在创建对象时被调用，每次创建对象都会被调用，并且构造代码块的执行次序优先于类构造函数。

每个静态代码块只会执行一次。由于JVM在加载类时会执行静态代码块，所以静态代码块先于主方法执行。

执行顺序：

静态代码块 > mian方法 > 构造代码块 > 构造方法 > 普通代码块;

### static能修饰哪些结构？修饰以后，有什么特点（\*\*\*）？

在Java类中，可用static修饰属性、方法、代码块、内部类。

被修饰后的成员具备以下特点：

static修饰变量，我们称之为静态变量。静态变量属于类，在内存中只有一个副本（所有实例都指向同一个内存地址）。只要静态变量所在的类被加载，这个静态变量就会被分配空间，

static修饰方法，我们称之为静态方法。随着类的加载而加载，优先于对象存在。static 方法中不能使用this和super关键字，不能调用非static方法，只能访问所属类的静态成员变量和成员方法。

static修饰代码块，我们称之为静态代码块。static代码块在类中是独立与成员变量和成员函数的代码块，他不在任何一个方法体内，JVM在加载类的时候会执行static代码块，如果有多个static代码块，JVM将会按顺序来执行，static代码块经常会被用来初始化静态变量，需要注意的是static代码块只会被执行一次。

static也可以修饰类，但是只能修饰内部类。

### 是否可以在static 环境中访问非static 变量（\*）？

static 变量在 Java 中是属于类的，它在所有的实例中的值是一样的。当类被 Java 虚拟机载入的时候，会对 static 变量进行初始化。如果你的代码尝试不用实例来访问非 static的变量，编译器会报错，因为这些变量还没有被创建出来，还没有跟任何实例关联上。

### 静态方法和实例方法有何不同（\*\*\*）？

**静态方法和实例方法的区别主要体现在两个方面：**

在外部调用静态方法时，可以使用"类名.方法名"的方式，也可以使用"对象名.方法名"的方式。而实例方法只有后面这种方式。也就是说，调用静态方法可以无需创建对象。

静态方法在访问本类的成员时，只允许访问静态成员（即静态成员变量和静态方法），而不允许访问实例成员变量和实例方法；实例方法则无此限制。

### static关键字会改变类中成员的访问权限吗（\*）？

**Java中的static关键字不会影响到变量或者方法的作用域。**在Java中能够影响到访问权限的只有private、public、protected（包括包访问权限）这几个关键字。

### 能通过this访问静态成员变量吗？

可以；

### final 可以用来修饰哪些结构，分别表示什么意思（\*\*\*）?

在Java中声明类、变量和方法时，可使用关键字final来修饰,表示“最终的”。

final修饰的类不能被继承。

final修饰的方法不能被子类重写，方法一旦被修饰，JVM会尝试将其内联，以提高运行效率。

final修饰的变量不可以被改变，如果修饰引用，那么表示引用不可变，引用指向的内容可变。final修饰的变量我们称之为常量，会在编译阶段存入常量池中。

### static能作用于局部变量么（\*）？

在C/C++中static是可以作用域局部变量的，但是在Java中切记：static是不允许用来修饰局部变量。不要问为什么，这是Java语法的规定。

### 类的属性赋值的位置有哪些？先后顺序为何？

①默认初始化

②显式初始化/⑤在构造代码块中赋值

③构造器中初始化

④有了对象以后，可以通过"对象.属性"或"对象.方法"的方式，进行赋值

① - ② / ⑤ - ③ - ④

### 如下代码的运行结果?

String a = "hello2";

final String b = "hello"; // 编译期常量

String d = "hello"; // 运行时常量

String c = b + 2; //hello2

String e = d + 2; //hello2

System.out.println((a == c)); //true

System.out.println((a == e)); //false

### 抽象类的意义（\*\*\*）?

1. 为其他子类提供一个公共的类型。
2. 封装子类中重复定义的内容。
3. 定义抽象方法，子类虽然有不同的实现，但是定义时是一致的。

### 一个抽象类中可以定义构造器吗？

可以；

### 是否可以这样理解：抽象类就是比普通类多定义了抽象方法，除了不能直接进行类的实例化操作之外，并没有任何的不同？

可以；

### Java能不能多继承（\*\*\*）?

Java类和类之间只能是单继承,不能是多继承;但是可以多层继承;

Java接口和接口之间可以是多继承;

### 接口是否能继承接口? 抽象类是否能实现(implements)接口? 抽象类是否能继承非抽象的类?

能，能，能

### abstract class和 interface的区别（\*\*\*）?

1、抽象类是由abstract修饰的类,接口不是类。

2、抽象类可以定义任意访问权限的数据、抽象方法、普通成员方法; 接口只能定义公共的静态的常量、公共抽象方法;

3、对于子类而言,它只能继承一个抽象类,但是却可以实现多个接口。如果一个类实现了接口，它必须实现接口中声明的所有方法；但是类可以不实现抽象类声明的所有抽象方法，但是前提该类必须也得声明为抽象类。

4、抽象层次不同。抽象类是对类抽象，而接口是对行为的抽象。抽象类是对整个类整体进行抽象，包括属性、行为，但是接口却是对类局部（行为）进行抽象。

5、跨域不同。抽象类所跨域的是具有相似特点的类，而接口却可以跨域不同的类。

6、设计层次不同。抽象类是自底向上抽象而来的，接口是自顶向下设计出来的。

7、抽象类和接口都不能被实例化。

### 内部类的分类（\*\*\*）?

**成员内部类:**

最普通的内部类,它的定义位于另一个类的内部;

成员内部类可以无条件访问外部类的所有成员变量、成员方法、静态变量、静态方法;但是外部类想要访问内部类的成员,必须先实例化内部类对象;

如果要创建成员内部类的对象,前提是必须存在一个外部类的对象。

成员内部类可以使用任意的访问修饰符。

成员内部类不能有静态方法、静态属性和静态初始化块。

**静态内部类:**

静态内部类也是定义在另一个类里面的类,只不过在类的前面多了一个关键字static。

静态内部类不能使用外部类的非static成员变量或者方法。外部类想要访问静态内部类成员,必须先实例化内部类对象;

如果要创建静态内部类的对象,不需要依赖于外部类对象,但是需要依赖外部类。

静态内部类可以定义成员变量、成员方法甚至于静态变量、静态方法和静态代码块;

静态内部类可以使用任意的访问修饰符。

**局部内部类:**

局部内部类是定义在一个方法或者一个作用域里面的类

局部内部类可以无条件访问外部类的所有成员变量、成员方法、静态变量、静态方法;

如果要创建局部内部类的对象,前提是必须存在一个外部类的对象。

局部内部类只能使用默认的访问修饰符;

局部内部类不能有静态方法、静态属性和静态初始化块。

简单来说: 它和成员内部类的区别在于局部内部类的访问仅限于方法内或者该作用域内。

**匿名内部类:**

匿名内部类有多种形式,其中最常见的一种形式莫过于新建一个接口对象 / 抽象类对象,并且实现这个接口声明 / 类中原有的方法了

直接new一个接口,并实现这个接口声明的方法,在这个过程其实会创建一个匿名内部类实现这个接口,并重写接口声明的方法,然后再创建一个这个匿名内部类的对象

new 一个已经存在的接口/抽象类，并且选择性的实现这个类中的一个或者多个非 final 的方法，这个过程会创建一个匿名内部类对象继承对应的类 / 抽象类，并且重写对应的方法。

### 类和类之间的关系（\*\*\*）?

根据类与类之间的耦合度从弱到强排列,UML 中的类图有以下几种关系：依赖关系、关联关系、聚合关系、组合关系、泛化关系和实现关系。其中泛化和实现的耦合度相等,它们是最强的。

依赖（Dependency）关系是一种使用关系,它是对象之间耦合度最弱的一种关联方式,是临时性的关联。在代码中,某个类的方法通过局部变量、方法的参数来访问另一个类（被依赖类）中的某些方法来完成一些职责。在 UML 类图中,依赖关系使用带箭头的虚线来表示,箭头从使用类指向被依赖的类。

关联（Association）关系是对象之间的一种引用关系,用于表示一类对象与另一类对象之间的联系。在 UML 类图中,双向的关联可以用带两个箭头或者没有箭头的实线来表示,单向的关联用带一个箭头的实线来表示,箭头从使用类指向被关联的类。也可以在关联线的两端标注角色名,代表两种不同的角色。

聚合（Aggregation）关系是关联关系的一种,是强关联关系,是整体和部分之间的关系,是 has-a 的关系。聚合关系也是通过成员对象来实现的,其中成员对象是整体对象的一部分,但是成员对象可以脱离整体对象而独立存在。在 UML 类图中，聚合关系可以用带空心菱形的实线来表示,菱形指向整体。

组合（Composition）关系也是关联关系的一种,也表示类之间的整体与部分的关系,但它是一种更强烈的聚合关系,是 contains-a 关系。在组合关系中，整体对象可以控制部分对象的生命周期,一旦整体对象不存在,部分对象也将不存在,部分对象不能脱离整体对象而存在。在 UML 类图中,组合关系用带实心菱形的实线来表示,菱形指向整体。

一般关联关系、聚合关系、组合关系在代码中通常将一个类的对象作为另一个类的成员变量来实现关联关系。

泛化（Generalization）关系是对象之间耦合度最大的一种关系,表示一般与特殊的关系,是父类与子类之间的关系,是一种继承关系,是 is-a 的关系。在 UML 类图中,泛化关系用带空心三角箭头的实线来表示,箭头从子类指向父类。

实现（Realization）关系是接口与实现类之间的关系。在 UML 类图中,实现关系使用带空心三角箭头的虚线来表示,箭头从实现类指向接口。

### 什么是高内聚低耦合?

高内聚低耦合,是软件工程中的概念,是判断软件设计好坏的标准,主要用于程序的面向对象的设计,主要看类的内聚性是否高,耦合度是否低。

内聚是从功能角度来度量模块内的联系,一个好的内聚模块应当恰好做一件事,它描述的是模块内的功能联系;

耦合是软件结构中各模块之间相互连接的一种度量,耦合强弱取决于模块间接口的复杂程度、进入或访问一个模块的点以及通过接口的数据。

### 高内聚低耦合的目的?

目的是使程序模块的可重用性、移植性大大增强。

### 接口隔离原则和单一职责原则区别?

单一职责原则原注重的是职责; 而接口隔离原则注重对接口依赖的隔离。

单一职责原则主要是约束类,其次才是接口和方法,它针对的是程序中的实现和细节;而接口隔离原则主要约束接口,主要针对抽象,针对程序整体框架的构建;

### 什么叫Java的内存泄露与内存溢出（\*\*\*）?

内存泄漏（memoryleak）：就是存在一些被分配的对象但是这些对象不会再被使用,仍存在该内存对象的引用，导致无法释放内存空间。一次内存泄露危害可以忽略，但是任其发展最终会导致内存溢出，如读取文件后流要及时关闭、数据库连接要及时释放。

内存溢出（outofmemory）：就是指应用程序在申请内存时，没有足够的内存空间供其使用。如我们在项目中对于大批量数据的导入，尽量采用分批提交的方式。

### 如下代码的执行结果（\*）?

String str1 = "abc";

String str2 = "abc";

String str3 = new String("abc");

// == 比较引用类型,比较的是地址

System.out.println(str1 == str2); // true

System.out.println(str1 == str3); // false

// equals比较的是字符串的值

System.out.println(str1.equals(str2)); // true

System.out.println(str1.equals(str3)); // true

### Java对象的生命周期（\*\*\*）?

1.创建阶段(Created): 通过new 调用构造函数,就是创建阶段

2.应用阶段(In Use): 对象至少被一个强引用持有着。

3.不可见阶段(Invisible): 程序的执行已经超出了该对象的作用域了

4.不可达阶段(Unreachable): 指该对象不再被任何强引用所持有。

5.收集阶段(Collected): 该对象已经处于“不可达阶段”并且垃圾回收器已经对该对象的内存空间重新分配做好准备时,则对象进入了“收集阶段”。

6.终结阶段(Finalized): 当对象执行完finalize()方法后仍然处于不可达状态时,则该对象进入终结阶段。在该阶段是等待垃圾回收器对该对象空间进行回收。

7.对象空间重分配阶段(De-allocated): 垃圾回收器对该对象的所占用的内存空间进行回收或者再分配了,则该对象彻底消失了,称之为“对象空间重新分配阶段”。

### 请说一下递归的概念（\*）?

递归是计算机的重要概念,就是在过程或函数里调用自身,以达到能使程序更加简洁、清晰。

### 什么是异常处理机制?

异常处理机制能让程序在异常发生时，按照代码的预先设定的异常处理逻辑，针对性地处理异常，让程序尽最大可能恢复正常并继续执行，且保持代码的清晰。

### 异常对象的生成的两种方式?

由虚拟机自动生成：程序运行过程中，虚拟机检测到程序发生了问题，如果在当前代码中没有找到相应的处理程序，就会在后台自动创建一个对应异常类的实例对象并抛出——自动抛出。

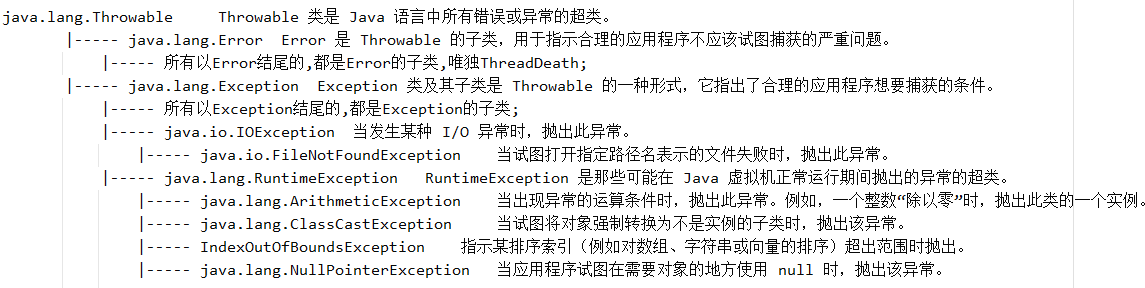
由开发人员手动创建：Exception exception = new ClassCastException();——创建好的异常对象不抛出对程序没有任何影响，和创建一个普通对象一样。

### 异常的抛出机制（\*）?

如果一个方法内抛出异常，该异常对象会被抛给调用者方法中处理。如果异常没有在调用者方法中处理，它继续被抛给这个调用方法的上层方法。这个过程将一直继续下去，直到异常被处理。这一过程称为捕获(catch)异常。

如果一个异常回到main()方法，并且main()也不处理，则程序运行终止。

### 异常体系结构?



Java标准库内建了一些通用的异常，这些类以Throwable为顶层父类。

Throwable又派生出Error类和Exception类。

错误：Error类以及它的子类的实例，代表了JVM本身的错误。错误不能被程序员通过代码处理，Error很少出现。因此，程序员应该关注Exception为父类的分支下的各种异常类。

异常：Exception以及它的子类，代表程序运行时发生的各种不期望发生的事件。可以被Java异常处理机制使用，是异常处理的核心。

### Error和Exception的区别（\*\*\*）?

Error和Exception都有一个共同的根类是Throwable类。

Error是系统中的错误，程序员是不能改变的和处理的，一般是指与虚拟机相关的问题,对于这类错误的导致的应用程序中断，仅靠程序本身无法恢复和和预防，遇到这样的错误，建议让程序终止。因此我们编写程序时不需要关心这类错误。

Exception，也就是我们经常见到的一些异常情况，表示程序可以处理的异常，可以捕获且可能恢复。遇到这类异常，应该尽可能处理异常，使程序恢复运行，而不应该随意终止异常。

### Java异常的分类（\*\*\*）?

Java的异常从广义上分为检查异常（checked exceptions）和非检查的异常（unchecked exceptions）。

检查异常: 编译器要求你必须处置的异常。也就是说，你代码还没运行呢，编译器就会检查你的代码，会不会出现异常，要求你对可能出现的异常必须做出相应的处理。

非检查异常:编译器不要求强制处置的异常，虽然你有可能出现错误，但是编译器不会在编译的时候检查，没必要，也不可能。

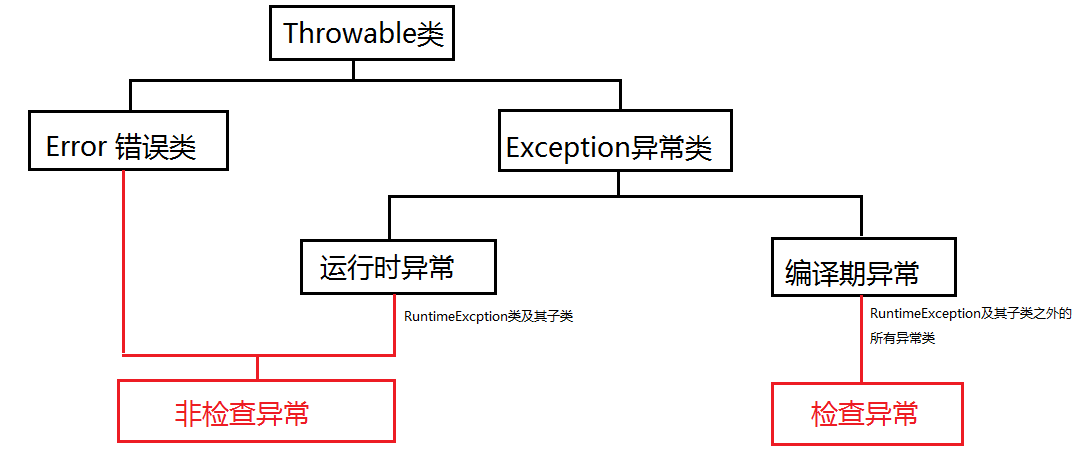
Java的异常从狭义上分为运行时异常和编译期异常。

运行时异常：是RuntimeException类及其子类异常，这些异常是非检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生。

编译期异常：是RuntimeException以外的异常，类型上都属于Exception类及其子类。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。

检查异常 = 编译期异常;

非检查异常 = 运行时异常 + Error;



### 常见的异常有哪些（\*\*\*）?

**java.lang.RuntimeException: 运行时异常**

ClassCastException: 类类型转换异常，当试图将对象强制转换为不是实例的子类时，抛出该异常；

ArrayIndexOutOfBoundsException: 数组下标越界异常，当你使用不合法的索引访问数组时会抛出该异常；

NullPointerException: 空指针异常，通过null进行方法和属性调用会抛出该异常；

ArithmeticException: 算术运算异常，除数为0，抛出该异常；

NumberFormatException: 数字转换异常，当试图将一个String转换为指定的数字类型，而该字符串确不满足数字类型要求的格式时，抛出该异常；

InputMismatchException: 输入不匹配异常，输入的值数据类型与设置的值数据类型不能匹配。

**java编译时异常：**

SQLException: SQL异常，提供有关数据库访问错误或其他错误的信息的异常；

IOExeption: 输入输出异常，表示发送了某种I/O异常的信号。

FileNotFoundException: 文件找不到异常，通常是两种情况：1、系统找不到指定的路径 2、拒绝访问（指定的是目录时，就会报拒绝访问异常）

EOFException: 文件已结束异常，抛出EOFException一定是因为连接断了还在继续read；

java.lang.ClassNotFoundException: 类找不到异常，当我们通过配置文件去查找一个类的时候，如果配置路径写错，就会抛出该异常，比如：web.xml文件中根本就不存在该类的配置或者配置的路径写错；（比较常见）

### try...catch...finally的作用（\*\*\*）?

try块中放可能发生异常的代码。

catch块用于捕获并处理一个特定的异常,catch块可以有多个;

finally块无论异常是否发生,异常是否匹配被处理,都会执行,主要做一些清理工作,比如释放资源;

try必须有,catch和finally都是可选的;但是try不能单独存在,至少要和catch或finally其中一个结合使用;

### try...catch...finally和throws处理异常的区别（\*）?

try...catch...finally是在方法内部处理异常,通过统一的异常处理方案;

throws是方法本身不知道如何处理这样的异常,或者说让调用者处理更好,调用者需要为可能发生的异常负责。

### throws和throw的区别（\*\*\*）?

throws定义在方法的声明中,表示调用该方法可能出现一个或多个异常,由该方法的调用者来处理;throws后面跟的是一个或多个异常类型，让它的使用者知道需要捕获的异常的类型; throws表示出现异常的一种可能性，并不一定会发生这种异常。

throw定义在方法内部,表示方法体内,表示抛出异常,由方法体内的语句处理;throw是具体向外抛出异常的动作，该语句的后面必须是一个异常对象; 执行throw一定抛出了某种异常。

两者都是消极处理异常的方式,只是抛出或者可能抛出异常,但是不会由函数去处理异常,真正的处理异常由函数的上层调用处理。

### finally和return代码阅读面试题（\*\*\*）?

在try块中即便有return,break,continue等改变执行流的语句,finally也会执行。

finally中的return会覆盖 try 或者catch中的返回值。

finally中的return会抑制（消灭）前面try或者catch块中的异常

finally中的异常会覆盖（消灭）前面try或者catch中的异常

### finally的使用注意事项（\*\*\*）?

不要在finally中使用return。

不要在finally中抛出异常。

减轻finally的任务,不要在finally中做一些其它的事情,finally块仅仅用来释放资源是最合适的。

尽量将所有的return写在函数的最后面，而不是try ... catch ... finally中。

### final，finally和finalize的比较（\*\*\*）?

final修饰类,类不能被继承 final 修饰方法,方法不能被重写，final 修饰变量,变量不能被修改。

finally 是异常语句块的一部分,无论是否有异常,都会被执行。

finalize 是Object 类的一个方法，在垃圾收集器执行的时候会调用被回收对象的此方法，可以覆盖此方法提供垃圾收集时的其他资源回收，例如关闭文件等。

### 进程和线程的区别?

1、线程是程序执行的最小单位，而进程是操作系统分配资源的最小单位；

2、一个进程由一个或多个线程组成，线程是一个进程中代码的不同执行路线；

3、进程之间相互独立，但同一进程下的各个线程之间共享程序的内存空间(包括代码段，数据集，堆等)及一些进程级的资源(如打开文件和信号等)，某进程内的线程在其他进程不可见；

4、调度和切换：线程上下文切换比进程上下文切换要快得多；

### 当我们实现Runnable接口实现多线程的时候,开启线程调用的Runnable接口实现类的run()还是Thread类的run()?

调用的是Thread类的run(),在Thread的run()内部判断target是否为null,当不为null的时候,调用Runnable接口实现类的run();

### 线程/操作系统的任务调度方式（\*\*\*）?

大部分操作系统的任务调度是**采用时间片轮转的抢占式调度方式**，也就是说一个任务执行一小段时间后强制暂停去执行下一个任务，每个任务轮流执行。

任务执行的一小段时间叫做时间片，任务正在执行时的状态叫运行状态，任务执行一段时间后强制暂停去执行下一个任务，被暂停的任务就处于就绪状态，等待下一个属于它的时间片的到来。

任务调度方式只提到了两种状态:

运行状态: 任务(线程)正在执行时的状态。

就绪状态: 任务等待被执行的状态。

只有处于就绪状态的线程才可能被任务调用机制调度到。

### 并行和并发?

并行就是两个任务同时运行，就是甲任务进行的同时，乙任务也在进行。

并发是指两个任务都请求运行，而处理器只能按受一个任务，就把这两个任务安排轮流进行，由于时间间隔较短，使人感觉两个任务都在运行。

### 多线程的创建方式（\*）?

JDK5之前:

继承Thread类

实现Runnable接口

JDK5之后:

使用Callable接口创建线程

通过线程池创建线程

### 继承Thread类 和 实现Runnable接口的对比（\*）?

查看源码的区别:

a、继承Thread : 由于子类重写了Thread类的run(), 当调用start()时, 直接找子类的run()方法。

b、实现Runnable : 构造函数中传入了Runnable的引用, 成员变量记住了它, start()调用run()方法时内部判断成员变量Runnable的引用是否为空, 不为空编译时看的是Runnable的run(),运行时执行的是子类的run()方法（也就是实现了Runnable接口并重写了run()方法的类中的run()方法）。

继承Thread：

好处是:可以直接使用Thread类中的方法,代码简单。

弊端是:如果已经有了父类,就不能用这种方法。

实现Runnable接口（推荐）：

好处是:即使自己定义的线程类有了父类也没关系,因为有了父类也可以实现接口,而且接口是可以多实现的，避免了单继承的局限性。多个线程可以共享同一个接口实现类的对象，非常适合多个相同线程来处理同一份资源。

弊端是:不能直接使用Thread中的方法需要先获取到线程对象后,才能得到Thread的方法,代码复杂。

### 什么是守护线程（\*\*\*）?

java线程中有两种线程,一种是用户线程(非守护线程),一种是守护线程。

守护线程是一种特殊的线程,它具有陪伴的含义。当进程中不存在非守护线程了,则守护线程自动销毁。

典型的就是垃圾回收线程。当进程中没有非守护线程了,则垃圾回收线程没有存在的必要,自动销毁。

### 线程的生命周期（\*\*\*）?

新建状态（New）：当一个Thread类或其子类的对象被声明并创建时，新生的线程对象处于新建状态。

就绪状态（Runnable）：处于新建状态的线程对象被start()后，线程即进入就绪状态。

运行状态（Running）：当就绪状态的线程被操作系统的任务调度机制调度到，此时线程才得以真正执行，即进入到运行状态。

阻塞状态（Blocked）：处于运行状态中的线程由于某种原因，暂时放弃对CPU的使用权，停止执行，此时进入阻塞状态，直到其再次进入到就绪状态，才有机会再次被CPU调用以进入到运行状态。

死亡状态（Dead）：线程完成了它的全部工作或线程被提前强制性地中止或出现异常导致结束。

### 阻塞状态的分类（\*\*\*）?

根据阻塞产生的原因不同，阻塞状态又可以分为三种：

1.等待阻塞：运行状态中的线程执行wait()方法，使本线程进入到等待阻塞状态；

2.同步阻塞：线程在获取synchronized同步锁失败(因为锁被其它线程所占用)，它会进入同步阻塞状态；

3.其他阻塞：通过调用线程的sleep()或join()或发出了I/O请求时，线程会进入到阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。

### 什么线程安全问题（\*）?

多线程并发操作同一数据时，会造成操作的不完整性，会破坏数据；

问题的原因：当多条语句在操作同一个线程共享数据时，一个线程对多条语句只执行了一部分，还没有执行完，另一个线程参与进来执行。导致共享数据的错误。

多个线程操作同一数据，如果当前线程a没有操作完该数据，其他线程参与进来执行，那么就会导致数据的错误，这就是线程安全问题；

一定要明确的是：同一数据

解决办法：

对多条操作共享数据的语句，只能让一个线程都执行完，在执行过程中，其他线程不可以参与执行。

使用同步技术可以解决这种问题, 把操作数据的代码进行同步, 不要多个线程一起操作；

### 什么是线程同步?

线程同步其实就是一种等待机制，多个需要同时访问此对象的线程进入这个对象的等待池形成队列，等待前面线程使用完毕，下个线程在使用。

每个对象都有一把锁;队列 + 锁 就是用来保证多线程安全性；

### 解决线程安全的方法（\*\*\*）?

synchronized代码块：使用synchronized关键字加上一个锁对象来定义一段代码, 这就叫同步代码块；

synchronized方法: 使用synchronized关键字修饰一个方法, 该方法中所有的代码都是同步的;

Lock锁: 从JDK5开始，Java提供了更强大的线程安全机制 - 通过显式定义同步锁对象来实现同步，同步锁使用Lock及其子类实现；

### Java 中 ++ 操作符是线程安全的吗（\*\*\*）？

不是线程安全的操作。它涉及到多个指令，如读取变量值，增加，然后存储回内存，这个过程可能会出现多个线程交差。

### 释放锁的操作（\*\*\*）?

当前线程的同步方法、同步代码块执行结束。

当前线程在同步代码块、同步方法中遇到break、return终止了该代码块、该方法的继续执行。

当前线程在同步代码块、同步方法中出现了未处理的Error或Exception，导致异常结束。

当前线程在同步代码块、同步方法中执行了线程对象的wait()方法，当前线程暂停，并释放锁。

### synchronized和Lock的对比（\*\*\*）?

1、Lock是显式锁（手动开启和关闭锁，别忘记关闭锁），synchronized是隐式锁，出了作用域自动释放;

2、Lock只有代码块锁，synchronized有代码块锁和方法锁；

3、使用Lock锁，JVM将花费较少的时间来调度线程，性能更好，而且具有更好的扩展性（提供更多子类）;

4、优先使用顺序:

Lock -> 同步代码块（已经进入了方法体，分配了相应资源） -> 同步方法（在方法体之外）

### 什么是死锁（\*）?

多个线程各自占用一些共享资源，并且互相等待其他线程占有的资源才能运行，而导致两个或多个线程都等待对方释放资源，都停止执行的情况，某一个同步块同时拥有“两个以上对象的锁”时，就可能会出现“死锁”问题；

### Java线程的通信方式（\*\*\*）?

线程通信主要可以分为三种方式，分别为共享内存、消息传递和管道流。

1、共享内存：线程之间共享程序的公共状态，线程之间通过读-写内存中的公共状态来隐式通信。

volatile共享内存

2、消息传递：线程之间没有公共的状态，线程之间必须通过明确的发送信息来显示的进行通信。

等待/通知机制、join方式、Condition

1. 管道流：管道输入/输出流的形式。

### sleep() 和 wait()的异同（\*\*\*）?

相同点：一旦执行方法，都可以使得当前的线程进入阻塞状态。

不同点：

1）两个方法声明的位置不同：Thread类中声明sleep() , Object类中声明wait()。

2）调用的要求不同：sleep()可以在任何需要的场景下调用。 wait()必须使用在同步代码块或同步方法中。

3）关于是否释放同步监视器：如果两个方法都使用在同步代码块或同步方法中，sleep()不会释放锁，wait()会释放锁。

### 线程中notify和notifyAll方法的作用（\*）?

notify和notifyAll方法是一个native方法,并且都是final的，不允许子类重写。

notify是唤醒一个在此对象监视器上等待的线程(监视器相当于就是锁的概念),选择是任意性的;

notifyAll是会唤醒在此对象监视器上等待的所有线程,而不是一个线程。

notify和notifyAll只能在拥有对象监视器的所有者线程中调用,否则会抛出IllegalMonitorStateException异常

### Runnable 和 Callable创建线程对比（\*\*\*）?

无论是Runnable 还是 Callable都是用来创建线程的；

与使用Runnable相比， Callable功能更强大些：

相比run()方法，call()可以有返回值；

方法可以抛出异常；

支持泛型的返回值；

### 什么是线程池?

装有线程的池子,通过维护一定数量的线程池来达到多个线程的复用。

### 什么时候使用线程池（\*\*\*）?

几乎所有需要并发执行任务的程序都可以使用线程池。

1：单个任务处理时间比较短

2：需要处理的任务数量很大

### 线程池的作用（\*）?

如果并发的线程数量很多,并且每个线程都是执行一个时间很短的任务就结束了,这样频繁创建线程就会大大降低系统的效率,因为频繁创建线程和销毁线程需要时间,而创建并销毁线程的过程势必会消耗内存。

作用:

第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

第二：提高响应速度。当任务到达时,任务可以不需要等到线程创建就能立即执行。

第三：提高线程的可管理性。线程是稀缺资源,如果无限制地创建,不仅会消耗系统资源,还会降低系统的稳定性,使用线程池可以进行统一分配、调优和监控。

### 线程池生命周期及五种状态（\*\*\*）?

RUNNING: 运行状态,能接受新提交的任务,并且也能处理阻塞队列中的任务;线程池的初始化状态是RUNNING。换句话说,线程池被一旦被创建,就处于RUNNING状态,并且线程池中的任务数为0！

SHUTDOWN: 关闭状态,不再接受新提交的任务,但却可以继续处理阻塞队列中已保存的任务。 在线程池处于 RUNNING 状态时，调用 shutdown()方法会使线程池进入到该状态。

STOP: 停止状态,不能接受新任务,也不处理队列中的任务,会中断正在处理任务的线程。在线程池处于 RUNNING 或 SHUTDOWN 状态时,调用 shutdownNow() 方法会使线程池进入到该状态;

TIDYING: 整理状态,如果所有的任务都已终止了,workerCount (有效线程数) 为0,当线程池变为TIDYING状态时,会执行钩子函数terminated()。terminated()在ThreadPoolExecutor类中是空的，若用户想在线程池变为TIDYING时，进行相应的处理；可以通过重载terminated()函数来实现。

TERMINATED: 终止状态,在terminated() 方法执行完后进入该状态,默认terminated()方法中什么也没有做。线程池彻底终止，就变成TERMINATED状态

### 线程池的原理（\*\*\*）?

1、我们通过execute()或submit()把任务提交给线程池;

2、如果线程池中的线程数量小于corePoolSize(核心线程数量),则创建新线程来处理任务,即使线程池中的其他线程是空闲的;我们可以简单的认为corePoolSize就是线程池里面的最少线程数,而且核心线程不会销毁;

3、如果线程池中的线程数量大于等于corePoolSize且小于maximumPoolSize(最大线程数),当workQueue未满的时候任务添加到workQueue(队列)中,当workQueue满时才创建新的线程去处理任务；

4、如果线程池中的线程数量大于等于maximumPoolSize,这时如果workQueue已经满了,则通过RejectedExecutionHandler(拒绝策略)所指定的策略来处理任务;

5、当线程池的线程执行完任务之后,不会被立马销毁,而是又放到线程池等待重复使用;

6、当线程池中的线程数量大于corePoolSize的时候,如果这时没有新的任务提交,非核心线程等待的时间超过了keepAliveTime(结合TimeUnit来使用),那么就会被销毁;

7、当线程不够的时候,通过Executors.defaultThreadFactory()来创建线程;

8、如果阻塞队列满了并且没有空闲的线程,这时如果继续提交任务,就需要采取一种饱和(拒绝)策略处理该任务。

AbortPolicy: 直接抛出异常，这是默认策略；

CallerRunsPolicy: 用调用者所在的线程来执行任务；

DiscardOldestPolicy: 丢弃阻塞队列中靠最前的任务，并执行当前任务；

DiscardPolicy: 直接丢弃任务；

### 线程池的分类（\*\*\*）?

newFixedThreadPool: 固定线程池,核心线程数和最大线程数固定相等,而空闲存活时间为0毫秒,说明此参数也无意义,工作队列为最大为Integer.MAX\_VALUE大小的阻塞队列。

newCachedThreadPool: 带缓冲线程池,从构造看核心线程数为0,最大线程数为Integer最大值大小,超过0个的空闲线程在60秒后销毁,SynchronousQueue这是一个直接提交的队列,意味着每个新任务都会有线程来执行,如果线程池有可用线程则执行任务,没有的话就创建一个来执行,线程池中的线程数不确定,一般建议执行速度较快较小的线程,不然这个最大线程池边界过大容易造成内存溢出。

newSingleThreadExecutor: 创建一个单线程化的Executor,核心线程数和最大线程数均为1,每次只执行一个线程,多余的先存储到工作队列,一个一个执行,保证了线程的顺序执行。如果这个线程异常结束,会有另一个取代它,保证顺序执行。

newScheduledThreadPool: 调度线程池,即按一定的周期执行任务,即定时任务,对ThreadPoolExecutor进行了包装而已。

newSingleThreadScheduledExecutor: 创建一个单线程执行程序,它可安排在给定延迟后运行命令或者定期地执行。

### 线程池都有哪几种工作队列（\*）?

ArrayBlockingQueue: 是一个基于数组结构的有界阻塞队列,此队列按 FIFO（先进先出）原则对元素进行排序。

LinkedBlockingQueue: 一个基于链表结构的阻塞队列,此队列按FIFO(先进先出)排序元素，吞吐量通常要高于ArrayBlockingQueue。

SynchronousQueue: 一个不存储元素的阻塞队列。

PriorityBlockingQueue: 一个具有优先级的无限阻塞队列。

### ==和equals()的区别（\*\*\*）?

==: 可以比较基本数据类型变量和引用数据类型变量中;

基本类型比较值: 只要两个变量的值相等，即为true。

引用类型比较引用(是否指向同一个对象)：只有指向同一个对象时，==才返回true。

equals:

只能比较引用数据类型,不能比较基本数据类型;

默认情况下,比较的是地址;当自定义类使用equals()时，可以重写。用于比较两个对象的“内容”是否都相等。

### Java方法分类（\*）?

从大层面来说,Java方法分为: Java方法和本地方法。

Java方法是由Java语言编写,编译成字节码,存储在class文件中。

Java方法又分为: 成员方法、构造方法、静态方法;

本地方法是由其他语言(比如C,C++,或者汇编)编写的,编译成和处理器相关的机器代码。本地方法保存在动态连接库(DLL)中,格式是各个平台专有的。

### Object类中的registerNatives方法的作用（\*）?

registerNatives本质上就是一个本地方法,但这又是一个有别于一般本地方法的本地方法,这个方法应该是用来注册本地方法的。

先定义了registerNatives()方法,然后当该类被加载的时候,调用该方法完成对该类中本地方法的注册。

### Object类中的getClass方法的作用（\*）?

getClass方法是一个final方法,不允许子类重写,并且也是一个native方法。

返回当前运行时对象的Class对象;

### Object类中的hasCode方法的作用（\*）?

hashCode方法也是一个native方法。该方法返回对象的哈希码;

该方法返回对象的哈希码,它是一种算法，让同一个类的对象按照自己不同的特征尽量的有不同的哈希码;

### equals方法的理解（\*）?

equals是定义在Object类中,所有类都间接或直接继承Object类;

那么如果子类没有重写equals方法,那么equals比较的是地址;

如果重写equals方法,常规来说就是用来比较内容;

注: 重写equals,最好把hasCode也重写了;一般情况下都是使用IDE工具生成;

### 为什么重写equals 时必须重写 hashCode 方法（\*\*\*）?

hashCode() 的作用是获取哈希码，也称为散列码；它实际上是返回一个int 整数。这个哈希码的作用是确定该对象在哈希表中的索引位置。如果两个对象相等，则 hashcode一定也是相同的。如果两个对象相等，对两个对象分别调用equals 方法都返回 true 。但是如果两个对象有相同的 hashcode 值，它们也不一定是相等的。因此，equals 方法被覆盖过，则 hashCode 方法也必须被覆盖。

hashCode()的默认行为是对堆上的对象产生独特值。如果没有重写 hashCode()，则该 class 的两个对象无论如何都不会相等(即使这两个对象有相同的数据)。

### Object类的clone方法的作用（\*）?

Object类的clone方法是一个protected的native方法。创建并返回当前对象的一份拷贝。

### CloneNotSupportedException异常（\*）?

当调用 Object 类中的 clone 方法复制对象,但该对象的类无法实现 Cloneable 接口时,抛出该异常。

### Cloneable接口的作用（\*）?

该接口里面没有任何定义,仅仅是一个标识,实现了该接口的类,可以合法地对该类实例进行按字段复制;

### Object类的finalize方法作用（\*）?

该方法的作用是实例被垃圾回收器回收的时候触发的操作,就好比 “死前的最后一波挣扎”。

### Serializable接口的作用（\*）?

该接口里面没有任何定义,仅仅是一个标识,实现了该接口的类,可以启用其序列化功能

### 字符串和数组之间的转换?

**字符串和byte数组之间的转换:**

String(byte[] bytes) 通过使用平台的默认字符集解码指定的 byte 数组，构造一个新的 String。

byte[] getBytes() 使用平台的默认字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中。

**字符串和char数组之间的转换:**

String(char[] value) 分配一个新的 String，使其表示字符数组参数中当前包含的字符序列。

static String valueOf(char[] data) 返回 char 数组参数的字符串表示形式。

char[] toCharArray() 将此字符串转换为一个新的字符数组。

### String、StringBuffer、StringBuilder的区别（\*\*\*）?

它们的相同点是都用来表示字符串常量；都实现了CharSequence接口。

**可变与不可变:** String是不可变的字符序列,StringBuffer、StringBuilder都是可变的字符序列;**String类的底层是使用final修饰的字符数组,不可变**; 但是StringBuffer、StringBuilder底层也是使用字符数组,但是没有使用final修饰,可变;

**是否线程安全:** String和StringBuffer都是线程安全的,效率低;StringBuilder是线程不安全的,效率高; 执行效率: StringBuilder > StringBuffer > String

**初始化方式:** String可以通过字符串字面量和构造器来创建实例;但是StringBuffer、StringBuilder只能通过构造器来创建实例;

**字符串修改方式:** String字符串修改方法是首先创建一个StringBuilder，其次调用StringBuilder的append方法，最后调用StringBuilder的toString()方法把结果返回，得到一个新的字符串对象;StringBuffer和StringBuilder在修改字符串方面比String的性能要高。

**是否实现了equals和hashCode方法:** String实现了equals()方法和hashCode()方法; 而StringBuffer没有实现equals()方法和hashCode()方法;

### String类是否可以被继承（\*）?

不可以,String类是由final修饰的类

### String的replace()和StringBuffer的replace()的区别?

String的replace()可以传递两个char参数、两个字符串序列参数;StringBuffer的replace()传递的是开始下标,结束下标,要替换的字符串;

String的replace()不是修改原字符串,生产一个新的字符序列;StringBuffer的replace()修改的是原字符序列;

### String str=new String("abc");创建了几个String对象？

两个或一个。

1、如果 abc 字符串之前没有用过，这毫无疑问创建了两个对象，一个是new String 创建的一个新的对象，一个是常量“abc”对象的内容创建出的一个新的String对象；

2、如果 abc 字符串之前有用过，那么是创建一个对象。

### String代码阅读题(理解内存,万变不离其宗)?

|  |
| --- |
| String s1 = "Hello";  String s2 = "Hello";  String s3 = "Hel" + "lo";  String s4 = "Hel" + new String("lo");  String s5 = new String("Hello");  String s6 = s5.intern();  String s7 = "H";  String s8 = "ello";  String s9 = s7 + s8;    System.out.println(s1 == s2); // true  System.out.println(s1 == s3); // true  System.out.println(s1 == s4); // false  System.out.println(s1 == s9); // false  System.out.println(s4 == s5); // false  System.out.println(s1 == s6); // true |

|  |
| --- |
| public class StringTest {  String str = new String("good");  char[] ch = { 't', 'e', 's', 't' };  public void change(String str, char ch[]) {  str = "test ok";  ch[0] = 'b';  }  public static void main(String[] args) {  StringTest ex = new StringTest();  ex.change(ex.str, ex.ch);  System.out.println(ex.str);// good  System.out.println(ex.ch); // best  }  } |

|  |
| --- |
| Person p1 = new Person();  p1.name = "jimbo";  Person p2 = new Person();  p2.name = "jimbo";  System.out.println(p1.name.equals(p2.name)); // true  System.out.println(p1.name == p2.name); // true  System.out.println(p1.name == "jimbo"); // true  String s1 = new String("bcde");  String s2 = new String("bcde");  System.out.println(s1 == s2); // false |

### 字符串常量池到底存在于内存空间的哪里（\*\*\*）？

jdk 6.0 字符串常量池在方法区，方法区的具体体现可以看做是堆中的永久区。

jdk 7.0 java 虚拟机规范中不再声明方法区，字符串常量池存放在堆空间中。

jdk 8.0 java 虚拟机规范中又声明了元空间，字符串常量池存放在元空间中。

### 八种基本数据类型及其包装类（\*）?

八种基本数据类型: byte、short、int、long、float、double、boolean、char

包装类: Byte、Short、**Integer**、Long、Float、Double、Boolean、**Character**

### 什么是装箱和拆箱（\*）?

装箱: 把基本数据类型转换成包装类,分为自动装箱和手动装箱

//自动装箱

Integer i1 = 10;

//手动装箱

Integer i2 = new Integer(10);

拆箱: 把包装类转换成基本数据类型,分为自动拆箱和手动拆箱

//自动拆箱

int i3 = i2;

//手动拆箱

int i4 = i2.intValue();

### int和Integer的区别（\*）?

Integer是int的包装类，在拆箱和装箱中，两者自动转换。

int是基本数据类型，直接存储数值；Integer是对象，用一个引用指向该对象。

由于Integer是一个对象，在JVM中对象需要一定的数据结构进行描述，相对于int而言，Integer占用的内存更大一些。

### 包装类代码阅读题（\*）?

|  |
| --- |
| Object o1 = true ? new Integer(1) : new Double(2.0);  System.out.println(o1);// 1.0  Object o2;  if (true){  o2 = new Integer(1);  }else{  o2 = new Double(2.0);  }  System.out.println(o2);// 1  public class WrapperClassTest {  public static void main(String[] args) {  Integer i = new Integer(1);  Integer j = new Integer(1);  System.out.println(i == j); // false  //Integer内部定义了IntegerCache结构，IntegerCache中定义了Integer[],保存了从-128~127范围的整数。如果我们使用自动装箱的方式，给Integer赋值的范围在-128~127范围内时，可以直接使用数组中的元素，不用再去new了。目的：提高效率  Integer m = 1;  Integer n = 1;  System.out.println(m == n);// true  Integer x = 128;//相当于new了一个Integer对象  Integer y = 128;  System.out.println(x == y);// false  }  } |

### Math类的ceil()、floor()、random()、round()的作用（\*）?

ceil() 向上取整,获取大于等于参数的最小整数

floor() 向下取整,获取小于等于参数的最大整数

random() 获取0-1的伪随机数,包含0不包含1

round() 四舍五入

### SimpleDateFormat的作用?

把日期对象转换为指定格式字符串: 通过SimpleDateFormat类的format()

把指定格式字符串转换为日期对象: 通过SimpleDateFormat类的parse()

### 枚举类的理解（\*）?

类的对象只有有限个，确定的。

当需要定义一组常量时，强烈建议使用枚举类。

若枚举只有一个对象，则可以作为一种单例模式的实现方式。

### 枚举类的创建方式?

JDK1.5之前需要自定义枚举类；

JDK1.5新增的 enum 关键字用于定义枚举类；

### 枚举类的特征?

1、私有化类的构造器，保证不能在类的外部创建其对象；

2、在类的内部创建枚举类的实例。声明为：public static final；

3、对象如果有实例变量，应该声明为private final，并在构造器中初始化；

注：enum声明的枚举类和class声明的枚举类,基本类似,唯独声明枚举类对象的语法简洁;

### enum关键字的作用（\*）?

首先，enum声明实际上定义了一个类。因此我们可以通过定义的enum调用其方法。

其次，Java编译器会自动在enum类型中插入一些方法，其中就包括values()——所以我们的程序在没编译的时候，自然没法查看values()方法的源码了。

### 使用enum关键字定义的枚举类实现接口?

若每个枚举值在调用实现的接口方法呈现相同的行为方式，则只要统一实现该方法即可。

若需要每个枚举值在调用实现的接口方法呈现出不同的行为方式, 则可以让每个枚举值分别来实现该方法。

### 什么是注解?以及它的作用?

从 JDK 5.0 开始, Java 增加了对元数据(MetaData) 的支持, 也就是Annotation(注解)。

Annotation 其实就是代码里的特殊标记, 这些标记可以在编译, 类加载, 运行时被读取, 并执行相应的处理。

Annotation 可以像修饰符一样被使用, 可用于修饰包, 类, 构造器, 方法, 成员变量, 参数, 局部变量的声明, 这些信息被保存在Annotation 的 “name=value” 对中。

### JAVASE常用的注解?

@Override: 限定重写父类方法, 该注解只能用于方法；在编译期检测是否是重写的方法;

@Deprecated: 用于表示所修饰的元素(类, 方法等)已过时。通常是因为所修饰的结构危险或存在更好的选择；

@SuppressWarnings: 抑制编译器警告；

### 怎么自定义注解（\*）?

1、定义新的 Annotation 类型使用 @interface 关键字，自定义注解自动继承了java.lang.annotation.Annotation接口。

2、Annotation 的成员变量在 Annotation 定义中以无参数方法的形式来声明。其方法名和返回值定义了该成员的名字和类型。我们称为配置参数。类型只能是八种基本数据类型、String类型、Class类型、enum类型、Annotation类型、以及所有类型的数组。

3、我们自定义的注解,如果有成员变量,那么在使用的时候必须赋值；如果自定义的注解的成员变量，赋予了默认值，那么我们使用的时候可以赋值，也可以不赋值；

我们需要default来给注解的成员变量赋值

4、如果只有一个参数成员，建议使用参数名为value；

5、没有成员定义的 Annotation 称为标记; 包含成员变量的 Annotation 称为元数据 Annotation；

我们都可以称之为注解；

### 什么元注解，元注解有哪些,作用是什么（\*）?

JDK 的元 Annotation 用于修饰其他 Annotation 定义；

@Retention: 只能用于修饰一个 Annotation 定义, 用于指定该 Annotation 的生命周期, @Rentention 包含一个RetentionPolicy 类型的成员变量, 使用@Rentention 时必须为该 value 成员变量指定值:

RetentionPolicy.SOURCE:在源文件中有效（即源文件保留），编译器直接丢弃这种策略的注释；

RetentionPolicy.CLASS:在class文件中有效（即class保留），当运行 Java 程序时, JVM 不会保留注解。 这是默认值；

RetentionPolicy.RUNTIME:在运行时有效（即运行时保留），当运行 Java 程序时, JVM 会保留注释。程序可以通过反射获取该注释。

@Target: 用于修饰 Annotation 定义, 用于指定被修饰的 Annotation 能用于修饰哪些程序元素。 @Target 也包含一个名为 value 的成员变量。

@Documented: 用于指定被该元 Annotation 修饰的 Annotation 类将被javadoc 工具提取成文档。默认情况下，javadoc是不包括注解的。

@Inherited: 被它修饰的 Annotation 将具有继承性。

### 泛型的理解（\*）?

泛型,即“参数化类型”。就是将类型由原来的具体的类型参数化,类似于方法中的变量参,此时类型也定义成参数形式,然后在使用/调用时传入具体的类型;

参数化类型必须是引用数据类型;

泛型被编译后直接被Object类取代,因为根据多态特性它可以指向任何类,泛型对它都是无压力的;

### 泛型的使用（\*\*\*）?

泛型类型用于类的定义中,被称为泛型类。在实例化类的时候指明泛型的具体类型;

泛型接口与泛型类的定义及使用基本相同。泛型接口常被用在各种类的生产器中;

当一个类去实现泛型接口,要么实现接口的时候必须指明泛型类型;要么实现类也必须是泛型类;

泛型方法,是在调用方法的时候指明泛型的具体类型.泛型方法可以出现在任何地方和任何场景中使用

### 泛型类和泛型接口的注意事项（\*）?

1、泛型类可能有多个参数，此时应将多个参数一起放在尖括号内。比如：<E1,E2,E3>；

2、泛型类的构造器如下：public GenericClass(){}，而下面是错误的：

public GenericClass<E>(){}；

3、实例化后，操作原来泛型位置的结构必须与指定的泛型类型一致。

4、泛型不同的引用不能相互赋值。

5、泛型如果不指定，将被擦除，泛型对应的类型均按照Object处理，但不等价于Object。经验：泛型要使用一路都用。要不用，一路都不要用。

6、如果泛型结构是一个接口或抽象类，则不可创建泛型类的对象。

7、jdk1.7，泛型的简化操作：ArrayList<Fruit> flist = new ArrayList<>(); 类型推断

8、泛型的指定中不能使用基本数据类型，可以使用包装类替换。

9、在类/接口上声明的泛型，在本类或本接口中即代表某种类型，可以作为非静态属性的类型、非静态方法的参数类型、非静态方法的返回值类型。但在静态方法中不能使用类的泛型。

10、异常类不能是泛型的；

11、不能使用new E[]。但是可以：E[] elements = (E[])new Object[capacity];参考：ArrayList源码中声明：Object[] elementData，而非泛型参数类型数组。

12、父类有泛型，子类可以选择保留泛型也可以选择指定泛型类型：

### 在类/接口上声明的泛型，是否可以作用于非静态属性的类型、非静态方法的参数类型、非静态方法的返回值类型，是否可以作用于静态成员、静态方法（\*）?

可以作用于非静态属性的类型、非静态方法的参数类型、非静态方法的返回值类型;

不可以作用域静态成员、静态方法；原因在于：泛型的类型是在实例化对象的时候传入的;

### 静态方法可不可以使用泛型（\*）?

如果静态方法要使用泛型的话，必须将静态方法也定义成泛型方法。

### 数组和集合的区别（\*）?

集合和数组都是容器,都可以用来存储和操作数据。

一、数组声明了它容纳的元素的类型，而集合不声明；简而言之: 数组必须指明存储类型，但是集合可以指明也可以不指明其存储类型；(如果集合没有指明其存储类型Object，如果想要指明其存储类型，可以使用我们学过的泛型)

二、数组是静态的，一个数组实例具有固定的大小，一旦创建了就无法改变容量了。而集合是可以动态扩展容量，可以根据需要动态改变大小，集合提供更多的成员方法，能满足更多的需求。

三、数组可以存储基本数据类型，也可以存储引用数据类型。而集合只能存储引用类型；

四、数组是java语言中内置的数据类型,是线性排列的,执行效率或者类型检查都是最快的。

### Java集合框架的6个接口和9个类（\*\*\*）?

**6个接口:**

**Collection接口:**Collection 接口是集合层次结构中的根接口。Collection 是用来存储单列数据对象；JDK 不提供此接口的任何直接实现：它提供更具体的子接口（如 Set 和 List）实现。

**List接口:**List 接口是一个有序的集合，可以包含重复的元素，提供了按索引访问的方式。

**Set接口:**Set 接口是无序的，不能包含重复的元素。

**Iterator接口:**Collection所有的集合类，都依赖了Iterator接口，这是一个用于遍历集合中元素的接口；每一个集合类都包含了可以返回迭代器实例的迭代方法。

**ListIterator接口:**ListIterator接口是列表迭代器，允许按任一方向遍历列表、迭代期间修改列表，并获得迭代器在列表中的当前位置。

**Map接口:**Map是Java.util包中的另一个接口，它和Collection接口没有关系，是相互独立的，但是都属于集合类的一部分。Map包含了key-value对，Map不能包含重复的key，但是可以包含相同的value。

**9个实现类:**

**ArrayList类:**ArrayList是一个动态数组，也是我们最常用的集合。它允许任何符合规则的元素插入甚至包括null。随着容器中的元素不断增加，容器的大小也会随着增加。在每次向容器中增加元素的同时都会进行容量检查，当快溢出时，就会进行扩容操作。ArrayList擅长于随机访问。同时ArrayList是非同步的。

**LinkedList类:**LinkedList是一个双向链表。LinkedList不能随机访问，它所有的操作都是要按照双重链表的需要执行。在列表中索引的操作将从开头或结尾遍历列表（从靠近指定索引的一端，节约一半时间）。这样做的好处就是可以通过较低的代价在List中进行插入和删除操作。LinkedList也是非同步的。

**Vector类:**与ArrayList相似，但是Vector是同步的。所以说Vector是线程安全的动态数组。它的操作与ArrayList几乎一样。

**HashSet类:**HashSet堪称查询速度最快的集合，因为其内部是以HashCode来实现的。集合元素可以是null,但只能放入一个null。它内部元素的顺序是由哈希码来决定的（根据哈希码,结合某种算法,得到存储元素的下标），所以它不保证set的迭代顺序；特别是它不保证该顺序恒久不变。

**LinkedHashSet类:**LinkedHashSet集合同样是根据元素的hashCode值来决定元素的存储位置，但是它同时使用链表维护元素的次序。LinkedHashSet在迭代访问Set中的全部元素时，性能比HashSet好，但是插入时性能稍微逊色于HashSet。

**TreeSet类:**TreeSet 是二叉树实现的，基于TreeMap，生成一个总是处于排序状态的set，内部以TreeMap来实现，不允许放入null值。它是使用元素的自然顺序对元素进行排序，或者根据创建Set时提供的 Comparator 进行排序，具体取决于使用的构造方法。TreeSet存储的元素是有序的。

**HashMap类:**以哈希表数据结构实现，查找对象时通过哈希函数计算其位置，它是为快速查询而设计的，其内部定义了一个hash表数组（Entry[] table/Node[] table），元素会通过哈希转换函数将元素的哈希地址转换成数组中存放的索引，如果有冲突，则使用散列链表的形式将所有相同哈希地址的元素串起来，可能通过查看HashMap.Entry的源码它是一个单链表结构。

**LinkedHashMap类:**LinkedHashMap和HashMap存储类似，只是在HashMap的基础上引用了链表维护元素的次序。

**TreeMap类:**有序散列表，实现SortedMap接口，底层通过红黑树实现。

### 遍历集合的时候能否增删元素（\*\*\*）？

1、foreach 遍历，iterator 遍历都不能在遍历的过程中使用 list.remove 或 list.add 操作，会报并发修改异常，遍历删除后加个 break 即可解决。

2、iterator 遍历过程中如果需要删除可以使用 iterator 提供的remove()方法。

3、遍历根据元素索引删除是可行的。

### ArrayList、LinkedList、Vector的区别（\*\*\*）?

ArrayList、LinkedList、Vector都是List接口的实现类;

ArrayList底层是基于数组的,通过Object[] elementData来存储元素;ArrayList是对象引用的一个”变长”数组(自动扩容)，它是用来存储有序（按照添加的先后顺序排列），可以重复的元素；

对于频繁的遍历/获取或修改元素的操作，建议使用ArrayList类，效率高。

LinkedList底层是基于双向链表，存储的每个元素都在链表中作为一个节点存在，每个节点维护了一个prev和next指针。这个链表，维护了first和last指针，first指向第一个元素，last指向最后一个元素。

对于频繁的插入或删除元素的操作，建议使用LinkedList类，效率较高。

Vector 是一个古老的集合，JDK1.0就有了。大多数操作与ArrayList相同，区别之处在于Vector是线程同步的，安全但是效率低。

ArrayList和LinkedList都不是线程同步的，不安全，效率高。

我们在使用List集合的时候，ArrayList是我们要优先考虑的；但是当插入、删除频繁时，使用LinkedList；

如果我们考虑线程安全问题，那么也不适用Vector，而是使用Collections.synchronizedList(List<T> list) 来让ArrayList成为线程安全的即可 或 线程安全的集合类（java.util.concurrent）；

ArrayList和Vector都会自动扩容，但是LinkedList不存在扩容；

如果集合中的元素的数目大于目前集合数组的长度时，Vector增长率为目前数组长度的100%，而Arraylist增长率为目前数组长度的50%。

**Vector和ArrayList的区别?**

Vector和ArrayList底层都是基于数组；

1、Vector是线程同步的，所以它也是线程安全的，而Arraylist是线程异步的，是不安全的。如果不考虑到线程的安全因素，一般用Arraylist效率比较高。

2、如果集合中的元素的数目大于目前集合数组的长度时，Vector增长率为目前数组长度的100%，而Arraylist增长率为目前数组长度的50%。如果在集合中使用数据量比较大的数据，用vector有一定的优势。

**ArrayList和LinkedList的区别?**

1.ArrayList是实现了基于动态数组的数据结构，LinkedList基于双向链表的数据结构。

2.对于随机访问get和set，ArrayList觉得优于LinkedList，因为ArrayList要移动指针。

3.对于插入和删除操作add和remove，LinkedList比较占优势，因为ArrayList要移动数据。

### ArrayList的JDK1.8之前与之后的实现区别?

JDK1.7：当我们通过无参构造器实例化ArrayList对象的时候,创建一个初始容量为10的数组。

JDK1.8：当我们通过无参构造器实例化ArrayList对象的时候,一开始创建一个长度为0的数组，当添加第一个元素时再创建一个始容量为10的数组。

### 怎么理解Set集合是无序的，不可以重复的（\*\*\*）?

无序: Set底层是数组来存储元素的，向Set存储元素，并不是按照下标顺序来存储元素的，而是通过元素的HashCode值，结合某种算法，得到存储元素的下标；

不可重复：向Set集合存储元素，会先判断hashCode得到的哈希码是否相等，如果相等再通过equals()比较是否相等；如果equals()也是true,那么就不会向集合中添加；

### HashSet的底层实现是什么（\*\*\*）?

HashSet的实现依赖于HashMap，HashSet的值都是存储在HashMap中。

在HashSet的构造器中会初始化一个HashMap对象，HashSet不允许值重复，因此，HashSet的值是作为HashMap的key存储在HashMap中的，当存储的值已经存在时返回false。

### HashSet、LinkedHashSet和TreeSet的区别（\*\*\*）?

HashSet是基于HashMap来实现的，HashMap底层是数组Node<K,V>[] table，更像是对HashMap做了一次“封装”，而且只使用了HashMap的key来实现各种特性，而HashMap的value始终都是PRESENT。HashSet 按 Hash 算法来存储集合中的元素，因此具有很好的存取、查找、删除性能。

LinkedHashSet 是 HashSet 的子类。LinkedHashSet 根据元素的 hashCode 值来决定元素的存储位置，但它同时使用双向链表维护元素的次序，这使得元素看起来是以插入顺序保存的。简单来说：在HashSet的基础上,引入了链表来维护元素的次序；

TreeSet底层使用红黑树结构存储数据，TreeSet 可以确保集合元素处于排序状态。

### HashSet集合判断两个元素相等的标准?

两个对象通过 hashCode()得到的哈希值比较相等，并且两个对象的 equals() 方法返回值也相等。

注意：对于存放在Set容器中的对象，对应的类一定要重写equals()和hashCode(Object obj)方法，以实现对象相等规则。即：“相等的对象必须具有相等的散列码”。

### Comparable 和 Comparator的异同（\*\*\*）?

Comparable和Comparator都是用于比较对象的大小的；

Comparable位于java.lang，我们称之为自然排序；Comparator位于java.util，我们称之为定制排序；

实现Comparable接口需要重写compareTo方法；实现Comparator接口需要重写compare方法；

这两个方法的返回值都是int，用int类型的值来确定比较结果；

如果返回值是正整数,那么表示当前对象大于参数对象;

如果返回值是0,那么表示当前对象等于参数对象;

如果返回值是负整数,那么表示当前对象小于参数对象;

Comparable 定义在类的内部，我们称之为内部比较器；Comparator 是定义在类的外部，我们称之为外部比较器；

在设计初时有需求就选择Comparable，若后期需要扩展或增加排序需求是，再增加一个比较器Comparator；

### HashMap和Hashtable的异同（\*\*\*）?

HashMap和Hashtable都实现了Map接口，都是用来存储双列数据的；

HashMap的key和value，都可以存储null；Hashtable的key和value，都不可以存储null；

HashMap是非线程同步（非线程安全）的，效率高；Hashtable是线程同步（线程安全）的，效率低；

### HashMap、Hashtable、TreeMap、LinkedHashMap的区别（\*\*\*）?

HashMap、Hashtable、TreeMap、LinkedHashMap都实现了Map接口，都是用来存储双列数据的，key不可以重复，value允许重复；

**HashMap**是 Map 接口使用频率最高的实现类。底层是Node<K,V>[]数组来存储数据；**允许使用null键和null值，不保证映射的顺序；线程非安全，效率高**；扩容机制：旧容量 \* 2；

**Hashtable是个古老的 Map 实现类**，JDK1.0就提供了。它和HashMap非常类似；底层是Entry<?,?>[]数组来存储数据；不允许使用null键和null值，不保证映射的顺序；**线程安全，效率低**；

**LinkedHashMap 是 HashMap 的子类。**在HashMap存储结构的基础上，使用了一对**双向链表来记录添加元素的顺序**。它是由内部类 Entry<K,V> extends HashMap.Node<K,V>来记录元素的添加顺序；

**TreeMap存储 Key-Value 对时，需要根据 key-value 对进行排序。**TreeMap 可以保证所有的 Key-Value 对处于有序状态。TreeMap底层使用红黑树结构存储数据，实现了SortMap接口。能够把它保存的记录根据键排序,默认是按键值的升序排序，也可以指定排序的比较器，当用 Iterator 遍历TreeMap 时，得到的记录是排过序的。

### Map集合的containsKey和containsValue的判断标准?

HashMap 判断两个 key 相等的标准是：hashCode 值相等，两个 key 通过 equals() 方法返回 true。

HashMap 判断两个 value相等的标准是：两个 value 通过 equals() 方法返回 true。

### ArrayList底层实现原理（\*\*\*）?

ArrayList底层是一个动态自增的数组，该类封装了一个动态再分配的Object[] elementData数组；

**（1）当调用无参空构造器实例化ArrayList的时候；**

JDK1.7：创建一个初始容量为10的数组。

JDK1.8：一开始创建一个长度为0的数组，当添加第一个元素时再创建一个始容量为10的数组。

当我们调用有参数构造器实例化ArrayList的时候，都是创建指定长度的数组；

**（2）ArrayList 允许空值和重复元素，当往 ArrayList 中添加的元素数量大于其底层数组容量时，其会通过扩容机制重新生成一个更大的数组。**

**（3）什么时候扩容**

当数组存储满的时候,再向其添加元素,才会扩容;

**（4）扩容机制**

ArrayList里面的grow()就是扩容的具体实现;

1、默认的计算方式: 新容量 = 旧容量 \* 1.5,

2、如果新的容量不满足最小容量,把最小容量作为新的容量；

3、如果新的容量大于最大容量,新的容量Integer.MAX\_VALUE;

4、通过Arrays.copyOf(elementData, newCapacity)进行扩容；

（5）ArrayList 是非线程安全类，并发环境下，多个线程同时操作 ArrayList，会引发不可预知的异常或错误。

（6）顺序添加很方便，删除和插入需要复制数组，性能差（可以使用 LinkindList）

### LinkedList底层实现原理（\*\*\*）?

LinkedList底层是基于双向链表实现的。它也可以被当作堆栈、队列或双端队列进行操作。LinkedList 是非同步的。双向链表每个结点除了数据域之外，还有一个前指针和后指针，分别指向前驱结点和后继结点（如果有前驱/后继的话）。另外，双向链表还有一个 first 指针，指向头节点，和 last 指针，指向尾节点。

LinkedList是基于双向链表的,不存在扩容问题；

### transient关键字的作用（\*）?

将不需要序列化的属性前添加关键字transient,序列化对象的时候,这个属性就不会被序列化。

### 数组复制的相关方法（\*\*\*）?

Arrays类 copyOf(U[] original, int newLength, 类<? extends T[]> newType) 复制指定的数组,用空值截断或填充（如有必要）,以便复制具有指定的长度。

original - 要复制的数组

newLength - 要返回的副本的长度

newType - 要返回的副本的类

注:生成一个新的数组

System类 arraycopy(Object src, int srcPos, Object dest, int destPos, int length) 将指定源数组中的数组从指定位置复制到目标数组的指定位置。

src - 源数组。

srcPos - 源数组中的起始位置。

dest - 目标数组。

destPos - 目的地数据中的起始位置。

length - 要复制的数组元素的数量。

集合框架的底层扩容机制经常调用该方法;

### ArrayList中removeAll(collection c)和clear()的区别(\*)?

removeAll可以删除批量指定的元素,而clear是全是删除集合中的元素。

### HashMap底层原理(非常非常...重要\*\*\*)?

HashMap基于Map接口实现，元素以键值对的方式存储，并且允许使用null 键和null 值，因为key不允许重复，因此只能有一个键为null,另外HashMap不能保证放入元素的顺序，它是无序的，和放入的顺序并不能相同。HashMap是线程不安全的。

影响 HashMap 性能的两个重要参数：“initial capacity”（初始化容量）和”load factor“（负载因子）。简单来说，容量就是哈希表桶的个数，负载因子就是键值对个数与哈希表长度的一个比值，当比值超过负载因子之后，HashMap 就会进行 rehash操作来进行扩容。

#### JDK 7及以前版本：HashMap是数组+链表结构(即为链地址法)。

当实例化一个HashMap时，系统会创建一个长度为Capacity的Entry数组，这个长度在哈希表中被称为容量(Capacity)，在这个数组中可以存放元素的位置我们称之为“桶”(bucket)，每个bucket都有自己的索引，系统可以根据索引快速的查找bucket中的元素。

每个bucket中存储一个元素，即一个Entry对象，但每一个Entry对象可以带一个引用变量，用于指向下一个元素，因此，在一个桶中，就有可能生成一个Entry链。而且新添加的元素作为链表的head。

添加元素的过程:

向HashMap中添加entry1(key，value)，需要首先计算entry1中key的哈希值(根据key所在类的hashCode()计算得到)，此哈希值经过处理以后，得到在底层Entry[]数组中要存储的位置i。

如果位置i上没有元素，则entry1直接添加成功。 **- 情况一**

如果位置i上已经存在entry2(或还有链表存在的entry3，entry4)，则需要通过循环的方法，依次比较entry1中key和其他的entry的key的hash值。

如果彼此hash值都不同，则直接添加成功，entry1指向原有的entry元素。 **- 情况二**

如果hash值相同，继续比较二者是否equals。

如果遍历一遍以后，发现所有的equals返回都为false,则entry1仍可添加成功。entry1指向原有的entry元素。 **- 情况三**

如果返回值为true，则使用entry1的value去替换equals为true的entry的value。

**HashMap什么时候进行扩容呢?**

当HashMap中的元素个数(size)超过数组大小\*loadFactor时，就会进行数组扩容。

HashMap的扩容机制:

当HashMap中的元素越来越多的时候，hash冲突的几率也就越来越高，因为数组的长度是固定的。所以为了提高查询的效率，就要对HashMap的数组进行扩容，而在HashMap数组扩容之后，最消耗性能的点就出现了：原数组中的数据必须重新计算其在新数组中的位置，并放进去，这就是resize。

扩容为原有长度的两倍；

### Hashtable底层原理(源码分析)?

Hashtable我们可以认为是线程安全的HashMap，HashTable 是一个线程安全的哈希表，它通过使用synchronized 关键字来对方法进行加锁，从而保证了线程安全。

底层数据结构就是数组+链表；

但这也导致了在单线程环境中效率低下等问题。

Hashtable 与 HashMap 不同，它不允许插入 null 值和 null 键。

### Collection 和Collections 的区别（\*\*\*）？

Collection 是集合类的上级接口，继承与他的接口主要有Set 和List。

Collections 是针对集合类的一个工具类，他提供一系列静态方法实现对各种集合的搜索、排序、线程安全化等操作。

### 用哪两种方式来实现集合的排序（\*）？

1. 使用有序集合，如 TreeSet 或 TreeMap。
2. 使用有顺序的的集合，如 list，然后通过 Collections.sort() 来排序。

### 正则表达式的作用?

正则表达式(Regular Expression)是一种文本模式，包括普通字符（例如，a 到 z 之间的字母）和特殊字符（称为"元字符"）。

正则表达式使用单个字符串来描述、匹配一系列匹配某个句法规则的字符串。

匹配: 测试字符串内的模式,是否符合某个规则;

提取: 基于模式匹配从字符串中提取子字符串;

替换: 替换字符串的指定文本;

### 正则表达式组成部分?

字符类: 从指定的字符范围中取一个

[abc] 只能是a b c中的任意一个,也就是说[]里面的内容只能选择任意一个

[^abc] 任何一个字符,除了 a、b 或 c（否定）

[a-z] a 到 z 中的任意一个

[^a-z] 非a 到 z 中的任意一个

[a-zA-Z] a 到 z 或 A 到 Z,两头的字母包括在内（范围）

[0-9] 0-9之间的任意一个

[\u4e00-\u9fa5] 表示汉字的取值范围

元字符: 正则表达式中有特殊意义的字符

\d 匹配数字,等价于[0-9]

\D 匹配任意非数字的字符,等价于[^0-9]

\w 匹配字母或数字或下划线,等价于[0-9a-zA-Z\_]

\W 匹配任意不是字母或数字或下划线,等价于[^0-9a-zA-Z\_]

\s 匹配任意的空白符,等价于空格

\S 匹配任意不是空白符的字符

. 匹配除换行符以外的任意单个字符

^ 表示匹配行首的文本

$ 表示匹配行尾的文本

限定符: 限定字符串的长度,n代表的是一个正整数

\* 重复零次或更多次,可有可无

+ 重复一次或更多次,至少一次

? 重复零次或一次,最多一次

{n} 重复n次

{n,} 重复n次或更多次

{n,m} 重复n到m次

条件:

(?![0-9]+$) 不能全是数字

(?![a-zA-Z]+$) 不能全是字母

其他:

[] 字符串用中括号括起来，表示匹配其中的任一字符

[^] 匹配除中括号以内的内容

\ 转义符

| 或者，选择两者中的一个。注意 \| 将左右两边分为两部分，而不管左右两边有多长多乱

() 从两个直接量中选择一个,分组

### 集合排序的两种方式?

java.util.Collections.sort(List<T> list)

要求所排序的元素类必须实现java.lang.Comparable接口。

java.util.Collections.sort(List<T> list, Comparator<? super T> c)

要求实现一个java.util.Comparator接口。

### 什么是数据结构?

数据结构是以某种形式将数据组织在一起的集合,它不仅存储数据,还支持访问和处理数据的操作。

### 什么是线性表（\*）?

线性表,全名为线性存储结构。

使用线性表存储数据的方式可以这样理解,即“把所有数据用一根线儿串起来,再存储到物理空间中”。

线性表存储结构可细分为顺序存储结构和链式存储结构:

将数据依次存储在连续的整块物理空间中,这种存储结构称为顺序存储结构(数组);

数据分散的存储在物理空间中,通过一根线保存着它们之间的逻辑关系,这种存储结构称为链式存储结构(链表);

### 栈和队列（\*）?

栈是后进先出的表,栈是限制插入和删除只能在一个栈顶进行的表;对栈的基本操作有push(进栈)和pop(出栈),前者相当于插入,后者相当于删除最后一个元素;

队列是先进先出的表,队列只允许在对头进行删除操作,而在队尾进行插入操作;数据元素进队列的过程称为 "入队",出队列的过程称为 "出队"。

### 数据结构中的栈有六个元素6，5，4，3，2，1 的顺序进栈，问下列哪一个不是合法的出栈序列（\*）？（ ）

A. 5 4 3 6 1 2 B. 4 5 3 1 2 6 C. 3 4 6 5 2 1 D. 2 3 4 1 5 6

答案是：C

6 5 4 3入栈 3出，4出，此时不可能6先出，只能5先出

其余：

A 6入 5入 5出 4入 4出 3入 3出 6出 2入 1入 1出 2出

B 6入 5入 4入 4出 5出 3入 3出 2入 1入 1出 2出 6出

D 6入 5入 4入 3入 2入 2出 3出 4出 1入 1出 5出 6出

### 树和二叉树的区别?

二叉树每个节点最多有2个子节点,树则无限制。

二叉树中节点的子树分为左子树和右子树,即使某节点只有一棵子树,也要指明该子树是左子树还是右子树,即二叉树是有序的。

树决不能为空,它至少有一个节点;而一棵二叉树可以是空的。

### 二叉查找(排序)树的性质（\*）?

二叉查找树就是二叉排序树，也叫二叉搜索树。

二叉查找树或者是一棵空树，或者是具有下列性质的二叉树：

(1) 若左子树不空，则左子树上所有结点的值均小于它的根结点的值；

(2) 若右子树不空，则右子树上所有结点的值均大于它的根结点的值；

(3) 左、右子树也分别为二叉查找树；

(4) 没有键值相等的结点。

### 平衡二叉树的性质?

平衡二叉树肯定是一个二叉查找树,它的左子树和右子树都是平衡二叉树,且左子树和右子树的深度之差的绝对值不超过1。

### 红黑树的性质（\*）?

红黑树又称红-黑二叉树，它首先是一颗二叉树，它具体二叉树所有的特性。同时红黑树更是一颗自平衡的排序二叉树。

红黑树顾名思义就是节点是红色或者黑色的平衡二叉树，它通过颜色的约束来维持着二叉树的平衡。

对于一棵有效的红黑树二叉树而言我们必须增加如下规则：

1、每个节点都只能是红色或者黑色。

2、根节点是黑色。

3、每个叶节点（NIL节点，空节点）是黑色的，加入NIL，就是为了让每一个路径都差不多长度相等，进而达到平衡的效果。

4、如果一个结点是红的，则它两个子节点都是黑的。也就是说在一条路径上不能出现相邻的两个红色结点。

5、从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

树 -> 二叉树 -> 二叉查找树 -> 平衡二叉树 -> 红黑树；

### 什么是IO流以及IO流的分类（\*\*\*）?

IO流是一种数据的流从源头流到目的地。比如文件拷贝，输入流和输出流都包括了。输入流从文件中读取数据存储到进程(process)中，输出流从进程中读取数据然后写入到目标文件。

**根据数据流向不同分为: 输入流、输出流;**

此输入、输出是相对于我们写的代码程序而言;

输入流: 从别的地方(本地文件，网络上的资源等)获取资源 输入到 我们的程序中; (读取)

输出流: 从我们的程序中 输出到 别的地方(本地文件),将一个字符串保存到本地文件中,就需要使用输出流（写入）。

**按处理数据单位不同分为: 字节流、字符流;**

字节流: 每次读取(写出)一个字节，当传输的资源文件有中文时，就会出现乱码;

字符流: 每次读取(写出)一个字符，有中文时，使用该流就可以正确传输显示中文。

后缀是Stream是字节流,而后缀是Reader,Writer是字符流。

**按功能不同分为: 节点流、处理流;**

节点流: 直接与数据源相连,读入或读出。直接使用节点流,读写不方便,为了更快的读写文件,才有了处理流。

处理流: 是对一个已存在的节点流的连接和封装,通过所封装的流的功能调用实现数据读写。与节点流一块使用,在节点流的基础上,再套接一层,套接在节点流上的就是处理流。

### 字节流和字符流的区别（\*）?

读写单位不同: 字节流以字节（8bit）为单位,字符流以字符为单位,根据码表映射字符,一次可能读多个字节。

处理对象不同: 字节流能处理所有类型的数据（如图片、avi等）,而字符流只能处理字符类型的数据。

### 缓冲流读和写的原理（\*）?

读取数据时，一次性尽可能多的读取字节，并缓存起来，这样实际上就是从缓冲区中读取数据，所以读取效率高。当缓冲区内容都读取完毕后，该流会再次尽可能多的读取字节缓存起来，等待再次被读取。

向流中写入字节时，不会直接写到文件，先写到缓冲区中直到缓冲区写满，才会把缓冲区中的数据一次性写到文件里。使用方法flush()可以强制将缓冲区的内容全部写入输出流。

缓冲区就是一段特殊的内存区域，很多情况下当程序需要频繁地操作一个资源（如文件或数据库）则性能会很低，所以为了提升性能就可以将一部分数据暂时读写到缓存区，以后直接从此区域中读写数据即可，这样就可以显著的提升性能。

对于 Java 字符流的操作都是在缓冲区操作的，所以如果我们想在字符流操作中主动将缓冲区刷新到文件则可以使用 flush() 方法操作。

### 什么是缓冲区？有什么作用（\*\*\*）？

缓冲区就是一段特殊的内存区域，很多情况下当程序需要频繁地操作一个资源（如文件或数据库）则性能会很低，所以为了提升性能就可以将一部分数据暂时读写到缓存区，以后直接从此区域中读写数据即可，这样就可以显著的提升性能。

对于 Java 字符流的操作都是在缓冲区操作的，所以如果我们想在字符流操作中主动将缓冲区刷新到文件则可以使用 flush() 方法操作。

### Java对象序列化的前提（\*）?

1、对象对应的类需要实现接口：Serializable

2、对象对应的类提供一个全局常量：serialVersionUID

3、除了当前对象对应的类需要实现Serializable接口之外，还必须保证其内部所有属性也必须是可序列化的。（默认情况下，基本数据类型可序列化）

### 同一对象序列化多次,会将这个对象序列化多次吗（\*）?

不会;

### 编译期和运行期（\*）?

编译期是指把源码交给编译器编译成计算机可以执行的文件的过程。在Java中也就是把Java源代码编成class文件的过程。编译期只是做了一些翻译功能，并没有把代码放在内存中运行起来，而只是把代码当成文本进行操作，比如检查错误。

运行期是把编译后的文件交给计算机执行,直到程序运行结束。所谓运行期就把在磁盘中的代码放到内存中执行起来。

### Lambda表达式的作用?

Lambda 表达式是 JDK8 的一个新特性,可以取代大部分的匿名内部类,写出更优雅的 Java 代码,尤其在集合的遍历和其他集合操作中,可以极大地优化代码结构。

JDK 也提供了大量的内置函数式接口供我们使用,使得 Lambda 表达式的运用更加方便、高效。

Lambda表达式其实完成了实现接口并且实现接口里的方法这一功能,也可以认为Lambda表达式代表一种动作,我们可以直接把这种特殊的动作进行传递。

### 为什么要使用Lambda?

可以取代大部分的匿名内部类,写出更优雅的 Java 代码。

在集合的遍历和其他集合操作中,可以极大地优化代码结构。

### Lambda表达式的基本语法（\*）?

三部分组成: 参数列表、箭头符号、Lambda体

参数列表其实函数式接口抽象方法定义的参数个数

Lambda体以前在抽象方法要实现的代码

核心用法:

只保留参数及括号和方法体,其他都可以省略;在参数和方法体之间添加->

上联：左右遇一括号省

下联：左侧推断类型省

横批：能省则省

### 什么是函数式接口以及常用的函数式接口（\*）?

函数式接口: 接口中只有一个抽象方法的接口,称为函数式接口。

JDK提供了大量常用的函数式接口,它们主要在 java.util.function 包中被提供。

常用四大函数式接口:

Consumer<T>: 消费型接口,消费一个数据,其数据类型由泛型决定。(抽象方法只有一个参数没有返回值)

void accept(T t): 意为消费一个指定泛型的数据。

Supplier<T>: 供给型接口,生产一个数据,其数据类型由泛型决定。(抽象方法没有参数有一个返回值)

T get()：用来获取一个泛型参数指定类型的对象数据。

Function<T, R>: 函数型接口，接受一个输入参数，返回一个结果。(抽象方法有一个参数有一个返回值)

R apply(T t): 将此函数应用于给定的参数。

Predicate<T>: 断言式接口,也就是给一个参数T,返回boolean类型的结果。跟Function类似,Predicate的具体实现也是根据传入的lambda表达式来决定的。

boolean test(T t)：接受一个参数返回一个布尔类型

### 什么是Stream（\*）?

Stream 使用一种类似用 SQL 语句从数据库查询数据的直观方式来提供一种对 Java 集合运算和表达的高阶抽象。

将要处理的元素集合看作一种流,流在管道中传输,并且可以在管道的节点上进行处理,比如筛选,排序,聚合等。元素流在管道中经过中间操作的处理,最后由最终操作得到前面处理的结果。

Stream的操作分为三个步骤：

1、创建Stream

2、中间操作（对数据进行操作）

3、终止操作（如果没有终止操作，中间操作是不执行的）

### Stream的三大特性（\*\*\*）?

1、Stream不存储数据: 对于Stream的聚合、消费或收集操作只能进行一次,再次操作会报错;

2、Stream不改变源数据: 对Stream的操作,仅仅只是对于流里面的数据进行处理和操作,不会影响数据源;

3、Stream的延迟执行特性: 最终操作执行之前,中间操作都不会执行;

### Stream的创建方式?

1、通过数组创建

Arrays.stream(数组)

Stream.of(泛型可变参数)

2、通过集合创建流

集合名.stream() 创建普通流,普通流是单线程,按照顺序来执行

集合名.parallelStream() 创建并行流,并行流是多线程,异步执行

3、创建空流

Stream.empty()

4、创建无限流和有限流

Stream.generate().limit(数值)

5、创建规律的无限流

Stream.iterate()

### List 去重的实现方式（\*\*\*）？

**方法 1：contains 判断去重(有序)：**

要进行数据去重，我们首先想到的是新建一个集合，然后循环原来的集合，每次循环判断原集合中的循环项，如果当前循环的数据，没有在新集合中存在就插入，已经存在了就舍弃，这样当循环执行完，我们就得到了一个没有重复元素的集合了。

此方法的优点的：理解起来比较简单，并且最终得到的集合也是有序的，这里的有序指的是新集合的排列顺序和原集合的顺序是一致的；但缺点是实现代码有点多，不够简洁优雅。

**方法 2：HashSet 去重(无序)**

我们知道 HashSet 天生具备“去重”的特性，那我们只需要将 List 集合转换成 HashSet集合就可以了。此方法的实现代码较为简洁，但缺点是 HashSet 会自动排序，这样新集合的数据排序就和原集合不一致了，如果对集合的顺序有要求，那么此方法也不能满足当前需求。

**方法 3：LinkedHashSet 去重(有序)**

既然 HashSet 会自动排序不能满足需求，那就使用 LinkedHashSet，它既能去重又能保证集合的顺序，LinkedHashSet 是到目前为止，实现比较简单，且最终生成的新集合与原集合顺序保持一致的实现方法，是我们可以考虑使用的一种去重方法。

**方法 4：TreeSet 去重(无序)**

除了以上的 Set 集合之外，我们还可以使用 TreeSet 集合来实现去重功能，比较遗憾的是，TreeSet 虽然实现起来也比较简单，但它有着和 HashSet 一样的问题，会自动排序，因此也不能满足我们的需求。

**方法 5：Stream 去重(有序)**

JDK 8 为我们带来了一个非常实用的方法 Stream，使用它可以实现很多功能，包括我们的去重，只需要使用 distinct 方法即可去重。

### jdk8之后对于接口的改变（\*\*\*）?

Jdk1.8之前的接口中通常只有方法的声明,没有方法体;接口中可以实现具体的方法体,只需要加上关键字static或者default修饰即可。

使用static来修饰的称之为静态方法,静态方法通过接口名来调用;

使用default来修饰的称之为默认方法,默认方法通过对象实例来调用;

接口中的静态方法不能被重写,但是默认方法可以重写,也可以不重写;

### jdk8时间和日期?

Java 8 吸收了 Joda-Time 的精华，以一个新的开始为 Java 创建优秀的 API。 新的 java.time包 中包含了所有关于本地日期（LocalDate）、本地时间 （LocalTime）、本地日期时间（LocalDateTime）、时区（ZonedDateTime） 和持续时间（Duration）的类。历史悠久的 Date 类新增了 toInstant() 方法，用于把 Date 转换成新的表示形式。这些新增的本地化时间日期 API 大大简化了日期时间和本地化的管理。

新时间日期API常用的包：

java.time 包含值对象的基础包；

java.time.chrono 提供对不同的日历系统的访问；

java.time.format 格式化和解析时间和日期；

java.time.temporal 包括底层框架和扩展特性；

java.time.zone 包含时区支持的类；

说明：大多数开发者只会用到基础包和format包，也可能会用到temporal包。因此，尽管有68个新的公开类型，大多数开发者，大概将只会用到其中的三分之一。

### 说出 5 个 JDK 1.8 引入的新特性（\*\*\*）？

1. Lambda表达式，允许像对象一样传递匿名函数。
2. 函数式接口，接口中只有一个抽象方法的接口。
3. Stream API，充分利用现代多核 CPU，可以写出很简洁的代码。
4. Date 与 Time API，最终，有一个稳定、简单的日期和时间库可供你使用。
5. 扩展方法，现在，接口中可以有静态、默认方法。
6. 重复注解，现在你可以将相同的注解在同一类型上使用多次。