**Java知识点**

1.常量与变量

1.1常量

常量：程序中固定不变的量，一旦被定义，它的值就不能被修改。

声明常量：在定义常量时，需要对常量进行初始化。初始化后，常量的值不能被改变。（为了区分常量与变量，常量名称通常都用大写字母）

声明常量的语法：final 数据类型 变量名称 *=* 值

（final表示表示这个变量只能被赋值一次。一旦被赋值以后，就不能再修改。它不仅可以用来修饰基本数据类型的变量，还可以用来修饰对象的引用和方法）

Java有2种数据类型，分别是基本数据类型和引用诗句数据类型。

8种基本数据类型：byte、short、int、long、float、double、char、boolean。

3种引用数据类型：数组、类、接口。

例子可见test1

1.2变量

变量：程序运行中，数据的值会发生改变。

声明变量：所有变量必须在使用前进行声明，声明多个变量时，必须用逗号隔开，声明变量的基本格式：int x，y，z或者int x=10，y=5，z=7等。

（声明一个变量时，编译程序会在内存中开辟一块足以容纳这个变量的内存空间给它。不管变量的值如何改变，都永远使用相同的内存空间）

变量分类：按作用范围分类，分为成员变量、局部变量、类变量（静态变量）。

成员变量声明在类中，但在方法、构造方法和语句块之外。作用范围为整个类。当一个对象被实例化之后，成员变量就随之确定，成员变量在对象创建的时候创建，在对象销毁的时候销毁。成员变量有默认值，数值型为0，布尔型为false，引用型为null。

局部变量声明在方法、构造方法和和语句块中，当方法、构造方法和语句块被执行时创建，当他们执行完毕之后，变量将会被销毁。访问修饰符不能用于局部变量。局部变量没有默认值，在声明之后，需要进行初始化。

静态变量在类中以static关键字声明，必须在方法、构造方法和语句块之外。无论一个类中创建了多少个对象，类只拥有静态变量的一份拷贝。访问访问格式一般为：对象名.静态变量名。

静态变量的默认值和成员变量的默认值一样，变量的值可以在声明时指定，也可以在构造方法中指定，还可以在静态语句块中指定。

静态变量和成员变量的区别：

（1）静态变量在内存中只有一份拷贝，JVM只为静态变量分配一次内存，在加载类的过程中完成静态变量的内存分配，可以用类名直接调用。

（2）成员变量是每创建一个对象，就会为成员变量分配一次内存，成员变量可以在内存中拥有多个拷贝，互不影响。

2.函数与运算符

1.1函数

进行开平方运算：Math.sqrt(x)

进行幂运算：Math.pow(x,y) x代表底数，y代表指数 输出结果为double类型

常用的三角函数：Math.sin Math.cos Math.tan Math.atan

反函数：Math.exp

自然对数：Math.log

以10为底的对数：Math.log10

π：Math.PI

e：Math.E

1.2数值类型

不会有精度丢失的转换：byte（1字节）🡪short（2字节）🡪int（4字节）🡪long（8字节） char（1字节）🡪int（4字节） int（4字节）🡪double（8字节）

可能会有精度丢失的的转换：int（4字节）🡪float（8字节） long（8字节）🡪 float（8字节） long（8字节）🡪double（8字节）

1.3位运算符

https://blog.csdn.net/weixin\_41050155/article/details/79725555

&：按位与（两个数补码同为1，则为1，反之为0）

|：按位或（两个数补码全为0，则为0，反之为1）

~：按位非（遇0则为1，遇1则为0）

^：按位异或（两个数的相同位为0，反之为1）

3.数组

形式：int []a=new int[5]或int a[]=new int[5]

创建数值类型的数组时，所有元素的值被初始化为0；创建一个boolean类型的数组时，所有元素的值被初始化为false；创建一个对象类型的数组时，所有元素的值被初始化为null（如String类型的数组）

数组的长度，创建之后不能被修改（允许为0）

初始化匿名数组：new int[]{1,2,3}

数组变量能够拷贝给另一个数组变量。拷贝之后，两个数组变量指向同一个数组的地址。如：int []a=b; a[5]=12;//b[5]=12

4.类与对象

结构化编程：算法+数据结构=程序，算法在第一位，数据结构是第二位，首先确定如何操作数据，然后确定如何组织数据。适用于规模较小的问题；

面向对象编程：由对象组成，数据在第一位，算法在第二位。适用于规模较大的问题。

类：构造对象的模板和蓝图。

类的实例：由类构造对象的过程。使用构造器构造而成。

构造器：用来构造并初始化对象。

实例域：对象中的数据。（每个特定的类实例，即对象，都有一个特定的实例域）

方法：操纵数据的过程。

封装：给对象赋予了“黑盒”特性，提高了代码的重用性和可靠性。关键在于不能让类中的方法直接访问到其它类中的实例域，而是通过对象中的数据和方法来访问。

类之间的关系：依赖（uses-a）、聚合（has-a）、继承（is-a）。

依赖：一个类的方法操纵另一个类的对象。尽量减少类的相互依赖，从而降低耦合性。

聚合：一个类的对象包含其它类的对象。如：depart类中含有Employee e属性。详见<https://blog.csdn.net/weixin_43101144/article/details/84757851>

继承：一个类为另一个类的扩展。

对象变量并没有包含一个对象，而仅仅是引用一个对象；对象变量必须经过初始化，初始化即为类的实例。

例如：Date date=new Date(); =右边为类的实例，即构造了一个Date类型的对象，new操作符的返回值为新创建对象的引用；=左边为创建一个Date类型的对象变量。=将引用存储在对象变量中。

推论：在为两个对象赋值时，直接使用=赋值，其实只是把一个对象变量引用的对象赋给另一个对象变量，实际为一个对象，两个引用对象：

Date date1=new Date(); Date date2=date1;

date1和date2实际上为一个对象的两个引用。将一个对象赋给两个对象，一般使用clone方法。

https://blog.csdn.net/dc\_space/article/details/81186617

日历：标准Java类库中包含两个类：

1.显示格林威治时间的Date类（原理：通过当前时间减去1970年1月1日 00:00:00秒得到毫秒数，构造器调用currentTimeMills方法，显示格式：Mon Feb 10 16:21:43 CST 2020）；

2.表示日历表示法的LocalDate类（不能使用构造器创建对象，而是使用静态工厂方法代表调用构造器创造对象，也可以使用of方法来构造一个特定日期的对象，之后使用getYear、getMonthValue、getDayOfMonth方法获得年、月、日）

更改器方法：访问对象时，会修改对象的实例域，即对象的数据

访问器方法：访问对象时，不修改对象。如plusDays方法

栈：

1.描述的是方法执行的内存模型。每个方法被调用都会创建一个栈帧（存储局部变量、操作数、方法出口等）；

2.JVM为每一个线程创建一个栈，用于存放该线程执行方法的信息（实际参数、局部变量）

3.栈属于线程私有，不能实现线程间的共享；

4.栈的存储特性“先进后出，后进先出”；

5.栈是由系统自动分配的，速度快，栈是一个连续的内存空间。

堆：

1.堆用于存储创建好的对象和数组；

2.JVM只有一个堆，被所有线程共享；

3.堆是一个不连续的内存空间，分配灵活，速度慢；

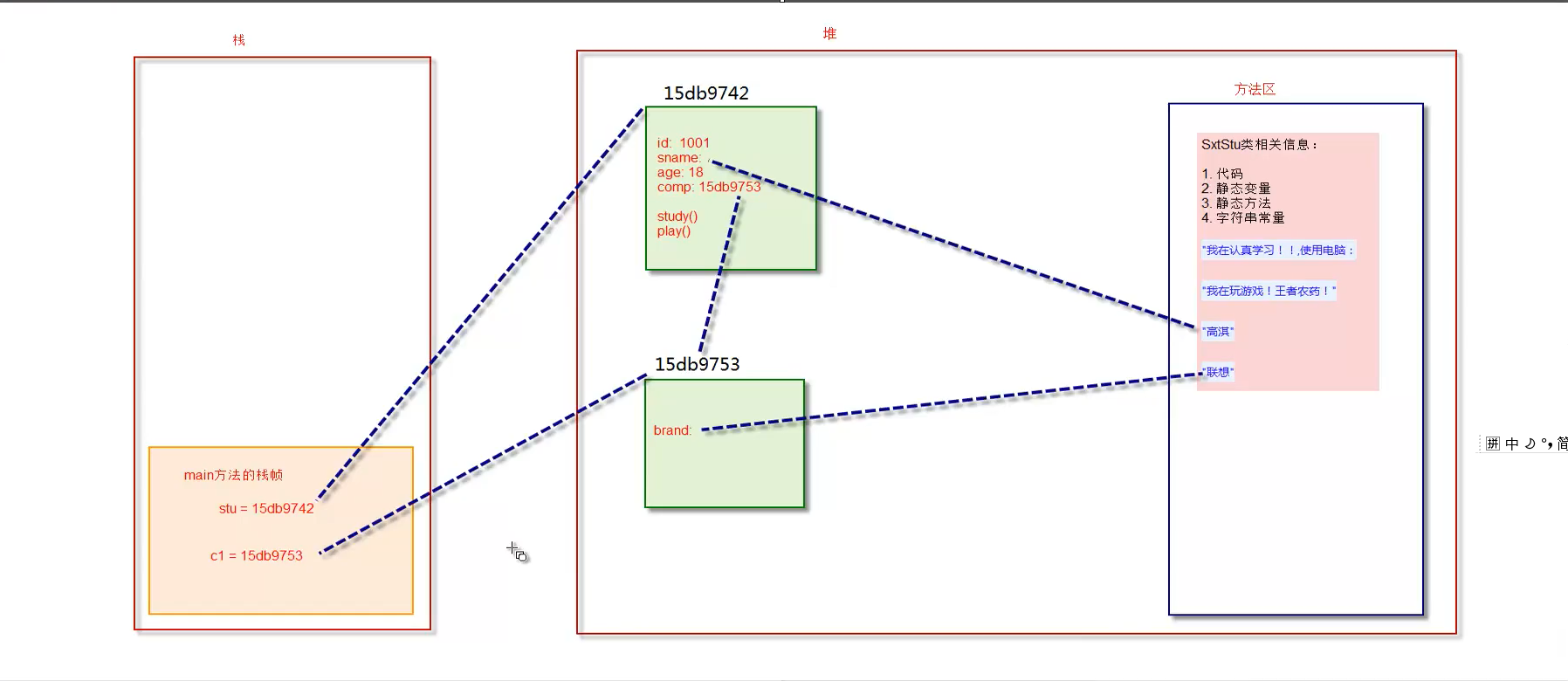
方法区：

1.JVM只有一个方法区，被所有线程共享；

2.方法区实际上也是堆，只是用于存储类、常量的信息；

3.用来存放程序中永远是不变或唯一的内容。（类对象，静态变量、字符串常量等）

内存分析



在一个源文件中，只能有一个公有类，但能有任意数量的非公有类。

程序执行的过程：将程序中包含main方法的类名提供给字节码解释器，启动程序：java 类名 编译程序：javac 类名.java

构造器与类同名，总是随着new操作符的执行而调用，而不能对已经存在的对象调用构造器来达到重新设置实例域的目的。所有的java对象都是在堆中构造的。不能在构造器中定义与实例域重名的局部变量（局部变量会屏蔽同名的实例域）。

不能编写返回引用可变对象的访问器方法，如Date（通过setTime方法，可以绕过对象，从而直接修改实例域，解决方法：返回clone可变数据域的拷贝），原因：详见Java核心技术卷一110页。

一个方法可以访问所属类的所有对象的私有数据。

final实例域：

一般使用在基本数据类型和不可变类型（类中的每个方法都不会改变其对象）。

只有get方法，没有set方法。

其他数据类型可以直接使用对象.属性名的方法或对象.set方法的方式修改实例域的数据，final不能修改。

静态域与静态方法：

静态域，即将实例域定义为static。每创建一个实例都有一份所有实例域的拷贝，而静态实例域在类中只会初始化一次（加载类的时候），之后被所有实例域共享。可以使用对象调用静态域，但默认使用类名调用静态域，又称类变量。

静态常量：类中定义的静态常量，只能通过类名调用，不能被重新赋值，如：System.out、Math.PI等。

静态方法：没有this参数（隐式的参数）的方法，可以访问自身的静态域。在静态set方法中不能使用this.静态域=静态域，而是使用类名.静态域=静态域。默认使用类名.静态方法名调用静态方法。

需要使用静态方法的两种情形：

1.不需要访问对象状态（即对象中的数据等），所需参数都是显示参数提供（即不含this）；

2.只需要访问类的静态域。

方法参数：

一个方法不能修改一个基本数据类型的参数（数值型和布尔型）。原因：方法得到的是参数值的拷贝。

一个方法可以改变一个对象参数的状态。原因：对象引用及其他拷贝引用的是同一个对象。

一个方法不能让对象参数引用一个新的对象。原因：交换的只是对象引用的拷贝，而不是实际对象。

引用传递和值传递：基本数据都为值传递（传递的为参数值），除此之外，都是引用传递（传递的为地址）。

重载：多个方法具有相同的名字，不同的参数（参数类型和参数数量）。

重载解析：

1.编译器查看对象的声明类型和方法名；

2.编译器查看调用方法时提供的参数类型。

方法的签名：完整的描述一个方法，指出方法名和参数类型，即方法名(参数类型,…)，如getSalary(int,int)。不能有两个名字相同，参数类型也相同，但返回不同参数类型的方法。

构造器：当类没有提供任何构造器时，系统会提供一个默认无参构造器，将实例域初始化为默认值。当类提供至少一个的构造器，但没有提供无参构造器，此时，构造对象时必须提供参数。

在类调用构造器时，应保证无论调用哪个构造器，每个实例域都可以被设置为一个有意义的处值。即在定义实例域时，直接进行赋值操作（不一定赋常量，还可以调用方法进行初始化），如：private String name=””（原因：String类型的数据默认值为null）

当类调用构造器时，如果第一行形如this(…)，该构造器调用另一个类的构造器，如果为super，该构造器调用父类的构造器（调用构造器的类型以super方法中的参数为准）。先调用父类构造器，再调用子类构造器。

this关键字：调用当前对象，有多少个对象就调用多少个this对象指向各个对象。

1.当局部变量和成员变量重名时，可以使用this对象指定调用成员变量。

2.通过this调用构造方法。

初始化数据域的方法：

1.在构造器中设置值；

2.在声明中赋值；

3.在初始化块中赋值。

只要构造类的对象，代码块（又称对象初始化块）都会被调用，首先运行初始化块，然后再调用构造器的主体部分（先执行父类构造器，再执行初始化块，最后执行子类构造器）。通常直接把初始化块放在构造器里。

Java中有自动的垃圾回收器，因此不支持析构器。finalize方法在垃圾回收器清除对象之前调用。

finalize方法，一个对象只能执行一次，只能在第一次进入被回收的队列，而且对象所属于的类重写了finalize方法才会被执行。第二次进入回收队列的时候，不会再执行其finalize方法，而是直接被二次标记，在下一次GC的时候被GC。

Java 技术允许使用 finalize() 方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在确定这个对象没有被引用时对这个对象调用的。它是在 Object 类中定义的，因此所有的类都继承了它。子类覆盖 finalize()方法以整理系统资源或者执行其他清理工作。注意：finalize不一定被jvm调用，只有当垃圾回收器要清除垃圾时才被调用。

使用情形：某些对象使用了内存之外的资源（如文件、数据库连接等），或者调用非Java方法（native方法）时分配的内存。

在实际应用中，不能用finalize方法回收任何短缺的资源，一般提供close方法供客户端手动调用。原因：由于GC的自动回收机制，很难知道这个方法什么时候调用以及是否调用。

https://blog.csdn.net/crazylai1996/article/details/84900818

四种访问控制符

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | pubic | protected | default | private |
| 同包同类 | ✓ | ✓ | ✓ | ✓ |
| 同包不同类 | ✓ | ✓ | ✓ |  |
| 同包继承 | ✓ | ✓ | ✓ |  |
| 不同包继承 | ✓ | ✓ |  |  |
| 不同包不继承 | ✓ |  |  |  |

5.继承

this的两个用途：引用隐式参数；调用该类其它的构造器。

super的两个用途：调用超类方法；调用超类构造器。

构造函数既可以传递给本类的其它构造器，也可以传递给超类的构造器。

出现父类对象的任何地方都可以用子类对象调换。父类对象引用变量既可以引用一个父类对象，也可以引用任何一个子类对象。不能反过来把父类引用赋给子类对象。

静态绑定：private方法、static方法、final方法或者构造器，编译器可以准确的知道调用哪个方法。

原因：子类不能重写此类方法。

动态绑定：调用方法依赖于隐式参数的实际类型，并且在运行时实现动态绑定。优先调用当前类中的该方法，若没有该方法，则调用父类中的该方法，还没找到，报错。

为了降低搜索的时间开销，虚拟机预先为每个类创建方法表，方法表中为方法的签名。

在覆盖（重写）一个方法的时候，子类方法的可见性不能低于父类方法的可见性。如果父类方法为public，子类也必须为public。

final类：使用final关键字修饰的类，不能被扩展。

final方法：使用final关键字修饰的方法，在子类继承时，不能重写（覆盖）此方法。final类中的方法自动被声明为final方法，而不包括域。

final域：声明为final的域，构造对象之后就不能改变其值。只能读取，不能修改。变量的引用（指向的地址）不可变。

将方法和类声明为final原因：确保它们不会在子类中改变语义，确保引用的为该类的对象。

类型转换：将一种数据类型强制转换为另一种数据类型。（尽量少用）

编译器允许直接将父类对象赋给子类引用，将子类对象赋给父类引用需要进行强制类型转换。

只能在继承层次内进行类型转换，在将超类转化为子类之前，应使用instanceof方法进行检查。父类引用 instanceof 子类：false 子类引用 instanceof 父类：true

包括一个或多个抽象方法的类必须声明为抽象类。

抽象方法：使用abstract方法修饰，不需要实现。

抽象类中可以定义实例域、普通方法、抽象方法、构造方法。

抽象类不一定有抽象方法，但有抽象方法的类必定是抽象类。

抽象类不能被实例化，可以定义一个抽象类的对象变量，引用非抽象子类的对象。使用该对象变量调用方法，实质是调用子类中具体的实现方法。

Object类是Java中所有类的始祖，在Java中每个类都是由它继承而来的，但并不需要使用extend关键字继承。

如果没有明确指定超类，Object就被认为是这个类的超类。

Object可以引用任何类型的对象。

Object方法中的equals方法用于检测一个对象是否等于另一个对象。实质是判断两个对象是否具有相同的引用。

自反性、传递性、一致性、对称性。

Employee other = (Employee) obj;//将Object类型的对象转换成为Employee类型的对象。

Objects.equals(type[]a, type[]b)：如果a和b的数组长度相同，并且在对应位置的数组元素也均相同，返回true。

hash code：散列码，由对象导出的整型值，没有任何规律。

相同的对象必须具有相同的散列码，如果两个对象的hashcode相同，它们并不一定相同。

哈希算法也称散列算法，是将数据依特定算法直接指定到一个物理地址上。

集合中保证元素唯一性的实现原理：通过哈希算法将元素定位到它存放的物理位置，当要添加新元素时，通过哈希算法确定它应该存放的位置，如果该位置没有元素，直接存储；如果该位置有元素，使用equals方法比较，相同的话，不存储，不相同的话，散列到其他位置。

<https://blog.csdn.net/zj15527620802/article/details/88547914>

泛型数组列表：ArrayList

ArrayList实现List、Cloneable、RandomAccess、Serializable接口。

Serializable：序列化接口。序列化：即将对象通过ObjectOutputStream转化为字节流；反序列化：字节流通过ObjectInputStrean将字节流转化为对象。serialVersionUID：JVM允许进行序列化与反序列的标识。常用在网络传输或从一个应用程序域传递到另一个应用程序域。

Cloneable：标记接口，只有实现这个接口的类，才能使用clone类(kao)

优点：允许在运行时确定数组的大小，原理：ArrayList在添加或删除数组元素时，具有自动调节数组容量的功能。

ArrayList是一个使用类型参数的泛型类。

ArrayList可以在声明时直接指定大小，ArrayList<Employee> list = new ArrayList<>(10); Ps：这只是表示数组列表有保存100个元素的潜力。

一旦确定数组列表的内容不会继续增加，可以调用trimToSize方法，该方法将会把数据列表的容量变为当前元素的所需要的存储空间数目，垃圾回收器将会回收多余的存储空间。（Ps：由于确定数组列表的元素之后，再次添加元素需要移动存储块，所以确定不会添加任何元素之后，再调用该方法）

size方法返回数组列表中的实际元素数目。

使用get或set方法访问或改变数据列表中的内容，比数组使用[]访问复杂。

add方法为数组列表添加元素，set方法只能改变数组列表中已有元素的内容。

没有泛型类时，原始的ArrayList类使用get方法获取数组元素内容时，返回的是Object类型的数据。

可以在存数据时，使用ArrayList存储，避免数组容量不足，访问数据时，将ArrayList使用Arrays.toArray方法转化为数组，使用[]下标访问。

可以使用add方法向指定位置添加元素，形如add(n，m);可以通过remove元素删除已经插入的元素，形如：remove(n)

如果在进行类型转化时，存在警告信息，并且确保不会造成严重后果，可以使用@SuppressWarnings(unchecked)来标记这个变量能接受类型转换

对象包装类：每个基本类型所对应的类（int：Integer；String：String；char：Character；float：Float；double：Double；long：Long；byte：Byte；boolean：Boolean）

对象包装器不可变，一旦构造了包装器，就不允许更改包装器中的值，同时，对象包装器是使用final修饰的类，不能被继承。

自动装箱：Java编译器自动将基本数据类型转换为对象。如：Integer i = 100;=>实质相当于Integer I = Integer.valueOf(100)

自动拆箱：Java编译器自动将对象转换为基本数据类型。如：int n = i;=>实质相当于int n = i.intValue()

判断两个包装类对象的值是否相等，要使用equals方法，而不能使用==方法，原因：==方法还检测对象是否位于同一内存区域。

缓存[-128,127]之间，实际就是系统初始的时候，创建了[-128,127]之间的缓存数组。

当我们调用valueOf的时候，首先检查是否在这个范围内，如果在则直接从缓存数组中拿出已经存在的数据，如果不在，则创建新的Integer对象

toString方法：默认返回该数的十进制表示形式，还可以指定进制，如：toString(10,2)：将10以2进制的形式输出。

parseInt方法：默认将字符串以十进制形式返回，也可以指定进制，类似toString方法。

Enum：枚举类型

类中具有有限且固定的常量值。

枚举类中可以添加构造方法和域，构造方法私有，不支持public和protected修饰构造方法，因此不能创建类的枚举实例。

所有枚举类型都是Enum类的子类。

toString方法：将枚举类的常量以字符串的形式输出。

valueOf方法：toString方法的逆应用，将字符串赋给为枚举常量。

values方法：将枚举类型的变量以数组的形式输出。

original方法：获取枚举类声明中枚举常量的位置。

equals方法：比较两个枚举常量是否一致，底层实现为==。

compareTo方法：比较两个枚举常量的位置关系，返回位置之差（a在b前，返回负值；a=b，返回0；a在b之后，返回正值）。底层实现：先获取枚举类的全限定名，之后比较是否一致，若一致，获取枚举常量的下标，相减。

类的加载机制：

当程序要使用某个类时，则系统会通过加载、连接、初始化三步来实现对这个类进行初始化。

加载：将class文件加载入内存，并为之创建一个Class对象，任何类被使用时，系统都会建立一个Class对象。

连接：验证是否有正确的协调结构，并与其它类协调一致；负责为默认静态

类的初始化时机：

创建类的实例；

访问类的静态变量，或者为静态变量赋值；

调用类的静态方法；

使用反射方式来强制创建某个类或接口对应的java.lang.Class对象；

初始化某个类的主类；

直接使用java.exe命令来运行某个主类。

类加载器：负责将.class文件加载到内存中，并为之生成对应的Class对象。

类加载器的组成：

BootStrap ClassLoader：根类加载器（也被称之为引导类加载器，负责Java核心类的加载，在JDK中JRE的lib目录下的rt.jar中）；

Extension ClassLoader：扩展类加载器（负责JRE的扩展目录中jar包的加载，在JDK中JRE的lib目录下的ext目录）；

System ClassLoader：系统类加载器（负责在JVM启动时，加载来自Java命令的class文件，以及classpath环境变量所指定jar包和类路径）

反射机制：在运行状态中，对于任意一个类，都能知道这个类的所有属性和方法；对于任意一个对象，都能调用它的任意一个方法；这种动态获取的信息以及动态调用对象的方法称为Java语言的反射机制。（通过Class字节码文件对象，去使用该文件的成员变量、成员方法、构造方法）实质是将Java类中的各种成分映射成一个个的Java对象。

反射机制的功能：在运行时分析类的能力；在运行时查看类的对象；实现通用的数组操作代码。

原来创建对象：Person p = new Person(); p.成员方法（成员变量）

通过反射创建对象：Class.forName(String className);(开发常用)

void printStackTrace()：将Throwable对象和栈的轨迹输出到标准错误流

Class getDeclaringClass方法：返冋一个用于描述类中定义的构造器、方法或域的Class对象。

Class[] getExcepti onTypes方法：(在Constructor和Method类中）返回一个用于描述方法抛出的异常类型的Class对象数组。

int getModifier方法：返回一个用于描述构造器、方法或域的修饰符的整型数值。使用 Modifier 类中的这个方法可以分析这个返回值。缺省：0；public：1；private：2；protected：4

String getName方法：返冋一个用于描述构造器、方法或域名的字符串。

Class[] getParameterTypes方法：(在Constructor和Method类中）返回一个用于描述参数类型的 Class 对象数组。

Class getReturnType方法：(在 Method类中）返回一个用于描述返回类型的 Class 对象。

接口中的所有方法自动属于public，域内自动设为public static final。因此在接口中声明方法时，不必加public访问修饰符。

接口不是类，不能使用new操作符实例化一个接口，但可以进行声明；

接口变量必须引用已经实现了接口的类对象。（Spring中的依赖注入）

可以使用instanceof方法检查一个对象是否属于某个特定类或实现某个接口。

接口可以被扩展，即先定义一个通用性高的接口，之后再定义一个较高专用性的接口，之后实现接口。

接口中不能包含实例域和静态方法，但可以包含常量，父接口中的常量可以直接在子接口中使用。

每个类只能继承一个超类，但可以实现多个接口，使用，隔开，如：

**public** **class** ArrayList **extends** AbstractList **implements** List, RandomAccess, Cloneable, Serializable

为什么已有抽象类的情况下，还要引入接口的概念？

每个类只能扩展（extends）一个类，不能同时扩展多个类，而每个类都能实现多个接口。接口可以提供多重继承的大多数好处，同时还能避免多重继承的复杂性和低效性。

JDK1.8之后，允许在接口中定义静态方法，但有违接口作为抽象规范的初衷，因此通常的做法是把静态方法放到伴随类中，如Collection/Collections，Path/Paths

默认方法：即接口的默认实现，只能在接口中定义，以default修饰。如果实现该接口的类没有重写此方法，不会报错，按默认方法的结果输出；如果实现该接口的类重写此方法，按重写之后的结果输出。

优点：可以将接口中的所有方法设置为默认方法，但在使用时，只需要重写用到的方法；为已经写好的接口以及实现类增加默认方法，不会破坏原有代码结构。

如果先在一个接口中将一个方法定义为默认方法，然后又在超类及另一个接口中定义了同样的方法，在同时继承该接口和扩展该超类的类中会发生什么情况？

1.超类优先被调用方法，接口中的默认方法被忽略；

2.接口冲突。如果实现的接口中有两个默认方法的方法签名相同，则需要覆盖该方法以解决该冲突。具体做法：在该类中定义一个同名的方法（非默认），之后返回接口名.super.方法（仅会实现该接口的方法）。

浅拷贝与深拷贝：

= 只是把该对象变量引用的实例地址赋给另一个对象变量

浅拷贝的特点：

浅拷贝会对基本数据类型的值进行拷贝，引用数据类型拷贝内存地址；因此，一个对象中基本数据类型实例域的值发生改变，不会影响另一个，而引用类型实例域由于拷贝的是内存地址，一个对象中的值发生改变时，另一个对象也随之改变。

如果引用类型不可变，则向拷贝对象（原元对象）重新赋值之后，会再次开辟一个空间，来存储新的内容。

如果引用类型可变，则重新赋值之后，两个对象的值都会被改变，原因：引用类型在浅拷贝时，拷贝的是内存地址。

浅拷贝的实现：实现Cloneable接口，并重写clone方法

示例：

@Override

**protected** Object clone() **throws** CloneNotSupportedException {

**return** **super**.clone();

}

深拷贝的特点（相较于浅拷贝）：

深拷贝对引用类型的实例域进行拷贝时，会新建一个对象空间，然后拷贝其中的内容，因此它们指向不同的内存空间，改变其中一个，另一个不会受到影响。

深拷贝需要在可变引用实例域所属的类上实现Cloneable接口，并重写clone方法（如实例域中聚合Dept类和对象，需要Dept类实现接口并重写方法）。

深拷贝相较于浅拷贝的速度较慢，而且花销较大。

示例：

@Override

**public** Object clone() **throws** CloneNotSupportedException { **return** **super**.clone();

}

@Override

**public** Employee clone() **throws** CloneNotSupportedException {

Employee e = (Employee) **super**.clone();

e.dept = (Dept) dept.clone();

**return** e;

}

内部类：定义在外部类内部的类。

使用内部类的原因：

1.可以将内部类声明为私有类，以封装类中的具体实现；（主要）

2.可以通过多个内部类继承不同的方法来间接实现外部类的多继承；

3.使用匿名内部类实现回调功能；

4.解决超类和接口中的方法同名的问题。

内部类分为：成员内部类、局部内部类、匿名内部类、静态内部类。

内部类既可以访问自身的数据域，也可以访问创建它的外部类对象的数据域。内部类的对象总有一个隐式引用，指向创建它的外部域的对象，该引用在内部类的定义中是不可见的。

原理：外部类的引用在内部类中的构造器设置。编译器修改了所有内部类的构造器，添加一个外部类引用的参数。当内部类没有定义构造器时，系统为这个类生成默认构造器。在构造器中将内部类的对象赋给外部类的对象，之后使用外部类调用外部类的实例域。

外部类只能使用缺省或public修饰，内部类还可以使用private、protected修饰。

在内部类中使用外部类的实例域时，可以直接使用，也可以使用类名.this.实例域的方式使用。如：Student.this.name;

调用内部类构造器时，可以使用外部类对象.new 内部类构造器或者new 外部类构造器.new 内部类构造器，如：dept.new Employee();或 new Dept().new Employee();

在外部类的作用域之外，引用内部类的方法：外部类.内部类，如：Dept.Employee employee = dept.new Employee();或 Dept.Employee employee = new Dept().new Employee();

当成员内部类以private修饰时，可以使用get内部类名创建一个方法，返回内部类的实例。如：

**public** Test4 getTest4(){

**return** **new** Test4();

}

局部内部类：定义在一个方法体或属性中的内部类，主要是作用域发生了变化，只能在自身所在方法或属性中使用。

为什么局部内部类访问局部变量必须加final修饰符？

因为局部变量随着方法的调用而调用，使用完毕就消失，而堆内存中的数据并不会消失，所以堆内存还是使用该变量，但该变量已经没有了，为了让该值还存在，就加final修饰符，原因是：当我们使用final修饰变量后，堆内存存储的是值，而不是变量名。

局部内部类在调用时，需要先创建外部类的实例，之后通过实例调用内部域外围的方法（该方法中需要创建内部类的实例，并调用内部类中的方法）。

静态内部类：使用static关键字修饰的内部类，特点：可以不通过外部类的对象，直接使用外部类名调用。（不需要内部类引用外部类对象，不能引用外部类对象）

静态内部类可以有静态域和方法，声明在接口中的内部类自动成为static和public类。

匿名内部类：没有类名、访问修饰符、构造器的内部类。（更加简洁）

定义匿名内部类的前提是，内部类必须是继承一个类或者实现接口，格式为new 父类(){子类内容}.方法；匿名内部类最终提供给我们的是一个匿名子类的对象。

所在方法的形参想被匿名内部类使用时，必须声明为final。

**new** Test7(name){

**public** **void** getName(){

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

}

}.getName();

内部类中声明的所有静态域都必须是final static。原因：一个静态域只有一个实例，而每个外部类对象，都分别有一个单独的内部类实例，如果这个类不是final，它可能就不是唯一的。？

动态代理：利用反射机制在运行时创建代理类，方便运行时动态处理代理方法的调用机制。（为其他对象提供一种代理以控制对这个对象的访问）

通过代理，可以让调用者与实现者之间解耦。

常见的动态代理有：JDK动态代理，Cglib（基于ASM）等。

JDK动态代理基于Java的反射机制实现，主要涉及到java.lang.reflect包中的Proxy和InvocationHandler。（必须有接口，否则不能实现）

实现步骤：

1.实现InvocationHandler接口；

2.重写invoke方法；

@Override

public Object invoke(Object obj, Method method, Object[] aobj) throws Throwable {

Object result = method.invoke(target, aobj);

return result;

}

3.创建该类对象，并使用构造器初始化实例；

TransactionHandler handler = new TransactionHandler(ud);

4.创建代理对象；

UserDao proxy = (UserDao) Proxy.newProxyInstance(ud.getClass().getClassLoader(), ud.getClass().getInterfaces(), handler);

第一个参数：类加载器；

第二个参数：Class对象数组，每个元素都是要实现的接口。

第三个参数：调用处理器。

5.通过代理对象调用接口中的方法。

代理类是在程序运行过程中创建的，一旦被创建，就变成了常规类。所有的代理类都扩展自Proxy类，一个代理类只有一个实例域——调用处理器，它定义在Proxy的超类中。所需要的任何数据都存储在调用处理器中（即包装了实际对象）。

对于特定的类加载器和接口来说，只能有一个代理类。使用同一个类加载器和接口数组调用两次newProxyInstance方法时，只能得到同一个类的两个对象。

异常处理是将错误产生的地方转移到能够处理这种情况的错误处理器。

所有异常都是由Throwable继承而来，往下分解为Error和Exception。

Error类层次结构描述了Java运行时系统的内部错误和资源耗尽错误，应用程序不应该抛出（程序无法处理的错误，表示运行程序中较严重的问题）。

Exception层次结构又分解为两个分支：一个派生于RuntimeException；另一个分支是其它异常。划分两个分支的规则是：由程序错误导致的异常属于RuntimeException；而程序本身没有错误，但由于I/O错误等异常导致的问题属于其它异常。

派生于RuntimeException的异常包含的情况：错误的类型转换、数组访问越界、访问null指针。

不是派生于RuntimeException的异常包括：试图在文件尾部后面读取数据、试图打开一个不存在的文件、根据给定的字符串查找Class对象，而该字符串表示的类不存在。

Java语言规范将派生于Error类和RuntimeException类的所有异常称为非受查异常（unchecked，编译器要求必须处理），所有其它的异常称为受查异常（checked，编译器不要求强制处理的异常）。

四种情况下抛出异常：调用一个抛出受查异常的方法、程序运行过程中发现错误，并且利用throw语句抛出一个受查错误、程序出现错误、Java虚拟机和运行时库出现的内部错误。

自定义异常的步骤：创建一个类，扩展已知异常类，并创建构造器；在方法声明时抛出自定义异常类（throws）；抛出自定义异常类对象（throw）。

堆栈轨迹（stack trace）是一个方法调用过程的列表，它包含了程序执行过程中方法调用的特定位置。

调用Throwable类的printStackTrace方法访问堆栈轨迹的文本描述信息。

Throwable类有带参构造方法和无参构造方法，带参构造方法可以将cause作为封装到Throwable类的实例中，cause指的是由底层传递而来的异常。可以理解为使用给定的原因，给定的错误构造一个Throwable对象。

Logger.getGlobal()：全局日志记录器

Logger.getGlobal().info(“”)：生成简单的日志记录

Logger.getGlobal().setLevel(Level.OFF)：取消所有的日志

getLogger()：创建或获取记录器。

未被任何变量引用的日志记录可能会被垃圾回收。

日志与包名类似，都具有层次结构（更强），日志级别可以从父类记录器继承到子类记录器。

通常共有7个级别的日志记录器：SEVERE、WARNING、INFO、CONFIG、FINE、FINER、FINEST

logger.setLevel(Level.ALL)：开启所有级别的日志记录；

logger.setLevel(Level)：关闭所有级别的日志记录

一个泛型类就是具有一个或多个类型变量的类。

类型变量使用大写，E表示集合的元素类型，K和V分别表示表的关键字和值的类型，T表示任意类型。

泛型类可以看作普通类的工厂。

只能在泛型类中定义泛型实例域。

虚拟机没有泛型类型对象，所有对象都属于普通类。

无论何时定义一个泛型类型，都自动提供一个相应的原始类型，原始类型的名字是删去类型参数后的泛型类型名。擦除类型变量，并替换为限定类型（无限定的变量用Object），即进行类型替换。

原始类型用第一个限定的类型变量替换，如果没有给定限定，使用Object替换。

桥方法：通过调用子类中的同名参数类型不同的方法进行中转，之后调用泛型类中的该方法

为保持类型安全性，必要时插入强制类型转换。

不能使用基本数据类型作为泛型（使用包装类）

运行时的类型查询只适合于原始类型，不适合于泛型类，如getClass方法、instanceof方法，原因：类型擦除

Java不支持泛型类型的数组，不能创建参数化类型数组，但可以去声明泛型类的变量，解决方法：使用ArrayList存储。

禁止使用带有类型变量的静态域和方法。

不能抛出或捕获泛型类的实例

泛型类之间没有继承关系，比如：Employee是Manager的父类，但Test<Employee>和Test<Manager>之间并没有关系

通配符设置：<Employee>和<? extends Employee>，后者表示的是继承自Employee的所有子类，可以为方法提供返回值，但不能为方法提供参数；相反的：<? super Manager>，表示Manager类的所有超类型，可以为方法提供参数，但不能为方法提供返回值

集合：

队列：Queue接口，可以使用循环数组实现和使用链表实现。

两者优缺点：

循环数组比链表更加高效，但由于循环数组容量有限，所以在不确定程序中所搜集的元素数量时，最好使用链表来实现队列。

集合中的基本接口为Collection接口和Map，Collection接口下的子接口为Queue、List、Set

Iterator迭代器包含4个方法：

next()：返回集合中的元素，通过反复调用此方法，可以逐个访问集合中的每个元素。

hasNext()：判断集合中是否含有数据元素。

remove()：删除上次调用next方法返回的元素。（必须先执行next方法，不然报错）

迭代器位于集合中两个元素之间，当调用next方法时，迭代器就越过下一个元素，并返回刚刚越过的那个元素。

迭代器是描述集合中位置的，依赖于位置的add方法使用迭代器完成。对自然有序的集合添加元素才有意义。

使用iterator迭代器的方式：

Iterator<与集合中的泛型保持一致> it = 集合对象.iterator();

子接口ListIterator，下面有add、previous、hasprevious方法；

previous、hasprevious方法：用于反向遍历链表

add方法：可以向集合内添加元素（Collections.add方法只能在集合末尾添加元素）

add方法依赖于迭代器的位置，remove方法依赖于迭代器的状态。

由数组支持的有序集合可以快速地随机访问，即使用索引访问，由链表支持的集合随机访问很慢，推荐使用迭代器遍历。

RandomAccess：标记接口，实现该接口的类支持高效的随机访问。

Set方法：不允许存储重复值，实现的原理：equals方法和hashCode方法

实现类由：HashSet、TreeSet、EnumSet。

HashSet：没有重复元素的无序集合

TreeSet：有序集

LinkedHashSet：可以记住元素插入次序的集，是LinkedHashSet的子类

链表（LinkedList）是一个有序的集合。

使用链表的原因：尽可能减少在链表中间插入或删除数据所付出的代价。如果集合中元素较少，采用ArrayList。

使用动态数组时，可以使用Vector，原因：线程安全，同步，Vector中的每个方法都加了锁；使用ArrayList取代Vector的原因：如果由两个线程访问，用Vector安全，但如果只有一个线程访问，代码要在同步操作上花费很长时间。因此，不需要线程同步时，用ArrayList，需要时，用Vector。

散列表（hastable）使用链表数组实现，每个列表被称为桶，保存在桶中的元素索引：散列码/桶总数的余数

Java SE 8之后，桶满之后，会从链表变成平衡二叉树。

通常情况下，桶的个数为元素个数的75%-150%，标准类库中桶的个数为2的幂，默认值为16。

如果不知道元素个数，而导致散列表太满，就需要对散列表进行再散列，也可以通过装载因子（负载因子）来决定何时对散列表进行再散列。

对散列表进行再散列：创建一个桶数更多的表，并将所有元素都插入到新表中，之后舍弃旧表。

装填因子默认为0.75，即散列表中75%的位置填满元素之后，该表会使用双倍的桶数进行再散列。

HashSet实现了基于散列表的集，不关心集合中元素的顺序时，使用HashSet。

TreeSet是一个有序集合，可以任意顺序插入到集合中，在对集合中的元素进行遍历时，每个值自动按排序后的顺序输出。

排序使用树结构来完成，底层为红黑树。将一个元素添加到树中比添加到散列表中慢，但比起查找数组或链表中的重复项快的多。从JDK1.6开始，TreeSet实现了NavigableTree接口。

队列的初始容量为16。

优先级队列没有对所有的元素进行排序，却总会获得当前优先级队列中最小的元素。

优先级队列使用堆来实现。堆是一个可以自我调整的二叉树，对树进行添加、删除时，可以将最小的元素移动到根，而不必进行排序。

典型示例：任务调度。

映射的通用实现：HashMap、TreeMap，这两个类都实现了Map接口。

要求同步，使用HashTable；对同步性没有要求，使用HashMap。

属性映射：properties

键与值都是字符串；

表可以保存在一个文件中，也可以从文件中加载。

形式：Properties p = new Properties();

栈（Stack）扩展为Vector（线程安全），它额可以使用insert和remove方法，在任何地方进行插入和删除操作，而不仅仅是在栈顶。

操作系统的多任务：同一时刻运行多个系统的能力。

多线程：一个程序可以执行多个任务，通常把每个任务都称为线程。可以同时运行一个以上线程的程序称为多线程程序。

单线程：程序只有一条执行路径；多线程：程序有多条执行路径。

线程依赖于进程而存在。进程是正在运行的程序，是系统进行资源分配和调用的独立单位，每一个进程都有它的内存单元。

多进程的意义：提高CPU的使用率。

一边玩游戏，一边听音乐，是同时进行的吗？

不是，原因：单CPU在某一个时间点只能做一件事情。CPU在做着程序之间的高效切换，让用户觉得是同时进行，又名（伪）并行。

伪并行：指单核或多核处理器同时执行多个进程，从而提高程序的执行效率，通过以非常有限的时间间隔在程序之间快速切换CPU，因此产生并行感，缺点是CPU时间可能分配给下一个进程，也可能不分配。

什么是线程？在一个进程内又可以执行多个任务，而每一个任务就可以看成一个线程。

线程：是程序的执行单元，又叫执行路径，是程序执行程序的最基本单位。

多线程的存在意义：不是为了提高程序的执行速度，其实是为了提高应用程序的使用率。程序的执行其实是在抢CPU的资源，CPU的执行权。多个线程其实是在抢资源，而其中的某一个进程如果执行路径比较多，就会有更高的几率抢到CPU的执行权。不能保证哪个线程在哪个时刻抢到，所以程序的执行有随机性。

并发和并行的区别：

并发：逻辑上同时发生，指在某一个时间内同时运行多个程序（时间段）。

并行：物理上同时发生，指在某一个时间点同时运行多个程序（时间点）。

Java程序运行原理：Java命令会启动Java虚拟机，启动JVM，等于启动了一个应用程序，也就是启动了一个进程，该进程会自动启动一个主线程，然后主线程去调用某个类的main方法，所以main方法运行在主线程中，在此之间的所有程序都是单线程的。

JVM虚拟机的启动时单线程还是多线程的？

多线程。原因：垃圾回收器要先启动，否则很容易会出现内存溢出。垃圾回收器加上主线程，最低启动了2个线程，因此，JVM的启动是多线程的。

创建多线程的两种方法：继承Thread类、重写Runnable接口。

继承Thread类：

1、自定义类重写Thread类；

2、在类中重写run方法；

3、创建对象；

4、启动线程。

为什么需要重写run方法？

run中包含了被线程执行的代码，以区分不需要被线程执行的代码。

线程的调度模型：

分时调度模型：所有的线程轮流使用CPU的使用权，平均分配每个线程占用CPU的时间一样。

抢占式调度：优先让优先级高的线程使用CPU，如果线程的优先级相同，则会随机选择一个，优先级高的线程获取的CPU时间片相对多一些。

Java使用的是抢占式调度。

面试题：线程的生命周期？

NEW（新建）：创建线程对象；

Waiting（等待）：有执行资格，没有执行权；

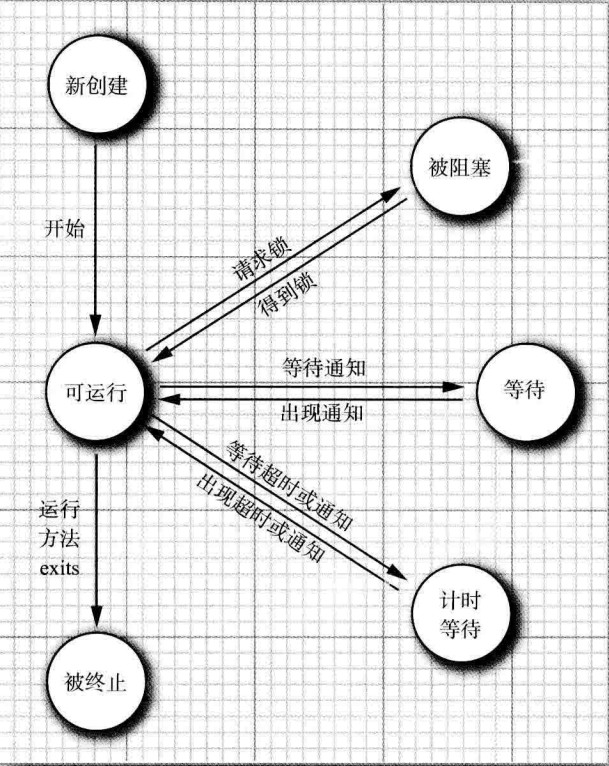
Runnable（运行）：有执行资格，有执行权；

Blocked（被阻塞）：由于一些操作让线程处于该状态，没有执行资格，没有执行权，而另一些操作却可以把它给激活，激活后处于就绪状态；调用 Object.wait方法或 Thread.join方法，或者是等待java.util.concurrent库中的Lock或Condition时，就会出现这种情况.

Timed\_Waiting（计时等待）：当线程进入Runnable状态，但是还没有开始运行的时候，此时发现需要的资源处于同步状态synchronized，这个时候线程将会进入Time\_Waiting，JVM会使用队列对这些线程进行控制，既先进行Time \_Waiting的线程会先得到JVM资源进行执行进入Waiting；

Terminated（死亡）：线程对象编程垃圾，等待被激活。

图解：



方式2：实现Runnable接口

1、自定义类实现Runnable接口；

2、重写run方法

3、创建自定义类的对象；

4、创建Thread类的对象，并把自定义类的对象作为构造参数传递

实现接口方式的好处？

1、避免Java单继承带来的局限性；

2、适合多个相同程序的代码去处理同一个资源的情况下，把线程同程序的代码、数据有效分离，极好的体现了面向对象的思想。

判断程序是否有线程安全问题的标准：

A：是否有多线程环境；

B：是否有多条数据；

C；是否有多条语句同时操作数据。

解决方法：把多条操作共享数据的代码当成一个整体，某个线程在执行时，别的线程不能来执行。

同步的好处：解决了多线程的安全问题；

同步的弊端：当线程相当多的时候，每个线程都会去判断同步上的锁，耗费了资源，降低了程序的执行效率。

同步代码块的锁对象是谁？任意对象

同步方法的锁对象是谁？this，即当前对象

同步静态方法的锁对象是谁？类名.class

Java中的每一个对象都有一个内部锁。 如果一个方法用 synchronized关键字声明，那么对象的锁将保护整个方法。也就是说，要调用该方法，线程必须获得内部的对象锁。

内部锁和条件存在一些局限。包括：

•不能中断一个正在试图获得锁的线程；

•试图获得锁时不能设定超时；

•每个锁仅有单一的条件， 可能是不够的。

同步的弊端：效率低；容易产生死锁；

死锁问题：两个或两个以上的线程在执行的过程中，因争夺资源产生的一种互相等待的现象。

线程间的通信问题：不同种类的线程针对同一资源进行操作的过程。可以使用构造方法来完成，存在线程安全问题，需要加锁（不同种类的线程都要加锁，不同种类的线程加的锁是同一把锁）。

生产者：先看是否有数据，有就等待，没有就生产；

消费者：先看是否有数据，有就消费，没有就生产。

处理该问题的方法：等待唤醒机制。

线程安全的类：StringBuffer、Vector、Hashtable

原因：类中所有的方法都加synchronized关键字，这样线程安全了，但会降低程序的效率。

得到线程安全的list的方法：List<Integer> list = Collections.synchronized(new ArrayList<>());

void interrupt()：向线程发送中断请求，线程的中断状态将被设置为 true。如果目前该线程被一个 sleep调用阻塞，那么，InterruptedException 异常被抛出。

static boolean interrupted()：测试当前线程（即正在执行这一命令的线程）是否被中断。注意，这是一个静态方法，这一调用会产生副作用，它将当前线程的中断状态重置为 false。

boolean isInterrupted()：测试线程是否被终止。不像静态的中断方法，这一调用不改变线程的中断状态。

void setPriority(int newPriority)：设置线程的优先级。允许的最大优先级为10，最小优先级为1，默认优先级为5。

static void yield()：导致当前执行线程处于让步状态。如果有其他的可运行线程具有至少与此线程同样高的优先级，那么这些线程接下来会被调度。

void join()：等待终止指定的线程。

void setDaemon(boolean isDaemon)：标识该线程为守护线程或用户线程，这一方法必须在线程启动之前调用。

void notifyAl1()：解除那些在该对象上调用wait方法的线程的阻塞状态。该方法只能在同步方法或同步块内部调用，如果当前线程不是对象锁的持有者，该方法拋出一个 IllegalMonitorStateException异常。

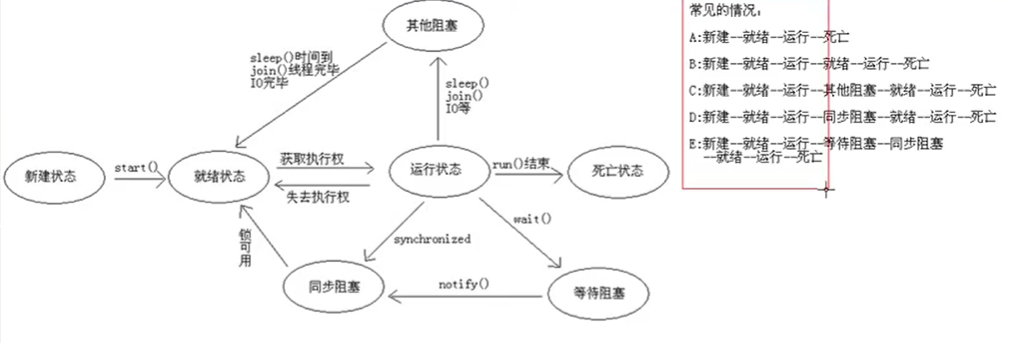
void notify()：随机选择一个在该对象上调用wait方法的线程，解除其阻塞状态。该方法只能在一个同步方法或同步块中调用，如果当前线程不是对象锁的持有者， 该方法抛出一个IllegalMonitorStateException 异常。

void wait()：导致线程进人等待状态直到它被通知。该方法只能在一个同步方法中调用，如果当前线程不是对象锁的持有者，该方法拋出一个 IllegalMonitorStateException异常。

为什么notify、wait、notifyAll不定义在Thread类中？

因为这些方法的调用必须通过锁对象调用，而锁对象是任意锁对象，因此，这些方法必须定义在Object中。

线程的状态转换图：



面试题：

1.多线程的实现方案，分别时哪几种？

2种，分别是：

继承Thread类和实现Callable接口，该方法需要和线程池结合。

2.同步有几种方式，分别是什么？

2种，分别是：同步代码块和同步方法。

3. 启动一个线程时run()还是start()，run和start的区别？

run方法：仅仅是封装被线程执行的代码，直接调用是普通方法。

start方法：首先启动了线程，之后再由JVM去调用该线程的run方法。

4.sleep和wait方法的区别？

sleep：必须指时间，执行完之后不释放锁

wait：可以不指定时间，也可以指定时间，释放锁。

5.为什么wait()，notify()，notifyAll()等方法都定义在Object类中？

因为这些方法的调用必须依赖锁对象，而同步代码块的锁对象可以是任意对象的锁，而Object代表任意的对象，因此定义在Object类中。

6.多进程：每个进程拥有自己的一整套变量，进程之间的通信较麻烦；

多线程：共享数据，使得进程之间的通信更加容易，在某些操作系统中，与进程相比，线程更轻量级，创建、撤销一个线程比启动一个新进程的开销要小的多。

7.线程生命周期图

面试题：IOC容器系列的实现：BeanFactory和ApplicationContext

BeanFactory定义了IOC容器的基本接口，是IOC容器的最基本形式。它只是一个接口类，并没有给出容器的基本实现。

ApplicationContext是容器的具体实现，即容器体系中的具体容器产品。