### heap和stack有什么区别。

java的内存分为两类，一类是栈（stack）内存，一类是堆(heap)内存。栈内存是指程序进入一个方法时，会为这个方法单独分配一块私属存储空间，用于存储这个方法内部的局部变量，当这个方法结束时，分配给这个方法的栈会释放，这个栈中的变量也将随之释放。

堆是与栈作用不同的内存，一般用于存放不放在当前方法栈中的那些数据，例如，使用new创建的对象都放在堆里，所以，它不会随方法的结束而消失。方法中的局部变量使用final修饰后，放在堆中，而不是栈中。

### java中堆（heap)和堆栈（stack)详解

stack 和 heap 都是内存的一部分

stack 空间小，速度比较快， 用来放对象的引用

heap 大，一般所有创建的对象都放在这里。

栈(stack):是一个先进后出的数据结构,通常用于保存方法(函数)中的参数,局部变量.

在java中,所有基本类型和引用类型都在栈中存储.栈中数据的生存空间一般在当前scopes内(就是由{...}括起来的区域).

堆(heap):是一个可动态申请的内存空间(其记录空闲内存空间的链表由操作系统维护),C中的malloc语句所产生的内存空间就在堆中.

在java中,所有使用new xxx()构造出来的对象都在堆中存储,当垃圾回收器检测到某对象未被引用,则自动销毁该对象.所以,理论上说java中对象的生存空间是没有限制的,只要有引用类型指向它,则它就可以在任意地方被使用.

1. 栈(stack)与堆(heap)都是Java用来在Ram中存放数据的地方。与C++不同，Java自动管理栈和堆，程序员不能直接地设置栈或堆。

2. 栈的优势是，存取速度比堆要快，仅次于直接位于CPU中的寄存器。但缺点是，存在栈中的数据大小与生存期必须是确定的，缺乏灵活性。另外，栈数据可以共享，详见第3点。堆的优势是可以动态地分配内存大小，生存期也不必事先告诉编译器，Java的垃圾收集器会自动收走这些不再使用的数据。但缺点是，由于要在运行时动态分配内存，存取速度较慢。

3. Java中的数据类型有两种。

一种是基本类型(primitive types), 共有8种，即int, short, long, byte, float, double, boolean, char(注意，并没有string的基本类型)。这种类型的定义是通过诸如int a = 3; long b = 255L;的形式来定义的，称为自动变量。值得注意的是，自动变量存的是字面值，不是类的实例，即不是类的引用，这里并没有类的存在。如int a = 3; 这里的a是一个指向int类型的引用，指向3这个字面值。这些字面值的数据，由于大小可知，生存期可知(这些字面值固定定义在某个程序块里面，程序块退出后，字段值就消失了)，出于追求速度的原因，就存在于栈中。

另外，栈有一个很重要的特殊性，就是存在栈中的数据可以共享。假设我们同时定义：

int a = 3;

int b = 3；

编译器先处理int a = 3；首先它会在栈中创建一个变量为a的引用，然后查找有没有字面值为3的地址，没找到，就开辟一个存放3这个字面值的地址，然后将a指向3的地址。接着处理int b = 3；在创建完b的引用变量后，由于在栈中已经有3这个字面值，便将b直接指向3的地址。

这样，就出现了a与b同时均指向3的情况。特别注意的是，这种字面值的引用与类对象的引用不同。

假定两个类对象的引用同时指向一个对象，如果一个对象引用变量修改了这个对象的内部状态，那么另一个对象引用变量也即刻反映出这个变化。

相反，通过字面值的引用来修改其值，不会导致另一个指向此字面值的引用的值也跟着改变的情况。

如上例，我们定义完a与b的值后，再令a=4；那么，b不会等于4，还是等于3。在编译器内部，遇到a=4；时，它就会重新搜索栈中是否有4的字面值，如果没有，重新开辟地址存放4的值；如果已经有了，则直接将a指向这个地址。因此a值的改变不会影响到b的值。

另一种是包装类数据，如Integer, String, Double等将相应的基本数据类型包装起来的类。这些类数据全部存在于堆中，Java用new()语句来显式地告诉编译器，在运行时才根据需要动态创建，因此比较灵活，但缺点是要占用更多的时间。

4.String是一个特殊的包装类数据。即可以用String str = new String("abc");的形式来创建，也可以用String str = "abc"；的形式来创建(作为对比，在JDK 5.0之前，你从未见过Integer i = 3;的表达式，因为类与字面值是不能通用的，除了String。而在JDK 5.0中，这种表达式是可以的！因为编译器在后台进行Integer i = new Integer(3)的转换！)。

前者是规范的类的创建过程，即在Java中，一切都是对象，而对象是类的实例，全部通过new()的形式来创建。Java中的有些类，如DateFormat类，可以通过该类的getInstance()方法来返回一个新创建的类，似乎违反了此原则。其实不然。该类运用了单例模式来返回类的实例，只不过这个实例是在该类内部通过new()来创建的，而getInstance()向外部隐藏了此细节。那为什么在String str = "abc"；中，并没有通过new()来创建实例，是不是违反了上述原则？其实没有。

5. 关于String str = "abc"的内部工作。Java内部将此语句转化为以下几个步骤：

(1)先定义一个名为str的对String类的对象引用变量：String str；

(2)在栈中查找有没有存放值为"abc"的地址，如果没有，则开辟一个存放字面值为"abc"的地址，接着创建一个新的String类的对象o，并将o的字符串值指向这个地址，而且在栈中这个地址旁边记下这个引用的对象o。如果已经有了值为"abc"的地址，则查找对象o，并返回o的地址。

(3)将str指向对象o的地址。

值得注意的是，一般String类中字符串值都是直接存值的。但像String str = "abc"；这种场合下，其字符串值却是保存了一个指向存在栈中数据的引用！

为了更好地说明这个问题，我们可以通过以下的几个代码进行验证。

String str1 = "abc";

String str2 = "abc";

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

注意，我们这里并不用str1.equals(str2)；的方式，因为这将比较两个字符串的值是否相等。==号，根据JDK的说明，只有在两个引用都指向了同一个对象时才返回真值。而我们在这里要看的是，str1与str2是否都指向了同一个对象。

结果说明，JVM创建了两个引用str1和str2，但只创建了一个对象，而且两个引用都指向了这个对象。

我们再来更进一步，将以上代码改成：

String str1 = "abc";

String str2 = "abc";

str1 = "bcd";

System.out.println(str1 + "," + str2); //bcd, abc

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

这就是说，赋值的变化导致了类对象引用的变化，str1指向了另外一个新对象！而str2仍旧指向原来的对象。上例中，当我们将str1的值改为"bcd"时，JVM发现在栈中没有存放该值的地址，便开辟了这个地址，并创建了一个新的对象，其字符串的值指向这个地址。

事实上，String类被设计成为不可改变(final)的类。如果你要改变其值，可以，但JVM在运行时根据新值悄悄创建了一个新对象，然后将这个对象的地址返回给原来类的引用。这个创建过程虽说是完全自动进行的，但它毕竟占用了更多的时间。在对时间要求比较敏感的环境中，会带有一定的不良影响。

再修改原来代码：

String str1 = "abc";

String str2 = "abc";

str1 = "bcd";

String str3 = str1;

System.out.println(str3); //bcd

String str4 = "bcd";

System.out.println(str1 == str4); //true

str3这个对象的引用直接指向str1所指向的对象(注意，str3并没有创建新对象)。当str1改完其值后，再创建一个String的引用str4，并指向因str1修改值而创建的新的对象。可以发现，这回str4也没有创建新的对象，从而再次实现栈中数据的共享。

我们再接着看以下的代码。

String str1 = new String("abc");

String str2 = "abc";

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

创建了两个引用。创建了两个对象。两个引用分别指向不同的两个对象。

String str1 = "abc";

String str2 = new String("abc");

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

创建了两个引用。创建了两个对象。两个引用分别指向不同的两个对象。

以上两段代码说明，只要是用new()来新建对象的，都会在堆中创建，而且其字符串是单独存值的，即使与栈中的数据相同，也不会与栈中的数据共享。

6. 数据类型包装类的值不可修改。不仅仅是String类的值不可修改，所有的数据类型包装类都不能更改其内部的值。

7. 结论与建议：

(1)我们在使用诸如String str = "abc"；的格式定义类时，总是想当然地认为，我们创建了String类的对象str。担心陷阱！对象可能并没有被创建！唯一可以肯定的是，指向String类的引用被创建了。至于这个引用到底是否指向了一个新的对象，必须根据上下文来考虑，除非你通过new()方法来显要地创建一个新的对象。因此，更为准确的说法是，我们创建了一个指向String类的对象的引用变量str，这个对象引用变量指向了某个值为"abc"的String类。清醒地认识到这一点对排除程序中难以发现的bug是很有帮助的。

(2)使用String str = "abc"；的方式，可以在一定程度上提高程序的运行速度，因为JVM会自动根据栈中数据的实际情况来决定是否有必要创建新对象。而对于String str = new String("abc")；的代码，则一概在堆中创建新对象，而不管其字符串值是否相等，是否有必要创建新对象，从而加重了程序的负担。

(3)当比较包装类里面的数值是否相等时，用equals()方法；当测试两个包装类的引用是否指向同一个对象时，用==。

(4)由于String类的final性质，当String变量需要经常变换其值时，应该考虑使用StringBuffer类，以提高程序效率。

### java中会存在内存泄漏吗，请简单描述。

所谓内存泄露就是指一个不再被程序使用的对象或变量一直被占据在内存中。java中有垃圾回收机制，它可以保证一对象不再被引用的时候，即对象变成了了孤儿的时候，对象将自动被垃圾回收器从内存中清除掉。由于Java 使用有向图的方式进行垃圾回收管理，可以消除引用循环的问题，例如有两个对象，相互引用，只要它们和根进程不可达的，那么GC也是可以回收它们的，例如下面的代码可以看到这种情况的内存回收：

**package** com.huawei.interview;

**import** java.io.IOException;

**public** **class** GarbageTest {

/\*\*

\* **@param** args

\* **@throws** IOException

\*/

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

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

**try** {

*gcTest*();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

System.*out*.println("has exited gcTest!");

System.*in*.read();

System.*in*.read();

System.*out*.println("out begin gc!");

**for**(**int** i=0;i<100;i++)

{

System.*gc*();

System.*in*.read();

System.*in*.read();

}

}

**private** **static** **void** gcTest() **throws** IOException {

System.*in*.read();

System.*in*.read();

Person p1 = **new** Person();

System.*in*.read();

System.*in*.read();

Person p2 = **new** Person();

p1.setMate(p2);

p2.setMate(p1);

System.*out*.println("before exit gctest!");

System.*in*.read();

System.*in*.read();

System.*gc*();

System.*out*.println("exit gctest!");

}

**private** **static** **class** Person

{

**byte**[] data = **new** **byte**[20000000];

Person mate = **null**;

**public** **void** setMate(Person other)

{

mate = other;

}

}

}

java中的内存泄露的情况：长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄露，尽管短生命周期对象已经不再需要，但是因为长生命周期对象持有它的引用而导致不能被回收，这就是java中内存泄露的发生场景，通俗地说，就是程序员可能创建了一个对象，以后一直不再使用这个对象，这个对象却一直被引用，即这个对象无用但是却无法被垃圾回收器回收的，这就是java中可能出现内存泄露的情况，例如，缓存系统，我们加载了一个对象放在缓存中(例如放在一个全局map对象中)，然后一直不再使用它，这个对象一直被缓存引用，但却不再被使用。

检查java中的内存泄露，一定要让程序将各种分支情况都完整执行到程序结束，然后看某个对象是否被使用过，如果没有，则才能判定这个对象属于内存泄露。

如果一个外部类的实例对象的方法返回了一个内部类的实例对象，这个内部类对象被长期引用了，即使那个外部类实例对象不再被使用，但由于内部类持久外部类的实例对象，这个外部类对象将不会被垃圾回收，这也会造成内存泄露。

下面内容来自于网上（主要特点就是清空堆栈中的某个元素，并不是彻底把它从数组中拿掉，而是把存储的总数减少，本人写得可以比这个好，在拿掉某个元素时，顺便也让它从数组中消失，将那个元素所在的位置的值设置为null即可）：

我实在想不到比那个堆栈更经典的例子了,以致于我还要引用别人的例子，下面的例子不是我想到的，是书上看到的，当然如果没有在书上看到，可能过一段时间我自己也想的到，可是那时我说是我自己想到的也没有人相信的。

    public class Stack {  
    private Object[] elements=new Object[10];  
    private int size = 0;  
    public void push(Object e){  
    ensureCapacity();  
    elements[size++] = e;  
    }  
    public Object pop(){  
    if( size == 0)

    throw new EmptyStackException();  
    return elements[--size];  
    }  
    private void ensureCapacity(){  
    if(elements.length == size){  
    Object[] oldElements = elements;  
    elements = new Object[2 \* elements.length+1];  
    System.arraycopy(oldElements,0, elements, 0, size);  
    }  
    }  
    }  
    上面的原理应该很简单，假如堆栈加了10个元素，然后全部弹出来，虽然堆栈是空的，没有我们要的东西，但是这是个对象是无法回收的，这个才符合了内存泄露的两个条件：无用，无法回收。

    但是就是存在这样的东西也不一定会导致什么样的后果，如果这个堆栈用的比较少，也就浪费了几个K内存而已，反正我们的内存都上G了，哪里会有什么影响，再说这个东西很快就会被回收的，有什么关系。下面看两个例子。

    例子1  
    public class Bad{  
    public static Stack s=Stack();  
    static{  
    s.push(new Object());  
    s.pop(); //这里有一个对象发生内存泄露  
    s.push(new Object()); //上面的对象可以被回收了，等于是自愈了  
    }  
    }  
    因为是static，就一直存在到程序退出，但是我们也可以看到它有自愈功能，就是说如果你的Stack最多有100个对象，那么最多也就只有100个对象无法被回收其实这个应该很容易理解，Stack内部持有100个引用，最坏的情况就是他们都是无用的，因为我们一旦放新的进取，以前的引用自然消失！

内存泄露的另外一种情况：当一个对象被存储进HashSet集合中以后，就不能修改这个对象中的那些参与计算哈希值的字段了，否则，对象修改后的哈希值与最初存储进HashSet集合中时的哈希值就不同了，在这种情况下，即使在contains方法使用该对象的当前引用作为的参数去HashSet集合中检索对象，也将返回找不到对象的结果，这也会导致无法从HashSet集合中单独删除当前对象，造成内存泄露。