* 基本类型数据的值存在于栈中（先查找，没有再创建）；包装类的值存在于堆中；
* 系统已经把-128到127之间的Integer缓存到一个Integer数组中去了，如果你要把一个int变成一个Integer对象，首先去缓存中找，找到的话直接返回引用给你就行了，不必再new一个；不在范围用==返回false；

**1：String：**

String str = "abc"创建对象的过程：

1 首先在常量池中查找是否存在内容为"abc"的字符串对象；

2 如果不存在则在常量池中创建一个"abc"的字符串对象，并让str引用该对象；

3 如果存在则直接让str引用该对象；

**注意：**常量池属于类信息的一部分，而类信息反映到JVM内存模型中是对应存在于JVM内存模型的方法区，也就是说这个类信息中的常量池概念是存在于在方法区中。一般这种情况下，"abc"在编译时就被写入字节码中，所以class被加载时，JVM就为"abc"在常量池中分配内存，所以和静态区差不多；

String str = new String("abc")创建实例的过程：

1首先定义一个str的String类型的引用并存放在栈中；

2在字符串常量池中查看是否存在内容为"abc"字符串对象 ；

3若存在则跳过这个步骤，若不存在，则在字符串常量池中创建一个内容为"abc"的字符串对象；

4执行new操作，在堆中创建一个指定的对象"abc"，这里堆的对象是字符串常量池“abc”对象的一个拷贝对象；

5让str指向堆中“abc”这个对象（也就是存储这个对象的在堆中的地址）；

String类和StringBuffer类都在java.lang中定义。String类重写了equals方法，使其比较的是字符的序列，而不再是内存地址。对于字符串变量来说，使用“==”和“equals()”方法比较字符串时，其比较方法不同。“==”比较两个变量本身的值，即两个对象在内存中的首地址；“equals()”比较字符串中所包含的内容是否相同。String(字符串)、StringBuffer(线程安全的可变字符序列)、StringBuilder(可变字符序列)。

**java中String 的常用方法有如下**：

1）String.concat(String str)：该方法的参数为一个String类对象,作用是将参数中的字符串str连接到原来字符串的后面；

2）String.length()：返回字串的长度,这里的长度指的是字符串中Unicode字符的数目； **和StringBuffer一样**

3）String.charAt(int index)：该方法在一个特定的位置索引一个字符串,以得到字符串中指定位置的字符.值得注意的是,在字符串中第一个字符的索引是0,第二个字符的索引是1,依次类推,最后一个字符的索引是length()-1；**和StringBuffer一致**

4）String. equalsIgnoreCase(String anotherString)：该方法和equals方法相似,不同的地方在于,equalsIgnoreCase方法将忽略字母大小写的区别；

5）String.substring(int beginindex, int endindex)：该方法从当前字符串中取出一个子串,该子串从beginIndex位置起至endIndex-1为结束.子串返的长度为endIndex-beginIndex.；如果没有endindex，则截取biginindex之后的所有字符串； **和StringBuffer一致**

6）String.startsWith(String prefix)：该方法用于判断当前字符串的前缀是否和参数中指定的字符串prefix一致,如果是,返回true,否则返回false；

7）String.startsWith(String prefix, int toffset)：该方法用于判断当前字符串从toffset位置开始的子串的前缀是否和参数中指定的字符串prefix一致,如果是,返回true,否则返回false；

8）String.indexOf(int ch, int fromindex)：该方法从fromIndex位置向后查找,返回的仍然是字符ch在字符串第一次出现的位置；如果未设置fromindex，则默认从头开始查找；

9）String.lastIndexOf(int ch, int fromindex)：该方法从fromIndex位置向前查找,返回的仍然是字符ch在字符串第一次出现的位置，未设置fromindex的话，默认从末尾开始查找； （8与9也可以把字符替换成字符串，进行查找）

10）String.toLowerCase()：该方法将字符串中所有字符转换成小写,并返回转换后的新串；toUpperCase（）是转换成大写的哦；

11）String.replace(char oldchar, char newchar)：该方法用字符newChar替换当前字符串中所有的字符oldChar,并返回一个新的字符串；**和StringBuffer一致**

12）String.replaceFirst(String regex, String replacement)：该方法用字符串replacement的内容替换当前字符串中遇到的第一个和字符串regex相一致的子串,并将产生的新字符串返回；

13）String.replaceAll(String regex, String replacement)：该方法用字符串replacement的内容替换当前字符串中遇到的所有和字符串regex相一致的子串,并将产生的新字符串返回；

14）String.trim()：返回一个字符串，该字符串是删除调用字符串前后的空白符所得的字符串；

**2：StringBuffer:**

**StringBuffer类中没重新定义equals这个方法，因此这个方法就来自Object类，而Object类中的equals方法是用来比较“地址”的，所以等于false。StringBuffer中的常用方法：**

1）StingBuffer(int size)：设置指定缓存区大小，无参数时默认为16个字符的空间；

2）调用length()方法可以得到StringBuffer对象的长度，调用capacity()可以得到总的分配容量。两个方法都是返回一个int类型的值；

3）如果想在构造StringBuffer对象后为某些字符预分配空间，可以使用ensureCapacity(int capacity)方法，设置缓冲区的大小；

4）使用setLength(int len)方法可以设置StringBuffer对象的长度，如果len大于StringBuffer对象当前的length()值的话，那么会在StringBuffer对象后面加上空字符；如果比length()小的话，则len后面的字符串会丢失；

5） 使用charAt(int where)方法可以得到StringBuffer对象中指定位置上的字符； setCharAt(int where, char ch)可以设置指定位置上的字符；对于这两种方法，where值必须是非负的，同时不能超过或等于StringBuffer对象的长度；

6） append()方法将任一其他类型数据的字符串形式连接到调用StringBuffer对象的后面，对所有内置的类型和Object，它都有重载形式；

7）insert(int index, String str /char ch)方法将一个字符串插入另一个字符串中。

8) delete(int startIndex, int endIndex) 删除两个索引之间的字符串； deleteCharAt(int loc)指定位置的字符。例如删除第一个字符后的所有字符： strbf.delete(1,strbf.length())；

9） replace完成在StringBuffer内部用一个字符串代替另一个指定起始位置和结束位置的字符串的功能，注意的是，被代替的字符不包括结束位置上的字符，它的一般形式是，：replace(int startIndex, int endIndex, String str)；没有位置参数的话，默认全部；

10 substring(int startIndex, int endIndex) 返回StringBuffer的一部分值；

**3：equals方法：**

**引用数据类型中**，对于非字符串变量来说，"=="和"equals"方法的作用是相同的都是用来比较其对象在堆内存的首地址，即用来比较两个引用变量是否指向同一个对象。如果是**基本类型**比较，那么只能用==来比较，不能用equals（会出现编译错误）, equals方法不能运用于基本类型的比较。对于**基本类型的包装类型**，比如Boolean、Character、Byte、Shot、Integer、Long、Float、Double等的引用变量，==是比较地址的，而equals是比较内容的。

默认情况下也就是从超类Object继承而来的equals方法与‘==’是完全等价的，比较的都是对象的内存地址，但我们可以重写equals方法，使其按照我们的需求的方式进行比较，我们在重写equals方法时，还是需要注意如下几点规则的：**自反性：**对于任何非null的引用值x，x.equals(x)应返回true； **对称性：**对于任何非null的引用值x与y，当且仅当y.equals(x)返回true时，x.equals(y)才返回true；**传递性：**对于任何非null的引用值x、y与z，如果y.equals(x)返回true，y.equals(z)返回true，那么x.equals(z)也应返回true；**一致性**：对于任何非null的引用值x与y，假设对象上equals比较中的信息没有被修改，则多次调用x.equals(y)始终返回true或者始终返回false；对于任何非空引用值x，x.equal(null)应返回false。

**hashCode（）：**

**为什么重写equals()的同时还得重写hashCode()？**

在java中，我们可以使用hashCode()来获取对象的哈希码，其值就是对象的存储地址，这个方法在Object类中声明，因此所有的子类都含有该方法。hashCode的意思就是散列码，也就是哈希码，是由对象导出的一个整型值。字符串s与t拥有相同的散列码，这是因为字符串的散列码是由内容导出的。而字符串缓冲sb与tb却有着不同的散列码，这是因为StringBuilder没有重写hashCode方法，它的散列码是由Object类默认的hashCode方法计算出来的对象存储地址，所以散列码自然也就不同了。重写equals方法时也必须重写hashCode方法。

* 在java应用程序执行期间，如果在equals方法比较中所用的信息没有被修改，那么在同一个对象上多次调用hashCode方法时必须一致地返回相同的整数。如果多次执行同一个应用时，不要求该整数必须相同。
* **如果两个对象通过调用equals方法是相等的，那么这两个对象调用hashCode方法必须返回相同的整数**。
* 如果两个对象通过调用equals方法是不相等的，不要求这两个对象调用hashCode方法必须返回不同的整数。但是程序员应该意识到对不同的对象产生不同的hash值可。

通过前面的分析，我们知道在Object类中，hashCode方法是通过Object对象的地址计算出来的，因为Object对象只与自身相等，所以同一个对象的地址总是相等的，计算取得的哈希码也必然相等，对于不同的对象，由于地址不同，所获取的哈希码自然也不会相等。因此到这里我们就明白了，**如果一个类重写了equals方法，但没有重写hashCode方法**，将会直接违法了第2条规定，这样的话，如果我们通过映射表(Map接口)操作相关对象时，就无法达到我们预期想要的效果。

**4：数组：**

**Java 语言中提供的数组是用来存储固定大小的同类型元素。**

下面声明数组的方法是一样的：

dataType[] arrayRefVar; // 首选的方法

dataType arrayRefVar[]; // 效果相同，但不是首选方法、

注：JDK 1.5 引进了一种新的循环类型，被称为 foreach 循环或者加强型循环，它能在不使用下标的情况下遍历数组；for (double element: myList){}；

二维数组的声明方式：

1. **int** a[][]={{1,2,3},{4,5,6}};
2. //第二种方式；
3. **int**[][] ints = **new** **int**[4][2];
4. ints[i][j] =\_\_; //分别赋值
5. //第三种方式：第二维的长度可以动态申请
6. **int**[][] arr3 = **new** **int**[5][];//五行的长度

声明一个数组：

1. String[] aArray = **new** String[5];
2. String[] bArray = {"a","b","c", "d", "e"};
3. String[] cArray = **new** String[]{"a","b","c","d","e"};

输出一个数组：

1. **int**[] intArray = { 1, 2, 3, 4, 5 };
2. String intArrayString = Arrays.toString(intArray);
4. // print directly will print reference value
5. System.out.println(intArray);
6. // [I@7150bd4d
8. System.out.println(intArrayString);
9. // [1, 2, 3, 4, 5]

根据数组创建ArrayList：

1. String[] stringArray = { "a", "b", "c", "d", "e" };
2. ArrayList<String> arrayList = **new** ArrayList<String>(Arrays.asList(stringArray));
3. // [a, b, c, d, e]
4. System.out.println(arrayList);

检查数组是否包含某个值：

1. String[] stringArray = { "a", "b", "c", "d", "e" };
2. **boolean** b = Arrays.asList(stringArray).contains("a");
3. // true
4. System.out.println(b);

合并连接两个数组：

1. **int**[] intArray = { 1, 2, 3, 4, 5 };
2. **int**[] intArray2 = { 6, 7, 8, 9, 10 };
3. // Apache Commons Lang 库
4. **int**[] combinedIntArray = ArrayUtils.addAll(intArray, intArray2);

将ArrayList转换为数组：

1. String[] stringArray = { "a", "b", "c", "d", "e" };
2. ArrayList<String> arrayList = **new** ArrayList<String>(Arrays.asList(stringArray));
3. String[] stringArr = **new** String[arrayList.size()];
4. arrayList.toArray(stringArr);
5. **for** (String s : stringArr)  //超级for循环
6. System.out.println(s);

将数组转换为set：

1. Set<String> set = **new** HashSet<String>(Arrays.asList(stringArray));
2. //[d, e, b, c, a]
3. System.out.println(set);

将数组元素反转：

1. **int**[] intArray = { 1, 2, 3, 4, 5 };
2. ArrayUtils.reverse(intArray);
3. //[5, 4, 3, 2, 1]
4. System.out.println(Arrays.toString(intArray));

**5：toString：可以结合7看**

Object 类的 toString 方法返回一个字符串，该字符串由类名（对象是该类的一个实例）加上“@”和此对象哈希码的无符号十六进制表示组成。a.toString()，a虽然是数组名，但那是在这是一个对象实例。Arrays的toString方法是返回指定数组内容的字符串表示形式。

**6：Arrays用法整理：**

**java.util.Arrays:**

1） asList 方法返回一个固定大小的List：List<Integer> intList = Arrays.asList(1, 2, 3, 4);

2） 填充数组：以int型数组为例,Arrays.fill( int[] a, int val) 表示val将填充a数组的每一个元素；Arrays. fill(int[] a,int fromIndex,int toIndex,int val) 将val插入到a数组的指定范围内；不会包括toindex位置；

3） 数组排序：Arrays.sort(a， fromindex， toindex); 不指定排序范围默认为全部；

4） 比较一维数组元素是否相等：Arrays.equals(a, b); Arrays.deepequals(array)适用于多维数组；

5） 二分查找法找指定元素的索引值（下标）：Arrays.binarySearch(array, fromindex, toindex, 要查找的数)，返回的是要查找数的索引，如果有的话，没有返回的是负数，查找范围可以不指定；

6） Arrays的toString方法可以方便我们打印出数组内容，Arrays.toString(array)适用于一维数组；Arrays.deeptoString(array)适用于二维数组；

**7：java中的字符串转型：**

1） toString，需要保证调用这个方法的类、方法、变量不为null，否则会报空指针。在这种使用方法中，因为java.lang.Object类里已有public方法对象.toString()，所以对任何严格意义上的java对象都可以调用此方法。但在使用时要注意，必须保证object不是null值，否则将抛出NullPointerException异常。采用这种方法时，通常派生类会覆盖Object里的toString（）方法。

2） String.valueOf。这个方法在使用的时候是有些特殊的。一般情况下，如果是确定类型的null传入，返回的是字符串“null”，而如果直接传入null，则会发生错误。可以把将基本数据型态转换成 String 的 static 方法。如果要将 String 转换成基本数据型态，大多需要使用基本数据型态的包装类别，比如说 String 转换成 byte，可以使用 Byte.parseByte(String s) ；这一类的方法如果无法将 s 分析 则会丢出 NumberFormatException 。转成其他的基本类型的方法类似；

3） (String) 字符串类型强转。需要保证的是类型可以转成String类型。因null值可以强制转换为任何java类类型，(String)null也是合法的。

**8：Array与List：**

Array—是基于索引(index)的数据结构，它使用索引在数组中搜索和读取数据是很快的。Array获取数据的时间复杂度是O(1),但是要删除数据却是开销很大的，因为这需要重排数组中的所有数据

List—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，它继承Collection。List有两个重要的实现类：ArrayList和LinkedList。

**ArrayList：**可以看作是能够自动增长容量的数组，利用ArrayList的toArray返回一个数组；Arrays.asList返回一个列表：

1） ArrayList底层采用数组实现，当使用不带参数的构造方法生成ArrayList对象时，实际上会在底层生成一个长度为10的Object类型的数组；

2） 如果增加的元素个数超过10个，那么ArrayList底层会生成一个新的数组，长度为原数组的1.5倍+1，然后将原数组的内容复制到新数组中，并且后续增加的内容都会放到新的数组当中，当新的数组无法容纳增加的元素时，重读该过程；

3） 对于ArrayList元素的删除操作，需要将被删除元素的后续元素向前移动，代价比较大；

4） 集合当中只能放置对象的引用，无法放置原生数据类型，我们必须使用原生数据的包装类才能加入到集合当中；

5） 集合当中都是Object类型，因此取出来的也是Object类型，那么必须要使用强制类型转化将其转换成真正的类型（放置进去的类型）；

**LinkedList:** 是一个双链表,在添加和删除元素时具有比ArrayList更好的性能.但在get与set方面弱于ArrayList.当然,这些对比都是指数据量很大或者操作很频繁的情况下的对比,如果数据和运算量很小,那么对比将失去意义；

**LinkedList和ArrayList区别：**

LinkedList和ArrayList的差别主要来自于Array和LinkedList数据结构的不同。如果你很熟悉Array和LinkedList，你容易得出下面的结论：

1) 因为**Array是基于索引(index)的数据结构**，它使用索引在数组中搜索和读取数据是很快的。Array获取数据的时间复杂度是O(1),但是要删除数据却是开销很大的，因为这需要重排数组中的所有数据。

2) 相对于ArrayList，LinkedList插入是更快的。因为LinkedList不像ArrayList一样，不需要改变数组的大小，也不需要在数组装满的时候要将所有的数据重新装入一个新的数组，这是ArrayList最坏的一种情况，时间复杂度是O(n)，而LinkedList中插入或删除的时间复杂度仅为O(1)。ArrayList在插入数据时还需要更新索引（除了插入数组的尾部）。

3) 类似于插入数据，删除数据时，LinkedList也优于ArrayList。

4) **LinkedList需要更多的内存**，因为**ArrayList的每个索引的位置是实际的数据**，而LinkedList中的每个节点中存储的是实际的数据和前后节点的位置。

什么情况下用LinkedList或者使用ArrayList？

1) 你的应用不会随机访问数据。因为如果你需要LinkedList中的第n个元素的时候，你需要从第一个元素顺序数到第n个数据，然后读取数据。

2) 你的应用更多的插入和删除元素，更少的读取数据。因为插入和删除元素不涉及重排数据，所以它要比ArrayList要快。

以上就是关于ArrayList和LinkedList的差别。你需要一个不同步的基于索引的数据访问时，请尽量使用ArrayList。ArrayList很快，也很容易使用。但是要记得要给定一个合适的初始大小，尽可能的减少更改数组的大小；

采用集合的toArray()方法直接把**List集合转换成数组**，这里需要注意，不能这样写：

String[] array = (String[]) mlist.toArray(); 这样写的话，编译运行时会报类型无法转换java.lang.ClassCastException的错误，这是为何呢，这样写看起来没有问题啊？因为java中的**强制类型转换是针对单个对象才有效果的**，而List是多对象的集合，所以将整个List强制转换是不行的 正确的写法应该是这样的：String[] array = mlist.toArray(new String[0]);

1） 数组中存放的数据类型必须一致，可以存放基本类型数据；list中可以存放不同类型数据，但不能存放基本类型数据 ；

2） 数组初始化后大小固定，且数据都已经被赋值；int [] numbers = new int[10];则初始化后数组内为十个0 ；而list numbers = new ArrayList<>(10); numbers.size()=0 ，number.get(2)返回的不是null，而是报越界错误；

**9：列表：**

集合是Java API所提供的一系列类，可以用于动态存放多个对象。集合与数组的不同在于，集合是大小可变的序列，而且元素类型可以不受限定，只要是引用类型。集合中不能放基本数据类型，但可以放基本数据类型的包装类。 集合类全部支持泛型，是一种数据安全的用法。java列表是集合的一种，并且列表中的元素是有序的，且允许重复。java列表又称为java list。

实际上有两种List：一种是基本的ArrayList，其优点在于随机访问元素，另一种是更强大的LinkedList，它并不是为快速随机访问设计的，而是具有一套更通用的方法。次序是List最重要的特点：它保证维护元素特定的顺序。List为Collection添加了许多方法，使得能够向List中间插入与移除元素(这只推荐LinkedList使用)。一个List可以生成ListIterator，使用它可以从两个方向遍历List，也可以从List中间插入和移除元素。

ArrayList：是由数组实现的List。允许对元素进行快速随机访问，但是向List中间插入与移除元素的速度很慢。ListIterator只应该用来由后向前遍历 ArrayList，而不是用来插入和移除元素。因为那比LinkedList开销要大很多。

|  |
| --- |
| public static void main(String[] args) { |
| // ArrayList类实现一个可增长的动态数组 |
| List<String> list = new ArrayList<String>(); |
| // 插入元素 |
| list.add("list1"); |
| list.add("list2"); |
| // 打印list的大小 |
| System.out.println(list.size()); |
| // 按索引移除元素 |
| list.remove(0); |
| // 按对象移除元素 |
| list.remove("list2"); |
| // 打印list的大小 |
| System.out.println(list.size()); |
| // 清空list |
| list.clear(); |
| } |

LinkedList ：对顺序访问进行了优化，向List中间插入与删除的开销并不大。随机访问则相对较慢。(使用ArrayList代替)还具有下列方法：addFirst(), addLast(), getFirst(), getLast(), removeFirst() 和 removeLast()， 这些方法 (没有在任何接口或基类中定义过)使得LinkedList可以当作堆栈、队列和双向队列使用；

|  |
| --- |
| public static void main(String[] args) { |
| // LinkedList类实现了链表，可初始化化为空或者已存在的集合 |
| LinkedList<String> list = new LinkedList<String>(); |
| // 插入元素 |
| list.add("list2"); |
| list.add("list3"); |
| // 向链表头插入数据 |
| list.addFirst("list1"); |
| // 向链表尾插入数据 |
| list.addLast("list4"); |
| for (String str : list) { |
| System.out.println(str); |
| } |
| // 获取链表头数据 |
| System.out.println("链表头数据:" + list.getFirst()); |
| // 获取链表尾数据 |
| System.out.println("链表尾数据:" + list.getLast()); |

注：此外如果需要对某个列表进行逆序，可以使用Collections中的reverse方法；

Collections中的sort方法对列表排序；

列表的遍历方法：

方法一：

　　超级for循环遍历：

|  |
| --- |
| for(String attribute : list) |
| { |
| System.out.println(attribute); |
| } |

内部调用第一种，换汤不换药，因此比Iterator 慢，这种循环方式还有其他限制，不建议使用它

方法二：

　　对于ArrayList来说速度比较快，用for循环，以size为条件遍历：

|  |
| --- |
| for(int i = 0 ; i < list.size() ; i++) |
| { |
| system.out.println(list.get(i)); |
| } |

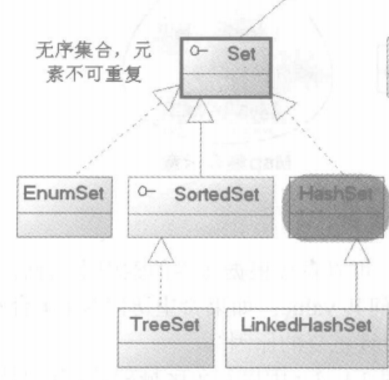
内部不锁定，效率最高，但是当写多线程时要考虑并发操作的问题。

方法三：

　　集合类的通用遍历方式，从很早的版本就有，用迭代器迭代：

|  |
| --- |
| Iterator it = list.iterator(); |
| while(it.hasNext()) { |
| System.ou.println(it.next); |
| } |

这种方式在循环执行过程中会进行数据锁定，性能稍差，同时，如果你想在循环过程中去掉某个元素，只能调用it.remove方法，不能使用list.remove方法，否则一定出现并发访问的错误；

**10：set**

|  |
| --- |
| **add**(o)如果 set 中尚未存在指定的元素，则添加此元素。 |
| **addAll**(Collection o) 如果 set 中没有指定 collection 中的所有元素，则将其添加到此 set 中。 |
| **clear**() 移除 set 中的所有元素。 |
| **contains**(Object o)  如果 set 包含指定的元素，则返回 true。 |
| **containsAll**(Collection o) 如果此 set 包含指定 collection 的所有元素，则返回 true。 |
| **equals**(Object o)    比较指定对象与此 set 的相等性。 |
| **hashCode**() 返回set 的哈希码值。 |
| **isEmpty**()  如果 set 不包含元素，则返回 true。 |
| **iterator**() 返回在此 set 中的元素上进行迭代的迭代器。 |
| **remove**(Object o) set 中存在指定的元素，则将其移除。 |
| **removeAll**(Collection c)移除 set 中那些包含在指定 collection 中的元素。 |
| **retainAll**(Collection o) 仅保留 set 中那些包含在指定 collection 中的元素。 |
| **size**() 返回 set 中的元素数（其容量）。 |
| **toArray**() 返回一个包含 set 中所有元素的数组。 |
| **toArray**(T[] a)    返回一个包含 set 中所有元素的数组；返回数组的运行时类型是指定数组的类型。 |

**HashSet：**

HashSet底层用的是哈希表，它把对象根据其哈希值存放到对应的区域里。由于这种特性，两个在不同区域的对象会被认为不相同的。所以如果对象要存放到Hash集合里面，则需要重写对象的hashCode方法，让相等的对象的hashCode的值也相等。

**TreeSet：**

TreeSet采用的数据结构是红黑树，我们可以让它按指定规则对其中的元素进行排序。它又是如何判断两个元素是否相同呢？除了用equals方法检查两个元素是否相同外，还要检查compareTo方法是否返回为0，所以如果对象要存放到Tree集合里，需要在重写compareTo时，把相同的对象的比较值定为0，防止相同的元素被重复添加进集合中。

我们有时候需要的是一个有序的集合，所以排序是很有必要的，在**Set中使用treeset排序是很有必要的：当存储的数据为int、字符、String类型的时候，treeset是可以自动进行排序的；**

**列表的方法：**

|  |
| --- |
| **add**(o)  向列表的尾部追加指定的元素。 |
| **add**(int index, element) 在列表的指定位置插入指定元素 |
| **addAll**(Collection o) 追加指定 collection 中的所有元素到此列表的结尾，顺序是指定 collection 的迭代器返回这些元素的顺序 |
| **addAll**(int index，Collection o) 将指定 collection 中的所有元素都插入到列表中的指定位置。 |
| **clear**()       从列表中移除所有元素。 |
| **contains**(Object o) 如果列表包含指定的元素，则返回 true。 |
| **containsAll**(Collection o)  如果列表包含指定 collection 的所有元素，则返回true。 |
| **equals**(Object o)   比较指定的对象与列表是否相等。 |
| **get**(int index)      返回列表中指定位置的元素。 |
| **hashCode**()      返回列表的哈希码值。 |
| **indexOf**(Object o)      返回列表中首次出现指定元素的索引，如果列表不包含此元素，则返回 -1。 |
| **isEmpty**()    如果列表不包含元素，则返回 true。 |
| **iterator**()  返回以正确顺序在列表的元素上进行迭代的迭代器。 |
| **lastIndexOf**(Object o)   返回列表中最后出现指定元素的索引，如果列表不包含此元素，则返回 -1。 |
| **listIterator**()  返回列表中元素的列表迭代器（以正确的顺序）。 |
| **listIterator**(int index)  返回列表中元素的列表迭代器（以正确的顺序），从列表的指定位置开始。 |
| **remove**(int index) 移除列表中指定位置的元素。 |
| **remove**(Object o)  移除列表中出现的首个指定元素。 |
| **removeAll**(Collection o)从列表中移除指定 collection 中包含的所有元素。 |
| **retainAll**(Collection o)仅在列表中保留指定 collection 中所包含的元素。 |
| **set**(int index, element) 用指定元素替换列表中指定位置的元素。 |
| **size**()      返回列表中的元素数。 |
| **subList**(int fromIndex, int toIndex)   返回列表中指定 fromIndex（包括 ）和toIndex（不包括）之间的部分视图。 |
| **toArray**()  返回以正确顺序包含列表中的所有元素的数组。 |
| **toArray**(T[] a) 返回以正确顺序包含列表中所有元素的数组；返回数组的运行时类型是指定数组的运行时类型。 |

**11：ArrayUtils：相当于Array的包装类**

1）ArrayUtils.isEmpty(strs) : 判断数组是否为空 , 不为空返回false,为空true；

2）ArrayUtils.isNotEmpty(strs) : 判断数组是否不为空 , 不为空返回true,为空false；

3）ArrayUtils.isSameLength(strs, strs2) : 判断两个数组长度是否相等, 长度相等返回true,否则返回false 。相比较的两个数组类型必须相同；

4）ArrayUtils.isEquals(strs, strs2) 判断两个数组是否相等；

5）ArrayUtils.toString() 将一个数组转换成String,用于打印；

6）ArrayUtils.subarray 截取子数组 ： 根据 起始索引 startIndexInclusive 到 结束索引startIndexInclusive，包括结束索引；

7）ArrayUtils.indexOf 查询某个object在数组中的位置，可是指定起始搜索位置；

8）ArrayUtils.lastIndexOf 反向查询某个object在数组中的位置 可是指定起始搜索位置；

9）ArrayUtils.contains 查询某个object是否在数组中；

10）ArrayUtils.reverse 反转数组；

11）ArrayUtils.add 添加一object到数组；

12）ArrayUtils.addAll 合并两个数组；

13）ArrayUtils.remove 删除数组某个位置的元素；

14）ArrayUtils.removeElement 删除数组中某个对象；