

## 13.01\_常见对象(StringBuffer类的概述)

A:StringBuffer类概述:

通过JDK提供的API,查看StringBuffer类的说明:  
线程安全的可变字符序列

B:StringBuffer和String的区别:

String是一个不可变的字符序列;  
StringBuffer是一个可变的字符序列

## 13.02\_常见对象(StringBuffer类的构造方法)

A:StringBuffer的构造方法:

public StringBuffer():构造一个其中不带字符的字符串缓冲区,其初始容量为16个字符  
public StringBuffer(int capacity):指定容量的字符串缓冲区对象  
public StringBuffer(String str):指定字符串内容的字符串缓冲区对象

B:StringBuffer的方法:

public int capacity():返回当前容量,理论值(不掌握),等于默认值16+length()实际值为容量值  
public int length():返回长度(即字符个数),实际值

C:案例演示:

构造方法和长度方法的使用:

代码演示如下:

```
public static void main(String[] args) {  
    StringBuffer sb = new StringBuffer();//无参构造,默认容量为16  
    System.out.println(sb.length());//0,容器中的字符个数,实际值  
    System.out.println(sb.capacity());//16+0=16,容器的初始容量,理论值  
  
    StringBuffer sb2 = new StringBuffer(10);  
    System.out.println(sb2.length());//0,return, the number of chars in this sequence, 里面没有字符  
    System.out.println(sb2.capacity());//10+0=10  
  
    StringBuffer sb3 = new StringBuffer("heima");  
    System.out.println(sb3.length());//5, 实际字符的个数  
    System.out.println(sb3.capacity());//16+5=21, 字符串的length + 初始容量  
}
```

## 13.03\_常见对象(StringBuffer的添加功能)

A:StringBuffer的添加功能:

### public StringBuffer append(String str):

可以把任意类型数据,添加到字符串缓冲区里面,并返回字符串缓冲区本身

注意:StringBuffer是字符串缓冲区,当new的时候是在堆内存创建了一个对象,底层是一个长度为16的字符数组,当调用添加的方法时,不会再重新创建对象,而是在不断向原缓冲区添加字符

### public StringBuffer insert(int offset,String str):

在调用者的字符串索引应该排位的顺序插入指定字符串:

在指定位置把任意类型的数据插入到字符串缓冲区里面,并返回字符串缓冲区本身

总结规律发现:只要在字符串引号和数字或者数字与数字之间的空隙插入都没问题,过了就报字符串索引越界

代码演示如下:

```
public static void main(String[] args) {
    //demo1();
    StringBuffer sb = new StringBuffer("1234");
    sb.insert(3, "heima");//在指定位置包括长度值添加元素,如果没有指定位置的索引,就会报字符串索引越界异常
    //sb.insert(4, "heima");//4还是可以插入的,插入到最后,输出1234heima,但是5就报错了
    System.out.println(sb);//123heima4,3应该排在1234的索引3位置,插入字符串heima

    //总结:只要在字符串引号和数字或者数字与数字之间的空隙插入都没问题,//如"1234"有5个空隙都可以插入,对应
    //0,1,2,3,4
    //也就是说这个方法可以在字符串的前后或者后面加上你指定的字符串,如果别人要你在一个字符串两端加上空格,懂?
}

private static void demo1() {
    StringBuffer sb = new StringBuffer();
    StringBuffer sb2 = sb.append(true);
    StringBuffer sb3 = sb.append("heima");
    StringBuffer sb4 = sb.append(100);

    //不跟着输出,后面再一起输出,会输出一样的结果,因为四个引用指向的是同一个对象,如下所示:
    System.out.println(sb.toString());//trueheima100,StringBuffer类中重写了toString方法,显示的是对象
    中的属性值
    System.out.println(sb2.toString());//trueheima100
    System.out.println(sb3.toString());//trueheima100
    System.out.println(sb4.toString());//trueheima100//说明四个引用,都是指向同一个对象
}
```

## 13.04\_常见对象(StringBuffer的删除功能)

A:StringBuffer的删除功能:

## public StringBuffer deleteCharAt(int index):

删除指定位置的字符,并返回字符串缓冲区本身

## public StringBuffer delete(int start,int end):

删除从指定位置开始到位置结束的内容,不包含末尾,并返回字符串缓冲区本身,神龙见首不见尾!

代码演示如下:

```
public static void main(String[] args) {
    StringBuffer sb = new StringBuffer();
    //sb.deleteCharAt(5);//当缓冲区中这个索引上没有元素的时候就会报StringIndexOutOfBoundsException
    sb.append("heima");
    sb.deleteCharAt(4);//heim根据索引删除掉索引位置上对应的字符

    //sb.delete(0, 2);//ima,删除的时候是包含头,不包含尾,神龙见首不见尾!

    //System.out.println(sb);
    //sb.delete(0, sb.length());//清空缓冲区
    System.out.println(sb);//sb.toString()转换成字符串

    //sb = new StringBuffer();//不要用这种方式清空缓冲区,原来的会变成垃圾,浪费内存
    //System.out.println(sb);
}
```

## 13.05\_常见对象(StringBuffer的替换和反转功能)

A:StringBuffer的替换功能:

## public StringBuffer replace(int start,int end,String str):

从start开始到end用str替换,也是包含头不包含尾,神龙见首不见尾

B:StringBuffer的反转功能:

## public StringBuffer reverse():

字符串反转,即把原出现的字符顺序倒转过来

代码演示如下:

```
public static void main(String[] args) {
    StringBuffer sb = new StringBuffer("我爱总复习");
    sb.replace(0, 3, "bai");//替换,也是包含头不包含尾,神龙见首不见尾
    System.out.println(sb);//bai复习

    sb.reverse();
    System.out.println(sb);//习复iab
}
```

## 13.06\_常见对象(StringBuffer的截取功能及注意事项)

A:StringBuffer的截取功能:

### public String substring(int start):

从指定位置开始截取到末尾

### public String substring(int start,int end):

截取从指定位置开始到结束位置,包括开始位置,不包括结束位置,神龙见首不见尾!

B:注意事项:

注意:返回值类型不再是StringBuffer本身,而是返回字符串!!!

代码演示如下:

```
public static void main(String[] args) {
    StringBuffer sb = new StringBuffer("woaiheima");
    String str = sb.substring(4);//从4索引开始截取到最后,返回值变成了String类型,而不再是StringBuffer类型
    System.out.println(str);//heima

    System.out.println(sb);//woaiheima,sb没有变

    String str3 = sb.substring(4, 7);//截取包含4不包含7那段的字符串,神龙见首不见尾!
    System.out.println(str3);//hei
    System.out.println(sb);//woaiheima,sb没有变
}
```

## 13.07\_常见对象(StringBuffer和String的相互转换)

A:String -- StringBuffer

a:通过构造方法  
b:通过append()方法

B:StringBuffer -- String

a:通过构造方法  
b:通过toString()方法  
c:通过substring(0,length);

代码演示如下:

```
public static void main(String[] args) {  
    //String转StringBuffer  
    //demo1();  
  
    //StringBuffer转String  
    StringBuffer sb = new StringBuffer("heima");  
  
    String s1 = new String(sb);//通过有参构造将StringBuffer转换为String  
    System.out.println(s1);//heima  
  
    String s2 = sb.toString();//通过toString方法将StringBuffer转换为String  
    System.out.println(s2);//heima  
  
    String s3 = sb.substring(0, sb.length());//通过截取子字符串将StringBuffer转换为String  
    System.out.println(s3);//heima  
}  
  
private static void demo1() {  
    StringBuffer sb1 = new StringBuffer("heima");//通过构造方法将字符串转换为StringBuffer对象  
    System.out.println(sb1);//heima  
  
    StringBuffer sb2 = new StringBuffer();  
    sb2.append("heima");//通过append方法将字符串转换为StringBuffer对象  
    System.out.println(sb2);//heima  
}
```

总结:再次强调StringBuffer的截取功能返回的是String字符串,而不再是像其他大多数方法一样返回的StringBuffer本身

## 13.08\_常见对象(把数组转成字符串)

A:案例演示:

需求:把数组中的数据按照指定个格式拼接成一个字符串:

举例:

```
int[] arr = {1,2,3};
```

输出结果:

```
"[1, 2, 3]"
```

用StringBuffer的功能实现

代码演示如下:

```
public static void main(String[] args) {
    int[] arr = {1,2,3};
    System.out.println(arrayToString(arr));
}

public static String arrayToString(int[] arr) {
    StringBuffer sb = new StringBuffer();//创建字符串缓冲区对象
    sb.append("[");//将[添加到缓冲区

    //{1,2,3}
    for (int i = 0; i < arr.length; i++) { //遍历数组
        //sb.append(arr[i] + ", ");//拼接也是可以的,但底层会再创建一个StringBuffer对象,不推荐
        if(i == arr.length - 1) {
            sb.append(arr[i]).append("");//[1, 2, 3]
        }else {
            sb.append(arr[i]).append(", ");//[1, 2,
        }
    }

    return sb.toString();
}
```

## 13.09\_常见对象(字符串反转)

A:案例演示:

需求:把字符串反转

举例:键盘录入"abc"

输出结果:"cba"

用StringBuffer的功能实现

代码演示如下:

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);//创建键盘录入对象
    String line = sc.nextLine();//将键盘录入的字符串存储在line中

    StringBuffer sb = new StringBuffer(line);//将字符串转换为StringBuffer对象
    sb.reverse();//调用字符串缓冲区反转方法将缓冲区的内容反转

    System.out.println(sb.toString());//转成字符串输出
}
```

## 13.10\_常见对象(StringBuffer和StringBuilder的区别)

A:StringBuilder的概述:

通过查看API了解一下StringBuilder类

B:面试题:

String,StringBuffer,StringBuilder的区别:

StringBuffer和StringBuilder的区别:

StringBuffer是jdk1.0版本的,是线程安全的,效率低

StringBuilder是jdk1.5版本的,是线程不安全的,效率高

String和StringBuffer,StringBuilder的区别:

String是一个不可变的字符序列

StringBuffer,StringBuilder是可变的字符序列

## 13.11\_常见对象(String和StringBuffer分别作为参数传递)

A:形式参数问题:

String作为参数传递  
StringBuffer作为参数传递

B:案例演示:

String和StringBuffer分别作为参数传递问题:

代码演示如下:

```

public static void main(String[] args) {
    String s = "heima";
    System.out.println(s);//heima

    change(s);//s += "itcast"; //输出还是heima
    //基本数据类型的值传递,不改变其值//引用数据类型的值传递,改变其值,但是String类除外
    //String类虽然是引用数据类型,但是他当作参数传递时和基本数据类型是一样的
    System.out.println(s);//heima

    StringBuffer sb = new StringBuffer();
    sb.append("heima");
    System.out.println(sb);//heima

    change(sb);
    System.out.println(sb);//heimaitcast
}

public static void change(StringBuffer sb) {
    sb.append("itcast");
}

public static void change(String s) { //因为方法弹栈后,里面的s就消失了,"heimaitcast"变成了垃圾
    s += "itcast";//是因为没有定义变量接收新的字符串,输出的还是原来的 输出还是heima
}

```

## 13.12\_常见对象(数组高级冒泡排序原理图解)

A:画图演示:见截图

需求:

数组元素:{24, 69, 80, 57, 13}  
请对数组元素进行排序

冒泡排序:

相邻元素两两比较,大的往后放,第一次完毕,最大值出现在了最大索引处

总结:代码演示如下:



```

/*
冒泡排序:相邻元素两两比较,大的往后放,第一次完毕,最大值出现在了最大索引处
1,返回值类型,void
2,参数列表,int[] arr

    第一次:arr[0]与arr[1],arr[1]与arr[2],arr[2]与arr[3],arr[3]与arr[4]比较4次
    第二次:arr[0]与arr[1],arr[1]与arr[2],arr[2]与arr[3]比较3次
    第三次:arr[0]与arr[1],arr[1]与arr[2]比较2次
    第四次:arr[0]与arr[1]比较1次

*/
public static void bubbleSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) { //外循环只需要比较arr.length-1次,就可以了
        for (int j = 0; j < arr.length - 1 - i; j++) { //-1为了防止索引越界,-i为了提高效率
            if(arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}

```

总结:无非是双层for循环,内外循环判断条件都是数组长度-1,内循环还要多-外循环起始数i,里面相邻元素前一个大于后一个,  
才进行位置交换,这样就把大的往后放了,循环就是不断这样做,最终实现冒泡排序

## 13.13\_常见对象(数组高级冒泡排序代码实现)

A:案例演示:

数组高级冒泡排序代码:

## 13.14\_常见对象(数组高级选择排序原理图解)

A:画图演示:见截图:

需求:

数组元素:{24, 69, 80, 57, 13}

请对数组元素进行排序:

选择排序:

从0索引开始,依次和后面元素比较,小的往前放,第一次完毕,最小值出现在了最小索引处

总结:代码演示如下:

```

/*
选择排序:从0索引开始,依次和后面元素比较,小的往前放,第一次完毕,最小值出现在了最小索引处:
1,返回值类型void
2,参数列表int[] arr

    第一次:arr[0]分别与arr[1-4]比较,比较4次
    第二次:arr[1]分别与arr[2-4]比较,比较3次
    第三次:arr[2]分别与arr[3-4]比较,比较2次
    第四次:arr[3]与arr[4]比较,比较1次

*/
public static void selectSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) { //只需要比较arr.length-1次
        for (int j = i + 1; j < arr.length; j++) {
            if(arr[i] > arr[j]) {
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;

                //swap(arr,i,j); //可以抽取交换位置为一个功能方法
            }

        }

    }

}

private static void swap(int[] arr,int i,int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

```

总结:总结:无非是双层for循环,外循环判断条件为数组长度-1,内循环初始化变量j赋值为i+1,里面如果i索引对应的元素大于

j索引对应元素,才交换它们的位置即可-考点是找到它们的规律用双层for循环来表示

## 13.15\_常见对象(数组高级选择排序代码实现)

A:案例演示:

数组高级选择排序代码:

## 13.16\_常见对象(数组高级二分查找原理图解)

A:画图演示:见截图:

二分查找,联想到猜数字小游戏就知道是什么了,比如说猜数字1到100之间的数字,你猜50,不对,说小了,那你肯定猜50的一半即25,以此类推最终就能快速猜出数字,但是前提是你猜的数字要有序,同理,二分查找,前提:数组元素有序,这个条件一定要满足,否则无法实现二分查找!!!

## 13.17\_常见对象(数组高级二分查找代码实现及注意事项)

A:案例演示:

数组高级二分查找代码:

B:注意事项:

如果数组无序,就不能使用二分查找;  
因为如果你排序了,但是你排序的时候,已经改变了我最原始的元素索引

总结:代码演示如下:

```
/*
 * 二分查找
 * 1,返回值类型,int
 * 2,参数列表int[] arr,int value
 */
public static int getIndex(int[] arr, int value) {
    int min = 0;
    int max = arr.length - 1;
    int mid = (min + max) / 2;

    while(arr[mid] != value) {        //当中间值不等于要找的值,就开始循环查找
        if(arr[mid] < value) {        //当中间值小于了要找的值
            min = mid + 1;            //最小的索引改变
        }else if (arr[mid] > value){    //当中间值大于了要找的值
            max = mid - 1;            //最大的索引改变
        }

        mid = (min + max) / 2;        //无论最大还是最小改变,中间索引都会随之改变

        if(min > max) {                //如果最小索引大于了最大索引,就没有查找的可能性了
            return -1;                //返回-1,表示找不到,方法调用完毕跳出循环
        }
    }

    return mid;
}
```

总结:如果中间索引对应元素等于要查找的值就返回中间索引,否则一直循环,小了,最小索引就等于中间索引加1,大了最大索引就等于中间索引减1,改变后中间索引还是等于最大索引加最小索引之和的一半,如果最后最小索引,大于最大索引,那就是找不到了返回-1表示-其实说白了,有点类似于指针的移动或者游标的感觉

## 13.18\_常见对象(Arrays类的概述和方法使用)

A:Array类概述:

针对数组进行操作的工具类;  
提供了排序,查找等功能

B:成员方法:

**public static String toString(int[] a)**//将整型数组转成字符串,输出形式如  
**[33, 22, 11, 44, 66, 55]**

**public static void sort(int[] a)**//数组工具类的静态方法按从小到大即升序  
排序,底层是**快速排序**

**public static int binarySearch(int[] a,int key)**//二分查找指定值key在数  
组中的索引,前提是数组有序

代码演示如下:

```
public static void main(String[] args) {  
    int[] arr = {33,22,11,44,66,55};  
    System.out.println(Arrays.toString(arr));//[33, 22, 11, 44, 66, 55]数组转字符串,含中括号,逗号  
    和空格  
  
    Arrays.sort(arr);//数组工具类调用静态方法按从小到大即升序排序,底层是快速排序  
    System.out.println(Arrays.toString(arr));//[11, 22, 33, 44, 55, 66]  
  
    int[] arr2 = {11,22,33,44,55,66};  
    System.out.println(Arrays.binarySearch(arr2, 22));//1,二分查找22在数组中的索引,前提是数组有序  
    System.out.println(Arrays.binarySearch(arr2, 66));//5  
    System.out.println(Arrays.binarySearch(arr2, 9));//-1,找不到返回-插入点-1,9的插入点应该在0位  
    置                                返回-1:- (插入点)-1  
}
```

## 13.19\_常见对象(基本类型包装类的概述)

A:为什么会有基本类型包装类:

将基本数据类型封装成对象,好处在于可以在对象中定义更多的功能方法操作该数据

B:常用操作:

常用的操作之一:用于基本数据类型与字符串之间的转换

C:基本类型和包装类的对应:

byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

总结:特殊的是Integer和Character,其他的都是基本数据类型的首字母大写,其他字符小写  
代码演示如下:

```
public static void main(String[] args) {  
    System.out.println(Integer.toBinaryString(60)); //111100, 整数转成二进制字符串  
    System.out.println(Integer.toOctalString(60)); //74, 整数转成八进制字符串  
    System.out.println(Integer.toHexString(60)); //3c, 整数转成十六进制字符串  
}
```

## 13.20\_常见对象(Integer类的概述和构造方法)

A:Integer类概述:

通过JDK提供的API,查看Integer类的说明:  
Integer类在对象中包装了一个基本类型int的值,  
该类提供了多个方法,能在int类型和String类型之间互相转换,  
还提供了处理int类型时非常用的其他一些常量和方法

B:构造方法:

```
public Integer(int value)  
public Integer(String s) //这个构造要求传入数字字符串,而不是普通字符串!否则运行输出,会报数字格式化异常
```

C:案例演示:

使用构造方法创建对象:

代码演示如下:

```

public static void main(String[] args) {
    System.out.println(Integer.MAX_VALUE); // 2147483647 整型的最大值
    System.out.println(Integer.MIN_VALUE); // -2147483648 整型的最小值

    Integer i1 = new Integer(100); // 传入整型构造
    System.out.println(i1); // 100

    // Integer i2 = new Integer("abc"); // java.lang.NumberFormatException 数字格式异常
    // System.out.println(i2); // 编译没问题, 运行报数字格式化异常: 因为abc不是数字字符串, 所以转换会报错

    Integer i3 = new Integer("100"); // 传入数字字符串构造
    System.out.println(i3); // 100
}

```

## 13.21\_常见对象(String和int类型的相互转换)


A: `int -- String`   `int`转换为`String`

a: 和""进行拼接  
b: `public static String valueOf(int i)`  
c: `int -- Integer -- String(Integer类的toString()方法)`  
d: `public static String toString(int i)(Integer类的静态方法)`

代码演示如下:

```

public static void main(String[] args) {
    int i = 100;
    String s1 = i + ""; // 100, 推荐用
    System.out.println(s1);
    String s2 = String.valueOf(i); // 100, 推荐用
    System.out.println(s2);

    Integer i2 = new Integer(i);   int转为Integer
    String s3 = i2.toString(); // 100, 整型包装类对象的toString方法
    System.out.println(s3);
    
    String s4 = Integer.toString(i); // 100, 整型包装类的静态toString方法
    System.out.println(s4);
}

```

B: `String -- int`   `String`转换为`int`

a: `String -- Integer -- int`  
`public static int parseInt(String s)` // 要求传入的字符串是数字字符串, 跟`public Integer(String s)`完全一样

代码演示如下:

```

public static void main(String[] args) {
    String s = "200";
    Integer i3 = new Integer(s);
    int i4 = i3.intValue();//200,Integer对象的整型值方法转换成了int数
    System.out.println(i4);

    int i5 = Integer.parseInt(s);//200,Integer的静态解析整型方法,将String转换为int,推荐用这种
    System.out.println(i5);//注意,此解析方法要求传入的字符串是数字字符串,否则运行时报数字格式化异常

    //另外,基本数据类型包装类有八种,其中七种都有parseXxx的方法,只有Character没有,擦,巧记!
    String s1 = "true";
    boolean b = Boolean.parseBoolean(s1);
    System.out.println(b);//true

    //String s2 = "abc";//字符串到字符的转换,通过toCharArray()就可以把字符串转换为字符数组,
    //char c = Character.p//char的包装类Character中没有pareseXxx的方法
}

```

将这七种的字符串表现形式转换为基本数据类型

## 13.22\_常见对象(JDK5的新特性自动装箱和拆箱)

A:JDK5的新特性:

自动装箱:把基本类型转换为包装类类型  
 自动拆箱:把包装类类型转换为基本类型

B:案例演示:

JDK5的新特性自动装箱和拆箱:  
 Integer ii = 100;//基本类型转换为包装类类型,自动装箱  
 ii += 200;//把包装类类型转换为基本类型,自动拆箱,底层调用了包装类对象的intValue方法,所以ii不能为null

C:注意事项:

在使用时,Integer x = null;代码就会出现NullPointerException,自动拆箱底层调用了包装类对象的intValue方法,建议先判断是否为null,然后再使用

看下面代码就明白了:

```

public static void main(String[] args) {
    //int x = 100;
    //Integer i1 = new Integer(x);//将基本数据类型包装成对象,装箱

    //int y = i1.intValue();//将对象转换为基本数据类型,拆箱

    Integer i2 = 100;//自动装箱,底层调用了Integer i2 = Integer.valueOf(100);把基本数据类型转换成对象
    int z = i2 + 200;//自动拆箱,底层调用了int z = i2.intValue() + 200;把对象转换为基本数据类型
    System.out.println(z);//300

    Integer i3 = null;//编译没问题,运行时异常
    int a = i3 + 100;//底层用i3调用intValue,不信?看看字节码文件就知道了,所以当i3是null时调用方法就会出现,
    System.out.println(a);//空指针异常java.lang.NullPointerException

    //上面的代码反编译字节码文件查看底层调用如下:
    /*
    public class Demo4_JDK5 {
        public static void main(String[] args) {
            Integer i2 = Integer.valueOf(100);
            int z = i2.intValue() + 200;
            System.out.println(z);

            Integer i3 = null;
            int a = i3.intValue() + 100;
            System.out.println(a);
        }
    }
    */
}

```

总结:自动拆箱,底层调用了包装类对象的intValue方法,如int z = i2 + 200;底层调用的是int z = i2.intValue() + 200;所以,对象i2不能赋值为null再调用方法,否则会报运行时异常的空指针异常,但是编译符合语法是通过的,这个要区别开哦!

## 13.23\_常见对象(Integer的面试题)

A:Integer的面试题:

看程序写结果:



```
Integer i1 = new Integer(97);
Integer i2 = new Integer(97);
System.out.println(i1 == i2); //false
System.out.println(i1.equals(i2)); //true
System.out.println("-----");

Integer i3 = new Integer(197);
Integer i4 = new Integer(197);
System.out.println(i3 == i4); //false
System.out.println(i3.equals(i4)); //true, 比较属性值
System.out.println("-----");

Integer i5 = 97; 127
Integer i6 = 97; 127
System.out.println(i5 == i6); //true, 哈哈自动装箱, 真相只有一个, 在字节范围内就使用, 否则创建对象!
System.out.println(i5.equals(i6)); //true
System.out.println("-----");

Integer i7 = 197; 128
Integer i8 = 197; 128
System.out.println(i7 == i8); //false
System.out.println(i7.equals(i8)); //true
```

总结:这里凡是**new**出来的地址值都不同,另外如果是**自动装箱**,基本类型值范围在**byte**字节范围内**-128到+127**,就**不用创建对象**,直接使用**byte**常量池的值,地址值相同,否则要创建新对象,地址值不同,而**equals**方法重写了,比较的是内容,所以都相等

看下面的代码就明白了:

```

public static void main(String[] args) {
    Integer i1 = new Integer(97);
    Integer i2 = new Integer(97);
    System.out.println(i1 == i2);           //false
    System.out.println(i1.equals(i2));      //true
    System.out.println("-----");

    Integer i3 = new Integer(197);
    Integer i4 = new Integer(197);
    System.out.println(i3 == i4);           //false
    System.out.println(i3.equals(i4));      //true
    System.out.println("-----");

    Integer i5 = 127; //自动装箱底层调用的是valueOf方法, Integer i5 = Integer.valueOf(127);
    Integer i6 = 127; //自动装箱底层调用的是valueOf方法, Integer i6 = Integer.valueOf(127);
    System.out.println(i5 == i6);           //true, 真相只有一个, 在字节范围内就使用, 否则创建对象!
    System.out.println(i5.equals(i6));      //true
    System.out.println("-----");

    Integer i7 = 128;
    Integer i8 = 128;
    System.out.println(i7 == i8);           //false
    System.out.println(i7.equals(i8));      //true
    /*
    * -128到127是byte的取值范围, 如果在这个取值范围内, 自动装箱就不会新创建对象, 而是从常量池中获取,
    * 如果超过了byte取值范围就会再新创建对象,
    * 看自动装箱底层调用的valueOf方法的源码就知道了, 如下所示:↓
    * public static Integer valueOf(int i) {
    *     assert IntegerCache.high >= 127;
    *     if (i >= IntegerCache.low && i <= IntegerCache.high) // i >= -128 && i <= 127, 字节范围内,
    *         return IntegerCache.cache[i + (-IntegerCache.low)]; // 满足条件, 从常量池获取
    *
    *     return new Integer(i); // 否则, 创建新对象!!!
    * }
    */
}

```

**总结:自动装箱底层调用的是valueOf方法,如Integer i6 = 127;底层调用的是Integer i6 = Integer.valueOf(127);**

对于valueOf方法看其源码知道,如果传入参数的整型值在byte字节取值范围-128到+127内,就直接使用常量池中的值,不会创建新对象,否则,如果超过了byte取值范围,就会再新创建对象,这会作为面试题考查,其实考的是你阅读源码的能力!

## 13.24\_day13总结

把今天的知识点总结一遍：

**StringBuffer**类-数组高级冒泡排序-数组高级选择排序-数组高级二分查找-**Arrays**数组工具类-基本类型包装类-**String**和**int**类型的相互转换-JDK5的新特性自动装箱和拆箱-自动装箱底层调用方法**valueOf**的特别考查面试题**byte**常量池

