常用类和基础 API

1、String 的特性

1> **java.lang.String 类**代表字符串。Java 程序中所有的字符串文字（例如"hello" ）都可以看作是实现此类的实例；

2> 字符串是**常量**，用双引号引起来表示。它们的值在创建之后**不能更改**；

3> 字符串 String 类型本身是 **final 声明**的，意味着我们**不能继承 String**；

4> String 对象的**字符内容**是**存储**在一个**字符数组 value[]** 中的。"abc" 等效于 char[] data={'h','e','l','l','o'}；

5> 一旦对字符串进行修改，就会产生新对象；

6> Java 语言提供对字符串**串联符号（"+"）**以及将**其他对象转换为字符串**的特殊支持（**toString()**方法）。

2、类的声明

public final class String implements java.io.Serializable, Comparable<String>, CharSequence

1）final：String 是不可被继承的；

2）Serializable：可序列化的接口，凡是实现此接口的类的对象就可以通过网络或本地流进行数据的传输；

3）Comparable：凡是实现此接口的类，其对象都可以比较大小。

3、内部声明的属性

jdk8 中：

private final **char** value[];//存储字符串数据

>final：指明此 value 数组一旦初始化，其地址就不可变；

jdk9 开始：为了节省内存空间，做了优化

private final **byte** value[];//存储字符串数据

4、字符串常量的存储位置

1）字符串常量都存储在(StringTable)字符串常量池中；

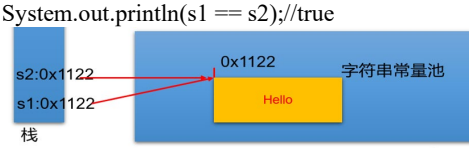
2）字符串常量池不允许存放两个相同的字符串常量；

3）字符串常量池，在不同的 jdk 版本中，存放位置不同，jdk7 之前字符串常量池存放在方法区；jdk7 及之后，字符串常量池存放在堆空间中。（堆空间 GC 回收更及时）

5、String 的不可变性

1）当对字符串变量**重新赋值时**，需要**重新指定一个字符串常量的位置**赋值，不能在原有位置修改；

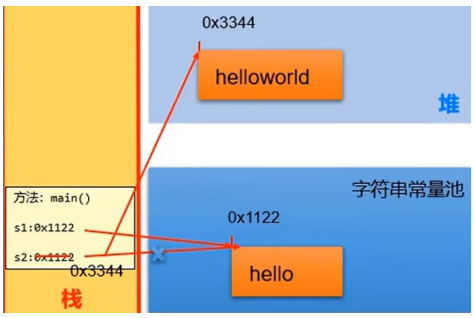
```
String s1 = "hello";
String s2 = "hello";
System.out.println(s1 == s2);//true
```




```
String s1 = "hello";
String s2 = "hello";
s1 = "hi";
```

2）当对现有的字符串进行**拼接操作**时，需要**重新开辟空间**保存拼接以后的字符串，不能在原有位置修改；

```
String s1 = "hello"; String s2 = "hello";
s2 += "world";
System.out.println(s1);//hello
System.out.println(s2);//helloworld
```



3）当调用字符串的 replace()替换现有的某个字符时，需要重新开辟空间保存修改以后的字符串，不能在原有位置修改；

```
String s1 = "hello"; String s2 = "hello";
String s3 = s2.replace('l', 'w');
System.out.println(s1);//hello
System.out.println(s2);//hello
System.out.println(s3);//hewwo
```

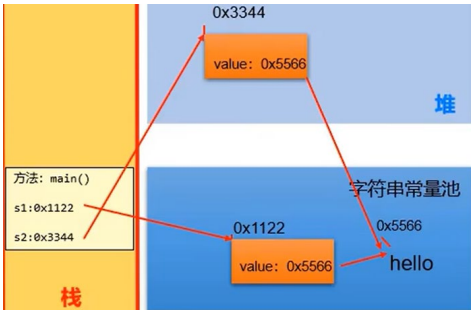
（replace 中新 new 了一个 String 对象）

6、String 实例化的两种方式

字符串常量存在**字符串常量池**，目的是共享。**字符串非常量对象**存储在**堆**中。s3 首先指向堆中的一个字符串对象，然后堆中字符串的 value 数组指向常量池中常量对象的 value 数组。

```
String s1 = "javaEE";
String s2 = "javaEE";
String s3 = new String("javaEE");
String s4 = new String("javaEE");
System.out.println(s1 == s2);//true
System.out.println(s1 == s3);//false
System.out.println(s1 == s4);//false
System.out.println(s3 == s4);//false
System.out.println(s1.equals(s2));//true
```

（s1.equals(s2)每个字符比较，都是 true）

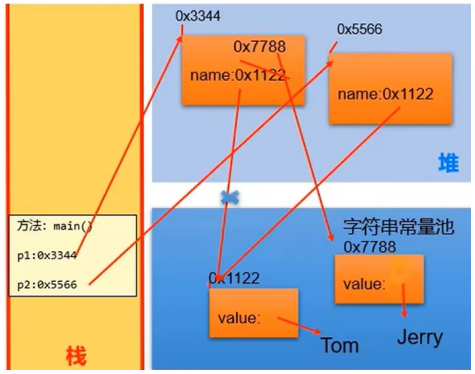


7、String str2 = new String("hello"); 在内存中**创建了几个对象**？

两个。一个是堆空间中 new 的对象；另一个是在字符串常量池中生成的字面量。

8、在对象属性中

```
Person p1 = new Person();
p1.name = "Tom";
Person p2 = new Person();
p2.name = "Tom";
p1.name = "Jerry";
System.out.println(p1.name); // Jerry
System.out.println(p2.name); // Tom
```



9、拼接

1）常量+常量：结果仍然存在字符串常量池中，返回此字面量的地址。注：此时的常量可能是字面量，也可能是 final 修饰的常量。

2）常量+变量 或 变量+变量：都会通过 new 的方式创建一个新的字符串，返回堆空间中此字符串对象的地址；

3）拼接后调用 intern 方法：返回的是字符串常量池中字面量的地址；

4）concat()：不管是常量调用此方法，还是变量调用，同样不管参数是常量还是变量，总之，调用完 concat()方法都返回一个新 new 的对象。

```
String s1 = "hello";
String s2 = "world";
String s3 = "helloworld";
String s4 = "hello" + "world";
String s5 = s1 + "world";//调用了 StringBuilder 的 toString()，new String()
String s6 = "hello" + s2;
String s7 = s1 + s2;
String s8 = s5.intern();
System.out.println(s3 == s4);//true
System.out.println(s3 == s5);//false
System.out.println(s3 == s6); //false
System.out.println(s3 == s7); //false
System.out.println(s3 == s8);//true
final String s1 = "hello";//final 修饰的常量
System.out.println(s3 == s5);//true
String str4 = "hello".concat("world");
String str5 = "hello"+"world";
System.out.println(str3 == str4);//false
System.out.println(str3 == str5);//true
```

10、值传递和 String 不可变性

```
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
        }
}
```

11、String 的构造器

- **public String()**：初始化新创建的 String 对象，以使其表示**空字符序列**。""
- **String(String original)**：初始化一个新创建的 String 对象，使其表示一个与参数相同的字符序列；换句话说，**新创建的字符串**是**该参数字符串**的**副本**。
- **public String(char[] value)**：通过当前参数中的**字符数组**来构造新的 String。
- **public String(char[] value,int offset, int count)**：通过**字符数组的一部分**来构造新的 String。
- **public String(byte[] bytes)**：通过使用**平台的默认字符集*解码***当前参数中的**字节数组**来构造新的 String。
- public String(byte[] bytes,String charsetName)：通过使用**指定的字符集*解码***当前参数中的**字节数组**来构造新的 String。

```
//字面量定义方式：字符串常量对象
String str = "hello";
//构造器方式：无参构造
String str1 = new String();//等价 new String("")
//构造器方式：创建"hello"字符串常量的副本
String str2 = new String("hello");
//构造器方式：通过字符数组构造
char chars[] = {'a', 'b', 'c','d','e'};
```

String str3 = new String(chars);
String str4 = new String(chars,0,3);
//构造器方式：通过字节数组构造
byte bytes[] = {97, 98, 99 };
String str5 = new String(bytes);
String str6 = new String(bytes,"GBK");

12、String 与其他结构间的转换

1)字符串→基本数据类型、包装类：(parseXxx)

1>Integer 包装类的 public static int **parseInt**(String s)：可以将由“数字”字符组成的字符串转换为整型；

2>类似地，使用 java.lang 包中的 Byte、Short、Long、Float、Double 类调相应的类方法可以将由“数字”字符组成的字符串，转化为相应的基本数据类型。

2)基本数据类型、包装类→字符串：valueOf(x)

1>调用 String 类的 public String valueOf(int n)可将 int 型转换为字符串；

2>相应的 valueOf(byte b)、valueOf(long l)、valueOf(float f)、valueOf(double d)、valueOf(boolean b)可由参数的相应类型到字符串的转换。

3）字符数组→字符串：

1>String 类的**构造器：String(char[])** 和 **String(char[]，int offset，int length)** 分别用字符数组中的**全部字符**和**部分字符**创建字符串对象。

2> static String **copyValueOf(char[] data)**：返回指定数组中表示该字符序列的 String；

static String **copyValueOf(char[] data, int offset, int count)**：返回指定数组中表示该字符序列的 String

4）字符串→字符数组：

1>public char[] **toCharArray()**：将字符串中的**全部字符存放在一个字符数组**中的方法；

2>public void **getChars**(int srcBegin, int srcEnd, char[] dst, int dstBegin)：提供了将**指定索引范围内**的字符串**存放到数组**中的方法。

5）字符串→字节数组：（编码）

1>public 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；

2>public byte[] **getBytes**(String charsetName)：使用**指定的字符集**将此 String 编码到 **byte 序列**，并将结果**存储到新的 byte 数组**。

6）字节数组→字符串：（解码）

1>**String(byte[])**：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；

2>**String(byte[]，int offset，int length)**：用**指定的字节数组的一部分**，即从数组起始位置 offset 开始取 length 个字节构造一个字符串对象；

3>**String(byte[], String charsetName)**或 **String(byte[], int, int,String charsetName)**：解码，按照**指定的编码方式**进行解码。

【补充】

1）不乱码：1>保证编码与解码的**字符集名称**一样；2>**不缺字节**。

2）GBK 字符集，一个汉字占用 2 个字节，一个字母占用 1 个字节；UTF-8 字符集，一个汉字占用 3 个字节，一个字母占用 1 个字节。

3）GBK 或 UTF-8 都向下兼容 ASCII 码。

13、String 常用方法

1）boolean **isEmpty()**：字符串是否为空；

2）int **length()**：返回字符串的长度；

3）String **concat(xx)**：拼接；

4）boolean **equals(Object obj)**：比较字符串是否相等，**区分大小写**；

5）boolean **equalsIgnoreCase**(Object obj)：比较字符串是否相等，**不区分大小写**；

6）int **compareTo(String other)**：比较字符串大小，**区分大小写**，按照 **Unicode 编码值**比较大小；

7）int **compareToIgnoreCase(String other)**：比较字符串大小，**不区分大小写**；

8）String **toLowerCase()**：将字符串中**大写字母转为小写**；

9）String **toUpperCase()**：将字符串中**小写字母转为大写**；

10）String **trim()**：**去掉**字符串**前后空白符**；

11）public String **intern()**：结果在常量池共享。

14、String 查找方法

1）boolean **contains(xx)**：是否包含 xx；

2）int **indexOf(xx)**：从前往后找当前字符串中 xx，即如果**有返回第一次出现的下标**，要是**没有返回-1**；

3）int **indexOf(String str, int fromIndex)**：返回指定子字符串在此字符串中第一次出现处的索引，从指定的索引开始；

4）int **lastIndexOf(xx)**：从后往前找当前字符串中 xx，即如果**有返回最后一次出现的下标**，要是**没有返回-1**；

5）int **lastIndexOf(String str, int fromIndex)**：返回指定子字符串在此字符串中最后一次出现处的索引，从指定的索引开始**反向搜索**。

15、字符串截取方法

1）String **substring(int beginIndex)**：返回一个新的字符串，它是此字符串的**从 beginIndex 开始**截取**到最后**的一个子字符串；

2)String **substring**(int beginIndex, int endIndex)：返回一个新字符串，它是此字符串从 **beginIndex** 开始截取到 **endIndex(不包含)**的一个子字符串。（**前闭后开**）

//截取文件名

fileName.substring(0,fileName.lastIndexOf("."))

//截取后缀名

fileName.substring(fileName.lastIndexOf("."))

16、和字符/字符数组相关

1）char **charAt(index)**：返回[index]位置字符；

2）char[] **toCharArray()**：将此字符串转换为一个新的字符数组返回；

3）static String **valueOf**(char[] data)：返回指定数组中表示该字符序列的 String；

4）static String **valueOf**(char[] data, int offset, int count)：返回指定数组中表示该字符序列的 String；

5）static String **copyValueOf**(char[] data)：返回指定数组中表示该字符序列的 String；

6）static String **copyValueOf**(char[] data, int offset, int count)：返回指定数组中表示该字符序列的 String。

17、开头与结尾

1）boolean **startsWith(xx)**：测试此字符串是否**以指定的前缀开始**；

2）boolean **startsWith(String prefix, int toffset)**：测试此字符串从**指定索引开始**的子字符串是否**以指定前缀开始**；

3）boolean **endsWith(xx)**：测试此字符串是否**以指定的后缀结束**。

18、替换

1）String **replace**(char oldChar, char newChar)：返回一个新的字符串，它是通过用 newChar 替换此字符串中出现的所有 oldChar 得到的。**不支持正则**；

2）String **replace**(CharSequence target, CharSequence replacement)：使用**指定的字面值序列**替换此字符串**所有匹配字面值目标序列**的子字符串；

3）String **replaceAll**(String regex, String replacement)：使用**给定的 replacement** 替换此字符串**所有匹配给定的正则表达式**的子字符串；

4）String **replaceFirst**(String regex, String replacement)：使用给定的 replacement 替换此字符串**匹配给定的正则表达式**的**第一个**子字符串。

19、String/StringBuffer/StringBuilder 对比

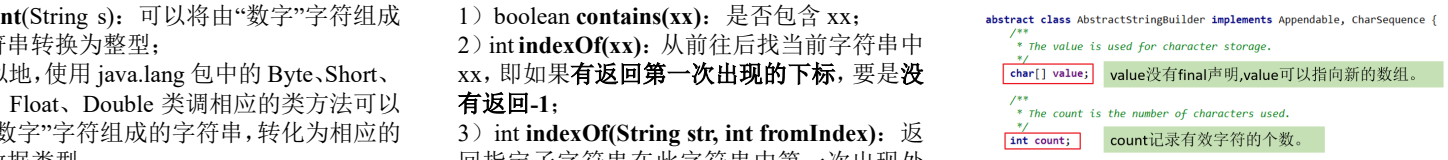
String：不可变的字符序列；

StringBuffer：可变的字符序列；jdk1.0 声明，线程安全的（方法有 synchronized 修饰），效率低；

StringBuilder：可变的字符序列；jdk5.0 声明，线程不安全的，效率高；

可以对字符串内容增删，且不会产生新对象。

20、StringBuffer/StringBuilder 的可变性



String s1 = new String();
//char[] value =new char[0];
String s2 = new String("abc");
//char[] value =new char[]{‘a’, ‘b’, ‘c’};

针对 StringBuilder 来说，

内部属性有：

char[] value;//存储字符序列

int count;//实际存储的字符个数

StringBuilder s1 = new StringBuilder();
//char[] value = new char[16];
StringBuilder s2 = new StringBuilder("abc");
//char[] value = new char[16 + "abc".length];
s1.append("ac")//value[0] = ‘a’, value[1] = ‘c’

…不断添加…

一旦 count 要超过 value.length 时，就需要扩容，**默认**扩容为**原有容量的 2 倍+2**。并将原有 value 数组中的元素复制到新的数组中。

【源码启示】

1）如果开发中需要频繁的针对字符串进行增、删、改等操作，建议使用 StringBuffer 或 StringBuilder 替换 String；

2）如果开发中不涉及到线程安全问题，建议使用 StringBuilder 替换 StringBuffer，因为使用 StringBuilder 效率高；

3）如果开发中大体确定要操作的字符个数，建议使用带 int capacity 参数的构造器，因为可以避免底层多次扩容操作。

21、StringBuffer/StringBuilder 中常用方法

1）StringBuffer **append(xx)**：提供了很多的 append()方法，用于字符串追加的方式拼接；

2）StringBuffer **delete**(int start, int end)：删除[start,end)**之间字符**；

3）StringBuffer **deleteCharAt**(int index)：删除[index]位置**字符**；

4）StringBuffer **replace**(int start, int end, String str)：替换[start,end)范围的字符序列为 str；

5)void **setCharAt**(int index, char c)：替换[index]位置字符；

6）char **charAt(int index)**：**查找**指定 index 位置上的字符；

7）StringBuffer **insert**(int index, xx)：在**[index]位置**插入 xx；

8）int **length()**：返回实际存储字符数据长度；

9）StringBuffer **reverse()**：反转；

【注意】当 append 和 insert 时，如果原来 value 数组长度不够，可扩容。如上(1)(2)(3)(4)(9)这些方法支持方法链操作。原理：有 return this;

10）void **setLength**(int newLength)：设置当前字符序列长度为 newLength；

StringBuilder s = new StringBuilder("hello");
s.setLength(2);
System.out.println(s);//he
//此时 count=2,只显示 he，llo 还在内存中但是不显示，相当于已经删除了
s.append("c");//hec,即 l 的位置被 c 覆盖
s.setLength(10);
System.out.println(s);//hec 后面跟着‘\0’来赋值

22、效率：StringBuilder >StringBuffer>String

23、日期时间 API(JDK8 之前)

1）**System 类**的 **currentTimeMillis()**：

1>获取当前时间对应的毫秒数，long 类型，时间戳；

2>当前时间与 1970 年 1 月 1 日 0 时 0 分 0 秒之间的毫秒数；

3>常用来计算时间差。

2）java.util.Date

表示特定的瞬间，精确到毫秒。

1> 构造器：

- **Date()**：使用**无参构造器**创建一个基于当前本地时间的 Date 的实例；
- **Date(long 毫秒数)**：创建一个基于指定时间戳事物 Date 的实例。

2> 方法

- **getTime()**：返回自 1970 年 1 月 1 日 00:00:00 GMT 以来 Date 对象表示的毫秒数；
- **toString()**：把此 Date 对象转换为以下形式的 String： dow mon dd hh:mm:ss zzz yyyy 其中： dow 是一周中的某一天 (Sun, Mon, Tue, Wed, Thu, Fri, Sat)，zzz 是时间标准；
- 其它很多方法都过时了。

3）java.sql.Date

对应着数据库中的 date 类型；只有一个带参构造器 **Date(long 毫秒数)**：创建一个基于指定时间戳事物 Date 的实例。

4）java.text.SimpleDateFormat

一个不与语言环境有关的方式来格式化和解析日期的具体类。

1> 构造器：

- **SimpleDateFormat()**：默认的模式和语言环境创建对象；
- public **SimpleDateFormat(String pattern)**：该构造方法可以用参数 pattern 指定的格式创建一个对象；

2> 格式化：

- public String **format(Date date)**：方法格式化时间对象 date；

3> 解析：

- public Date **parse(String source)**：从给定字符串的开始解析文本，以生成一个日期。

```
//格式化
@Test
public void test1(){
    Date d = new Date();
   SimpleDateFormat sf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    //把Date 日期转成字符串，按照指定的格式转
    String str = sf.format(d);
    System.out.println(str);
}
//解析
@Test
public void test2() throws ParseException{
    String str = "2022-06-06 16:03:14";
   SimpleDateFormat sf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    Date d = sf.parse(str);
    System.out.println(d);
}
```

5) java.util.Calendar(日历类)

1>实例化：由于 Calendar 是一个抽象类，所以我们需要创建其子类的实例，通过 Calendar 的静态方法 **getInstance()**即可获取。

Calendar c = Calendar.getInstance();

2>常用方法

- public int **get**(int field)：返回给定日历字段的值；
- public void **set**(int field,int value) ：将给定的日历字段设**置为指定的值**；
- public void **add**(int field,int amount)：根据日历的规则，为给定的日历字段**添加或者减去**指定的时间量；
- public final Date **getTime()**：将 Calendar 转成 Date 对象；
- public final void **setTime**(Date date)：使用指定的 Date 对象**重置 Calendar 的时间**。

3>常用字段

- 获取**月份**时：一月是 0，二月是 1，以此类推，12 月是 11；- 获取**星期**时：周日是 1，周二是 2…周六是 7。

字段值	含义
YEAR	年
MONTH	月（从0开始，可以+1使用）
DAY_OF_MONTH	月中的天（几号）
HOUR	时（12小时制）
HOUR_OF_DAY	时（24小时制）
MINUTE	分
SECOND	秒
DAY_OF_WEEK	周中的天（周几，周日为1，可以+1使用）

```
public void test1(){
    Calendar c = Calendar.getInstance();
    System.out.println(c);

    int year = c.get(Calendar.YEAR);
    int month = c.get(Calendar.MONTH)+1;
    int day = c.get(Calendar.DATE);
    int hour = c.get(Calendar.HOUR_OF_DAY);
    int minute = c.get(Calendar.MINUTE);

    System.out.println(year + "-" + month + "-" + day + " " + hour + ":" + minute);
}

@Test
public void test3(){
    Calendar calendar = Calendar.getInstance();
    // 从一个 Calendar 对象中获取 Date 对象
    Date date = calendar.getTime();
    // 使用给定的 Date 设置此 Calendar 的时间
    date = new Date(234234235235L);
    calendar.setTime(date);
    calendar.set(Calendar.DAY_OF_MONTH, 8);
System.out.println("当前时间日设置为 8 后,时间是:" + calendar.getTime());
    calendar.add(Calendar.HOUR, 2);
System.out.println("当前时间加 2 小时后,时间是:" + calendar.getTime());
    calendar.add(Calendar.MONTH, -2);
System.out.println("当前日期减 2 个月后,时间是:" + calendar.getTime());
}
}
```

24、日期时间 API(JDK8 中)

1）JDK8 之前日期时间 API 面临的问题：

1>可变性：像日期和时间这样的类应该是不可变的；

2>偏移性：Date 中的年份是从 1900 开始的，而月份都从 0 开始。

3>格式化：格式化只对 Date 有用，Calendar 则不行。

4>线程不安全的；不能处理闰秒等。

2）新的日期时间 API 包含

1>java.time：包含值对象的基础包；

2>java.time.chrono：提供对不同日历系统的访问；

3>java.time.format：格式化和解析时间和日期；

4>java.time.temporal：包括底层框架和扩展特性；

5>java.time.zone：包含时区支持的类。

25、本地日期时间：LocalDate、LocalTime、LocalDateTime

1）实例化：

now()/ now(ZoneId zone)：静态方法，根据**当前时间**创建对象/指定时区的对象；

LocalDate now = LocalDate.now();

of(xxxx,xx,xx,xx,xx,xx)：静态方法，根据**指定日期/时间**创建对象。

LocalDate lai = LocalDate.of(2019, 5, 13);

LocalDate go = lai.plusDays(160);// 2019-10-20

2）常用方法：get/with/plus/minus

getDayOfMonth()/getDayOfYear()：获得月份天数(1-31) /获得年份天数(1-366)；

getDayOfWeek()：获得**星期几**(返回一个 DayOfWeek **枚举值**)；

getMonth()：获得月份，返回一个 **Month 枚举值**；

getMonthValue()/getYear()：获得月份(1-12) /获得年份；

getHours()/getMinute()/getSecond()：获得当前对象对应的小时、分钟、秒；

withDayOfMonth()/withDayOfYear()/withMonth()/withYear()：将月份天数、年份天数、月份、年份**修改为指定的值**并**返回新的对象**；

with(TemporalAdjuster t)：将当前日期时间设置为**校对器指定的日期时间**；

```
// 获取当前日期的下一个周日是哪天？
TemporalAdjuster temporalAdjuster = TemporalAdjusters.next(DayOfWeek.SUNDAY);
LocalDateTime localDateTime = LocalDateTime.now().with(temporalAdjuster);
```

plusDays(),plusWeeks(),plusMonths(),plusYears(),plusHours()：向当前对象**添加**几天、几周、几个月、几年、几小时；

minusMonths()/minusWeeks()/minusDays()/minusYears()/minusHours()：从当前对象**减去**几月、几周、几天、几年、几小时；

plus(TemporalAmount t)/minus(TemporalAmount t)：添加或减少一个 Duration 或 Period；

isBefore()/isAfter()：比较两个 LocalDate；

isLeapYear()：判断是否是闰年（在 LocalDate 类中声明）；

format(DateTimeFormatter t)：格式化本地日期、时间，返回一个字符串；

parse(Charsequence text)：将指定格式的字符串解析为日期、时间。

26、瞬时：Instant（时间戳）

java.time.Instant 表示时间线上的一点，可以被用来记录应用程序中的事件时间戳。

1）实例化：

now()：静态方法，返回默认 UTC 时区的 Instant 类的对象；

ofEpochMilli(long epochMilli)：静态方法，返回在 1970-01-01 00:00:00 基础上加上指定毫秒数之后的 Instant 类的对象。

2）常用方法：

atOffset(ZoneOffset offset)：结合即时的偏移来创建一个 OffsetDateTime；

toEpochMilli()：返回 1970-01-01 00:00:00 到当前时间的毫秒数，即为时间戳。

Instant instant = Instant.now();//实例化

OffsetDateTime instant1 = instant.atOffset(ZoneOffset.ofHours(8))

long milliTime = instant.toEpochMilli();

27、日期时间格式化：DateTimeFormatter

(了解)**预定义的标准格式**：如：ISOLOCALDATETIME、ISOLOCALDATE、ISOLOCAL_TIME。

(了解)**本地化相关的格式**：

ofLocalizedDateTime()适用 LocalDateTime；参数 FormatStyle.MEDIUM/FormatStyle.SHORT

ofLocalizedDate()适用于 LocalDate；参数 FormatStyle.FULL/FormatStyle.LONG/FormatStyle.MEDIUM/FormatStyle.SHORT :

（重点）**自定义的格式**：如：ofPattern(“yyyy-MM-dd hh:mm:ss”)。

```
public void test3(){
    //方式三：自定义的方式（关注、重点）
    DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
    //格式化
    String strDateTime = dateTimeFormatter.format(LocalDateTime.now());
    System.out.println(strDateTime); //2022/12/04 21:05:42
    //解析
    TemporalAccessor accessor = dateTimeFormatter.parse("2022/12/04 21:05:42");
    LocalDateTime localDateTime = LocalDateTime.from(accessor);
    System.out.println(localDateTime); //2022-12-04T21:05:42
}
```

28、持续日期/时间：Period 和 Duration

Duration：用于计算两个“时间”间隔；

Period：用于计算两个“日期”间隔。

（between()静态方法）

```
LocalDate t1 = LocalDate.now();
LocalDate t2 = LocalDate.of(2018, 12, 31);
Period between = Period.between(t1, t2);
System.out.println(between);

System.out.println("相差的年数: "+between.getYears());
System.out.println("相差的月数: "+between.getMonths());
System.out.println("相差的天数: "+between.getDays());
System.out.println("相差的总数: "+between.toTotalMonths());


public void test02(){
    LocalDateTime t1 = LocalDateTime.now();
    LocalDateTime t2 = LocalDateTime.of(2017, 8, 29, 0, 0, 0, 0);
    Duration between = Duration.between(t1, t2);
    System.out.println(between);

    System.out.println("相差的总天数: "+between.toDays());
    System.out.println("相差的总小时数: "+between.toHours());
    System.out.println("相差的总分钟数: "+between.toMinutes());
    System.out.println("相差的总秒数: "+between.getSeconds());
    System.out.println("相差的总毫秒数: "+between.toMillis());
    System.out.println("相差的总纳秒数: "+between.toNanos());
    System.out.println("不够一秒的纳秒数: "+between.getNano());
}
```

29、自然排序：java.lang.Comparable

Comparable 接口强行对实现它的每个类的对象进行整体排序。实现 Comparable 的类必须实现 compareTo(Object obj)方法.

1）两个对象即通过 compareTo(Object obj) 方法的**返回值**来比较大小。1>如果当前对象 this 大于形参对象 obj，则返回正整数；2>如果当前对象 this 小于形参对象 obj，则返回负整数；3>如果当前对象 this 等于形参对象 obj，则返回零。

2）实现 Comparable 接口的对象列表（和数组）可以通过 Collections.sort 或 Arrays.sort 进行自动排序。实现此接口的对象可以用作有序映射中的键或有序集合中的元素，无需指定比较器。

3）对于类 C 的每一个 e1 和 e2 来说，当且仅当 e1.compareTo(e2) == 0 与 e1.equals(e2) 具有相同的 boolean 值时，类 C 的自然排序才叫做与 equals 一致。建议（虽然不是必需的）最好使自然排序与 equals 一致。

4）Comparable 的典型实现：(默认从小到大)

1>String：按字符串中字符 **Unicode 值**比较；

2>Character：按字符 **Unicode 值**来比较；

3>数值类型对应的包装类以及 BigInteger、BigDecimal：按它们对应的**数值大小**进行比较；

4>Boolean：**true** 对应的包装类实例**大于 false** 对应的包装类实例；

5>Date、Time 等：**后面**的日期时间**比前面**的日期时间**大**；

【说明】实现 Comparable 接口的实现步骤：

1>具体的类 A 实现 Comparable 接口；

2>重写 Comparable 接口中 compareTo(Object obj)方法，在此方法中指明比较类 A 的对象的大小的标准；

3>创建类 A 的多个实例，进行大小的比较或排序。

```
public class Product implements Comparable{ //商品类
    private String name;//商品名称
    private double price;//价格

    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }
    ……
    /* 当前的类需要实现Comparable中的抽象方法：compareTo(Object o)
    * 在此方法中，指明如何判断当前类的对象的大小。比如：按照价格的高低进行大小的比较。（或从低到高排序）
    * 如果返回值是正数：当前对象大。
    * 如果返回值是负数：当前对象小。
    * 如果返回值是 0，一样大。* */
    //比较的标准：先比较价格（从大到小），价格相同，进行名字的比较（从小到大）
@Override
public int compareTo(Object o) {
    if(o == this){
        return 0;
    }
    if(o instanceof Product){
        Product p = (Product) o;
int value = Double.compare(this.price,p.price);
        if(value != 0){
            return -value;
        }
        return this.name.compareTo(p.name);
    }
        //手动抛出一个异常类的对象
throw new RuntimeException("类型不匹配");}
public void test2(){
    Product[] arr = new Product[5];
    arr[0] = new Product("Huawei",6299);
    arr[1] = new Product("Xiaomi13pro",4999);
    arr[2] = new Product("VivoX90pro",5999);
    arr[3] = new Product("Iphone14",9999);
    arr[4] = new Product("HonorMagic4",6299);
    Arrays.sort(arr);
    //排序后，遍历
    for (int i = 0; i < arr.length; i++) {
        System.out.println(arr[i]);
    }
}
```

30、定制排序：java.util.Comparator

当元素的类型没有实现 java.lang.Comparable 接口而又不方便修改代码时或如果一个类，实现了 Comparable 接口，也指定了两个对象的比较大小的规则，但是此时此刻我不想按照它预定义的方法比较大小时，用定制排序。

1）重写 compare(Object o1,Object o2)方法，比较 o1 和 o2 的大小：如果方法返回正整数，则表示 o1 大于 o2；如果返回 0，表示相等；返回负整数，表示 o1 小于 o2。

2）可以将 Comparator 传递给 sort 方法（如 Collections.sort 或 Arrays.sort），从而允许在排序顺序上实现精确控制。

【说明】实现 Comparator 接口的实现步骤：

1>创建一个实现 Comparator 接口的实现类 A；

2>实现类 A 要求重写 Comparator 接口中的抽象方法 compare(Object o1,Object o2)，在此方法中指明要比较大小的对象的大小关系。（比如，String 类、Product 类）；

3>创建此实现类 A 的对象，并将此对象传入到相关方法的参数位置即可。（比如：Arrays.sort(..,实现类 A 的实例)）。

```
public void test1(){
    Product[] arr = new Product[5];
    arr[0] = new Product("Huawei",6299);
    arr[1] = new Product("Xiaomi13pro",4999);
    arr[2] = new Product("VivoX90pro",5999);
    arr[3] = new Product("Iphone14",9999);
    arr[4] = new Product("HonorMagic4",6299);
  //创建一个实现 Comparator 接口的实现类的对象
  Comparator comparator = new Comparator(){
  //如果判断两个对象 o1,o2 的大小，其标准就是此方法的方法体要编写的逻辑。
  //比如：按照价格从高到低排序
  @Override
  public int compare(Object o1, Object o2) {
      if(o1 instanceof Product && o2 instanceof Product){
          Product p1 = (Product) o1;
          Product p2 = (Product) o2;
return -Double.compare(p1.getPrice(),p2.getPrice());
          }
  throw new RuntimeException("类型不匹配");
      }
  };
 Comparator comparator1 = new Comparator(){
//如果判断两个对象 o1,o2 的大小，其标准就是此方法的方法体要编写的逻辑。
//比如：按照 name 从低到高排序
    @Override
    public int compare(Object o1, Object o2) {
        if(o1 instanceof Product && o2 instanceof Product){
            Product p1 = (Product) o1;
            Product p2 = (Product) o2;
return p1.getName().compareTo(p2.getName());
        }
  throw new RuntimeException("类型不匹配");
    }};
    Arrays.sort(arr,comparator1);
    //排序后，遍历
    for (int i = 0; i < arr.length; i++) {
        System.out.println(arr[i]);
    }
}
```

31、对比两种方式：

1）自然排序：单一的，唯一的；

定制排序：灵活的、多样的。

2）自然排序：一劳永逸的；

定制排序：临时的。

3）自然排序：对应的接口是 Comparable，对应的抽象方法是 compareTo(Object obj)；

定制排序：对应的接口是 Comparator，对应的抽象方法是 compare (Object o1,Object o2)。

32、系统相关类(java.lang.System)

1）System 类代表系统，系统级的很多属性和控制方法都放置在该类的内部。该类位于 java.lang 包。

2）由于该类的**构造器**是 **private 的**，所以无法创建该类的对象。其内部**成员变量**和**成员方法**都是 **static** 的，所以也可以很方便的进行调用。

3）成员变量 Scanner scan = new Scanner(System.in); System 类内部包含 **in**、**out** 和 **err** 三个成员变量，分别代表**标准输入流**(键盘输入)，**标准输出流**(显示器)和**标准错误输出流**(显示器)。

4）成员方法

1>native long **currentTimeMillis()**： 该方法的作用是返回当前的计算机时间，时间的表达格式为当前计算机时间和 GMT 时间(格林威治时间)1970 年 1 月 1 号 0 时 0 分 0 秒所差的**毫秒数**。

2>void **exit(int status)**： 该方法的作用是**退出程序**。其中 status 的值为 **0** 代表**正常退出**，**非零**代表**异常退出**。使用该方法可以在图形界面编程中实现程序的退出功能等。(后面代码不会执行)

3>void **gc()**： 该方法的作用是请求系统进行**垃圾回收**。至于系统是否立刻回收，则**取决于系统中垃圾回收算法的实现**以及**系统执行时的情况**。

4>String **getProperty(String key)**： 该方法的作用是获得系统中**属性名为 key 的**属性对应的**值**。系统中常见的属性名以及属性的作用如下表所示：

属性名	属性说明
java.version	Java 运行时环境版本
java.home	Java 安装目录
os.name	操作系统的名称
os.version	操作系统的版本
user.name	用户的账户名称
user.home	用户的主目录
user.dir	用户的当前工作目录

5>static void **arraycopy**(Object src, int srcPos, Object dest, int destPos, int length)：从指定源数组中复制一个数组，复制从指定的位置开始，到目标数组的指定位置结束。常用于数组的插入和删除。

33、系统相关类(java.lang.Runtime)

每个 Java 应用程序都有一个 Runtime 类实例，使应用程序能够与其运行的环境相连接；是一个单例模式。

1）public static Runtime **getRuntime()**： 返回与当前 Java 应用程序相关的运行时对象。应用程序不能创建自己的 Runtime 类实例。

2）public long totalMemory()：返回 Java 虚拟机中初始化时的内存总量。此方法返回的值可能随时间的推移而变化，这取决于主机环境。默认为物理电脑内存的 **1/64**。

3）public long maxMemory()：返回 Java 虚拟机中最大程度能使用的内存总量。默认为物理电脑内存的 **1/4**。

4）public long freeMemory()：返回 Java 虚拟机中的空闲内存量。调用 gc 方法可能导致 freeMemory 返回值的增加。

34、数学相关的类(java.lang.Math)

java.lang.Math 类包含用于执行基本数学运算的方法，如初等指数、对数、平方根和三角函数。类似这样工具类，**其所有方法均为静态方法**，并且不会创建对象，调用起来非常简单。

1）public static double **abs**(double a)：返回 double 值的**绝对值**；

double d1 = Math.abs(-5); //d1 的值为 5

double d2 = Math.abs(5); //d2 的值为 5

2）public static double **ceil**(double a)：返回**大于等于**参数的最小的整数（**天花板往上取整**）；

double d1 = Math.ceil(3.3); //d1 的值为 4.0

double d2 = Math.ceil(-3.3); //d2 的值为 -3.0

double d3 = Math.ceil(5.1); //d3 的值为 6.0

3）public static double **floor**(double a)：返回**小于等于**参数最大的整数（**地板往下取整**）；

double d1 = Math.floor(3.3); //d1 的值为 3.0

double d2 = Math.floor(-3.3); //d2 的值为-4.0

double d3 = Math.floor(5.1); //d3 的值为 5.0

4）public static long **round**(double a) ：返回**最接近参数**的 long。(相当于四舍五入方法) ；

【判断方法】x+0.5 后向下取整

long d1 = Math.round(5.5); //d1 的值为 6

long d2 = Math.round(5.4); //d2 的值为 5

long d3 = Math.round(-3.3); //d3 的值为-3

long d4 = Math.round(-3.8); //d4 的值为-4

5）public static double **pow**(double a,double b)：返回 a 的 b **幂次**方法；

6）public static double **sqrt**(double a)：返回 a 的**平方根**；

7）public static double **random()**：返回**[0,1)**的**随机值**；

8）public static final double **PI**：返回**圆周率**；

9）public static double **max**(double x, double y)：返回 x,y 中的**最大值**；

10）public static double **min**(double x, double y)：返回 x,y 中的**最小值**；

11）其它：acos,asin,atan,cos,sin,tan 三角函数

double result = Math.pow(2,31);

double sqrt = Math.sqrt(256);

double rand = Math.random();

double pi = Math.PI;

35、数学相关的类(java.math 包)

1）BigInteger

1>Integer 类作为 int 的包装类，能存储的最大整型值为 2^31-1，Long 类也是有限的，最大为 2^63-1。如果要表示再大的整数，不管是基本数据类型还是他们的包装类都无能为力，更不用说进行运算了。

2>java.math 包的 **BigInteger** 可以表示**不可变**的**任意精度**的**整数**。BigInteger 提供所有 Java 的基本整数操作符的对应物，并提供 java.lang.Math 的所有相关方法。另外，BigInteger 还提供以下运算：模算术、GCD 计算、质数测试、素数生成、位操作以及一些其他操作。

3>构造器

- BigInteger(String val)：根据字符串构建 BigInteger 对象。

4>方法

- public BigInteger abs()：返回此 BigInteger 的绝对值的 BigInteger；
- BigInteger add(BigInteger val) ：返回其值为 (this + val) 的 BigInteger；
- BigInteger subtract(BigInteger val) ：返回其值为 (this - val) 的 BigInteger；
- BigInteger multiply(BigInteger val) ：返回其值为 (this * val) 的 BigInteger；
- BigInteger divide(BigInteger val) ：返回其值为 (this / val) 的 BigInteger。整数相除只保留整数部分；
- BigInteger remainder(BigInteger val) ：返回其值为 (this % val) 的 BigInteger；
- BigInteger[] divideAndRemainder(BigInteger val)：返回包含 (this / val) 后跟 (this % val) 的两个 BigInteger 的数组；
- BigInteger pow(int exponent)：返回其值为 (this^exponent)的 BigInteger。

2）BigDecimal

1>一般的 Float 类和 Double 类可以用来做科学计算或工程计算，但在商业计算中，要求数字精度比较高，故用 java.math.BigDecimal 类。

2>BigDecimal 类支持**不可变的**、**任意精度的**有符号十进制**定点数**。

3>构造器

- public BigDecimal(double val)
- public BigDecimal(String val) --> 推荐

4>常用方法

- public BigDecimal add(BigDecimal augend)
- public BigDecimal subtract(BigDecimal subtrahend)
- public BigDecimal multiply(BigDecimal multiplicand)
- public BigDecimal divide(BigDecimal divisor, int scale, int roundingMode)：**divisor** 是**除数**，**scale** 指明**保留几位小数**，**roundingMode** 指明**舍入模式**（ROUNDUP：向上加 1、ROUNDDOWN:直接舍去、ROUNDHALFUP:四舍五入）

36、数学相关的类(java.util.Random)

用于产生随机数

1）boolean **nextBoolean()**：返回下一个伪随机数，它是取自此随机数生成器序列的**均匀分布的 boolean 值**。

2）void **nextBytes(byte[] bytes)**：生成**随机字节**并将其**置于**用户提供的 **byte 数组中**。

3）double **nextDouble()**：返回下一个伪随机数，它是取自此随机数生成器序列的、在 **0.0 和 1.0 之间均匀分布**的 **double 值**。

4）float **nextFloat()**：返回下一个伪随机数，它是取自此随机数生成器序列的、**在 0.0 和 1.0 之间均匀分布**的 **float 值**。

5）double **nextGaussian()**：返回下一个伪随机数，它是取自此随机数生成器序列的、**呈高斯（“正态”）分布**的 double 值，其**平均值是 0.0**，**标准差是 1.0**。

6）int **nextInt()**：返回下一个伪随机数，它是此随机数生成器的序列中**均匀分布的 int 值**。

7）int **nextInt(int n)**：返回一个伪随机数，它是取自此随机数生成器序列的、**在 0（包括）和指定值（不包括）之间**均匀分布的 int 值。

8）long **nextLong()**：返回下一个伪随机数，它是取自此随机数生成器序列的**均匀分布的 long 值**。

```
Random r = new Random();
System.out.println("随机整数：" + r.nextInt());
```