JAVA 中的一些基本类方法（成员函数）

1. (typeName)Math.round(value)---(int)Math.round(x);
2. String Class: s.substring(x，y)---提取 x 到 y-1 共 y-x 位---如（0,3）是 0,1,2 共 3 位;

   当参数仅一位时 substring(n)返回有索引 n 到结尾的字符串。
3. String Class: s.length（）---计算 s 的代码单元数---str.length()=4
4. String Class: s.codePointCount (0, s.length())---计算 s 的代码点数量
5. String Class: s.charAt(n)---返回 s 位置（索引位置）n 处代码单元的符号 **(UTF-16)**，其中 0<=n<=s.length()-1.
6. StringClass :intindex=greeting.offsetByCodePoints(0,i);--- s.offsetByCodePoints，返回从 0 处开始的第 i 个 Code Point 的位置

   int cp = greeting.codePointAt(index); //获取第 index 位置的 Code Point
7. JAVA 中不用 char,

   遍历代码点可用：

   int cp=sentence.codePointAt(n)

   If (Character.isSupplementaryCodePoint(cp))

   I+=2;

   else

   i++;

8. String Class: s.replace(old,new)---如果 old 没有出现，将原字符串返回。如

果 old 出现多次，就全部替换掉。

9. String Class: stirng search :

   s.indexOf(“s”)---return 找到的第一个的索引位置。Eg:“xutao” 中“t”是 2，找不到就返回-1.

   s.lastIndexOf(“s”)---返回找到的最后一个的索引位置。如果参数为空，即“”（无空格），返回字符代码单元，（长度）；

10. String 去中间空格: str.trim()---返回的是去除前后空格的 副本。单词 trim 修饰整理。

11. 判断 string 开始与结尾: 返回 boolean 类型，判断字符串是不是一参数指定的字符串结尾或开始; 写成单引号时会报错;

    str,startsWith(str)

    str.endsWith(str)

12. 比较字符串：返回 boolean 值;

    Str.equals(str1)

    Str.equalsIgnoreCase(str1),忽略大小写;

    以 **String 声明的两个相同内容的变量，用‘==’会 true ，但是用 new 创建的两个相同内容的对象会 False**

13. 按字典序比较两个字符串:

    Str.compareTo(str)在前

    参数在前或参数是原数的前半部分（先结束）返回一个正数（+1）

    参数在后或原数的一部分是参数（后结束）返回一个负数（-1）

    相同返回 0

14. 大小写转换：

str.toUpperCase()

str.tolowerCase()

其他的不影响。

Str.split(String sign)：　以 sign 为分隔符分割 str

Str.split(A | B | C …)：多个分割符

Str.split(string sign ,int n)：分割成 n 份：注意：n=1 分割无效，n=0 无效，

故不要用

15. 时间日期的格式化输出：

包：import java.util.Date;

Date date=new Date();

String Time=String;format(“格式说明符”，date);

Date 类方法：

1	**boolean after(Date date)** 若当调用此方法的 Date 对象在指定日期之后返回 true,否则返回 false。
2	**boolean before(Date date)** 若当调用此方法的 Date 对象在指定日期之前返回 true,否则返回 false。
3	**Object clone( )** 返回此对象的副本。

4	**int compareTo(Date date)** 比较当调用此方法的 Date 对象和指定日期。两者相等时候返回 0。调用对象在指定日期之前则返回负数。调用对象在指定日期之后则返回正数。
5	**int compareTo(Object obj)** 若 obj 是 Date 类型则操作等同于 compareTo(Date) 。否则它抛出 ClassCastException。
6	**boolean equals(Object date)** 当调用此方法的 Date 对象和指定日期相等时候返回 true,否则返回 false。
7	**long getTime( )** getDay,Year,Month,Year……. 返回自 1970 年 1 月 1 日 00:00:00 GMT 以来此 Date 对象表示的毫秒数。
8	**int hashCode( )** 返回此对象的哈希码值。
9	**void setTime(long time)** 用自 1970 年 1 月 1 日 00:00:00 GMT 以后 time 毫秒数设置时间和日期。
10	**String 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)。 Fri Apr 13 21:33:12 CST 2018

使用 SimpleDateFormat 类格式化日期：自定义

G	纪元标记	AD
y	四位年份	2001
M	月份	July or 07
d	一个月的日期	10
h	A.M./P.M. (1~12)格式小时	12
H	一天中的小时 (0~23)	22
m	分钟数	30
s	秒数	55
S	毫秒数	234
E	星期几	Tuesday
D	一年中的日子	360
F	一个月中第几周的周几	2 (second Wed. in July)
w	一年中第几周	40
W	一个月中第几周	1
a	A.M./P.M. 标记	PM

k	一天中的小时(1~24)	24
K	A.M./P.M. (0~11)格式小时	10
z	时区	Eastern Standard Time
'	文字定界符	Delimiter
''	单引号	`

import java.text.SimpleDateFormat; 导入 text

Date dNow = new Date( ); 先用 Date 类创建对象 date

SimpleDateFormat ft = new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz"); 再用 SimpleDateFormat 类创建一个对象 sdf

System.out.println("Current Date: " + ft.format(dNow)); 格式：sdf.format(date)

使用 printf 格式化日期:

两个字母格式，它以 **%t** 开头并且以下面表格中的一个字母结尾。

c	包括全部日期和时间信息	星期六 十月 27 14:21:20 CST 2007
F	"年-月-日"格式	2007-10-27
D	"月/日/年"格式	10/27/07
r	"HH:MM:SS PM"格式（12 时制）	02:25:51 下午
T	"HH:MM:SS"格式（24 时制）	14:28:16

R	"HH:MM"格式（24 时制）	14:28

System.out.printf("年-月-日格式：%tF%n",date);-注意：%n 表示行分隔符相当换行

如果你需要重复提供日期，那么利用这种方式来格式化它的每一部分就有点复杂了。因此，可以利用一个格式化字符串指出要被格式化的参数的索引。？？？？？

或者，你可以使用 < 标志。它表明先前被格式化的参数要被再次使用

常规格式化：

%b, %B---布尔

%h,%H ---散列码

%s%S 字符串

%c%C—字符

%d, %o, %x%X

%e—科学记数法十进制

%a---十六进制浮点小数

%n—换行

%%---%

```
String df = String.format("%o", 8)
```

Character 类：

用于对单个字符操作，是数据类型 char 的包装类，在需要是编译器会自动将 char 数据转化为类对象，即装箱，相反过程为拆箱。

import java.lang.Character;

Character vb = **new** Character ('g') ---创建对象；

方法：

1	isLetter()　是否是一个字母
2	isDigit()　是否是一个数字字符
3	isWhitespace()是否是一个空格
4	isUpperCase()是否是大写字母
5	isLowerCase()是否是小写字母
6	toUpperCase()指定字母的大写形式
7	toLowerCase()指定字母的小写形式
8	toString()返回字符的字符串形式，字符串的长度仅为 1

String 类：

连接：　string1.concat(string2)　相当与+

1	char charAt(int index) 返回指定索引处的 char 值。

2	int compareTo(Object o) 把这个字符串和另一个对象比较。
3	int compareTo(String anotherString) 按字典顺序比较两个字符串。
4	int compareToIgnoreCase(String str) 按字典顺序比较两个字符串，不考虑大小写。
5	String concat(String str) 将指定字符串连接到此字符串的结尾。
6	boolean contentEquals(StringBuffer sb) 当且仅当字符串与指定的 StringBuffer 有相同顺序的字符时候返回真。
7	static String copyValueOf(char[] data) 返回指定数组中表示该字符序列的 String。
8	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。
9	boolean endsWith(String suffix) 测试此字符串是否以指定的后缀结束。
10	boolean equals(Object anObject) 将此字符串与指定的对象比较。

11	boolean equalsIgnoreCase(String anotherString) 将此 String 与另一个 String 比较，不考虑大小写。
12	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。
13	byte[] getBytes(String charsetName) 使用指定的字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中。
14	void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) 将字符从此字符串复制到目标字符数组。
15	int hashCode() 返回此字符串的哈希码。
16	int indexOf(int ch) 返回指定字符在此字符串中第一次出现处的索引。
17	int indexOf(int ch, int fromIndex) 返回在此字符串中第一次出现指定字符处的索引，从指定的索引开始搜索。
18	int indexOf(String str) 返回指定子字符串在此字符串中第一次出现处的索引。

19	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。
20	String intern() 返回字符串对象的规范化表示形式。
21	int lastIndexOf(int ch) 返回指定字符在此字符串中最后一次出现处的索引。
22	int lastIndexOf(int ch, int fromIndex) 返回指定字符在此字符串中最后一次出现处的索引，从指定的索引处开始进行反向搜索。
23	int lastIndexOf(String str) 返回指定子字符串在此字符串中最右边出现处的索引。
24	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。
25	int length() 返回此字符串的长度。
26	boolean matches(String regex) 告知此字符串是否匹配给定的正则表达式。

27	boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len) 测试两个字符串区域是否相等。
28	boolean regionMatches(int toffset, String other, int ooffset, int len) 测试两个字符串区域是否相等。
29	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。
30	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。
31	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。
32	String[] split(String regex) 根据给定正则表达式的匹配拆分此字符串。
33	String[] split(String regex, int limit) 根据匹配给定的正则表达式来拆分此字符串。
34	boolean startsWith(String prefix)

	测试此字符串是否以指定的前缀开始。
35	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。
36	CharSequence subSequence(int beginIndex, int endIndex) 返回一个新的字符序列，它是此序列的一个子序列。
37	String substring(int beginIndex) 返回一个新的字符串，它是此字符串的一个子字符串。
38	String substring(int beginIndex, int endIndex) 返回一个新字符串，它是此字符串的一个子字符串。
39	char[] toCharArray() 将此字符串转换为一个新的字符数组。
40	String toLowerCase() 使用默认语言环境的规则将此 String 中的所有字符都转换为小写。
41	String toLowerCase(Locale locale) 使用给定 Locale 的规则将此 String 中的所有字符都转换为小写。
42	String toString() 返回此对象本身（它已经是一个字符串！）。
43	String toUpperCase()

	使用默认语言环境的规则将此 String 中的所有字符都转换为大写。
44	String toUpperCase(Locale locale) 使用给定 Locale 的规则将此 String 中的所有字符都转换为大写。
45	String trim() 返回字符串的副本，忽略前导空白和尾部空白。
46	static String valueOf(primitive data type x) 返回给定 data type 类型 x 参数的字符串表示形式。

StringBuffer 类 StringBuilder 类：

当对字符串进行修改的时候，需要使用 StringBuffer 和 StringBuilder 类。

和 String 类不同的是，**StringBuffer 和 StringBuilder 类的对象能够被多次的修改，并且不产生新的未使用对象。**

StringBuilder 类在 Java 5 中被提出，它和 StringBuffer 之间的最大不同在于 StringBuilder 的方法不是线程安全的（不能同步访问）。

由于 StringBuilder 相较于 StringBuffer 有速度优势，所以多数情况下建议使用 StringBuilder 类。然而在应用程序要求线程安全的情况下，则必须使用 StringBuffer 类。

1	public StringBuffer **append(**String s) 将指定的字符串追加到此字符序列。
2	public StringBuffer **reverse()**

	将此字符序列用其反转形式取代。
3	public **delete(int start, int end)** 移除此序列的子字符串中的字符。
4	public **insert(int offset, int i)** 将 int 参数的字符串表示形式插入此序列中。
5	**replace(int start, int end, String str)** 使用给定 String 中的字符替换此序列的子字符串中的字符

其他一些方法与 string 类似：

Str.capacity()---分配的容量，比如：字符串长度+16 ，给 16 个附加的字符自动增加了存储空间

void setLength(int newLength) 设置字符序列的长度。

setCharAt(int index, char ch) 将给定索引处的字符设置为 ch。

注意：String 类是被 final 修饰，即不可变的，操作时会重新创建一个对象，而 StringBuilder 是可变的

StringBuffer 线程安全 StringBuilder 线程不安全

StringBuilder 速度快

数组:

创建数组:

Eg :Int [] arr = new int [23]

String [][] arr = new Srring [2][3]　　//可以表明数组长度来分配空间

Int arr [ ]　=　new int [ ] {1,2,3}　　//也可以直接赋值直接分配空间，此时不要加数组长度,物等号=

整数数组初始值为 0

小数数组初始值为小数 0.0

Cha 初始值为空

Arrays 类:

Arrays Class(java.util packet):　　(方法多为静态)

Import java.util,Arrays

1. arr.length---array length
2. foreach 语句格式:

for(元素类型 type　元素变量 value ： 遍历对象 obj) {

引用 x 的 java 语句; }

Eg:　for(int x,arr)

for(int x[]:arr) //把二位看成维，自然要是 int x[]

for(int y:x) {//真正的一维，其中 x 继承第一个语句

System.*out*.println(y);//y 才是真正的元素；输出 y

3. 静态方法 fill（）

fill (int[] a , int n)---将 n 赋值给 a[]的每个元素

fill（int[]a, int first, int last, int value）---将 value 赋值给索引在 first 到 last 的元素

4. 静态方法分类 sort（）---Arrays.sort(arr)—**按字典序：数字+大写+小写**
5. 复制： 都会得到一个新的数组

   copyOf(arr, int linelength)---有第一个开始复制，line 大于数组长度是整数用 0 填充，char 用 null 填充

   copyOfRange(arr, int first, int last)---将 first 到 last 的复制到新的数组：注意：新数组长度为 last-first，即 last 索引取不到： 前取后不娶
6. 数组查询“

   Arrays.binarySearch(arr, target)　　返回索引

   Arrays.binarySearch(arr,int first , int last , target)

7.**数组长度： arr . length　　无括号**

正则表达式：

定义字符串模式的方法

须导入包：java.util.regex　　：　**import** java.util.regex.*;

java.util.regex 包主要包括以下三个类：

- **Pattern 类：**

pattern 对象是一个正则表达式的编译表示。Pattern 类没有公共构造方法。要创

建一个 Pattern 对象，你必须首先调用其公共静态编译方法，它返回一个 Pattern 对象。该方法接受一个正则表达式作为它的第一个参数。

- **Matcher 类：**

Matcher 对象是对输入字符串进行解释和匹配操作的引擎。与 Pattern 类一样，Matcher 也没有公共构造方法。你需要调用 Pattern 对象的 matcher 方法来获得一个 Matcher 对象。

- **PatternSyntaxException：**

PatternSyntaxException 是一个非强制异常类，它表示一个正则表达式模式中的语法错误。

Eg: boolean isMatch = Pattern.matches(pattern, content);

\	将下一字符标记为特殊字符、文本、反向引用或八进制转义符。例如，"n"匹配字符"n"。"\n"匹配换行符。**序列"\\\\"匹配"\\"，"\\("匹配"("**。
^	**匹配输入字符串开始的位置**。如果设置了 **RegExp** 对象的 **Multiline** 属性，^ 还会与"\n"或"\r"之后的位置匹配。
$	**匹配输入字符串结尾的位置**。如果设置了 **RegExp** 对象的 **Multiline** 属性，$ 还会与"\n"或"\r"之前的位置匹配。
*	**零次或多次匹配前面的字符或子表达式**。例如，zo* 匹配"z"和"zoo"。*** 等效于 {0,}**。
+	**一次或多次匹配前面的字符或子表达式**。例如，"zo+"与"zo"和"zoo"匹配，但与"z"不匹配。**+ 等效于 {1,}**。

?	**零次或一次匹配前面的字符或子表达式**。例如，"do(es)?"匹配"do"或"does"中的"do"。**? 等效于 {0,1}**。
{n}	***n* 是非负整数。正好匹配 *n* 次**。例如，"o{2}"与"Bob"中的"o"不匹配，但与"food"中的两个"o"匹配。
{n,}	***n* 是非负整数。至少匹配 *n* 次**。例如，"o{2,}"不匹配"Bob"中的"o"，而匹配"foooood"中的所有 o。"o{1,}"等效于"o+"。"o{0,}"等效于"o*"。
{n,m}	***M* 和 *n* 是非负整数，其中 *n* <= *m*。匹配至少 *n* 次，至多 *m* 次**。例如，"o{1,3}"匹配"fooooood"中的头三个 o。'o{0,1}' 等效于 'o?'。注意：**您不能将空格插入逗号和数字之间**。
?	**当此字符紧随任何其他限定符（*、+、?、*{n}*、*{n,}*、*{n,m}*）之后时，匹配模式是"非贪心的"。"非贪心的"模式匹配搜索到的、尽可能短的字符串，而默认的"贪心的"模式匹配搜索到的、尽可能长的字符串**。例如，在字符串"oooo"中，"o+?"只匹配单个"o"，而"o+"匹配所有"o"。
. 不是\.	**匹配除"\r\n"之外的任何单个字符**。**若要匹配包括"\r\n"在内的任意字符，请使用诸如"[\s\S]"之类的模式。**
(*pattern*)	匹配 *pattern* 并捕获该匹配的子表达式。可以使用 **$0…$9** 属性从结果"匹配"集合中检索捕获的匹配。若要匹配括号字符 ( )，请使用"\("或者"\)"。
(?:*pattern*)	匹配 *pattern* 但不捕获该匹配的子表达式，即它是一个非捕获匹配，不

	存储供以后使用的匹配。这对于用"or"字符 (|) 组合模式部件的情况很有用。例如，'industr(?:y|ies) 是比 'industry|industries' 更经济的表达式。
(?=*pattern*)	执行正向预测先行搜索的子表达式，该表达式匹配处于匹配 *pattern* 的字符串的起始点的字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹配。例如，'Windows (?=95|98|NT|2000)' 匹配"Windows 2000"中的"Windows"，但不匹配"Windows 3.1"中的"Windows"。预测先行不占用字符，即发生匹配后，下一匹配的搜索紧随上一匹配之后，而不是在组成预测先行的字符后。
(?!*pattern*)	执行反向预测先行搜索的子表达式，该表达式匹配不处于匹配 *pattern* 的字符串的起始点的搜索字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹配。例如，'Windows (?!95|98|NT|2000)' 匹配"Windows 3.1"中的 "Windows"，但不匹配"Windows 2000"中的"Windows"。预测先行不占用字符，即发生匹配后，下一匹配的搜索紧随上一匹配之后，而不是在组成预测先行的字符后。
x|*y*	**匹配 *x* 或 *y***。例如，'z|food' 匹配"z"或"food"。'(z|f)ood' 匹配"zood"或"food"。
[*xyz*]	**字符集**。**匹配包含的任一字符**。例如，"[abc]"匹配"plain"中的"a"。
[^*xyz*]	**反向字符集。匹配未包含的任何字符**。例如，"[^abc]"匹配"plain"中"p"，"l"，"i"，"n"。

[*a-z*]	字符范围。匹配指定范围内的任何字符。例如，"[a-z]"匹配"a"到"z"范围内的任何小写字母。
[^*a-z*]	**反向范围字符。匹配不在指定的范围内的任何字符**。例如，"[^a-z]"匹配任何不在"a"到"z"范围内的任何字符。
\b	**匹配一个字边界，即字与空格间的位置**。例如，"er\b"匹配"never"中的"er"，但不匹配"verb"中的"er"。
\B	**非字边界匹配**。"er\B"匹配"verb"中的"er"，但不匹配"never"中的"er"。
\c*x*	**匹配 *x* 指示的控制字符**。例如，**\cM 匹配 Control-M 或回车符**。*x* 的值必须在 A-Z 或 a-z 之间。如果不是这样，则假定 c 就是"c"字符本身。
\d	**数字字符匹配。等效于 [0-9]。**
\D	**非数字字符匹配。等效于 [^0-9]。**
\f	**换页符匹配。等效于 \x0c 和 \cL。**
\n	**换行符匹配。等效于 \x0a 和 \cJ。**
\r	匹配一个回车符。等效于 \x0d 和 \cM。
\s	**匹配任何空白字符，包括空格、制表符、换页符等。与 [\f\n\r\t\v] 等效。**
\S	**匹配任何非空白字符。与 [^ \f\n\r\t\v] 等效。**

\t	制表符匹配。与 \x09 和 \cI 等效。
\v	**垂直制表符匹配**。与 \x0b 和 \cK 等效。
\w	**匹配任何字类字符，包括下划线。与"[A-Za-z0-9_]"等效**。***可用作标识符的字符***
\W	**与任何非单词字符匹配。与"[^A-Za-z0-9_]"等效**。
\x*n*	匹配 *n*，此处的 *n* 是一个十六进制转义码。十六进制转义码必须正好是两位数长。例如，"\x41"匹配"A"。"\x041"与"\x04"&"1"等效。允许在正则表达式中使用 ASCII 代码。
num	匹配 *num*，此处的 *num* 是一个正整数。到捕获匹配的反向引用。例如，"(.)\1"匹配两个连续的相同字符。
n	标识一个八进制转义码或反向引用。如果 *n* 前面至少有 *n* 个捕获子表达式，那么 *n* 是反向引用。否则，如果 *n* 是八进制数 (0-7)，那么 *n* 是八进制转义码。
nm	标识一个八进制转义码或反向引用。如果 *nm* 前面至少有 *nm* 个捕获子表达式，那么 *nm* 是反向引用。如果 *nm* 前面至少有 *n* 个捕获，则 *n* 是反向引用，后面跟有字符 *m*。如果两种前面的情况都不存在，则 *nm* 匹配八进制值 *nm*，其中 *n* 和 *m* 是八进制数字 (0-7)。
\nml	当 *n* 是八进制数 (0-3)，*m* 和 *l* 是八进制数 (0-7) 时，匹配八进制转义码 *nml*。

\u*n*	匹配 *n*，其中 *n* 是以四位十六进制数表示的 Unicode 字符。例如，\u00A9 匹配版权符号 (©)。
以下为元字符（单个）	
\\p{Lower}	小写
\\p{Upper}	大写
\\p{ASCII}	ASCII 码
\\p{Alpha}	字母
\\p{Digit}	数字
\\p{Alnum}	字母数字
\\p{Punct}	标点
\\p{Graph}	可见字符
\\p{Print}	可打印字符
\\p{Blank}	空格或制表符
\\p{Cntrl}	控制字符

根据 Java Language Specification 的要求，Java 源代码的字符串中的反斜线被解释为 Unicode 转义或其他字符转义。因此必须在字符串字面值中使用两个反

斜线，表示正则表达式受到保护，不被 Java 字节码编译器解释。例如，当解释为正则表达式时，字符串字面值 "\b" 与单个退格字符匹配，而 "\\b" 与单词边界匹配。字符串字面值 "\(hello\)" 是非法的，将导致编译时错误；要与字符串 (hello) 匹配，必须使用字符串字面值 "\\(hello\\)"。

捕获组：

捕获组是把多个字符当一个单独单元进行处理的方法，它通过对括号内的字符分组来创建。在表达式（(A)（B（C））），有四个这样的组：（(A)(B(C))），(A)，(B(C))，(C)

调用 matcher 对象的 groupCount 方法来查看表达式有多少个分组

特殊的组（group(0)），它总是代表整个表达式。该组不包括在 groupCount 的返回值中。

```
import java.util.regex.*;
      String xx = "123workharder12315";
//    String re = "((\\d*)(\\D*)(.*))";最外的大括号表示一个单元组，即先整体，再从左到右;
      String re = "(\\d*)(\\D*)(.*)";  预定的正则表达式
      Pattern r = Pattern.compile(re); 先compile
      Matcher m = r.matcher(xx);再matcher
      int num = m.groupCount(); groupCount单元组数量
      if(m.find()) {  //find （） 表示是否找到—布尔
         for (int i=0;i<=num;i++)
            System.out.println(m.group(i));
```

```
pattern = Pattern.compile(REGEX);

matcher = pattern.matcher(INPUT);
```

Matcher 类方法：

索引方法：

1	**public int start()** 返回以前匹配的初始索引。
2	**public int start(int group)** 返回在以前的匹配操作期间，由给定组所捕获的子序列的初始索引
3	**public int end()** 返回最后匹配字符之后的偏移量。
4	**public int end(int group)** 返回在以前的匹配操作期间，由给定组所捕获子序列的最后字符之后的偏移量。

研究方法：

1	**public boolean lookingAt()** 尝试将从区域开头开始的输入序列与该模式匹配。
2	**public boolean find()** 尝试查找与该模式匹配的输入序列的下一个子序列。
3	**public boolean find(int start)**

	重置此匹配器，然后尝试查找匹配该模式、从指定索引开始的输入序列的下一个子序列。
4	**public boolean matches()** 尝试将整个区域与模式匹配。

替换方法：

1	**public Matcher appendReplacement(StringBuffer sb, String replacement)** 实现非终端添加和替换步骤。
2	**public StringBuffer appendTail(StringBuffer sb)** 实现终端添加和替换步骤。
3	**public String replaceAll(String replacement)** 替换模式与给定替换字符串相匹配的输入序列的每个子序列。
4	**public String replaceFirst(String replacement)** 替换模式与给定替换字符串匹配的输入序列的第一个子序列。
5	**public static String quoteReplacement(String s)** 返回指定字符串的字面替换字符串。这个方法返回一个字符串，就像传递给 Matcher 类的 appendReplacement 方法一个字面字符串一样工作。

matches 和 lookingAt 方法都用来尝试匹配一个输入序列模式。它们的不同是 matches 要求整个序列都匹配，而 lookingAt 不要求。

lookingAt 方法虽然不需要整句都匹配，但是需要从第一个字符开始匹配。

这两个方法经常在输入字符串的开始使用。

replaceFirst 和 replaceAll 方法用来替换匹配正则表达式的文本。不同的是，replaceFirst 替换首次匹配，replaceAll 替换所有匹配。

Matcher 类也提供了 appendReplacement 和 appendTail 方法用于文本替换

PatternSyntaxException 是一个非强制异常类，它指示一个正则表达式模式中的语法错误

1	**public String getDescription()** 获取错误的描述。
2	**public int getIndex()** 获取错误的索引。
3	**public String getPattern()** 获取错误的正则表达式模式。
4	**public String getMessage()** 返回多行字符串，包含语法错误及其索引的描述、错误的正则表达式模式和模式中错误索引的可视化指示。

Java 方法

- println() 是一个方法。System 是系统类。out 是标准输出对象
- 可变参数：typeName... parameterName

double... numbers

控制台输入

方法 1:

输入由 System.in 完成，可将 system.in 包装在一个 BufferedReader 对象中来**创建一个字符流：**

import java.io.*;

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

在主方法后加 throws IOException 使输入流为空时抛出异常

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

BufferedReader 对象创建后使用 read(), readLine()从输入流中读取字符，字符串

read() 方法从输入流**读取一个字符**并把该字符**作为整数值返回**。 当流结束的时候返回 -1，抛出 IOException

readLine()方法可以直接在控制台显示出来

方法 2—Scanner 类

Import java,util.Scanner （不在 io 里）

Scanner s = new Scanner(System.in)

next(), nextLine() 获取**字符串，后台可显示**

区别:

next():

- 1、一定要读取到有效字符后才可以结束输入。
- 2、对输入有效字符之前遇到的空白，next() 方法会自动将其去掉。
- 3、只有输入有效字符后才将其后面输入的空白作为分隔符或者结束符。
- next() 不能得到带有空格的字符串。

nextLine():

- 1、以 Enter 为结束符,也就是说 nextLine()方法返回的是输入回车之前的所有字符。
- 2、可以获得空白。

hasNext(), hasNextLine()布尔—判断是否好有输入的数据

```
if (scan.hasNextLine())
    String str2 = scan.nextLine();
```

对于其他数据类型:

nextInt() hasNextInt()

nextDouble() hasNextDouble()

.....

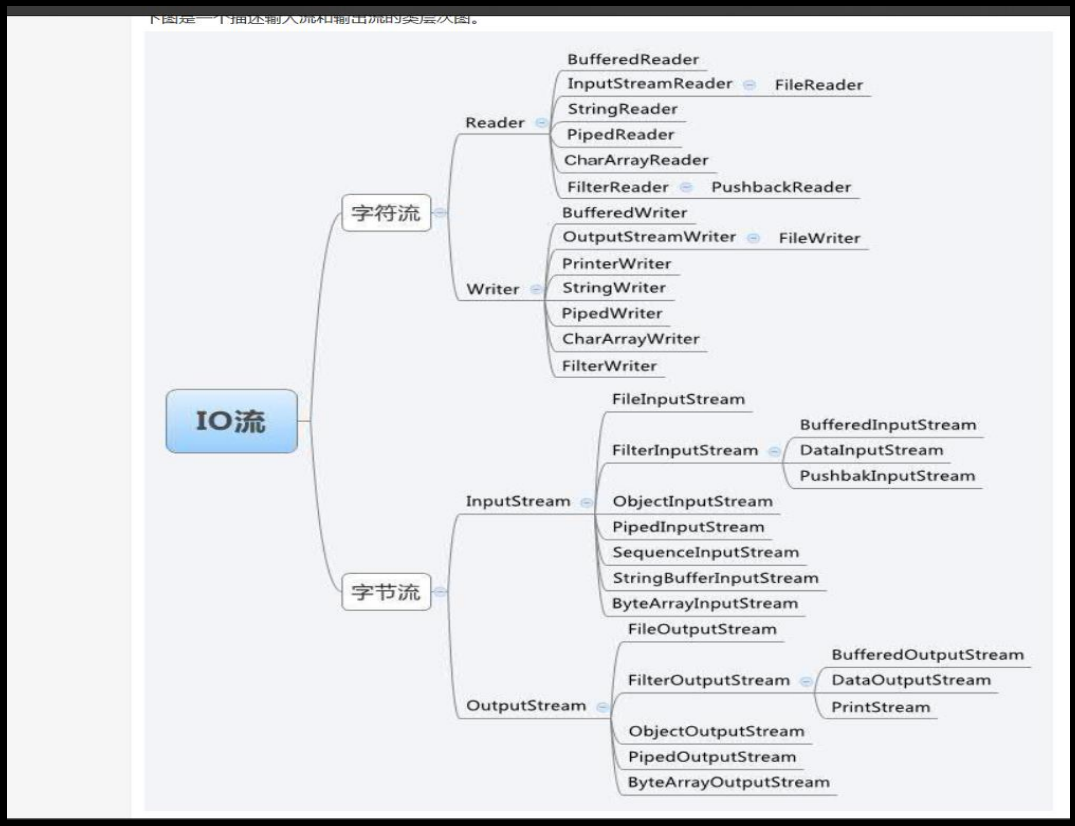
Scanner 对象使用后要关闭: **scan.close（）**

控制台输出：

System.out.println()

System.out.write()

文件操作



后面看吧，书上比较后面

异常处理

异常是错误，错误不都是异常，一些错误可避免

原因分类：非法输入、文件不存在、网络中断、内存溢出

类型：检查型异常、运行时异常、错误

Exception 类：

所有异常是从 Java.lang.Excetion 类继承的子集

|--Error 运行时环境发生变化程序处理范围之外

Throwable |--IOException

|--**Exception**

|--RuntimeException

内置异常：

非检查性异常：

ArithmeticException	当出现异常的运算条件时，抛出此异常。例如，一个整数"除以零"时，抛出此类的一个实例。
ArrayIndexOutOfBoundsException	用非法索引访问数组时抛出的异常。如果索引为负或大于等于数组大小，则该索引为非

	法索引。
ArrayStoreException	试图将错误类型的对象存储到一个对象数组时抛出的异常。
ClassCastException	当试图将对象强制转换为不是实例的子类时，抛出该异常。
IllegalArgumentException	抛出的异常表明向方法传递了一个不合法或不正确的参数。
IllegalMonitorStateException	抛出的异常表明某一线程已经试图等待对象的监视器，或者试图通知其他正在等待对象的监视器而本身没有指定监视器的线程。
IllegalStateException	在非法或不适当的时间调用方法时产生的信号。换句话说，即 Java 环境或 Java 应

	用程序没有处于请求操作所要求的适当状态下。
IllegalThreadStateException	线程没有处于请求操作所要求的适当状态时抛出的异常。
IndexOutOfBoundsException	指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。
NegativeArraySizeException	如果应用程序试图创建大小为负的数组，则抛出该异常。
NullPointerException	当应用程序试图在需要对象的地方使用 null 时，抛出该异常
NumberFormatException	当应用程序试图将字符串转换成一种数值类型，但该字符串不能转换为适当格式时，抛

	出该异常。
SecurityException	由安全管理器抛出的异常，指示存在安全侵犯。
StringIndexOutOfBoundsException	此异常由 String 方法抛出，指示索引或者为负，或者超出字符串的大小。
UnsupportedOperationException	当不支持请求的操作时，抛出该异常。

检查性异常：

ClassNotFoundException	应用程序试图加载类时，找不到相应的类，抛出该异常。
CloneNotSupportedException	当调用 Object 类中的 clone 方法克隆对象，但该对象的类无法实现 Cloneable 接口时，抛出该异常。

IllegalAccessException	拒绝访问一个类的时候，抛出该异常。
InstantiationException	当试图使用 Class 类中的 newInstance 方法创建一个类的实例，而指定的类对象因为是一个接口或是一个抽象类而无法实例化时，抛出该异常。
InterruptedException	一个线程被另一个线程中断，抛出该异常。
NoSuchFieldException	请求的变量不存在
NoSuchMethodException	请求的方法不存在

Throwable 类方法：

1	**public String getMessage()** 返回关于发生的异常的详细信息。这个消息在 Throwable 类的构造函数中初始化了。
2	**public Throwable getCause()** 返回一个 Throwable 对象代表异常原因。
3	**public String toString()**

	使用 getMessage()的结果返回类的串级名字。
4	**public void printStackTrace()** 打印 toString()结果和栈层次到 System.err，即错误输出流。
5	**public StackTraceElement [] getStackTrace()** 返回一个包含堆栈层次的数组。下标为 0 的元素代表栈顶，最后一个元素代表方法调用堆栈的栈底。
6	**public Throwable fillInStackTrace()** 用当前的调用栈层次填充 Throwable 对象栈层次，添加到栈层次任何先前信息中。

捕获异常：try-catch

```
try{
//保护代码
}catch（ExceptionName  e）
{
//捕获异常类型的的声明
}catch(exce…  e1){     //多重
}
```

throws/throw 关键字：

如果一个方法没有捕获一个检查性异常，那么该方法必须使用 throws 关键字来声明。throws 关键字放在方法签名的尾

部。
也可以使用 throw 关键字抛出一个异常，无论它是新实例化的还是刚捕获到的。
一个方法可以声明抛出多个异常，多个异常之间用逗号隔开

finally 关键字用来创建在 try 代码块后面执行的代码块。
无论是否发生异常，finally 代码块中的代码总会被执行。
在 finally 代码块中，可以运行清理类型等收尾善后性质的语句。
finally 代码块出现在 catch 代码块最后，