String x="nihao3woshi9linran0this89is990non3pppp1111";

StringTokenizer y=new StringTokenizer(x, "123456789");

while(y.hasMoreTokens()) {

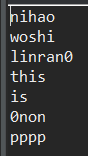
System.out.println(y.nextToken());

}

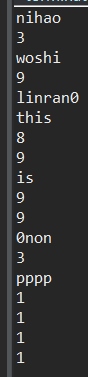
将第一个参数分割，分割标记默认是空格，制表符，换行符，回车符

如果有第二个参数，则分隔符变为第二个参数字符串中的任意字符

第三个参数为true，nextToken()中分隔符也包括。为false，则不包括

示例代码打印

StringTokenizer y=new StringTokenizer(x, "123456789", true);

打印

**正则表达式**

^\d+(\.\d+)?

^ 定义了以什么开始

\d+ 匹配一个或多个数字

? 设置括号内的选项是可选的

\. 匹配 "."

可以匹配的实例："5", "1.5" 和 "2.21

在其他语言中，\\ 表示：我想要在正则表达式中插入一个普通的（字面上的）反斜杠，请不要给它任何特殊的意义。

在 Java 中，\\ 表示：我要插入一个正则表达式的反斜线，所以其后的字符具有特殊的意义。

所以，在其他的语言中（如Perl），一个反斜杠 \ 就足以具有转义的作用，而在 Java 中正则表达式中则需要有两个反斜杠才能被解析为其他语言中的转义作用。也可以简单的理解在 Java 的正则表达式中，两个 \\ 代表其他语言中的一个 \，这也就是为什么表示一位数字的正则表达式是 \\d，而表示一个普通的反斜杠是 \\\\。

\ [\\d](file:///\\d) \\( \\) \\\\

^ 匹配输入字符串开始的位置

$ 匹配输入字符串结尾的位置。

\* 零次或多次匹配前面的字符或子表达式，等效于 {0,}。

+ 一次或多次匹配前面的字符或子表达式，等效于 {1,}。

? 零次或一次匹配前面的字符或子表达式。等效于 {0,1}。

{n} 正好匹配 n 次

{n,} 至少匹配 n 次。"o{0,}"等效于"o\*"。

{n,m} 匹配至少 n 次，至多 m 次。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"。跟在括号后的话就不是这种用法

. 匹配除"\r\n"之外的任何单个字符

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

[^xyz] 反向字符集。匹配未包含的任何字符

[a-z] 字符范围。匹配指定范围内的任何字符

[^a-z] 反向范围字符。匹配不在指定的范围内的任何字符

\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]

\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"。\\bcat\\b匹配cat单词

\B 非字边界匹配。"er\B"匹配"verb"中的"er"，但不匹配"never"中的"er"。

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches

{

public static void main( String args[] ){

// 1.创建 Pattern 对象

Pattern p = Pattern.compile("[\\bcat\\b](file:///\\bcat\\b)");

//2.创建Matcher对象

Matcher m = p.matcher("cat cat cat cattie cat");

//m.matches()返回 整个字符串 匹不匹配

System.out.println(m.matches());

//m.lookingAt()返回 从第一个字符开始的部分字符串 匹不匹配

System.out.println(m.lookingAt());

int count = 0;

while(m.find()) {

//find()尝试查找与该模式匹配的输入序列的下一个子序列

count++;

System.out.println("Match number "+count);

//start()返回以前匹配的初始索引

System.out.println("start(): "+m.start());

//end()返回结束索引+1

System.out.println("end(): "+m.end());

//m.group(0)提取字符串

}

}

}

还有一些替换的方法，不写了

false

true

Match number 1

start(): 0

end(): 3

Match number 2

start(): 4

end(): 7

Match number 3

start(): 8

end(): 11

Match number 4

start(): 19

end(): 22

**重载**(overloading) 是在一个类里面，方法名字相同，而参数不同。返回类型可以相同也可以不同。

public class Overloading {

public int test(){

System.out.println("test1");

return 1;

}

public void test(int a){

System.out.println("test2");

}

}

Eclipse

Debug线程堆栈视图

Variables变量视图

BreakPoints断点视图

Expressions表达式视图

List<> values=new ArrayList<>();

values.add();

**数据类型**

1. 基本数据类型

* boolean
* char 和ASCII码对应 2字节
* byte 1字节

-128~127

* short 2字节

-2^15~2^15-1

2^15=32768

2^16=65536

* int 4字节

-2^31~2^31-1

-2147483648~2147483647 10位

2^32=4294967296

* long 8字节

-2^63~2^63-1

-9223372036854775808 19位

2^64=18446744073709551616 20位

* double 8字节

2^1024 308位

2^2048 616位

* float 4字节

2^128 38位

2^256 77位

2. 引用类型

数组

类

接口

* 未声明数据类型的整形，默认为int型
* 未声明数据类型的浮点型，默认为double型。

float a=1.5;

ERROR! Type mismatch: cannot convert from double to float

一个数值范围小的类型，赋给一个数值范围大的数值型变量，jvm在编译过程中将此数值的类型进行了自动提升。

在数值类型的自动类型提升过程中，数值精度至少不应该降低

但是byte a=5;可以，jvm隐式转换了。整型的是特例

long a = 10000000000; //编译出错: The literal 10000000000 of type int is out of range

long b = 10000000000L; //编译正确

10000000000默认是int类型， int类型的数值范围是-2^31 ~ 2^31-1，因此，10000000000已经超过此范围。故而其自身已经编译出错，更谈不上赋值给long型变量a了。

将一个大数值范围的变量，赋给小范围的变量，可能丢失精度。需要强制转换

byte p=3;//隐式类型转换

int a=0;

byte b=a; //error，注意这种情况不发生隐式转换

byte c=(byte)a;

float d=(float)4.0;

byte a1=(byte)128; //-128

int i = Integer.parseInt("13");

String s1 = String.valueOf(123);

int i;

i=-i;

这种操作时不安全的，如果i=-2147483648