Text files in Java

Peter Sestoft, Department of Mathematics and Physics
Royal Veterinary and Agricultural University, Denmark

English version 1.01, 1998-08-29

This note describes how to read and write text files in Java.

Contents

1 Reading words and numbers from a text file

A text file is just a sequence of characters, such as `'A'`, `'B'`, `'7'`, `'9'`, `'+'`, `'='`, `' '`, and so on. Note especially the last example: it is the space character, which prints as a blank. Newline is also a character, written in Java as `'\n'`. In MS DOS and MS Windows a newline in fact consists of two characters: `'\n'` and `'\r'`, corresponding to newline and carriage return — a reminder of the times when electrical typewriters were used for computer output. In any case, Java pretends that a newline consists of a single character.

Thus the two-line text file:

```
Ole 312.2
Ib 11117
```

consists of the the following characters:

O	l	e		3	1	2	.	2	\n	I	b		1	1	1	1	7	\n

When reading from a text file, one usually does not want to handle the characters one by one, but rather to collect them into numbers and words. For instance, the five characters `'3'` `'1'` `'2'` `'.'` `'2'` should be considered together, as the number 312.2.

For this purpose one may use the Java library class StreamTokenizer. Numbers and words in the input are called *tokens*, and a text file can be considered a stream of tokens. Hence the name StreamTokenizer: it turns a character stream into a token stream.

In the above text file, two tokens are separated by *whitespace*: space (blank), tabulators, newline, or form feed (which skips to a new page). The file consists of the following tokens:

`Ole`	`312.2`	`Ib`	`11117`	`TT_EOF`

The nineteen characters in the text file have been grouped to form four tokens, the whitespace (spaces and newlines) has been thrown away, and a special fifth token `TT_EOF` marks the end of the token stream. The letters eof in `TT_EOF` abbreviate 'end of file', and tt abbreviates 'token type'.

1.1 Reading numbers

Assume we have a text file called `"numbers.txt"` with the following contents:

```
14.5 20  18
19.1
```

We want to read the contents of this file as numerical tokens (followed by end of file, `TT_EOF`):

`14.5`	`20`	`18`	`19.1`	`TT_EOF`

Assume we want to read the numbers from the file and compute their sum. We may use the following program fragment:

```
Reader inp = new FileReader("numbers.txt");                // 1
StreamTokenizer tstream = new StreamTokenizer(inp);        // 2
tstream.parseNumbers();                                    // 3
double sum = 0;                                            // 4
tstream.nextToken();                                       // 5
while (tstream.ttype != StreamTokenizer.TT_EOF)            // 6
  {
    sum += tstream.nval;                                   // 7
    tstream.nextToken();                                   // 8
  }
```

Explanation:

1 The text file `"numbers.txt"` is opened and connected to a character stream (a reader) called `inp`.
2 The character stream `inp` is converted into a token stream, called `tstream`.
3 Tokens that look like numbers (e.g. 14.5) should be read as numbers, not as strings (e.g. `"14.5"`).
4 So far the sum is 0: no numbers have been read yet.
5 Read the first token from `tstream`.
6 If the current token is not `TT_EOF` (end of file) but a number, then execute the loop body in lines 7 and 8. Otherwise, terminate the loop (at which point the variable `sum` will contain the sum of all the numbers).
7 The number represented by the current token is obtained as `tstream.nval`, which is added to the sum.
8 Try to read a new token from `tstream`, and continue at line 6.

1.2 Reading numbers line by line

If one wants to compute the sum of numbers line by line, then one needs two while loops. The outer loop performs one iteration for each line of text, and the inner loop performs one iteration for each number on the line.

For instance, one might want to compute the sum for each line in the text file `"numbers.txt"`, storing all the sums in a single string `sums`.

To do this, the program needs to 'see' where each line ends. Fortunately, newline can be considered a separate token (rather than whitespace, as above):

14.5	20	18	TT_EOL	19.1	TT_EOL	TT_EOF

Line A below tells the tokenizer that newline must be considered a token, not whitespace. The lines B through I contains the nested `while` loops which read the file:

```
String sums = "";
Reader inp = new FileReader("numbers.txt");               // 1
StreamTokenizer tstream = new StreamTokenizer(inp);       // 2
tstream.parseNumbers();                                   // 3
tstream.eolIsSignificant(true);                           // A
tstream.nextToken();                                      // B
while (tstream.ttype != StreamTokenizer.TT_EOF)           // C
 {
   double sum = 0;                                        // D
   while (tstream.ttype != StreamTokenizer.TT_EOL)        // E
     {
       sum += tstream.nval;                               // F
       tstream.nextToken();                               // G
     }
   sums += sum + " ";                                     // H
   tstream.nextToken();                                   // I
 }
```

Explanation:

A. A newline (eol = end of line) must be considered a token, not whitespace, in `tstream`.
B. Read the first token from `tstream`.
C. If the current token is not `TT_EOF` (end of file), then execute the loop body in lines D–I. Otherwise terminate the loop.
D. So far the line sum is 0: no number has been read yet.
E. If the current token is not `TT_EOL` (end of line), but a number, then execute the body of the inner loop in lines F and G. Otherwise, terminate the inner loop and go to H.
F. The number represented by the current token, `tstream.nval`, is added to the line sum.
G. Try to read a new token from `tstream`, and continue the inner loop at line E.
H. When the inner loop terminates, line H is reached. Now `sum` is the sum for the current line, and is appended (as a text) to the sums of previous lines.
I. Try to read a new token from `tstream`, and continue the outer loop at line C.

1.3 Reading entire lines of text

If one needs to read entire lines of text, regardless of spaces, then one must tell the tokenizer that space `' '` must be considered part of a word, not whitespace. This is done by the method call `tstream.wordChars(' ', ' ')`. If one reads `numbers.txt` in this way, then every line becomes one (large) token:

`14.5 20   18`	`19.1`	`TT_EOF`

If one want to consider newlines as tokens too, then one may specify `eolIsSignificant(true)` as in the above example.

Reading the file `numbers.txt` in this way is fairly useless. On the other hand, when reading a text file `"addrlist.txt"` containing a list of addresses, it makes good sense to read entire lines at a time. In this example, we store the lines in an array `arr` of strings:

```
String[] arr = new String[100];
Reader inp = new FileReader("addrlist.txt");
StreamTokenizer tstream = new StreamTokenizer(inp);
tstream.wordChars(' ', ' ');
int n = 0;
tstream.nextToken();
while (n < arr.length && tstream.ttype != StreamTokenizer.TT_EOF)
  {
    arr[n] = tstream.sval; tstream.nextToken();
    n++;
  }
```

1.4 Declaring imports

When using StreamTokenizer, Reader, or FileReader, one must specify

```
import java.io.*;
```

at the beginning of the file.

1.5 Declaring exceptions

When reading the input, a so-called *exception* may be *thrown* by the program, signalling that an error occurred. For instance, an exception of class IOException may be thrown if one attempts to read further tokens after `TT_EOF`. Also, an exception of class FileNotFoundException, which is a subclass of IOException, may be thrown if one attempts to open a non-existing file for reading.

A thrown exception may be *caught* by a `catch` statement. An exception *escapes* from a method if it may be thrown without being caught during execution of the method. In Java, every method must declare what exceptions can escape from it. Therefore a method that reads from a file should be declared as follows:

```
public static void main(String[] args)
   throws IOException
{ ... }
```

The line `throws IOException` in the method head says that exceptions belonging to class IOException (and its subclasses) may escape this method.

1.6 Buffered reading for faster input

The above recipes for reading input from text files work, but may be slow. The problem is that every request for a new token (using `nextToken()`) may lead to a request for a few more characters from the character stream, which may lead to a request to the operating system for a few more characters from the text file. This is slow.

To speed up reading, turn the character reader into a buffered character reader, so that larger chunks of text can be read from the file at a time:

```
Reader inp = new FileReader("numbers.txt");
inp = new BufferedReader(inp);
StreamTokenizer tstream = new StreamTokenizer(inp);
```

For brevity, we do not use buffering in examples.

1.7 Summary of class StreamTokenizer

The following operations on a streamtokenizer `t` are used frequently:

Creating a streamtokenizer	
`t.parseNumbers()`	Let `t` parse numbers. A number token is a 'word' beginning with a digit (0–9) or a decimal point (.) or the minus sign (−). Number tokens have type `TT_NUMBER`.
`t.eolIsSignificant()`	Let `t` consider newline as a separate token `TT_EOL`, not as whitespace.
`t.whitespaceChars(c1, c2)`	Let `t` consider the characters in the interval `c1`–`c2` (inclusive) as whitespace also, that is, as token separators.
`t.wordChars(c1, c2)`	Let `t` consider the characters in the interval `c1`–`c2` (inclusive) as parts of words also, not as whitespace.
Reading from a streamtokenizer	
`t.nextToken()`	Read the next (or first) token from `t`
`t.nval`	The number value of the current token, as a `double`
`t.sval`	The string value of the current token, as a `String`
`t.ttype`	The type of the current token (see below)

Every token has a 'type'. The four types of tokens are

`TT_NUMBER`	number
`TT_WORD`	word
`TT_EOL`	newline
`TT_EOF`	end of file (no more tokens)

2 Reading from the keyboard

The streamtokenizer can be used to read from the keyboard too. This is particularly useful for writing command line applications in Java. To read from the keyboard, one creates a character stream `inp` from `System.in`, the standard input stream of the system:

```
Reader inp = new InputStreamReader(System.in);
```

This change affects only the first line of the examples above (Section 1.1, 1.2, and 1.3).

In MS DOS and MS Windows, the end of file marker `TT_EOF` can be entered from the keyboard as ctrl-Z followed by newline.

3 Reading from a text string

A streamtokenizer can read from a text string, too. This is particularly useful for when writing Java applications with graphical user interfaces, or applets, since the contents of a TextField or TextArea is given as a text string. To read from a string `s`, one creates a character stream `inp` from the string:

```
Reader inp = new StringReader(s);
```

This change affects only the first line of the examples above (Section 1.1, 1.2, and 1.3).

4 Formatted text output

Unless one takes special care, the output from a program may be as ugly as this:

```
Odense 17.5
Assens 19.1
Slagelse 19.775000000000002
Longyearbyen 8.7
```

In this section, we shall see how to control the number of digits after the decimal point, and how to align data in columns.

The Java libraries (version 1.1) include a package `java.text` which provides for formatted output. Here we give only a brief description; there are many more advanced possibilities.

4.1 Format patterns for fixed-comma output of numbers

To format numbers with a specified number of digits after the decimal point, one uses a *format pattern*, such as `"0.00"` to create a DecimalFormat object `fmt`, then uses `fmt` to format the number as a string:

```
DecimalFormat fmt = new DecimalFormat("0.00");
System.out.println(fmt.format(3.1415926));
```

The format pattern `"0.00"` requests that there should be two digits after the decimal point, and at least one before it. Thus the output will be `3.14` in the above example.

More generally, the characters in a format pattern have the following meaning:

Character	Meaning
`#`	any number of digits; zeroes are left blank
`0`	at least one one digit; zeroes are printed
`.`	the decimal point symbol
`,`	the thousands (digit grouping) symbol

The effect[1] of some typical format patterns is illustrated by this table:

	Format patterns							
Number	`#`	`#.#`	`#.##`	`0.0`	`0.0#`	`0.00`	`000.0`	`#,##0.00`
0.0	`0`			`0.0`	`0.0`	`0.00`	`000.0`	`0.00`
0.1	`0`	`.1`	`.1`	`0.1`	`0.1`	`0.10`	`000.1`	`0.10`
1.0	`1`	`1`	`1`	`1.0`	`1.0`	`1.00`	`001.0`	`1.00`
1.1	`1`	`1.1`	`1.1`	`1.1`	`1.1`	`1.10`	`001.1`	`1.10`
-1.1	`-1`	`-1.1`	`-1.1`	`-1.1`	`-1.1`	`-1.10`	`-001.1`	`-1.10`
330.8	`331`	`330.8`	`330.8`	`330.8`	`330.8`	`330.80`	`330.8`	`330.80`
1234.516	`1235`	`1234.5`	`1234.52`	`1234.5`	`1234.52`	`1234.52`	`1234.5`	`1,234.52`

[1] This output is from JDK 1.1.6 under Linux. Presumably, this is how the output *should* look, but it is *not* what is produced by JDK 1.1.6 under MS Windows. Netscape Communicator appears to work correctly.

The same DecimalFormat object `fmt` may be used to format several numbers. For instance, this program fragment would produce the last column above:

```
DecimalFormat fmt = new DecimalFormat("#,##0.00");

double[] arr = { 0, 0.1, 1.0, 1.1, -1.1, 330.8, 1234.516 };
for (int i=0; i < arr.length; i++)
  System.out.println(fmt.format(arr[i]));
```

4.2 Danish number formats

Traditional Danish number formatting has the opposite conventions of English and American: Danish uses the comma `','` as decimal separator and the point `'.'` as thousands separator.

The Java libraries attempt produce the proper 'national' output, and so will normally use the Danish conventions when run under e.g. a Danish version of MS Windows, or a Danish version of Netscape Communicator or MS Internet Explorer.

If you want to make sure that the output is always produced using particular symbols, then you may create a DecimalFormatSymbols object and use that when creating a DecimalFormat object. The format pattern remains unchanged. To always produce Danish output, change the above program as follows:

```
DecimalFormatSymbols decsyms = new DecimalFormatSymbols();
decsyms.setDecimalSeparator(',');
decsyms.setGroupingSeparator('.');
DecimalFormat fmt = new DecimalFormat("#,##0.00", decsyms);

double[] arr = { 0, 0.1, 1.0, 1.1, -1.1, 330.8, 1234.516 };
for (int i=0; i < arr.length; i++)
  System.out.println(fmt.format(arr[i]));
```

The output will be a Danish version of the last column in the above table:

```
0,00
0,10
1,00
1,10
-1,10
330,80
1.234,52
```

4.3 Declaring imports

When using DecimalFormat, etc. one must specify

```
import java.text.*;
```

at the beginning of the file.

4.4 Padding text on the left and right

As shown by the examples above, the DecimalFormat class does not align the formatted numbers on the decimal point. A simple way to do this is to pad the numbers with sufficiently many blanks on the left so that they align on the right, regardless whether they take up three or eight characters when printed. Similarly, words may be padded with sufficiently many blanks on the right, so that the next column aligns correctly.

For this to produce the desired visual effect, your output device must use a fixed-pitch font, in which all characters have the same width. This is usually the case for the typewriter font (often called Courier).

A Java method to pad a strings on the left can be defined as follows, using a string buffer:

```
public static String padLeft(String s, int width) {
  int filler = width - s.length();
  if (filler > 0)          // and therefore width > 0
    {
      StringBuffer res = new StringBuffer(width);
      for (int i=0; i<filler; i++)
        res.append(' ');
      return res.append(s).toString();
    }
  else
    return s;
}
```

If `s` has length `width` or more, then `padLeft(s, width)` just returns `s`. Otherwise, it returns a new string which consists of `s`, padded with sufficiently many blanks on the left to have length `width`.

Similarly, the method below pads `s` on the right:

```
public static String padRight(String s, int width) {
  int filler = width - s.length();
  if (filler > 0)          // and therefore width > 0
    {
      StringBuffer res = new StringBuffer(width).insert(0, s);
      for (int i=0; i<filler; i++)
        res.append(' ');
      return res.toString();
    }
  else
    return s;
}
```

5 Output to a text file

For output to a text file, create a FileWriter from a given file name, then create a PrintWriter from the FileWriter:

```
Writer wri = new FileWriter("output.txt");
PrintWriter out = new PrintWriter(wri);
```

The PrintWriter object `out` behaves just the same as the well-known `System.out`. That is, one may just replace `System.out` with `out` and execute `out.println("This is a text")`, or `out.print(12.3)` and similar. After one has finished printing to the file, one must close the PrintWriter by executing `out.close()`. Otherwise there is a risk that some or all of the output may be missing from the text file.

For example, the following program fragment prints the numbers 0 through 19 and their squares and cubes on the text file `"output.txt"`:

```
Writer wri = new FileWriter("output.txt");
PrintWriter out = new PrintWriter(wri);
for (int i=0; i<20; i++)
  out.println(i + " " + i*i + " " + i*i*i);
out.close();
```

5.1 Declaring imports

When using PrintWriter, Writer, or FileWriter, one must specify

```
import java.io.*;
```

at the beginning of the file.

5.2 Declaring exceptions

When writing the output, an exception may be thrown by the program, signalling that an error occurred (see Section 1.5).

Therefore a method that writes to a file should include a `throws` clause:

```
public static void main(String[] args)
   throws IOException
{ ... }
```

5.3 Buffered writing for faster output

As when reading from a file (see Section 1.6), one may increase speed by turning the FileWriter into a buffered writer before creating a PrintWriter:

```
Writer wri = new FileWriter("output.txt");
wri = new BufferedWriter(wri);
PrintWriter out = new PrintWriter(wri);
```

6 A complete example of text input and output

Assume there is a text file "places.txt" in which each line contains the name of a town (in one word), and a sequence of numbers, e.g. temperatures:

```
Odense 14.5 20  18
Assens 19.1
Slagelse    23.1 25.1 12.1 18.8
Longyearbyen 8.1  10.2 7.8
...
```

For each line we want to print the name and the average of the numbers. In the printout the name must be to the left on the line, and the average should be printed with two digits after the decimal point, to the right on the line. We may assume that no name is longer than 30 characters, and that the average can be printed in a field 10 characters wide (including the decimal point).

```
public static void main(String[] args)
     throws FileNotFoundException, IOException            // 0
{
  Reader inp = new FileReader("places.txt");              // 1
  StreamTokenizer tstream = new StreamTokenizer(inp);     // 2
  tstream.parseNumbers();                                 // 3
  tstream.eolIsSignificant(true);                         // 4
  tstream.nextToken();                                    // 5
  while (tstream.ttype != StreamTokenizer.TT_EOF)         // 6
    {
      double sum = 0;                                     // 7
      int count = 0;                                      // 8
      String name = tstream.sval;                         // 9
      tstream.nextToken();                                // 10
      while (tstream.ttype != StreamTokenizer.TT_EOL)     // 11
        {
          sum += tstream.nval;                            // 12
          count++;                                        // 13
          tstream.nextToken();                            // 14
        }
      double avg = sum / count;                           // 15
      DecimalFormat fmt = new DecimalFormat("0.00");      // 16
      System.out.println(padRight(name, 30)               // 17
                         + padLeft(fmt.format(avg), 10)); // 18
      tstream.nextToken();                                // 19
    }
}
```

Explanation:

0 Declare that the method may throw the exceptions `FileNotFoundException` and `IOException`, as discussed in Section 1.5.

1–7 Correspond to line 1–D in Section 1.2.

8 At the beginning of the line we have not read any numbers yet, so `count` is 0.

9–10 The line's first token is the name (a string), which is read using `sval`.

11–14 Correspond to line E–G in Section 1.2, except that `count` must be incremented.

16 A formatter `fmt` is created for fixed-comma formatting with two digits after the decimal point.

17 The name is padded on the right to make it 30 characters wide.

18 The average is formatted using the formatter `fmt`, and is padded on the left to make it 10 characters wide.

19 The next token is read.

The output of the above program will be a neat version of that shown at the beginning of Section 4:

```
Odense                             17.50
Assens                             19.10
Slagelse                           19.78
Longyearbyen                        8.70
...
```

7 Exercises

1. Modify the program in Section 6 so that the name of a town may consist of more than one word (e.g., 'Los Angeles', or 'Frankfurt an der Oder').
2. Write a program to print a neatly formatted table of accumulated savings. For instance, if we set aside 1,000 dollars every year at 5 per cent annual interest, we should get:

```
year     balance
  1      1,050.00
  2      2,152.50
  3      3,310.13
  4      4,525.63
  ...
```