

The StringTokenizer Class

In our demo programs thus far, input from the keyboard was processed one line at a time. Each line was converted to a `String` and operated on using available methods. However, it is often necessary to decompose a full line of input into *tokens* that are processed separately. Each token might be a word, for example. The `StringTokenizer` class in the `java.util` package provides services for this purpose.

By default, tokens are delimited by *whitespace* characters, but other delimiters are also possible. A whitespace is a space, return, tab, or newline character. The default delimiters are held in a string, "`\r\t\n`", which can be changed using methods of the `StringTokenizer` class.

Unlike the `Math` class, objects of the `StringTokenizer` class can be instantiated; thus, the `StringTokenizer` class includes constructor methods. The methods of the `StringTokenizer` class are summarized in Table 1.

Table 1. Methods of the `StringTokenizer` Class

Method	Description
Constructors	
<code>StringTokenizer(String str)</code>	Constructs a string tokenizer for the specified string <code>str</code> ; returns a reference the new object
<code>StringTokenizer(String str, String delim)</code>	Constructs a string tokenizer for the specified string <code>str</code> using <code>delim</code> as the delimiter set
<code>StringTokenizer(String str, String delim, boolean returnTokens)</code>	Constructs a string tokenizer for the specified string <code>str</code> using <code>delim</code> as the delimiter set; returns the tokens with the string if <code>returnTokens</code> is true
Instance Methods	
<code>countTokens()</code>	returns an <code>int</code> equal to the number of times that this tokenizer's <code>nextToken()</code> method can be called before it generates an exception
<code>hasMoreTokens()</code>	returns a <code>boolean</code> value: <code>true</code> if there are more tokens available from this tokenizer's string, <code>false</code> otherwise
<code>nextToken()</code>	returns a <code>String</code> equal to the next token from this string tokenizer
<code>nextToken(String delim)</code>	returns a <code>String</code> equal to the next token from this string tokenizer using <code>delim</code> as a new delimiter set

Count Words

Let's begin with a demo program. A basic tokenizing operation is to divide lines of input into words. The program `CountWords` demonstrates how this is done using methods of the `StringTokenizer` class (see Figure 1).

```

1  import java.io.*;
2  import java.util.*;
3
4  public class CountWords
5  {
6      public static void main(String[] args) throws IOException
7      {
8          BufferedReader stdin =
9              new BufferedReader(new InputStreamReader(System.in), 1);
10
11         String line;
12         StringTokenizer words;
13         int count = 0;
14
15         // process lines until no more input
16         while ((line = stdin.readLine()) != null)
17         {
18             // process words in line
19             words = new StringTokenizer(line);
20             while (words.hasMoreTokens())
21             {
22                 words.nextToken();
23                 count++;
24             }
25         }
26         System.out.println("\n" + count + " words read");
27     }
28 }

```

Figure 1. CountWords.java

A sample dialogue with this program follows:

```

PROMPT>java CountWords
To be or not to be.
That is the question.
^Z(or ^D)
10 words read

```

In the sample dialogue, two lines of text containing 10 words were inputted via the keyboard. The third line of input shows "^Z" for "Control-Z". This is entered by depressing the control key while entering "Z". When executing a Java program in a DOS window, Control-Z indicates an end-of-input or end-of-file condition. When Control-Z is detected, the `readLine()` method returns `null`. This condition signals that there is no more input available for processing. On systems running a *unix*-like operating system (e.g., *linux* or *Solaris*) an end-of-file condition is entered as Control-D.

In line 2, an `import` statement informs the compiler of the location of the `StringTokenizer` class. (It is in the `java.util` package.) Line 12 contains a declaration of an object variable named `words` of the `StringTokenizer` class. The while loop in lines 16-25 performs the task of inputting lines from the keyboard until a `null` (end-of-input) condition occurs. The relational expression in line 16 conveniently combines inputting a line of input with the relational test for `null`. Since assignment is of lower precedence than the "not equal to" (`!=`) relational operator, parentheses are required for the assignment expression.

Each line inputted is converted to a `String` object. A reference to the object is assigned to object variable `line` (lines 16). Line 19 instantiates a `StringTokenizer` object from the

string `line` and assigns a reference to it to the object variable `words`. This sets-up the conditions to begin processing the tokens — the words — in each line of text input. Two instance methods of the `StringTokenizer` class do all the work. The `hasMoreTokens()` method returns a `boolean` (`true` or `false`) indicating if there are more tokens to be processed. This method is called in line 20 via the instance variable `words` within the relational expression of an inner `while` loop. So, the outer `while` loop processes lines, the inner `while` loop processes the tokens in each line.

In the inner `while` loop, line 22 contains an expression to retrieve the next token in the `StringTokenizer` object `words`. Note that the token is retrieved, but is not assigned to anything. This is fine, because our task is simply to count words; we aren't interested in processing the words in any manner. The counting occurs in line 23, where the `int` variable `count` is incremented for each token retrieved. After both loops are finished the result is printed (line 26).¹

Count Alpha-only Words

For our next example, we'll make a small but interesting change to `CountWords`. Suppose we want to count words if they contain only letters of the alphabet. To count only "alpha" words, we must not only tokenize strings into words, but each token must be inspected character-by-character to determine if all characters are letters. The demo program `CountWords2` shows the necessary modifications to perform this task (see Figure 2).

¹ The presence of a leading newline character (“\n”) in the print statement is a fix for a quirk in *Windows*. If input arrives from the keyboard followed by Control-Z, the first line of subsequent output is overwritten by the operating system. The leading newline character ensures that the result printed is not overwritten.

```

1  import java.io.*;
2  import java.util.*;
3
4  public class CountWords2
5  {
6      public static void main(String[] args) throws IOException
7      {
8          BufferedReader stdin =
9              new BufferedReader(new InputStreamReader(System.in), 1);
10
11         String line;
12         StringTokenizer words;
13         int count = 0;
14
15         // process lines until no more input
16         while ((line = stdin.readLine()) != null)
17         {
18             // process words in line
19             words = new StringTokenizer(line);
20             while (words.hasMoreTokens())
21             {
22                 String word = words.nextToken();
23                 boolean isAlphaWord = true;
24                 int i = 0;
25
26                 // process characters in word
27                 while (i < word.length() && isAlphaWord)
28                 {
29                     String c = word.substring(i, i + 1).toUpperCase();
30                     if (c.compareTo("A") < 0 || c.compareTo("Z") > 0)
31                         isAlphaWord = false;
32                     i++;
33                 }
34                 if (isAlphaWord)
35                     count++;
36             }
37         }
38         System.out.println("\n" + count + " alpha words read");
39     }
40 }

```

Figure 2. CountWords2.java

A sample dialogue with this program follows:

```

PROMPT>java CountWords2
Maple Leafs 7 Flyers 1
^z
3 alpha words read

```

Although the inputted line has five tokens, only three are alpha-only. Lines 22-35 perform the added task of determining if each token contains only letters. The first task is to retrieve a token as the `String` object `word` (line 22). Initially, we assume `word` contains only letters, so the boolean variable `isAlphaWord` is declared and set to `true` (line 23). Then, each character in `word` is tested (lines 27-33) using the `compareTo()` method of the `String` class. If a character is less than 'A' or greater than 'Z', it is not a letter and `isAlphaWord` is set false. This also causes an early termination of the inner while loop. If all characters are checked and `isAlphaWord` is still true, then the word is, indeed, alpha-only and `count` is incremented (line 35).

Determining if a word contains only letters, as in lines 27-33 of `CountWords2`, uses methods seen in earlier programs. In fact, this task can be simplified:

```
for (int i = 0; i < word.length(); i++)
    if (!Character.isLetter(word.charAt(i)))
        isAlphaWord = false;
```

The work is now performed by the `isLetter()` method of the `Character` wrapper class (see Appendix C). The `isLetter()` method receives a `char` argument and returns a `boolean`: `true` if the character is a letter, `false` otherwise. The `char` argument is obtained via the `charAt()` method of the `String` class. It receives an integer argument specifying the character to extract from the string. Note that `isLetter()` is a class method, whereas `charAt()` is an instance method. Since we are using `isLetter()` to determine if a character is *not* a letter, the unary NOT (!) operator precedes the relational expression.

The technique above is not only shorter, it also allows the program to work with the Unicode definition of a letter. Thus, the program will work for non-ASCII interpretations of letters, such as occur in scripts other than English.

Find Palindromes

Now that we are comfortable with the `StringTokenizer` class, let's look at another interesting example. A palindrome is a word that is spelled the same forwards as backwards, for example, "rotator". The program `Palindrome` reads lines of input from the keyboard until an end-of-file condition occurs. As each line is read, it is tokenized and each token is inspected to determine if it is a palindrome. If so, it is echoed to the standard output. The listing is given in Figure 3.

```

1  import java.io.*;
2  import java.util.*;
3
4  public class Palindrome
5  {
6      public static void main(String[] args) throws IOException
7      {
8          // open keyboard for input (call it 'stdin')
9          BufferedReader stdin =
10             new BufferedReader(new InputStreamReader(System.in), 1);
11
12         // prepare to extract words from lines
13         String line;
14         StringTokenizer words;
15         String word;
16
17         // main loop, repeat until no more input
18         while ((line = stdin.readLine()) != null)
19         {
20             // tokenize the line
21             words = new StringTokenizer(line);
22
23             // process each word in the line
24             while (words.hasMoreTokens())
25             {
26                 word = words.nextToken();
27                 if (word.length() > 2) // must be at least 3 characters long
28                 {
29                     boolean isPalindrome = true;
30                     int i = 0;
31                     int j = word.length() - 1;
32                     while (i < j && isPalindrome)
33                     {
34                         if ( !(word.charAt(i) == word.charAt(j)) )
35                             isPalindrome = false;
36                         i++;
37                         j--;
38                     }
39                     if (isPalindrome)
40                         System.out.println(word);
41                 }
42             }
43         }
44     }
45 }

```

Figure 3. Palindrome.java

A sample dialogue with this program follows:

```

PROMPT>java Palindrome
a deed worth doing
deed
rotator is a palindrome
rotator
^z

```

Two lines of input are entered and these contain two palindromes, "deed" and "rotator". The structure of the program is very similar to that of CountWords2. Instead of inspecting each

token to determine if it contains only letters, we check if characters are mirrored about an imaginary center point. The general idea is shown in Figure 4 for "rotator".

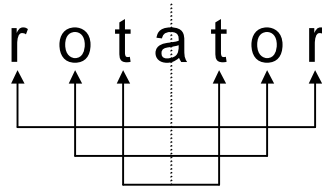


Figure 4. Letter comparisons for palindrome test

Lines 26-40 do the work. First, we set the `boolean` variable `isPalindrome` to `true` with an assumption that the token is a palindrome (line 29). Then, variables `i` and `j` are initialized with the index of the first and last character in the token, respectively (lines 30-31). The `while` loop in lines 32-38 processes pairs of characters starting at the outside, working toward the center. The loop continues only while `i` is less than `j` and `isPalindrome` remains `true`.

The crucial test is in line 34. The `charAt()` method in the `String` class extracts the characters at indices `i` and `j` from the token. These are compared using the "is equal to" relational operator (`==`). Note that `char` variables can be compared as though they were integers. The relational test is inverted with the NOT (`!`) operator. If the overall result is `true`, the characters are *not* equal and the token is *not* a palindrome. In this case, `isPalindrome` is set to `false` (line 35). After each test, `i` is incremented and `j` is decremented, thus moving the indices closer to the center of the token. Tokens with less than two characters are not checked in this program (line 27).

We will re-visit the `Palindrome` program later when we discuss recursion and debugging.