

Redirection and Pipes

Providing input via the keyboard to test demo program is tedious. Conveniently, data can be inputted from a file as though they were typed on the keyboard using *redirection*. The "<" input redirection symbol appended to a command line followed by the name of a file forces input to be read from a file instead of from the keyboard. This is purely as system-level service. No modification to the program is required. The following is an example dialogue for the CountWords2 program using input redirection:

```
PROMPT>java CountWords2 < CountWords2.java
119 alpha words read
```

In this example, the CountWords2 program is executed in the usual way; however input is redirected. Instead of reading input line-by-line from the keyboard, input is read from the file CountWords2.java. (The name of any other text file could be substituted for CountWords2.java.) The end-of-input condition generated by entering Control-Z (or Control-D) on the keyboard is generated automatically by the system when the end of the file is reached. Evidently the version of the CountWords2.java used for the example above contained 119 tokens containing only letters.

Here is another example using the Palindrome program:

```
PROMPT>java Palindrome < Palindrome.java
rotator
```

The only palindrome in the file Palindrome.java is the token "rotator". It does not appear in **Error! Reference source not found.** because the comment lines were deleted to keep the listing short.

Output is redirected by appending ">" and a filename. Output normally sent to the standard output and displayed on the host system's CRT display is written to the specified file instead.

Input redirection and output redirection apply only to the "standard input" and the "standard output". Reading data from files that are opened and read explicitly in a program is unaffected. Likewise writing data to files that are opened and written explicitly in a program is unaffected. "Explicitly" opening and reading or writing data files is discussed later. Just think of input redirection as a convenient way to input data from a file instead of entering it on the keyboard. Output redirection is just a convenient way to save data in a file that are normally sent to the host system's display.

A close cousin to input/output redirection is a *pipe*. A pipe provides a link between two programs. The output of the first is presented as input to the second. As with redirection, a pipe affects only the standard output and the standard input. The pipe symbol is " | ".

With redirection and pipes, we are in a position to perform substantial processing on large amounts of data — provided the data are stored in a text file. A text file contains lines of "displayable" characters separated by newline characters. This is in contrast to binary files that contain non-displayable characters representing computer instructions, formatting codes, etc. To demonstrate redirection and pipes, let's examine two more Java programs.

Echo Words

An interesting variation of the CountWords2 programs is shown in Figure 1. The program EchoAlphaWords reads lines from the standard input, tokenizes each line, and echoes to the standard output each token that contains only letters.

```
1 import java.io.*;
2 import java.util.*;
3
4 public class EchoAlphaWords
5 {
6     public static void main(String[] args) throws IOException
7     {
8         // prepare keyboard for input
9         BufferedReader stdin =
10             new BufferedReader(new InputStreamReader(System.in), 1);
11
12         // prepare to extract words from a line
13         StringTokenizer words;
14         String line;
15         String word;
16
17         // process lines until 'null' (no more input)
18         while ((line = stdin.readLine()) != null)
19         {
20             // prepare to tokenize line
21             words = new StringTokenizer(line);
22
23             // process words in line
24             while (words.hasMoreTokens())
25             {
26                 word = words.nextToken();
27                 boolean isAlphaWord = true;
28
29                 // process characters in word
30                 for (int i = 0; i < word.length(); i++)
31                     if (!Character.isLetter(word.charAt(i)))
32                         isAlphaWord = false;
33                 if (isAlphaWord)
34                     System.out.println(word);
35             }
36         }
37     }
38 }
```

Figure 1. EchoAlphaWords.java

EchoAlphaWords is very similar to the CountWords presented earlier. The main difference is that it sends words to the standard output (see line 34, Figure 1) rather than simply counting words (see line 23, **Error! Reference source not found.**). A sample dialogue with EchoAlphaWords follows:

```
PROMPT>java EchoAlphaWords
java is fun
java
is
fun
^z
```

Of course, EchoAlphaWords can be used with input redirection. For example,

```
PROMPT>java EchoAlphaWords < CountWords2.java
```

echoes all words in the `CountWords2.java` file to the standard output, one word per line. The output is shown Figure 2. To conserve space, the output is shown across four columns. Read down the first column, then left to right. Note that the file contains words in comment lines that do not appear in **Error! Reference source not found.**

program	to	demonstrate	loops
within	loops	Same	as
CountWords	except	only	words
containing	letters	of	the
alphabet	are	There	are
three	levels	of	outer
loop	read	lines	of
input	until	no	more
input	middle	loop	process
words	in	a	line
inner	loop	process	characters
in	a	word	A
useful	source	of	input
is	a	disk	file
which	may	be	read
using	input	A	sample
dialog	is	alpha	words
read	Scott	import	import
public	class	public	static
void	throws	IOException	BufferedReader
stdin	new	String	StringTokenizer
int	count	process	lines
until	no	more	input
while	process	words	in
line	words	new	while
String	word	boolean	isAlphaWord
int	i	process	characters
in	word	while	String
c	i	if	isAlphaWord
if	alpha	words	

Figure 2. Alpha words in `CountWords2.java`

Let's develop this idea further. Suppose we want to count the number of *unique* alpha-only words in a file. We're not quite there yet, because `EchoAlphaWords` echoes words in the order they appear, so the output may contain multiple instances of the same word. There is a DOS utility called `sort` that can help. By default, `sort` reads the standard input and sorts lines in ascending order. Since `EchoAlphaWords` outputs one word per line, "piping" the output of `EchoAlphaWords` through `sort` generates a sorted list of words. Here is a simple dialogue without using input redirection:

```
PROMPT>java EchoAlphaWords | sort
programming in java is fun
^z
fun
in
is
java
programming
```

One line of input containing five words was entered. The line was tokenized and the tokens were sent to the standard output, one token per line. However, the output was "caught" by the pipe and sent as input to the `sort` utility. The output of the `sort` utility is the same five tokens, sorted in ascending order.

A sorted list of the words in the file `CountWords2.java` is generated as follows:

```
PROMPT>java EchoAlphaWords < CountWords2.java | sort
```

Lots of words are repeated. For example, the word "while" appears three times in the output. Before we can count unique words we need to remove duplicate words. The program `FilterDuplicateWords` does this task (see Figure 3).

```

1  import java.io.*;
2  import java.util.*;
3
4  public class FilterDuplicateWords
5  {
6      public static void main(String[] args) throws IOException
7      {
8          // prepare keyboard for input
9          BufferedReader stdin =
10             new BufferedReader(new InputStreamReader(System.in), 1);
11
12         // prepare to extract words from a line
13         StringTokenizer words;
14         String line;
15         String word;
16         String oldWord = "";
17
18         // process lines until 'null' (no more input)
19         while ((line = stdin.readLine()) != null)
20         {
21             // prepare to tokenize line
22             words = new StringTokenizer(line);
23
24             // process words in line
25             while (words.hasMoreTokens())
26             {
27                 word = words.nextToken();
28                 boolean isAlphaWord = true;
29                 int i = 0;
30                 if (!oldWord.equals(word))
31                 {
32                     oldWord = word;
33                     System.out.println(word);
34                 }
35             }
36         }
37     }
38 }

```

Figure 3. FilterDuplicateWords.java

This program keeps a temporary copy of each word in the String object oldWord (line 32). As each new word is read, it is sent to the standard output only if it differs from the previous word (lines 30-34).

With FilterDuplicateWords added to our trick bag, we are ready to count the unique words in the file CountWords2.java. The following command line performs this task:

```

PROMPT>java EchoAlphaWords < CountWords2.java | sort | java
FilterDuplicateWords | java CountWords2
48 alpha words read

```

The command overflows to a second line, but it is processed as a single command. Three pipes are used in combining three Java programs with a DOS utility. Of the 119 alpha words in CountWords2.java, 48 are unique.