2 Text I/O

1 Text input: Console class

Java provides the library class Scanner for inputting text from the keyboard. Because this is a little clumsy and a little inflexible, you are provided with a class called Console instead. We will use Console throughout. You may choose to use Scanner if you wish, but be warned that you will find it more difficult. Console includes the methods to read, respectively, an integer, a double, a boolean, and a word (not containing any blanks):

Console.readInt() Console.readDouble() Console.readBoolean()
Console.readToken()

The items to be input may be freely laid out on one or more lines, just as long as the items on each line are separated by one or more spaces.

The following method reads a single character:

Console.readChar()

The character returned could be an end-of-line character (denoted in programs by '\n').

The following method reads a string consisting of all the characters remaining in the current line (the end-of-line-character is discarded):

Console.readString()

Console input is line-buffered. This means that the data entered by the user is passed to the running program only at each press of the return key. For example, if you type three items on a line and then notice a mistake in the second item, say, you can backspace to the error and correct it, and then re-enter the rest of the line. Only when you press the return key is the data irrevocably transmitted to the program.

The following method yields information about the state of the input, without reading anything:

```
Console.endOfFile()
```

It yields True if there is no more input, and otherwise yields False. Under Windows, the end of keyboard input is signalled by typing control-Z on a line by itself.

Example: Counting the number of lines

The following program counts the number of lines the user enters:

```
class CountLines {
        public static void main(String[] args) {
                int numLines = 0;
                while (!Console.endOfFile()) {
                        String s = Console.readString();
                        numLines++;
                }
                System.out.println(numLines + " lines");
        }
}
```

As an alternative way of handling the end of input, both readToken() and readString() return the special value *null* if no input remains. For example, the following is an alternative way to count lines entered at the keyboard:

```
class CountLines {
        public static void main(String[] args) {
                int numLines = 0;
                String s = Console.readString();  // note first readString()!
                while (s!=null) {
                        numLines++;
                        s = Console.readString(); // and readString() again
                }
                System.out.println(numLines + " lines");
        }
}
```

endOfFile() is the more general technique for detecting end of input because only readToken() and readString() return the special value *null* if no input remain (readInt(), readChar() etc *never* return null).

2 Console class: additional details

In class Console, the methods readToken(), readInt(), readBoolean(), and readDouble() are based on the notion of "tokens". A token is a maximal sequence of characters other than white space (white space consists of spaces, tabs, and end-of-line characters). For example, the tokens in the following input (where - represents a press of the space bar and ® a press of the RETURN key)

------23----true-X---Java123---the--rest---®
--3.14®

are "23", "true", "X", "Java123", "the", "rest", and "3.14". readToken() reads and returns the next token from the keyboard (note that the final delimiting white space character is read but discarded). readToken() always returns a string, even it consists entirely of digits, say. readInt() reads the next token, converts it to an integer, and returns that integer value; it is an error if the token does not represent an integer (i.e. if it is not a sequence of digits, optionally preceded by a minus sign). readBoolean() and readDouble() behave analogously.

readChar() reads and returns the next character in the input (which could be a space, tab, or end-of-line character). Executing a sequence of readChar()'s will yield each character in the input, including each end-of-line character. The end-of-line character is always returned as '\n', even in systems which mark the end of line with a carriage return and a linefeed in succession.

readString() reads the remainder of the text on the current line; the delimiting end-of-line character does not appear at the end of the string that is returned (it is discarded). Usually readString() is invoked to get a line of input, but you may also invoke it if you want the execution of your program to be delayed until the user has pressed the return key.

As an example, suppose a program executes the statements

```
i=Console.readInt(); b=Console.readBoolean();
c=Console.readChar(); t=Console.readToken();
s=Console.readString(); d=Console.readDouble();
```

where the user types

------23----true-X---Java123---the--rest---®
--3.14®

Then the variables would acquire values as follows:

i: 23 b: true c:'X' t:"Java123" s:"--the--rest---" d: 3.14

The following method is also provided, although it is not much used

Console.skipLine()

It discards any remaining input supplied on the current line (including the end-of-line character).

3 Redirecting standard input and output

Every execution of a Java program has associated with it a standard input device and a standard output device. By default, the standard input is the keyboard, and the standard output is the screen. The Console class reads from the standard input, and both print() and println() write to the standard output.

It is possible to redirect the standard input and output for each execution of a program. Almost always, the redirection associates the input with a particular text file, and/or the output with another text file. For example, when testing programs it is tedious to have to repeatedly key in the same data each time we run the program. It is easier to type the data into a file, and let the program read from the file rather than the keyboard. Indeed real programs usually process lots of data, prepared in advance and placed in a file. Suppose, for example that we wish to test the program BestStudent above. We prepare a file called, say, students.txt (it doesn't have to have a .txt suffix) containing the desired input data. We can use any text editor (such as NotePad), but not a word processor (such as Word). The file might look like (let's assume here that the input should consist of three lines, rather than 100 as previously)

```
57 Michael Murphy
89 Patrick James McMahon
78 Jenny Smith
```

Then instead of executing the program via java BestStudent, we execute the command

```
java BestStudent < students.txt
```

which will run BestStudent with the standard input redirected to from the keyboard to the file students.txt. As Console reads from the standard input, the program now takes its input from students.txt without any change being needed in the program itself.

You can also cause output to be sent to a file rather than the screen. This is useful if you want subsequently to print the output. The command

```
java BestStudent > output.txt
```

executes BestStudent with the standard output redirected to a file called output.txt. Any file name will do, and the file need not be created in advance. The program will take input from the keyboard, but no output will be displayed on the screen. Instead, file output.txt will be created containing the program's output; it can be inspected after the program has run by using a text editor. The command

```
java BestStudent < students.txt > output.txt
```

will run the program with both input and output being redirected as indicated.

Note on end of input from files: The endOfFile() method determines whether there are any *characters* remaining in the input. Some editors/systems supply an extra end-of-line at the end of the file by default, and so it is best not to insert a carriage return at the end of the final line if you are using endOfFile(). If you do, endOfFile() may see an extra end-of-line (the one supplied by the system) and will report that there is more data when in fact there isn't. Do not type Control-Z or similar at the end of a text file.

4 Formatted output: printf

`System.out.printf` is similar to `System.out.print`, except that it allows us to control the layout of what is printed. The first argument of `printf` specifies the format of the items to be printed. A typical form is

```
System.out.printf("%5d", 87)
```

which causes " 87" to be printed (with 3 leading spaces). The format string "%5d" indicates that one item will be printed (there's just one % symbol), it will be a decimal number (indicated by d), and it will have a "field length" of 5 (i.e. it will occupy 5 positions) with spaces added as necessary on the left. A string that is padded on the left with spaces is said to be "right-justified". The spaces can be added on the right rather than the left by inserting a hyphen as follows:

```
System.out.printf("%-5d", 87)
```

– this will cause "87 " to be printed. A string that is padded on the right with spaces is said to be "left-justified". The fill character can be 0 instead of a space, as in the following:

```
System.out.printf("%05d", 87)
```

which causes "00087" to be printed. To print strings use the symbol s instead of d. For example,

```
System.out.printf("%-8s", "Java")
```

causes "Java " to be printed. Multiple items can be printed in one printf() statement:

```
System.out.printf("%-8s%5d", "Java", 87)
```

which causes "Java   87".

Characters for printing can also be placed in the format string. For example:

```
System.out.printf("Results: %-8s and %5d", "Java", 87)
```

causes "Results: Java and 87" to be printed. This can be used to cause the output to be printed on a line to itself (like println) by including the carriage-return character "\n", as in

```
System.out.printf("%-5d\n", 87)
```

The letter f is used in format strings to indicate a real number. The number of positions allocated to the fraction part is indicated by writing .*n* after the field length where *n* stands for

the number of positions for the fraction. If the fraction requires more than *n* positions, it will be rounded off to *n* decimal places. For example,

```
System.out.printf("%6.2f", 2.7182)
```

causes “ 2.72” to be printed (two leading spaces because the digits and the decimal point occupy 4 positions and the field length is specified as 6).)

In summary, with – representing a single space:

System.out.printf("%5d", 87) ---87
`System.out.printf("%-5d", 87)` `87---`
`System.out.printf("%05d", 87)` `00087`
System.out.printf("%-8s", "Java") Java----
`System.out.printf("%-8s%5d", "Java", 87)` `Java-------87`
`System.out.printf("Results: %-8s and %5d", "Java", 87)`
`Results:-Java-----and----87`
`System.out.printf("%-5d\n", 87)` `87---®`
System.out.printf("%6.2f", 2.7182) --2.72