APROG – Algoritmia e Programação

Emanuel Cunha Silva
ecs@isep.ipp.pt

Bibliography:
Big Java Late Objects
Book by Cay S. Horstmann

Chapter Goals

- To read and write text files

Reading and Writing Text Files

- Text files are very commonly used to store information
  - Both numbers and words can be stored as text
  - They are the most ‘portable’ types of data files
- The `Scanner` class can be used to read text files
  - We have used it to read from the keyboard
  - Reading from a file requires using the `File` class
- The `PrintWriter` class will be used to write text files
  - Using familiar `print`, `println` and `printf` tools

Text File Input

- Create an object of the `File` class
  - Pass it the name of the file to read in quotes

```
File inputFile = new File("input.txt");
```

- Then create an object of the `Scanner` class
  - Pass the constructor the new `File` object

```
Scanner in = new Scanner(inputFile);
```

- Then use `Scanner` methods such as:
  - `next()`
  - `nextLine()`
  - `hasNextLine()`
  - `hasNext()`
  - `nextDouble()`
  - `nextInt()...`

```
while (in.hasNextLine())
{
   String line = in.nextLine();
   // Process line;
}
```

Text File Output

- Create an object of the PrintWriter class
  - Pass it the name of the file to write in quotes

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

    - If output.txt exists, it will be emptied
    - If output.txt does not exist, it will create an empty file
    - PrintWriter is an enhanced version of PrintStream
    - System.out is a PrintStream object!

```
System.out.println(“Hello World!”);
```

- Then use PrintWriter methods such as:
  - print()
  - println()
  - printf()

```
out.println("Hello, World!");
out.printf("Total: %8.2f\n", totalPrice);
```

Closing Files

- You must use the `close` method when file reading and writing is complete
- Closing a `Scanner`

```
while (in.hasNextLine())
{
   String line = in.nextLine();
   // Process line;
}
in.close();
```

Your text may not be saved to the file until you use the `close` method!

- Closing a `PrintWriter`

```
out.println("Hello, World!");
out.printf("Total: %8.2f\n", totalPrice);
out.close();
```

Exceptions Preview

- One additional issue that we need to tackle:
  - If the input or output file for a `Scanner` doesn't exist, a `FileNotFoundException` occurs when the `Scanner` object is constructed.
  - The `PrintWriter` constructor can generate this exception if it cannot open the file for writing.
    - If the name is illegal or the user does not have the authority to create a file in the given location
  - Add two words to any method that uses File I/O

```
public static void main(String[] args) throws
    FileNotFoundException
```

    - Until you learn how to handle exceptions yourself

And an important `import` or two..

- Exception classes are part of the `java.io` package
  - Place the `import` directives at the beginning of the source file that will be using File I/O and exceptions

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

public class LineNumberer
{
   public void openFile() throws FileNotFoundException
   {
      . . .
   }
}
```

Total.java (1)

```
1  import java.io.File;
2  import java.io.FileNotFoundException;
3  import java.io.PrintWriter;
4  import java.util.Scanner;
5
6  /**
7     This program reads a file with numbers, and writes the numbers to another
8     file, lined up in a column and followed by their total.
9  */
10 public class Total
11 {
12    public static void main(String[] args) throws FileNotFoundException
13    {
14       // Prompt for the input and output file names
15
16       Scanner console = new Scanner(System.in);
17       System.out.print("Input file: ");
18       String inputFileName = console.next();
19       System.out.print("Output file: ");
20       String outputFileName = console.next();
21
22       // Construct the Scanner and PrintWriter objects for reading and writing
23
24       File inputFile = new File(inputFileName);
25       Scanner in = new Scanner(inputFile);
26       PrintWriter out = new PrintWriter(outputFileName);
```

More import statements required! Some examples may use `import java.io.*;`

Note the throws clause

Total.java (2)

```
28          // Read the input and write the output
29
30          double total = 0;
31
32          while (in.hasNextDouble())
33          {
34             double value = in.nextDouble();
35             out.printf("%15.2f\n", value);
36             total = total + value;
37          }
38
39          out.printf("Total: %8.2f\n", total);
40
41          in.close();
42          out.close();
43       }
44    }
```

Don’t forget to close the files before your program ends.

Self Check

What happens when you supply the same name for the input and output files to the `Total` program? Try it out if you are not sure.

Answer: When the `PrintWriter` object is created, the output file is emptied. Sadly, that is the same file as the input file. The input file is now empty and the `while` loop exits immediately.

What happens when you supply the name of a nonexistent input file to the `Total` program? Try it out if you are not sure.

Answer: The program throws a `FileNotFoundException` and terminates.

Common Error

- Backslashes in File Names
  - When using a `String` literal for a file name with path information, you need to supply each backslash twice:

```
File inputFile = new File("c:\\homework\\input.dat");
```

  - A single backslash inside a quoted string is the *escape character*, which means the next character is interpreted differently (for example, '`\n`' for a newline character)
  - When a user supplies a filename into a program, the user should not type the backslash twice

Common Error

- Constructing a `Scanner` with a `String`
  - When you construct a `PrintWriter` with a `String`, it writes to a file:

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

  - This does *not* work for a `Scanner` object

```
Scanner in = new Scanner("input.txt"); // Error?
```

  - It does *not* open a file. Instead, it simply reads through the `String` that you passed ("`input.txt`")
  - To read from a file, pass `Scanner` a `File` object:

```
Scanner in = new Scanner(new File ("input.txt") );
```

  - or

```
File myFile = new File("input.txt");
Scanner in = new Scanner(myFile);
```

Special Topic: Reading Web Pages

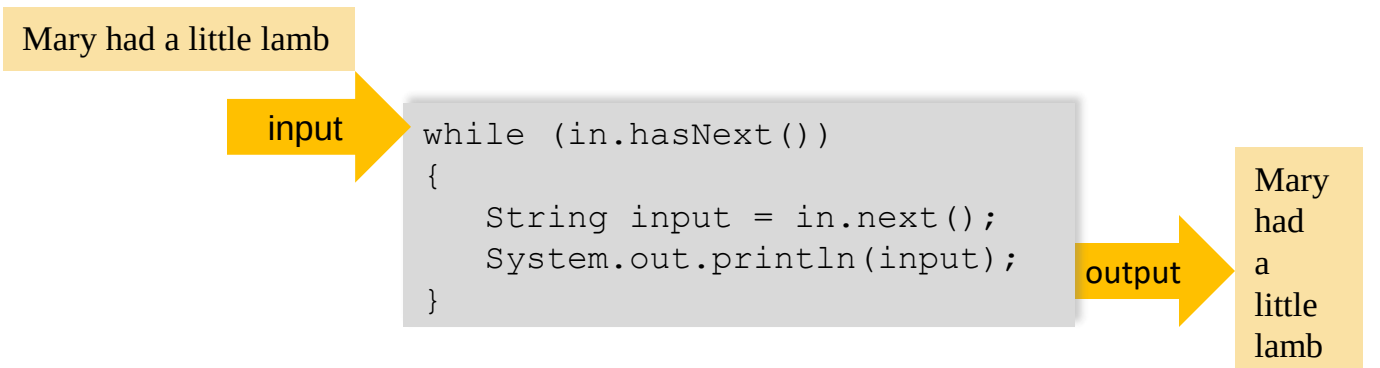
- You can use a `Scanner` to read a web page

```
String address = "http://horstmann.com/index.html";
URL pageLocation = new URL(address);
Scanner in = new Scanner(pageLocation.openStream());
```

- Read the contents of the page with the `Scanner` in the usual way
- The `URL` constructor and the `openStream` method can throw an `IOException`, so tag the `main` method with throws `IOException`
- The `URL` class is contained in the `java.net` package

Text Input and Output

- In the following sections, you will learn how to process text with complex contents, and you will learn how to cope with challenges that often occur with real data.
- Reading Words Example:



Processing Text Input

- There are times when you want to read input by:
  - Each Word
  - Each Line
  - One Number
  - One Character
- Java provides methods of the Scanner and String classes to handle each situation
  - It does take some practice to mix them though!

Processing input is required for almost all types of programs that interact with the user.

Reading Words

- In the examples so far, we have read text one line at a time
- To read each word one at a time in a loop, use:
  - The `Scanner` object's `hasNext()` method to test if there is another word
  - The `Scanner` object's `next()` method to read one word

```
while (in.hasNext())
{
   String input = in.next();
   System.out.println(input);
}
```

Input:

```
Mary had a little lamb
```

Output:

```
Mary
had
a
little
lamb
```

White Space

- The Scanner's `next()` method has to decide where a word starts and ends.

- It uses simple rules:
  - It consumes all white space before the first character
  - It then reads characters until the first white space character is found or the end of the input is reached

White Space

- What is whitespace?
  - Characters used to separate:
    - Words
    - Lines

Common White Space

‘ ‘	Space
\n	NewLine
\r	Carriage Return
\t	Tab
\f	Form Feed

“Mary had a little lamb,\n
her fleece was white as\tsnow”

The `useDelimiter` Method

- The `Scanner` class has a method to change the default set of delimiters used to separate words.

  - The `useDelimiter` method takes a `String` that lists all of the characters you want to use as delimiters:

```
Scanner in = new Scanner(. . .);
in.useDelimiter("[^A-Za-z]+");
```

  - You can also pass a `String` in *regular expression* format inside the `String` parameter as in the example above.

  - `[^A-Za-z]+` says that all characters that `^` not either `A-Z` uppercase letters A through Z or `a-z` lowercase a through `z` are delimiters.

  - Search the Internet to learn more about regular expressions

Reading Characters

- There are no `hasNextChar()` or `nextChar()` methods of the `Scanner` class
  - Instead, you can set the `Scanner` to use an 'empty' delimiter (`""`)

```
Scanner in = new Scanner(. . .);
in.useDelimiter("");

while (in.hasNext())
{
  char ch = in.next().charAt(0);
  // Process each character
}
```

  - `next` returns a one character `String`
  - Use `charAt(0)` to extract the character from the `String` at index `0` to a `char` variable

Classifying Characters

- The `Character` class provides several useful methods to classify a character:
  - Pass them a `char` and they return a boolean

```
if ( Character.isDigit(ch) ) …
```

Table 1 Character Testing Methods

Method	Examples of Accepted Characters
`isDigit`	0, 1, 2
`isLetter`	A, B, C, a, b, c
`isUpperCase`	A, B, C
`isLowerCase`	a, b, c
`isWhiteSpace`	space, newline, tab

Reading Lines

- Some text files are used as simple databases
  - Each line has a set of related pieces of information
  - This example is complicated by:
    - Some countries use two words
      - "United States"

China 1330044605
India 1147995898
United States 303824646

  - It would be better to read the entire line and process it using powerful `String` class methods

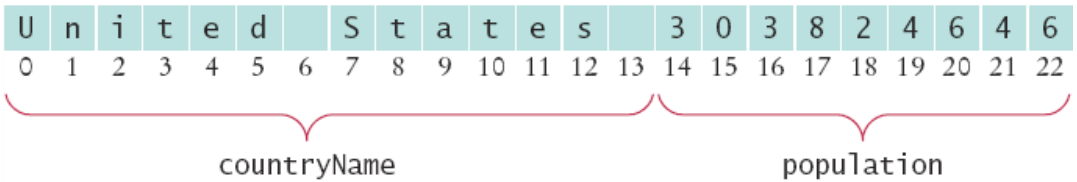
```
while (in.hasNextLine())
{
   String line = in.nextLine();
   // Process each line
}
```

- `nextLine()` reads one line and consumes the ending '`\n`'

U	n	i	t	e	d		S	t	a	t	e	s		3	0	3	8	2	4	6	4	6
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Breaking Up Each Line

- Now we need to break up the line into two parts
  - Everything before the first digit is part of the country



  - Get the index of the first digit with `Character.isdigit`

```
int i = 0;
while (!Character.isDigit(line.charAt(i))) { i++; }
```

  - Use `String` methods to extract the two parts

```
String countryName = line.substring(0, i);
String population = line.substring(i);
// remove the trailing space in countryName
countryName = countryName.trim();
```

United States

303824646

`trim` removes white space at the beginning and the end.

Or Use Scanner Methods

- Instead of `String` methods, you can sometimes use `Scanner` methods to do the same tasks
  - Read the line into a `String` variable
    - Pass the `String` variable to a new `Scanner` object
  - Use `Scanner hasNextInt` to find the numbers
    - If not numbers, use `next` and concatenate words

United States 303824646

```
Scanner lineScanner = new Scanner(line);

String countryName = lineScanner.next();
while (!lineScanner.hasNextInt())
{
  countryName = countryName + " " + lineScanner.next();
}
```

Remember the `next` method consumes white space.

Converting Strings to Numbers

- Strings can contain *digits*, not *numbers*
  - They must be converted to numeric types
  - 'Wrapper' classes provide a `parseInt` method


```
String pop = "303824646";
int populationValue = Integer.parseInt(pop);
```

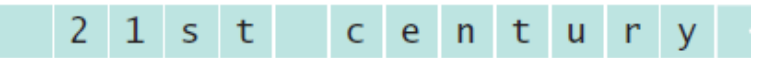

```
String priceString = "3.95";
double price = Double.parseDouble(priceString);
```

Caution: The argument must be a string containing only digits without any additional characters. Not even spaces are allowed! So… Use the `trim` method before parsing!

```
int populationValue = Integer.parseInt(pop.trim());
```

Safely Reading Numbers

- `Scanner nextInt` and `nextDouble` can get confused
  - If the number is not properly formatted, an “Input Mismatch Exception” occurs



  - Use the `hasNextInt` and `hasNextDouble` methods to test your input first

```
if (in.hasNextInt())
{
  int value = in.nextInt();  // safe
}
```

- They will return `true` if digits are present
  - If true, `nextInt` and `nextDouble` will return a value
  - If not true, they would ‘throw’ an ‘Input Mismatch Exception’

Reading Other Number Types

- The `Scanner` class has methods to test and read almost all of the primitive types

Data Type	Test Method	Read Method
`byte`	`hasNextByte`	`nextByte`
`short`	`hasNextShort`	`nextShort`
`int`	`hasNextInt`	`nextInt`
`long`	`hasNextLong`	`nextLong`
`float`	`hasNextFloat`	`nextFloat`
`double`	`hasNextDouble`	`nextDouble`
`boolean`	`hasNextBoolean`	`nextBoolean`

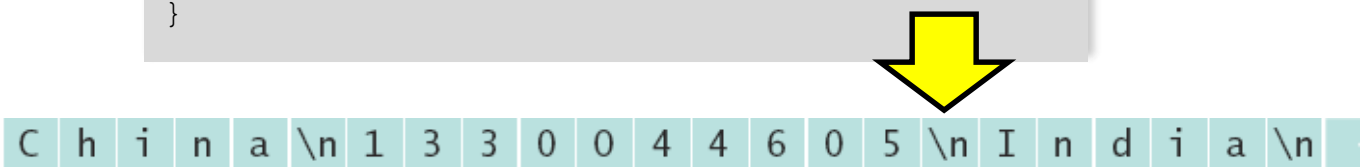
- What is missing?
  - Right, no `char` methods!

Mixing Number, Word and Line Input

- `nextDouble` (and `nextInt`…) do not consume white space following a number
  - This can be an issue when calling `nextLine` after reading a number
  - There is a 'newline' at the end of each line
  - After reading 1330044605 with `nextInt`
    - `nextLine` will read until the `\n` (an empty `String`)

China
1330044605
India

```
while (in.hasNextInt())
{
   String countryName = in.nextLine();
   int population = in.nextInt();
   in.nextLine();   // Consume the newline
}
```

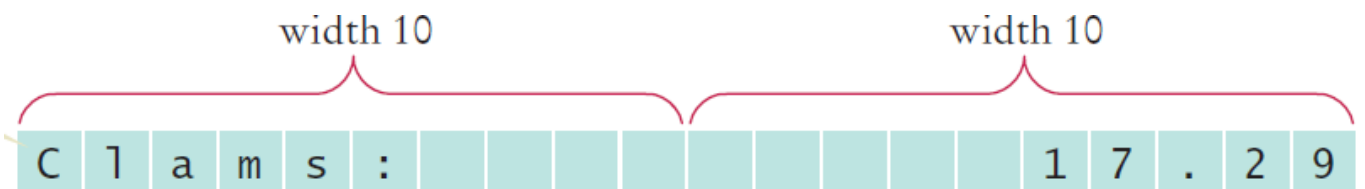


Formatting Output

- Advanced `System.out.printf`
  - Can align strings and numbers
  - Can set the field width for each
  - Can left align (default is right)
- Two format specifiers example:

```
System.out.printf("%-10s%10.2f", items[i] + ":", prices[i]);
```

  - `%-10s` : Left justified String, width 10
  - `%10.2f` : Right justified, 2 decimal places, width 10



`printf` Format Specifier

- A format specifier has the following structure:
  - The first character is a `%`
  - Next, there are optional “flags” that modify the format, such as `-` to indicate left alignment. See Table 2 for the most common format flags
  - Next is the field width, the total number of characters in the field (including the spaces used for padding), followed by an optional precision for floating-point numbers
- The format specifier ends with the format type, such as `f` for floating-point values or `s` for strings. See Table 3 for the most important formats

`printf` Format Flags

Table 2 Format Flags

Flag	Meaning	Example
-	Left alignment	`1.23` followed by spaces
0	Show leading zeroes	`001.23`
+	Show a plus sign for positive numbers	`+1.23`
(	Enclose negative numbers in parentheses	`(1.23)`
,	Show decimal separators	`12,300`
^	Convert letters to uppercase	`1.23E+1`

`printf` Format Types

Table 3 Format Types		
Code	Type	Example
`d`	Decimal integer	`123`
`f`	Fixed floating-point	`12.30`
`e`	Exponential floating-point	`1.23e+1`
`g`	General floating-point (exponential notation is used for very large or very small values)	`12.3`
`s`	String	`Tax:`

Self Check

Suppose the input contains the characters `Hello, World!`. What are the values of `word` and `input` after this code fragment?

```
String word = in.next();
String input = in.nextLine();
```

Answer: `word` is "`Hello,`" and `input` is "`World!`"

Suppose the input contains the characters `995.0 Fred`. What are the values of `number` and `input` after this code fragment?

```
int number = 0;
if (in.hasNextInt()) { number = in.nextInt(); }
String input = in.next();
```

Answer: Because `995.0` is not an integer, the call `in.hasNextInt()` returns false, and the call `in.nextInt()` is skipped. The value of `number` stays 0, and `input` is set to the string "`995.0`".

Suppose the input contains the characters `6E6 $6,995.00`. What are the values of `x1` and `x2` after this code fragment?

```
double x1 = in.nextDouble();
double x2 = in.nextDouble();
```

Answer: `x1` is set to 6000000. Because a dollar sign is not considered a part of a floating-point number in Java, the second call to `nextDouble` causes an input mismatch exception and `x2` is not set.

Steps to Processing Text Files

- Read two country data files, `worldpop.txt` and `worldarea.txt`.

- Write a file `world_pop_density.txt` that contains country names and population densities with the country names aligned left and the numbers aligned right.

```
Afghanistan                 50.56
Akrotiri                    127.64
Albania                     125.91
Algeria                     14.18
American Samoa              288.92
. . .
```

Steps to Processing Text Files

1) Understand the Processing Task

-- Process ‘on the go’ or store data and then process?

2) Determine input and output files

3) Choose how you will get file names

4) Choose line, word or character based input processing

-- If all data is on one line, normally use line input

5) With line-oriented input, extract required data

-- Examine the line and plan for whitespace, delimiters…

6) Use methods to factor out common tasks

Processing Text Files: Pseudocode

- Step 1: Understand the Task

```
While there are more lines to be read
        Read a line from each file
        Extract the country name
        population = number following the country name in
          the line from the first file
        area = number following the country name in the line
          from the second file
        If area != 0
                density = population / area
        Print country name and density
```

Afghanistan	50.56
Akrotiri	127.64
Albania	125.91
Algeria	14.18
American Samoa	288.92
. . .	

Closing Resources

- Special treatment must be given to resources that must be closed
- Consider the code below:

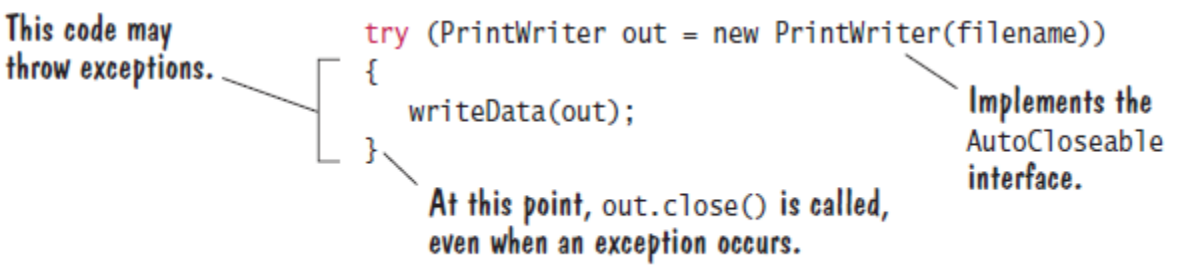
```
PrintWriter out = new PrintWriter(filename);
writeData(out);
out.close();  // May never get here
```

- The `try-with-resources` statement calls the `close` method of the named resource automatically when the try block is completed
- Named resources must implement the `AutoCloseable` interface

try-with-resources

Syntax

```
try (Type1 variable1 = expression1; Type2 variable2 = expression2; . . .)
{
   . . .
}
```



Self Check

Why is an `ArrayIndexOutOfBoundsException` not a checked exception?

Answer: Because programmers should simply check that their array index values are valid instead of trying to handle an `ArrayIndexOutOfBoundsException`.

What is wrong with the following code, and how can you fix it?

```
public static void writeAll(String[] lines, String filename)
{
   PrintWriter out = new PrintWriter(filename);
   for (String line : lines)
   {
      out.println(line.toUpperCase());
   }
   out.close();
}
```

Answer: There are two mistakes. The `PrintWriter` constructor can throw a `FileNotFoundException`. You should supply a `throws` clause. And if one of the array elements is `null`, a `NullPointerException` is thrown. In that case, the `out.close()` statement is never executed. You should use a try-with-resources statement.