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 before 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

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
   {
      . . .
   }
}
```

Example: 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

Example: 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.

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:

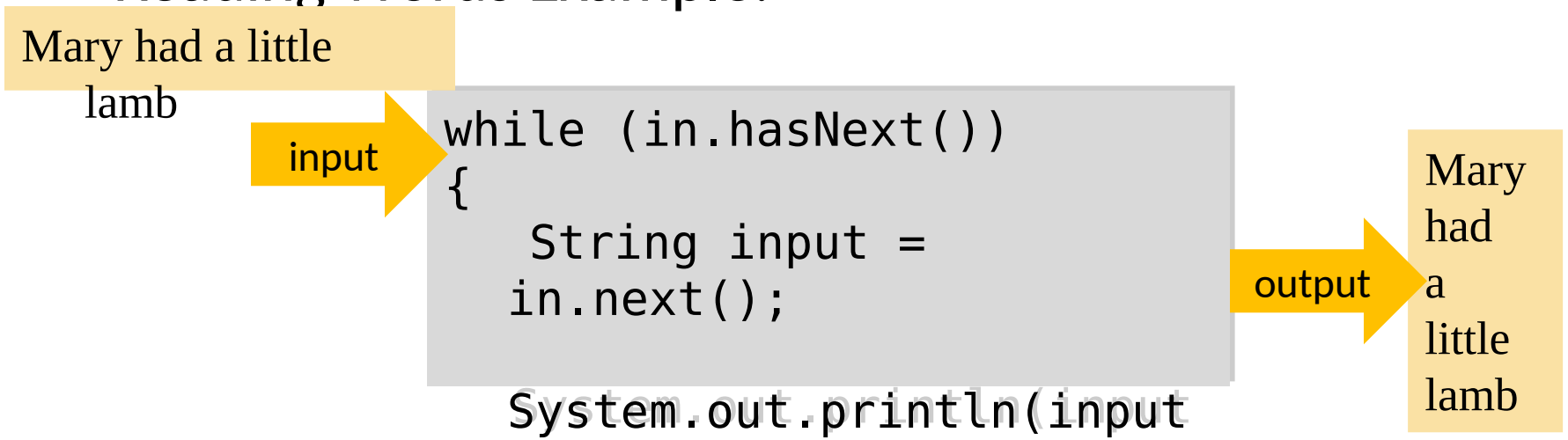
```
Scanner in = new Scanner(new File
   ("input.txt"));
```

  - or

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

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

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

- Java provides methods of the `Scanner` and `String` classes to handle each situation
  - It does take some practice to mix them though!

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

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(“,”);
```

The `useDelimiter` Method

```
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 but it is not required for this course.

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

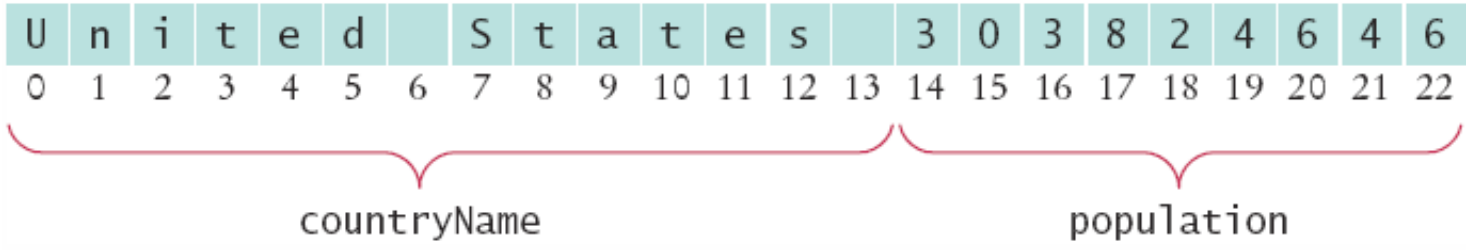
China 1330044605
India 1147995898
United States 303824646

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

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

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




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

Breaking Up Each Line

- 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

'3'	'0'	'3'	'8'	'2'	'4'	'6'	'4'	'6'

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

'3'	'.'	'9'	'5'

```
String priceString = "3.95";
int price = Double.parseInt(priceString);
```

Converting Strings to Numbers

- 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 | 2 | 1 | s | t | | c | e | n | t | u | r | y | Exception" occurs
  - Use the `hasNextInt` and `hasNextDouble` methods to test your in

```
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
}
```

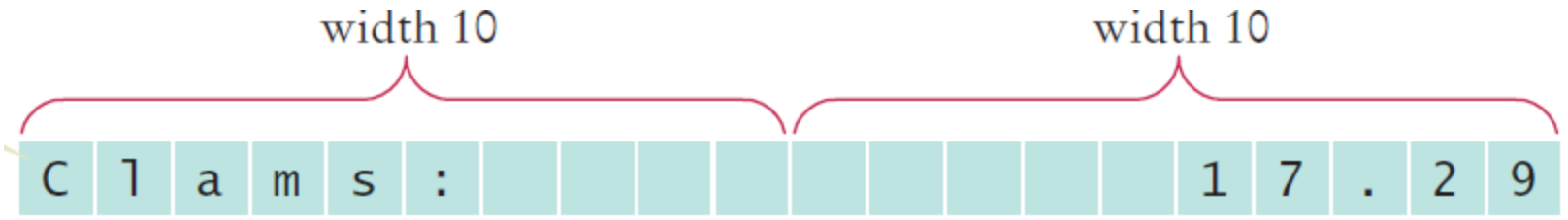
Formatting Output

- Advanced `System.out.print`
  - Can align strings and numbers
  - Can set the field width for each
  - Can left align (default is right)

```
Cookies:        3.20
Linguine:       2.95
Clams:         17.29
```

```
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:`