Object Oriented Programming

#6 I/O and Exception Handling

Hilmy A. Tawakal & Agung Prayoga

October 17, 2017

Table of contents

Using Scanner Class

In Java, the most convenient mechanism for reading text is to use the **Scanner** class..

- Construct a File object with the name of the input file.

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

- Use the File object to construct a Scanner object.

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

- Use loop to process input.

```
while (in.hasNextDouble())
{
  double value = in.nextDouble();
  Process value.
}
```

Writing output

- To write output to a file, you construct a **PrintWriter** object with the desired file name

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

- You can use the familiar *print* , *println* , and *printf* methods with any **PrintWriter** object

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

Close Scanner or Print Writer

- When you are done processing a file, be sure to *close* the **Scanner** or **PrintWriter**

```
in.close();
out.close();
```

- If your program exits without closing the **PrintWriter** , some of the output may not be written to the disk file.

Example

- For example, the input file has the contents:

```
32 54 67.5 29 35 80
115 44.5 100 65
```

- The output file is:

```
    32.00
    54.00
    67.50
    29.00
    35.00
    80.00
   115.00
    44.50
   100.00
    65.00
Total: 622.00
```

Total.java

```
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    {
```

Total.java

```
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);
27
```

Total.java

```
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    }
```

Questions

1. 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.
2. What happens when you supply the name of a nonexistent input file to the Total program? Try it out if you are not sure.
3. How do you modify the program so that it shows the average, not the total, of the inputs?
4. How can you modify the Total program so that it writes the values in two columns, like this:

```
 32.00  54.00
 67.50  29.00
 35.00  80.00
115.00  44.50
100.00  65.00
Total:  622.00
```

Reading Words

- The *next* method of the **Scanner** class reads the next string.

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

- If the user provides the input:

```
Mary had a little lamb
```

- this loop prints each word on a separate line:

```
Mary
had
a
little
lamb
```

White Space

- However, the words can contain punctuation marks and other symbols.
- The *next* method returns any sequence of characters that is not white space.
- **White space** includes spaces, tab characters, and the newline characters that separate lines.
- For example, the following strings are considered "words" by the *next* method:

```
snow.
1729
C++
```

- Note the period after *snow*, it is considered a part of the word because it is not white space.

Reading Characters

- Sometimes, you want to read just the words and discard anything that isnt́ a letter.
- You achieve this task by calling the *useDelimiter* method on your Scanner object:

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

- Sometimes, you want to read a file one character at a time.

```
Scanner in = new Scanner(. . .);
in.useDelimiter("");
while (in.hasNext())
{
  char ch = in.next().charAt(0);
  Process ch.
}
```

Classifying Characters

- When you read a character, or when you analyze the characters in a word or line,you often want to know what kind of character it is.
- The Character class declares several useful methods for this purpose.
- For example, the call

```
Character.isDigit(ch)
```

- returns **true** if ch is a digit ('0' . . . '9' or a digit in another writing system), **false** otherwise.

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

- When each line of a file is a data record, it is often best to read entire lines with the *nextLine* method

```
String line = in.nextLine();
```

- The *hasNextLine* method returns *true* if there is at least one more line in the input, *false* when all lines have been read.
- To ensure that there is another line to process, call the *hasNextLine* method before calling *nextLine*.

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

Scanning a String

- Here is a typical example of processing lines in a file.

```
China 1330044605
India 1147995898
United States 303824646
. . .
```

- You can use a **Scanner** object to read the characters from a string

```
String line = in.nextLine();
Scanner lineScanner = new Scanner(line);
String countryName = lineScanner.next();
while (!lineScanner.hasNextInt())
{
  countryName = countryName + " " + lineScanner.next();
}
int populationValue = lineScanner.nextInt();
```

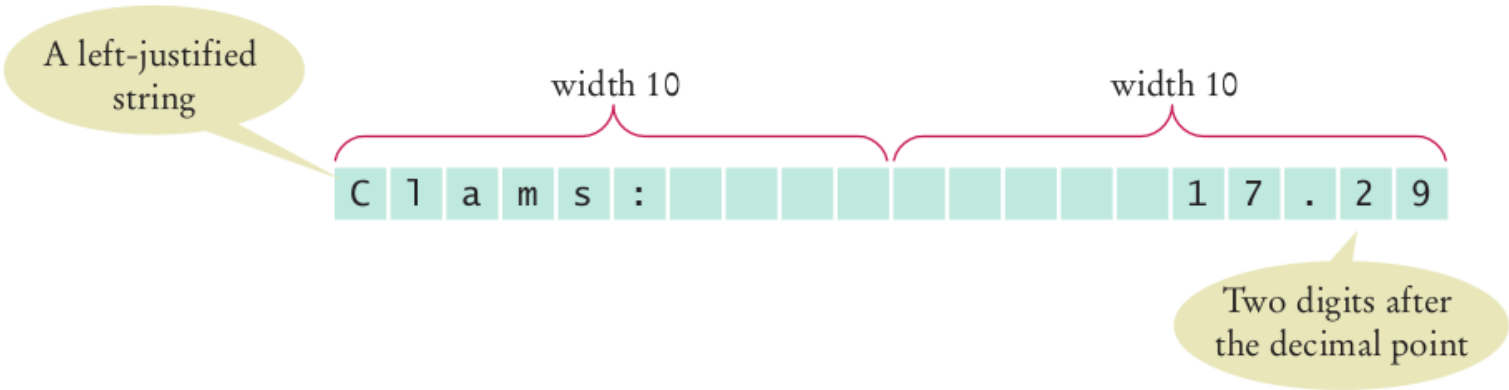
Formatting output

- Suppose you need to print a table of items and prices, each stored in an array, such as

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

- To specify left alignment, you add a hyphen (-) before the field width:

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



Formatting specifier

- A construct such as *%-10s* or *%10.2f* is called a **format specifier**
- It describes how a value should be formatted
- 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
  - 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.

Formatting flag

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

Formatting type

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:

Exception Handling

- There are two aspects to dealing with program errors: **detection** and **handling**.
- For example, the *Scanner* constructor can detect an attempt to read from a non-existent file.
- However, it cannot handle that error.
- A satisfactory way of handling the error might be to terminate the program, or to ask the user for another file name.
- The *Scanner* class cannot choose between these alternatives.
- It needs to report the error to another part of the program.
- In Java, **exception handling** provides a flexible mechanism for passing control from the point of error detection to a handler that can deal with the error.

Throwing Exceptions

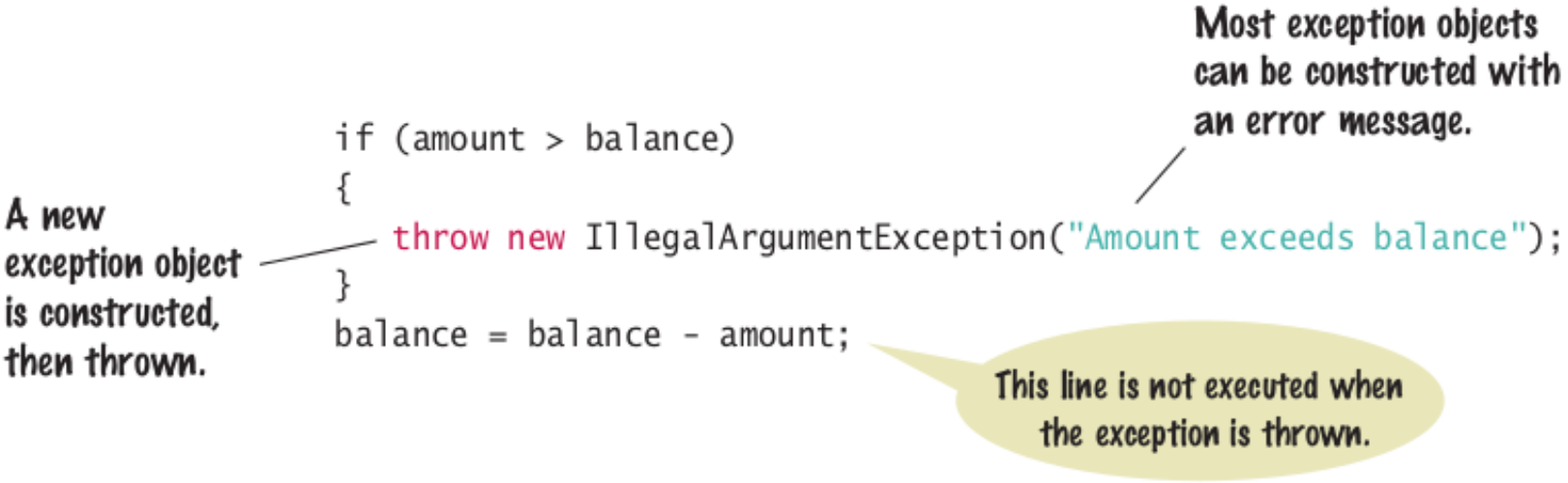
- When you detect an error condition, your job is really easy.
- You just throw an appropriate exception object, and you are done.
- For example, suppose someone tries to withdraw too much money from a bank account.

```
if (amount > balance)
{
    throw new IllegalArgumentException("Amount exceeds
        balance");
}
```

- When you throw an exception, execution does not continue with the next statement but with an exception handler.

Throwing Exceptions

Syntax `throw` *exceptionObject*;



Catching Exceptions

- Every exception should be handled somewhere in your program.
- If an exception has no handler, an error message is printed, and your program terminates.
- You handle exceptions with the *try* / *catch* statement.
- Place the statement into a location of your program that knows how to handle a particular exception.
- The *try* block contains one or more statements that may cause an exception of the kind that you are willing to handle.
- Each *catch* clause contains the handler for an exception type.

Example

```
try
{
    String filename = . . .;
    Scanner in = new Scanner(new File(filename));
    String input = in.next();
    int value = Integer.parseInt(input);
    . . .
}
catch (IOException exception)
{
    exception.printStackTrace();
}
catch (NumberFormatException exception)
{
    System.out.println(exception.getMessage());
}
```

Catching Exceptions

Syntax

```
try
{
   statement
   statement
   . . .
}
catch (ExceptionClass exceptionObject)
{
   statement
   statement
   . . .
}
```

```
try
{
   Scanner in = new Scanner(new File("input.txt"));
   String input = in.next();
   process(input);
}
catch (IOException exception)
{
   System.out.println("Could not open input file");
}
catch (Exception except)
{
   System.out.println(except.getMessage());
}
```

This constructor can throw a FileNotFoundException.

When an IOException is thrown, execution resumes here.

This is the exception that was thrown.

Additional catch clauses can appear here. Place more specific exceptions before more general ones.

A FileNotFoundException is a special case of an IOException.

Throw Clause

Syntax *modifiers returnType methodName(parameterType parameterName, . . .)*
`throws` *ExceptionClass, ExceptionClass, . . .*

```
public void readData(String filename)
      throws FileNotFoundException, NumberFormatException
```

You must specify all checked exceptions that this method may throw.

You may also list unchecked exceptions.

Finally clause

Syntax

```
try
{
   statement
   statement
   . . .
}
finally
{
   statement
   statement
   . . .
}
```

This variable must be declared outside the `try` block so that the `finally` clause can access it.

This code may throw exceptions.

This code is always executed, even if an exception occurs.

```
PrintWriter out = new PrintWriter(filename);
try
{
   writeData(out);
}
finally
{
   out.close();
}
```