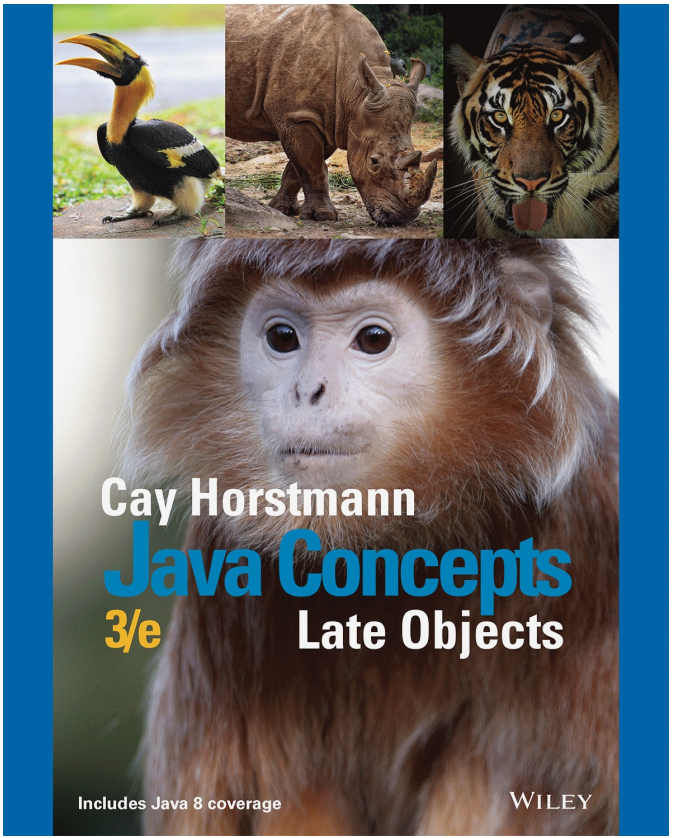
Chapter 7 - Input/Output and Exception Handling



Chapter Goals

- To read and write text files
- To process command line arguments
- To throw and catch exceptions
- To implement programs that propagate checked exceptions

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
  - 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 7.1

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.

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.

Self Check 7.2

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

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

Self Check 7.3

Suppose you wanted to add the total to an existing file instead of writing a new file. Self Check 1 indicates that you cannot simply do this by specifying the same file for input and output. How can you achieve this task? Provide the pseudocode for the solution.

Answer:

```
Open a scanner for the file.
For each number in the scanner
   Add the number to an array.
Close the scanner.
Set total to 0.
Open a print writer for the file.
For each number in the array
   Write the number to the print writer.
   Add the number to total.
Write total to the print writer.
Close the print writer.
```

Self Check 7.4

How do you modify the program so that it shows the average, not the total, of the inputs?

Answer: Add a variable count that is incremented whenever a number is read. In the end, print the average, not the total, as

```
out.printf("Average: %8.2f%n", total / count);
```

Because the string "`Average`" is three characters longer than "`Total`", change the other output to `out.printf("%18.2f%n", value)`.

Self Check 7.5

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
```

Answer: Add a variable `count` that is incremented whenever a number is read. Only write a new line when it is even.

```
count++;
out.printf("%8.2f", value);
if (count % 2 == 0) { out.println(); }
```

At the end of the loop, write a new line if count is odd, then write the total:

```
if (count % 2 == 1) { out.println(); }
out.printf("Total: %10.2f%n", total);
```

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

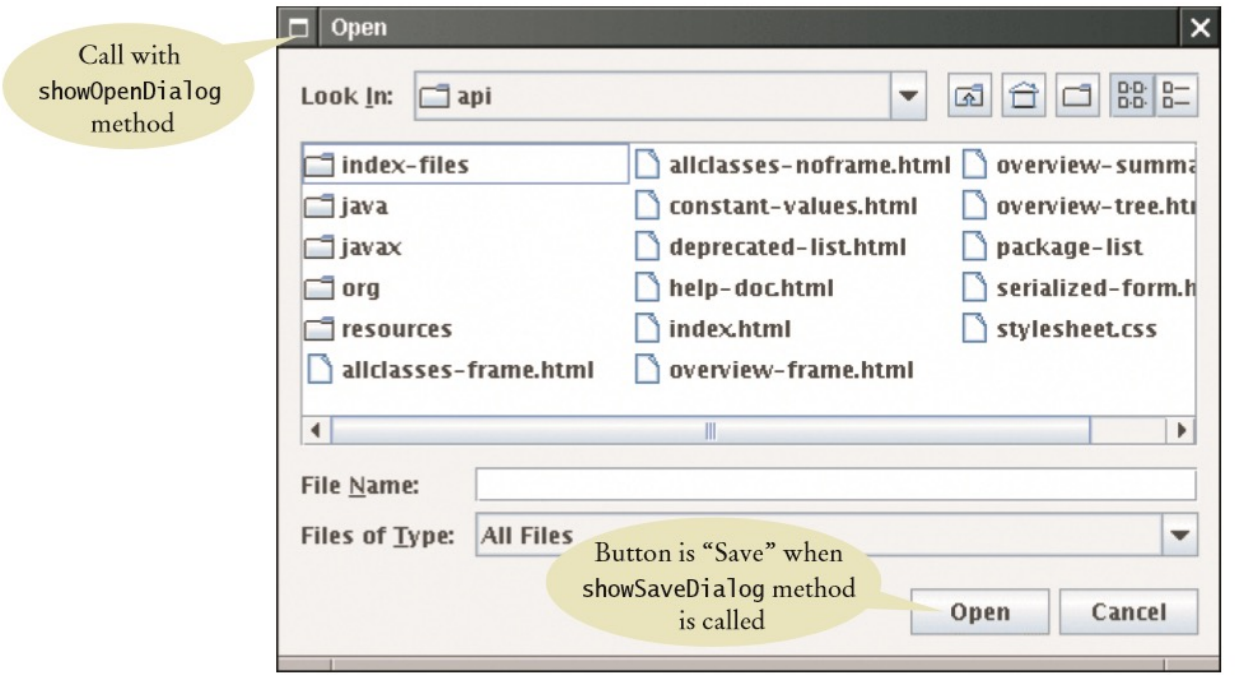
Special Topic: File Dialog Boxes

- You may want to use a file dialog box whenever users of your program need to pick a file.
- The `JFileChooser` class implements a file dialog box for the `Swing` user-interface toolkit.

```
JFileChooser chooser = new JFileChooser();
Scanner in = null;
if (chooser.showOpenDialog(null) == JFileChooser.APPROVE_OPTION)
{
   File selectedFile = chooser.getSelectedFile();
   in = new Scanner(selectedFile);
   . . .
}
```

The `showOpenDialog` and `showSaveDialog` methods return either `JFileChooser.APPROVE_OPTION`, if the user has chosen a file, or `JFi leChooser.CANCEL_OPTION`, if the user canceled the selection.

Special Topic: File Dialog Boxes



A `JFileChooser` *Dialog Box*

Special Topic: Reading and Writing Binary Data

- `Scanner` and `PrintWriter` classes read and write text files which are comprised of sequences of characters.
- Other files, such as images, are not made up of characters but of bytes.
- A byte is a fundamental storage unit in a computer—a number consisting of eight binary digits.
- A byte can represent unsigned integers between 0 and 255 or signed integers between –128 and 127
- The Java library has a different set of classes, called streams, for working with binary files

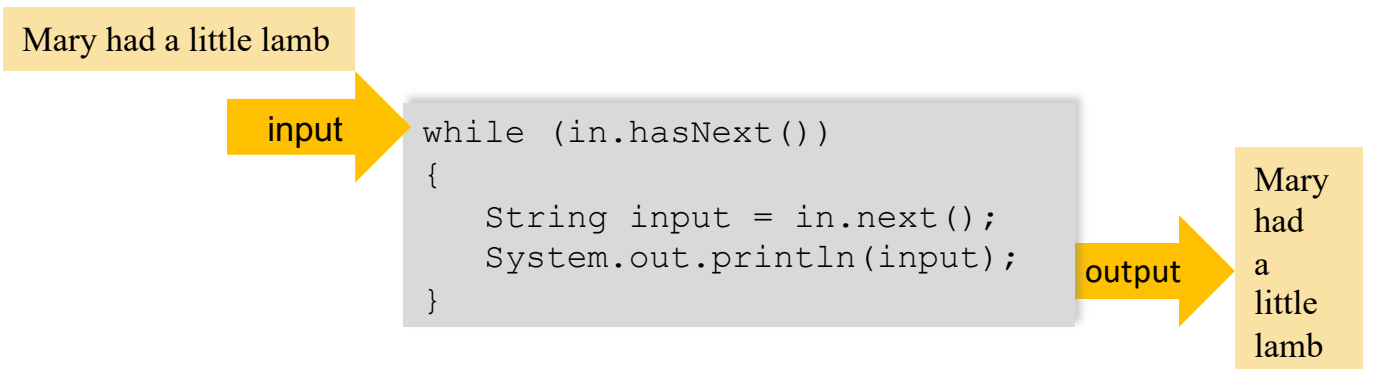
Special Topic: Reading and Writing Binary Data

- We cover a simple example of copying binary data from a web site to a file
- Use an `InputStream` to read binary data:

```
URL imageLocation =
      new URL("http://horstmann.com/java4everyone/duke.gif");
InputStream in = imageLocation.openStream();
FileOutputStream out = new FileOutputStream("duke.gif");
boolean done = false;
while (!done)
{
   int input = in.read(); // -1 or a byte between 0 and 255
   if (input == -1) { done = true; }
   else { out.write(input); }
}
```

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

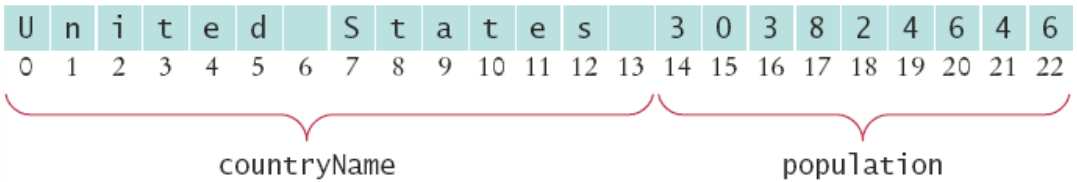
```
while (in.hasNextLine())
{
   String line = in.nextLine();
   // Process each line
}
```

- `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);
```

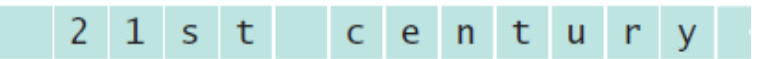

```
String priceString = "3.95";
int price = Double.parseDouble(priceString);
```

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`

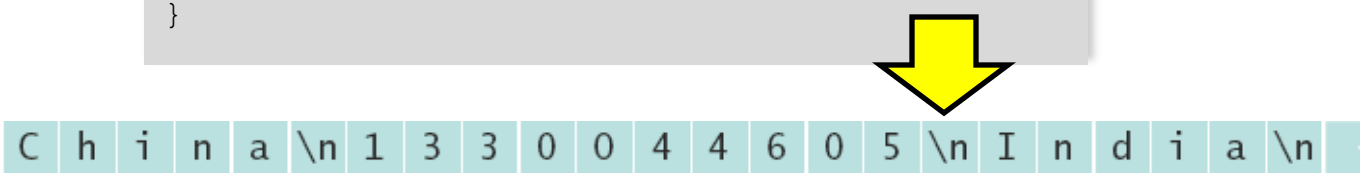
- What is missing?
  - Right, no `char` methods!

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
}
```

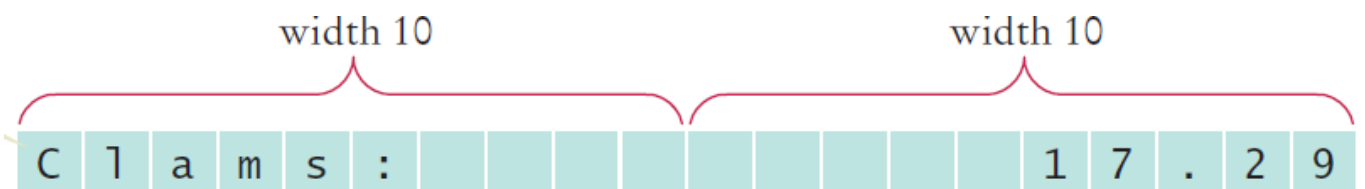


Formatting Output

- 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 7.6

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!`"

Self Check 7.7

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`".

Self Check 7.8

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.

Self Check 7.9

Your input file contains a sequence of numbers, but sometimes a value is not available and is marked as `N/A`. How can you read the numbers and skip over the markers?

Answer: Read them as strings, and convert those strings to numbers that are not equal to `N/A`:

```
String input = in.next();
if (!input.equals("N/A"))
{
   double value = Double.parseDouble(input);
   Process value.
}
```

Self Check 7.10

How can you remove spaces from the country name in Section 7.2.4 without using the `trim` method?

Answer: Locate the last character of the country name:

```
int j = i - 1;
while (!Character.isWhiteSpace(line.charAt(j))) {
   j--;
}
```

Then extract the country name:

```
String countryName = line.substring(0, j + 1);
```

Special Topic

- Regular Expressions
  - Regular expressions describe character patterns, for example, `[0-9]+` describes a sequence of one or more digits between 0 and 9
  - Grep is a program that displays a line if it matches a regular expression
  - `grep [0-9]+ Homework.java` would list all lines in the file that contain a sequence of digits
  - `grep [^A-Za-z][0-9]+ Homework.java` lists lines with sequences of digits that do not immediately follow letters
  - Some Java methods accept regular expressions as parameters
  - `String[] tokens = line.split("\\s+")` splits input along whitespace

Special Topic

- Reading an Entire File
  - You can read an entire file into a list of lines, or into a single string
  - Use the `Files` and `Paths` classes

```
String filename = . . .;
List<String> lines = Files.readAllLines(Paths.get(filename));
String content =
   new String(Files.readAllBytes(Paths.get(filename)));
```

Command Line Arguments

- Text based programs can be 'parameterized' by using command line arguments
  - Filename and options are often typed after the program name at a command prompt:

```
>java ProgramClass -v input.dat
```

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

  - Java provides access to them as an array of `Strings` parameter to the main method named `args`

```
args[0]: "-v"
args[1]: "input.dat"
```

  - The `args.length` variable holds the number of `args`
  - Options (switches) traditionally begin with a dash '–'

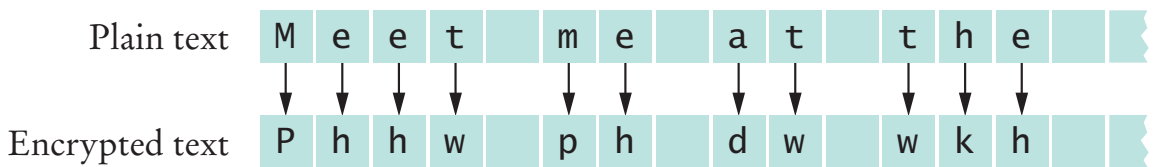
Caesar Cipher Example

- Write a command line program that uses character replacement (Caesar cipher) to:
  - Encrypt a file provided input and output file names

```
>java CaesarCipher input.txt encrypt.txt
```

  - Decrypt a file as an option

```
>java CaesarCipher -d encrypt.txt output.txt
```



CaesarCipher.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 encrypts a file using the Caesar cipher.
8  */
9  public class CaesarCipher
10 {
11    public static void main(String[] args) throws FileNotFoundException
12    {
13       final int DEFAULT_KEY = 3;
14       int key = DEFAULT_KEY;
15       String inFile = "";
16       String outFile = "";
17       int files = 0; // Number of command line arguments that are files
18
```

This method uses file I/O and can throw this exception.

CaesarCipher.java (2)

```
19      for (int i = 0; i < args.length; i++)
20      {
21         String arg = args[i];
22         if (arg.charAt(0) == '-')
23         {
24            // It is a command line option
25
26            char option = arg.charAt(1);
27            if (option == 'd') { key = -key; }
28            else { usage(); return; }
29         }
30         else
31         {
32            // It is a file name
33
34            files++;
35            if (files == 1) { inFile = arg; }
36            else if (files == 2) { outFile = arg; }
37         }
38      }
39      if (files != 2) { usage(); return; }
```

If the switch is present, it is the first argument

Call the usage method to print helpful instructions

CaesarCipher.java (3)

```
41          Scanner in = new Scanner(new File(inFile));
42          in.useDelimiter(""); // Process individual characters
43          PrintWriter out = new PrintWriter(outFile);
44
45          while (in.hasNext())
46          {
47             char from = in.next().charAt(0);
48             char to = encrypt(from, key);
49             out.print(to);
50          }
51          in.close();
52          out.close();
53       }
```

Process the input file one character at a time

Don't forget the close the files!

```
74
75       /**
76          Prints a message describing proper usage.
77       */
78       public static void usage()
79       {
80          System.out.println("Usage: java CaesarCipher [-d] infile outfile");
81       }
82    }
```

Example of a 'usage' method

Self Check 7.11

If the program is invoked with `java CaesarCipher -d file1.txt`, what are the elements of `args`?

Answer: `args[0]` is "`-d`" and `args[1]` is "`file1.txt`"

Self Check 7.12

Trace the program when it is invoked as in Self Check 11.

Answer:

key	inFile	outFile	i	arg
~~3~~	~~null~~	null	~~0~~	~~-d~~
-3	file1.txt		~~1~~	file1.txt
			2	

Then the program prints a message
Usage: `java CaesarCipher [-d] infile outfile`

Self Check 7.13

Will the program run correctly if the program is invoked with `java CaesarCipher file1.txt file2.txt -d`? If so, why? If not, why not?

Answer: The program will run correctly. The loop that parses the options does not depend on the positions in which the options appear.

Self Check 7.14

Encrypt CAESAR using the Caesar cipher.

Answer: FDHVDU

Self Check 7.15

How can you modify the program so that the user can specify an encryption key other than 3 with a `-k` option, for example

```
java CaesarCipher -k15 input.txt output.txt
```

Answer: Add the lines

```
else if (option == 'k')
{
   key = Integer.parseInt(
   args[i].substring(2));
}
```

after line 27 and update the usage information.

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
. . .	

Exception Handling

- There are two aspects to dealing with run-time program errors:
  - Detecting Errors
  - This is the easy part. You can 'throw' an exception

Use the throw statement to signal an exception

```
if (amount > balance)
{
  // Now what?
}
```

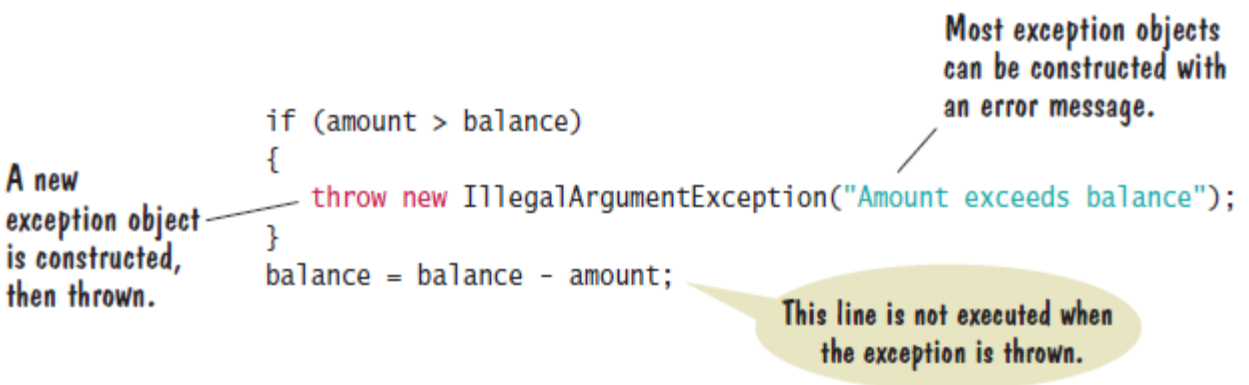
  - Handling Errors
  - This is more complex. You need to 'catch' each possible exception and react to it appropriately

- Handling recoverable errors can be done:
  - Simply: exit the program
  - User-friendly: Ask the user to correct the error

Syntax 7.1 Throwing an Exception

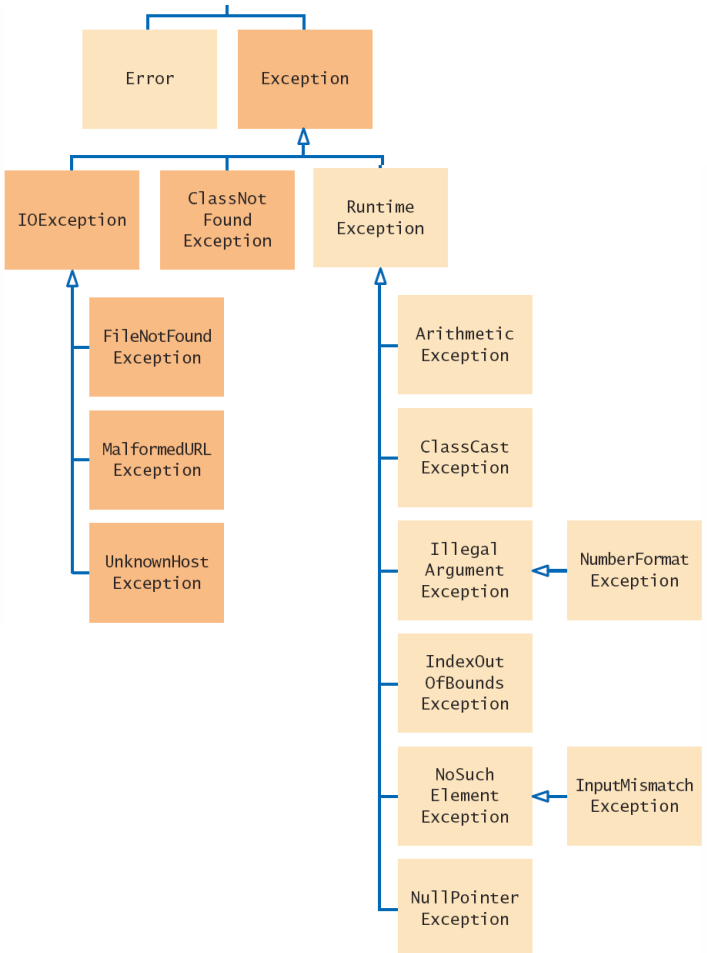
- When you throw an exception, you are throwing an object of an exception class
  - Choose wisely!
  - You can also pass a descriptive `String` to most exception objects

Syntax `throw` *exceptionObject*;



Exception Classes

- Partial hierarchy of exception classes
  - More general are above
  - More specific are below
  - Darker are Checked exceptions



Catching Exceptions

- Exceptions that are thrown must be 'caught' somewhere in your program

```
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("Input was not a number");
}
```

Surround method calls that can throw exceptions with a 'try block'.

FileNotFoundException

NoSuchElementException

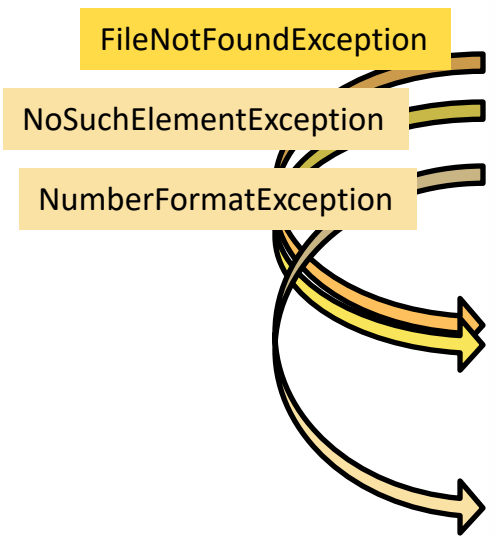
NumberFormatException

Write 'catch blocks' for each possible exception.

It is customary to name the exception parameter either '`e`' or '`exception`' in the catch block.

Catching Exceptions

- When an exception is detected, execution 'jumps' immediately to the first matching `catch` block
  - `IOException` catches `FileNotFoundException` and `NoSuchElementException` is not caught




```
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("Input was not a number");
}
```

Syntax 7.2 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.

This is the exception that was thrown.

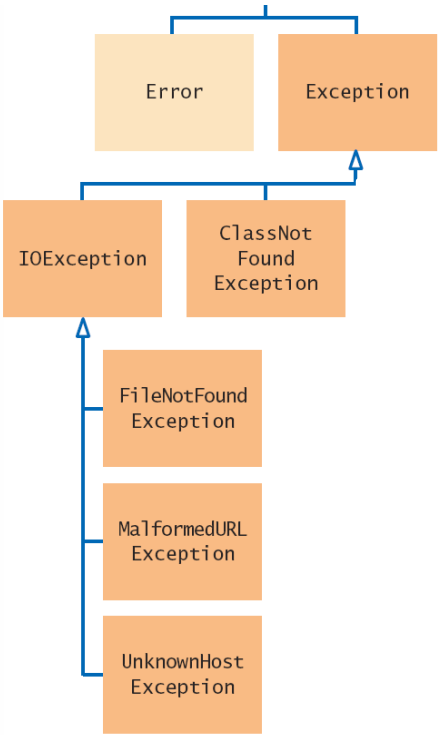
When an IOException is thrown, execution resumes here.

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

A FileNotFoundException is a special case of an IOException.

Checked Exceptions

- Throw/catch applies to three types of exceptions:
  - **Error:** Internal Errors
    - not considered here
  - **Unchecked:** RunTime Exceptions
    - Caused by the programmer
    - Compiler **does not check** how you handle them
  - **Checked:** All other exceptions
    - Not the programmer's fault
    - Compiler **checks** to make sure you handle these
    - Shown darker in Exception Classes



Syntax 7.3 The `throws` Clause

- Methods that use other methods that may throw exceptions must be declared as such
  - Declare all **checked** exceptions a method throws
  - You may also list **unchecked** exceptions

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

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

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

You may also list unchecked exceptions.

- Coded on a method's header
- Signals a caller of your method that it may encounter the named exceptions
- Provides a way to pass an exception up the calling chain to a more appropriate handler
- Ensures exceptions are not lost

The `throws` Clause

- If a method handles a checked exception internally, it will no longer throw the exception.
  - The method does not need to declare it in the throws clause
- Declaring exceptions in the `throws` clause 'passes the buck' to the calling method to handle it or pass it along.

Closing Resources

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

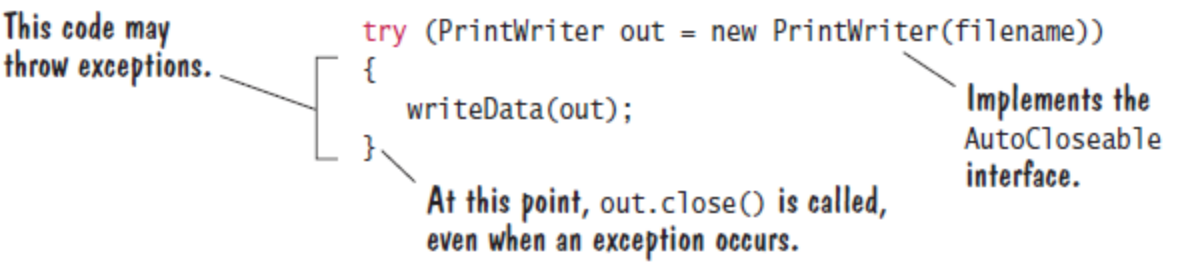
```
PrintWriter out = new PrintWriter(filename);
writeData(out);
out.close();  // May never get here
```

- 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

Syntax 7.3 try-with-resources

Syntax

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



Self Check 7.16

Suppose `balance` is 100 and `amount` is 200. What is the value of `balance` after these statements?

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

Answer: It is still 100. The last statement was not executed because the exception was thrown.

Self Check 7.17

When depositing an amount into a bank account, we don't have to worry about overdrafts—except when the amount is negative. Write a statement that throws an appropriate exception in that case.

Answer:

```
if (amount < 0)
{
   throw new IllegalArgumentException("Negative amount");
}
```

Self Check 7.18

Consider the method

```
public static void main(String[] args)
{
   try
   {
      Scanner in = new Scanner(new File("input.txt"));
      int value = in.nextInt();
      System.out.println(value);
   }
   catch (IOException exception)
   {
      System.out.println("Error opening file.");
   }
}
```

Suppose the file with the given file name exists and has no contents. Trace the flow of execution.

Answer: The `Scanner` constructor succeeds because the file exists. The `nextInt` method throws a `NoSuchElementException`. This is *not* an `IOException`. Therefore, the error is not caught. Because there is no other handler, an error message is printed and the program terminates.

Self Check 7.19

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`.

Self Check 7.20

Is there a difference between catching checked and unchecked exceptions?

Answer: No. You can catch both exception types in the same way, as you can see in the code example on page 354.

Self Check 7.21

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.

Programming Tip

- Throw Early
  - When a method detects a problem that it cannot solve, it is better to throw an exception rather than try to come up with an imperfect fix
- Catch Late
  - Conversely, a method should only catch an exception if it can really remedy the situation
  - Otherwise, the best remedy is simply to have the exception propagate to its caller, allowing it to be caught by a competent handler

Programming Tip

- Do Not Squelch Exceptions
  - When you call a method that throws a checked exception and you haven't specified a handler, the compiler complains
  - It is tempting to write a 'do-nothing' catch block to 'squelch' the compiler and come back to the code later. **Bad Idea!**
    - Exceptions were designed to transmit problem reports to a competent handler
    - Installing an incompetent handler simply hides an error condition that could be serious

Programming Tip

- Do Throw Specific Exceptions
  - When throwing an exception, choose an exception class that describes the situation as closely as possible
  - You can provide a new exception class yourself for particular errors

Special Topic

- Assertions
  - An assertion is a condition that you believe to be true at all times in a particular program location
  - An assertion check tests whether an assertion is true
  - When an assertion is correct, the program works in the normal way
  - When an assertion fails, and assertion checking is enabled, an `AssertionError` is thrown, causing the program to terminate

```
public double deposit(double amount)
{
    assert amount >= 0;
    balance = balance + amount;
}
```

By default, assertion checking is disabled when you execute a program. To execute a program with assertion checking turned on, use this command:

```
java -enableassertions MainClass
```

Special Topic

- `finally` is an optional clause in a `try/catch` block
  - Used to take some action in a method whether an exception is thrown or not
  - Rarely required because most Java library classes that require cleanup implement the `AutoCloseable` interface

```
public void printOutput(String filename) throws IOException
{
   PrintWriter out = new PrintWriter(filename);
   try
   {
      writeData(out);    // Method may throw an I/O Exception
   }
   finally
   {
      out.close();
   }
}
```

Once a try block is entered, the statements in a `finally` clause are guaranteed to be executed, whether or not an exception is thrown.

Handling Input Errors

- File Reading Application Example
  - Goal: Read a file of data values
    - First line is the count of values
    - Remaining lines have values
  - Risks:
    - The file may not exist
      - `Scanner` constructor will throw an exception
      - `FileNotFoundException`
    - The file may have data in the wrong format
      - Doesn't start with a `count`
        - `NoSuchElementException`
      - Too many items (`count` is too low)
        - `IOException`

```
3
1.45
-2.1
0.05
```

Handling Input Errors: `main`

- Outline for method with all exception handling

```
boolean done = false;
while (!done)
{
  try
  {
    // Prompt user for file name
    double[] data = readFile(filename);   // May throw exceptions
    // Process data
    done = true;
  }
  catch (FileNotFoundException exception)
  {      System.out.println("File not found.");  }
  catch (NoSuchElementException exception)
  {      System.out.println("File contents invalid.");  }
  catch (IOException exception)
  {      exception.printStackTrace();  }
}
```

Handling Input Errors: `readFile`

- Constructs the `Scanner` object and calls the `readData` method
- Does not handle exceptions
- Errors with the file are passed to the caller
- The file is closed even when an exception occurs
- Does not throw `File-Not-FoundException` because that is a special case of `IOException`

```
public static double[] readFile(String filename) throws IOException
{
   File inFile = new File(filename);
   try (Scanner in = new Scanner(inFile))
   {
      return readData(in);
   }
}
```

Handling Input Errors: readData

- No exception handling (no `try` or `catch` clauses)
- `throw` creates an `IOException` object and exits
- unchecked `NoSuchElementException` can occur

```
public static double[] readData(Scanner in) throws IOException
{
   int numberOfValues = in.nextInt();   // NoSuchElementException
   double[] data = new double[numberOfValues];
   for (int i = 0; i < numberOfValues; i++)
   {
      data[i] = in.nextDouble();          // NoSuchElementException
   }
   if (in.hasNext())
   {
      throw new IOException("End of file expected");
   }
   return data;
}
```

Self Check 7.22

Why doesn't the `readFile` method catch any exceptions?

Answer: The exceptions are better handled in the `main` method.

Self Check 7.23

What happens to the `Scanner` object if the `readData` method throws an exception?

Answer: The `close` method is called on the `Scanner` object before the exception is propagated to its handler.

Self Check 7.24

What happens to the `Scanner` object if the `readData` method doesn't throw an exception?

Answer: The `close` method is called on the `Scanner` object before the `readFile` method returns to its caller.

Self Check 7.25

Suppose the user specifies a file that exists and is empty. Trace the flow of execution in the `DataAnalyzer` program.

Answer: `main` calls `readFile`, which calls `readData`. The call `in.nextInt()` throws a `NoSuchElementException`. The `readFile` method doesn't catch it, so it propagates back to `main`, where it is caught. An error message is printed, and the user can specify another file.

Self Check 7.26

Why didn't the `readData` method call `hasNextInt/hasNextDouble` to ensure that the `NoSuchElementException` is not thrown?

Answer: We *want* to throw that exception, so that someone else can handle the problem of a bad data file.