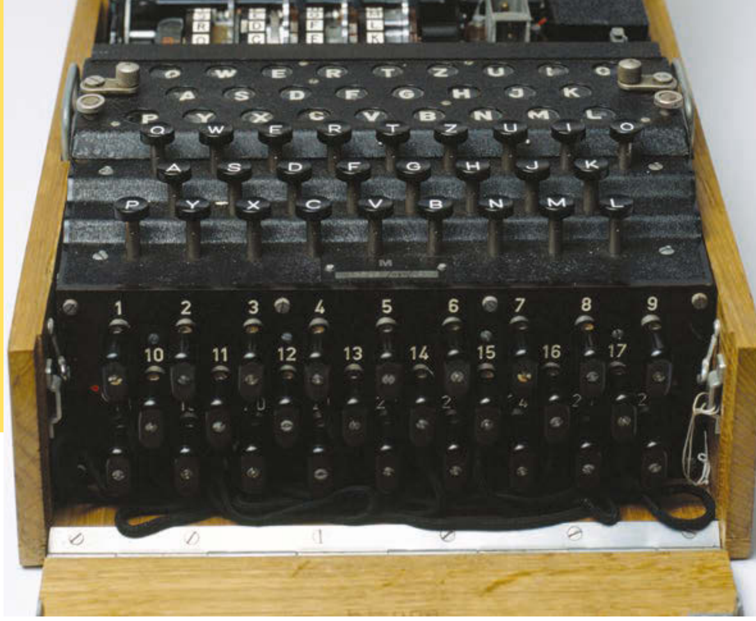
CHAPTER 7

INPUT/OUTPUT AND EXCEPTION HANDLING


James King-Holmes/Bletchley ParkTrust/Photo Researchers, Inc.

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

CHAPTER CONTENTS

In this chapter, you will learn how to read and write files—a very useful skill for processing real world data. As an application, you will learn how to encrypt data. (The Enigma machine shown here is an encryption device used by Germany in World War II. Pioneering British computer scientists broke the code and were able to intercept encoded messages, which was a significant help in winning the war.) The remainder of this chapter tells you how your programs can report and recover from problems, such as missing files or malformed content, using the exception-handling mechanism of the Java language.

James King-Holmes/Bletchley Park Trust/Photo Researchers, Inc.

7.1 Reading and Writing Text Files

We begin this chapter by discussing the common task of reading and writing files that contain text. Examples of text files include not only files that are created with a simple text editor, such as Windows Notepad, but also Java source code and HTML files.

Use the `Scanner` class for reading text files.

In Java, the most convenient mechanism for reading text is to use the `Scanner` class. You already know how to use a `Scanner` for reading console input. To read input from a disk file, the `Scanner` class relies on another class, `File`, which describes disk files and directories. (The `File` class has many methods that we do not discuss in this book; for example, methods that delete or rename a file.)

To begin, construct a `File` object with the name of the input file:

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

Then use the `File` object to construct a `Scanner` object:

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

This `Scanner` object reads text from the file `input.txt`. You can use the `Scanner` methods (such as `nextInt`, `nextDouble`, and `next`) to read data from the input file.

For example, you can use the following loop to process numbers in the input file:

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

When writing text files, use the `PrintWriter` class and the `print`/`println`/`printf` methods.

To write output to a file, you construct a `PrintWriter` object with the desired file name, for example

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

If the output file already exists, it is emptied before the new data are written into it. If the file doesn't exist, an empty file is created.

The `PrintWriter` class is an enhancement of the `PrintStream` class that you already know—`System.out` is a `PrintStream` object. 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 all files when you are done processing them.

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.

The following program puts these concepts to work. It reads a file containing numbers, and writes the numbers to another file, lined up in a column and followed by their total.

For example, if the input file has the contents

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

then 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
```

There is 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 compiler insists that we specify what the program should do when that happens. Similarly, the `PrintWriter` constructor generates this exception if it cannot open the file for writing. (This can happen if the name is illegal or the user does not have the authority to create a file in the given location.) In our sample program, we want to terminate the `main` method if the exception occurs. To achieve this, we label the `main` method with a `throws` declaration:

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

You will see in Section 7.4 how to deal with exceptions in a more professional way.

The `File`, `PrintWriter`, and `FileNotFoundException` classes are contained in the `java.io` package.

sec01/Total.java

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

/**
   This program reads a file with numbers, and writes the numbers to another
   file, lined up in a column and followed by their total.
*/
public class Total
{
   public static void main(String[] args) throws FileNotFoundException
   {
      // Prompt for the input and output file names

      Scanner console = new Scanner(System.in);
      System.out.print("Input file: ");
      String inputFileName = console.next();
      System.out.print("Output file: ");
      String outputFileName = console.next();

      // Construct the Scanner and PrintWriter objects for reading and writing

      File inputFile = new File(inputFileName);
      Scanner in = new Scanner(inputFile);
      PrintWriter out = new PrintWriter(outputFileName);

      // Read the input and write the output

      double total = 0;

      while (in.hasNextDouble())
      {
         double value = in.nextDouble();
         out.printf("%15.2f%n", value);
         total = total + value;
      }

      out.printf("Total: %8.2f%n", total);

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

SELF CHECK

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

Practice It Now you can try these exercises at the end of the chapter: R7.1, R7.2, E7.1.

Common Error 7.1

Backslashes in File Names

When you specify a file name as a string literal, and the name contains backslash characters (as in a Windows file name), you must supply each backslash twice:

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

A single backslash inside a quoted string is an **escape character** that is combined with the following character to form a special meaning, such as `\n` for a newline character. The `\\` combination denotes a single backslash.

When a user supplies a file name to a program, however, the user should not type the backslash twice.

Common Error 7.2

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");
```

However, this does *not* work for a `Scanner`. The statement

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

does *not* open a file. Instead, it simply reads through the string: `in.next()` returns the string `"input.txt"`. (This is occasionally useful—see Section 7.2.4.)

You must simply remember to use `File` objects in the `Scanner` constructor:

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

Special Topic 7.1

Reading Web Pages

You can read the contents of a web page with this sequence of commands:

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

Now simply read the contents of the web page with the `Scanner` in the usual way. The `URL` constructor and the `openStream` method can throw an `IOException`, so you need to tag the `main` method with `throws IOException`. (See Section 7.4.3 for more information on the `throws` clause.)

The `URL` class is contained in the `java.net` package.

FULL CODE EXAMPLE

Go to `wiley.com/go/bjlo2code` to download a program that reads data from a web page.

Special Topic 7.2

File Dialog Boxes

In a program with a graphical user interface, you will want to use a file dialog box (such as the one shown in the figure below) whenever the users of your program need to pick a file. The `JFileChooser` class implements a file dialog box for the Swing user-interface toolkit.

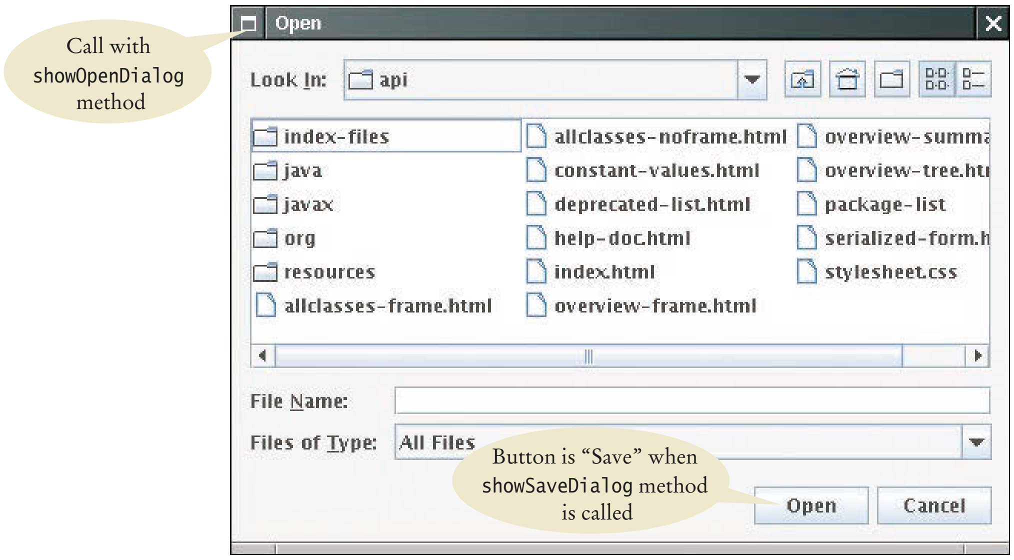
The `JFileChooser` class has many options to fine-tune the display of the dialog box, but in its most basic form it is quite simple: Construct a file chooser object; then call the `showOpenDialog` or `showSaveDialog` method. Both methods show the same dialog box, but the button for selecting a file is labeled "Open" or "Save", depending on which method you call.

For better placement of the dialog box on the screen, you can specify the user-interface component over which to pop up the dialog box. If you don't care where the dialog box pops up, you can simply pass `null`. The `showOpenDialog` and `showSaveDialog` methods return either `JFileChooser.APPROVE_OPTION`, if the user has chosen a file, or `JFileChooser.CANCEL_OPTION`, if the user canceled the selection. If a file was chosen, then you call the `getSelectedFile` method to obtain a `File` object that describes the file. Here is a complete example:

FULL CODE EXAMPLE

Go to wiley.com/go/bjlo2code to download a program that demonstrates how to use a file chooser.

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



A `JFileChooser` Dialog Box

Special Topic 7.3

Reading and Writing Binary Data

You use the `Scanner` and `PrintWriter` classes to read and write text files. Text files contain 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. While modifying binary files is quite challenging and beyond the scope of this book, we give you a simple example of copying binary data from a web site to a file.

You use an `InputStream` to read binary data. For example,

```
URL imageLocation = new URL("http://horstmann.com/java4everyone/duke.gif");
InputStream in = imageLocation.openStream();
```

To write binary data to a file, use a `FileOutputStream`:

```
FileOutputStream out = new FileOutputStream("duke.gif");
```

The `read` method of an input stream reads a single byte and returns –1 when no further input is available. The `write` method of an output stream writes a single byte.

The following loop copies all bytes from an input stream to an output stream:

```
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); }
}
```

7.2 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.

7.2.1 Reading Words

The `next` method reads a string that is delimited by white space.

The `next` method of the `Scanner` class reads the next string. Consider the loop

```
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
```

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.)

Here is precisely what happens when the `next` method is executed. Input characters that are white space are *consumed*—that is, removed from the input. However, they do not become part of the word. The first character that is not white space becomes the first character of the word. More characters are added until either another white space character occurs, or the end of the input file has been reached. However, if the end of the input file is reached before any character was added to the word, a “no such element exception” occurs.

Sometimes, you want to read just the words and discard anything that isn't a letter. You achieve this task by calling the `useDelimiter` method on your `Scanner` object:

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

Here, we set the character pattern that separates words to "any sequence of characters other than letters". (See Special Topic 7.4.) With this setting, punctuation and numbers are not included in the words returned by the `next` method.

7.2.2 Reading Characters

Sometimes, you want to read a file one character at a time. You will see an example in Section 7.3 where we encrypt the characters of a file. You achieve this task by calling the `useDelimiter` method on your `Scanner` object with an empty string:

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

Now each call to `next` returns a string consisting of a single character. Here is how you can process the characters:

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

7.2.3 Classifying Characters

The `Character` class has methods for 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. Each of them has an argument of type `char` and returns a `boolean` value (see Table 1).

For example, the call

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

returns `true` if `ch` is a digit (`'0'` . . . `'9'` or a digit in another writing system—see Computing & Society 2.2), `false` otherwise.

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

7.2.4 Reading Lines

The nextLine method reads an entire line.

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 next input line (without the newline character) is placed into the string line. You can then take the line apart for further processing.

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.

Here is a typical example of processing lines in a file. A file with population data from the CIA Fact Book site (https://www.cia.gov/library/publications/the-world-factbook/index.html) contains lines such as the following:

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

Because some country names have more than one word, it would be tedious to read this file using the next method. For example, after reading United, how would your program know that it needs to read another word before reading the population count?

Instead, read each input line into a string:

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

Use the isDigit and isWhiteSpace methods introduced to find out where the name ends and the number starts.

Locate the first digit:

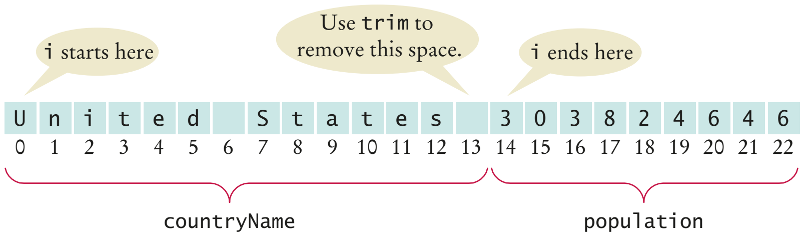
```
int i = 0;
while (!Character.isDigit(line.charAt(i))) { i++; }
```

Then extract the country name and population:

```
String countryName = line.substring(0, i);
String population = line.substring(i);
```

However, the country name contains one or more spaces at the end. Use the trim method to remove them:

```
countryName = countryName.trim();
```



The trim method returns the string with all white space at the beginning and end removed.

There is one additional problem. The population is stored in a string, not a number. In Section 7.2.6, you will see how to convert the string to a number.

7.2.5 Scanning a String

In the preceding section, you saw how to break a string into parts by looking at individual characters. Another approach is occasionally easier. You can use a `Scanner` object to read the characters from a string:

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

Then you can use `lineScanner` like any other `Scanner` object, reading words and numbers:

```
String countryName = lineScanner.next(); // Read first word
// Add more words to countryName until number encountered
while (!lineScanner.hasNextInt())
{
   countryName = countryName + " " + lineScanner.next();
}
int populationValue = lineScanner.nextInt();
```

7.2.6 Converting Strings to Numbers

Sometimes you have a string that contains a number, such as the `population` string in Section 7.2.4. For example, suppose that the string is the character sequence `"303824646"`. To get the integer value `303824646`, you use the `Integer.parseInt` method:

```
int populationValue = Integer.parseInt(population);
   // populationValue is the integer 303824646
```

If a string contains the digits of a number, you use the `Integer.parseInt` or `Double.parseDouble` method to obtain the number value.

To convert a string containing floating-point digits to its floating-point value, use the `Double.parseDouble` method. For example, suppose `input` is the string `"3.95"`.

```
double price = Double.parseDouble(input);
   // price is the floating-point number 3.95
```

You need to be careful when calling the `Integer.parseInt` and `Double.parseDouble` methods. The argument must be a string containing the digits of an integer, without any additional characters. Not even spaces are allowed! In our situation, we happen to know that there won't be any spaces at the beginning of the string, but there might be some at the end. Therefore, we use the `trim` method:

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

How To 7.1 on page 348 continues this example.

7.2.7 Avoiding Errors When Reading Numbers

You have used the `nextInt` and `nextDouble` methods of the `Scanner` class many times, but here we will have a look at what happens in "abnormal" situations. Suppose you call

```
int value = in.nextInt();
```

The `nextInt` method recognizes numbers such as `3` or `-21`. However, if the input is not a properly formatted number, an "input mismatch exception" occurs. For example, consider an input containing the characters

```
  2 1 s t   c e n t u r y
```

White space is consumed and the word `21st` is read. However, this word is not a properly formatted number, causing an input mismatch exception in the `nextInt` method.

If there is no input at all when you call `nextInt` or `nextDouble`, a "no such element exception" occurs. To avoid exceptions, use the `hasNextInt` method to screen the input when reading an integer. For example,

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

Similarly, you should call the `hasNextDouble` method before calling `nextDouble`.

7.2.8 Mixing Number, Word, and Line Input

The `nextInt`, `nextDouble`, and `next` methods *do not* consume the white space that follows the number or word. This can be a problem if you alternate between calling `nextInt`/`nextDouble`/`next` and `nextLine`. Suppose a file contains country names and population values in this format:

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

Now suppose you read the file with these instructions:

```
while (in.hasNextLine())
{
   String countryName = in.nextLine();
   int population = in.nextInt();
   Process the country name and population.
}
```

Initially, the input contains

```
C h i n a \n 1 3 3 0 0 4 4 6 0 5 \n I n d i a \n
```

After the first call to the `nextLine` method, the input contains

```
1 3 3 0 0 4 4 6 0 5 \n I n d i a \n
```

After the call to `nextInt`, the input contains

```
\n I n d i a \n
```

Note that the `nextInt` call did *not* consume the newline character. Therefore, the second call to `nextLine` reads an empty string!

The remedy is to add a call to `nextLine` after reading the population value:

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

The call to `nextLine` consumes any remaining white space *and* the newline character.

7.2.9 Formatting Output

When you write numbers or strings, you often want to control how they appear. For example, dollar amounts are usually formatted with two significant digits, such as

```
Cookies:       3.20
```

You know from Section 2.3.2 how to achieve this output with the `printf` method. In this section, we discuss additional options of the `printf` method.

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

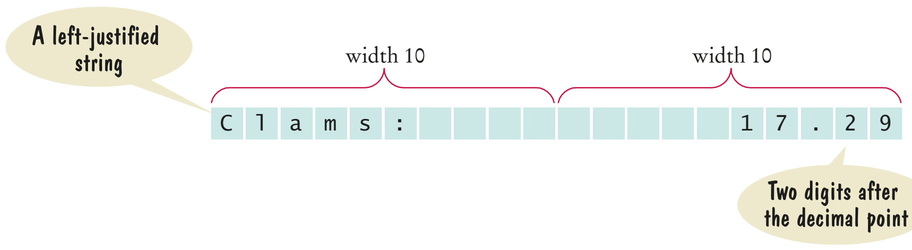
```
Cookies:        3.20
Linguine:       2.95
Clams:         17.29
```

Note that the item strings line up to the left, whereas the numbers line up to the right. By default, the `printf` method lines up values to the right. To specify left alignment, you add a hyphen (-) before the field width:

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

Here, we have two format specifiers.

- `%-10s` formats a left-justified string. The string `items[i] + ":"` is padded with spaces so it becomes ten characters wide. The - indicates that the string is placed on the left, followed by sufficient spaces to reach a width of 10.
- `%10.2f` formats a floating-point number, also in a field that is ten characters wide. However, the spaces appear to the left and the value to the right.



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

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`

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:

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. There are quite a few format types—Table 3 shows the most important ones.

FULL CODE EXAMPLE
Go to wiley.com/go/bjlo2code to download a program that processes a file containing a mixture of text and numbers.

SELF CHECK

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();
```

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();
```

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();
```

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

Practice It Now you can try these exercises at the end of the chapter: E7.4, E7.6, E7.7.

Special Topic 7.4

Regular Expressions

A **regular expression** describes a character pattern. For example, numbers have a simple form. They contain one or more digits. The regular expression describing numbers is `[0-9]+`. The set `[0-9]` denotes any digit between `0` and `9`, and the `+` means "one or more".

The search commands of professional programming editors understand regular expressions. Moreover, several utility programs use regular expressions to locate matching text. A commonly used program that uses regular expressions is **grep** (which stands for "global regular expression print"). You can run grep from a command line or from inside some compilation environments. Grep is part of the UNIX operating system, and versions are available for Windows. It needs a regular expression and one or more files to search. When grep runs, it displays a set of lines that match the regular expression.

Suppose you want to find all magic numbers (see Programming Tip 2.2) in a file.

```
grep [0-9]+ Homework.java
```

lists all lines in the file `Homework.java` that contain sequences of digits. That isn't terribly useful; lines with variable names `x1` will be listed. OK, you want sequences of digits that do *not* immediately follow letters:

```
grep [^A-Za-z][0-9]+ Homework.java
```

The set `[^A-Za-z]` denotes any characters that are *not* in the ranges `A` to `Z` and `a` to `z`. This works much better, and it shows only lines that contain actual numbers.

The `useDelimiter` method of the `Scanner` class accepts a regular expression to describe delimiters—the blocks of text that separate words. As already mentioned, if you set the delimiter pattern to `[^A-Za-z]+`, a delimiter is a sequence of one or more characters that are not letters.

There are two useful methods of the `String` class that use regular expressions. The `split` method splits a string into an array of strings, with the delimiter specified as a regular expressions. For example,

```
String[] tokens = line.split("\\s+");
```

splits input along white space. The `replaceAll` method yields a string in which all matches of a regular expression are replaced with a string. For example, `word.replaceAll("[aeiou]", "")` is the word with all vowels removed.

For more information on regular expressions, consult one of the many tutorials on the Internet by pointing your search engine to "regular expression tutorial".

Special Topic 7.5

Reading an Entire File

In the preceding section, you saw how to read lines, words, and characters from a file. Alternatively, you can read the entire file into a list of lines, or into a single string. To do so, use the `Files` class, which provides methods to work with files and directories. The `Files` class requires that you specify file paths as `Path` objects. The `Paths.get` method turns a string containing a file path into such an object. Use these commands:

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

HOW TO 7.1 Processing Text Files

Processing text files that contain real data can be surprisingly challenging. This How To gives you step-by-step guidance.

Problem Statement Read two country data files, `worldpop.txt` and `worldarea.txt` (supplied with the book's companion code). Both files contain the same countries in the same order. Write a file `world_pop_density.txt` that contains country names and population densities (people per square km), 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
. . .
```

© Oksana Perkins/iStockphoto.

Singapore is one of the most densely populated countries in the world.

Step 1 Understand the processing task.

As always, you need to have a clear understanding of the task before designing a solution. Can you carry out the task by hand (perhaps with smaller input files)? If not, get more information about the problem.

One important aspect that you need to consider is whether you can process the data as it becomes available, or whether you need to store it first. For example, if you are asked to write out sorted data, you first need to collect all input, perhaps by placing it in an array list. However, it is often possible to process the data "on the go", without storing it.

In our example, we can read each file a line at a time and compute the density for each line because our input files store the population and area data in the same order.

The following pseudocode describes our processing 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.
```

Step 2 Determine which files you need to read and write.

This should be clear from the problem. In our example, there are two input files, the population data and the area data, and one output file.

Step 3 Choose a mechanism for obtaining the file names.

There are three options:

- Hard-coding the file names (such as `"worldpop.txt"`).
- Asking the user:

```
Scanner in = new Scanner(System.in);
System.out.print("Enter filename: ");
String inFile = in.nextLine();
```

- Using command-line arguments for the file names.

In our example, we use hard-coded file names for simplicity.

Step 4 Choose between line, word, and character-based input.

As a rule of thumb, read lines if the input data is grouped by lines. That is the case with tabular data, such as in our example, or when you need to report line numbers.

When gathering data that can be distributed over several lines, then it makes more sense to read words. Keep in mind that you lose all white space when you read words.

Reading characters is mostly useful for tasks that require access to individual characters. Examples include analyzing character frequencies, changing tabs to spaces, or encryption.

Step 5 With line-oriented input, extract the required data.

It is simple to read a line of input with the `nextLine` method. Then you need to get the data out of that line. You can extract substrings, as described in Section 7.2.4.

Typically, you will use methods such as `Character.isWhitespace` and `Character.isDigit` to find the boundaries of substrings.

If you need any of the substrings as numbers, you must convert them, using `Integer.parseInt` or `Double.parseDouble`.

Step 6 Use methods to factor out common tasks.

Processing input files usually has repetitive tasks, such as skipping over white space or extracting numbers from strings. It really pays off to develop a set of methods to handle these tedious operations.

In our example, we have two common tasks that call for helper methods: extracting the country name and the value that follows. We will implement methods

```
public static String extractCountry(String line)
public static double extractValue(String line)
```

These methods are implemented as described in Section 7.2.4.

Here is the complete source code (`how_to_1/PopulationDensity.java`).

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

/**
   This program reads data files of country populations and areas and prints the
   population density for each country.
*/
public class PopulationDensity
{
   public static void main(String[] args) throws FileNotFoundException
   {
      // Construct Scanner objects for input files

      Scanner in1 = new Scanner(new File("worldpop.txt"));
      Scanner in2 = new Scanner(new File("worldarea.txt"));

      // Construct PrintWriter for the output file

      PrintWriter out = new PrintWriter("world_pop_density.txt");

      // Read lines from each file

      while (in1.hasNextLine() && in2.hasNextLine())
      {
         String line1 = in1.nextLine();
         String line2 = in2.nextLine();
```

```
            // Extract country and associated value
            String country = extractCountry(line1);
            double population = extractValue(line1);
            double area = extractValue(line2);

            // Compute and print the population density
            double density = 0;
            if (area != 0) // Protect against division by zero
            {
               density = population / area;
            }
            out.printf("%-40s%15.2f%n", country, density);
         }

         in1.close();
         in2.close();
         out.close();
      }

      /**
         Extracts the country from an input line.
         @param line a line containing a country name, followed by a number
         @return the country name
      */
      public static String extractCountry(String line)
      {
         int i = 0; // Locate the start of the first digit
         while (!Character.isDigit(line.charAt(i))) { i++; }
         return line.substring(0, i).trim(); // Extract the country name
      }

      /**
         Extracts the value from an input line.
         @param line a line containing a country name, followed by a value
         @return the value associated with the country
      */
      public static double extractValue(String line)
      {
         int i = 0; // Locate the start of the first digit
         while (!Character.isDigit(line.charAt(i))) { i++; }
         // Extract and convert the value
         return Double.parseDouble(line.substring(i).trim());
      }
   }
```

WORKED EXAMPLE 7.1 Analyzing Baby Names

In this Worked Example, you will use data from the Social Security Administration to analyze the most popular baby names. Go to wiley.com/go/bjlo2examples and download Worked Example 7.1.

© Nancy Ross/iStockphoto.