

Text and File Processing

11.1 Strings are Immutable Objects

11.1.1 String Tokenizers

11.2 Sequential Files

11.2.1 Output to Sequential Files

11.2.2 Input from Sequential Files

11.3 Sequential Input from the Command Window

11.4 Case Study: Payroll Processing

11.5 Exceptions and Exception Handlers

11.5.1 Restarting a Method with an Exception Handler

11.5.2 Interactive Input with Exception Handlers

11.6 Exceptions Are Objects

11.6.1 Programmer-Generated Exceptions

11.7 Summary

11.8 Programming Projects

11.9 Beyond the Basics

The applications presented in previous chapters accepted textual input by means of the user typing it into a window, one data item per line. Output was displayed in a similar way. Window-based input and output are inadequate for programs that must create and maintain large amounts of textual data, where multiple data items reside in each textual line and multiple lines of text reside in disk files. In this chapter, we study the following techniques:

- *how to decompose a line of text that contains multiple data items into its constituent parts, called tokens, by means of string tokenizers;*
- *how to read and write lines of text to and from sequential disk files;*
- *how to adapt the same techniques to read lines of text from the command window;*
- *how to use exception handlers to deal with errors in file processing.*

Because we work with files of text strings, we first examine how strings are represented in computer storage.

11.1 Strings are Immutable Objects

Since Chapter 2, we have worked with strings, which are represented in Java as sequences of characters enclosed by double quotes, e.g., "abcd". Special characters, like backspaces, tabs, and newlines, can be encoded within a string with these codings:

- `\b` (backspace)
- `\t` (tab)
- `\n` (newline)
- `\r` (return)
- `\"` (doublequote)
- `\'` (single quote)
- `\\` (backslash)

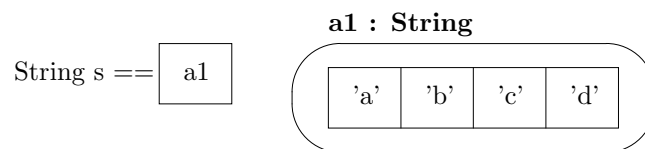
For example, `System.out.println("a\tb\nc\"d")` would print as

```
a b
c"d
```

In previous chapters, we have systematically ignored the fact that strings are objects in Java; there was no need to worry about this point. For the curious, we now reveal that a string is in fact saved in computer storage as an array of characters. The initialization,

```
String s = "abcd";
```

creates this object in storage:



This picture should make clear why the fundamental operations on strings are `length` and `charAt`. For example, `s.length()` returns 4, the length of the underlying array of characters, and `s.charAt(2)` returns `'c'`, because it is the character at element 2 of the underlying array.

Also, since strings are objects, it is no surprise that operations on strings are written as method invocations, where the string comes first, followed by a dot, followed by the operation (method) name, followed by arguments (if any), e.g., `s.charAt(2)`. At this point, you should study Table 5 of Chapter 3, which lists many of the methods associated with class `String`.

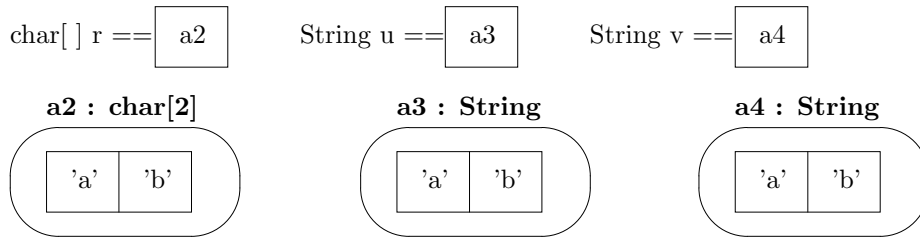
It is possible to create a string directly from an array of characters:

```

char[] r = new char[2];
r[0] = 'a';
r[1] = 'b';
String u = new String(r);
String v = "ab";

```

The above statements create the string, "ab", and assign its address to **t**:

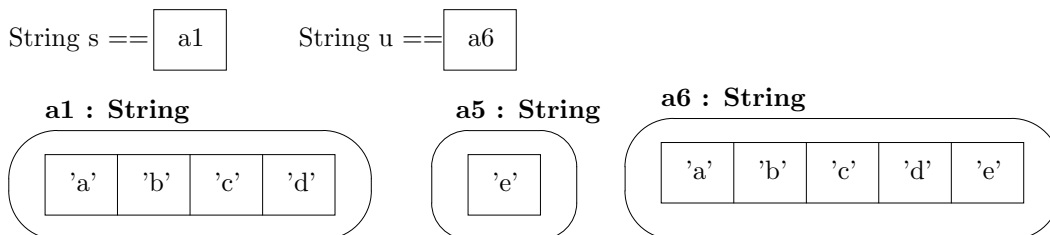


It is crucial that a separate object, **a3**, was created for **u**, because assignments to **r**'s elements must not affect **u**. Also, a separate string is created for **v**. For this reason, the expression, `u == v`, evaluates to **false**, because the two variables denote different objects! In contrast, the `equals` method produces a different answer—`u.equals(v)` computes to **true** (as does `v.equals(u)`), because `equals` compares the sequences of characters within the two objects.

Perhaps the most important feature of **String** is this:

*Java strings are immutable—once a **String** object is created, its contents cannot be altered.*

Unlike an array of characters, you cannot assign to a character (element) of a string. And when we write, say, `String u = s + "e"`, this creates a new string object; it does not alter the string denoted by **s**:



At this point, if we assigned,

```
s = u;
```

then **s**'s cell would hold the address, **a6**, also. This is no disaster, because the object at **a6** cannot be altered—it stays constant at "abcde". For this reason, we need not worry that character arrays underlie the strings we create, and we can pretend that

strings are primitive values, just like numbers. We return to this illusion for the remainder of the text.

(*BeginFootnote:* Java provides a version of strings that is mutable, that is, you can assign to the elements of the string. It is called a `StringBuffer` and is found in the package, `java.lang`. *EndFootnote.*)

With the assistance of the `length` and `charAt` methods, we can write loops that check two strings for equality, extract a substring from a string, find one string inside another, and so on. These are fundamental activities, and many of them are already written as methods that belong to `class String` in `java.lang`. Table 9 in Chapter 3 lists some of the most useful of these methods.

Exercises

Using only the `length`, `charAt`, and `+` methods, write the following methods:

1. `public boolean stringEquals(String s, String t)` returns `true` if strings `s` and `t` hold exactly the same sequences of characters and returns `false` otherwise.
2. `public String extractSubstring(String s, int start, int end)` returns the string within `s` that begins at index `start` and ends at index `end - 1`. If such a substring does not exist, because either of `start` or `end` has an invalid value, then `null` is returned.
3. `public String trimBlanks(String s)` returns a string that looks like `s` but all leading blanks and all trailing blanks are removed.

11.1.1 String Tokenizers

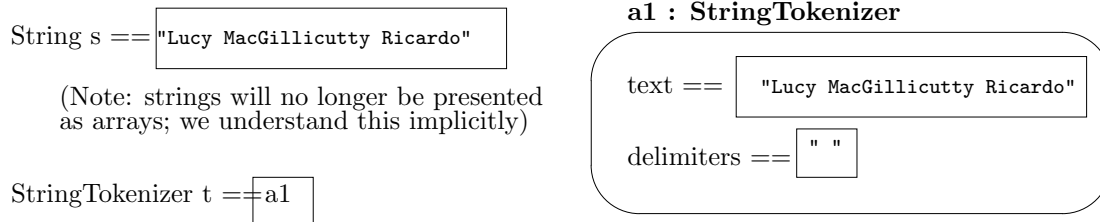
Our applications that read input have been designed to read one word—one *token*—per input line. But it is more convenient for a programmer to type multiple tokens on the same line. For example, a person would prefer to type her first, middle, and last names on one line rather than on three. And a user of the change-making program in Figure 2, Chapter 4, would prefer to type `$13.46` as the program's input rather than `13` on one line and `46` on another.

Of course, we can write an auxiliary method that decomposes a string into its tokens, but this task arises often, so the Java designers wrote a helper class, `class StringTokenizer` (found in the package, `java.util`), that contains methods for decomposing strings into tokens.

Perhaps we have a string, `s`, that contains several tokens that are separated by special characters, called *delimiters*. For example, a string, `Lucy MacGillicutty Ricardo`, contains the tokens `Lucy`, `MacGillicutty`, and `Ricardo`, separated by blank spaces; the blank serves as the delimiter. We construct a string tokenizer object that extracts the tokens as follows:

```
String s = "Lucy MacGillicutty Ricardo";
StringTokenizer t = new StringTokenizer(s, " ");
```

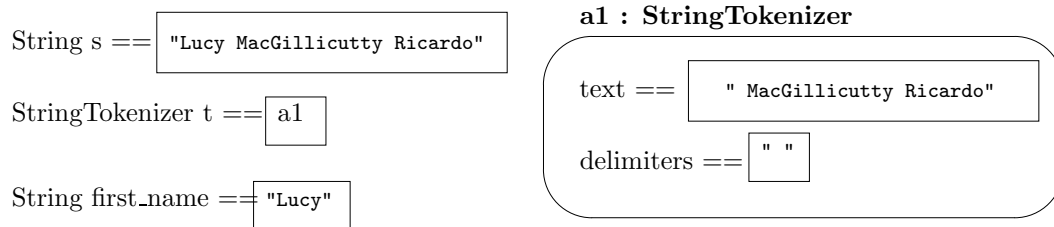
The string tokenizer object is constructed by the second statement, where the string to be disassembled and the character that is the delimiter are supplied as the two parameters. Here is a picture of the storage configuration:



To extract the first token from the string tokenizer object, we state,

```
String first_name = t.nextToken();
```

The `nextToken` method asks `t` to examine the string it holds, skip over leading delimiters (if any), and extract the maximal, nonempty sequence of characters until a delimiter is encountered. In the example, this assigns "Lucy" to `first_name`,



and we see that the text within the string tokenizer is shortened. Note that the delimiter, the blank space, is left attached to the front of the text within the string tokenizer.

When we extract the second token,

```
String middle_name = t.nextToken();
```

this skips over the leading blank, assigns "MacGillicutty" to `middle_name`, and leaves the object at `a1` holding the string, " Ricardo". This assignment,

```
String last_name = t.nextToken();
```

assigns "Ricardo" to `last_name`; the two leading blanks are discarded, and an empty string remains in the string tokenizer. If we attempt to extract a fourth token, the result is a `NoSuchElementException`, because no nonempty string can be returned.

Table 1 presents some of most useful methods for string tokenizers.

Here is a second example. Perhaps we wish to extract the tokens, 13 and 46 from the string, \$13.46. We do so most effectively with this sequence of statements:

Figure 11.1: string tokenizer methods

<code>class StringTokenizer</code>	
Constructor	
<code>new StringTokenizer(String text, String delim)</code>	Constructs a string tokenizer which extracts tokens from <code>text</code> , using as its separators all the characters listed in string <code>delim</code> .
Methods	
<code>nextToken(): String</code>	Examine the text held in the tokenizer: Remove leading delimiters, and remove the maximal, nonempty sequence of characters until a delimiter or the end of the text is encountered. Return the maximal, nonempty sequence of characters as the result.
<code>nextToken(String new_delimiters): String</code>	Like <code>nextToken()</code> , but first reset the delimiters used by the string tokenizer to the ones listed in <code>new_delimiters</code> .
<code>hasMoreTokens(): boolean</code>	Return whether the text held in the string tokenizer has any more tokens, based on the delimiters the tokenizer currently holds.
<code>countTokens(): int</code>	Return the number of tokens within the text held in the string tokenizer, based on the delimiters the tokenizer currently holds.

```
String s = "$13.46";
StringTokenizer t = new StringTokenizer(s, "$.");
String dollars = t.nextToken();
String cents = t.nextToken();
```

The second statement creates a string tokenizer that considers both \$ and . to be legal delimiters. A string tokenizer can use multiple distinct delimiters, and the delimiters are packaged as a string to the tokenizer's constructor method. The third statement assigns "13" to `dollars`, because the leading symbol, \$, is a delimiter and leading delimiters are always skipped over. The "13" is extracted and the . is detected as a delimiter. Next, `cents` is assigned 46, and the end of the string is encountered.

Exercises

1. What do each of the following display on the console?

(a)

```
String x = "12,3,4";
StringTokenizer y = new StringTokenizer(x, ",");
String s = y.nextToken();
```

```
int i = new Integer( y.nextToken() ).intValue()
        + new Integer( y.nextToken() ).intValue();
System.out.println(s + i);
```

(b)

```
String x = "12!3,4";
StringTokenizer y = new StringTokenizer(x, ",");
String s = y.nextToken();
int i = new Integer( y.nextToken() ).intValue();
StringTokenizer t = new StringTokenizer(s,"!");
String a = t.nextToken();
int j = new Integer( t.nextToken() ).intValue();
System.out.println(i+j);
```

2. Revise Figure 3, Chapter 3, so that the `MakeChange` reads its input as follows

```
String input =
    JOptionPane.showInputDialog("Type a dollars, cents amount for change: $:");
```

and uses a string tokenizer to assign values to variables `dollars` and `cents`.

3. Write an application that asks its user to type a complete sentence on one line. The application then displays in the console window the words in the sentence, one word per line, less any punctuation.

11.2 Sequential Files

Computer programs can read input from and write output to disk files. Simply stated, a file is a sequence of symbols stored on a diskette or on a computer's internal disk. There are two formats of files:

- a *character file*, which is a sequence of keyboard symbols, saved in the file as a sequence of bytes;
- a *binary file*, which is a sequence of ones-and-zeros.

A file is indeed one long sequence; for example, these three text lines,

```
Hello to    you!
How are you?
49
```

would be stored as a character file as this long sequence:

```
Hello to    you!\nHow are you?\n 49\n
```

Every symbol, including the exact quantity of blanks, are faithfully reproduced in the sequence. Newline characters (`'\n'`) mark the ends of lines. If you create a character file with a PC, the MS-based operating system marks the end of each line with the two-character sequence, `\r\n`, e.g.,

```
Hello to   you!\r\nHow are you?\r\n 49\r\n
```

The examples in this chapter are written so that this difference is unimportant.

If you could see with your own eyes the characters saved on the magnetic disk, you would not see letters like `H` and `e`. Instead, you would see that each character is translated to a standard sized, numeric coding. One standard coding is ASCII (pronounced “as-key”), which uses one byte to represent each character. Another coding is Unicode, which uses two bytes to represent a character. (Unlike ASCII, Unicode can represent accented and unlauded letters.) We will not concern ourselves with which coding your computer uses to encode characters in a character file; these details are hidden from us by the Java classes that read and write files.

A binary file is a sequence of ones and zeros. Binary files hold information that is not keyboard-based. For example, if one wishes to save information about the address numbers inside the computer’s primary storage, then a binary file is a good choice. Also, purely numerical data is often stored in a binary file: Inside the computer, an integer like 49 is used in its binary representation, 11001, and not in its character representation (the characters `'4'` and `'9'`). It is easier for the computer to do arithmetic on binary representations, and programs that work extensively with files of numbers work faster when the input and output data are saved in binary files—this saves the trouble of converting from character codings to binary codings and back again.

The Java language even allows a program to copy objects in computer storage to a binary file; this proves especially useful when one wants to archive the contents of an executing program for later use.

When we obtain input information from a file, we say that we *read* from the file; when we deposit output information into a file, we *write* to it. When we begin using a file, we say that we *open* it, and when we are finished using the file, we *close* it. The notions come from the old-fashioned concept of a filing cabinet whose drawer must be opened to get the papers (“files”) inside and closed when no longer used. Indeed, a computer must do a similar, internal “opening” of a disk file to see its contents, and upon conclusion a “closing” of the disk file to prevent damage to its contents.

Files can be read/written in two ways:

- A *sequential* file must be read (or written) from front to back; it is like a book whose pages can be turned in only one direction—forwards.
- A *random access* file allows reads (or writes) at any place within it; it is like a normal book that can be opened to any particular page number.

Sequential files are simple to manage and work well for standard applications. Random access files are used primarily for database applications, where specific bits of information must be found and updated. In this chapter, we deal only with sequential files.

When a Java program uses a sequential file, it must state whether the file is used for

- *input*: the file is read sequentially and cannot be written
- *output*: the file is written sequentially and cannot be read.

It is possible for a program to open a sequential file, read from it, close it, and then reopen it and write to it. If one desires a file that one can open and both read and write at will, then one must use a random-access file.

From this point onwards, we use the term *sequential file* to mean a sequential file of (codings of) characters. We will not deal with binary files in this text.

11.2.1 Output to Sequential Files

The `System.out` object used methods named `print` and `println` to write data to the console. We do something similar with sequential files.

The package, `java.io`, contains classes that have methods for file input and output. To write to a sequential character file, we construct an object that connects to the file and deposits text into it; say that the file that will hold the output is named `test.txt`:

1. First, we construct the object,

```
FileWriter w = new FileWriter("test.txt");
```

A `FileWriter` object holds the physical address of the named disk file and writes individual characters to the file.

2. Next, we *compose* the `FileWriter` with a `PrintWriter`:

```
PrintWriter outfile = new PrintWriter(w);
```

The `PrintWriter` object accepts `print(E)` and `println(E)` messages, where it translates the argument, `E`, into a sequence of characters and sends the characters, one by one, to the `FileWriter` object, `w`. For example,

```
outfile.println("Hello to you!");
```

Appends the string, "Hello to you!", followed by a newline character, into file `test.txt`.

Figure 11.2: sequential file output

```
import java.io.*;
/** Output1 writes three lines of characters to file test.txt */
public class Output1
{ public static void main(String[] args) throws IOException
  { PrintWriter outfile = new PrintWriter(new FileWriter("test.txt"));
    outfile.println("Hello to  you!");
    outfile.print("How are");
    outfile.println(" you?");
    outfile.println(47+2);
    outfile.close();
  }
}
```

Figure 2 displays an example application that writes three lines of characters into the sequential file named `test.txt`. The application's output-view object is `outfile`, and it can be constructed in just one statement as

```
PrintWriter outfile = new PrintWriter(new FileWriter("test.txt"));
```

When the `FileWriter` object is constructed, it locates and opens the file, `"test.txt"`, in the local directory (folder); if the file does not exist, a new, empty file with the name is created and opened. Files in other directories can be similarly opened provided a correct path name is specified, e.g., `"C:\\Fred\\DataFiles\\test.txt"` on an MS-based file system or `"/home/Fred/DataFiles/test.txt"` on a Unix-based system. (Recall that we must use `\\` to stand for one backslash in a Java string.)

The output-view object possesses methods, `println` and `print`, for writing strings and numbers, and a method, `close`, for closing a file. (As always, the `println` method appends a newline character, `'\n'`, to the end of the string that it writes.)

The program in Figure 2 writes three lines of text to `test.txt`, closes the file, and terminates. You can read the contents of `test.txt` with a text editor, and you will see

```
Hello to  you!
How are you?
49
```

The former contents of `test.txt`, if any, were destroyed.

The program begins with `import java.io.*`, which is the package where the new classes live, and `main`'s header line states `throws IOException`, which the Java compiler requires whenever file input/output appears. The phrase is a warning that that output transmission via might lead to an error (exception), e.g., the disk drive fails in the middle of output transmission. `IOExceptions` can be processed with exception

handlers, which we first studied in Chapter 4. We employ exception handlers later in the chapter.

It is perfectly acceptable to use multiple output files within the same program, and it is done in the expected way:

```
PrintWriter file1 = new PrintWriter(new FileWriter("test.txt"));
PrintWriter file2 = new PrintWriter(new FileWriter("AnotherFile.out"));
...
file1.println("abc");
file2.print("c");
file2.println();
file1.close();
...
```

The above code fragment creates two objects that write information to two files, `test.txt` and `AnotherFile.out`.

Exercises

1. Write an application that asks its user to type multiple lines of text into the console window; the application copies the lines the user types into a file, `text.txt`.
2. Modify the temperature-conversion program in Figure 2, Chapter 4, so that the program's output is displayed in a console window and also is written to a file, `change.out`.

11.2.2 Input from Sequential Files

For sequential file input, we must again construct an input-view object that connects to the sequential file that holds the input text. This is again constructed in two stages:

1. First, we construct an object that holds the file's physical address and reads individual characters from the file:

```
FileReader r = new FileReader("test.txt");
```

This constructs a `FileReader` object that holds the file's physical address and can read individual characters from the file. (If the file cannot be found, `new FileReader("test.txt")` generates an input-output exception.)

2. Next, we compose the `FileReader` object with a `BufferedReader` object, that has a method that reads one full text line:

```
BufferedReader infile = new BufferedReader(r);
```

Figure 11.3: program that copies a file

```
import java.io.*;
/** CopyFile copies the contents of an input file, f,
 * whose name is supplied by the user, into an output file, f.out */
public class CopyFile
{ public static void main(String[] args) throws IOException
  { String f = JOptionPane.showInputDialog("Input filename, please: ");
    // construct the view object that reads from the input file:
    BufferedReader infile = new BufferedReader(new FileReader(f));
    // construct the view object that writes to the output file:
    PrintWriter outfile = new PrintWriter(new FileWriter(f + ".out"));
    while ( infile.ready() ) // are there more lines to read in infile?
      { String s = infile.readLine();
        outfile.println(s);
      }
    infile.close();
    outfile.close();
  }
}
```

The `BufferedReader` object uses the `FileReader` object to collect a full line of characters, which it forms into a string. The message, `infile.readLine()`, returns the string as its result.

As an example, we write an application that reads the contents of a sequential file whose name is supplied by the user at the keyboard and copies the file, `f`, line by line, into another file, `f.out`. Figure 3 shows the application, class `CopyFile`.

The statement, `infile = new BufferedReader(new FileReader(f))`, creates a `BufferedReader` object that remembers the address of the filename, `f`, supplied by the user. In a similar fashion, `outfile = new PrintWriter(new FileWriter(f + ".out"))` creates an object that knows the address of the output file.

The while-loop within `main` uses a new method, `infile.ready()`, to determine if there are lines of text waiting to be read from `infile`. If so, then another line is read by `s = infile.readLine()` and is written by `outfile.println(s)`. (Note: if the `readLine` method attempts to read a string from a file that is exhausted, it returns `null` as its result.)

Errors can arise when we use disk files. For example, say that we tell the program to read a nonexistent file, `fred`. The program stops at the assignment to `infile` (Line 10) and announces:

```
java.io.FileNotFoundException: fred (No such file or directory)
...
```

```
at java.io.FileReader.<init>(...)
at CopyFile.main(CopyFile.java:10)
```

The error, `java.io.FileNotFoundException`, is an example of an `IOException`—the program has “thrown” an exception, as was warned in the header line of `main`. Such exceptions can be handled by exception handlers, as we see in the examples in the next section.

Exercises

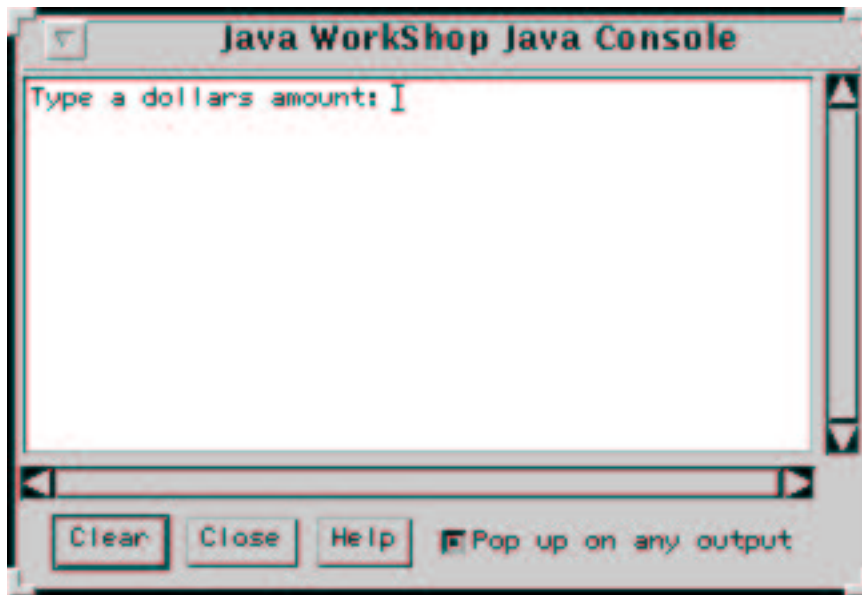
1. Create a file, `ints.dat`, which holds one integer per line. Write an application, `class Sum`, which reads `ints.dat` and prints the sum of the integers in the file.
2. Modify `CopyFile` in Figure 3 so that when it reads a line of text from the input file, it writes the words in the line to the output file, one word per output line. (Hint: use a string tokenizer.)
3. Modify `class VoteCount` in Figure 1, Chapter 8, so that the application reads its votes from a file, `votes.dat`. (Hint: Write a new input-view class, `class VoteReader`, whose `readInt` method reads a vote from `votes.dat`.)

11.3 Sequential Input from the Command Window

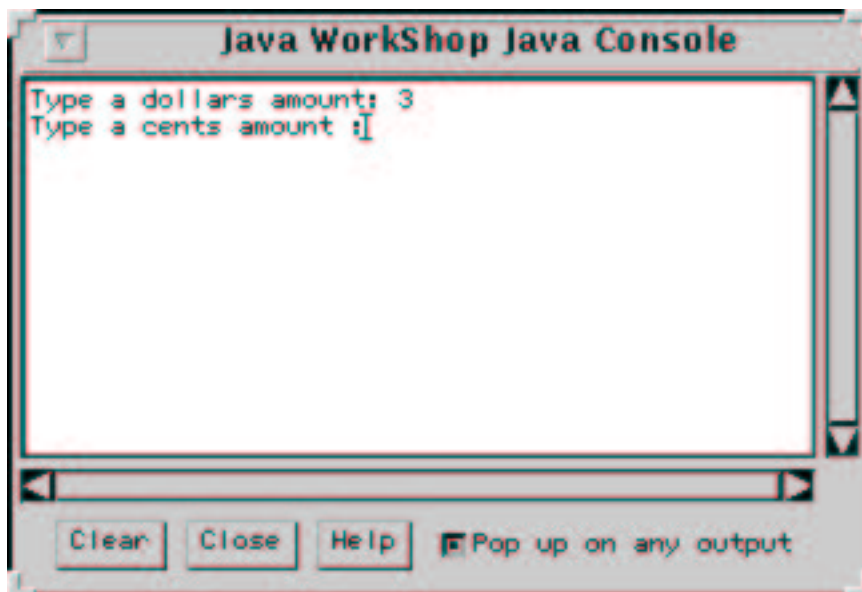
Since the beginning of this text, we used `System.out.println(E)` to display messages, `E`, within the command window. It is also possible to read input text that the user types into the command window—we use a preexisting object, `System.in`, which connects to the command window, along with the techniques seen in the previous section.

Here is an example: Recall that the change-making application, `MakeChange`, from Figure 3, Chapter 3, calculates the change one extracts from dollars-and-cents amounts. We can revise this program so the user starts it first and decides next what the dollars and cents inputs might be. When the revised application is started, this

request appears in the command window:

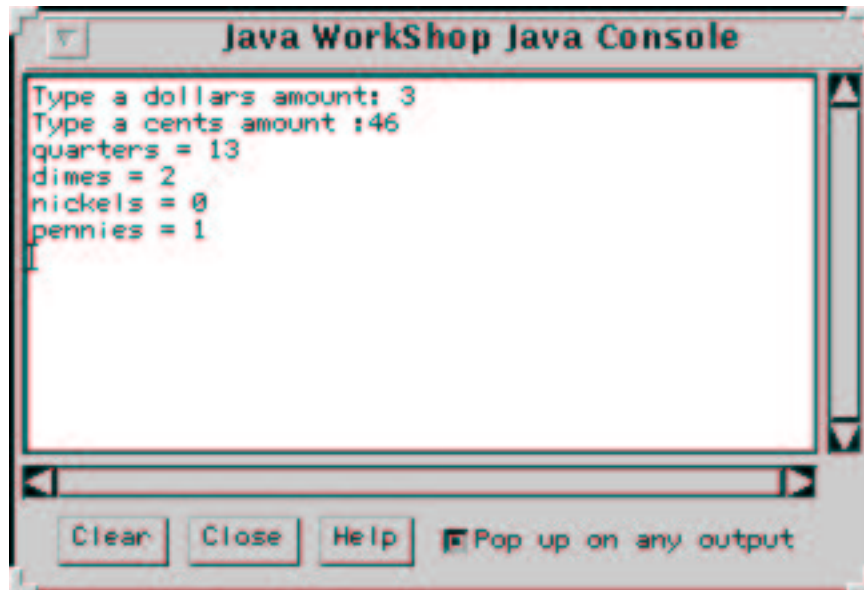


The user interacts with the application by typing at the keyboard, say, 3, and pressing the *newline* key; almost immediately, she sees another prompt:



Perhaps the user types 46 and *newline*. The answer then appears within the same

window:



In Java, `System.in` is the object that is connected to the computer's keyboard for interactive input. Unfortunately, `System.in` reads just one key press at a time and not an entire line. (For example, when the user types 4 then 6 then *newline*, this constitutes three key presses.) So, we must compose `System.in` with additional objects to read one full line of text:

- `System.in` is composed with an `InputStreamReader`:

```
new InputStreamReader(System.in)
```

- Next, the above object is composed with a `BufferedReader` object:

```
new BufferedReader(new InputStreamReader(System.in))
```

Like the `BufferedReader` objects seen earlier, this object has a method named `readLine` that can read one full line of input.

The input behavior we saw in the earlier demonstration is caused by these statements,

```
System.out.print("Type a dollars amount: ");
String dollars = keyboard.readLine();
System.out.print("Type a cents amount: ");
String cents = keyboard.readLine();
```

The first statement prints a partial line, which signals the user to type the dollars amount, and the second statement grabs what the user types. The third and fourth statements act similarly.

Here is the modified version of the change-making program:

```
import java.io.*;
/** MakeChange2 calculates change for a dollars-and-cents amount.
 * input: a dollars amount and a cents amount, both integers
 * output: the listing of the needed coins */
public class MakeChange2
{ public static void main(String[] args) throws IOException
  { BufferedReader keyboard
    = new BufferedReader(new InputStreamReader(System.in));

    System.out.print("Type a dollars amount: "); // print a partial line
    String dollars = keyboard.readLine(); // read one full line of typed input
    System.out.print("Type a cents amount: "); // do it again....
    String cents = keyboard.readLine();
    int money = (100 * new Integer(dollars).intValue())
                + new Integer(cents).intValue();
    // the remainder of the program stays the same:
    System.out.println( "quarters = " + (money/25) );
    money = money % 25;
    System.out.println( "dimes = " + (money/10) );
    money = money % 10;
    System.out.println( "nickels = " + (money/5) );
    money = money % 5;
    System.out.println( "pennies = " + money );
  }
}
```

11.4 Case Study: Payroll Processing

String tokenizers and disk file objects can help create input views that hide all details about file processing from the other components of an application. A standard example is payroll processing: Perhaps a company keeps its weekly payroll as a sequential file where each line contains an employee's name, hours worked, and hourly payrate, formatted as follows:

```
Fred Mertz|31|20.25
Lucy Ricardo|42|24.50
Ethel Mertz|18|18.00
!
```


Figure 11.4: input view interface

<code>class PayrollReader</code>	
Methods	
<code>getNextRecord(): boolean</code>	Attempt to read the next payroll record from the input file. Returns whether or not a properly formatted record was successfully read.
<code>nameOf(): String</code>	Return the name of the employee of the current payroll record.
<code>hoursOf(): int</code>	Return the hours worked of the employee of the current payroll record.
<code>payrateOf(): double</code>	Return the payrate of the employee of the current payroll record.
<code>close()</code>	Close the payroll file.

The sequence of lines is terminated by a special symbol, say, `!`. A payroll application reads the lines one by one, extracts the tokens from each line, converts them to names and numbers, and calculates paychecks. The effort spent on extracting names and numbers is best placed into a class that acts as the application's input view.

For the payroll application, the input view's responsibilities are to extract a name, hours worked, and payrate from each input record (line). Table 4 shows the interface for the payroll application's input view.

We can write an interface for the payroll application's output view as well, which will print the paychecks that are calculated. (Call the output view `class PayrollWriter`.) Based on these interfaces, we might write the controller for the payroll application that appears in Figure 5. The controller need not worry about the exact format of the records in the input payroll file, and it need not worry about how paychecks are nicely printed—these responsibilities are handled by the view objects.

Figure 6 presents one possible coding of the input view, `class PayrollReader`.

The hard work is performed within `getNextRecord`, which verifies the input file is nonempty before it reads the next record. The method also verifies that the record is not the end-of-file marker, `!`, and that the record indeed has a string name, an integer hours worked, and a double payrate. If all these conditions hold true, then the information from the record is saved, and `true` is returned. The method skips over badly formatted records and returns `false` when the end of the input is reached.

An exception handler recovers from exceptions that arise when `intValue` and `doubleValue` fail. Because we are forced to write a handler for these `RuntimeExceptions`, we use that handler to catch a `throw new RuntimeException(line)`, which announces an incorrect quantity of data on an input line. Since `getNextRecord` must also cope with a possible `IOException`, we attach two exception handlers to the same `try` clause.

Figure 11.5: controller for payroll application

```
/** Payroll prints a file of paychecks from an input payroll file. */
public class Payroll
{ public static void main(String[] args)
  { String in_name
    = JOptionPane.showInputDialog("Please type input payroll name:");
    String out_name
    = JOptionPane.showInputDialog("Please type output payroll name:");
    if ( in_name != null && out_name != null )
      { processPayroll(in_name, out_name); }
    System.out.println("finished");
  }

  private static void processPayroll(String in, String out)
  { PayrollReader reader = new PayrollReader(in);
    PayrollWriter writer = new PayrollWriter(out);
    while ( reader.getNextRecord() )
      { double pay = reader.hoursOf() * reader.payrateOf();
        writer.printCheck(reader.nameOf(), pay);
      }
    reader.close();
    writer.close();
  }
}
```

Figure 11.6: input-view class for payroll processing

```
import java.io.*;
import java.util.*;
/** PayrollReader reads records from a sequential file. The records have
 * the format, NAME|HOURS|PAYRATE. The file is terminated by a ! */
public class PayrollReader
{ private BufferedReader infile; // the address of the input file
  private String END_OF_FILE = "!";
  // the name, hours, and payrate of the most recently read record:
  private String name;
  private int hours;
  private double payrate;

  /** PayrollReader constructs the reader to read from file file_name */
  public PayrollReader(String file_name)
  { try { infile = new BufferedReader(new FileReader(file_name)); }
    catch (Exception e)
    { System.out.println("PayrollReader error: bad file name: "
      + file_name + " Aborting!");
      throw new RuntimeException(e.toString());
    }
  }

  public String nameOf() { return name; }

  public int hoursOf() { return hours; }

  public double payrateOf() { return payrate; }

  public void close()
  { try { infile.close(); }
    catch (IOException e)
    { System.out.println("PayrollReader warning: file close failed"); }
  }
  ...
}
```

Figure 11.6: input-view class for payroll processing (concl.)

```
/** getNextRecord attempts to read a new payroll record.
 * @return whether another record was read and is ready to process */
public boolean getNextRecord()
{ boolean result = false;
  name = null;
  hours = -1;
  payrate = -0.1;
  try { if ( infile.ready() )
        { String line = infile.readLine();
          StringTokenizer t = new StringTokenizer(line, "|");
          String s = t.nextToken().trim();
          if ( ! s.equals(END_OF_FILE) ) // finished?
            { if ( t.countTokens() == 2 ) // hours and payrate?
              { name = s;
                hours = new Integer
                  (t.nextToken().trim()).intValue();
                payrate = new Double
                  (t.nextToken().trim()).doubleValue();
                result = true;
              }
            }
          else { throw new RuntimeException(line); }
        }
      }
  catch (IOException e)
    { System.out.println("PayrollReader error: " + e.getMessage()); }
  catch (RuntimeException e)
    { System.out.println("PayrollReader error: bad record format: "
      + e.getMessage() + " Skipping record");
      result = getNextRecord(); // try again
    }
  return result;
}
```

These techniques with exceptions are examined in the sections that follow.

Exercise

Specify and write class `PayrollWriter` for the payroll example.

11.5 Exceptions and Exception Handlers

Even though it might be well written, a program can receive bad input data, for example, a sequence of letters when a number is required. When this has happened with earlier examples, the program stopped in the middle of its execution and announced that an error (*exception*) occurred. How can a program protect itself from this occurrence?

The Java language provides a construction, called an *exception handler*, that provides protection. Here is an example: A program must compute a division with an integer that is supplied as input:

```
import javax.swing.*;
/** DivideIntoTwelve reads an int and divides it into 12 */
public class DivideIntoTwelve
{ public static void main(String[] args)
  { int i = readAnInt();
    JOptionPane.showMessageDialog(null, "Answer is " + (12 / i));
    System.out.println("Finished.");
  }

  /** readAnInt reads an int and returns it */
  private static int readAnInt()
  { String s = JOptionPane.showInputDialog("Please type an int:");
    int num = new Integer(s).intValue();
    return num;
  }
}
```

Testing reveals that this program generates an exception when the user types a 0 for the integer input; we see in the command window this message:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at DivideIntoTwelve.main(DivideIntoTwelve.java:0)
```

A similar situation occurs if the user mistakenly types a non-integer, say, the text, one:

```
Type an int: one
Exception in thread "main" java.lang.NumberFormatException: one
```

```

at java.lang.Integer.parseInt(Integer.java:405)
at java.lang.Integer.<init>(Integer.java:540)
at DivideIntoTwelve.readAnIntFrom(DivideIntoTwelve.java:14)
at DivideIntoTwelve.main(DivideIntoTwelve.java:6)

```

The message documents the methods incompleting due to the exception.

Both errors are examples of *runtime exceptions*, so named because they occur when the program is “running.” The wordy messages produced from exceptions are confusing to the user, and an application can do better about reporting the exceptions.

We improve the program by inserting *exception handlers*. An exception handler is a kind of protective “cover” that surrounds a questionable statement and helps the program recover from exceptions generated by the statement.

To help the program recover from division-by-zero errors, we surround the statement with the division operation with an exception handler (note the keywords, `try` and `catch`):

```

public static void main(String[] args)
{ int i = readAnInt();
  try { JOptionPane.showMessageDialog(null, "Answer is " + (12 / i)); }
  catch(RuntimeException e)
    { JOptionPane.showMessageDialog(null,
      "Error in input: " + i + ". Restart program.");
    }
  System.out.println("Finished.");
}

```

Exception handlers are wordy—the `try` section brackets the questionable statement, and the `catch` section lists the statements that execute if an exception arises.

The statement in the `try` section is executed first; if `i` is nonzero (e.g., 2), the division proceeds without error, the dialog displaying the answer appears, and `Finished.` prints in the command window.

If `i` has value 0, however, then the division is stopped by a runtime exception—we say that *an exception is thrown*. At this moment, the statements in the `catch` section of the exception handler execute, displaying a dialog—the exception has been *handled*. When the exception handler completes its work, the program executes the statements that follow, as if nothing wrong had happened. (Here, `Finished.` prints.)

The phrase, `RuntimeException e`, documents the form of exception that can be *caught*—here, it is a runtime exception. (The purpose of the `e`, which is actually a formal parameter, is explained in Chapter 11.)

The general format of Java’s exception handler construction goes

```

try { STATEMENTS }
catch ( EXCEPTION )
  { HANDLER }

```

where `STATEMENTS` and `HANDLER` are sequences of zero or more statements. The `EXCEPTION` phrase is normally just `RuntimeException e`, but Chapter 11 documents other forms.

When executed, the `STATEMENTS` part begins. If no exceptions are thrown, then the `catch` part is ignored. But if an exception is thrown, and if it has the form named `EXCEPTION`, then the statements, `HANDLER`, are executed.

Exercises

1. Given the `main` method with its exception handler, how does the program behave when the user inputs the text, `zero`, as the input?
2. Insert an exception handler into the private method, `readAnInt`, so that a dialog appears with the phrase, `Error: Noninteger Input`, when a non-integer is supplied as input.
3. Insert an exception handler into Figure 5 so that if the user supplies a non-integer as input, the program responds with a dialog that complains about the invalid input and quits without printing an answer.

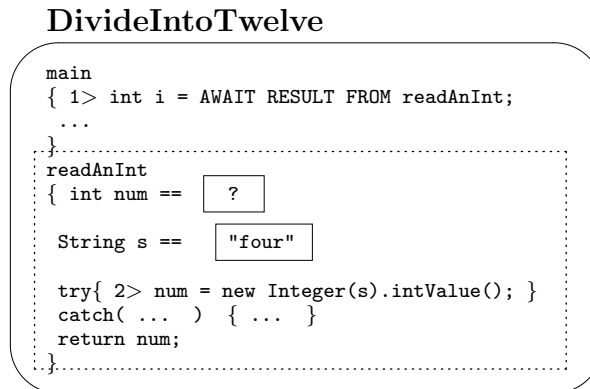
11.5.1 Restarting a Method with an Exception Handler

The example in the previous section used a private method, `readAnInt`, to read an integer from its user. What if the user types letters instead of an integer, e.g., `four` instead of `4`? To handle this situation, `readAnInt` can employ an exception handler that gives the application's user multiple tries at typing a proper integer—if the user types an improper input, the exception handler's `catch`-section restarts `readAnInt` by sending a message to the method *itself*:

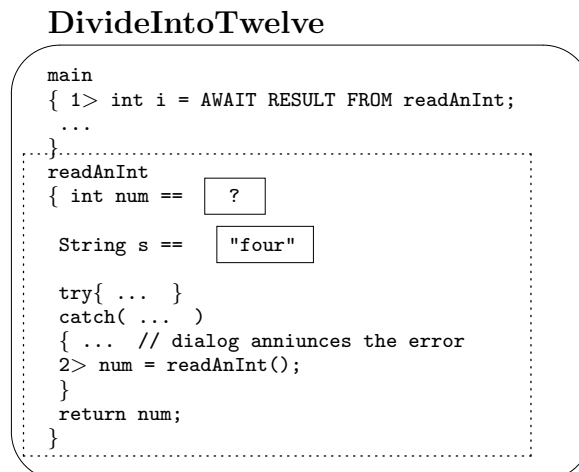
```
/** readAnInt reads an int and returns it */
private static int readAnInt()
{ int num;
  String s = JOptionPane.showInputDialog("Please type an int:");
  try { num = new Integer(s).intValue(); }
  catch(RuntimeException e)
    { JOptionPane.showMessageDialog(null,
      "Input " + s + " not an int; try again!");
      num = readAnInt(); // restart with a recursive invocation!
    }
  return num;
}
```

When a non-integer is supplied as input, a runtime exception is thrown at `new Integer(s)`, and the `catch` section goes to work: a dialog appears, then a message is sent to the method to restart itself, giving the user a new chance to type an integer. When a method restarts itself, it is called a *recursive invocation*.

Say that a user types `four` for the initial input; the execution configuration looks like this:



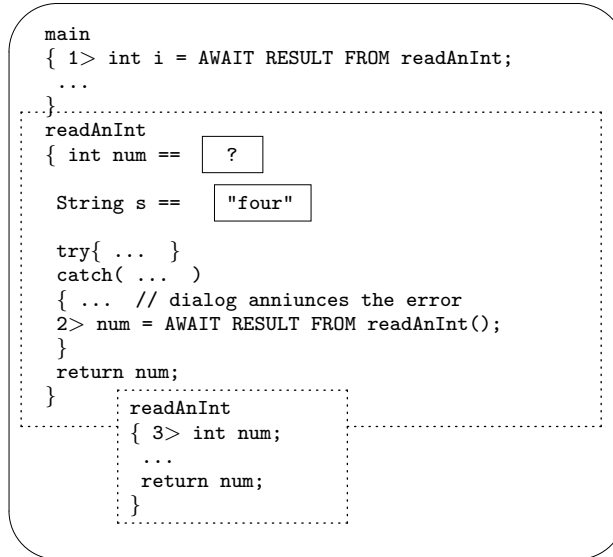
The attempted conversion of `four` to an integer throws a runtime exception, and execution moves into the `catch` section:



The restarted method asks the user for a fresh input. If the user types, say, `4`, the

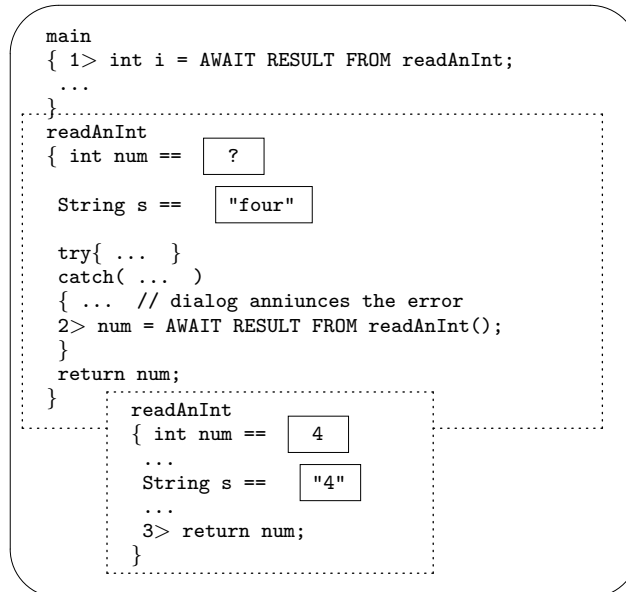
restarted method proceeds without exception:

DivideIntoTwelve



The restarted copy of `readAnInt` uses its own local variables to compute the integer, 4, that is returned to the original invocation of `readAnInt`:

DivideIntoTwelve



The exception handler in the initial invocation of `readAnInt` finishes its repair by assigning 4 to `num`, which is returned to the awaiting `main` method.

This example shows how an exception handler can repair a truly difficult situation so that a program can proceed to its conclusion. In this and subsequent chapters, we

Figure 11.7: interface for input views

This is a standardized interface for a class that reads interactive input.	
Methods (responsibilities)	
<code>readString(String prompt): String</code>	prompt the user for input with <code>prompt</code> , read a string, and return it as the result
<code>readInt(String prompt): int</code>	prompt the user for input with <code>prompt</code> , read an integer, and return it as the result
<code>readDouble(String prompt): double</code>	prompt the user for input with <code>prompt</code> , read a double, and return it as the result

will use recursive invocations to restart methods; Chapters 7 and 12 present other uses for recursive invocations.

11.5.2 Interactive Input with Exception Handlers

Whenever we required interactive input, we wrote a sequence of statements like the following:

```
String input = JOptionPane.showInputDialog("Type an integer:");
int value = new Integer(input).intValue();
```

And, if the user typed an incorrect input, the statements would throw an exception and terminate the program.

It would be better to have a standardized input-view that had skillful methods for fetching integers, doubles, and strings from the user and helping the user if an input was incorrectly typed. Once we had such an input view, we could use it as follows:

```
DialogReader reader = new DialogReader();
int value = reader.readInt("Type an integer:");
```

Better still, we could reuse this same input-view, `DialogReader`, for all the applications we write that require input.

Table 7 states the interface we shall use for the input-view class.

As an example, we can change the temperature conversion application seen earlier in the text to work with the interface in Table 7. This simplifies its main method to the following:

```
public static void main(String[] args)
{ DialogReader reader = new DialogReader(); // the new input-view
  int c = reader.readInt("Type an integer Celsius temperature:");
```

```

// this remains unchanged:
double f = celsiusIntoFahrenheit(c);
DecimalFormat formatter = new DecimalFormat("0.0");
System.out.println("For Celsius degrees " + c + ",");
System.out.println("Degrees Fahrenheit = " + formatter.format(f));
}

```

Next, we write a class that uses `JOptionPane` and exception handlers to match the interface. See Figure 8.

Method `readString` reads a line of text from the dialog, using the standard technique.

The `readInt` method uses `readString` to fetch the user’s input, which must be converted from a string, `s`, into an integer in the standard way:

```
answer = new Integer(s.trim()).intValue();
```

There is a small novelty: The string method, `trim()`, removes the leading and trailing blanks from the string before it is given to the `Integer` helper object for conversion. (Recall from Chapter 3 that string operations are written in the format, `S.op(args)`, for a string, `S`, operation, `op`, and arguments (if any), `args`.) For example, if `readString(prompt)` returned the string, " 123 ", then `" 123 ".trim()` computes to "123".

If the user typed a non-integer, then the attempt to convert it into an integer would throw a runtime exception. Fortunately, the exception handler handles the exception by restarting the `readInt` method, just like we saw in the previous section.

Method `readDouble` works similarly to `readInt`.

11.6 Exceptions Are Objects

We have seen how exceptions can terminate a program’s execution and how an exception handler can trap an exception so that execution can resume. Now, we learn that exceptions are objects that bind to the formal parameters of of exception handlers.

When an error (such as division by zero) occurs, the program must collect information about the error and deliver the information to a *handler* of the exception. The information collected about the error usually comes in two parts:

- a description of the nature of the error (e.g., “division by zero”);
- a description of where the error occurred, namely, the statement number, method, and class, and a history of all the method invocations that led to the statement.

These pieces of information are packaged into a newly constructed exception object, which becomes the “result” of the erroneous computation. This result object seeks its

Figure 11.8: class for input via a dialog

```
import javax.swing.*;
/** DialogReader accepts user input from a dialog */
public class DialogReader
{ /** Constructor DialogReader initializes the input view, a dialog */
  public DialogReader() { } // nothing to initialize

  /** readString reads a string from the input source.
   * @param prompt - the prompt that prompts the user for input
   * @return the string the user types in response */
  public String readString(String prompt)
  { return JOptionPane.showInputDialog(prompt); }

  /** readInt reads an integer from the input source
   * @param prompt - the prompt that prompts the user for input
   * @return the integer supplied in response */
  public int readInt(String prompt)
  { int answer = 0;
    String s = readString(prompt);
    try { answer = new Integer(s.trim()).intValue(); }
    catch (RuntimeException e)
      { JOptionPane.showMessageDialog(null,
        "DialogReader error: " + s + " not an int.");
        answer = readInt(prompt); // restart
      }
    return answer;
  }

  /** readDouble reads a double from the input source
   * @param prompt - the prompt that prompts the user for input
   * @return the double supplied in response */
  public double readDouble(String prompt)
  { double answer = 0;
    String s = readString(prompt);
    try { answer = new Double(s.trim()).doubleValue(); }
    catch (RuntimeException e)
      { JOptionPane.showMessageDialog(null,
        "DialogReader error: " + s + " not a double.");
        answer = readDouble(prompt); // restart
      }
    return answer;
  }
}
```

its handler, which is usually the IDE or JDK that started the program's execution. The handler displays the object's contents on the console window.

There are many classes from which exception objects can be built, e.g., `class ArrayIndexOutOfBoundsException` and `class NullPointerException` are two examples; both are found in the `java.lang` package. Figure 9 gives a partial listing of the exception hierarchy; no doubt, many of its class names are familiar.

Exception objects have several useful methods, presented in Table 10. The methods extract the information embedded within an exception. We use them momentarily.

As noted in Figure 9, the exceptions we have encountered fall into two groups: runtime exceptions and input-output exceptions. Runtime exceptions are subtle errors that are a fact of programming life. Since they can occur at almost every statement, the Java compiler assumes that every method in a program “throws `RuntimeException`.” In contrast, input-output exceptions are errors that might arise due to a serious problem with file usage. Such an error is uncommon, and the Java compiler requires that every method which might generate the error be labelled with `throws IOException`.

The usual outcome of an error is the creation of an exception object that finds its way to its default handler, the IDE or JDK. But in some cases, a programmer might prefer that the exception object find its way to a different handler, one that does not terminate execution—the programmer writes an exception handler to “catch” and “handle” exception objects.

The format of an exception handler goes

```
try { STATEMENTS }
catch (EXCEPTION_TYPE e)
    { HANDLER }
```

It defines a method, `catch`, that is “invoked” within `STATEMENTS` if an exception occurs there. If an exception object is created within `STATEMENTS`, and if the exception's type is a subtype of `EXCEPTION_TYPE`, then the exception object binds to formal parameter `e`, and `HANDLER` executes. (If the exception's type is not a subtype of `EXCEPTION_CLASS`, then the exception “escapes” to find its handler.) Once `HANDLER` finishes, execution continues with the statements that follow it. Within `HANDLER`, messages can be sent to the exception object, e.g., `e.printStackTrace()`.

If a programmer writes an exception handler, *she is obligated to make the handler repair the error to a degree that execution can safely proceed*. In this regard, programmer-written exception handlers can prove dangerous, because it is all too easy for a programmer to do too little to repair an error.

Here is a simple example. Perhaps we must write a method, `readAnIntFrom`, which reads a string from the input-view object and converts it into a string. An exception handler is added to cope with the possibility that the input string is not an integer:

```
/** readAnIntFrom reads an int from input-view, view, and returns it */
```

Figure 11.9: classes of exceptions

```
Object
|
+-Throwable: getMessage():String, toString():String, printStackTrace()
|
+-Exception
|
+-InterruptedException
|
+-RuntimeException
| |
| +-ArithmeticException
| |
| +-ArrayIndexOutOfBoundsException
| |
| +-ClassCastException
| |
| +-NullPointerException
| |
| +-NumberFormatException
| |
| +-StringIndexOutOfBoundsException
| |
| +-NoSuchElementException
|
+-IOException
|
+-FileNotFoundException
|
+-EOFException
|
+-InterruptedIOException
```

Figure 11.10: methods for exception objects

<code>getMessage(): String</code>	Return the message that describes the error that occurred
<code>toString(): String</code>	Return a string representation of the exception object; this is usually the class name of the exception and the message that describes the error.
<code>printStackTrace()</code>	Print on the console window the statement number, method, and class where the error occurred as well as all the method invocations that led to it.

```
private int readAnIntFrom(BufferedReader view) throws IOException
{ int num;
  System.out.print("Type an int: ");
  String s = view.readLine();
  try { num = new Integer(s).intValue(); }
  catch (RuntimeException e)
    { System.out.println("Non-integer error");
      num = -1;
    }
  return num;
}
```

If the string returned by `readLine` is nonnumeric, then a runtime exception arises at `new Integer(s).intValue()`. A `NumberFormatException` object is created and is promptly caught by the enclosing `catch`, which merely prints a message and lets the method return `-1` as its result. Since `-1` might be confused for a legitimate integer input, the exception handler has not done a good enough job. We improve it as seen in Figure 11.

In the Figure, the handler insists that the user type another string, and it invokes itself to restart the process. This ensures that the method concludes only when a proper integer has been typed. The message, `e.getMessage()`, extracts information about the nature of the error. Finally, the exception handler is refined to handle only `NumberFormatException`s, which ensures that the handler is used only for the form of error it is prepared to handle, namely, a format error for a string that is nonnumeric.

When an exception is thrown, it is not always obvious which `catch`-construct will handle it. If an exception is thrown within a method, and if the method does not catch the exception, then the exception seeks its handler by returning to the method that invoked the erroneous method and searching there.

The following artificial example explores this idea. Perhaps we have

```
ExceptionExample ex = new ExceptionExample();
```

Figure 11.11: improved exception handling for reading integers

```

/** readAnIntFrom reads an integer from the input source
 * @param view - the input-view object
 * @return the integer supplied in response. (And keep trying until the user
 * supplies a legal integer!) */
private int readAnIntFrom(BufferedReader view) throws IOException
{ int num;
  System.out.print("Type an int: ");
  String s = view.readLine();
  try { num = new Integer(s).intValue(); }
  catch(NumberFormatException e)
    { JOptionPane.showMessageDialog(null, "Error: " + e.getMessage()
                                     + " not an integer; try again.");
      num = readAnIntFrom(view); // restart
    }
  return num;
}

```

```
ex.f();
```

where class `ExceptionExample` is defined as

```

import java.io.*;
public class ExceptionExample
{ public ExceptionExample() { }

  public void f()
  { try { g(); }
    catch (RuntimeException e) { System.out.println("caught at f"); }
    System.out.println("f completes");
  }

  public void g()
  { try { PrintWriter outfile = new PrintWriter(new FileWriter("text.out"));
        try { outfile.println( h() ); }
        catch (NullPointerException e)
          {System.out.println("null pointer caught at g"); }
        }
    catch (IOException e)
      { System.out.println("io error caught at g"); }
    System.out.println("g completes");
  }
}

```



```

    private int h()
    { int[] r = new int[2];
      return r[3];
    }
}

```

The message to `f` causes a message to `g` which causes a message to `h`, where an array indexing error occurs. Within `h`, an `ArrayIndexOutOfBoundsException` object is created, and the plan of returning an integer from `h` is abandoned. Instead, the exception object seeks its handler. No handler appears in `h`, so the exception returns to the position where `h` was invoked. This places it at the statement, `outfile.println(h())` within `g`. The plan to perform the `outfile.println` is abandoned; the exception seeks its handler instead. It examines the data type, `NullPointerException`, of the nearest enclosing handler, but the type does not a subtype of `ArrayIndexOutOfBoundsException`, so the exception object searches further. The next enclosing handler has type `IOException`, but this is also unacceptable, so the exception returns from `g` to the position, `g()`, within method `f`.

Here, the exception finds a handler that can accommodate it: The exception's type is a subtype of `RuntimeException`, so the message, caught at `f`, is printed. The handler consumes the exception, and normal execution resumes at the statement, `System.out.println("f completes")`, which prints. Execution concludes at `ex.f()`, which is unaware of the problems that occurred.

The above example was meant to illustrate a technical point; it is not an example of good programming style. Indeed, exception handlers are best used to help a program recover from actions over which it has no influence, e.g., a user typing invalid input. Exception handlers are *not* so useful for detecting internal programming errors such as array indexing errors. As a rule,

it is better for a programmer to invest her time towards preventing errors with if-statements rather than recovering from errors with exception handlers.

For example, this statement sequence:

```

int i = reader.readInt("Please type an array index:");
if ( i > 0 && i < r.length )    // prevent an error
    { System.out.println(r[i]); }
else { System.out.println("invalid index: " + i); }

```

is *always* preferred to

```

int i = reader.readInt("Please type an array index:");
try { System.out.println(r[i]); }
catch (RuntimeException e)    // recover from an error
    { System.out.println(e.toString()); }

```

Exercises

1. It is usually disastrous for a program to terminate while reading or writing a file. Here is a program that reads integers from a file and squares them:

```
import java.io.*;
public class Squares
{ public static void main(String[] args) throws IOException
  { BufferedReader infile = new BufferedReader(new FileReader("ints.dat"));
    while ( infile.ready() )
      { int i = new Integer(infile.readLine().trim()).intValue();
        System.out.println(i + " squared is " + (i*i));
      }
    infile.close();
  }
}
```

Revise this class so that an invalid input line causes the program to display `Illegal input---skipping to next line` and causes the program to proceed to the next line of input.

2. Revise class `CopyFile` in Figure 3 so that an exception handler catches the `FileNotFoundException` that might arise when a user supplies an invalid name for the input file. The exception handler tells the user of the error and asks her to type a new file name.

11.6.1 Programmer-Generated Exceptions

In the rare case, it is possible for a programmer to escape from a disastrous situation by constructing an exception object:

```
try { ... // perhaps some complicated computation is undertaken
      if ( some_bad_condition_is_true )
        { throw new RuntimeException("terrible error"); }
      ... // otherwise, continue with the complicated computation
    }
catch (RuntimeException e)
  { ... e.getMessage() ... } // try to handle the bad condition
```

The statement, `throw new RuntimeException(S)`, constructs a runtime exception object and “throws” to its handler. String `S` is the message embedded within the exception and is the answer produced by a `getMessage`.

If one wants to be certain that the explicitly generated exception is not inadvertently confused with the other forms of runtime exceptions, one can create a new class for the new exception, e.g.,

```

public class ComplicatedException extends Exception
{ private int villain; // the integer that caused all the trouble

    public ComplicatedException(String error_message, int the_bad_number)
    { super(error_message);
      villain = the_bad_number;
    }

    public int getVillain()
    { return villain; }
}

```

Here, class `ComplicatedException` is a subclass of `Exception`, meaning that its objects can be “thrown.” Its constructor method has been enhanced to accept both an error message and an integer, which might be useful to its exception handler. The new class of exception might be used as follows:

```

try { int x = ... ;
    ... // some complicated computation is undertaken
    if ( some_bad_condition_is_true_about(x) )
        { throw new ComplicatedException("terrible error", x); }
    ... // otherwise, continue with the complicated computation
}
catch (ComplicatedException e)
{ ... e.getMessage() ...
  int i = e.getVillain();
  ...
}

```

The statement, `throw new ComplicatedException("terrible error", x)`, builds a `ComplicatedException` object and throws it to its handler, which sends messages to the exception to learn the nature of the error.

11.7 Summary

We summarize the main points of the chapter.

New Construction

- *exception handler*:

```

public int readInt(String prompt)
{ int answer = 0;
  String s = readString(prompt);
}

```

```

try { answer = new Integer(s.trim()).intValue(); }
catch (RuntimeException e)
    { JOptionPane.showMessageDialog(null,
        "DialogReader error: " + s + " not an int.");
      answer = readInt(prompt);    // restart
    }
return answer;
}

```

New Terminology

- *token*: a “word” within a string or a line of text
- *delimiter*: a character that is used to separate tokens in a string; examples are spaces and punctuation.
- *file*: a collection of symbols stored together on a disk
- *sequential file*: a file that is a sequence of symbols. A sequential file can be read (or written) only from front to back, like a book whose pages can be turned only forwards. Two forms of sequential file are
 1. a *character file*, which is a sequence of keyboard symbols, saved in the file as a sequence of bytes;
 2. a *binary file*, which is a sequence of ones-and-zeros.
- *random-access file*: a file that can be read or written in any order at all.
- *ASCII* and *Unicoding*: two coding formats for representing characters saved within a file
- *exception*: an error that occurs during program execution (e.g., executing `12/0` throws a division-by-zero exception). Normally, an exception halts a program unless an *exception handler* construction is present to *handle* (repair) the exception (see the exception handler in `readString` above, which handles an input-failure exception).

Points to Remember

Here are some notions regarding files to keep in mind:

- In Java, an output file of characters is most simply created from a `PrintWriter` object, e.g.,

```
PrintWriter outfile = new PrintWriter(new FileWriter("test.txt"));
```

One uses the `print` and `println` methods of the object to write text to the file. An input file is easily created from a `BufferedReader` object, e.g.,

```
BufferedReader infile = new BufferedReader(new FileReader("test.txt"));
```

One uses the `readLine` method to read one line of text from the file.

- Errors that might arise during file usage are called *input-output exceptions*, and a method in which an input-output exception might arise must be labelled with `throws IOException`.

Exceptions are in fact objects, and a programmer can write an *exception handler*, `try ... catch`, to “catch” and “handle” an exception.

New Classes for Future Use

- `class StringTokenizer`: used to disassemble a string into its constituent tokens; see Table 1.
- `class FileWriter`: used to construct a connection to an output sequential file; see Figure 2.
- `class PrintWriter`: used to write strings to an output sequential file; see Figure 2.
- `class FileReader`: used to construct a connection to an input sequential file; see Figure 3.
- `class BufferedReader`: used to read lines from an input output sequential file; see Figure 3.
- `System.in`: a preexisting object that can read symbols typed into the command window.
- `class InputStreamReader`: used to connect to `System.in`.

11.8 Programming Projects

1. Write a text formatting program.
 - (a) The first version, `PrettyPrint`, reads a nonempty sequential file, and produces a sequential file whose lines are “pretty printed,” that is, words are separated by exactly one blank and every input line has a length that is as close as possible, but does not exceed, 80 characters. (To simplify this Project, assume that no single word has a length that exceeds 80 characters.)

- (b) Next, modify `PrettyPrint` so that (i) hyphenated words and phrases connected by dashes can be broken across lines; (ii) two blanks, rather than one, are placed after a sentence-ending period.
- (c) Finally, modify `PrettyPrint` so that the output lines are right justified.
- Write a file merge program: It reads as its input two files of integers (one integer per line), where the integers in each file are sorted in nondescending order. The output is a new file, containing the contents of the two input files, of the integers sorted in nondescending order.
 - Write a program that reads a sequential file and calculates the average length of its words, its shortest nonnull word and its longest word. (Here, a “word” is a sequence of non-blank, non-tab, non-newline characters. One text line might contain multiple words.)
 - Write a program that reads a sequential file of integers and calculates the mean of the integers, the maximum and minimum integers, and the standard deviation of the integers, where mean and standard deviation are defined as

$$\text{Mean} = (n_0 + n_1 + \dots + n_m) / m$$

$$\text{StdDev} = \sqrt{-(\text{Mean}^2) + (n_0)^2 + (n_1)^2 + \dots + (n_m)^2}$$

where the input file contains the m integers, n_0, n_1, \dots, n_m . (Recall the standard deviation tells us that two-thirds of the integers in the file fall in the range $(\text{Mean} - \text{StdDev}) \dots (\text{Mean} + \text{StdDev})$.)

- Write an application that counts occurrences of words in a sequential file:
 - Make the program print the numbers of occurrences of words of lengths 1 to 9 and 10-or-more.
 - Next, modify the program to display its answers in a bar graph, e.g.,

Frequency of word occurrence:

```
-----
1: ****
2: ******
3: ******
4: ****
5: ******
6: **
...

```

- (c) Next, modify the program to count occurrences of distinct words only, that is, multiple appearances of the word, “the”, in the file count for only one occurrence of a three-letter word.
6. Write a program that reads a Java program, removes all `/* ... */` and `// ...` comments, and leaves the program otherwise unaltered.
7. Write a program that does simplistic text compression:
- (a) The input is a sequential text file, where sequences of n repeating characters, `aaa...a`, of length 4 to 9 are compressed into `\na`. For example, the line
- ```
Thisssss is aaaa testtt 22225.
```
- is compressed to
- ```
Thi\5s is \4a\6 testtt \425.
```
- (Assume that `\` does not appear in the input file.)
- (b) Modify the program so that repetitions of length 10 or greater are compressed as well.
- (c) Write the corresponding uncompression program.
8. Write a lexical analyzer for numerical and logical expressions: The program’s input is an arithmetic expression containing numerals, parentheses, and the operators `+`, `-`, `*`, `/`, `<`, `<=`, `&&`, and `!`. The output is a file of integers, where the input is translated as follows:
- a numeral, n , translates to two integers, 0 and the integer n .
 - a left parenthesis, `(`, translates to the integer, 1, and a right parenthesis, `)`, translates to the integer, 2.
 - the operators, `+`, `-`, `*`, `==`, `<`, `<=`, `&&`, and `!` translate to the integers 3 through 10, respectively.

For example, the input `(2+34 <=(-5+6)) && !(789==0)` translates to the output sequence 1 0 2 3 0 34 8 1 4 0 5 2 0 6 2 2 9 10 1 0 789 6 0 0 2 2.

9. Write a checkbook accountant program. The input is sequential file that begins with the account’s initial balance, followed by a sequence of deposit and withdrawal transactions. The output is a history of the account, showing each transaction and the current balance after the transaction.

Figure 11.12: copying a file character by character

```
import java.io.*;
/** CharCopy copies in.dat to out.dat character by character */
public class CharCopy
{ public static void main(String[] args) throws IOException
  { FileReader infile = new FileReader("in.dat");
    FileWriter outfile = new FileWriter("out.dat");
    while ( infile.ready() )
      { int c = infile.read(); // reads (the coding of) a character
        outfile.write(c);     // writes the character
      }
    infile.close();
    outfile.close();
  }
}
```

10. Write a program that reads a file of persons' names and addresses and a file that contains a "form letter" (e.g., a letter asking the recipient to subscribe to a magazine) and creates for its output a collection of output files, one output file for each person's name, where each output file contains the form letter personalized to the person's name.

11.9 Beyond the Basics

11.9.1 Character-by-Character File Processing

11.9.2 Binary Files and Files of Objects

11.9.3 A Taxonomy of File Classes

11.9.4 A GUI for File Selection

Here are some optional topics related to file processing.

11.9.1 Character-by-Character File Processing

It is tedious but nonetheless possible to process sequential files one character at a time. Figure 12 shows an application that copies the contents of a input file to an output file this way.

For reasons explained at the end of the chapter, it suffices to work with a `FileReader` object to read character input and a `FileWriter` object to write character output.

The statement, `int c = infile.read()` reads a character from the input file, but the character is returned as an integer. (This hides the differences between the one-byte ASCII codings of characters used by some computers and the two-byte Unicode codings used by others; both codings are accommodated by using an integer.) The `write` method copies the character to the output file.

If one wishes to work with the character that is read, it must be cast into type `char`, to inform the Java compiler that it is indeed a character:

```
int c = infile.read();
char d = (char)c;
String s = "abc" + d;
```

Conversely, if one tries to write an arbitrary integer, e.g., `outfile.write(-9999)`, the `write` method sends an “error character” to the output file.

11.9.2 Binary Files and Files of Objects

When you state,

```
double d = 1000007.5;
System.out.println(d);
```

A binary representation of one-million-seven-and-a-half is saved in a single cell, and this representation is fetched and converted into a sequence of nine characters for display on the console window. Conversions of numbers to characters are time consuming, and it is best to avoid them.

If a huge array of numbers must be saved in a file for later computation, it is best to use a binary file. Java provides `class ObjectOutputStream` and `class ObjectInputStream` for writing and reading binary files of numbers, booleans, and arbitrary objects. The values are copied in their binary, storage representations; there is no conversion to character format.

For example, to write the above number to a binary file, `data.out`, we create an output-file object and send a `writeDouble` message to it:

```
double d = 1000007.5;
ObjectOutputStream out = new ObjectOutputStream
                        (new FileOutputStream("data.out"));

out.writeDouble(d);
```

Similarly, to read a complete file, `data.in`, of doubles, we would state

```
ObjectInputStream in = new ObjectInputStream
                        (new FileInputStream("data.in"));
while ( in.available() > 0 ) // more data left to read?
    { double d = in.readDouble();
      ...
    }
in.close();
```

In a similar way, one can read and write integers and booleans.

As the name suggests, entire objects can be written to an `ObjectOutputStream` and read from an `ObjectInputStream`. Figure 12 displays a program that writes an integer and several objects to a binary file and then reads the file and restores the objects.

Objects are written to files with the `writeObject` method. Of course, objects often have fields that hold addresses of other objects, as is the case with `ComposedCell c` in the Figure. So, `writeObject(c)` saves not only `c`'s object but also the objects whose addresses are held by `c`'s fields, `x` and `y`. An integer and a `Cell` object are also written, to show that different types of values can be mixed in the binary file.

When the file `test.ob` is reopened as an input file, the file's contents must be read in the same order in which it was written. This means the `ComposedCell` object must be read first:

```
Object ob = in.readObject();  
ComposedCell m = (ComposedCell)ob; // cast ob into a ComposedCell
```

The `readObject` method returns a result of type `Object`; this object must be cast into type `ComposedCell`, and at this point, the object is restored. The integer and `Cell` object are read next.

Both classes `ComposedCell` and `Cell` are labelled `implements Serializable` to assert that objects created from the classes can be saved in sequential binary files. (The Java compiler verifies that a “serializable” class contains only fields that can be safely copied onto files.) Also, a method that reads a binary file of objects must state `throws ClassNotFoundException` to warn of the possibility that an object of unknown data type (class) might be read.

11.9.3 A Taxonomy of File Classes

The `java.io` package holds a variety of classes for input and output; Figure 13 lists those classes used in this text. Because of a history of alterations to the package, the intentions behind the class hierarchy are less clear than they might be. When studying the hierarchy, keep these ideas in mind:

- In Java, data are organized as sequences or *streams* of bytes. An *input stream* can be read; an *output stream* is written.
- A program rarely computes upon bytes directly; units like characters (or integers or objects) are read and written. Therefore, a stream must be composed with an object whose methods read/write bytes and collect them into characters, integers, objects, etc. This builds *character streams*, *object streams*, etc.
- Objects that manipulate character streams are called *readers* and *writers*. A reader or writer might be *buffered*, meaning that it collects characters into strings and reads/writes strings.

Figure 11.13: writing and reading objects from a file

```

import java.io.*;
/** ObjectFileTest writes and reads two objects and an integer */
public class ObjectFileTest
{ public static void main(String[] args)
    throws IOException, ClassNotFoundException
{ Cell b = new Cell(5);
  ComposedCell c = new ComposedCell(b);
  Cell d = new Cell(7);
  ObjectOutputStream out = new ObjectOutputStream
    (new FileOutputStream("test.ob"));

    out.writeObject(c);
    out.writeInt(49);
    out.writeObject(d);
    out.close();
  // now, read the objects and the integer and restore them:
  ObjectInputStream in = new ObjectInputStream
    (new FileInputStream("test.ob"));

  Object ob = in.readObject();
  ComposedCell m = (ComposedCell)ob; // cast ob into a ComposedCell
  System.out.println(m.answerOf());
  System.out.println(in.readInt());
  ob = in.readObject();
  Cell n = (Cell)ob; // cast ob into a Cell
  System.out.println(n.valueOf());
  in.close();
}
}

/** ComposedCell builds an object that holds within it two objects */
public class ComposedCell implements Serializable
{ private Cell x;
  private Cell y;

  public ComposedCell(Cell my_cell)
  { x = my_cell;
    y = new Cell(0);
  }

  public int answerOf()
  { return x.valueOf() + y.valueOf(); }
}

```

Figure 11.13: writing and reading objects from a file (concl.)

```

/** Cell holds an integer */
public class Cell implements Serializable
{ private int value;

    public Cell(int start) { value = start; }
    public int valueOf() { return value; }
}

```

Here are some examples of objects for input and output. To assemble an object that reads strings (lines of characters) from a file, `data.in`, we might say

```

BufferedReader infile = new BufferedReader
    (new InputStreamReader(new FileInputStream("data.in")));

```

First, `new FileInputStream("data.in")` locates `data.in` and creates an input stream whose `read` method reads one byte at a time. Next, `new InputStreamReader(...)` creates a character stream whose `read` method uses `FileInputStream`'s `read` to assemble one character at a time. Finally, `new BufferedReader(...)` creates a buffered reader whose `readLine` method uses `InputStreamReader`'s `read` to assemble one full line of characters and return the line as a string.

As a convenience, the composed object, `new InputStreamReader(new FileInputStream("data.in"))`, can be created in one step with `new FileReader("data.in")`:

```

BufferedReader infile = new BufferedReader(new FileReader("data.in"));

```

and this is the pattern used in this chapter to build most input-file objects. Also, the declaration,

```

FileReader in = new FileReader("data.in");

```

creates a reader from which we can read characters one at a time. (But the reader is not buffered.)

In a similar way,

```

PrintWriter outfile = new PrintWriter(new FileWriter("test.txt"));

```

creates a buffered writer that uses a writer, `new FileWriter("file")`. The latter abbreviates the character output stream, `new OutputStreamWriter(new FileOutputStream("test.txt"))`.

We create a file to which objects are written with this declaration:

```

ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("d.ob"));

```

This builds an object stream whose `writeObject` method uses `FileOutputStream`'s `write` to write objects as sequences of bytes to the file, `d.ob`.

The tables that follow provide details for the classes in Figure 13.

Figure 11.14: class hierarchy for input-output

```

Object
|
+-File: File(String), getPath():String, isFile():boolean, isDirectory():Boolean
|
+-InputStream (abstract): available():int, close()
| |
| +-FileInputStream: FileInputStream(String), FileInputStream(File)
| |                               read():int
| |
| +-FilterInputStream: read():int
| |
| +-ObjectInputStream: ObjectInputStream(InputStream), read():int,
|   readInt():int, readDouble():double, readBoolean():boolean,
|   readObject():Object
|
+-OutputStream (abstract): close()
| |
| +-FileOutputStream: FileOutputStream(String), FileOutputStream(File),
| |                               write(int)
| |
| +-FilterOutputStream: write(int)
| |
| +-ObjectOutputStream: ObjectOutputStream(OutputStream), write(int),
|   writeInt(int), writeDouble(double), writeBoolean(boolean),
|   writeObject(Object)
|
+-Reader (abstract)
| |
| +-InputStreamReader: InputStreamReader(InputStream), read():int,
| | |                               ready():boolean, close()
| | |
| | +-FileReader: FileReader(String)
| |
| +-BufferedReader: BufferedReader(Reader), read():int, ready():boolean,
|   readLine():String, close()
|
+-Writer (abstract)
|
+-OutputStreamWriter: OutputStreamWriter(OutputStream), write(int), close()
| |
| +-FileWriter: FileWriter(String)
|
+-PrintWriter: PrintWriter(Writer), PrintWriter(OutputStream), write(int),
  print(int), print(double), print(boolean), print(Object),
  println(int), println(double), println(boolean), println(Object),
  println(), close()

```

<code>class File</code>	contains the path name and basic properties of a file
Constructor	
<code>File(String path)</code>	Constructs the file object from the complete path name, <code>path</code> .
Methods	
<code>getPath(): String</code>	Return the complete directory path name of the file object.
<code>isFile(): boolean</code>	Return whether the file object is a file.
<code>isDirectory(): boolean</code>	Return whether the file object is a directory.

<code>abstract class InputStream</code>	a sequence (“stream”) of data items that can be read one byte at a time
Methods	
<code>available(): int</code>	Return how many bytes are available for reading.
<code>close()</code>	Close the file.

<code>class FileInputStream</code> <code>extends InputStream</code>	a stream that is a sequential disk file
Constructors	
<code>FileInputStream(String path)</code>	Open the disk file whose path name is <code>path</code> .
<code>FileInputStream(File f)</code>	Open the disk file named by <code>f</code> .
Method	
<code>read(): int</code>	Read the next unread byte from the file and return it; return -1 if the file is at the end.

<code>class FilterInputStream</code> <code>extends InputStream</code>	a basic input stream, e.g., from the keyboard.
Method	
<code>read(): int</code>	Return the next byte from the stream; return -1 if the end of the stream is reached.

<code>class ObjectInputStream</code> extends <code>InputStream</code>	an input stream from which binary values (primitive values and objects) can be read
Constructor	
<code>ObjectInputStream(InputStream stream)</code>	Uses <code>stream</code> to construct an object-input stream
Methods	
<code>read(): int</code>	Return the next byte from the stream; return -1 if the end of the stream is reached.
<code>readInt(): int</code>	Return a 32-bit integer assembled from bytes read from the stream; throws <code>EOFException</code> if end of the stream is reached.
<code>readDouble(): double,</code> <code>readBoolean(): boolean</code>	Similar to <code>readInt()</code> .
<code>readObject(): Object</code>	Return an object assembled from bytes read from the stream; throws an exception if a complete object cannot be assembled.

<code>abstract class OutputStream</code>	a sequence (“stream”) of data items that have been written
Method	
<code>close()</code>	Close the file.

<code>class FileOutputStream</code>	an output stream that is a sequential disk file
Constructors	
<code>FileOutputStream(String path)</code>	Open the disk file whose path name is <code>path</code> .
<code>FileOutputStream(File f)</code>	Open the disk file named by <code>f</code> .
Methods	
<code>write(int b)</code>	Write byte <code>b</code> to the file.
<code>close()</code>	Close the file.

<code>class FilterOutputStream</code> extends <code>OutputStream</code>	a basic output stream, e.g., to the console window.
Method	
<code>write(int b)</code>	Write byte <code>b</code> to the output stream.

<code>class OutputStream</code> <code>extends OutputStream</code>	an output stream to which binary values (primitive values and objects) can be written
Constructor	
<code>OutputStream(OutputStream stream)</code>	Uses <code>stream</code> to construct an object-output stream
Methods	
<code>write(int b)</code>	Write byte <code>b</code> to the stream.
<code>writeInt(int i)</code>	Writes <code>i</code> , a 32-bit integer, to the stream.
<code>writeDouble(double d),</code> <code>writeBoolean(boolean b)</code>	Similar to <code>writeInt()</code> .
<code>writeObject(Object ob)</code>	Writes <code>ob</code> to the output stream, provided that the class from which <code>ob</code> was constructed implements <code>Serializable</code> ; throws an exception, otherwise.

<code>abstract class Reader</code>	a sequence of characters that can be read
------------------------------------	---

<code>class InputStreamReader</code> <code>extends Reader</code>	a basic character stream
Constructor	
<code>InputStreamReader(InputStream stream)</code>	Construct a character stream that reads bytes from <code>stream</code> to assemble its characters.
Methods	
<code>read(): int</code>	Read the next unread character from the stream and return it as an integer; return -1 if the end of the stream is reached.
<code>ready(): boolean</code>	Return whether the stream contains more characters to be read.
<code>close()</code>	Close the stream.

<code>class FileReader</code> <code>extends Reader</code>	an <code>InputStreamReader</code> constructed from a <code>FileInputStream</code> .
Constructors	
<code>FileReader(String s),</code> <code>FileReader(File s),</code>	Constructs the input-stream reader—an abbreviation for <code>new InputStreamReader(new FileInputStream(s))</code> .

<code>class BufferedReader extends Reader</code>	a character stream from which complete lines of text can be read
Constructor	
<code>BufferedReader(Reader r)</code>	Construct a buffered reader that reads its characters from <code>r</code> .
Methods	
<code>read(): int</code>	Read the next unread character from the stream and return it as an integer; return -1 if the end of the stream is reached.
<code>readLine(): String</code>	Read a line of text from the stream, that is, a maximal sequence of characters terminated by a <i>newline</i> (<code>'\n'</code>) or <i>return-newline</i> (<code>'\r'</code> and <code>'\n'</code>). Return the line less its terminator characters; return <code>null</code> if the end of the stream is reached.
<code>ready(): boolean</code>	Return whether the stream contains more characters to be read.
<code>close()</code>	Close the stream.

<code>abstract class Writer</code>	a writable sequence of characters
------------------------------------	-----------------------------------

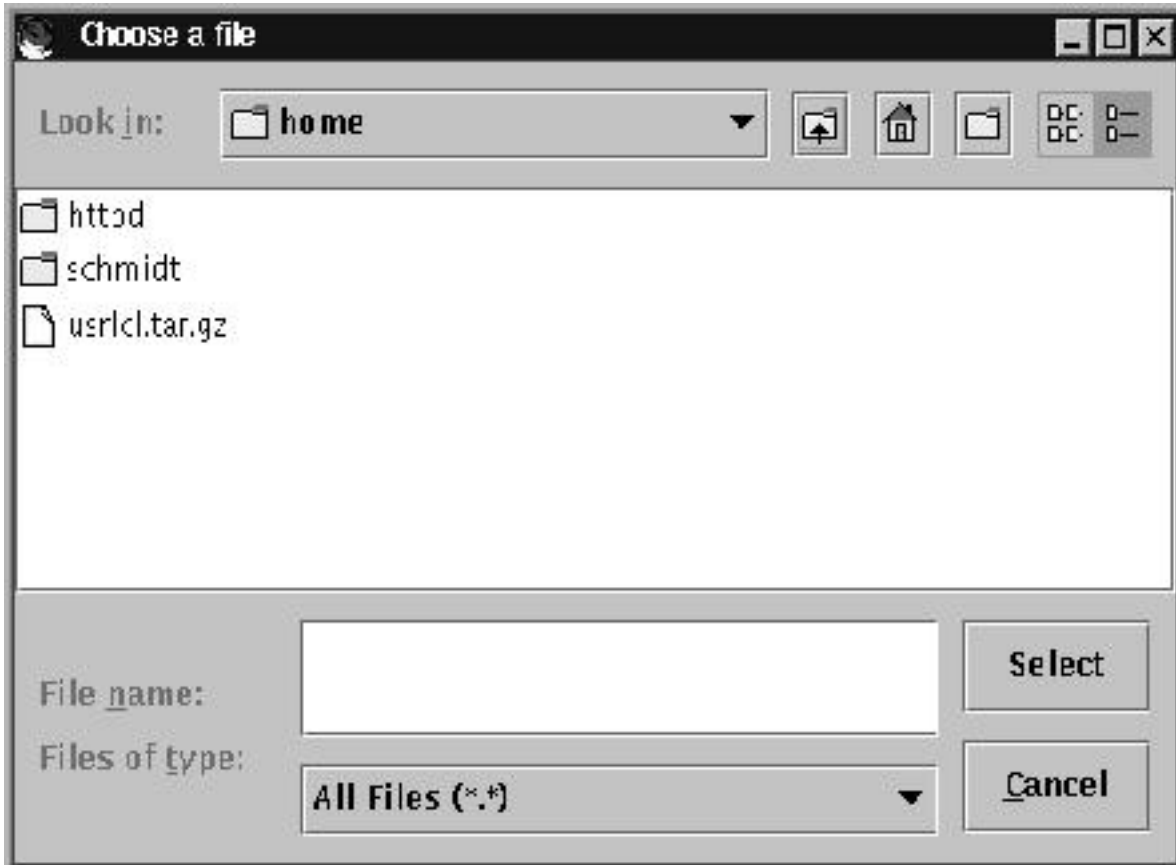
<code>class OutputStreamWriter extends Writer</code>	a basic character stream
Constructor	
<code>OutputStreamWriter(OutputStream stream)</code>	Construct a character stream to which characters can be written by means of <code>stream</code> .
Methods	
<code>write(int c)</code>	Write character <code>c</code> to the stream.
<code>close()</code>	Close the stream.

<code>class FileWriter extends Writer</code>	an <code>OutputStreamWriter</code> constructed from a <code>FileOutputStream</code>
Constructors	
<code>FileWriter(String s), FileWriter(File s),</code>	Constructs the output-stream writer—an abbreviation for <code>new OutputStreamWriter(new FileOutputStream(s))</code> .

<code>class PrintWriter extends Writer</code>	a buffered character stream to which primitive values and objects can be written (in their string depictions)
Constructors	
<code>PrintWriter(OutputStream s), PrintWriter(Writer s)</code>	Construct a print writer that sends its values as character sequences to <code>s</code> .
Methods	
<code>write(int c)</code>	Write character <code>c</code> to the output stream.
<code>print(int d), print(double d), print(boolean d), print(Object d)</code>	Write the character representation of <code>d</code> to the output stream. (When <code>d</code> is an object, <code>d</code> 's <code>toString</code> method is used to get its representation.)
<code>println(int d), println(double d), println(boolean d), println(Object d) println()</code>	Like <code>print</code> but appends line termination character(s).
<code>close()</code>	Close the stream.

11.9.4 A GUI for File Selection

The `javax.swing` package contains a component from which you can create a dialog that displays a folder (directory); the object, called a *file chooser*, looks like this:



The file chooser lets a user click on a folder to open it and click on a file name to select a file. When the user presses the **Select** button, the path name of the selected file is remembered, and the program that created the file chooser can retrieve the path name.

A simple way to create and show a file chooser is

```
JFileChooser chooser = new JFileChooser();
chooser.setDialogTitle("Choose a file");
int result = chooser.showDialog(null, "Select");
```

The resulting dialog displays the contents of the folder (directory) of the user's "home" folder; the dialog's title is set by `setDialogTitle`. After the user selects a file and presses the button labelled **Select**, the selected file's path name is retrieved and used as follows:

```
if ( result == JFileChooser.APPROVE_OPTION )
```

Figure 11.15: methods for JFileChooser

<code>class JFileChooser</code>	
Constructors	
<code>JFileChooser()</code>	Constructs a file chooser which points to the contents of the user's home folder (directory)
<code>JFileChooser(String path_name)</code>	Constructs a file chooser which points to the contents of the folder <code>path_name</code> .
<code>JFileChooser(File path_name)</code>	Constructs a file chooser which points to the contents of the folder <code>path_name</code> .
Methods	
<code>getSelectedFile(): File</code>	Return the file selected in the file chooser's view
<code>setDialogTitle(String title)</code>	Set the title of the file chooser's dialog to <code>title</code>
<code>showDialog(Component owner, String text_for_approve_button)</code>	Display the file chooser as a dialog, using <code>owner</code> as the dialog's owner component. The approval button is labelled with <code>text_for_approve_button</code> .

```

        { File f = chooser.getSelectedFile();
          String path_name = f.getPath();
          ... // use path_name to create a file object
        }
else { System.out.println
      ("Error: Select button was not pushed or pushed with no file selected");
    }

```

The file chooser returns an integer code which equals `JFileChooser.APPROVE_OPTION` if the user selected a file (or typed a file name into the dialog's text area) and terminated the dialog by pushing the `Select` button (or pressing *Enter* in the text area). The `getSelectedFile` method returns a `File` object from which one extracts the path name with `getPath`.

Figure 14 shows some of the methods for class `JFileChooser`.

Exercise

Extend the text editor in Figure 33, Chapter 10 so that it has a menu item that reads a file into the editor's text area and one that saves the contents of the text area into a file.