Exercises

15.4 *(File Matching)* Self-Review Exercise 15.2 asked you to write a series of single statements. Actually, these statements form the core of an important type of file-processing program—namely, a file-matching program. In commercial data processing, it's common to have several files in each application system. In an accounts receivable system, for example, there's generally a master file containing detailed information about each customer, such as the customer's name, address, telephone number, outstanding balance, credit limit, discount terms, contract arrangements and possibly a condensed history of recent purchases and cash payments.

As transactions occur (i.e., sales are made and payments arrive in the mail), information about them is entered into a file. At the end of each business period (a month for some companies, a week for others, and a day in some cases), the file of transactions (called `"trans.txt"`) is applied to the master file (called `"oldmast.txt"`) to update each account's purchase and payment record. During an update, the master file is rewritten as the file `"newmast.txt"`, which is then used at the end of the next business period to begin the updating process again.

File-matching programs must deal with certain problems that do not arise in single-file programs. For example, a match does not always occur. If a customer on the master file has not made any purchases or cash payments in the current business period, no record for this customer will appear on the transaction file. Similarly, a customer who did make some purchases or cash payments could have just moved to this community, and if so, the company may not have had a chance to create a master record for this customer.

Write a complete file-matching accounts receivable program. Use the account number on each file as the record key for matching purposes. Assume that each file is a sequential text file with records stored in increasing account-number order.

a) Define class `TransactionRecord`. Objects of this class contain an account number and amount for the transaction. Provide methods to modify and retrieve these values.
b) Modify class `Account` in Fig. 15.9 to include method `combine`, which takes a `TransactionRecord` object and combines the balance of the `Account` object and the amount value of the `TransactionRecord` object.
c) Write a program to create data for testing the program. Use the sample account data in Figs. 15.14 and 15.15. Run the program to create the files `trans.txt` and `oldmast.txt` to be used by your file-matching program.

Master file account number	Name	Balance
100	Alan Jones	348.17
300	Mary Smith	27.19
500	Sam Sharp	0.00
700	Suzy Green	-14.22

Fig. 15.14 | Sample data for master file.

Transaction file account number	Transaction amount
100	27.14
300	62.11
400	100.56
900	82.17

Fig. 15.15 | Sample data for transaction file.

d) Create class `FileMatch` to perform the file-matching functionality. The class should contain methods that read `oldmast.txt` and `trans.txt`. When a match occurs (i.e., records with the same account number appear in both the master file and the transaction

file), add the dollar amount in the transaction record to the current balance in the master record, and write the `"newmast.txt"` record. (Assume that purchases are indicated by positive amounts in the transaction file and payments by negative amounts.) When there's a master record for a particular account, but no corresponding transaction record, merely write the master record to `"newmast.txt"`. When there's a transaction record, but no corresponding master record, print to a log file the message `"Unmatched transaction record for account number..."` (fill in the account number from the transaction record). The log file should be a text file named `"log.txt"`.

15.6 ***(File Matching with Object Serialization)*** Recreate your solution for Exercise 15.5 using object serialization. Use the statements from Exercise 15.3 as your basis for this program. You may want to create applications to read the data stored in the `.ser` files—the code in Section 15.5.2 can be modified for this purpose.

15.7 ***(Telephone-Number Word Generator)*** Standard telephone keypads contain the digits zero through nine. The numbers two through nine each have three letters associated with them (Fig. 15.17). Many people find it difficult to memorize phone numbers, so they use the correspondence between digits and letters to develop seven-letter words that correspond to their phone numbers. For example, a person whose telephone number is 686-2377 might use the correspondence indicated in Fig. 15.17 to develop the seven-letter word "NUMBERS." Every seven-letter word corresponds to exactly one seven-digit telephone number. A restaurant wishing to increase its takeout business could surely do so with the number 825-3688 (i.e., "TAKEOUT").

Digit	Letters	Digit	Letters	Digit	Letters
2	A B C	5	J K L	8	T U V
3	D E F	6	M N O	9	W X Y
4	G H I	7	P R S		

Fig. 15.17 | Telephone keypad digits and letters.

Every seven-letter phone number corresponds to many different seven-letter words, but most of these words represent unrecognizable juxtapositions of letters. It's possible, however, that the owner of a barbershop would be pleased to know that the shop's telephone number, 424-7288, corresponds to "HAIRCUT." A veterinarian with the phone number 738-2273 would be pleased to know that the number corresponds to the letters "PETCARE." An automotive dealership would be pleased to know that the dealership number, 639-2277, corresponds to "NEWCARS."

Write a program that, given a seven-digit number, uses a `PrintStream` object to write to a file every possible seven-letter word combination corresponding to that number. There are 2,187 (3^7) such combinations. Avoid phone numbers with the digits 0 and 1.

15.8 ***(Student Poll)*** Figure 7.8 contains an array of survey responses that's hard coded into the program. Suppose we wish to process survey results that are stored in a file. This exercise requires two separate programs. First, create an application that prompts the user for survey responses and outputs each response to a file. Use a `Formatter` to create a file called `numbers.txt`. Each integer should be written using method `format`. Then modify the program in Fig. 7.8 to read the survey responses from `numbers.txt`. The responses should be read from the file by using a `Scanner`. Use method `nextInt` to input one integer at a time from the file. The program should continue to read responses until it reaches the end of the file. The results should be output to the text file `"output.txt"`.

15.9 ***(Adding Object Serialization to the `MyShape` Drawing Application)*** Modify Exercise 12.17 to allow the user to save a drawing into a file or load a prior drawing from a file using object serialization. Add buttons **Load** (to read objects from a file) and **Save** (to write objects to a file). Use an `ObjectOutputStream` to write to the file and an `ObjectInputStream` to read from the file. Write the array of `MyShape` objects using method `writeObject` (class `ObjectOutputStream`), and read the array using method `readObject` (`ObjectInputStream`). The object-serialization mechanism can read or write entire arrays—it's not necessary to manipulate each element of the array of `MyShape` objects individually. It's simply required that all the shapes be `Serializable`. For both the **Load** and **Save** buttons, use a `JFileChooser` to allow the user to select the file in which the shapes will be stored or from which they'll be read. When the user first runs the program, no shapes should be displayed on the screen. The user can display shapes by opening a previously saved file or by drawing new shapes. Once there are shapes on the screen, users can save them to a file using the **Save** button.