15.4 Sequential Text Files

Next, we create and manipulate *sequential files* in which records are stored in order by the record-key field. We begin with *text files*, enabling the reader to quickly create and edit human-readable files. We discuss creating, writing data to, reading data from and updating sequential text files. We also include a credit-inquiry program that retrieves data from a file. The programs in Sections 15.4.1–15.4.3 are all in the chapter's `TextFileApps` directory so that they can manipulate the same text file, which is also stored in that directory.

15.4.1 Creating a Sequential Text File

Java imposes no structure on a file—notions such as records do not exist as part of the Java language. Therefore, you must structure files to meet the requirements of your applications. In the example that follows, we see how to impose a *keyed* record structure on a file.

The program in this section creates a simple sequential file that might be used in an accounts receivable system to keep track of the amounts owed to a company by its credit clients. For each client, the program obtains from the user an account number and the client's name and balance (i.e., the amount the

client owes the company for goods and services received). Each client's data constitutes a "record" for that client. This application uses the account number as the *record key*—the file's records will be created and maintained in account-number order. The program assumes that the user enters the records in account-number order. In a comprehensive accounts receivable system (based on sequential files), a *sorting* capability would be provided so that the user could enter the records in *any* order. The records would then be sorted and written to the file.

Class `CreateTextFile`

Class `CreateTextFile` (Fig. 15.3) uses a `Formatter` to output formatted `String`s, using the same formatting capabilities as method `System.out.printf`. A `Formatter` object can output to various locations, such as to a command window or to a file, as we do in this example. The `Formatter` object is instantiated in the `try`-with-resources statement (line 13; introduced in Section 11.12)—recall that `try`-with-resources will close its resource(s) when the `try` block terminates successfully or due to an exception. The constructor we use here takes one argument—a `String` containing the name of the file, including its path. If a path is not specified, as is the case here, the JVM assumes that the file is in the directory from which the program was executed. For text files, we use the `.txt` file extension. If the file does *not* exist, it will be *created*. If an *existing* file is opened, its contents are **truncated**—all the data in the file is *discarded*. If

no exception occurs, the file is open for writing and the resulting `Formatter` object can be used to write data to the file.

```
1  // Fig. 15.3: CreateTextFile.java
2  // Writing data to a sequential text file with c
3  import java.io.FileNotFoundException;
4  import java.lang.SecurityException;
5  import java.util.Formatter;
6  import java.util.FormatterClosedException;
7  import java.util.NoSuchElementException;
8  import java.util.Scanner;
9
10 public class CreateTextFile {
11    public static void main(String[] args) {
12       // open clients.txt, output data to the fi
13       try (Formatter output = new Formatter("cli
14          Scanner input = new Scanner(System.in);
15          System.out.printf("%s%n%s%n? ",
16             "Enter account number, first name, l
17             "Enter end-of-file indicator to end
18
19          while (input.hasNext()) { // loop until
20             try {
21                // output new record to file; ass
22                output.format("%d %s %s %.2f%n",
23                   input.next(), input.next(), in
24             }
25             catch (NoSuchElementException elemen
26                System.err.println("Invalid input
27                input.nextLine(); // discard inpu
28             }
29
30             System.out.print("? ");
31          }
32       }
33       catch (SecurityException | FileNotFoundExc
34          FormatterClosedException e) {
```

```
            e.printStackTrace();
         }
      }
   }
```

```
Enter account number, first name, last name and balan
    Enter end-of-file indicator to end input.
? 100 Bob Blue 24.98
? 200 Steve Green -345.67
? 300 Pam White 0.00
? 400 Sam Red -42.16
? 500 Sue Yellow 224.62
? ^Z
```

Fig. 15.3

Writing data to a sequential text file with class `Formatter`.

Lines 33–36 are a multi-`catch` which handles several exceptions:

- the `SecurityException` that occurs if the user does not have permission to write data to the file opened in line 13
- the `FileNotFoundException` that occurs if the file does not exist and a new file cannot be created, or if there's an error *opening* the file in line 13, and
- the `FormatterClosedException` that occurs if the `Formatter` object is closed when you attempt to use it in lines 22–23 to write into a file.

For these exceptions, we display a stack trace, then the

program terminates.

Writing Data to the File

Lines 15–17 prompt the user to enter the various fields for each record or the end-of-file key sequence when data entry is complete. Figure 15.4 lists the key combinations for entering end-of-file for various computer systems' command windows—some IDEs do not support these for console-based input (so you might have to execute the programs from command windows). Line 19 uses `Scanner` method `hasNext` to determine whether the end-of-file key combination has been entered. The loop executes until `hasNext` encounters end-of-file.

Operating system	Key combination
macOS and Linux	*<Enter> <Ctrl> d*
Windows	*<Ctrl> z*

Fig. 15.4

End-of-file key combinations.

Lines 22–23 use a `Scanner` to read data from the user, then output the data as a record using the `Formatter`. Each `Scanner` input method throws a

`NoSuchElementException` (handled in lines 25–28) if the data is in the wrong format (e.g., a `String` when an `int` is expected) or if there's no more data to input.

If no exception occurs, the record's information is output using method `format`, which can perform identical formatting to `System.out.printf`. Method `format` writes a formatted `String` to the `Formatter` object's output destination—the file `clients.txt`. The format string `"%d %s %s %.2f%n"` indicates that the current record will be stored as an integer (the account number) followed by a `String` (the first name), another `String` (the last name) and a floating-point value (the balance). Each piece of information is separated from the next by a space, and the double value (the balance) is output with two digits to the right of the decimal point (as indicated by the `.2` in `%.2f`). The data in the text file can be viewed with a text editor or retrieved later by a program designed to read the file (Section 15.4.2). [*Note:* You can also output data to a text file using class `java.io.PrintWriter`, which provides `format` and `printf` methods for outputting formatted data.]

When the user enters the end-of-file key combination, the try-with-resources statement closes the `Formatter` and the underlying output file by calling `Formatter` method `close`. If a program does not explicitly call method `close`, the operating system normally will close the file when program execution terminates—this is an example of operating-system "housekeeping." However, you should always explicitly close a file when it's no longer needed.

Sample Output

The sample data for this application is shown in Fig. 15.5. In the sample output, the user enters information for five accounts, then enters end-of-file to signal that data entry is complete. The sample output does not show how the data records actually appear in the file. In the next section, to verify that the file was created successfully, we present a program that reads the file and prints its contents. Because this is a text file, you can also verify the information simply by opening the file in a text editor.

Sample data			
100	Bob	Blue	24.98
200	Steve	Green	-345.67
300	Pam	White	0.00
400	Sam	Red	-42.16
500	Sue	Yellow	224.62

Fig. 15.5

Sample data for the program in Fig. 15.3.

15.4.2 Reading Data from a

Sequential Text File

Data is stored in files so that it may be retrieved for processing when needed. Section 15.4.1 demonstrated how to create a file for sequential access. This section shows how to read data sequentially from a text file. We demonstrate how class `Scanner` can be used to input data from a file rather than the keyboard. The application (Fig. 15.6) reads records from the file `"clients.txt"` created by the application of Section 15.4.1 and displays the record's contents. Line 14 creates the `Scanner` that will be used to retrieve input from the file.

```
1   // Fig. 15.6: ReadTextFile.java
2   // This program reads a text file and displays e
3   import java.io.IOException;
4   import java.lang.IllegalStateException;
5   import java.nio.file.Files;
6   import java.nio.file.Path;
7   import java.nio.file.Paths;
8   import java.util.NoSuchElementException;
9   import java.util.Scanner;
10
11  public class ReadTextFile {
12     public static void main(String[] args) {
13        // open clients.txt, read its contents and
14        try(Scanner input = new Scanner(Paths.get(
15           System.out.printf("%-10s%-12s%-12s%10s%
16              "First Name", "Last Name", "Balance"
17
18           // read record from file
19           while (input.hasNext()) { // while ther
20              // display record contents
21              System.out.printf("%-10d%-12s%-12s%1
22                 input.next(), input.next(), input
23           }
```

```
                    }
        catch (IOException | NoSuchElementExceptio
            IllegalStateException e) {
               e.printStackTrace();
                    }
                 }
                }
```

Account	First Name	Last Name	Balance
100	Bob	Blue	24.98
200	Steve	Green	-345.67
300	Pam	White	0.00
400	Sam	Red	-42.16
500	Sue	Yellow	224.62

Fig. 15.6

Sequential file reading using a `Scanner`.

The `try`-with-resources statement opens the file for reading by instantiating a `Scanner` object (line 14). We pass a `Path` object to the constructor, which specifies that the `Scanner` object will read from the file `"clients.txt"` located in the directory from which the application executes. If the file cannot be found, an `IOException` occurs. The exception is handled in lines 25–28.

Lines 15–16 display headers for the columns in the

application's output. Lines 19–23 read and display the file's content until the *end-of-file marker* is reached—in which case, method `hasNext` will return `false` at line 19. Lines 21–22 use `Scanner` methods `nextInt`, `next` and `nextDouble` to input an `int` (the account number), two `String`s (the first and last names) and a `double` value (the balance). Each record is one line of data in the file. If the information in the file is not properly formed (e.g., there's a last name where there should be a balance), a `NoSuchElementException` occurs when the record is input. If the `Scanner` was closed before the data was input, an `IllegalStateException` occurs. These exceptions are handled in lines 25–28. Note in the format string in line 21 that the account number, first name and last name are left aligned, while the balance is right aligned and output with two digits of precision. Each iteration of the loop inputs one line of text from the text file, which represents one record. When the loop terminates and line 24 is reached, the `try`-with-resources statement implicitly calls the `Scanner`'s `close` method to close `Scanner` and the file.

15.4.3 Case Study: A Credit-Inquiry Program

To retrieve data sequentially from a file, programs start from the beginning of the file and read *all* the data consecutively until the desired information is found. It might be necessary to process the file sequentially several times (from the beginning of the file) during the execution of a program. Class `Scanner`

does *not* allow repositioning to the beginning of the file. If it's necessary to read the file again, the program must *close* the file and *reopen* it.

The program in Figs. 15.7–15.8 allows a credit manager to obtain lists of customers with *zero balances* (i.e., customers who do not owe any money), customers with *credit balances* (i.e., customers to whom the company owes money) and customers with *debit balances* (i.e., customers who owe the company money for goods and services received). A credit balance is a *negative* amount, a debit balance a *positive* amount.

`MenuOption` `enum`

We begin by creating an `enum` type (Fig. 15.7) to define the different menu options the credit manager will have—this is required if you need to provide specific values for the `enum` constants. The options and their values are listed in lines 5–8.

```
   1 // Fig. 15.7: MenuOption.java
2  // enum type for the credit-inquiry program's op
   3 public enum MenuOption {
4    // declare contents of enum type
   5    ZERO_BALANCE(1),
   6    CREDIT_BALANCE(2),
   7    DEBIT_BALANCE(3),
   8    END(4);
   9
10   private final int value; // current menu opti
   11
```

```
    // constructor
    private MenuOption(int value) {this.value = v
    }
```

Fig. 15.7

`enum` type for the credit-inquiry program's menu options.

`CreditInquiry` Class

Figure 15.8 contains the functionality for the credit-inquiry program. The program displays a text menu and allows the credit manager to enter one of three options to obtain credit information:

- Option 1 (`ZERO_BALANCE`) displays accounts with zero balances.
- Option 2 (`CREDIT_BALANCE`) displays accounts with credit (negative) balances.
- Option 3 (`DEBIT_BALANCE`) displays accounts with debit (positive) balances.
- Option 4 (`END`) terminates program execution.

```
// Fig. 15.8: CreditInquiry.java
// This program reads a file sequentially and di
// contents based on the type of account the use
// (credit balance, debit balance or zero balanc
import java.io.IOException;
import java.lang.IllegalStateException;
import java.nio.file.Paths;
```

```
8   import java.util.NoSuchElementException;
9   import java.util.Scanner;
10
11  public class CreditInquiry {
12     private final static MenuOption[] choices = M
13
14     public static void main(String[] args) {
15        Scanner input = new Scanner(System.in);
16
17        // get user's request (e.g., zero, credit
18        MenuOption accountType = getRequest(input)
19
20        while (accountType != MenuOption.END) {
21           switch (accountType) {
22              case ZERO_BALANCE:
23                 System.out.printf("%nAccounts wit
24                    break;
25              case CREDIT_BALANCE:
26                 System.out.printf("%nAccounts wit
27                    break;
28              case DEBIT_BALANCE:
29                 System.out.printf("%nAccounts wit
30                    break;
31           }
32
33           readRecords(accountType);
34           accountType = getRequest(input); // get
35        }
36     }
37
38     // obtain request from user
39     private static MenuOption getRequest(Scanner
40        int request = 4;
41
42        // display request options
43        System.out.printf("%nEnter request%n%s%n%s
44           " 1 - List accounts with zero balances"
45           " 2 - List accounts with credit balance
46           " 3 - List accounts with debit balances
47           " 4 - Terminate program");
```

```
                    48
                49          try {
     50           do { // input user request
   51               System.out.printf("%n? ");
   52               request = input.nextInt();
53           } while ((request < 1) || (request > 4)
                    54         }
55         catch (NoSuchElementException noSuchElemen
56           System.err.println("Invalid input. Term
                    57         }
                         58
59         return choices[request - 1]; // return enu
                   60      }
                         61
62      // read records from file and display only re
63      private static void readRecords(MenuOption ac
   64         // open file and process contents
65         try (Scanner input = new Scanner(Paths.get
66           while (input.hasNext()) { // more data
67              int accountNumber = input.nextInt();
68              String firstName = input.next();
 69              String lastName = input.next();
70              double balance = input.nextDouble();
                         71
72              // if proper account type, display r
73              if (shouldDisplay(accountType, balan
74                 System.out.printf("%-10d%-12s%-12
75                    firstName, lastName, balance);
               76              }
            77               else {
78                 input.nextLine(); // discard the
               79              }
                80           }
                  81         }
82         catch (NoSuchElementException | IllegalSta
          83            IOException e) {
84           System.err.println("Error processing fi
           85            System.exit(1);
                   86         }
                     87      }
```

```
                        88
89      // use record type to determine if record sho
  90      private static boolean shouldDisplay(
 91         MenuOption option, double balance) {
92         if ((option == MenuOption.CREDIT_BALANCE)
 93            return true;
               94         }
95         else if ((option == MenuOption.DEBIT_BALAN
 96            return true;
               97         }
98         else if ((option == MenuOption.ZERO_BALANC
 99            return true;
              100         }
                  101
 102          return false;
              103      }
                104  }
```

```
                Enter request
        1 - List accounts with zero balances
      2 - List accounts with credit balances
       3 - List accounts with debit balances
             4 - Terminate program

                    ? 1
```

```
            Accounts with zero balances:
```

300	Pam	White	0.00

```
                Enter request
        1 - List accounts with zero balances
      2 - List accounts with credit balances
```

```
        3 - List accounts with debit balances
        4 - Terminate program

             ? 2
```

```
        Accounts with credit balances:
```

200	Steve	Green	-345.67
400	Sam	Red	-42.16

```
             Enter request
        1 - List accounts with zero balances
        2 - List accounts with credit balances
        3 - List accounts with debit balances
        4 - Terminate program

             ? 3
```

```
        Accounts with debit balances:
```

100	Bob	Blue	24.98
500	Sue	Yellow	224.62

```
             Enter request
        1 - List accounts with zero balances
        2 - List accounts with credit balances
        3 - List accounts with debit balances
        4 - Terminate program
```

```
? 4
```

Fig. 15.8

Credit-inquiry program.

The record information is collected by reading through the file and determining if each record satisfies the criteria for the selected account type. Line 18 in `main` calls method `getRequest` (lines 39–60) to display the menu options, translates the number typed by the user into a `MenuOption` and stores the result in `MenuOption` variable `accountType`. Lines 20–35 loop until the user specifies that the program should terminate. Lines 21–31 display a header for the current set of records to be output to the screen. Line 33 calls method `readRecords` (lines 63–87), which loops through the file and reads every record.

Method `readRecords` uses a `try`-with-resources statement to create a `Scanner` that opens the file for reading (line 65). The file will be opened for reading with a new `Scanner` object each time `readRecords` is called, so that we can again read from the beginning of the file. Lines 67–70 read a record. Line 73 calls method `shouldDisplay` (lines 90–103) to determine whether the current record satisfies the account type requested. If `should-Display` returns `true`, the program displays the account information. When the *end-*

of-file marker is reached, the loop terminates and the `try`-with-resources statement closes the `Scanner` and the file. Once all the records have been read, control returns to `main` and `get-Request` is called again (line 34) to retrieve the user's next menu option.

15.4.4 Updating Sequential Files

The data in many sequential files cannot be modified without the risk of destroying other data in the file. For example, if the name "`White`" needs to be changed to "`Worthington`", the old name cannot simply be overwritten, because the new name requires more space. The record for `White` was written to the file as

```
300 Pam White 0.00
```

If the record is rewritten beginning at the same location in the file using the new name, the record will be

```
300 Pam Worthington 0.00
```

The new record is larger (has more characters) than the original record. "`Worthington`" would overwrite the "`0.00`" in the current record, and the characters beyond the

second “`o`” in “`Worthington`” will overwrite the beginning of the next sequential record in the file. The problem here is that fields in a text file—and hence records—can vary in size. For example, 7, 14, –117, 2074 and 27383 are all `int`s stored in the same number of bytes (4) internally, but they’re different-sized fields when written to a file as text. Therefore, records in a sequential file are not usually updated in place—instead, the entire file is rewritten. To make the preceding name change, the records before `300 Pam White 0.00` would be copied to a new file, the new record (which can be of a different size than the one it replaces) would be written and the records after `300 Pam White 0.00` would be copied to the new file. Rewriting the entire file is uneconomical to update just one record, but reasonable if a substantial number of records need to be updated.