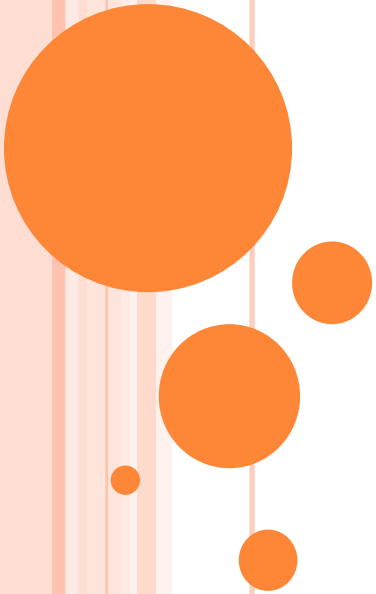
Chapter 17:
Files, Streams and Object Serialization
Part 3:4



Reading Data from a Sequential-Access Text File

- The application in Figs. 17.9 and 17.10 reads records from the file `"clients.txt"` created by the application of Section 17.4.1 and displays the record contents.

```
1  // Fig. 17.9: ReadTextFile.java
2  // This program reads a text file and displays each record.
3  import java.io.File;
4  import java.io.FileNotFoundException;
5  import java.lang.IllegalStateException;
6  import java.util.NoSuchElementException;
7  import java.util.Scanner;
8
9  import com.deitel.ch17.AccountRecord;
10
11 public class ReadTextFile
12 {
13    private Scanner input;
14
```

Fig. 17.9 | Sequential file reading using a `Scanner`. (Part 1 of 4.)

```
15      // enable user to open file
16      public void openFile()
17      {
18         try
19         {
20            input = new Scanner( new File( "clients.txt" ) );
21         } // end try
22         catch ( FileNotFoundException fileNotFoundException )
23         {
24            System.err.println( "Error opening file." );
25            System.exit( 1 );
26         } // end catch
27      } // end method openFile
28
```

Fig. 17.9 | Sequential file reading using a `Scanner`. (Part 2 of 4.)

```
29      // read record from file
30      public void readRecords()
31      {
32         // object to be written to screen
33         AccountRecord record = new AccountRecord();
34
35         System.out.printf( "%-10s%-12s%-12s%10s\n", "Account",
36            "First Name", "Last Name", "Balance" );
37
38         try // read records from file using Scanner object
39         {
40            while ( input.hasNext() )
41            {
42               record.setAccount( input.nextInt() ); // read account number
43               record.setFirstName( input.next() ); // read first name
44               record.setLastName( input.next() ); // read last name
45               record.setBalance( input.nextDouble() ); // read balance
46
47               // display record contents
48               System.out.printf( "%-10d%-12s%-12s%10.2f\n",
49                  record.getAccount(), record.getFirstName(),
50                  record.getLastName(), record.getBalance() );
51            } // end while
52         } // end try
```

Fig. 17.9 | Sequential file reading using a `Scanner`. (Part 3 of 4.)

```
53          catch ( NoSuchElementException elementException )
54          {
55             System.err.println( "File improperly formed." );
56             input.close();
57             System.exit( 1 );
58          } // end catch
59          catch ( IllegalStateException stateException )
60          {
61             System.err.println( "Error reading from file." );
62             System.exit( 1 );
63          } // end catch
64       } // end method readRecords
65
66       // close file and terminate application
67       public void closeFile()
68       {
69          if ( input != null )
70             input.close(); // close file
71       } // end method closeFile
72    } // end class ReadTextFile
```

Fig. 17.9 | Sequential file reading using a `Scanner`. (Part 4 of 4.)

```
1  // Fig. 17.10: ReadTextFileTest.java
2  // Testing the ReadTextFile class.
3
4  public class ReadTextFileTest
5  {
6     public static void main( String[] args )
7     {
8        ReadTextFile application = new ReadTextFile();
9
10       application.openFile();
11       application.readRecords();
12       application.closeFile();
13    } // end main
14 } // end class ReadTextFileTest
```

```
Account   First Name  Last Name      Balance
100       Bob         Jones            24.98
200       Steve       Doe            -345.67
300       Pam         White            0.00
400       Sam         Stone          -42.16
500       Sue         Rich           224.62
```

Fig. 17.10 | Testing the `ReadTextFile` class.

Reading Data from a Sequential-Access Text File

- If a `Scanner` is closed before data is input, an **`IllegalStateException` occurs.**

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.
  - The program must close the file and reopen it.

```
1  // Fig. 17.11: MenuOption.java
2  // Enumeration for the credit-inquiry program's options.
3
4  public enum MenuOption
5  {
6     // declare contents of enum type
7     ZERO_BALANCE( 1 ),
8     CREDIT_BALANCE( 2 ),
9     DEBIT_BALANCE( 3 ),
10    END( 4 );
11
```

Fig. 17.11 | Enumeration for the credit-inquiry program's menu options. (Part 1 of 2.)

```
12      private final int value; // current menu option
13
14      // constructor
15      MenuOption( int valueOption )
16      {
17         value = valueOption;
18      } // end MenuOptions enum constructor
19
20      // return the value of a constant
21      public int getValue()
22      {
23         return value;
24      } // end method getValue
25   } // end enum MenuOption
```

Fig. 17.11 | Enumeration for the credit-inquiry program's menu options. (Part 2 of 2.)

```
1  // Fig. 17.12: CreditInquiry.java
2  // This program reads a file sequentially and displays the
3  // contents based on the type of account the user requests
4  // (credit balance, debit balance or zero balance).
5  import java.io.File;
6  import java.io.FileNotFoundException;
7  import java.lang.IllegalStateException;
8  import java.util.NoSuchElementException;
9  import java.util.Scanner;
10
11 import com.deitel.ch17.AccountRecord;
12
13 public class CreditInquiry
14 {
15    private MenuOption accountType;
16    private Scanner input;
17    private final static MenuOption[] choices = { MenuOption.ZERO_BALANCE,
18       MenuOption.CREDIT_BALANCE, MenuOption.DEBIT_BALANCE,
19       MenuOption.END };
20
21    // read records from file and display only records of appropriate type
22    private void readRecords()
23    {
```

Fig. 17.12 | Credit-inquiry program. (Part 1 of 6.)

```
24          // object to store data that will be written to file
25          AccountRecord record = new AccountRecord();
26
27          try // read records
28          {
29             // open file to read from beginning
30             input = new Scanner( new File( "clients.txt" ) );
31
32             while ( input.hasNext() ) // input the values from the file
33             {
34                record.setAccount( input.nextInt() ); // read account number
35                record.setFirstName( input.next() ); // read first name
36                record.setLastName( input.next() ); // read last name
37                record.setBalance( input.nextDouble() ); // read balance
38
39                // if proper acount type, display record
40                if ( shouldDisplay( record.getBalance() ) )
41                   System.out.printf( "%-10d%-12s%-12s%10.2f\n",
42                      record.getAccount(), record.getFirstName(),
43                      record.getLastName(), record.getBalance() );
44             } // end while
45          } // end try
```

Fig. 17.12 | Credit-inquiry program. (Part 2 of 6.)

```
46          catch ( NoSuchElementException elementException )
47          {
48             System.err.println( "File improperly formed." );
49             input.close();
50             System.exit( 1 );
51          } // end catch
52          catch ( IllegalStateException stateException )
53          {
54             System.err.println( "Error reading from file." );
55             System.exit( 1 );
56          } // end catch
57          catch ( FileNotFoundException fileNotFoundException )
58          {
59             System.err.println( "File cannot be found." );
60             System.exit( 1 );
61          } // end catch
62          finally
63          {
64             if ( input != null )
65                input.close(); // close the Scanner and the file
66          } // end finally
67       } // end method readRecords
68
```

Fig. 17.12 | Credit-inquiry program. (Part 3 of 6.)

```
69      // use record type to determine if record should be displayed
70      private boolean shouldDisplay( double balance )
71      {
72         if ( ( accountType == MenuOption.CREDIT_BALANCE )
73            && ( balance < 0 ) )
74            return true;
75
76         else if ( ( accountType == MenuOption.DEBIT_BALANCE )
77            && ( balance > 0 ) )
78            return true;
79
80         else if ( ( accountType == MenuOption.ZERO_BALANCE )
81            && ( balance == 0 ) )
82            return true;
83
84         return false;
85      } // end method shouldDisplay
86
87      // obtain request from user
88      private MenuOption getRequest()
89      {
90         Scanner textIn = new Scanner( System.in );
91         int request = 1;
92
```

Fig. 17.12 | Credit-inquiry program. (Part 4 of 6.)

```
93          // display request options
94          System.out.printf( "\n%s\n%s\n%s\n%s\n%s\n",
95             "Enter request", " 1 - List accounts with zero balances",
96             " 2 - List accounts with credit balances",
97             " 3 - List accounts with debit balances", " 4 - End of run" );
98
99          try // attempt to input menu choice
100         {
101            do // input user request
102            {
103               System.out.print( "\n? " );
104               request = textIn.nextInt();
105            } while ( ( request < 1 ) || ( request > 4 ) );
106         } // end try
107         catch ( NoSuchElementException elementException )
108         {
109            System.err.println( "Invalid input." );
110            System.exit( 1 );
111         } // end catch
112
113         return choices[ request - 1 ]; // return enum value for option
114      } // end method getRequest
115
```

Fig. 17.12 | Credit-inquiry program. (Part 5 of 6.)

```
116     public void processRequests()
117     {
118        // get user's request (e.g., zero, credit or debit balance)
119        accountType = getRequest();
120
121        while ( accountType != MenuOption.END )
122        {
123           switch ( accountType )
124           {
125              case ZERO_BALANCE:
126                 System.out.println( "\nAccounts with zero balances:\n" );
127                 break;
128              case CREDIT_BALANCE:
129                 System.out.println( "\nAccounts with credit balances:\n" );
130                 break;
131              case DEBIT_BALANCE:
132                 System.out.println( "\nAccounts with debit balances:\n" );
133                 break;
134           } // end switch
135
136           readRecords();
137           accountType = getRequest();
138        } // end while
139     } // end method processRequests
140  } // end class CreditInquiry
```

Fig. 17.12 | Credit-inquiry program. (Part 6 of 6.)

```
1  // Fig. 17.13: CreditInquiryTest.java
2  // This program tests class CreditInquiry.
3
4  public class CreditInquiryTest
5  {
6     public static void main( String[] args )
7     {
8        CreditInquiry application = new CreditInquiry();
9        application.processRequests();
10    } // end main
11 } // end class CreditInquiryTest
```

Fig. 17.13 | Testing the `CreditInquiry` class.

```
Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run

? 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 - End of run

? 2

Accounts with credit balances:
200       Steve       Doe             -345.67
400       Sam         Stone            -42.16
```

Fig. 17.14 | Sample output of the credit-inquiry program in Fig. 17.13. (Part 1 of 2.)

```
Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run

? 3

Accounts with debit balances:
100       Bob         Jones            24.98
500       Sue         Rich            224.62

? 4
```

Fig. 17.14 | Sample output of the credit-inquiry program in Fig. 17.13. (Part 2 of 2.)

Updating Sequential-Access Files

- The data in many sequential files cannot be modified without the risk of destroying other data in the file.
- If the name "`White`" needed to be changed to "`Worthington`," the old name cannot simply be overwritten, because the new name requires more space.
- Fields in a text file—and hence records—can vary in size.
- Records in a sequential-access file are not usually updated in place. Instead, the entire file is usually rewritten.
- Rewriting the entire file is uneconomical to update just one record, but reasonable if a substantial number of records need to be updated.

Thank you