14

Files and Streams

I can only assume that a "Do Not File" document is filed in a "Do Not File" file.

— Senator Frank Church

Senate Intelligence Subcommittee Hearing, 1975

Consciousness does not appear to itself chopped up in bits. A "river" or a "stream" are the metaphors by which it is most naturally described.

— William James

I read part of it all the way through.

— Samuel Goldwyn

A great memory does not make a philosopher, any more than a dictionary can be called grammar.

— John Henry, Cardinal Newman

OBJECTIVES

In this chapter you will learn:

- To create, read, write and update files.
- To use class `File` to retrieve information about files and directories.
- The Java input/output stream class hierarchy.
- The differences between text files and binary files.
- Sequential-access and random-access file processing.
- To use classes `Scanner` and `Formatter` to process text files.
- To use the `FileInputStream` and `FileOutputStream` classes.
- To use a `JFileChooser` dialog.
- To use the `ObjectInputStream` and `ObjectOutputStream` classes

Outline

Outline

14.1 Introduction

- **Storage of data in variables and arrays is temporary**
- **Files used for long-term retention of large amounts of data, even after the programs that created the data terminate**
- **Persistent data – exists beyond the duration of program execution**
- **Files stored on secondary storage devices**
- **Stream – ordered data that is read from or written to a file**

14.2 Data Hierarchy

- **Computers process all data items as combinations of zeros and ones**
- **Bit – smallest data item on a computer, can have values 0 or 1**
- **Byte – 8 bits**
- **Characters – larger data item**
  - **Consists of decimal digits, letters and special symbols**
  - **Character set – set of all characters used to write programs and represent data items**
    - **Unicode – characters composed of two bytes**
    - **ASCII**

14.2 Data Hierarchy

- **Fields – a group of characters or bytes that conveys meaning**
- **Record – a group of related fields**
- **File – a group of related records**
- **Data items processed by computers form a data hierarchy that becomes larger and more complex from bits to files**
- **Record key – identifies a record as belonging to a particular person or entity – used for easy retrieval of specific records**
- **Sequential file – file in which records are stored in order by the record-key field**
- **Database – a group of related files**
- **Database Management System – a collection of programs designed to create and manage databases**

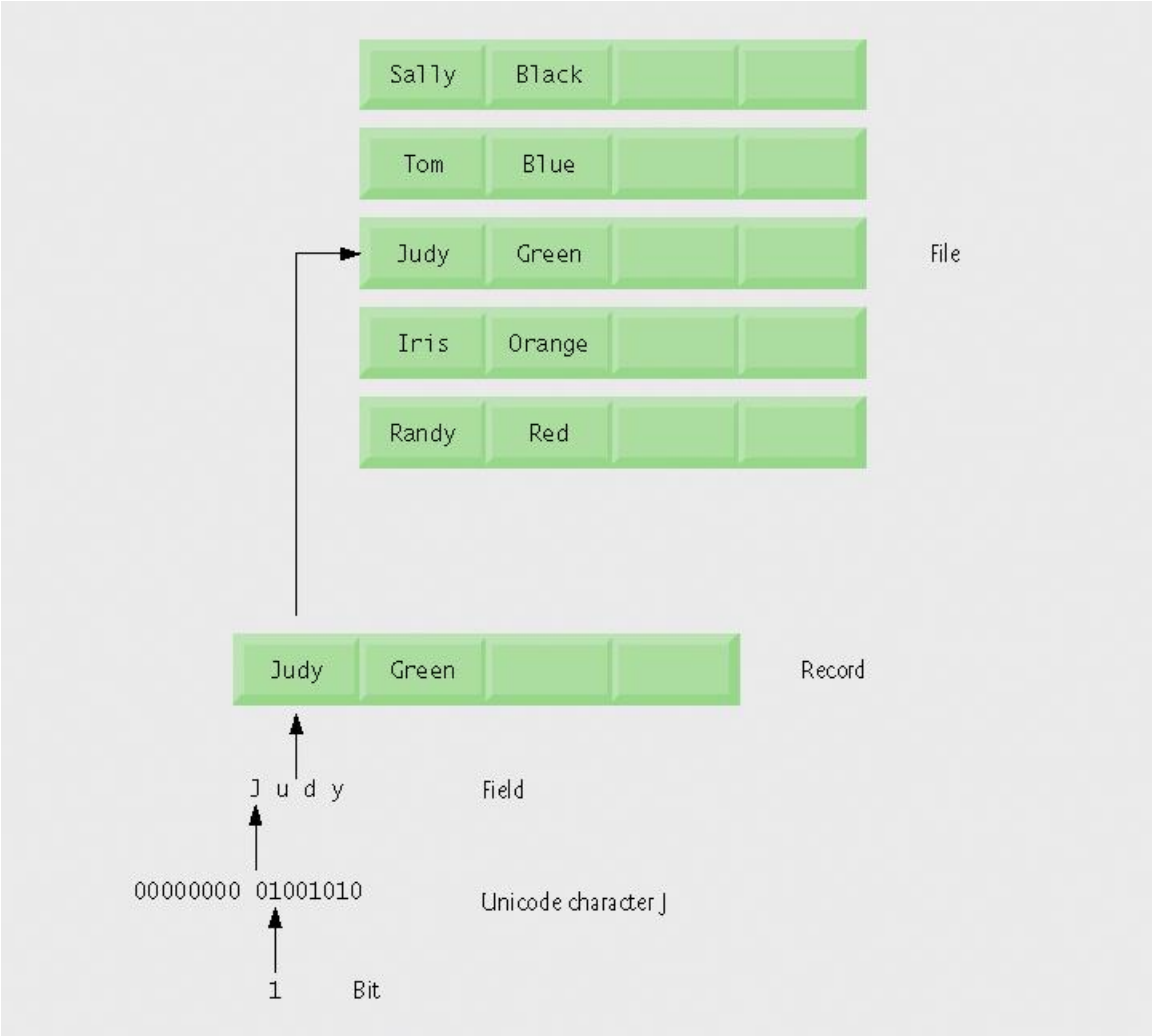

Fig. 14.1 | Data hierarchy.

14.3 Files and Streams

- **Java views each files as a sequential stream of bytes**
- **Operating system provides mechanism to determine end of file**
  - **End-of-file marker**
  - **Count of total bytes in file**
  - **Java program processing a stream of bytes receives an indication from the operating system when program reaches end of stream**

14.3 Files and Streams

- File streams
  - Byte-based streams – stores data in binary format
    - Binary files – created from byte-based streams, read by a program that converts data to human-readable format
  - Character-based streams – stores data as a sequence of characters
    - Text files – created from character-based streams, can be read by text editors
- Java opens file by creating an object and associating a stream with it
- Standard streams – each stream can be redirected
  - `System.in` – standard input stream object, can be redirected with method `setIn`
  - `System.out` – standard output stream object, can be redirected with method `setOut`
  - `System.err` – standard error stream object, can be redirected with method `setErr`

14.3 Files and Streams

- `java.io` classes
  - `FileInputStream` and `FileOutputStream` – byte-based I/O
  - `FileReader` and `FileWriter` – character-based I/O
  - `ObjectInputStream` and `ObjectOutputStream` – used for input and output of objects or variables of primitive data types
  - `File` – useful for obtaining information about files and directories
- Classes `Scanner` and `Formatter`
  - `Scanner` – can be used to easily read data from a file
  - `Formatter` – can be used to easily write data to a file

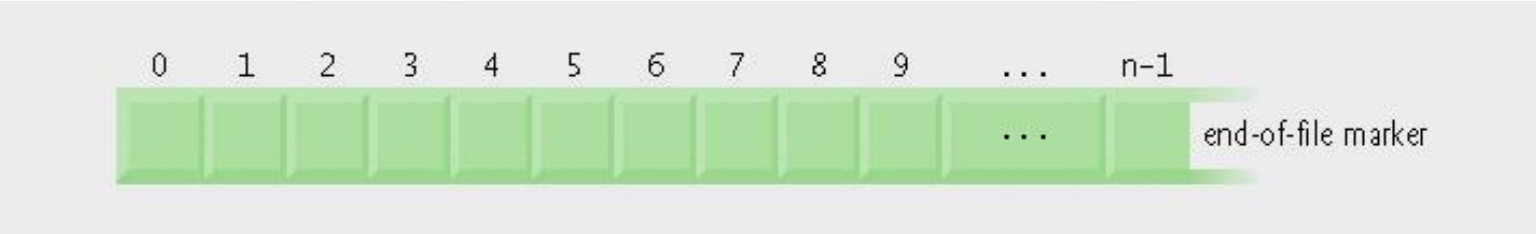

Fig. 14.2 | Java's view of a file of *n* bytes.

14.4 Class `File`

- Class `File` useful for retrieving information about files and directories from disk
- Objects of class `File` do not open files or provide any file-processing capabilities

Creating `File` Objects

- Class `File` provides four constructors:
  1. Takes `String` specifying name and path (location of file on disk)
  2. Takes two `String`s, first specifying path and second specifying name of file
  3. Takes `File` object specifying path and `String` specifying name of file
  4. Takes `URI` object specifying name and location of file
- Different kinds of paths
  - Absolute path – contains all directories, starting with the root directory, that lead to a specific file or directory
  - Relative path – normally starts from the directory in which the application began executing

Method	Description
`boolean canRead()`	Returns `true` if a file is readable by the current application; `false` otherwise.
`boolean canWrite()`	Returns `true` if a file is writable by the current application; `false` otherwise.
`boolean exists()`	Returns `true` if the name specified as the argument to the `File` constructor is a file or directory in the specified path; `false` otherwise.
`boolean isFile()`	Returns `true` if the name specified as the argument to the `File` constructor is a file; `false` otherwise.
`boolean isDirectory()`	Returns `true` if the name specified as the argument to the `File` constructor is a directory; `false` otherwise.
`boolean isAbsolute()`	Returns `true` if the arguments specified to the `File` constructor indicate an absolute path to a file or directory; `false` otherwise.

Fig. 14.3 | File methods. (Part 1 of 2)

Method	Description
`String getAbsolutePath()`	Returns a string with the absolute path of the file or directory.
`String getName()`	Returns a string with the name of the file or directory.
`String getPath()`	Returns a string with the path of the file or directory.
`String getParent()`	Returns a string with the parent directory of the file or directory (i.e., the directory in which the file or directory can be found).
`long length()`	Returns the length of the file, in bytes. If the `File` object represents a directory, `0` is returned.
`long lastModified()`	Returns a platform-dependent representation of the time at which the file or directory was last modified. The value returned is useful only for comparison with other values returned by this method.
`String[] list()`	Returns an array of strings representing the contents of a directory. Returns `null` if the `File` object does not represent a directory.

Fig.14.3 | File methods. (Part 2 of 2)

Error-Prevention Tip 14.1

Use `File` method `isFile` to determine whether a `File` object represents a file (not a directory) before attempting to open the file.

Demonstrating Class `File`

- Common `File` methods
  - `exists` – return `true` if file exists where it is specified
  - `isFile` – returns `true` if `File` is a file, not a directory
  - `isDirectory` – returns `true` if `File` is a directory
  - `getPath` – return file path as a string
  - `list` – retrieve contents of a directory
- Separator character – used to separate directories and files in a path
  - Windows uses \
  - UNIX uses /
  - Java process both characters, `File.pathSeparator` can be used to obtain the local computer's proper separator character

```
1  // Fig. 14.4: FileDemonstration.java
2  // Demonstrating the File class.
3  import java.io.File;
4
5  public class FileDemonstration
6  {
7     // display information about file user specifies
8     public void analyzePath( String path
9     {
10       // create File object based
11       File name = new File( path
12
13       if ( name.exists() ) // if name exists, output information about it
14       {
15          // display file (or directory) information
16          System.out.printf(
17             "%s%s\n%s\n%s\n%s\n
18             name.getName(), " exists",
19             ( name.isFile() ? "is a file
20             ( name.isDirectory() ? "is a d
21                "is not a directory" ),
22             ( name.isAbsolute() ? "is absc
23                "is not absolute path" ),
24             name.lastModified(), "Length:
25             "Path: ", name.getPath(), "Absolute path:
26             name.getAbsolutePath(), "Parent: ", name.getParent() );
27
```

Create new `File` object; user

Returns `true` if file or directory specified exists

Retri

Returns `true` if `name` is a directory, not a file

Returns `true` if `path` was an absolute path

Retrieve time file or dir

dependent value)

Retrieve length of file in bytes

Retrieve path entered as a string

Retrieve absolute path of file or direc

Retrieve parent directory (path where `File` object's file or directory can be found)

```
28          if ( name.isDirectory() ) // output directory listing
29          {
30             String directory[] = name.list();
31             System.out.println( "\n\nDirector
32
33             for ( String directoryName : directory )
34                System.out.printf( "%s\n", directoryName );
35          } // end else
36       } // end outer if
37       else // not file or directory, output error message
38       {
39          System.out.printf( "%s %s", path, "does not exist." );
40       } // end else
41    } // end method analyzePath
42 } // end class FileDemonstration
```

Returns true if `File` is a directory, not a file

Retrieve and display contents of directory

```
1  // Fig. 14.5: FileDemonstrationTest.java
2  // Testing the FileDemonstration class.
3  import java.util.Scanner;
4
5  public class FileDemonstrationTest
6  {
7     public static void main( String args[] )
8     {
9        Scanner input = new Scanner( System.in );
10       FileDemonstration application = new FileDemonstration();
11
12       System.out.print( "Enter file or directory name here: " );
13       application.analyzePath( input.nextLine() );
14    } // end main
15 } // end class FileDemonstrationTest
```

```
Enter file or directory name here: C:\Program Files\Java\jdk1.5.0\demo\jfc
jfc exists
is not a file
is a directory
is absolute path
Last modified: 1083938776645
Length: 0
Path: C:\Program Files\Java\jdk1.5.0\demo\jfc
Absolute path: C:\Program Files\Java\jdk1.5.0\demo\jfc
Parent: C:\Program Files\Java\jdk1.5.0\demo

Directory contents:

CodePointIM
FileChooserDemo
Font2DTest
Java2D
Metalworks
Notepad
SampleTree
Stylepad
SwingApplet
SwingSet2
TableExample
```

```
Enter file or directory name here:
C:\Program Files\Java\jdk1.5.0\demo\jfc\Java2D\readme.txt
readme.txt exists
is a file
is not a directory
is absolute path
Last modified: 1083938778347
Length: 7501
Path: C:\Program Files\Java\jdk1.5.0\demo\jfc\Java2D\readme.txt
Absolute path: C:\Program Files\Java\jdk1.5.0\demo\jfc\Java2D\readme.txt
Parent: C:\Program Files\Java\jdk1.5.0\demo\jfc\Java2D
```

Outline

FileDemonstration

Test.java

(3 of 3)

Common Programming Error 14.1

Using \ as a directory separator rather than \\ in a string literal is a logic error. A single \ indicates that the \ followed by the next character represents an escape sequence. Use \\ to insert a \ in a string literal.

14.5 Sequential-Access Text Files

- **Records are stored in order by record-key field**
- **Can be created as text files or binary files**

14.5.1 Creating a Sequential-Access Text File

- Java imposes no structure on a file, records do not exist as part of the Java language
- Programmer must structure files
- `Formatter` class can be used to open a text file for writing
  - Pass name of file to constructor
  - If file does not exist, will be created
  - If file already exists, contents are truncated (discarded)
  - Use method `format` to write formatted text to file
  - Use method `close` to close the `Formatter` object (if method not called, OS normally closes file when program exits)

14.5.1 Creating a Sequential-Access Text File

- **Possible exceptions**
  - `SecurityException` – occurs when opening file using `Formatter` object, if user does not have permission to write data to file
  - `FileNotFoundException` – occurs when opening file using `Formatter` object, if file cannot be found and new file cannot be created
  - `NoSuchElementException` – occurs when invalid input is read in by a `Scanner` object
  - `FormatterClosedException` – occurs when an attempt is made to write to a file using an already closed `Formatter` object

```
1  // Fig. 14.6: AccountRecord.java
2  // A class that represents one record of information.
3  package com.deitel.jhtp7.ch14; // packaged for reuse
4
5  public class AccountRecord
6  {
7     private int account;
8     private String firstName;
9     private String lastName;
10    private double balance;
11
12    // no-argument constructor calls other constructor with default values
13    public AccountRecord()
14    {
15       this( 0, "", "", 0.0 ); // call four-argument constructor
16    } // end no-argument AccountRecord constructor
17
18    // initialize a record
19    public AccountRecord( int acct, String first, String last, double bal )
20    {
21       setAccount( acct );
22       setFirstName( first );
23       setLastName( last );
24       setBalance( bal );
25    } // end four-argument AccountRecord constructor
26
```

```
27    // set account number
28    public void setAccount( int acct )
29    {
30       account = acct;
31    } // end method setAccount
32
33    // get account number
34    public int getAccount()
35    {
36       return account;
37    } // end method getAccount
38
39    // set first name
40    public void setFirstName( String first )
41    {
42       firstName = first;
43    } // end method setFirstName
44
45    // get first name
46    public String getFirstName()
47    {
48       return firstName;
49    } // end method getFirstName
50
51    // set last name
52    public void setLastName( String last )
53    {
54       lastName = last;
55    } // end method setLastName
56
```

```
57    // get last name
58    public String getLastName()
59    {
60       return lastName;
61    } // end method getLastName
62
63    // set balance
64    public void setBalance( double bal )
65    {
66       balance = bal;
67    } // end method setBalance
68
69    // get balance
70    public double getBalance()
71    {
72       return balance;
73    } // end method getBalance
74 } // end class AccountRecord
```

```
1  // Fig. 14.7: CreateTextFile.java
2  // Writing data to a text file with class Formatter.
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 import com.deitel.jhtp7.ch14.AccountRecord;
11
12 public class CreateTextFile
13 {
14    private Formatter output; // object used to output text to file
15
16    // enable user to open file
17    public void openFile()
18    {
19       try
20       {
21          output = new Formatter( "clients.txt" );
22       } // end try
23       catch ( SecurityException securityException )
24       {
25          System.err.println(
26             "You do not have write access to this file." );
27          System.exit( 1 );
28       } // end catch
```

Used for writing data to file

Used for retrieving input from user

Object used to output data to file

Open file `clients.txt` for writing

```
29       catch ( FileNotFoundException filesNotFoundException )
30       {
31          System.err.println( "Error creating file." );
32          System.exit( 1 );
33       } // end catch
34    } // end method openFile
35
36    // add records to file
37    public void addRecords()
38    {
39       // object to be written to file
40       AccountRecord record = new AccountRecord();
41
42       Scanner input = new Scanner( System.in );
43
44       System.out.printf( "%s\n%s\n%s\n%s
45          "To terminate input, type the e
46          "when you are prompted to enter
47          "On UNIX/Linux/Mac OS X type <ctrl> d then press Enter",
48          "On Windows type <ctrl> z then press Enter" );
49
50       System.out.printf( "%s\n%s",
51          "Enter account number (> 0), first name, last name and balance.",
52          "? " );
53
```

Create `AccountRecord` to be filled with user input

Create `Scanner` to retrieve input from user

```
54       while ( input.hasNext() ) // loop until end-of-file indicator
55       {
56          try // output values to file
57          {
58             // retrieve data to be output
59             record.setAccount( input.nextInt() ); // read account number
60             record.setFirstName( input.next() ); // read first name
61             record.setLastName( input.next() ); // read last name
62             record.setBalance( input.nextDouble() ); // read balance
63
64             if ( record.getAccount() > 0 )
65             {
66                // write new record
67                output.format( "%d %s %s %.2f\n", record.getAccount(),
68                   record.getFirstName(), record.getLastName(),
69                   record.getBalance() );
70             } // end if
71             else
72             {
73                System.out.println(
74                   "Account number must be greater than 0."
75             } // end else
76          } // end try
77          catch ( FormatterClosedException formatterClosedException )
78          {
79             System.err.println( "Error writing to file." );
80             return;
81          } // end catch
```

Loop while user is entering input

Retrieve input, store data in `AccountRecord`

Write `AccountRecord` information to file

File closed while trying to write to it

```
82          catch ( NoSuchElementException elementException )
83          {
84             System.err.println( "Invalid input. Please try again." );
85             input.nextLine(); // discard input
86          } // end catch
87
88          System.out.printf( "%s %s\n%s", "Enter account number (>0),",
89             "first name, last name and balance.", "? " );
90       } // end while
91    } // end method addRecords
92
93    // close file
94    public void closeFile()
95    {
96       if ( output != null )
97          output.close();
98    } // end method closeFile
99 } // end class CreateTextFile
```

Error with input entered by user

Close file

Operating system	Key combination
UNIX/Linux/Mac OS X	*<return> <ctrl> d*
Windows	*<ctrl> z*

Fig.14.8 | End-of-file key combinations for various popular operating systems.

```
1  // Fig. 14.9: CreateTextFileTest.java
2  // Testing the CreateTextFile class.
3
4  public class CreateTextFileTest
5  {
6     public static void main( String args[] )
7     {
8        CreateTextFile application = new CreateTextFile();
9
10       application.openFile();
11       application.addRecords();
12       application.closeFile();
13    } // end main
14 } // end class CreateTextFileTest
```

```
To terminate input, type the end-of-file indicator
when you are prompted to enter input.
On UNIX/Linux/Mac OS X type <ctrl> d then press Enter
On Windows type <ctrl> z then press Enter

Enter account number (> 0), first name, last name and balance.
? 100 Bob Jones 24.98
Enter account number (> 0), first name, last name and balance.
? 200 Steve Doe -345.67
Enter account number (> 0), first name, last name and balance.
? 300 Pam White 0.00
Enter account number (> 0), first name, last name and balance.
? 400 Sam Stone -42.16
Enter account number (> 0), first name, last name and balance.
? 500 Sue Rich 224.62
Enter account number (> 0), first name, last name and balance.
? ^Z
```

Sample data			
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.14.10 | Sample data for the program in Fig. 14.7.

14.5.2 Reading Data from a Sequential-Access Text File

- Data is stored in files so that it may be retrieved for processing when needed
- `Scanner` object can be used to read data sequentially from a text file
  - Pass `File` object representing file to be read to `Scanner` constructor
  - `FileNotFoundException` occurs if file cannot be found
  - Data read from file using same methods as for keyboard input – `nextInt`, `nextDouble`, `next`, etc.
  - `IllegalStateException` occurs if attempt is made to read from closed `Scanner` object

```
1  // Fig. 14.11: 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.jhtp7.ch14.AccountRecord;
10
11 public class ReadTextFile
12 {
13    private Scanner input;
14
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
```

Open file `clients.txt` for reading

```
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%
36          "First Name", "Last Name", "Balar
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
```

Create `AccountRecord` to store input from file

While there is data to be read from file

Read data from file, store in `AccountRecord`

Display `AccountRecord` contents

```
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
```

Close file

```
1  // Fig. 14.12: ReadTextFileTest.java
2  // This program test class ReadTextFile.
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
```

14.5.3 Case Study: A Credit-Inquiry Program

- To retrieve data sequentially from a file, programs normally start reading from beginning of the file and read all the data consecutively until desired information is found
- Class `Scanner` provides no way to reposition to beginning of file
- Instead, file is closed and reopened

```
1  // Fig. 14.13: MenuOption.java
2  // Defines an enum type 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
12    private final int value; // current menu option
13
14    MenuOption( int valueOption )
15    {
16       value = valueOption;
17    } // end MenuOptions enum constructor
18
19    public int getValue()
20    {
21       return value;
22    } // end method getValue
23 } // end enum MenuOption
```

```
1  // Fig. 14.14: 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.jhtp7.ch14.AccountRecord;
12
13 public class CreditInquiry
14 {
15    private MenuOption accountType;
16    private Scanner input;
17    private 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    {
24       // object to be written to file
25       AccountRecord record = new AccountRecord();
26
```

Scanner used to read data from file

AccountRecord stores record being read from file

```
27       try // read records
28       {
29          // open file to read from beginning
30          input = new Scanner( new File
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, displa
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
46       catch ( NoSuchElementException elementExc
47       {
48          System.err.println( "File improperly formed." );
49          input.close();
50          System.exit( 1 );
51       } // end catch
```

Open file `clients.txt` for reading

While there is data to read from file

Check if record is of requested type

Retrieve input, store data in `AccountRecord`

Display record data to screen

Close `Scanner`

```
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
69    // use record type to determine if record should be display
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
```

Close file

Method determines if record is of proper type

```
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
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
```

Loop until user enters valid request

Retrieve request entered

```
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
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
```

Read file, display proper records

```
1  // Fig. 14.15: 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
```

```
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

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
```

14.5.4 Updating Sequential-Access Files

- **Data in many sequential files cannot be modified without risk of destroying other data in file**
- **Old data cannot be overwritten if new data is not same size**
- **Records in sequential-access files are not usually updated in place. Instead, entire file is usually rewritten.**

14.6 Object Serialization

- With text files, data type information lost
- Object serialization – mechanism to read or write an entire object from a file
- Serialized object – object represented as sequence of bytes, includes object's data and type information about object
- Deserialization – recreate object in memory from data in file
- Serialization and deserialization performed with classes `ObjectInputStream` and `ObjectOutputStream`, methods `readObject` and `writeObject`

14.6.1 Creating a Sequential-Access File Using Object Serialization: *Defining the `AccountRecordSerializable` Class*

- `Serializable` interface – programmers must declare a class to implement the `Serializable` interface, or objects of that class cannot be written to a file
- To open a file for writing objects, create a `FileOutputStream` wrapped by an `ObjectOutputStream`
  - `FileOutputStream` provides methods for writing `byte`-based output to a file
  - `ObjectOutputStream` uses `FileOutputStream` to write objects to file
  - `ObjectOutputStream` method `writeObject` writes object to output file
  - `ObjectOutputStream` method `close` closes both objects

```
1  // Fig. 14.17: AccountRecordSerializable.java
2  // A class that represents one record of information.
3  package com.deitel.jhtp7.ch14; // packaged for reuse
4
5  import java.io.Serializable;
6
7  public class AccountRecordSerializable implements Serializable
8  {
9     private int account;
10    private String firstName;
11    private String lastName;
12    private double balance;
13
14    // no-argument constructor calls other constructor with default values
15    public AccountRecordSerializable()
16    {
17       this( 0, "", "", 0.0 );
18    } // end no-argument AccountRecordSerializable constructor
19
20    // four-argument constructor initializes a record
21    public AccountRecordSerializable(
22       int acct, String first, String last, double bal )
23    {
24       setAccount( acct );
25       setFirstName( first );
26       setLastName( last );
27       setBalance( bal );
28    } // end four-argument AccountRecordSerializable constructor
29
```

Interface `Serializable` specifies that `AccountRecordSerializable` objects can be written to file

```
30    // set account number
31    public void setAccount( int acct )
32    {
33       account = acct;
34    } // end method setAccount
35
36    // get account number
37    public int getAccount()
38    {
39       return account;
40    } // end method getAccount
41
42    // set first name
43    public void setFirstName( String first )
44    {
45       firstName = first;
46    } // end method setFirstName
47
48    // get first name
49    public String getFirstName()
50    {
51       return firstName;
52    } // end method getFirstName
53
54    // set last name
55    public void setLastName( String last )
56    {
57       lastName = last;
58    } // end method setLastName
59
```

```
60    // get last name
61    public String getLastName()
62    {
63       return lastName;
64    } // end method getLastName
65
66    // set balance
67    public void setBalance( double bal )
68    {
69       balance = bal;
70    } // end method setBalance
71
72    // get balance
73    public double getBalance()
74    {
75       return balance;
76    } // end method getBalance
77 } // end class AccountRecordSerializable
```

```
1  // Fig. 14.18: CreateSequentialFile.java
2  // Writing objects sequentially to a file with class ObjectOutputStream.
3  import java.io.FileOutputStream;
4  import java.io.IOException;
5  import java.io.ObjectOutputStream;
6  import java.util.NoSuchElementException;
7  import java.util.Scanner;
8
9  import com.deitel.jhtp7.ch14.AccountRecordSerializable;
10
11 public class CreateSequentialFile
12 {
13    private ObjectOutputStream output; // outputs data to file
14
15    // allow user to specify file name
16    public void openFile()
17    {
18       try // open file
19       {
20          output = new ObjectOutputStream(
21             new FileOutputStream( "clients.ser" ) );
22       } // end try
23       catch ( IOException ioException )
24       {
25          System.err.println( "Error opening file." );
26       } // end catch
27    } // end method openFile
28
```

Class used to create byte-based output stream

Class used to create output object data to byte-based stream

Open file `clients.ser` for writing

```
29    // add records to file
30    public void addRecords()
31    {
32       AccountRecordSerializable record; // object to be written to file
33       int accountNumber = 0; // account number for record object
34       String firstName; // first name for record object
35       String lastName; // last name for record object
36       double balance; // balance for record object
37
38       Scanner input = new Scanner( System.in );
39
40       System.out.printf( "%s\n%s\n%s\n%s\n\n",
41          "To terminate input, type the end-of-file indicator ",
42          "when you are prompted to enter input.",
43          "On UNIX/Linux/Mac OS X type <ctrl> d then press Enter",
44          "On Windows type <ctrl> z then press Enter" );
45
46       System.out.printf( "%s\n%s",
47          "Enter account number (> 0), first name, last name and balance.",
48          "? " );
49
50       while ( input.hasNext() ) // loop until end-of-file indicator
51       {
52          try // output values to file
53          {
54             accountNumber = input.nextInt(); // read account number
55             firstName = input.next(); // read first name
56             lastName = input.next(); // read last name
57             balance = input.nextDouble(); // read balance
58
```

```
59            if ( accountNumber > 0 )
60            {
61               // create new record
62               record = new AccountRecordSerializable( accoun
63                  firstName, lastName, balance );
64               output.writeObject( record ); // output rec
65            } // end if
66            else
67            {
68               System.out.println(
69                  "Account number must be greater than 0." );
70            } // end else
71         } // end try
72         catch ( IOException ioException )
73         {
74            System.err.println( "Error writing to file." );
75            return;
76         } // end catch
77         catch ( NoSuchElementException elementException )
78         {
79            System.err.println( "Invalid input. Please try again." );
80            input.nextLine(); // discard input so user can try again
81         } // end catch
82
83         System.out.printf( "%s %s\n%s", "Enter account number (>0),",
84            "first name, last name and balance.", "? " );
85      } // end while
86   } // end method addRecords
87
```

Write record object to file

Create `AccountRecord` based on user input

```
88    // close file and terminate application
89    public void closeFile()
90    {
91       try // close file
92       {
93          if ( output != null )
94             output.close();
95       } // end try
96       catch ( IOException ioException )
97       {
98          System.err.println( "Error closing file." );
99          System.exit( 1 );
100      } // end catch
101   } // end method closeFile
102} // end class CreateSequentialFile
```

```
1  // Fig. 14.19: CreateSequentialFileTest.java
2  // Testing class CreateSequentialFile.
3
4  public class CreateSequentialFileTest
5  {
6     public static void main( String args[] )
7     {
8        CreateSequentialFile application = new CreateSequentialFile();
9
10       application.openFile();
11       application.addRecords();
12       application.closeFile();
13    } // end main
14 } // end class CreateSequentialFileTest
```

```
To terminate input, type the end-of-file indicator
when you are prompted to enter input.
On UNIX/Linux/Mac OS X type <ctrl> d then press Enter
On Windows type <ctrl> z then press Enter

Enter account number (> 0), first name, last name and balance.
? 100 Bob Jones 24.98
Enter account number (> 0), first name, last name and balance.
? 200 Steve Doe -345.67
Enter account number (> 0), first name, last name and balance.
? 300 Pam White 0.00
Enter account number (> 0), first name, last name and balance.
? 400 Sam Stone -42.16
Enter account number (> 0), first name, last name and balance.
? 500 Sue Rich 224.62
Enter account number (> 0), first name, last name and balance.
? ^Z
```

Common Programming Error 14.2

It is a logic error to open an existing file for output when, in fact, the user wishes to preserve the file.

14.6.2 Reading and Deserializing Data from a Sequential-Access File

- To open a file for reading objects, create a `FileInputStream` wrapped by an `ObjectInputStream`
  - `FileInputStream` provides methods for reading `byte`-based input from a file
  - `ObjectInputStream` uses `FileInputStream` to read objects from file
  - `ObjectInputStream` method `readObject` reads in object, which is then downcast to proper type
    - `EOFException` occurs if attempt made to read past end of file
    - `ClassNotFoundException` occurs if the class for the object being read cannot be located
  - `ObjectInputStream` method `close` closes both objects

```
1  // Fig. 14.20: ReadSequentialFile.java
2  // This program reads a file of objects sequentially
3  // and displays each record.
4  import java.io.EOFException;
5  import java.io.FileInputStream;
6  import java.io.IOException;
7  import java.io.ObjectInputStream;
8
9  import com.deitel.jhtp7.ch14.AccountRecordSerializable;
10
11 public class ReadSequentialFile
12 {
13    private ObjectInputStream input;
14
15    // enable user to select file to open
16    public void openFile()
17    {
18       try // open file
19       {
20          input = new ObjectInputStream(
21             new FileInputStream( "clients.ser" ) );
22       } // end try
23       catch ( IOException ioException )
24       {
25          System.err.println( "Error opening file." );
26       } // end catch
27    } // end method openFile
28
```

Class used to create byte-based input stream

Class used to read input object data to byte-based stream

Open file `clients.ser` for reading

```
29    // read record from file
30    public void readRecords()
31    {
32       AccountRecordSerializable record;
33       System.out.printf( "%-10s%-12s%-12s%10s\n", "Account",
34          "First Name", "Last Name", "Balance" );
35
36       try // input the values from the file
37       {
38          while ( true )
39          {
40             record = ( AccountRecordSerializable ) input.readObject();
41
42             // display record contents
43             System.out.printf( "%-10d%-12s%-12s%10.2f\n",
44                record.getAccount(), record.getFirstName(),
45                record.getLastName(), record.getBalance() );
46          } // end while
47       } // end try
48       catch ( EOFException endOfFileException )
49       {
50          return; // end of file was reached
51       } // end catch
```

Read record from file

Output record information to screen

```
52       catch ( ClassNotFoundException classNotFoundException )
53       {
54          System.err.println( "Unable to create object." );
55       } // end catch
56       catch ( IOException ioException )
57       {
58          System.err.println( "Error during read from file." );
59       } // end catch
60    } // end method readRecords
61
62    // close file and terminate application
63    public void closeFile()
64    {
65       try // close file and exit
66       {
67          if ( input != null )
68             input.close();
69       } // end try
70       catch ( IOException ioException )
71       {
72          System.err.println( "Error closing file." );
73          System.exit( 1 );
74       } // end catch
75    } // end method closeFile
76 } // end class ReadSequentialFile
```

Close file

```
1  // Fig. 14.21: ReadSequentialFileTest.java
2  // This program test class ReadSequentialFile.
3
4  public class ReadSequentialFileTest
5  {
6     public static void main( String args[] )
7     {
8        ReadSequentialFile application = new ReadSequentialFile();
9
10       application.openFile();
11       application.readRecords();
12       application.closeFile();
13    } // end main
14 } // end class ReadSequentialFileTest
```

```
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
```

14.7 Additional `java.io` Classes: *Interfaces and Classes for Byte-Based Input and Output*

- `InputStream` and `OutputStream` classes
  - `abstract` classes that declare methods for performing byte-based input and output
- `PipedInputStream` and `PipedOutputStream` classes
  - Establish pipes between two threads in a program
  - Pipes are synchronized communication channels between threads
- `FilterInputStream` and `FilterOutputStream` classes
  - Provides additional functionality to stream, such as aggregating data byte into meaningful primitive-type units
- `PrintStream` class
  - Performs text output to a specified stream
- `DataInput` and `DataOutput` interfaces
  - For reading and writing primitive types to a file
  - `DataInput` implemented by classes `RandomAccessFile` and `DataInputStream`, `DataOutput` implemented by `RandomAccessFile` and `DataOuputStream`
- `SequenceInputStream` class enables concatenation of several `InputStream`s – program sees group as one continuous `InputStream`

Interfaces and Classes for Byte-Based Input and Output

- **Buffering is an I/O-performance-enhancement technique**
  - **Greatly increases efficiency of an application**
  - **Output (uses `BufferedOutputStream` class)**
    - **Each output statement does not necessarily result in an actual physical transfer of data to the output device – data is directed to a region of memory called a buffer (faster than writing to file)**
    - **When buffer is full, actual transfer to output device is performed in one large physical output operation (also called logical output operations)**
    - **Partially filled buffer can be forced out with method `flush`**
  - **Input (uses `BufferedInputStream` class)**
    - **Many logical chunks of data from a file are read as one physical input operation (also called logical input operation)**
    - **When buffer is empty, next physical input operation is performed**
- **`ByteArrayInputStream` and `ByteArrayOutputStream` classes used for inputting from `byte` arrays in memory and outputting to `byte` arrays in memory**

Performance Tip 14.1

Buffered I/O can yield significant performance improvements over unbuffered I/O.

Interfaces and Classes for Character-Based Input and Output

- `Reader` and `Writer` abstract classes
  - Unicode two-byte, character-based streams
- `BufferedReader` and `BufferedWriter` classes
  - Enable buffering for character-based streams
- `CharArrayReader` and `CharArrayWriter` classes
  - Read and write streams of characters to character arrays
- `LineNumberReader` class
  - Buffered character stream that keeps track of number of lines read
- `PipedReader` and `PipedWriter` classes
  - Implement piped-character streams that can be used to transfer information between threads
- `StringReader` and `StringWriter` classes
  - Read characters from and write characters to `String`s

14.8 Opening Files with `JFileChooser`

- `JFileChooser` – class used to display a dialog that enables users to easily select files
  - Method `setFileSelectionMode` specifies what user can select from `JFileChooser`
    - `FILES_AND_DIRECTORIES` constant indicates files and directories
    - `FILES_ONLY` constant indicates files only
    - `DIRECTORIES_ONLY` constant indicates directories only
  - Method `showOpenDialog` displays `JFileChooser` dialog titled Open, with Open and Cancel buttons (to open a file/directory or dismiss the dialog, respectively)
    - `CANCEL_OPTION` constant specifies that user click Cancel button
  - Method `getSelectedFile` retrieves file or directory user selected

```
1  // Fig. 14.22: FileDemonstration.java
2  // Demonstrating the File class.
3  import java.awt.BorderLayout;
4  import java.awt.event.ActionEvent;
5  import java.awt.event.ActionListener;
6  import java.io.File;
7  import javax.swing.JFileChooser;
8  import javax.swing.JFrame;
9  import javax.swing.JOptionPane;
10 import javax.swing.JScrollPane;
11 import javax.swing.JTextArea;
12 import javax.swing.JTextField;
13
14 public class FileDemonstration extends JFrame
15 {
16    private JTextArea outputArea; // used for output
17    private JScrollPane scrollPane; // used to provide scrolling to output
18
19    // set up GUI
20    public FileDemonstration()
21    {
22       super( "Testing class File" );
23
24       outputArea = new JTextArea();
25
26       // add outputArea to scrollPane
27       scrollPane = new JScrollPane( outputArea );
28
29       add( scrollPane, BorderLayout.CENTER ); // add scrollPane to GUI
30
```

Class for display `JFileChooser` dialog

```
31       setSize( 400, 400 ); // set GUI size
32       setVisible( true ); // display GUI
33
34       analyzePath(); // create and analyze File object
35    } // end FileDemonstration constructor
36
37    // allow user to specify file name
38    private File getFile()
39    {
40       // display file dialog, so user can choose file to open
41       JFileChooser fileChooser = new JFileChooser();
42       fileChooser.setFileSelectionMode(
43          JFileChooser.FILES_AND_DIRECTORIES );
44
45       int result = fileChooser.showOpenDialog( this );
46
47       // if user clicked Cancel button on dialog, return
48       if ( result == JFileChooser.CANCEL_OPTION )
49          System.exit( 1 );
50
51       File fileName = fileChooser.getSelectedFile
52
53       // display error if invalid
54       if ( ( fileName == null ) || ( fileName.getName().equals( "" ) ) )
55       {
56          JOptionPane.showMessageDialog( this, "Invalid File Name",
57             "Invalid File Name", JOptionPane.ERROR_MESSAGE );
58          System.exit( 1 );
59       } // end if
60
```

Create `JFileChooser`

Allow user to select both files and directories

Display dialog

User clicked Cancel

Retrieve file or directory selected by user

```
61       return fileName;
62    } // end method getFile
63
64    // display information about file user specifies
65    public void analyzePath()
66    {
67       // create File object based on user input
68       File name = getFile();
69
70       if ( name.exists() ) // if name exists, output information about it
71       {
72          // display file (or directory) information
73          outputArea.setText( String.format(
74             "%s%s\n%s\n%s\n%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s",
75             name.getName(), " exists",
76             ( name.isFile() ? "is a file" : "is not a file" ),
77             ( name.isDirectory() ? "is a directory" :
78                "is not a directory" ),
79             ( name.isAbsolute() ? "is absolute path" :
80                "is not absolute path" ), "Last modified: ",
81             name.lastModified(), "Length: ", name.length(),
82             "Path: ", name.getPath(), "Absolute path: ",
83             name.getAbsolutePath(), "Parent: ", name.getParent() ) );
84
```

Display information about file

```
85          if ( name.isDirectory() ) // output directory listing
86          {
87             String directory[] = name.list();
88             outputArea.append( "\n\nDirectory contents:\n" );
89
90             for ( String directoryName : directory )
91                outputArea.append( directoryName + "\n" );
92          } // end else
93       } // end outer if
94       else // not file or directory, output error message
95       {
96          JOptionPane.showMessageDialog( this, name +
97             " does not exist.", "ERROR", JOptionPane.ERROR_MESSAGE );
98       } // end else
99    } // end method analyzePath
100} // end class FileDemonstration
```

```
1  // Fig. 14.23: FileDemonstrationTest.java
2  // Testing the FileDmonstration class.
3  import javax.swing.JFrame;
4
5  public class FileDemonstrationTest
6  {
7     public static void main( String args[] )
8     {
9        FileDemonstration application = new FileDemonstration();
10       application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11    } // end main
12 } // end class FileDemonstrationTest
```