

# **Lesson 13 – Part I**

## **Files, Streams and Object Serialization**

Assoc. Prof. Marenglen Biba

In this Chapter you'll learn:

- What files are and how they are used to retain application data between successive executions.
- 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 file processing.
- To use classes `Scanner` and `Formatter` to process text files.
- To use classes `FileInputStream` and `FileOutputStream` to read from and write to files.
- To use classes `ObjectInputStream` and `ObjectOutputStream` to read objects from and write objects to files.
- To use a `JFileChooser` dialog.

## **17.1** Introduction

## **17.2** Data Hierarchy

## **17.3** Files and Streams

## **17.4** Class File

## **17.5** Sequential-Access Text Files

17.5.1 Creating a Sequential-Access Text File

17.5.2 Reading Data from a Sequential-Access Text File

17.5.3 Case Study: A Credit-Inquiry Program

17.5.4 Updating Sequential-Access Files

## **17.6** Object Serialization

17.6.1 Creating a Sequential-Access File Using Object Serialization

17.6.2 Reading and Deserializing Data from a Sequential-Access File

## **17.7** Additional `java.io` Classes

17.7.1 Interfaces and Classes for Byte-Based Input and Output

17.7.2 Interfaces and Classes for Character-Based Input and Output

## **17.8** Opening Files with `JFileChooser`

## **17.9** Wrap-Up

# 17.1 Introduction

- ▶ Data stored in variables and arrays is **temporary**
  - It's lost when a local variable goes out of scope or when the program terminates
- ▶ For long-term retention of data, computers use **files**.
- ▶ Computers store files on **secondary storage devices**
  - hard disks, optical disks, flash drives and magnetic tapes.
- ▶ Data maintained in files is **persistent data** because it exists beyond the duration of program execution.

## 17.2 Data Hierarchy: Character Set

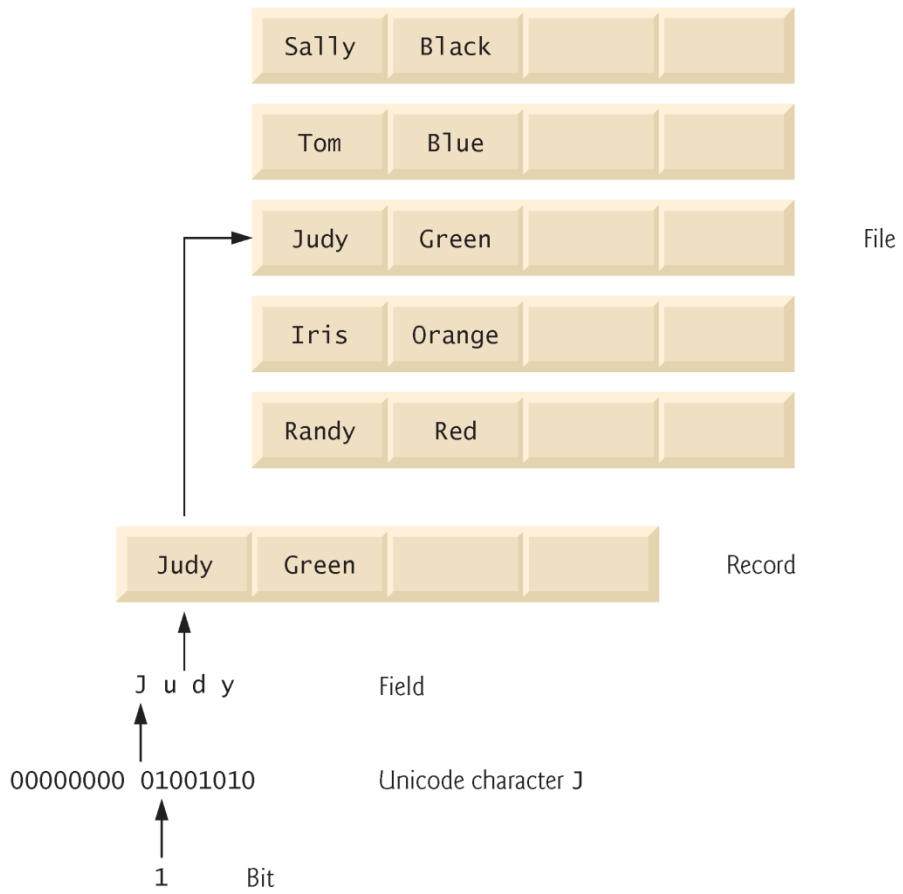
- ▶ Programmers prefer to work with **decimal digits** (0–9), **letters** (A–Z and a–z), and **special symbols** (e.g., \$, @, %, &, \*, (, ), –, +, ", :, ? and / ).
  - Known as **characters**.
- ▶ **Character set** — the set of all the characters used to write programs and represent data items.
- ▶ Java uses **Unicode** characters that are composed of two **bytes**, each composed of eight bits
- ▶ Java type **byte** can be used to represent byte data.
- ▶ Unicode contains characters for many of the world's languages.

## 17.2 Data Hierarchy: fields

- ▶ **Fields** are composed of characters or bytes.
- ▶ A field is a **group of characters** or **bytes** that conveys meaning.
- ▶ Data items processed by computers form a **data hierarchy** that becomes larger and more complex in structure as we progress from bits to characters to fields, and so on.

## 17.2 Data Hierarchy (cont.)

- ▶ Typically, several fields compose a **record** (implemented as a `class` in Java).
- ▶ A record is a group of related fields.
- ▶ A **file** is a group of related records.



**Fig. 17.1** | Data hierarchy.

## 17.3 Files and Streams

- ▶ Java views each file as a sequential **stream of bytes** (Fig. 17.2).
- ▶ Every operating system provides a mechanism to determine the end of a file, such as an **end-of-file marker** or a count of the total bytes in the file that is recorded in a system-maintained administrative data structure.
- ▶ A Java program simply receives an **indication** from the operating system when it **reaches the end** of the stream



**Fig. 17.2** | Java's view of a file of  $n$  bytes.

## 17.3 Files and Streams (cont.)

- ▶ **File streams** can be used to input and output data as bytes or characters.
- ▶ Streams that input and output bytes are known as **byte-based streams**, representing data in its binary format.
- ▶ Streams that input and output characters are known as **character-based streams**, representing data as a sequence of characters.

## 17.3 Files and Streams (cont.)

- ▶ Files that are created using byte-based streams are referred to as **binary files**.
- ▶ Files created using character-based streams are referred to as **text files**. Text files can be read by text editors.
- ▶ **Binary files** are read by programs that understand the specific content of the file and the ordering of that content.

## 17.3 Files and Streams (cont.)

- ▶ A Java program **opens** a file by **creating an object** and associating a stream of bytes or characters with it.
- ▶ Java creates three stream objects when a program begins executing
  - `System.in` (the standard input stream object) normally **inputs bytes** from the keyboard
  - `System.out` (the standard output stream object) normally **outputs character data** to the screen
  - `System.err` (the standard error stream object) normally **outputs character-based** error messages to the screen.
- ▶ Class `System` provides methods **setIn**, **setOut** and **setErr** to **redirect** the standard input, output and error streams, respectively.

## 17.3 Files and Streams (cont.)

- ▶ Java programs perform file processing by using classes from package **java.io**.
- ▶ Includes definitions for stream classes
  - **FileInputStream** (for byte-based input from a file)
  - **FileOutputStream** (for byte-based output to a file)
  - **FileReader** (for character-based input from a file)
  - **FileWriter** (for character-based output to a file)
- ▶ You open a file by creating an object of one these stream classes. The object's **constructor** opens the file.

## 17.3 Files and Streams (cont.)

- ▶ Java can perform input and **output of objects** or variables of primitive data types **without** having to worry about the **details of converting** such values to byte format.
- ▶ To perform such input and output, objects of classes **ObjectInputStream** and **ObjectOutputStream** can be used together with the byte-based file stream classes **FileInputStream** and **FileOutputStream**.
- ▶ The complete hierarchy of classes in package `java.io` can be viewed in the online documentation at
  - <http://docs.oracle.com/javase/8/docs/api/java/io/package-tree.html>

## 17.3 Files and Streams (cont.)

- ▶ Class **File** provides information about files and directories.
- ▶ Character-based input and output can be performed with classes **Scanner** and **Formatter**.
  - Class **Scanner** is used extensively to **input data** from the keyboard. This class can also read data from a file.
  - Class **Formatter** enables **formatted data to be output** to any text-based stream in a manner similar to method `System.out.printf`.

## 17.4 Class File

- ▶ Class `File` provides **four constructors**.
- ▶ The one with a **`String` argument** specifies the **name** of a file or directory to associate with the `File` object.
  - The **name** can contain **path information** as well as a file or directory name.
  - A file or directory's **path** specifies its location on disk.
    - An **absolute path** contains all the directories, starting with the **root directory**, that lead to a specific file or directory.
    - A **relative path** normally starts from the directory in which the application **began executing** and is therefore “relative” to the current directory.

## 17.4 Class File (cont.)

- ▶ The constructor with **two String arguments** specifies an absolute or relative **path** and the **file** or directory to associate with the **File** object.
- ▶ The constructor with **File and String arguments** uses an existing **File** object that specifies the parent directory of the file or directory specified by the **String** argument.
- ▶ The fourth constructor uses a **URI** object to locate the file.
  - A **Uniform Resource Identifier (URI)** is a more general form of the **Uniform Resource Locators (URLs)** that are used to locate websites.
- ▶ Figure 17.3 lists some common **File** methods.
  - <http://docs.oracle.com/javase/8/docs/api/java/io/File.html>

Method	Description
<code>boolean canRead()</code>	Returns <code>true</code> if a file is readable by the current application; <code>false</code> otherwise.
<code>boolean canWrite()</code>	Returns <code>true</code> if a file is writable by the current application; <code>false</code> otherwise.
<code>boolean exists()</code>	Returns <code>true</code> if the file or directory represented by the <code>File</code> object exists; <code>false</code> otherwise.
<code>boolean isFile()</code>	Returns <code>true</code> if the name specified as the argument to the <code>File</code> constructor is a file; <code>false</code> otherwise.
<code>boolean isDirectory()</code>	Returns <code>true</code> if the name specified as the argument to the <code>File</code> constructor is a directory; <code>false</code> otherwise.
<code>boolean isAbsolute()</code>	Returns <code>true</code> if the arguments specified to the <code>File</code> constructor indicate an absolute path to a file or directory; <code>false</code> otherwise.
<code>String getAbsolutePath()</code>	Returns a <code>String</code> with the absolute path of the file or directory.
<code>String getName()</code>	Returns a <code>String</code> with the name of the file or directory.
<code>String getPath()</code>	Returns a <code>String</code> with the path of the file or directory.

**Fig. 17.3** | File methods. (Part 1 of 2.)

Method	Description
<code>String getParent()</code>	Returns a <code>String</code> with the parent directory of the file or directory (i.e., the directory in which the file or directory is located).
<code>long length()</code>	Returns the length of the file, in bytes. If the <code>File</code> object represents a directory, an unspecified value is returned.
<code>long lastModified()</code>	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.
<code>String[] list()</code>	Returns an array of <code>Strings</code> representing a directory's contents. Returns <code>null</code> if the <code>File</code> object does not represent a directory.

**Fig. 17.3** | `File` methods. (Part 2 of 2.)

```
1 // Fig. 17.4: FileDemonstration.java
2 // File class used to obtain file and directory information.
3 import java.io.File;
4 import java.util.Scanner;
5
6 public class FileDemonstration
7 {
8     public static void main( String[] args )
9     {
10         Scanner input = new Scanner( System.in );
11
12         System.out.print( "Enter file or directory name: " );
13         analyzePath( input.nextLine() );
14     } // end main
15
16     // display information about file user specifies
17     public static void analyzePath( String path )
18     {
19         // create File object based on user input
20         File name = new File( path );
21
```

Associates a file or directory with a File object.

**Fig. 17.4** | File class used to obtain file and directory information. (Part 1 of 5.)



---

```
46     else // not file or directory, output error message
47     {
48         System.out.printf( "%s %s", path, "does not exist." );
49     } // end else
50 } // end method analyzePath
51 } // end class FileDemonstration
```

---

**Fig. 17.4** | File class used to obtain file and directory information. (Part 3 of 5.)

```
Enter file or directory name: E:\Program Files\Java\jdk1.6.0_11\demo\jfc
jfc exists
is not a file
is a directory
is absolute path
Last modified: 1228404395024
Length: 4096
Path: E:\Program Files\Java\jdk1.6.0_11\demo\jfc
Absolute path: E:\Program Files\Java\jdk1.6.0_11\demo\jfc
Parent: E:\Program Files\Java\jdk1.6.0_11\demo

Directory contents:

CodePointIM
FileChooserDemo
Font2DTest
Java2D
Laffy
Metalworks
Notepad
SampleTree
Stylepad
SwingApplet
SwingSet2
SwingSet3
```

**Fig. 17.4** | File class used to obtain file and directory information. (Part 4 of 5.)

```
Enter file or directory name: C:\Program Files\Java\jdk1.6.0_11\demo\jfc
\Java2D\README.txt
README.txt exists
is a file
is not a directory
is absolute path
Last modified: 1228404384270
Length: 7518
Path: E:\Program Files\Java\jdk1.6.0_11\demo\jfc\Java2D\README.txt
Absolute path: E:\Program Files\Java\jdk1.6.0_11\demo\jfc\Java2D\RE-
ADME.txt
Parent: E:\Program Files\Java\jdk1.6.0_11\demo\jfc\Java2D
```

**Fig. 17.4** | File class used to obtain file and directory information. (Part 5 of 5.)

## 17.4 Class File (cont.)

- ▶ A **separator character** is used to separate directories and files in the path.
- ▶ On Windows, the separator character is a backslash (\).
- ▶ On Linux/UNIX, it's a forward slash (/).
- ▶ Java processes **both** characters **identically** in a path name.
- ▶ When building `Strings` that represent path information, use `File.separator` to **obtain the local computer's proper separator**.
  - This constant returns a `String` consisting of one character — the proper separator for the system.



## Common Programming Error 17.1

*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.*

## 17.5 Sequential-Access Text Files

- ▶ Sequential-access files store records **in order** by the record-key field.
- ▶ Text files are human-readable files.

## 17.5.1 Creating a Sequential-Access Text File

- ▶ Java imposes **no structure** on a file
- Notions such as records **do not exist** as part of the Java language.

---

```
1 // Fig. 17.5: AccountRecord.java
2 // AccountRecord class maintains information for one account.
3 package com.deitel.ch17; // 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
```

---

**Fig. 17.5** | AccountRecord class maintains information for one account. (Part I of 4.)

---

```
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
```

---

**Fig. 17.5** | AccountRecord class maintains information for one account. (Part 2 of 4.)

---

```
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
```

---

**Fig. 17.5** | AccountRecord class maintains information for one account. (Part 3 of 4.)

---

```
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
```

---

**Fig. 17.5** | AccountRecord class maintains information for one account. (Part 4 of 4.)

## 17.5.1 Creating a Sequential-Access Text File (cont.)

- ▶ `Formatter` outputs formatted `Strings` to the specified stream.
- ▶ The constructor with one `String` argument receives the **name of the file, including its path**.
  - If a path is not specified, the JVM assumes that the file is in the directory from which the program was executed.
- ▶ If the file does not exist, it will be created.
- ▶ If an existing file is opened, its contents are **truncated**.

```
1 // Fig. 17.6: CreateTextFile.java
2 // Writing data to a sequential 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.ch17.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" ); // open the file
22         } // end try
```

Used to output text to a file.

Opens the file clients.txt.

**Fig. 17.6** | Writing data to a sequential text file with class Formatter. (Part I of 5.)

---

```
23     catch ( SecurityException securityException )
24     {
25         System.err.println(
26             "You do not have write access to this file." );
27         System.exit( 1 ); // terminate the program
28     } // end catch
29     catch ( FileNotFoundException fileNotFoundException )
30     {
31         System.err.println( "Error opening or creating file." );
32         System.exit( 1 ); // terminate the program
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
```

---

**Fig. 17.6** | Writing data to a sequential text file with class Formatter. (Part 2 of 5.)

```
44 System.out.printf( "%s\n%s\n%s\n%s\n\n",
45 "To terminate input, type the end-of-file indicator ",
46 "when you are prompted to enter input.",
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
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         {
```

Has end-of-file been reached?

**Fig. 17.6** | Writing data to a sequential text file with class Formatter. (Part 3 of 5.)

```
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
82 catch ( NoSuchElementException elementException )
83 {
84     System.err.println( "Invalid input. Please try again." );
85     input.nextLine(); // discard input so user can try again
86 } // end catch
87
```

Writes text to the file associated with output.

**Fig. 17.6** | Writing data to a sequential text file with class Formatter. (Part 4 of 5.)

---

```
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
```

Closes the file.

---

**Fig. 17.6** | Writing data to a sequential text file with class Formatter. (Part 5 of 5.)

---

```
1 // Fig. 17.8: 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
```

---

**Fig. 17.8** | Testing the CreateTextFile class. (Part I of 2.)

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
```

**Fig. 17.8** | Testing the CreateTextFile class. (Part 2 of 2.)

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. 17.9** | Sample data for the program in Figs. 17.6–17.8.

## 17.5.1 Creating a Sequential-Access Text File (cont.)

- ▶ A **SecurityException** occurs if the user does not have permission to write data to the file.
- ▶ A **FileNotFoundException** occurs if the file does not exist and a new file cannot be created.
- ▶ `static` method **System.exit** terminates an application.
  - An argument of `0` indicates **successful** program termination.
  - A nonzero value, normally indicates that an **error** has occurred.
  - The argument is useful if the program is executed from a **batch file** on Windows or a **shell script** on UNIX/Linux/Mac OS X.

Operating system	Key combination
UNIX/Linux/Mac OS X	<i>&lt;Enter&gt; &lt;Ctrl&gt; d</i>
Windows	<i>&lt;Ctrl&gt; z</i>

**Fig. 17.7** | End-of-file key combinations.

## 17.5.1 Creating a Sequential-Access Text File (cont.)

- ▶ Scanner method **hasNext** determines whether the end-of-file key combination has been entered.
- ▶ A **NoSuchElementException** occurs if the data being read by a Scanner method is in the wrong format or if there is no more data to input.
- ▶ Formatter method **format** works like `System.out.printf`
- ▶ A **FormatterClosedException** occurs if the Formatter is closed when you attempt to output.
- ▶ Formatter method **close** closes the file.
  - If method `close` is **not called explicitly**, the operating system normally will **close the file** when program execution terminates.

## 17.5.1 Creating a Sequential-Access Text File (cont.)

- ▶ Different platforms use different **line-separator** characters.
- ▶ On UNIX/Linux-/Mac OS X, the **line-separator** is a newline (`\n`).
- ▶ On Windows, it is a combination of a carriage return and a line feed — represented as `\r\n`.
- ▶ You can use the `%n` format specifier in a format control string to output a **platform-specific line separator**.
- ▶ Method `System.out.println` outputs a platform-specific line separator after its argument.
- ▶ Regardless of the line separator used in a text file, a Java program can still recognize the lines of text and read them.

## 17.5.2 Reading Data from a Sequential-Access Text File

- ▶ The application in Figs. 17.10 and 17.11 reads records from the file "clients.txt" created by the application of Section 17.5.1 and displays the record contents.

---

```
1 // Fig. 17.10: 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
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
```

Opens clients.txt for reading.

---

**Fig. 17.10** | Sequential file reading using a Scanner. (Part I of 4.)

```

22     catch ( FileNotFoundException fileNotFoundException )
23     {
24         System.err.println( "Error opening file." );
25         System.exit( 1 );
26     } // end catch
27 } // end method openFile
28
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

```

Has end-of-file been reached?

**Fig. 17.10** | Sequential file reading using a Scanner. (Part 2 of 4.)

---

```
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
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
```

---

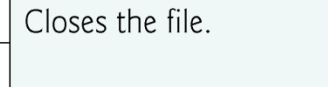
**Fig. 17.10** | Sequential file reading using a Scanner. (Part 3 of 4.)

---

```
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
```

---

Closes the file.



**Fig. 17.10** | Sequential file reading using a Scanner. (Part 4 of 4.)

```
1 // Fig. 17.11: 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. 17.11** | Testing the ReadTextFile class.

## 17.5.3 Reading Data from a Sequential-Access Text File

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

## 17.5.4 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. 17.12: 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
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 }
```

---

**Fig. 17.12** | Enumeration for the credit-inquiry program's menu options.

---

```
1 // Fig. 17.13: 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. 17.13** | Credit-inquiry program. (Part I 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 account 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
```

Opens clients.txt for reading.

**Fig. 17.13** | 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. 17.13** | 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. 17.13** | 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. 17.13** | 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. 17.13** | Credit-inquiry program. (Part 6 of 6.)

---

```
1 // Fig. 17.14: 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. 17.14** | 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. 17.15** | Sample output of the credit-inquiry program in Fig. 17.14. (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. 17.15** | Sample output of the credit-inquiry program in Fig. 17.14. (Part 2 of 2.)

## 17.5.5 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.

## 17.6 Object Serialization

- ▶ To read an entire object from or write an entire object to a file, Java provides **object serialization**.
- ▶ A **serialized object** is represented as a sequence of bytes that includes the object's data and its type information.
- ▶ After a serialized object has been written into a file, it can be read from the file and **deserialized** to recreate the object in memory.



## Software Engineering Observation 17.1

*The serialization mechanism makes exact copies of objects. This makes it a simple way to clone objects without having to override Object method clone.*

## 17.6 Object Serialization (cont.)

- ▶ Classes `ObjectInputStream` and `ObjectOutputStream`, which respectively implement the **ObjectInput** and **ObjectOutput** interfaces, enable entire objects to be read from or written to a stream.
- ▶ To use serialization with files, initialize `ObjectInputStream` and `ObjectOutputStream` objects with `FileInputStream` and `FileOutputStream` objects.

## 17.6 Object Serialization (cont.)

- ▶ `ObjectOutput` interface method **`writeObject`** takes an `Object` as an argument and writes its information to an `OutputStream`.
- ▶ A class that implements `ObjectOutput` (such as `ObjectOutputStream`) declares this method and ensures that the **object being output** implements `Serializable`.
- ▶ `ObjectInput` interface method **`readObject`** reads and returns a reference to an `Object` from an `InputStream`.
  - After an object has been read, its reference **can be cast** to the object's actual type.

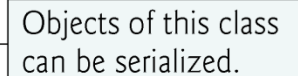
## 17.6.1 Creating a Sequential-Access File Using Object Serialization

- ▶ Objects of classes that implement interface **Serializable** can be serialized and deserialized with `ObjectOutputStreams` and `ObjectInputStreams`.
- ▶ Interface `Serializable` is a **tagging interface**.
  - It does not contain methods.
- ▶ A class that implements `Serializable` is tagged as being a **Serializable object**.
- ▶ An `ObjectOutputStream` will not output an object **unless** it *is a* `Serializable` object.

---

```
1 // Fig. 17.16: AccountRecordSerializable.java
2 // AccountRecordSerializable class for serializable objects.
3 package com.deitel.ch17; // 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
```

Objects of this class  
can be serialized.



---

**Fig. 17.16** | AccountRecordSerializable class for serializable objects. (Part I of 4.)

---

```
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
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
```

---

**Fig. 17.16** | AccountRecordSerializable class for serializable objects. (Part 2 of 4.)

---

```
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
```

---

**Fig. 17.16** | AccountRecordSerializable class for serializable objects. (Part 3 of 4.)

---

```
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
```

---

**Fig. 17.16** | AccountRecordSerializable class for serializable objects. (Part 4 of 4.)

## 17.6.1 Creating a Sequential-Access File Using Object Serialization (cont.)

- ▶ In a class that implements `Serializable`, every variable must be `Serializable`.
- ▶ Any one that is not must be declared **`transient`** so it will be ignored during the serialization process.
- ▶ All primitive-type variables are serializable.
- ▶ For reference-type variables, check the class's documentation (and possibly its superclasses) to ensure that the type is `Serializable`.

```
1 // Fig. 17.17: 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.ch17.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
```

Associates an ObjectOutputStream with a file on disk.

**Fig. 17.17** | Sequential file created using ObjectOutputStream. (Part 1 of 5.)

---

```
23     catch ( IOException ioException )
24     {
25         System.err.println( "Error opening file." );
26     } // end catch
27 } // end method openFile
28
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
```

---

**Fig. 17.17** | Sequential file created using `ObjectOutputStream`. (Part 2 of 5.)

```
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( accountNumber,
63                 firstName, lastName, balance );
64             output.writeObject( record ); // output record
65         } // end if
```

← Outputs an object to the file on disk.

**Fig. 17.17** | Sequential file created using ObjectOutputStream. (Part 3 of 5.)

---

```
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
```

---

**Fig. 17.17** | Sequential file created using ObjectOutputStream. (Part 4 of 5.)

---

```
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
```

---

**Fig. 17.17** | Sequential file created using `ObjectOutputStream`. (Part 5 of 5.)

---

```
1 // Fig. 17.18: 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
```

---

**Fig. 17.18** | Testing class CreateSequentialFile. (Part 1 of 2.)

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
```

**Fig. 17.18** | Testing class CreateSequentialFile. (Part 2 of 2.)



## Common Programming Error 17.2

*It's a logic error to open an existing file for output when, in fact, you wish to preserve the file. Class `FileOutputStream` provides an overloaded constructor that enables you to open a file and append data to the end of the file. This will preserve the contents of the file.*

## 17.6.2 Reading and Deserializing Data from a Sequential-Access File

- ▶ The program in Figs. 17.19–17.20 reads records from a file created by the program in Section 17.6.1 and displays the contents.

```
1 // Fig. 17.19: ReadSequentialFile.java
2 // Reading a file of objects sequentially with ObjectInputStream
3 // and displaying 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.ch17.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
```

Associates an ObjectInputStream  
with a file on disk.

**Fig. 17.19** | Reading a file of objects sequentially with ObjectInputStream and displaying each record. (Part 1 of 4.)

```
23     catch ( IOException ioException )
24     {
25         System.err.println( "Error opening file." );
26     } // end catch
27 } // end method openFile
28
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         }

```

Reads one object from the file and casts it to the appropriate type for processing in the program.

**Fig. 17.19** | Reading a file of objects sequentially with `ObjectInputStream` and displaying each record. (Part 2 of 4.)

---

```
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 eofException )
49 {
50     return; // end of file was reached
51 } // end catch
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
```

---

**Fig. 17.19** | Reading a file of objects sequentially with `ObjectInputStream` and displaying each record. (Part 3 of 4.)

---

```
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
```

---

**Fig. 17.19** | Reading a file of objects sequentially with `ObjectInputStream` and displaying each record. (Part 4 of 4.)

```
1 // Fig. 17.20: ReadSequentialFileTest.java
2 // Testing 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

**Fig. 17.20** | Testing class ReadSequentialFile.

## 17.6.2 Reading and Deserializing Data from a Sequential-Access File (cont.)

- ▶ `ObjectInputStream` method `readObject` reads an `Object` from a file.
- ▶ Method `readObject` throws an **`EOFException`** if an attempt is made to read beyond the end of the file.
- ▶ Method `readObject` throws a `ClassNotFoundException` if the class for the object being read cannot be located.

## 17.8 Opening Files with `JFileChooser`

- ▶ Class `JFileChooser` displays a dialog that enables the user to easily select files or directories.

---

```
1 // Fig. 17.21: FileDemonstration.java
2 // Demonstrating JFileChooser.
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();
```

---

**Fig. 17.21** | Demonstrating JFileChooser. (Part I of 5.)

```
25
26 // add outputArea to scrollPane
27 scrollPane = new JScrollPane( outputArea );
28
29 add( scrollPane, BorderLayout.CENTER ); // add scrollPane to GUI
30
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 or directory name
38 private File getFileOrDirectory()
39 {
40 // display file dialog, so user can choose file or directory to open
41 JFileChooser fileChooser = new JFileChooser();
42 fileChooser.setSelectionMode(
43     JFileChooser.FILES_AND_DIRECTORIES );
44
45 int result = fileChooser.showOpenDialog( this );
46
```

Creates a  
JFileChooser for  
selecting files and  
directories.

Displays the  
JFileChooser  
centered over the  
parent window.

**Fig. 17.21** | Demonstrating JFileChooser. (Part 2 of 5.)

---

```
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(); // get File
52
53 // display error if invalid
54 if ( ( fileName == null ) || ( fileName.getName().equals( "" ) ) )
55 {
56     JOptionPane.showMessageDialog( this, "Invalid Name",
57         "Invalid Name", JOptionPane.ERROR_MESSAGE );
58     System.exit( 1 );
59 } // end if
60
61 return fileName;
62 } // end method getFile
63
64 // display information about file or directory user specifies
65 public void analyzePath()
66 {
67     // create File object based on user input
68     File name = getFileOrDirectory();
69
```

Retrieves the selected  
file or directory name.

---

**Fig. 17.21** | Demonstrating JFileChooser. (Part 3 of 5.)

---

```
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
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
```

---

**Fig. 17.21** | Demonstrating JFileChooser. (Part 4 of 5.)

---

```
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
```

---

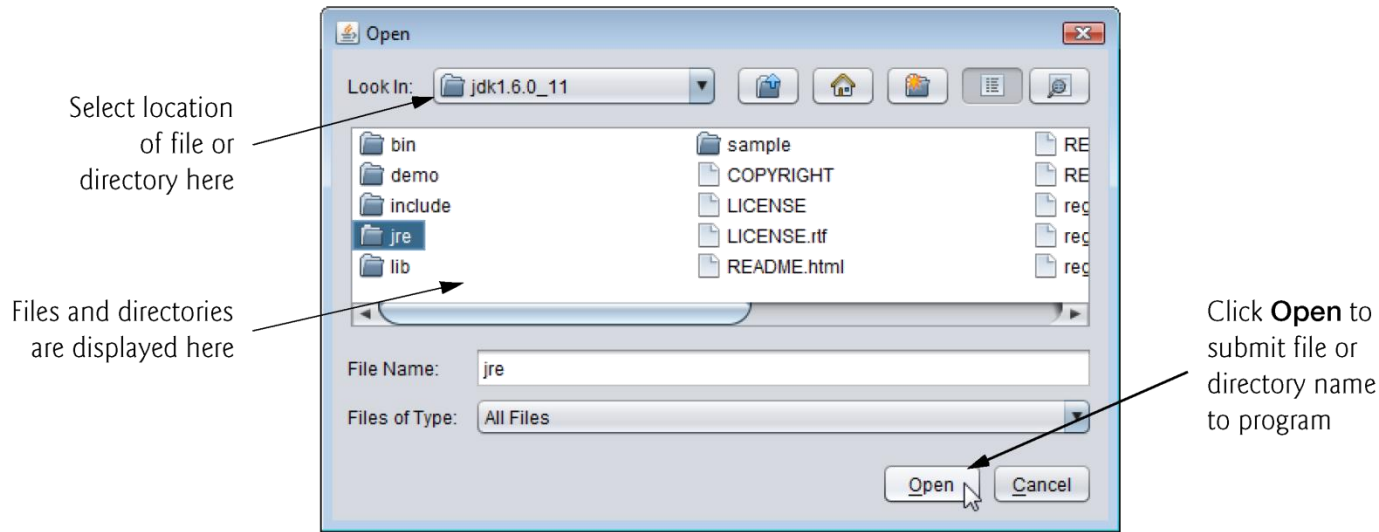
**Fig. 17.21** | Demonstrating JFileChooser. (Part 5 of 5.)

---

```
1 // Fig. 17.22: FileDemonstrationTest.java
2 // Testing class FileDemonstration.
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
```

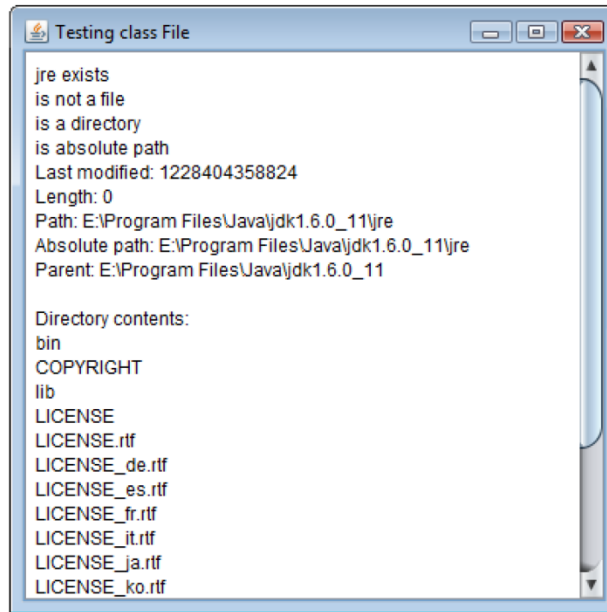
---

**Fig. 17.22** | Testing class FileDemonstration. (Part I of 3.)



---

**Fig. 17.22** | Testing class FileDemonstration. (Part 2 of 3.)



**Fig. 17.22** | Testing class FileDemonstration. (Part 3 of 3.)

## 17.9 Opening Files with JFileChooser (cont.)

- ▶ JFileChooser method `setFileSelectionMode` specifies what the user can select from the fileChooser.
- ▶ JFileChooser static constant `FILES_AND_DIRECTORIES` indicates that files and directories can be selected.
- ▶ Other static constants include `FILES_ONLY` (the default) and `DIRECTORIES_ONLY`.
- ▶ Method `showOpenDialog` displays a JFileChooser dialog titled Open.
- ▶ A JFileChooser dialog is a modal dialog.
- ▶ Method `showOpenDialog` returns an integer specifying which button (**Open** or **Cancel**) the user clicked to close the dialog.
- ▶ JFileChooser method `getSelectedFile` returns the selected file as a `File` object.

# End of Part I

- ▶ Reading
  - Chapter 17