

# **Chapter 15**

# **Files, Input/Output Streams, NIO and**

# **XML Serialization**

Java How to Program, 11/e  
Questions? E-mail [paul.deitel@deitel.com](mailto:paul.deitel@deitel.com)

# OBJECTIVES

In this chapter you'll:

- Create, read, write and update files.
- Retrieve information about files and directories using features of the NIO.2 APIs.
- Learn the differences between text files and binary files.
- Use class **Formatter** to output text to a file.
- Use class **Scanner** to input text from a file.

## OBJECTIVES (cont.)

- Use sequential file processing to develop a real-world credit-inquiry program.
- Write objects to and read objects from a file using XML serialization and the JAXB (Java Architecture for XML Binding) APIs.
- Use a `JFileChooser` dialog to allow users to select files or directories on disk.
- Optionally use `java.io` interfaces and classes to perform byte-based and character-based input and output.

# OUTLINE

## 15.1 Introduction

## 15.2 Files and Streams

## 15.3 Using NIO Classes and Interfaces to Get File and Directory Information

## 15.4 Sequential Text Files

### 15.4.1 Creating a Sequential Text File

### 15.4.2 Reading Data from a Sequential Text File

### 15.4.3 Case Study: A Credit-Inquiry Program

### 15.4.4 Updating Sequential Files

## OUTLINE (cont.)

### 15.5 XML Serialization

- 15.5.1 Creating a Sequential File Using XML Serialization
- 15.5.2 Reading and Deserializing Data from a Sequential File

### 15.6 FileChooser and DirectoryChooser dialogs

### 15.7 (Optional) Additional `java.io` Classes

- 15.7.1 Interfaces and Classes for Byte-Based Input and Output
- 15.7.2 Interfaces and Classes for Character-Based Input and Output

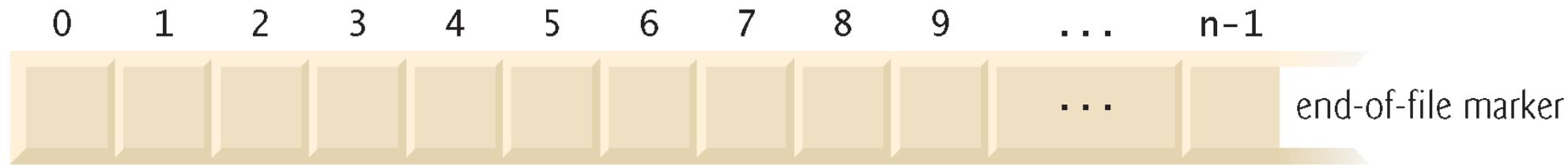
### 15.8 Wrap-Up

## 15.1 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, flash drives, DVDs and more.
- ▶ Data maintained in files is **persistent data** because it exists beyond the duration of program execution.

## 15.2 Files and Streams

- ▶ Java views each file as a sequential **stream of bytes** (Fig. 15.1).
- ▶ 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. 15.1** | Java's view of a file of  $n$  bytes.

## 15.2 Files and Streams (cont.)

- ▶ File streams can be used to input and output data as bytes or characters.
  - **Byte-based streams** output and input data in its *binary* format—a `char` is two bytes, an `int` is four bytes, a `double` is eight bytes, etc.
  - **Character-based streams** output and input data as a *sequence of characters* in which every character is two bytes—the number of bytes for a given value depends on the number of characters in that value.
- ▶ Files 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.

## 15.2 Files and Streams (cont.)

- ▶ A Java program **opens** a file by creating an object and associating a stream of bytes or characters with it.
  - Can also associate streams with different devices.
- ▶ Java creates three stream objects when a program begins executing
  - `System.in` (standard input stream) object normally inputs bytes from the keyboard
  - Object `System.out` (the standard output stream object) normally outputs character data to the screen
  - Object `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.

## 15.2 Files and Streams (cont.)

- ▶ Java programs perform file processing by using classes from package **java.io** and the subpackages of **java.nio**.
- ▶ 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**.

## 15.2 Files and Streams (cont.)

### *Java SE 8 Adds Another Type of Stream*

- ▶ Chapter 17, Java SE 8 Lambdas and Streams, introduces a new type of stream that's used to process collections of elements (like arrays and `ArrayLists`), rather than the streams of bytes we discuss in this chapter's file-processing examples.

## 15.3 Using NIO Classes and Interfaces to Get File and Directory Information

- ▶ Interfaces Path and DirectoryStream and classes Paths and Files (all from package `java.nio.file`) are useful for retrieving information about files and directories on disk:
  - **Path** interface—Objects of classes that implement this interface represent the location of a file or directory. Path objects do not open files or provide any file-processing capabilities.
  - **Paths** class—Provides static methods used to get a Path object representing a file or directory location.

## 15.3 Using NIO Classes and Interfaces to Get File and Directory Information (Cont.)

- **Files** class—Provides static methods for common file and directory manipulations, such as copying files; creating and deleting files and directories; getting information about files and directories; reading the contents of files; getting objects that allow you to manipulate the contents of files and directories; and more
- **DirectoryStream** interface—Objects of classes that implement this interface enable a program to iterate through the contents of a directory.

## 15.3 Using NIO Classes and Interfaces to Get File and Directory Information (Cont.)

- ▶ A file or directory's path specifies its location on disk. The path includes some or all of the directories leading to the file or directory.
- ▶ An **absolute path** contains *all* directories, starting with the **root directory**, that lead to a specific file or directory.
- ▶ Every file or directory on a particular disk drive has the *same* root directory in its path.
- ▶ A **relative path** is “relative” to another directory—for example, a path relative to the directory in which the application began executing.

## 15.3 Using NIO Classes and Interfaces to Get File and Directory Information (Cont.)

- ▶ An overloaded version of Files static method get uses a URI object to locate the file or directory.
- ▶ A **Uniform Resource Identifier (URI)** is a more general form of the **Uniform Resource Locators (URLs)** that are used to locate websites.
- ▶ On Windows platforms, the URI
  - `file://C:/data.txt`
- ▶ identifies the file `data.txt` stored in the root directory of the C: drive. On UNIX/Linux platforms, the URI
  - `file:/home/student/data.txt`
- ▶ identifies the file `data.txt` stored in the home directory of the user student.

## 15.3 Using NIO Classes and Interfaces to Get File and Directory Information (Cont.)

- ▶ Figure 15.2 prompts the user to enter a file or directory name, then uses classes `Paths`, `Path`, `Files` and `DirectoryStream` to output information about that file or directory.
- ▶ A **separator character** is used to separate directories and files in a path.
  - On a Windows computer, the *separator character* is a backslash (\).
  - On a Linux or Mac OS X system, it's a forward slash (/).
- ▶ Java processes both characters identically in a path name.

---

```
1 // Fig. 15.2: FileAndDirectoryInfo.java
2 // File class used to obtain file and directory information.
3 import java.io.IOException;
4 import java.nio.file.DirectoryStream;
5 import java.nio.file.Files;
6 import java.nio.file.Path;
7 import java.nio.file.Paths;
8 import java.util.Scanner;
9
10 public class FileAndDirectoryInfo {
11     public static void main(String[] args) throws IOException {
12         Scanner input = new Scanner(System.in);
13
14         System.out.println("Enter file or directory name:");
15
16         // create Path object based on user input
17         Path path = Paths.get(input.nextLine());
```

---

**Fig. 15.2** | File class used to obtain file and directory information. (Part I of 5.)

---

```
18
19      if (Files.exists(path)) { // if path exists, output info about it
20          // display file (or directory) information
21          System.out.printf("%n%s exists%n", path.getFileName());
22          System.out.printf("%s a directory%n",
23              Files.isDirectory(path) ? "Is" : "Is not");
24          System.out.printf("%s an absolute path%n",
25              path.isAbsolute() ? "Is" : "Is not");
26          System.out.printf("Last modified: %s%n",
27              Files.getLastModifiedTime(path));
28          System.out.printf("Size: %s%n", Files.size(path));
29          System.out.printf("Path: %s%n", path);
30          System.out.printf("Absolute path: %s%n", path.toAbsolutePath());
31
```

---

**Fig. 15.2** | File class used to obtain file and directory information. (Part 2 of 5.)

---

```
32     if (Files.isDirectory(path)) { // output directory listing
33         System.out.printf("%nDirectory contents:%n");
34
35         // object for iterating through a directory's contents
36         DirectoryStream<Path> directoryStream =
37             Files.newDirectoryStream(path);
38
39         for (Path p : directoryStream) {
40             System.out.println(p);
41         }
42     }
43     else { // not file or directory, output error message
44         System.out.printf("%s does not exist%n", path);
45     }
46 }
47 } // end main
48 } // end class FileInfo
```

---

**Fig. 15.2** | File class used to obtain file and directory information. (Part 3 of 5.)

```
Enter file or directory name:
```

```
c:\examples\ch15
```

```
ch15 exists
```

```
Is a directory
```

```
Is an absolute path
```

```
Last modified: 2013-11-08T19:50:00.838256Z
```

```
Size: 4096
```

```
Path: c:\examples\ch15
```

```
Absolute path: c:\examples\ch15
```

```
Directory contents:
```

```
C:\examples\ch15\fig15_02
```

```
C:\examples\ch15\fig15_12_13
```

```
C:\examples\ch15\SerializationApps
```

```
C:\examples\ch15\TextFileApps
```

**Fig. 15.2** | File class used to obtain file and directory information. (Part 4 of 5.)

```
Enter file or directory name:  
C:\examples\ch15\fig15_02\FileAndDirectoryInfo.java  
  
FileAndDirectoryInfo.java exists  
Is not a directory  
Is an absolute path  
Last modified: 2013-11-08T19:59:01.848Z  
Size: 2952  
Path: C:\examples\ch15\fig15_02\FileAndDirectoryInfo.java  
Absolute path: C:\examples\ch15\fig15_02\FileAndDirectoryInfo.java
```

**Fig. 15.2** | File class used to obtain file and directory information. (Part 5 of 5.)



## Error-Prevention Tip 15.1

Once you've confirmed that a Path exists, it's still possible that the methods demonstrated in Fig. 15.2 will throw `IOExceptions`. For example, the file or directory represented by the Path could be deleted from the system after the call to `Files` method `exists` and before the other statements in lines 21–42 execute. Industrial strength file- and directory-processing programs require extensive exception handling to deal with such possibilities.



## Good Programming Practice 15.1

When building **Strings** that represent path information, use **File.separator** to obtain the local computer's proper separator character rather than explicitly using / or \. This constant is a **String** consisting of one character—the proper separator for the system.



## Common Programming Error 15.1

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.

## 15.4 Sequential Text Files

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

## 15.4.1 Creating a Sequential Text File

- ▶ Java imposes no structure on a file
  - Notions such as records do not exist as part of the Java language.
  - You must structure files to meet the requirements of your applications.

## 15.4.1 Creating a Sequential 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. 15.3: 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 public class CreateTextFile {
11     public static void main(String[] args) {
12         // open clients.txt, output data to the file then close clients.txt
13         try (Formatter output = new Formatter("clients.txt")) {
14             Scanner input = new Scanner(System.in);
15             System.out.printf("%s%n%s%n? ",
16                 "Enter account number, first name, last name and balance.",
17                 "Enter end-of-file indicator to end input.");
18     }
}
```

---

**Fig. 15.3** | Writing data to a sequential text file with class `Formatter`. (Part 1 of 3.)

---

```
19     while (input.hasNext()) { // loop until end-of-file indicator
20         try {
21             // output new record to file; assumes valid input
22             output.format("%d %s %s %.2f%n", input.nextInt(),
23                         input.next(), input.next(), input.nextDouble());
24         }
25         catch (NoSuchElementException elementException) {
26             System.out.println("Invalid input. Please try again.");
27             input.nextLine(); // discard input so user can try again
28         }
29
30         System.out.print("?");
31     }
32     catch (SecurityException | FileNotFoundException |
33             FormatterClosedException e) {
34         e.printStackTrace();
35     }
36 }
37 }
38 }
```

---

**Fig. 15.3** | Writing data to a sequential text file with class `Formatter`. (Part 2 of 3.)

Enter account number, first name, last name and balance.  
Enter end-of-file indicator to end input.

```
? 100 Bob Blue 24.98
? 200 Steve Green -345.67
? 300 Pam White 0.00
? 400 Sam Red -42.16
? 500 Sue Yellow 224.62
? ^Z
```

**Fig. 15.3** | Writing data to a sequential text file with class Formatter. (Part 3 of 3.)

## 15.4.1 Creating a Sequential 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
macOS and Linux	$\langle Enter \rangle \langle Ctrl \rangle d$
Windows	$\langle Ctrl \rangle z$

**Fig. 15.4** | End-of-file key combinations.

## 15.4.1 Creating a Sequential 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`
- ▶ 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.

## Sample data

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

**Fig. 15.5** | Sample data for the program in Fig. 15.3.

## 15.4.2 Reading Data from a Sequential Text File

- ▶ The application (Fig. 15.6) reads records from the file "clients.txt" created by the application of Section 15.4.1 and displays the record contents.  
If a Scanner is closed before data is input, an **IllegalStateException** occurs.

---

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

---

**Fig. 15.6** | Sequential file reading using a Scanner. (Part I of 2.)

```
18     // read record from file
19     while (input.hasNext()) { // while there is more to read
20         // display record contents
21         System.out.printf("%-10d%-12s%-12s%10.2f%n", input.nextInt(),
22                           input.next(), input.next(), input.nextDouble());
23     }
24 }
25 catch (IOException | NoSuchElementException |
26       IllegalStateException e) {
27     e.printStackTrace();
28 }
29 }
30 }
```

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

**Fig. 15.6** | Sequential file reading using a Scanner. (Part 2 of 2.)

## 15.4.3 Case Study: A Credit-Inquiry Program

- ▶ To retrieve data sequentially from a file, programs start from the beginning of the file and read *all* the data consecutively until the desired information is found.
- ▶ It might be necessary to process the file sequentially several times (from the beginning of the file) during the execution of a program.
- ▶ Class Scanner does *not* allow repositioning to the beginning of the file.
  - The program must *close* the file and *reopen* it.

---

```
1 // Fig. 15.7: MenuOption.java
2 // enum type for the credit-inquiry program's options.
3 public enum MenuOption {
4     // declare contents of enum type
5     ZERO_BALANCE(1),
6     CREDIT_BALANCE(2),
7     DEBIT_BALANCE(3),
8     END(4);
9
10    private final int value; // current menu option
11
12    // constructor
13    private MenuOption(int value) {this.value = value;}
14 }
```

---

**Fig. 15.7** | enum type for the credit-inquiry program's menu options.

---

```
1 // Fig. 15.8: 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.IOException;
6 import java.lang.IllegalStateException;
7 import java.nio.file.Paths;
8 import java.util.NoSuchElementException;
9 import java.util.Scanner;
10
11 public class CreditInquiry {
12     private final static MenuOption[] choices = MenuOption.values();
13
14     public static void main(String[] args) {
15         Scanner input = new Scanner(System.in);
16
17         // get user's request (e.g., zero, credit or debit balance)
18         MenuOption accountType = getRequest(input);
19
```

---

**Fig. 15.8** | Credit-inquiry program. (Part I of 7.)

---

```
20     while (accountType != MenuOption.END) {
21         switch (accountType) {
22             case ZERO_BALANCE:
23                 System.out.printf("%nAccounts with zero balances:%n");
24                 break;
25             case CREDIT_BALANCE:
26                 System.out.printf("%nAccounts with credit balances:%n");
27                 break;
28             case DEBIT_BALANCE:
29                 System.out.printf("%nAccounts with debit balances:%n");
30                 break;
31         }
32
33         readRecords(accountType);
34         accountType = getRequest(input); // get user's request
35     }
36 }
```

---

**Fig. 15.8** | Credit-inquiry program. (Part 2 of 7.)

---

```
37
38     // obtain request from user
39     private static MenuOption getRequest(Scanner input) {
40         int request = 4;
41
42         // display request options
43         System.out.printf("%nEnter request%n%s%n%s%n%s%n%s%n",
44                         " 1 - List accounts with zero balances",
45                         " 2 - List accounts with credit balances",
46                         " 3 - List accounts with debit balances",
47                         " 4 - Terminate program");
48
49         try {
50             do { // input user request
51                 System.out.printf("%n? ");
52                 request = input.nextInt();
53             } while ((request < 1) || (request > 4));
54         }
55         catch (NoSuchElementException noSuchElementException) {
56             System.err.println("Invalid input. Terminating.");
57         }
58
59         return choices[request - 1]; // return enum value for option
60     }
```

---

**Fig. 15.8** | Credit-inquiry program. (Part 3 of 7.)

---

```
61
62     // read records from file and display only records of appropriate type
63     private static void readRecords(MenuOption accountType) {
64         // open file and process contents
65         try (Scanner input = new Scanner(Paths.get("clients.txt"))) {
66             while (input.hasNext()) { // more data to read
67                 int accountNumber = input.nextInt();
68                 String firstName = input.next();
69                 String lastName = input.next();
70                 double balance = input.nextDouble();
71
72                 // if proper account type, display record
73                 if (shouldDisplay(accountType, balance)) {
74                     System.out.printf("%-10d%-12s%-12s%10.2f%n", accountNumber,
75                                     firstName, lastName, balance);
76                 }
77                 else {
78                     input.nextLine(); // discard the rest of the current record
79                 }
80             }
81         }
```

---

**Fig. 15.8** | Credit-inquiry program. (Part 4 of 7.)

---

```
82     catch (NoSuchElementException | IllegalStateException |  
83         IOException e) {  
84         System.err.println("Error processing file. Terminating.");  
85         System.exit(1);  
86     }  
87 }  
88  
89 // use record type to determine if record should be displayed  
90 private static boolean shouldDisplay(  
91     MenuOption option, double balance) {  
92     if ((option == MenuOption.CREDIT_BALANCE) && (balance < 0)) {  
93         return true;  
94     }  
95     else if ((option == MenuOption.DEBIT_BALANCE) && (balance > 0)) {  
96         return true;  
97     }  
98     else if ((option == MenuOption.ZERO_BALANCE) && (balance == 0)) {  
99         return true;  
100    }  
101  
102    return false;  
103 }  
104 }
```

---

**Fig. 15.8** | Credit-inquiry program. (Part 5 of 7.)

Enter request

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

? 1

Accounts with zero balances:

300	Pam	White	0.00
-----	-----	-------	------

Enter request

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

? 2

Accounts with credit balances:

200	Steve	Green	-345.67
400	Sam	Red	-42.16

**Fig. 15.8** | Credit-inquiry program. (Part 6 of 7.)

Enter request

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

? 3

Accounts with debit balances:

100	Bob	Blue	24.98
500	Sue	Yellow	224.62

Enter request

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

? 4

**Fig. 15.8** | Credit-inquiry program. (Part 7 of 7.)

## 15.4.4 Updating Sequential Files

- ▶ The data in many sequential files cannot be modified without the risk of destroying other data in the file.
- ▶ 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 file are not usually updated in place. Instead, the entire file is rewritten.
- ▶ Rewriting the entire file is uneconomical to update just one record, but reasonable if a substantial number of records need to be updated.

## 15.5 XML Serialization

- ▶ Sometimes we want to write an entire object to or read an entire object from a file or over a network connection
- ▶ XML (eXtensible Markup Language) is a widely used language for describing data
- ▶ APIs for manipulating objects as XML are built into Java SE

## 15.5 XML Serialization

- ▶ We'll manipulate objects using JAXB (Java Architecture for XML Binding)
- ▶ JAXB enables you to perform **XML serialization**—which JAXB refers to as **marshaling**
- ▶ A **serialized** object is represented by XML that includes the object's data
- ▶ After a serialized object has been written into a file, it can be read from the file and **deserialized**—that is, the XML that represents the object and its data can be used to recreate the object in memory\

## 15.5.1 Creating a Sequential File Using XML Serialization

- ▶ The serialization we show in this section is performed with character-based streams, so the result will be a text file that you can view in standard text editors.
- ▶ Class **Account** (Fig. 15.9) encapsulates the client record information used by the serialization examples
- ▶ Contains **private** instance variables **account**, **firstName**, **lastName** and **balance** (lines 4–7) and **set** and **get** methods for accessing these instance variables

---

```
1 // Fig. 15.9: Account.java
2 // Account class for storing records as objects.
3 public class Account {
4     private int accountNumber;
5     private String firstName;
6     private String lastName;
7     private double balance;
8
9     // initializes an Account with default values
10    public Account() {this(0, "", "", 0.0);}
11
12    // initializes an Account with provided values
13    public Account(int accountNumber, String firstName,
14                   String lastName, double balance) {
15        this.accountNumber = accountNumber;
16        this.firstName = firstName;
17        this.lastName = lastName;
18        this.balance = balance;
19    }
20}
```

---

**Fig. 15.9** | Account class for storing records as objects. (Part I of 3.)

---

```
21 // get account number
22 public int getAccountNumber() {return accountNumber;}
23
24 // set account number
25 public void setAccountNumber(int accountNumber)
26     {this.accountNumber = accountNumber;}
27
28 // get first name
29 public String getFirstName() {return firstName;}
30
31 // set first name
32 public void setFirstName(String firstName)
33     {this.firstName = firstName;}
34
```

---

**Fig. 15.9** | Account class for storing records as objects. (Part 2 of 3.)

---

```
35 // get last name
36 public String getLastName() {return lastName;}
37
38 // set last name
39 public void setLastName(String lastName) {this.lastName = lastName;}
40
41 // get balance
42 public double getBalance() {return balance;}
43
44 // set balance
45 public void setBalance(double balance) {this.balance = balance;}
46 }
```

---

**Fig. 15.9** | Account class for storing records as objects. (Part 3 of 3.)

## 15.5.1 Creating a Sequential File Using XML Serialization (cont.)

### *Plain Old Java Objects*

- ▶ JAXB works with **POJOs (plain old Java objects)**—no special superclasses or interfaces are required for XML-serialization support
- ▶ By default, JAXB serializes only an object's `public` instance variables and `public read-write` properties.
- ▶ Class `Account` defines a `read-write` properties `accountNumber`, `firstName`, `lastName` and `balance`
- ▶ The class must also provide a `public` default or no-argument constructor to recreate the objects when they're read from the file

## 15.5.1 Creating a Sequential File Using XML Serialization (cont.)

### Declaring Class Accounts

- ▶ Fig. 15.11 stores Account objects in a `List<Account>`, then serializes the entire `List` into a file with one operation
- ▶ To serialize a `List`, it must be defined as an instance variable of a class.
- ▶ For that reason, we encapsulate the `List<Account>` in class `Accounts` (Fig. 15.10).
- ▶ Lines 9–10 declare and initialize the `List<Account>` instance variable

## 15.5.1 Creating a Sequential File Using XML Serialization (cont.)

- ▶ Lines 9–10 declare and initialize the `List<Account>` instance variable `accounts`
- ▶ JAXB enables you to customize many aspects of XML serialization, such as serializing a private instance variable or a read-only property
- ▶ Annotation `@XMLElement` (line 9; package `javax.xml.bind.annotation`) indicates that the private instance variable should be serialized
- ▶ The annotation is required because the instance variable is not `public` and there's no corresponding `public` read–write property.

## 15.5.1 Creating a Sequential File Using XML Serialization (cont.)

- ▶ To open the file, lines 14–15 call Files static method **newBufferedWriter**, which receives a Path specifying the file to open for writing ("clients.xml") and—if the file exists—returns a **BufferedWriter** that class JAXB will use to write text to the file
- ▶ Existing files that are opened for output in this manner are *truncated*
- ▶ Line 20 creates the **Accounts** object that contains the **List<Account>**. Lines 26–41 input each record, create an **Account** object (lines 29–30) and add to the **List** (line 33).

---

```
1 // Fig. 15.10: Accounts.java
2 // Maintains a List<Account>
3 import java.util.ArrayList;
4 import java.util.List;
5 import javax.xml.bind.annotation.XmlElement;
6
7 public class Accounts {
8     // @XmlElement specifies XML element name for each object in the List
9     @XmlElement(name="account")
10    private List<Account> accounts = new ArrayList<>(); // stores Accounts
11
12    // returns the List<Accounts>
13    public List<Account> getAccounts() {return accounts;}
14 }
```

---

**Fig. 15.10** | Account class for serializable objects.

---

```
1 // Fig. 15.11: CreateSequentialFile.java
2 // Writing objects to a file with JAXB and BufferedWriter.
3 import java.io.BufferedWriter;
4 import java.io.IOException;
5 import java.nio.file.Files;
6 import java.nio.file.Paths;
7 import java.util.NoSuchElementException;
8 import java.util.Scanner;
9 import javax.xml.bind.JAXB;
10
11 public class CreateSequentialFile {
12     public static void main(String[] args) {
13         // open clients.xml, write objects to it then close file
14         try(BufferedWriter output =
15             Files.newBufferedWriter(Paths.get("clients.xml"))) {
16
17             Scanner input = new Scanner(System.in);
18
```

---

**Fig. 15.11** | Writing objects to a file with JAXB and BufferedWriter. (Part 1 of 3.)

---

```
19     // stores the Accounts before XML serialization
20     Accounts accounts = new Accounts();
21
22     System.out.printf("%s%n%s%n? ",
23                         "Enter account number, first name, last name and balance.",
24                         "Enter end-of-file indicator to end input.");
25
26     while (input.hasNext()) { // loop until end-of-file indicator
27         try {
28             // create new record
29             Account record = new Account(input.nextInt(),
30                                           input.next(), input.next(), input.nextDouble());
31
32             // add to AccountList
33             accounts.getAccounts().add(record);
34         }
35         catch (NoSuchElementException elementException) {
36             System.err.println("Invalid input. Please try again.");
37             input.nextLine(); // discard input so user can try again
38         }
39
40         System.out.print("?");
41     }
```

---

**Fig. 15.11** | Writing objects to a file with JAXB and BufferedWriter. (Part 2 of 3.)

```
42
43     // write AccountList's XML to output
44     JAXB.marshal(accounts, output);
45 }
46 catch (IOException ioException) {
47     System.err.println("Error opening file. Terminating.");
48 }
49 }
50 }
```

Enter account number, first name, last name and balance.  
Enter end-of-file indicator to end input.

```
? 100 Bob Blue 24.98
? 200 Steve Green -345.67
? 300 Pam White 0.00
? 400 Sam Red -42.16
? 500 Sue Yellow 224.62
? ^Z
```

**Fig. 15.11** | Writing objects to a file with JAXB and BufferedWriter. (Part 3 of 3.)

## 15.5.1 Creating a Sequential File Using XML Serialization (cont.)

- ▶ When the user enters the end-of-file indicator to terminate input, line 44 uses JAXB static method **marshal** to serialize as XML the **Accounts** object containing the **List<Account>**
  - The first argument is the object to serialize
  - The second argument to this particular overload of method **marshal** is a **Writer** (package **java.io**) that's used to output the XML—**BufferedWriter** is a subclass of **Writer**
  - The **BufferedWriter** obtained in lines 14–15 outputs the XML to a file
- ▶ One statement writes the *entire* **Accounts** object and all of the objects in its **List<Account>**

## 15.5.1 Creating a Sequential File Using XML Serialization (cont.)

- ▶ Fig. 15.12 shows the contents of the file clients.xml.
- ▶ Though you do not need to know XML to work with this example, note that the XML is human readable.
- ▶ When JAXB serializes an object of a class, it uses the class's name with a lowercase first letter as the corresponding XML element name, so the accounts element represents the Accounts object.

---

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <accounts>
3   <account>
4     <accountNumber>100</accountNumber>
5     <balance>24.98</balance>
6     <firstName>Bob</firstName>
7     <lastName>Blue</lastName>
8   </account>
9   <account>
10    <accountNumber>200</accountNumber>
11    <balance>-345.67</balance>
12    <firstName>Steve</firstName>
13    <lastName>Green</lastName>
14  </account>
15  <account>
16    <accountNumber>300</accountNumber>
17    <balance>0.0</balance>
18    <firstName>Pam</firstName>
19    <lastName>White</lastName>
20  </account>
```

---

**Fig. 15.12** | Contents of clients.xml. (Part 1 of 2.)

---

```
21   <account>
22     <accountNumber>400</accountNumber>
23     <balance>-42.16</balance>
24     <firstName>Sam</firstName>
25     <lastName>Red</lastName>
26   </account>
27   <account>
28     <accountNumber>500</accountNumber>
29     <balance>224.62</balance>
30     <firstName>Sue</firstName>
31     <lastName>Yellow</lastName>
32   </account>
33 </accounts>
```

---

**Fig. 15.12** | Contents of clients.xml. (Part 2 of 2.)

## 15.5.1 Creating a Sequential File Using XML Serialization (cont.)

- ▶ Recall that line 9 in class `Accounts` preceded the `List<Account>` instance variable with the annotation
  - `@XmlElement(name="account")`
- ▶ In addition to enabling JAXB to serialize the instance variable, this annotation specifies the XML element name ("account") used to represent each of the `List`'s `Account` objects in the serialized output.
- ▶ Many other aspects of JAXB XML serialization are customizable. For more details, see
  - <https://docs.oracle.com/javase/tutorial/jaxb/intro/>

## 15.5.2 Reading and Deserializing Data from a Sequential File

- ▶ In this section, we *read serialized data* from a file
- ▶ Fig. 15.13 reads objects from the file, then displays the contents
- ▶ The program opens the file for input by calling `Files static` method **newBufferedReader**, which receives a `Path` specifying the file to open and, if the file exists and no exceptions occur, returns a **Buffered-Reader** for reading from the file.

---

```
1 // Fig. 15.13: ReadSequentialFile.java
2 // Reading a file of XML serialized objects with JAXB and a
3 // BufferedReader and displaying each object.
4 import java.io.BufferedReader;
5 import java.io.IOException;
6 import java.nio.file.Files;
7 import java.nio.file.Paths;
8 import javax.xml.bind.JAXB;
9
10 public class ReadSequentialFile {
11     public static void main(String[] args) {
12         // try to open file for deserialization
13         try(BufferedReader input =
14             Files.newBufferedReader(Paths.get("clients.xml"))) {
15             // unmarshal the file's contents
16             Accounts accounts = JAXB.unmarshal(input, Accounts.class);
17
18             // display contents
19             System.out.printf("%-10s%-12s%-12s%10s%n", "Account",
20                             "First Name", "Last Name", "Balance");
```

---

**Fig. 15.13** | Reading a file of XML serialized objects with JAXB and a BufferedReader and displaying each object. (Part I of 2.)

---

```
21
22     for (Account account : accounts.getAccounts()) {
23         System.out.printf("%-10d%-12s%-12s%10.2f%n",
24             account.getAccountNumber(), account.getFirstName(),
25             account.getLastName(), account.getBalance());
26     }
27 }
28 catch (IOException ioException) {
29     System.err.println("Error opening file.");
30 }
31 }
32 }
```

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

No more records

**Fig. 15.13** | Reading a file of XML serialized objects with JAXB and a BufferedReader and displaying each object. (Part 2 of 2.)

## 15.5.2 Reading and Deserializing Data from a Sequential File (cont.)

- ▶ JAXB static method **unmarshal** reads the contents of `clients.xml` and converts the XML into an `Accounts` object.
- ▶ This overload of `unmarshal` reads XML from a Reader (package `java.io`) and creates an object of the type specified as the second argument
- ▶ The `BufferedReader` reads text from a file.
- ▶ Method `unmarshal`'s second argument is a `Class<T>` object (package `java.lang`) representing the type of the object to create
- ▶ The notation `Accounts.class` is a Java compiler shorthand for
  - `new Class<Accounts>`
- ▶ One statement reads the entire file and recreates the `Accounts` object

## 15.6 FileChooser and DirectoryChooser Dialogs

- ▶ JavaFX classes **FileChooser** and **DirectoryChooser** (package `javafx.stage`) display dialogs that enable the user to select a file or directory, respectively
- ▶ Class **FileChooserTest** (Fig. 15.14) launches the JavaFX application, using the same techniques you learned in Chapters 12–13
- ▶ Class **FileChooserTestController** (Fig. 15.15) responds to the Buttons' events
- ▶ Both event handlers call method `analyzePath` (defined in lines 70–110) to determine whether a `Path` is a file or directory, display information about the `Path` and, if it's a directory, list its contents

---

```
33 // Fig. 15.14: FileChooserTest.java
34 // App to test classes FileChooser and DirectoryChooser.
35 import javafx.application.Application;
36 import javafx.fxml.FXMLLoader;
37 import javafx.scene.Parent;
38 import javafx.scene.Scene;
39 import javafx.stage.Stage;
40
41 public class FileChooserTest extends Application {
42     @Override
43     public void start(Stage stage) throws Exception {
44         Parent root =
45             FXMLLoader.load(getClass().getResource("FileChooserTest.fxml"));
46
47         Scene scene = new Scene(root);
48         stage.setTitle("File Chooser Test"); // displayed in title bar
49         stage.setScene(scene);
50         stage.show();
51     }
52
53     public static void main(String[] args) {
54         launch(args);
55     }
56 }
```

---

**Fig. 15.14** | Demonstrating JFileChooser.

---

```
1 // Fig. 15.15: FileChooserTestController.java
2 // Displays information about a selected file or folder.
3 import java.io.File;
4 import java.io.IOException;
5 import java.nio.file.DirectoryStream;
6 import java.nio.file.Files;
7 import java.nio.file.Path;
8 import java.nio.file.Paths;
9 import javafx.event.ActionEvent;
10 import javafx.fxml.FXML;
11 import javafx.scene.control.Button;
12 import javafx.scene.control.TextArea;
13 import javafx.scene.layout.BorderPane;
14 import javafx.stage.DirectoryChooser;
15 import javafx.stage.FileChooser;
16
```

---

**Fig. 15.15** | Displays information about a selected file or folder. (Part I of II.)

---

```
17 public class FileChooserTestController {  
18     @FXML private BorderPane borderPane;  
19     @FXML private Button selectFileButton;  
20     @FXML private Button selectDirectoryButton;  
21     @FXML private TextArea textArea;  
22  
23     // handles selectFileButton's events  
24     @FXML  
25     private void selectFileButtonPressed(ActionEvent e) {  
26         // configure dialog allowing selection of a file  
27         FileChooser fileChooser = new FileChooser();  
28         fileChooser.setTitle("Select File");  
29  
30         // display files in folder from which the app was launched  
31         fileChooser.setInitialDirectory(new File("."));  
32  
33         // display the FileChooser  
34         File file = fileChooser.showOpenDialog(  
35             borderPane.getScene().getWindow());  
36
```

---

**Fig. 15.15** | Displays information about a selected file or folder. (Part 2 of 11.)

---

```
37     // process selected Path or display a message
38     if (file != null) {
39         analyzePath(file.toPath());
40     }
41     else {
42         textArea.setText("Select file or directory");
43     }
44 }
45
46 // handles selectDirectoryButton's events
47 @FXML
48 private void selectDirectoryButtonPressed(ActionEvent e) {
49     // configure dialog allowing selection of a directory
50     DirectoryChooser directoryChooser = new DirectoryChooser();
51     directoryChooser.setTitle("Select Directory");
52
53     // display folder from which the app was launched
54     directoryChooser.setInitialDirectory(new File("."));
55 }
```

---

**Fig. 15.15** | Displays information about a selected file or folder. (Part 3 of 11.)

---

```
56     // display the FileChooser
57     File file = directoryChooser.showDialog(
58         borderPane.getScene().getWindow());
59
60     // process selected Path or display a message
61     if (file != null) {
62         analyzePath(file.toPath());
63     }
64     else {
65         textArea.setText("Select file or directory");
66     }
67 }
68
69 // display information about file or directory user specifies
70 public void analyzePath(Path path) {
71     try {
72         // if the file or directory exists, display its info
73         if (path != null && Files.exists(path)) {
74             // gather file (or directory) information
75             StringBuilder builder = new StringBuilder();
76             builder.append(String.format("%s:%n", path.getFileName()));
```

---

**Fig. 15.15** | Displays information about a selected file or folder. (Part 4 of 11.)

---

```
77     builder.append(String.format("%s a directory%n",
78         Files.isDirectory(path) ? "Is" : "Is not"));
79     builder.append(String.format("%s an absolute path%n",
80         path.isAbsolute() ? "Is" : "Is not"));
81     builder.append(String.format("Last modified: %s%n",
82         Files.getLastModifiedTime(path)));
83     builder.append(String.format("Size: %s%n", Files.size(path)));
84     builder.append(String.format("Path: %s%n", path));
85     builder.append(String.format("Absolute path: %s%n",
86         path.getAbsolutePath()));

87
88     if (Files.isDirectory(path)) { // output directory listing
89         builder.append(String.format("%nDirectory contents:%n"));

90
91         // object for iterating through a directory's contents
92         DirectoryStream<Path> directoryStream =
93             Files.newDirectoryStream(path);

94
95         for (Path p : directoryStream) {
96             builder.append(String.format("%s%n", p));
97         }
98     }
```

---

**Fig. 15.15** | Displays information about a selected file or folder. (Part 5 of 11.)

---

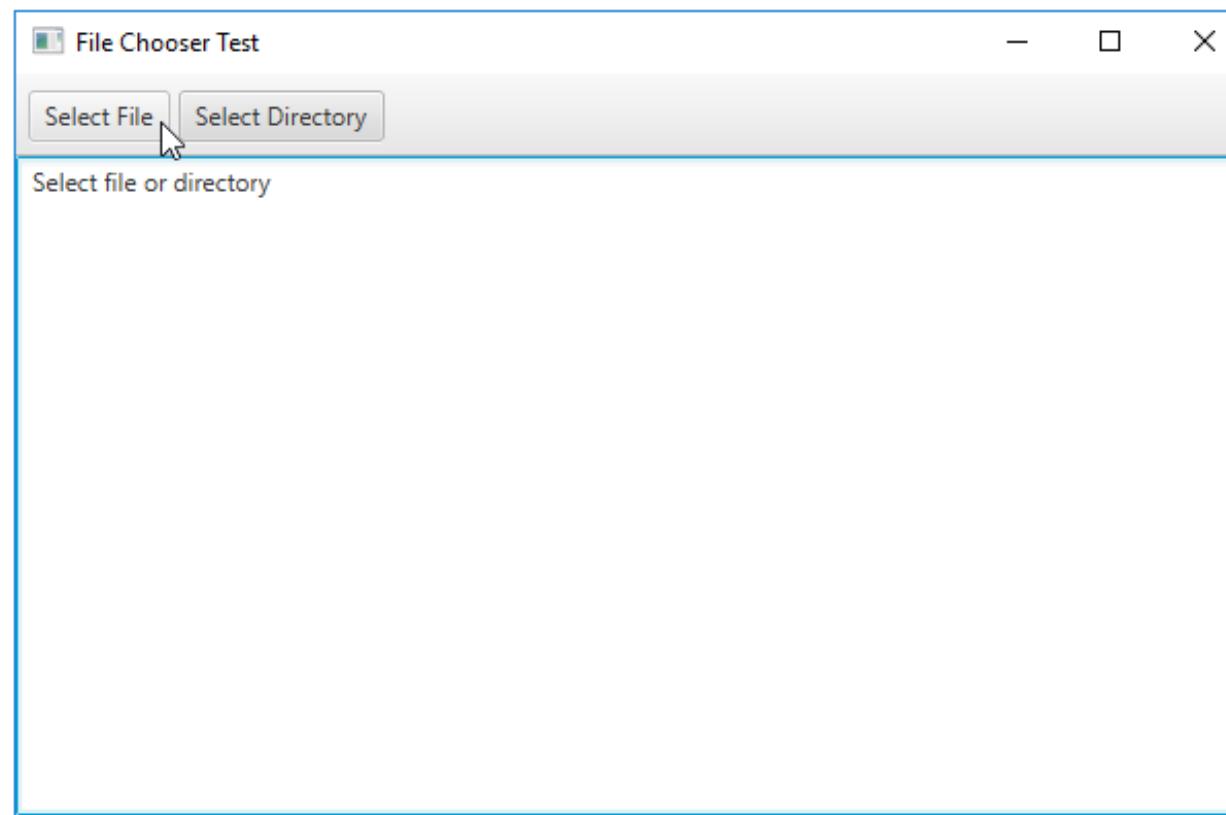
```
99
100         // display file or directory info
101         textArea.setText(builder.toString());
102     }
103     else { // Path does not exist
104         textArea.setText("Path does not exist");
105     }
106 }
107 catch (IOException ioException) {
108     textArea.setText(ioException.toString());
109 }
110 }
111 }
```

---

**Fig. 15.15** | Displays information about a selected file or folder. (Part 6 of 11.)

---

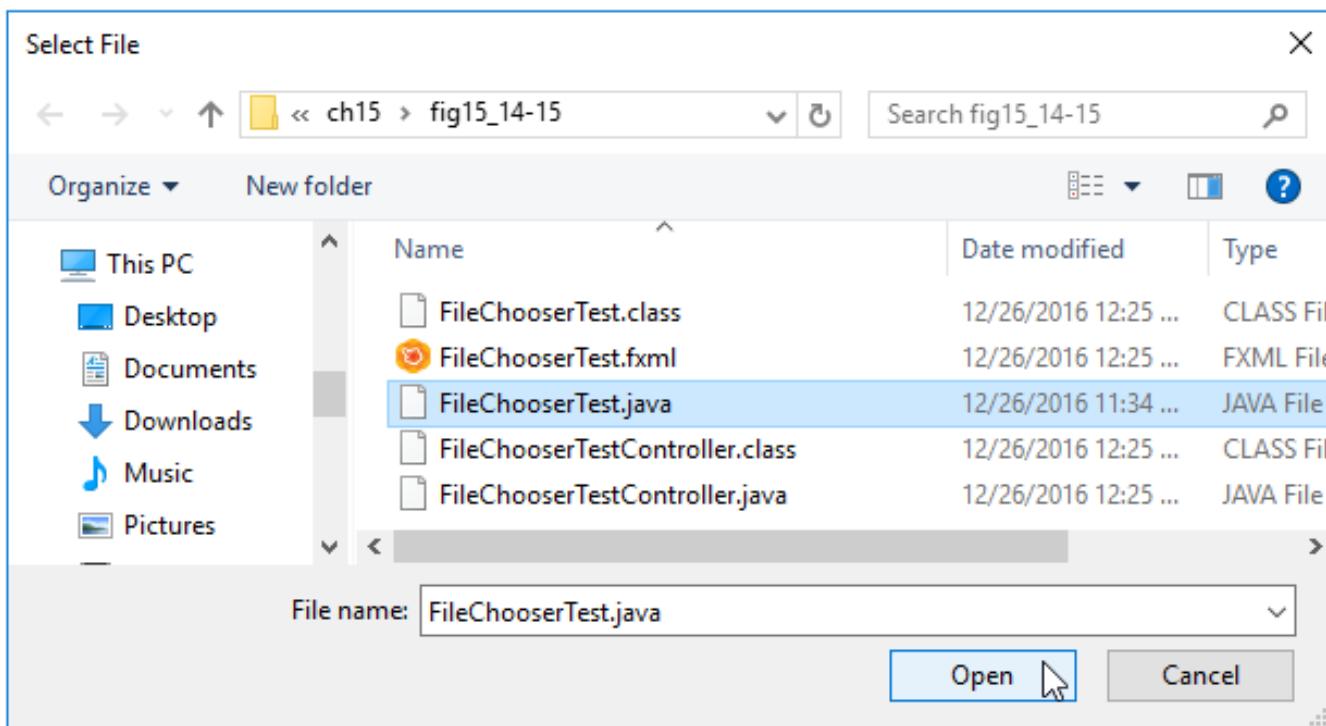
a) Initial app window.



---

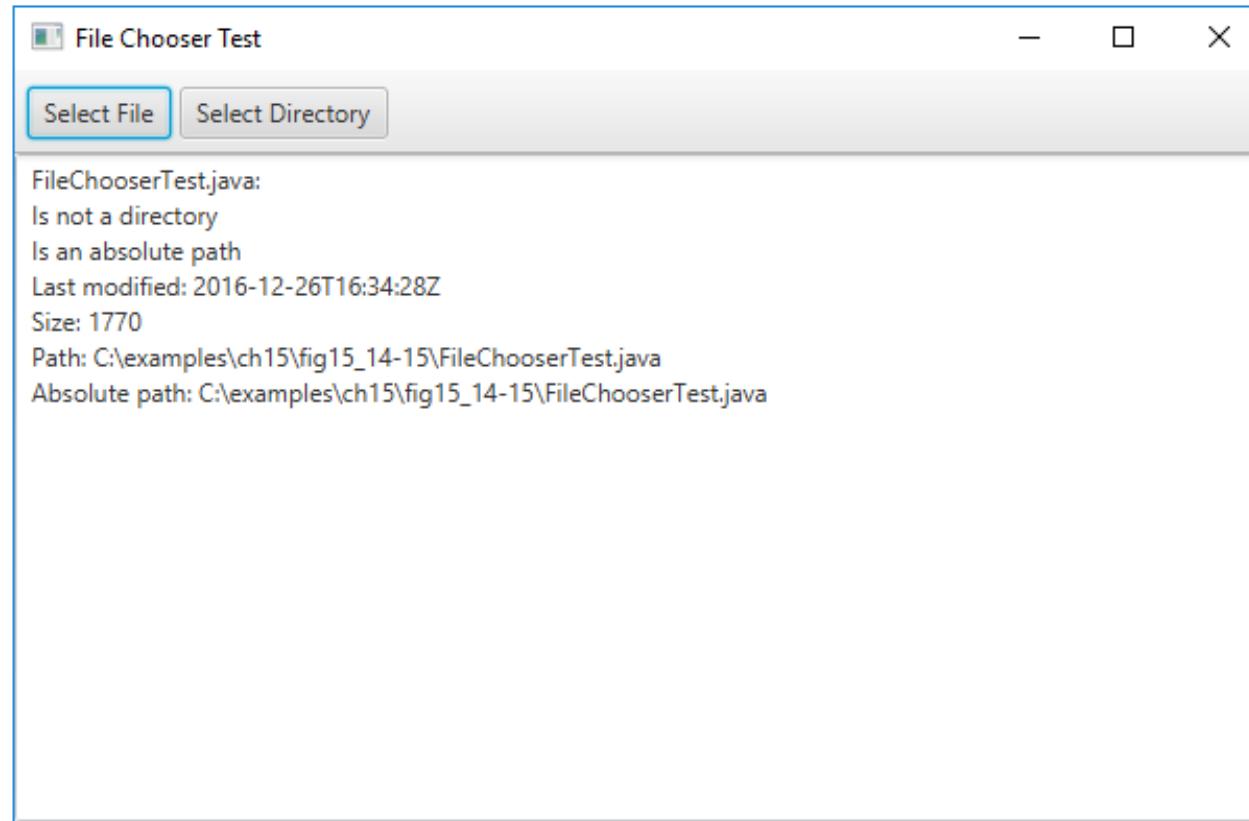
**Fig. 15.15** | Displays information about a selected file or folder. (Part 7 of 11.)

b) Selecting `FileChooserTest.java` from the `FileChooser` dialog displayed when the user clicked the **Select File Button**.



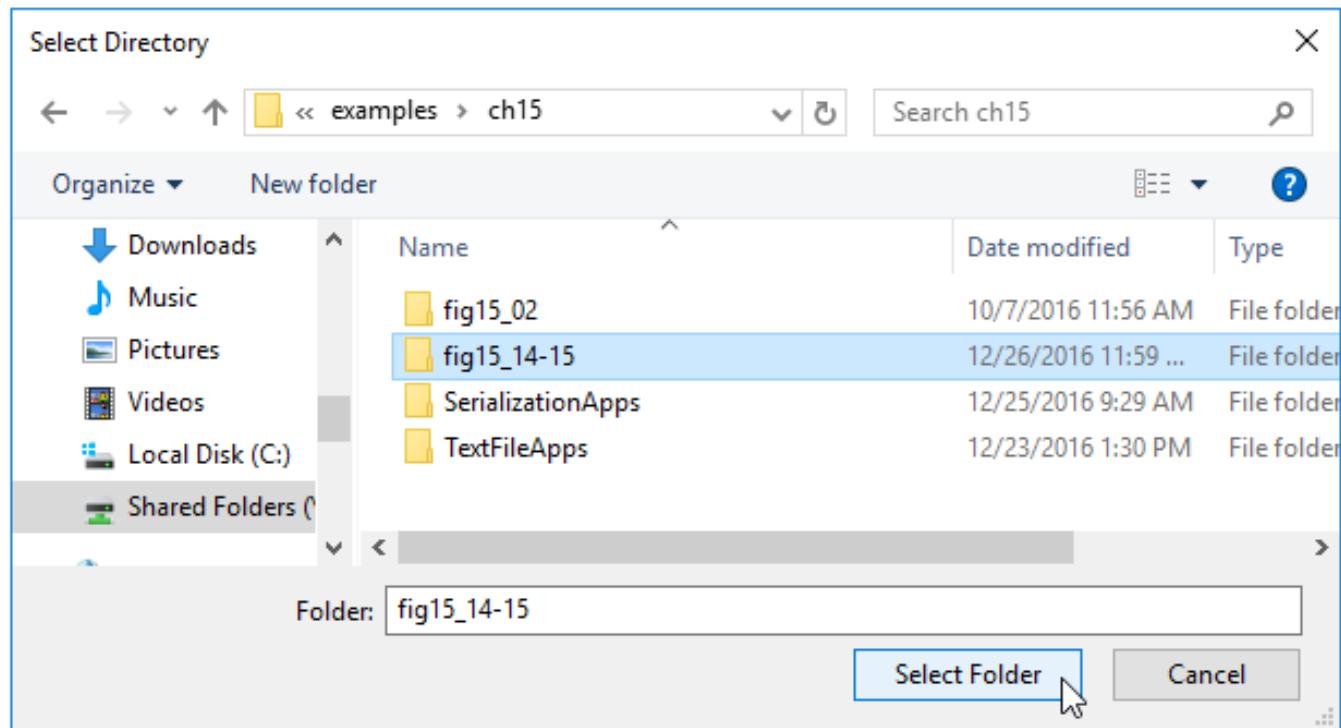
**Fig. 15.15** | Displays information about a selected file or folder. (Part 8 of 11.)

c) Displaying information about the file `FileChooserTest.java`.



**Fig. 15.15** | Displays information about a selected file or folder. (Part 9 of 11.)

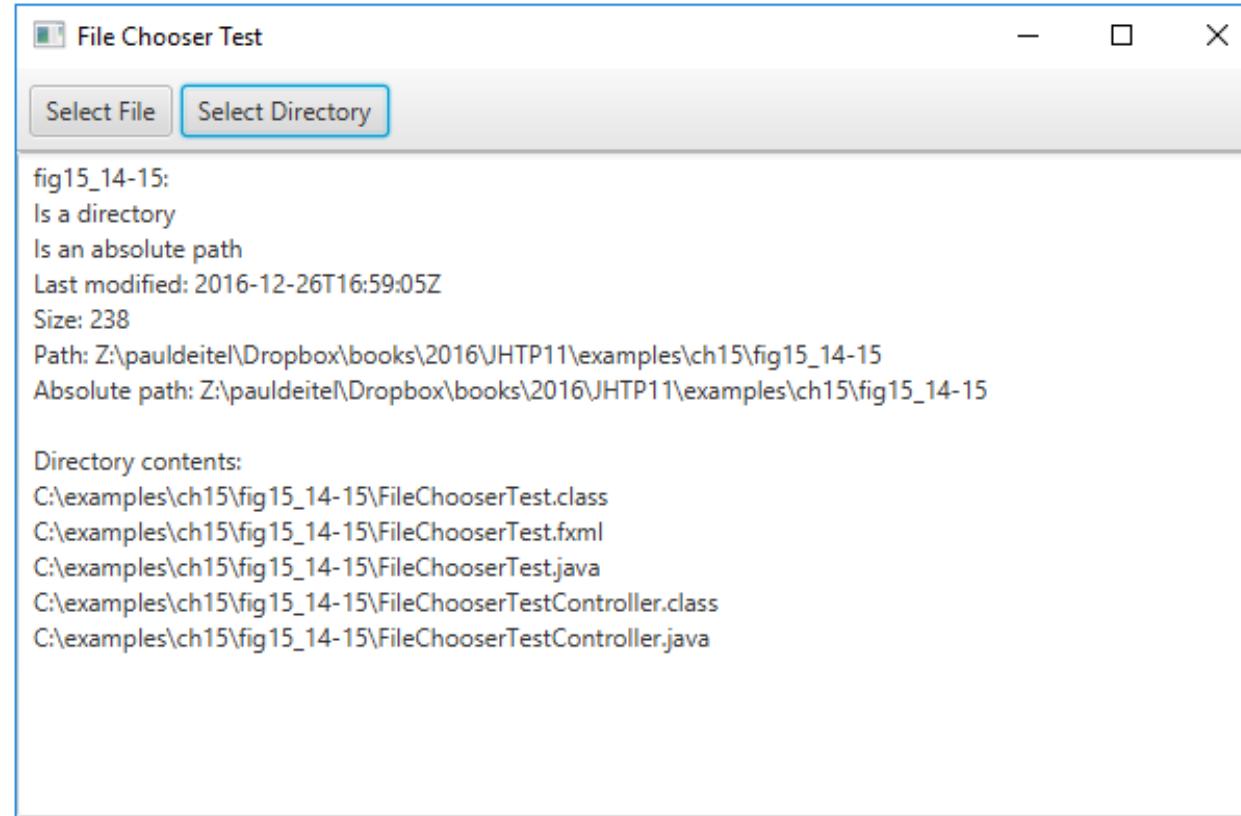
- 
- d) Selecting **fig15\_14-15** from the **DirectoryChooser** dialog displayed when the user clicked the **Select Directory Button**.



---

**Fig. 15.15** | Displays information about a selected file or folder. (Part 10 of 11.)

e) Displaying information about the directory **fig15\_14-15**.



**Fig. 15.15** | Displays information about a selected file or folder. (Part II of II.)

## 15.6 FileChooser and DirectoryChooser Dialogs (cont.)

### ***Method selectFileButtonPressed***

- ▶ Creates, configures and displays a FileChooser.
- ▶ Line 28 sets the text displayed in the FileChooser's title bar.
- ▶ Line 31 specifies the initial directory that should be opened when the FileChooser is displayed.
- ▶ Method `setInitialDirectory` receives a `File` object representing the directory's location—"." represents the current folder from which the app was launched.

## 15.6 FileChooser and DirectoryChooser Dialogs (cont.)

### *Method selectFileButtonPressed*

- ▶ Lines 34–35 display the FileChooser by calling its **showOpenDialog** method to display a dialog with an **Open** button for opening a file.
  - There's also a **showSaveDialog** method that displays a dialog with a **Save** button for saving a file.
- ▶ This method receives as its argument a reference to the app's Window.
  - A non-null argument makes the FileChooser a modal dialog that prevents the user from interacting with the rest of the app until the dialog is dismissed—when the user selects a file or clicks Cancel.

## 15.6 FileChooser and DirectoryChooser Dialogs (cont.)

### *Method selectFileButtonPressed*

- ▶ To obtain the app's Window, we use the borderPane's getScene method to get a reference to its parent Scene, then Scene's getWindow method to get a reference to the Window
- ▶ Method showOpenDialog returns a File representing the selected file's location, or null if the user clicks the **Cancel** button.

## 15.6 FileChooser and DirectoryChooser Dialogs (cont.)

### ***Method selectDirectoryButtonPressed***

- ▶ Creates, configures and displays a DirectoryChooser.
- ▶ DirectoryChooser method **showDialog** to display the dialog—there are not separate open and save dialogs for selecting folders.
- ▶ Method **showDialog** returns a **File** representing the location of the selected directory, or **null** if the user clicks **Cancel**.

## 15.7 (Optional) Additional `java.io` Classes

- ▶ This section overviews additional interfaces and classes (from package `java.io`).

## 15.7.1 Interfaces and Classes for Byte-Based Input and Output

- ▶ **InputStream** and **OutputStream** are abstract classes that declare methods for performing byte-based input and output, respectively.
- ▶ **Pipes** are synchronized communication channels between threads.
  - **PipedOutputStream** (a subclass of **OutputStream**) and **PipedInputStream** (a subclass of **InputStream**) establish pipes between two threads in a program.
  - One thread sends data to another by writing to a **PipedOutputStream**.
  - The target thread reads information from the pipe via a **PipedInputStream**.

## 15.7.1 Interfaces and Classes for Byte-Based Input and Output (cont.)

- ▶ A **FilterInputStream** filters an **InputStream**, and a **FilterOutputStream** filters an **OutputStream**.
- ▶ **Filtering** means simply that the filter stream provides additional functionality, such as aggregating bytes into meaningful primitive-type units.
- ▶ **FilterInputStream** and **FilterOutputStream** are typically used as superclasses, so some of their filtering capabilities are provided by their subclasses.

## 15.7.1 Interfaces and Classes for Byte-Based Input and Output (cont.)

- ▶ A **PrintStream** (a subclass of `FilterOutputStream`) performs text output to the specified stream.
- ▶ `System.out` and `System.err` are `PrintStream` objects.

## 15.7.1 Interfaces and Classes for Byte-Based Input and Output (cont.)

- ▶ Usually, programs read data as aggregates of bytes that form `ints`, `floats`, `doubles` and so on.
- ▶ Java programs can use several classes to input and output data in aggregate form.
- ▶ Interface `DataInput` describes methods for reading primitive types from an input stream.
- ▶ Classes **`DataInputStream`** and `RandomAccessFile` each implement this interface to read sets of bytes and process them as primitive-type values.

## 15.7.1 Interfaces and Classes for Byte-Based Input and Output (cont.)

- ▶ Interface `DataOutput` describes a set of methods for writing primitive types to an output stream.
- ▶ Classes **`DataOutputStream`** (a subclass of `FilterOutputStream`) and `RandomAccessFile` each implement this interface to write primitive-type values as bytes.

## 15.7.1 Interfaces and Classes for Byte-Based Input and Output (cont.)

- ▶ **Buffering** is an I/O-performance-enhancement technique.
- ▶ With a **BufferedOutputStream**, each output operation is directed to a **buffer**
  - holds the data of many output operations
- ▶ Transfer to the output device is performed in one large **physical output operation** each time the buffer fills.
- ▶ The output operations directed to the output buffer in memory are often called **logical output operations**.
- ▶ A partially filled buffer can be forced out to the device at any time by invoking the stream object's **flush** method.
- ▶ Using buffering can greatly increase the performance of an application.



## Performance Tip 15.1

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

## 15.7.1 Interfaces and Classes for Byte-Based Input and Output (cont.)

- ▶ With a **BufferedInputStream**, many “logical” chunks of data from a file are read as one large **physical input operation** into a memory buffer.
- ▶ As a program requests each new chunk of data, it’s taken from the buffer.
- ▶ This procedure is sometimes referred to as a **logical input operation**.
- ▶ When the buffer is empty, the next actual physical input operation from the input device is performed.

## 15.7.1 Interfaces and Classes for Byte-Based Input and Output (cont.)

- ▶ Java stream I/O includes capabilities for inputting from byte arrays in memory and outputting to byte arrays in memory.
- ▶ A `ByteArrayInputStream` (a subclass of `InputStream`) reads from a byte array in memory.
- ▶ A `ByteArrayOutputStream` (a subclass of `OutputStream`) outputs to a byte array in memory.

## 15.7.1 Interfaces and Classes for Byte-Based Input and Output (cont.)

- ▶ A `SequenceInputStream` (a subclass of `InputStream`) logically concatenates several `InputStreams`
- ▶ The program sees the group as one continuous `InputStream`.
- ▶ When the program reaches the end of one input stream, that stream closes, and the next stream in the sequence opens.

## 15.7.2 Interfaces and Classes for Character-Based Input and Output

- ▶ The **Reader** and **Writer** abstract classes are Unicode two-byte, character-based streams.
- ▶ Most of the byte-based streams have corresponding character-based concrete Reader or Writer classes.

## 15.7.2 Interfaces and Classes for Character-Based Input and Output (cont.)

- ▶ Classes **BufferedReader** (a subclass of abstract class Reader) and **BufferedWriter** (a subclass of abstract class Writer) enable buffering for character-based streams.
- ▶ Classes **CharArrayReader** and **CharArrayWriter** read and write, respectively, a stream of characters to a `char` array.
- ▶ A **LineNumberReader** (a subclass of BufferedReader) is a buffered character stream that keeps track of the number of lines read.

## 15.7.2 Interfaces and Classes for Character-Based Input and Output (cont.)

- ▶ An InputStream can be converted to a Reader via class **InputStreamReader**.
- ▶ An OutputStream can be converted to a Writer via class **OutputStreamWriter**.
- ▶ Class **FileReader** and class **FileWriter** read characters from and write characters to a file.
- ▶ Class **PipedReader** and class **PipedWriter** implement piped-character streams for transferring data between threads.
- ▶ Class **StringReader** b**StringWriter** read characters from and write characters to Strings.
- ▶ A PrintWriter writes characters to a stream.