Chapter 15
Files, Input/Output Stream, NIO and XML Serialization

Java How to Program, 11/e, Global Edition

Questions? E-mail 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

OUTLINE (cont.)

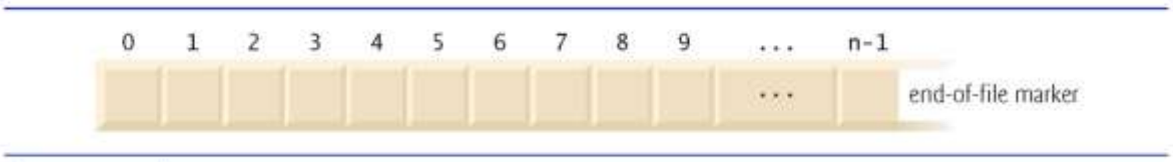


Fig. 15.1 | Java's view of a file of *n* bytes.

```java
// Fig. 15.2: FileAndDirectoryInfo.java
// File class used to obtain file and directory information.
import java.io.IOException;
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Scanner;

public class FileAndDirectoryInfo {
   public static void main(String[] args) throws IOException {
      Scanner input = new Scanner(System.in);

      System.out.println("Enter file or directory name:");

      // create Path object based on user input
      Path path = Paths.get(input.nextLine());
```

Fig. 15.2 | File class used to obtain file and directory information. (Part 1 of 5.)

```java

      if (Files.exists(path)) { // if path exists, output info about it
         // display file (or directory) information
         System.out.printf("%n%s exists%n", path.getFileName());
         System.out.printf("%s a directory%n",
            Files.isDirectory(path) ? "Is" : "Is not");
         System.out.printf("%s an absolute path%n",
            path.isAbsolute() ? "Is" : "Is not");
         System.out.printf("Last modified: %s%n",
            Files.getLastModifiedTime(path));
         System.out.printf("Size: %s%n", Files.size(path));
         System.out.printf("Path: %s%n", path);
         System.out.printf("Absolute path: %s%n", path.toAbsolutePath());

```

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

```
            if (Files.isDirectory(path)) { // output directory listing
               System.out.printf("%nDirectory contents:%n");

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

               for (Path p : directoryStream) {
                  System.out.println(p);
               }
            }
         }
         else { // not file or directory, output error message
            System.out.printf("%s does not exist%n", path);
         }
      } // end main
   } // end class FileAndDirectoryInfo
```

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: 2013-11-08T19:59:01.848255Z
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 `String`s 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.

```java
// Fig. 15.3: CreateTextFile.java
// Writing data to a sequential text file with class Formatter.
import java.io.FileNotFoundException;
import java.lang.SecurityException;
import java.util.Formatter;
import java.util.FormatterClosedException;
import java.util.NoSuchElementException;
import java.util.Scanner;

public class CreateTextFile {
   public static void main(String[] args) {
      // open clients.txt, output data to the file then close clients.txt
      try (Formatter output = new Formatter("clients.txt")) {
         Scanner input = new Scanner(System.in);
         System.out.printf("%s%n%s%n? ",
            "Enter account number, first name, last name and balance.",
            "Enter end-of-file indicator to end input.");

```

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

```java
            while (input.hasNext()) { // loop until end-of-file indicator
               try {
                  // output new record to file; assumes valid input
                  output.format("%d %s %s %.2f%n", input.nextInt(),
                     input.next(), input.next(), input.nextDouble());
               }
               catch (NoSuchElementException elementException) {
                  System.err.println("Invalid input. Please try again.");
                  input.nextLine(); // discard input so user can try again
               }

               System.out.print("? ");
            }
         }
         catch (SecurityException | FileNotFoundException |
            FormatterClosedException e) {
            e.printStackTrace();
         }
      }
   }
```

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.)

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

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

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.

```java
// Fig. 15.6: ReadTextFile.java
// This program reads a text file and displays each record.
import java.io.IOException;
import java.lang.IllegalStateException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.NoSuchElementException;
import java.util.Scanner;

public class ReadTextFile {
   public static void main(String[] args) {
      // open clients.txt, read its contents and close the file
      try(Scanner input = new Scanner(Paths.get("clients.txt"))) {
         System.out.printf("%-10s%-12s%-12s%10s%n", "Account",
            "First Name", "Last Name", "Balance");

```

Fig. 15.6 | Sequential file reading using a Scanner. (Part 1 of 2.)

```java
            // read record from file
            while (input.hasNext()) { // while there is more to read
               // display record contents
               System.out.printf("%-10d%-12s%-12s%10.2f%n", input.nextInt(),
                  input.next(), input.next(), input.nextDouble());
            }
         }
         catch (IOException | NoSuchElementException |
            IllegalStateException e) {
            e.printStackTrace();
         }
      }
   }
```

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

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

```java
// Fig. 15.7: MenuOption.java
// enum type for the credit-inquiry program's options.
public enum MenuOption {
   // declare contents of enum type
   ZERO_BALANCE(1),
   CREDIT_BALANCE(2),
   DEBIT_BALANCE(3),
   END(4);

   private final int value; // current menu option

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

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

```java
// Fig. 15.8: CreditInquiry.java
// This program reads a file sequentially and displays the
// contents based on the type of account the user requests
// (credit balance, debit balance or zero balance).
import java.io.IOException;
import java.lang.IllegalStateException;
import java.nio.file.Paths;
import java.util.NoSuchElementException;
import java.util.Scanner;

public class CreditInquiry {
   private final static MenuOption[] choices = MenuOption.values();

   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);

      // get user's request (e.g., zero, credit or debit balance)
      MenuOption accountType = getRequest(input);

```

Fig. 15.8 | Credit-inquiry program. (Part 1 of 7.)

```java
      while (accountType != MenuOption.END) {
         switch (accountType) {
            case ZERO_BALANCE:
               System.out.printf("%nAccounts with zero balances:%n");
               break;
            case CREDIT_BALANCE:
               System.out.printf("%nAccounts with credit balances:%n");
               break;
            case DEBIT_BALANCE:
               System.out.printf("%nAccounts with debit balances:%n");
               break;
         }

         readRecords(accountType);
         accountType = getRequest(input); // get user's request
      }
   }
```

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

```java

   // obtain request from user
   private static MenuOption getRequest(Scanner input) {
      int request = 4;

      // display request options
      System.out.printf("%nEnter request%n%s%n%s%n%s%n%s%n",
         " 1 - List accounts with zero balances",
         " 2 - List accounts with credit balances",
         " 3 - List accounts with debit balances",
         " 4 - Terminate program");

      try {
         do { // input user request
            System.out.printf("%n? ");
            request = input.nextInt();
         } while ((request < 1) || (request > 4));
      }
      catch (NoSuchElementException noSuchElementException) {
         System.err.println("Invalid input. Terminating.");
      }

      return choices[request - 1]; // return enum value for option
   }
```

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

```java

   // read records from file and display only records of appropriate type
   private static void readRecords(MenuOption accountType) {
      // open file and process contents
      try (Scanner input = new Scanner(Paths.get("clients.txt"))) {
         while (input.hasNext()) { // more data to read
            int accountNumber = input.nextInt();
            String firstName = input.next();
            String lastName = input.next();
            double balance = input.nextDouble();

            // if proper account type, display record
            if (shouldDisplay(accountType, balance)) {
               System.out.printf("%-10d%-12s%-12s%10.2f%n", accountNumber,
                  firstName, lastName, balance);
            }
            else {
               input.nextLine(); // discard the rest of the current record
            }
         }
      }
```

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

```
      catch (NoSuchElementException | IllegalStateException |
         IOException e) {
         System.err.println("Error processing file. Terminating.");
         System.exit(1);
      }
   }

   // use record type to determine if record should be displayed
   private static boolean shouldDisplay(
      MenuOption option, double balance) {
      if ((option == MenuOption.CREDIT_BALANCE) && (balance < 0)) {
         return true;
      }
      else if ((option == MenuOption.DEBIT_BALANCE) && (balance > 0)) {
         return true;
      }
      else if ((option == MenuOption.ZERO_BALANCE) && (balance == 0)) {
         return true;
      }

      return false;
   }
}
```

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.)

```java
// Fig. 15.9: Account.java
// Account class for storing records as objects.
public class Account {
   private int accountNumber;
   private String firstName;
   private String lastName;
   private double balance;

   // initializes an Account with default values
   public Account() {this(0, "", "", 0.0);}

   // initializes an Account with provided values
   public Account(int accountNumber, String firstName,
      String lastName, double balance) {
      this.accountNumber = accountNumber;
      this.firstName = firstName;
      this.lastName = lastName;
      this.balance = balance;
   }

```

Fig. 15.9 | Account class for storing records as objects. (Part 1 of 3.)

```
    // get account number
    public int getAccountNumber() {return accountNumber;}

    // set account number
    public void setAccountNumber(int accountNumber)
       {this.accountNumber = accountNumber;}

    // get first name
    public String getFirstName() {return firstName;}

    // set first name
    public void setFirstName(String firstName)
       {this.firstName = firstName;}

```

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

```
    // get last name
    public String getLastName() {return lastName;}

    // set last name
    public void setLastName(String lastName) {this.lastName = lastName;}

    // get balance
    public double getBalance() {return balance;}

    // set balance
    public void setBalance(double balance) {this.balance = balance;}
 }
```

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

```java
// Fig. 15.10: Accounts.java
// Maintains a List<Account>
import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlElement;

public class Accounts {
   // @XmlElement specifies XML element name for each object in the List
   @XmlElement(name="account")
   private List<Account> accounts = new ArrayList<>(); // stores Accounts

   // returns the List<Accounts>
   public List<Account> getAccounts() {return accounts;}
}
```

Fig. 15.10 | Account class for serializable objects.

```java
// Fig. 15.11: CreateSequentialFile.java
// Writing objects to a file with JAXB and BufferedWriter.
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.NoSuchElementException;
import java.util.Scanner;
import javax.xml.bind.JAXB;

public class CreateSequentialFile {
   public static void main(String[] args) {
      // open clients.xml, write objects to it then close file
      try(BufferedWriter output =
         Files.newBufferedWriter(Paths.get("clients.xml"))) {

         Scanner input = new Scanner(System.in);

```

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

```
         // stores the Accounts before XML serialization
         Accounts accounts = new Accounts();

         System.out.printf("%s%n%s%n? ",
            "Enter account number, first name, last name and balance.",
            "Enter end-of-file indicator to end input.");

         while (input.hasNext()) { // loop until end-of-file indicator
            try {
               // create new record
               Account record = new Account(input.nextInt(),
                  input.next(), input.next(), input.nextDouble());

               // add to AccountList
               accounts.getAccounts().add(record);
            }
            catch (NoSuchElementException elementException) {
               System.err.println("Invalid input. Please try again.");
               input.nextLine(); // discard input so user can try again
            }

            System.out.print("? ");
         }
```

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

```

         // write AccountList's XML to output
         JAXB.marshal(accounts, output);
      }
      catch (IOException ioException) {
         System.err.println("Error opening file. Terminating.");
      }
   }
}
```

```
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.)

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<accounts>
    <account>
        <accountNumber>100</accountNumber>
        <balance>24.98</balance>
        <firstName>Bob</firstName>
        <lastName>Blue</lastName>
    </account>
    <account>
        <accountNumber>200</accountNumber>
        <balance>-345.67</balance>
        <firstName>Steve</firstName>
        <lastName>Green</lastName>
    </account>
    <account>
        <accountNumber>300</accountNumber>
        <balance>0.0</balance>
        <firstName>Pam</firstName>
        <lastName>White</lastName>
    </account>
```

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

```xml
    <account>
        <accountNumber>400</accountNumber>
        <balance>-42.16</balance>
        <firstName>Sam</firstName>
        <lastName>Red</lastName>
    </account>
    <account>
        <accountNumber>500</accountNumber>
        <balance>224.62</balance>
        <firstName>Sue</firstName>
        <lastName>Yellow</lastName>
    </account>
</accounts>
```

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

```
// Fig. 15.13: ReadSequentialFile.java
// Reading a file of XML serialized objects with JAXB and a
// BufferedReader and displaying each object.
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import javax.xml.bind.JAXB;

public class ReadSequentialFile {
   public static void main(String[] args) {
      // try to open file for deserialization
      try(BufferedReader input =
         Files.newBufferedReader(Paths.get("clients.xml"))) {
         // unmarshal the file's contents
         Accounts accounts = JAXB.unmarshal(input, Accounts.class);

         // display contents
         System.out.printf("%-10s%-12s%-12s%10s%n", "Account",
            "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 1 of 2.)

```

         for (Account account : accounts.getAccounts()) {
            System.out.printf("%-10d%-12s%-12s%10.2f%n",
               account.getAccountNumber(), account.getFirstName(),
               account.getLastName(), account.getBalance());
         }
      }
      catch (IOException ioException) {
         System.err.println("Error opening file.");
      }
   }
}
```

```
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.)

```java
// Fig. 15.14: FileChooserTest.java
// App to test classes FileChooser and DirectoryChooser.
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class FileChooserTest extends Application {
   @Override
   public void start(Stage stage) throws Exception {
      Parent root =
         FXMLLoader.load(getClass().getResource("FileChooserTest.fxml"));

      Scene scene = new Scene(root);
      stage.setTitle("File Chooser Test"); // displayed in title bar
      stage.setScene(scene);
      stage.show();
   }

   public static void main(String[] args) {
      launch(args);
   }
}
```

Fig. 15.14 | Demonstrating JFileChooser.

```java
// Fig. 15.15: FileChooserTestController.java
// Displays information about a selected file or folder.
import java.io.File;
import java.io.IOException;
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.Button;
import javafx.scene.control.TextArea;
import javafx.scene.layout.BorderPane;
import javafx.stage.DirectoryChooser;
import javafx.stage.FileChooser;

```

Fig. 15.15 | Displays information about a selected file or folder. (Part 1 of 11.)

```java
public class FileChooserTestController {
   @FXML private BorderPane borderPane;
   @FXML private Button selectFileButton;
   @FXML private Button selectDirectoryButton;
   @FXML private TextArea textArea;

   // handles selectFileButton's events
   @FXML
   private void selectFileButtonPressed(ActionEvent e) {
      // configure dialog allowing selection of a file
      FileChooser fileChooser = new FileChooser();
      fileChooser.setTitle("Select File");

      // display files in folder from which the app was launched
      fileChooser.setInitialDirectory(new File("."));

      // display the FileChooser
      File file = fileChooser.showOpenDialog(
         borderPane.getScene().getWindow());

```

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

```java
      // process selected Path or display a message
      if (file != null) {
         analyzePath(file.toPath());
      }
      else {
         textArea.setText("Select file or directory");
      }
   }

   // handles selectDirectoryButton's events
   @FXML
   private void selectDirectoryButtonPressed(ActionEvent e) {
      // configure dialog allowing selection of a directory
      DirectoryChooser directoryChooser = new DirectoryChooser();
      directoryChooser.setTitle("Select Directory");

      // display folder from which the app was launched
      directoryChooser.setInitialDirectory(new File("."));

```

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

```java
      // display the FileChooser
      File file = directoryChooser.showDialog(
         borderPane.getScene().getWindow());

      // process selected Path or display a message
      if (file != null) {
         analyzePath(file.toPath());
      }
      else {
         textArea.setText("Select file or directory");
      }
   }

   // display information about file or directory user specifies
   public void analyzePath(Path path) {
      try {
         // if the file or directory exists, display its info
         if (path != null && Files.exists(path)) {
            // gather file (or directory) information
            StringBuilder builder = new StringBuilder();
            builder.append(String.format("%s:%n", path.getFileName()));
```

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

```java
            builder.append(String.format("%s a directory%n",
               Files.isDirectory(path) ? "Is" : "Is not"));
            builder.append(String.format("%s an absolute path%n",
               path.isAbsolute() ? "Is" : "Is not"));
            builder.append(String.format("Last modified: %s%n",
               Files.getLastModifiedTime(path)));
            builder.append(String.format("Size: %s%n", Files.size(path)));
            builder.append(String.format("Path: %s%n", path));
            builder.append(String.format("Absolute path: %s%n",
               path.toAbsolutePath()));

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

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

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

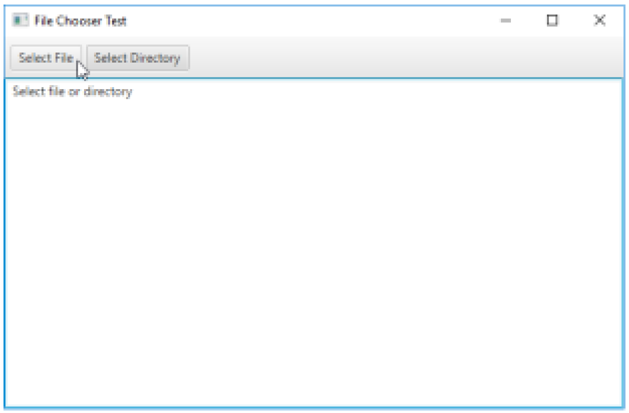
Fig. 15.15 | Displays information about a selected file or folder. (Part 5 of 11.)

```

               // display file or directory info
               textArea.setText(builder.toString());
            }
            else { // Path does not exist
               textArea.setText("Path does not exist");
            }
         }
         catch (IOException ioException) {
            textArea.setText(ioException.toString());
         }
      }
   }
```

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

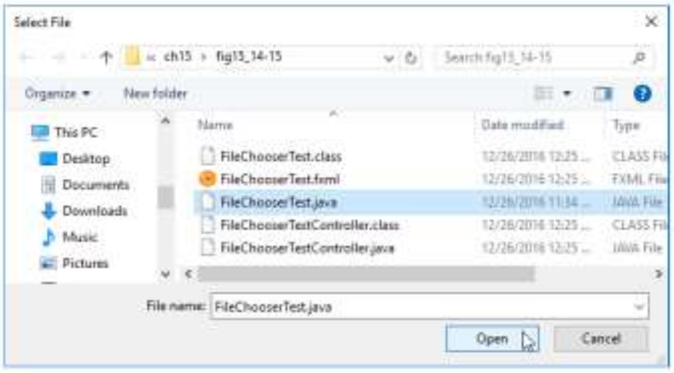
a) Initial app window.



Select file or directory

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**.

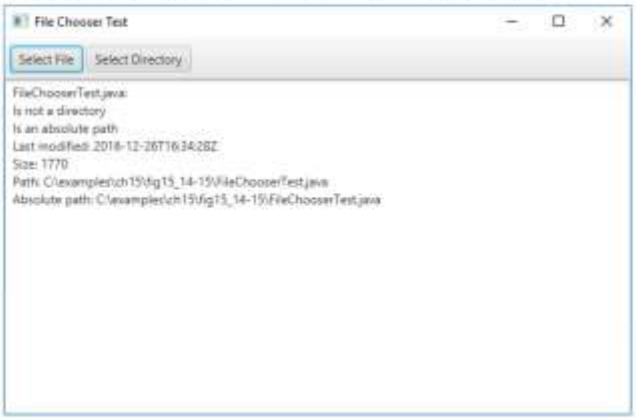


Name	Date modified	Type
FileChooserTest.class	12/26/2016 12:25 ...	CLASS Fi
FileChooserTest.fxml	12/26/2016 12:25 ...	FXML Fil
FileChooserTest.java	12/26/2016 11:34 ...	JAVA File
FileChooserTestController.class	12/26/2016 12:25 ...	CLASS Fi
FileChooserTestController.java	12/26/2016 12:25 ...	JAVA File

File name: FileChooserTest.java

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

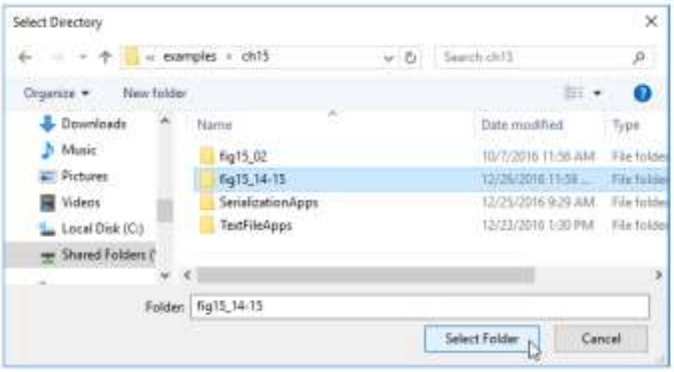
c) Displaying information about the file **FileChooserTest.java**.



```
FileChooserTest.java:
Is not a directory
Is an absolute path
Last modified: 2016-12-26T16:34:28Z
Size: 1770
Path: C:\examples\ch15\fig15_14-15\FileChooserTest.java
Absolute path: C:\examples\ch15\fig15_14-15\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`**.

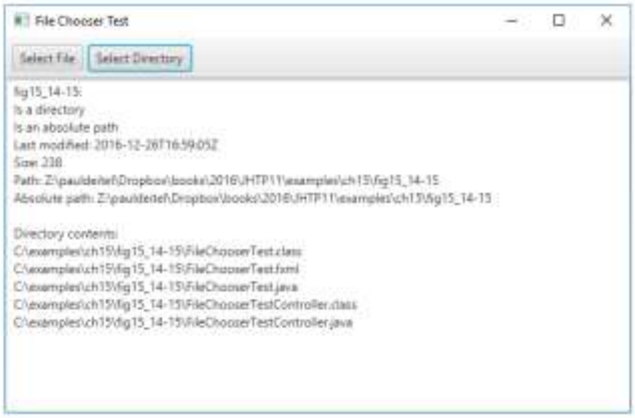


Name	Date modified	Type
fig15_02	10/7/2016 11:56 AM	File folder
fig15_14-15	12/26/2016 11:58 ...	File folder
SerializationApps	12/25/2016 9:29 AM	File folder
TextFileApps	12/23/2016 1:30 PM	File folder

Folder: fig15_14-15

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

e) Displaying information about the directory **`fig15_14-15`**.



```
fig15_14-15:
Is a directory
Is an absolute path
Last modified: 2016-12-26T16:59:05Z
Size: 238
Path: Z:\pauldeitel\Dropbox\books\2016\JHTP11\examples\ch15\fig15_14-15
Absolute path: Z:\pauldeitel\Dropbox\books\2016\JHTP11\examples\ch15\fig15_14-15

Directory contents:
C:\examples\ch15\fig15_14-15\FileChooserTest.class
C:\examples\ch15\fig15_14-15\FileChooserTest.fxml
C:\examples\ch15\fig15_14-15\FileChooserTest.java
C:\examples\ch15\fig15_14-15\FileChooserTestController.class
C:\examples\ch15\fig15_14-15\FileChooserTestController.java
```

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

Performance Tip 15.1

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