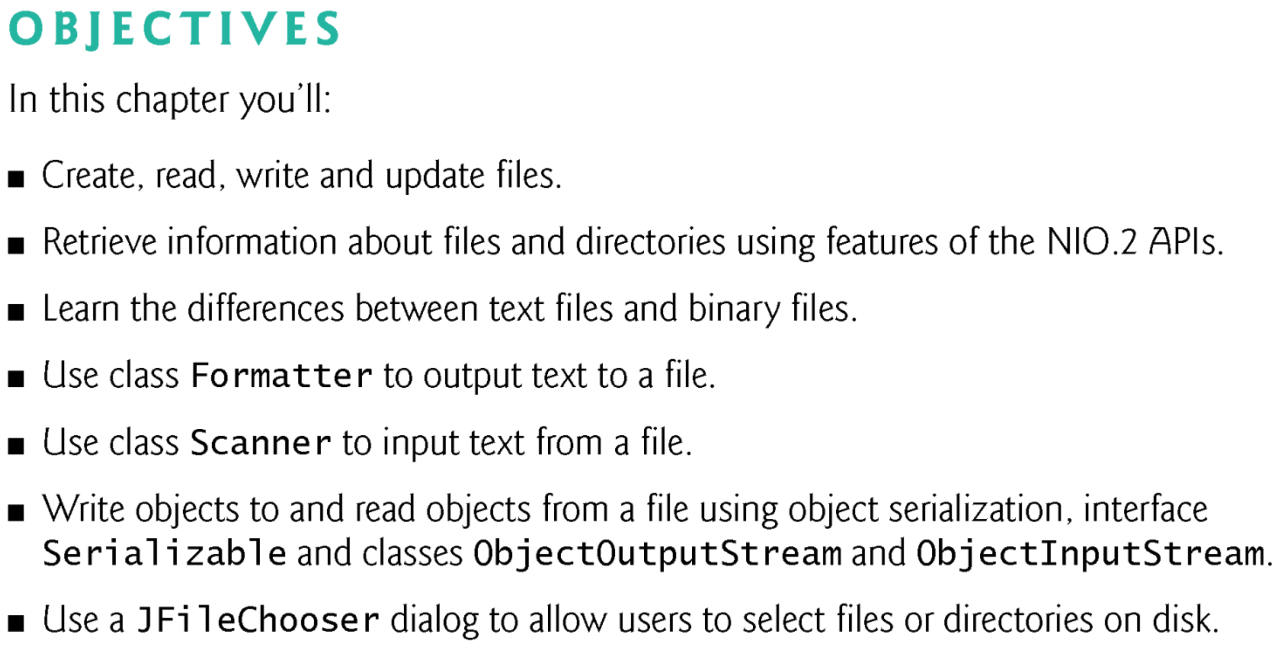
Chapter 15
Files, Streams and Object Serialization

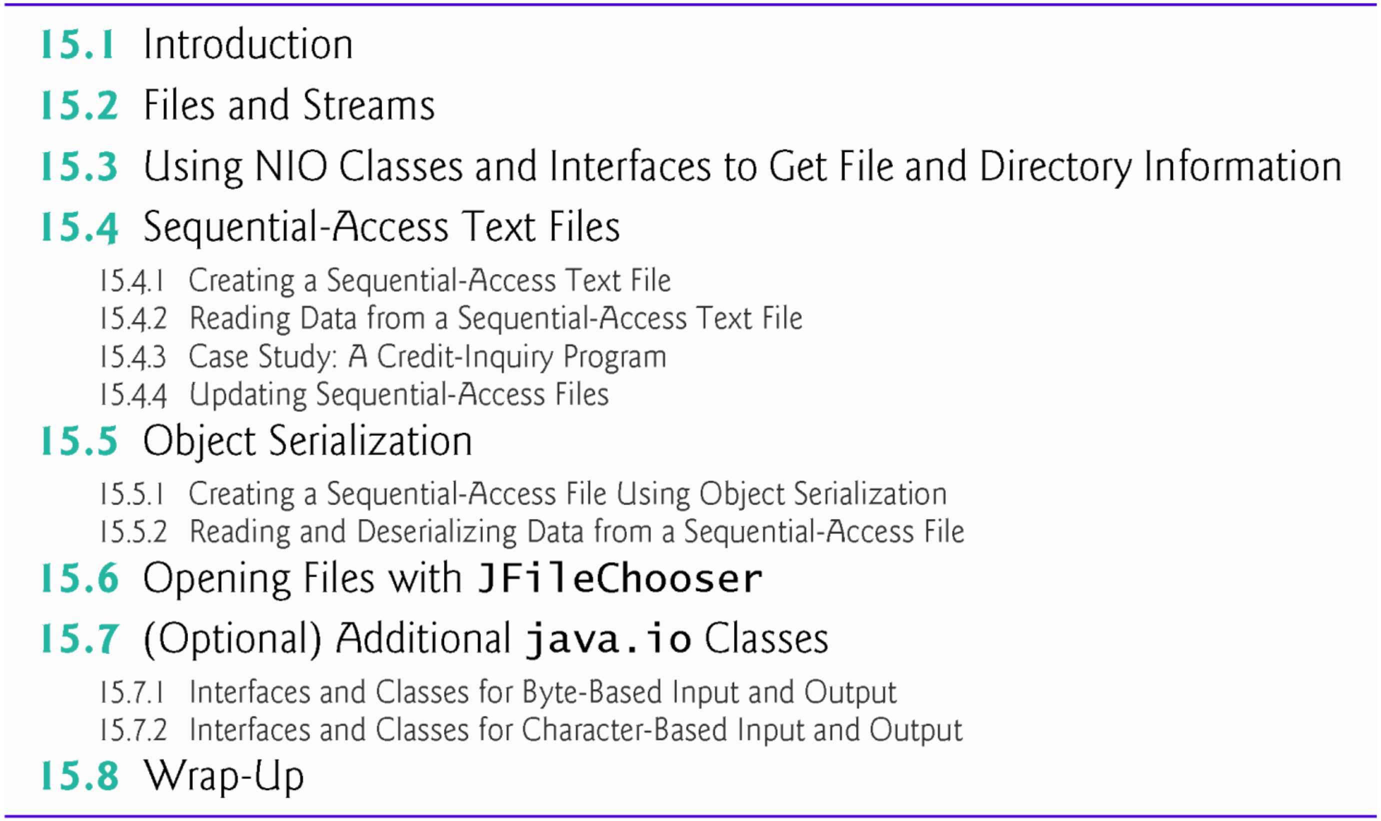
Java How to Program, 10/e



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.
- Write objects to and read objects from a file using object serialization, interface **`Serializable`** and classes **`ObjectOutputStream`** and **`ObjectInputStream`**.
- Use a **`JFileChooser`** dialog to allow users to select files or directories on disk.



15.1 Introduction

15.2 Files and Streams

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

15.4 Sequential-Access Text Files

- 15.4.1 Creating a Sequential-Access Text File
- 15.4.2 Reading Data from a Sequential-Access Text File
- 15.4.3 Case Study: A Credit-Inquiry Program
- 15.4.4 Updating Sequential-Access Files

15.5 Object Serialization

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

15.6 Opening Files with `JFileChooser`

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

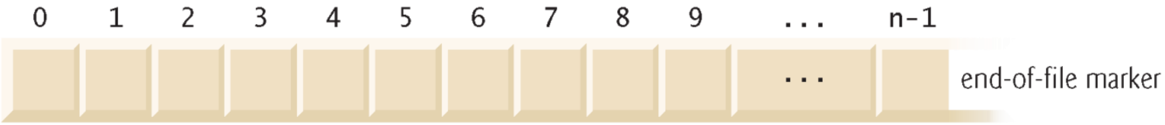


Fig. 15.1 | Java's view of a file of *n* bytes.

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 `ArrayList`s), 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.

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

- 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.
- For example, if we were to use the path
  - `c:\Program Files\Java\jdk1.6.0_11\demo/jfc`
- which employs each separator character, Java would still process the path properly.

```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());

```

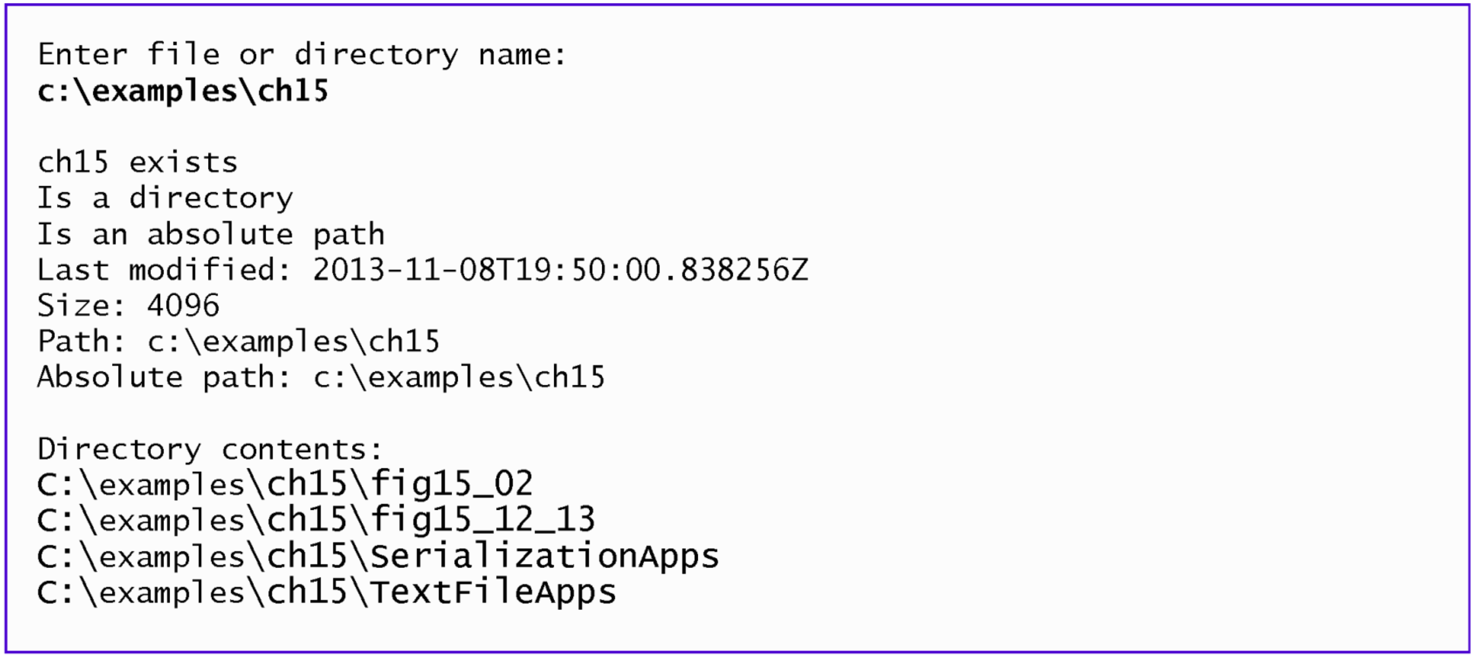
Fig. 15.2 | File class used to obtain file and directory information. (Part 1 of 5.)

```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());

         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);

```

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

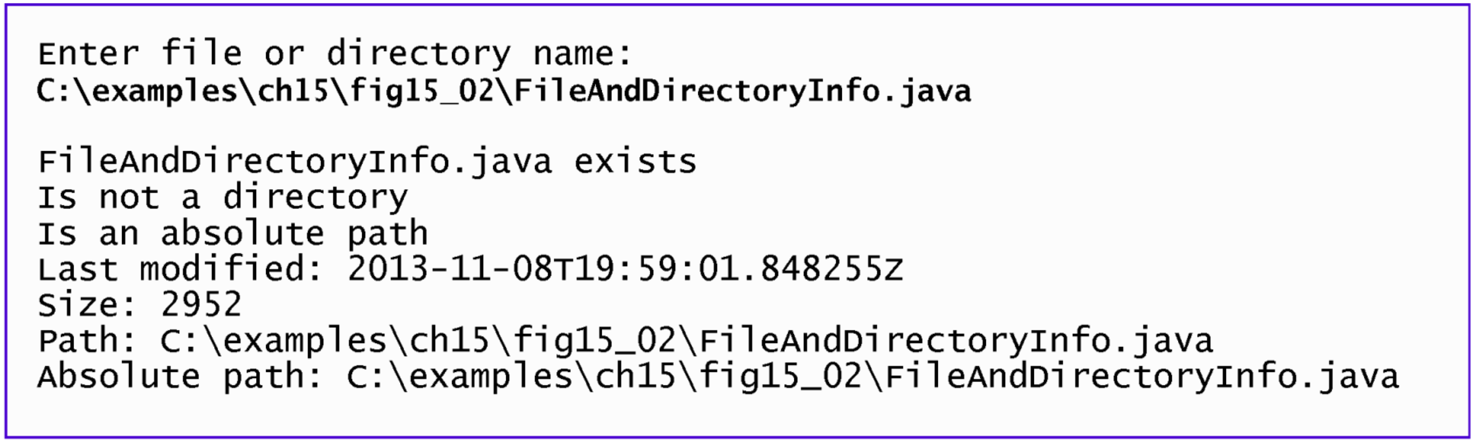
```java
            for (Path p : directoryStream)
               System.out.println(p);
         }
      }
      else // not file or directory, output error message
      {
         System.out.printf("%s does not exist%n", path);
      }
   } // end main
} // end class FileAndDirectoryInfo
```

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



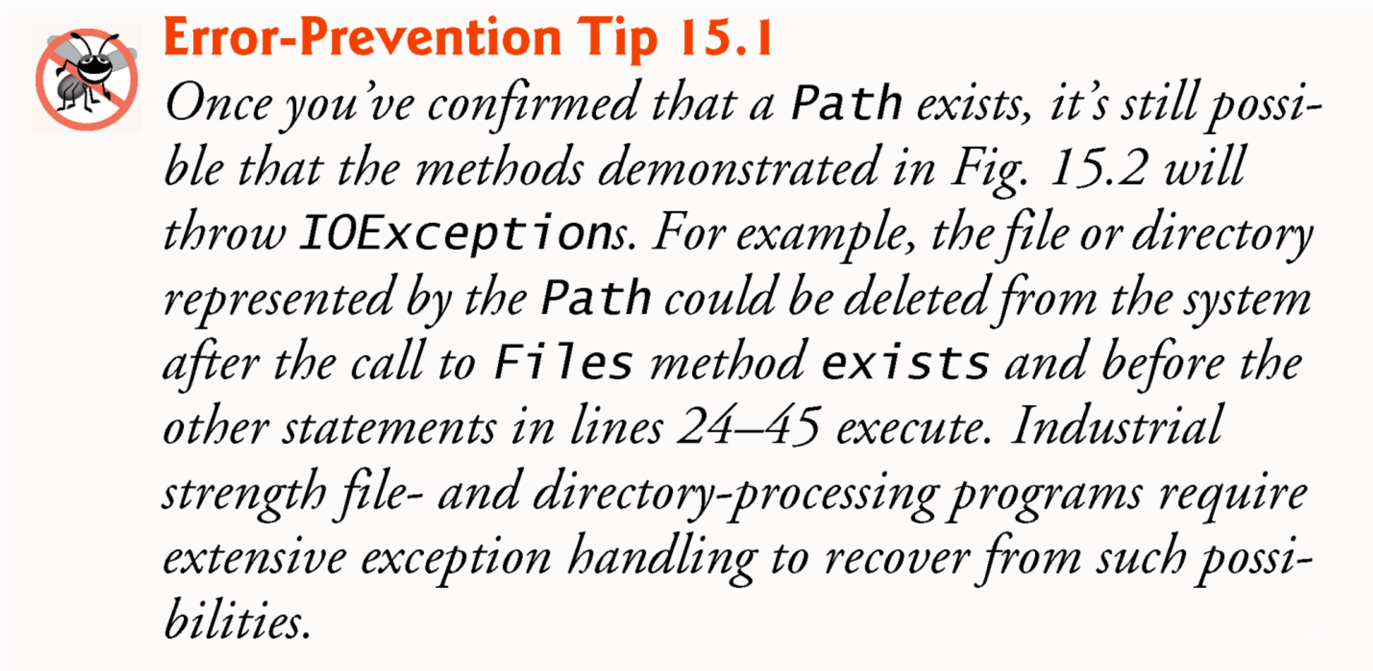
```
Enter file or directory name:
c:\examples\ch15

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.)



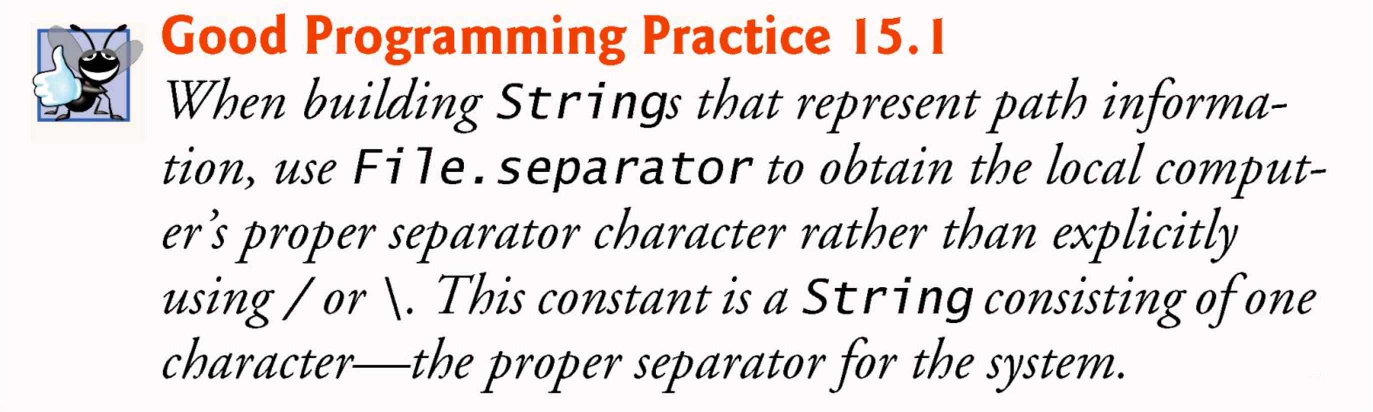
```
Enter file or directory name:
C:\examples\ch15\fig15_02\FileAndDirectoryInfo.java

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.)



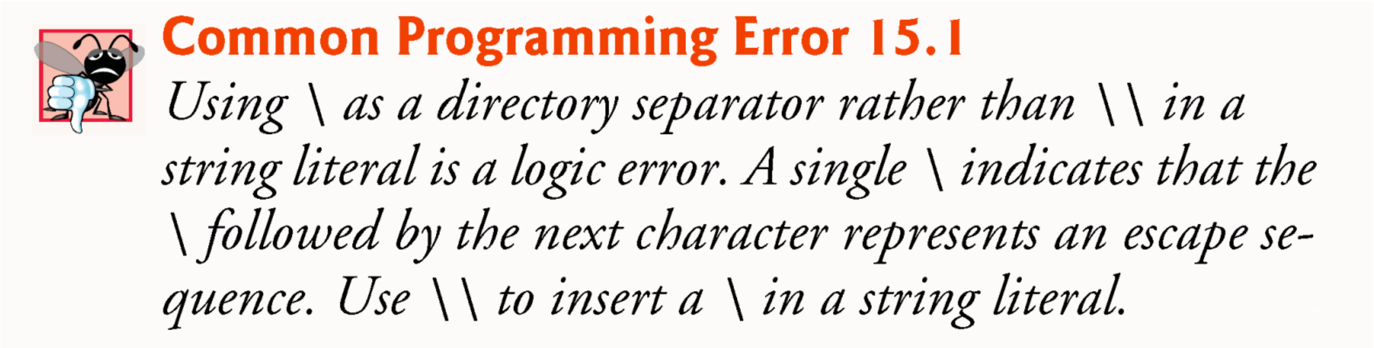
Error-Prevention Tip 15.1

Once you’ve confirmed that a `Path` *exists, it’s still possible that the methods demonstrated in Fig. 15.2 will throw* `IOException`*s. 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 24–45 execute. Industrial strength file- and directory-processing programs require extensive exception handling to recover from 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.

15.4 Sequential-Access 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-Access Text File

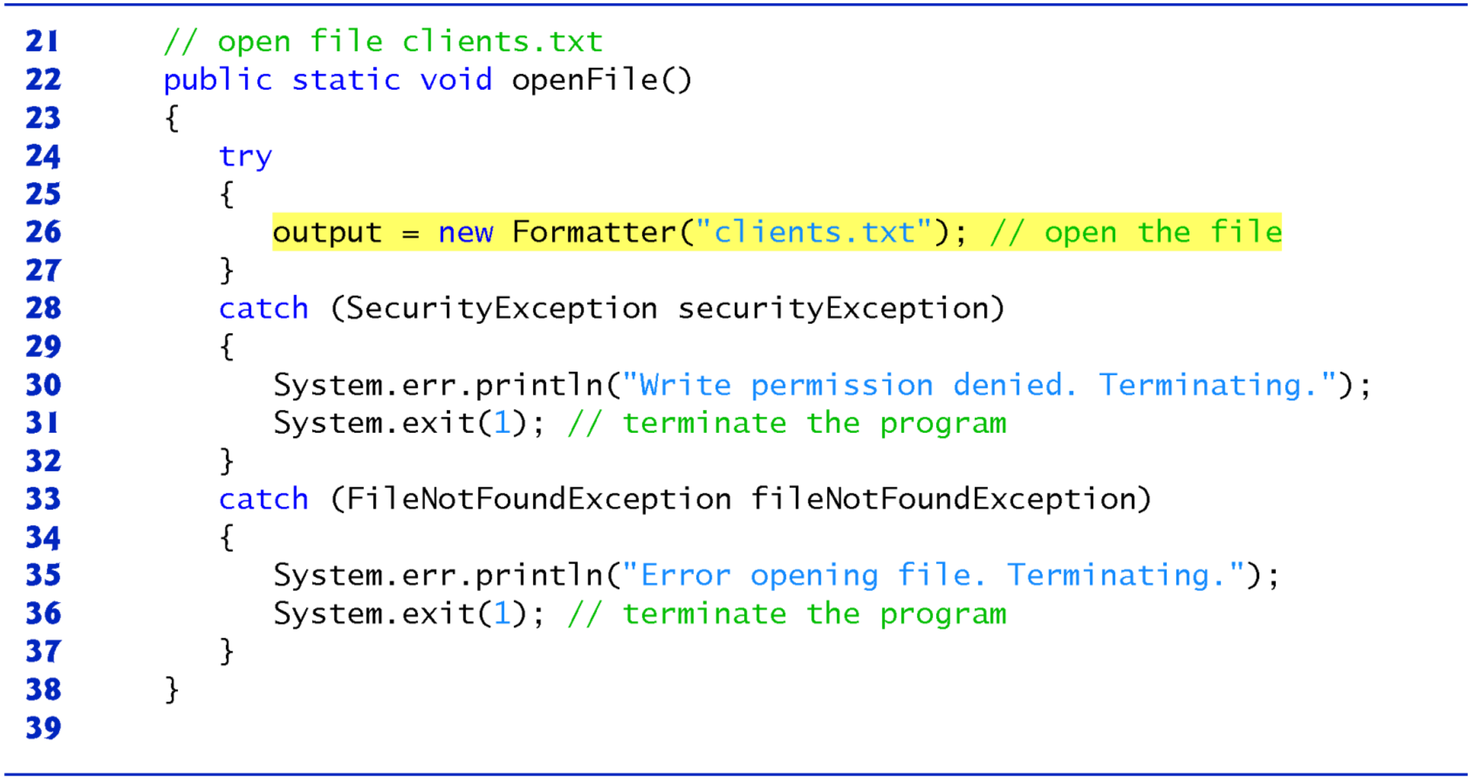
- Java imposes no structure on a file
  - Notions such as records do not exist as part of the Java language.
  - You must structure files to meet the requirements of your applications.

15.4.1 Creating a Sequential-Access Text File (cont.)

- `Formatter` outputs formatted `String`s 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**.

```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
{
   private static Formatter output; // outputs text to a file

   public static void main(String[] args)
   {
      openFile();
      addRecords();
      closeFile();
   }

```

Fig. 15.3 | Writing data to a sequential text file with class Formatter. (Part 1 of 5.)



```
    // open file clients.txt
    public static void openFile()
    {
       try
       {
          output = new Formatter("clients.txt"); // open the file
       }
       catch (SecurityException securityException)
       {
          System.err.println("Write permission denied. Terminating.");
          System.exit(1); // terminate the program
       }
       catch (FileNotFoundException fileNotFoundException)
       {
          System.err.println("Error opening file. Terminating.");
          System.exit(1); // terminate the program
       }
    }

```

Fig. 15.3 | Writing data to a sequential text file with class Formatter. (Part 2 of 5.)

```
   // add records to file
   public static void addRecords()
   {
      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.");

      while (input.hasNext()) // loop until end-of-file indicator
      {
         try
         {
```

Fig. 15.3 | Writing data to a sequential text file with class `Formatter`. (Part 3 of 5.)

```
            // output new record to file; assumes valid input
            output.format("%d %s %s %.2f%n", input.nextInt(),
               input.next(), input.next(), input.nextDouble());
         }
         catch (FormatterClosedException formatterClosedException)
         {
            System.err.println("Error writing to file. Terminating.");
            break;
         }
         catch (NoSuchElementException elementException)
         {
            System.err.println("Invalid input. Please try again.");
            input.nextLine(); // discard input so user can try again
         }

         System.out.print("? ");
      } // end while
   } // end method addRecords

```

Fig. 15.3 | Writing data to a sequential text file with class `Formatter`. (Part 4 of 5.)

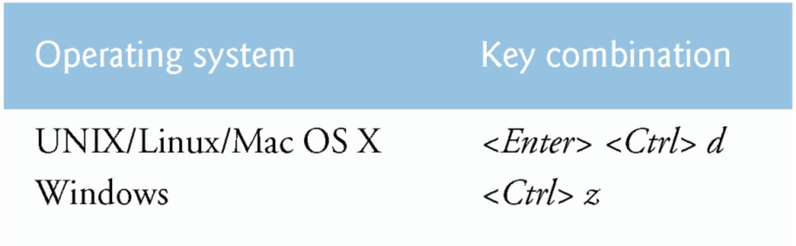
```java
   // close file
   public static void closeFile()
   {
      if (output != null)
         output.close();
   }
} // end class CreateTextFile
```

```
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 5 of 5.)

15.4.1 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.

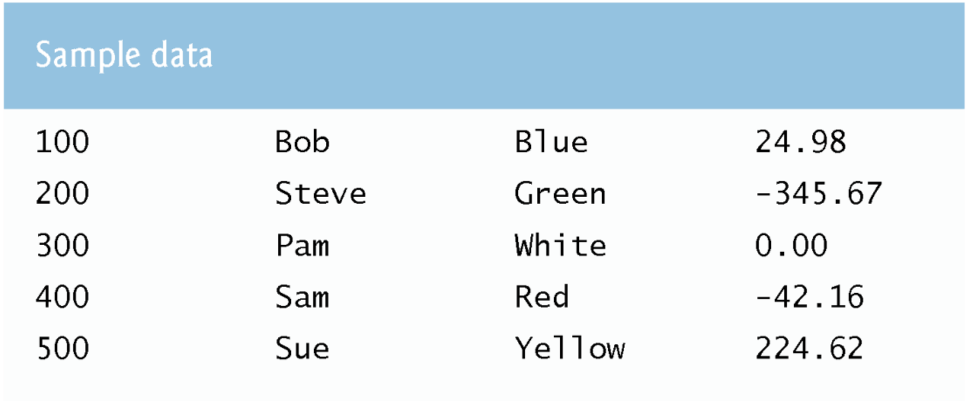


Operating system	Key combination
UNIX/Linux/Mac OS X	*<Enter> <Ctrl> d*
Windows	*<Ctrl> z*

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

15.4.1 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 sys-tem 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

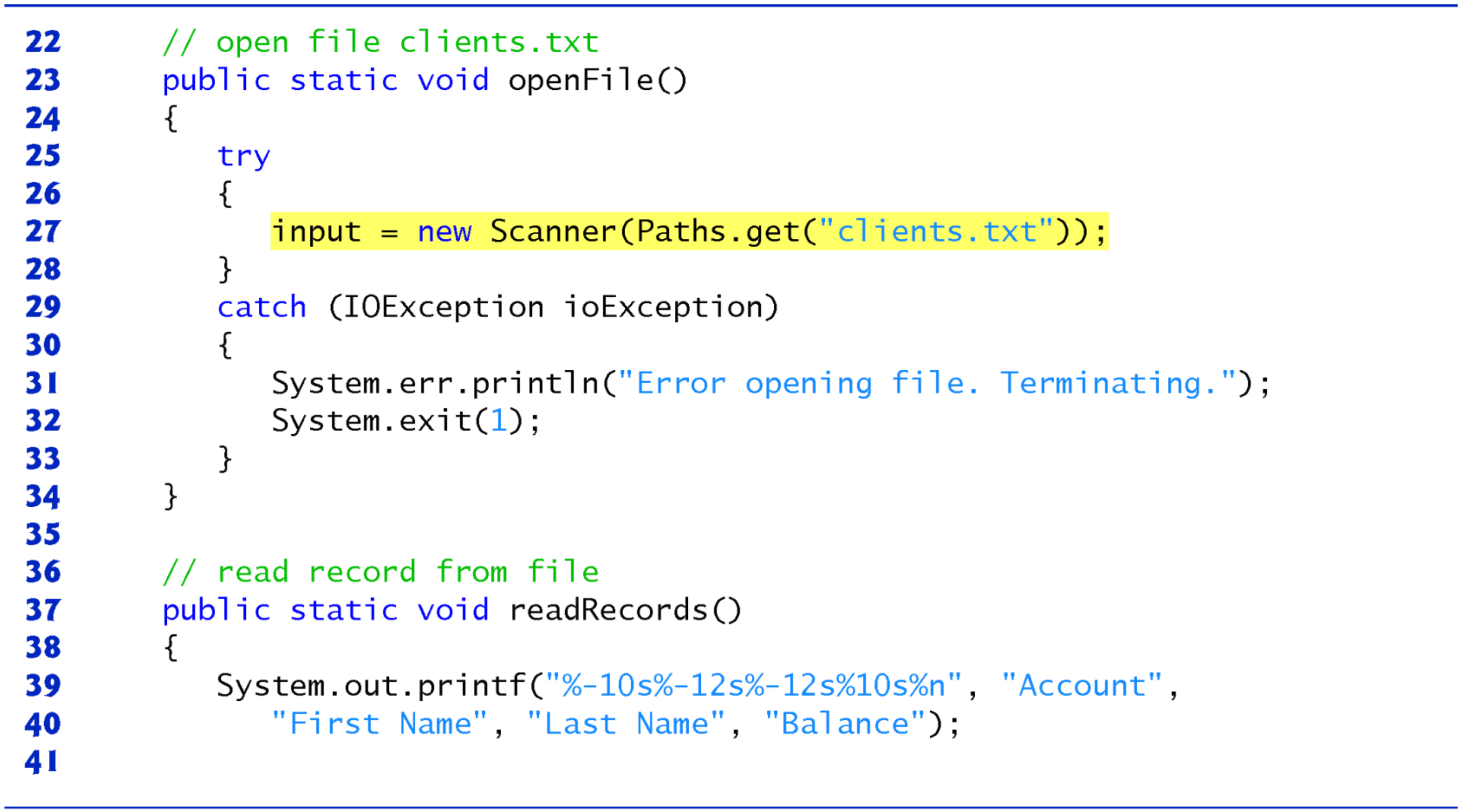
Fig. 15.5 | Sample data for the program in Fig. 15.3.

15.4.2 Reading Data from a Sequential-Access 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.

```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
{
   private static Scanner input;

   public static void main(String[] args)
   {
      openFile();
      readRecords();
      closeFile();
   }

```

Fig. 15.6 | Sequential file reading using a `Scanner`. (Part 1 of 4.)



```java
    // open file clients.txt
    public static void openFile()
    {
       try
       {
          input = new Scanner(Paths.get("clients.txt"));
       }
       catch (IOException ioException)
       {
          System.err.println("Error opening file. Terminating.");
          System.exit(1);
       }
    }

    // read record from file
    public static void readRecords()
    {
       System.out.printf("%-10s%-12s%-12s%10s%n", "Account",
          "First Name", "Last Name", "Balance");

```

Fig. 15.6 | Sequential file reading using a Scanner. (Part 2 of 4.)

```java
      try
      {
         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 (NoSuchElementException elementException)
      {
         System.err.println("File improperly formed. Terminating.");
      }
      catch (IllegalStateException stateException)
      {
         System.err.println("Error reading from file. Terminating.");
      }
   } // end method readRecords

```

Fig. 15.6 | Sequential file reading using a Scanner. (Part 3 of 4.)

```
   // close file and terminate application
   public static void closeFile()
   {
      if (input != null)
         input.close();
   }
} // end class ReadTextFile
```

```
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
```

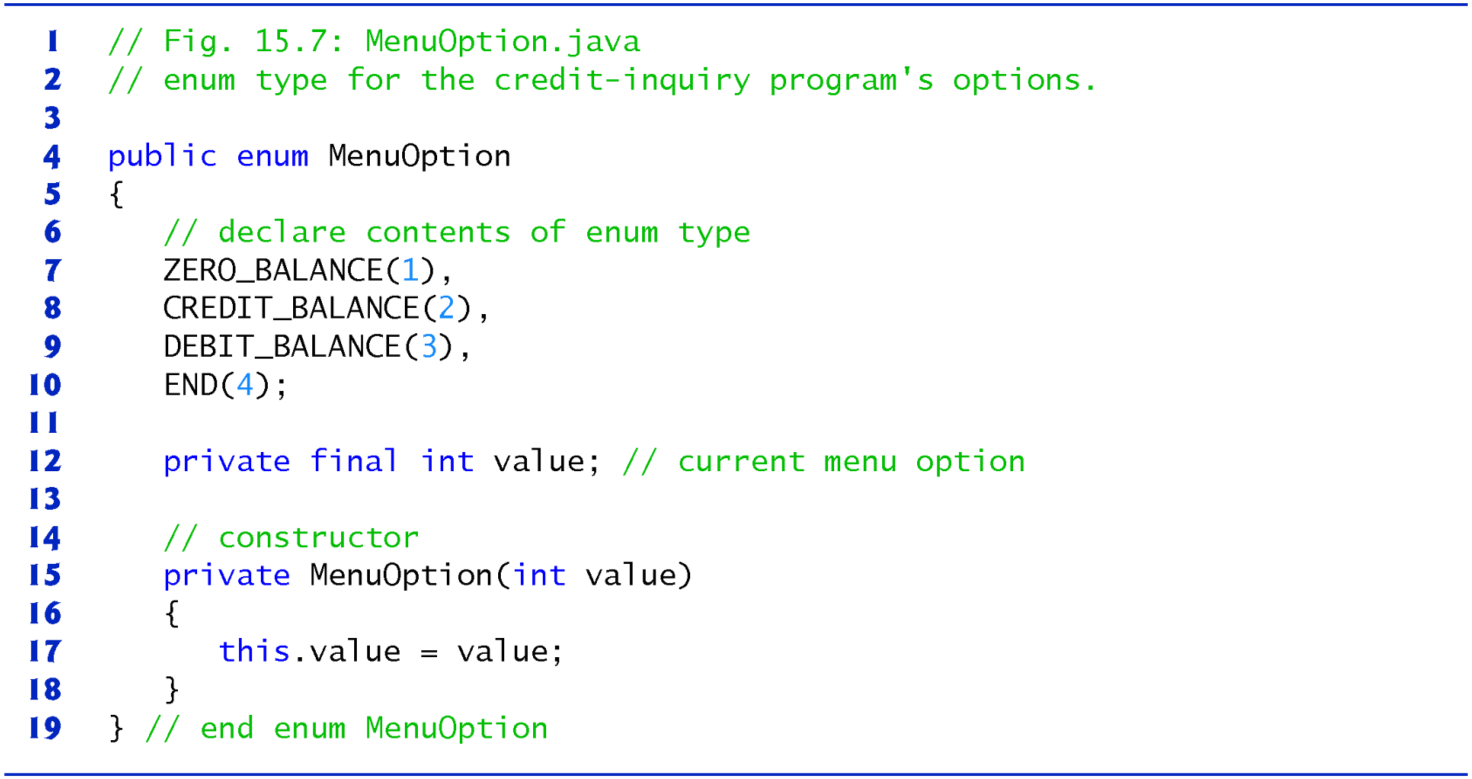
Fig. 15.6 | Sequential file reading using a `Scanner`. (Part 4 of 4.)

15.4.2 Reading Data from a Sequential-Access Text File

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

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.



```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;
   }
} // end enum MenuOption
```

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

```

Fig. 15.8 | Credit-inquiry program. (Part 1 of 8.)

```
      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(); // get user's request
      }
   }

```

Fig. 15.8 | Credit-inquiry program. (Part 2 of 8.)

```java
   // obtain request from user
   private static MenuOption getRequest()
   {
      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
      {
         Scanner input = new Scanner(System.in);

         do // input user request
         {
            System.out.printf("%n? ");
            request = input.nextInt();
         } while ((request < 1) || (request > 4));
      }
```

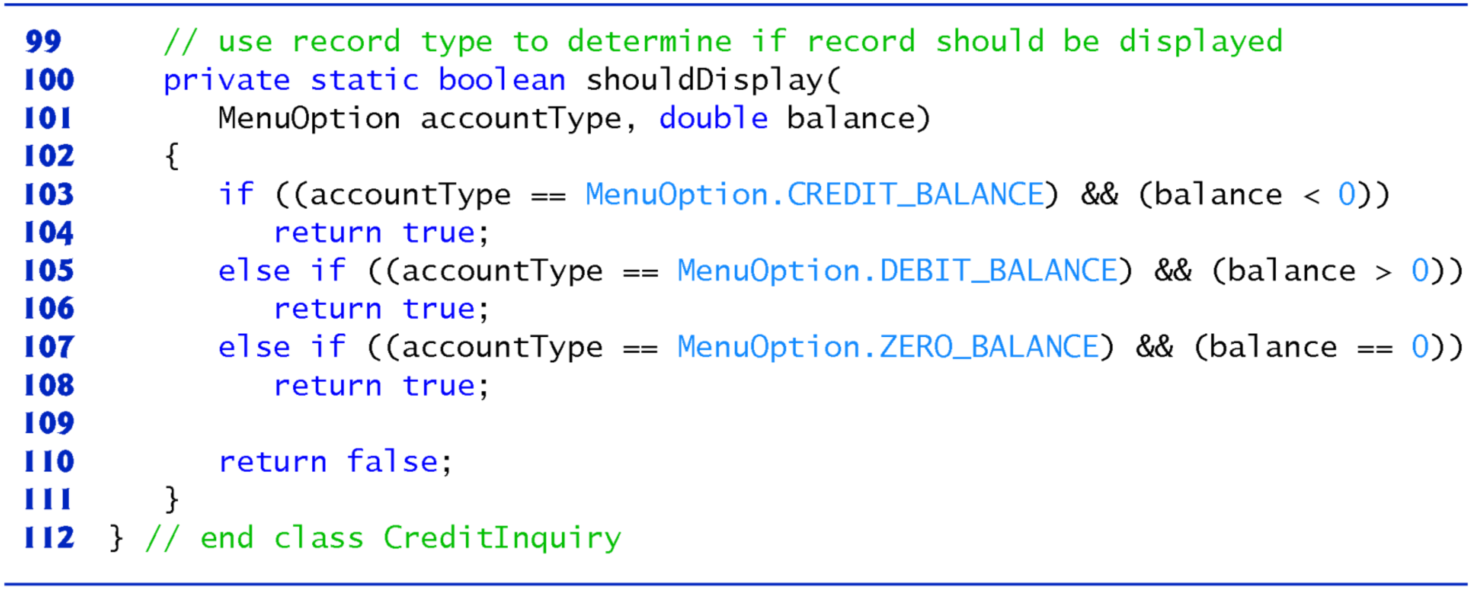
Fig. 15.8 | Credit-inquiry program. (Part 3 of 8.)

```
      catch (NoSuchElementException noSuchElementException)
      {
         System.err.println("Invalid input. Terminating.");
      }

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

   // 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();

```

Fig. 15.8 | Credit-inquiry program. (Part 4 of 8.)

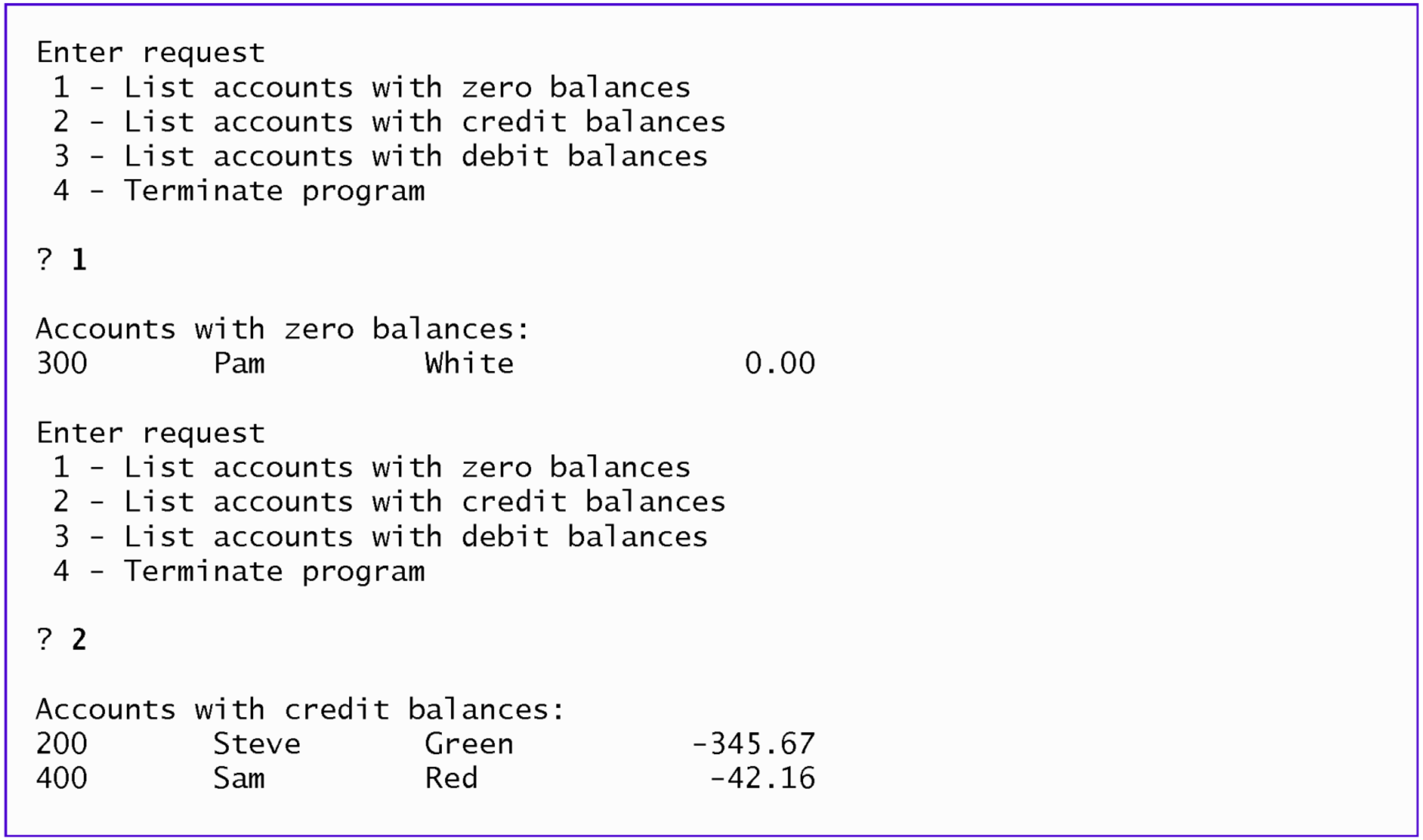
```java
            // if proper acount type, display record
            if (shouldDisplay(accountType, balance))
               System.out.printf("%-10d%-12s%-12s%10.2f%n", accountNumber,
                  firstName, lastName, balance);
            else
               input.nextLine(); // discard the rest of the current record
         }
      }
      catch (NoSuchElementException |
         IllegalStateException | IOException e)
      {
         System.err.println("Error processing file. Terminating.");
         System.exit(1);
      }
   } // end method readRecords

```

Fig. 15.8 | Credit-inquiry program. (Part 5 of 8.)



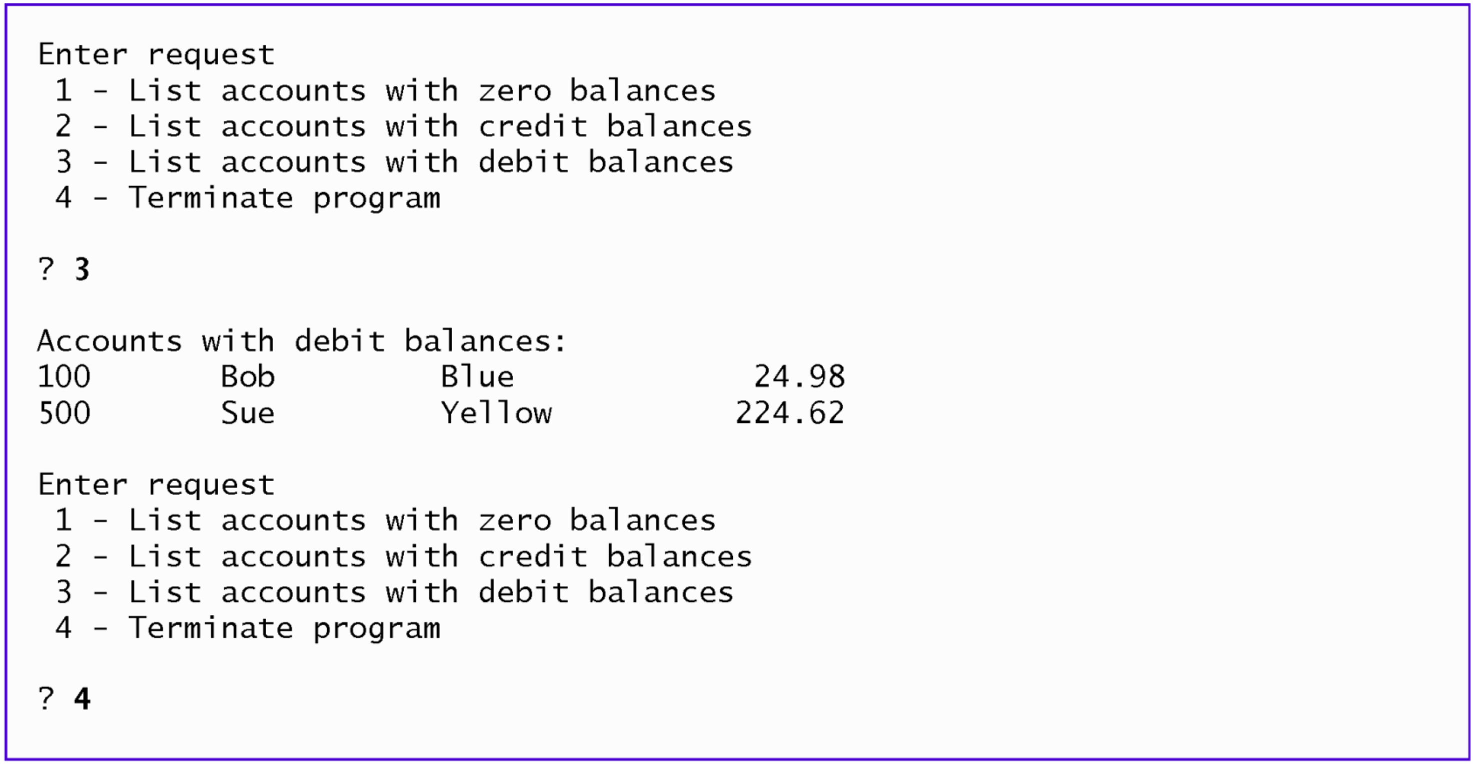
```java
   // use record type to determine if record should be displayed
   private static boolean shouldDisplay(
      MenuOption accountType, double balance)
   {
      if ((accountType == MenuOption.CREDIT_BALANCE) && (balance < 0))
         return true;
      else if ((accountType == MenuOption.DEBIT_BALANCE)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

      return false;
   }
} // end class CreditInquiry
```

Fig. 15.8 | Credit-inquiry program. (Part 6 of 8.)



```
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 7 of 8.)



```
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 8 of 8.)

15.4.4 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 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 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.

15.5 Object Serialization (cont.)

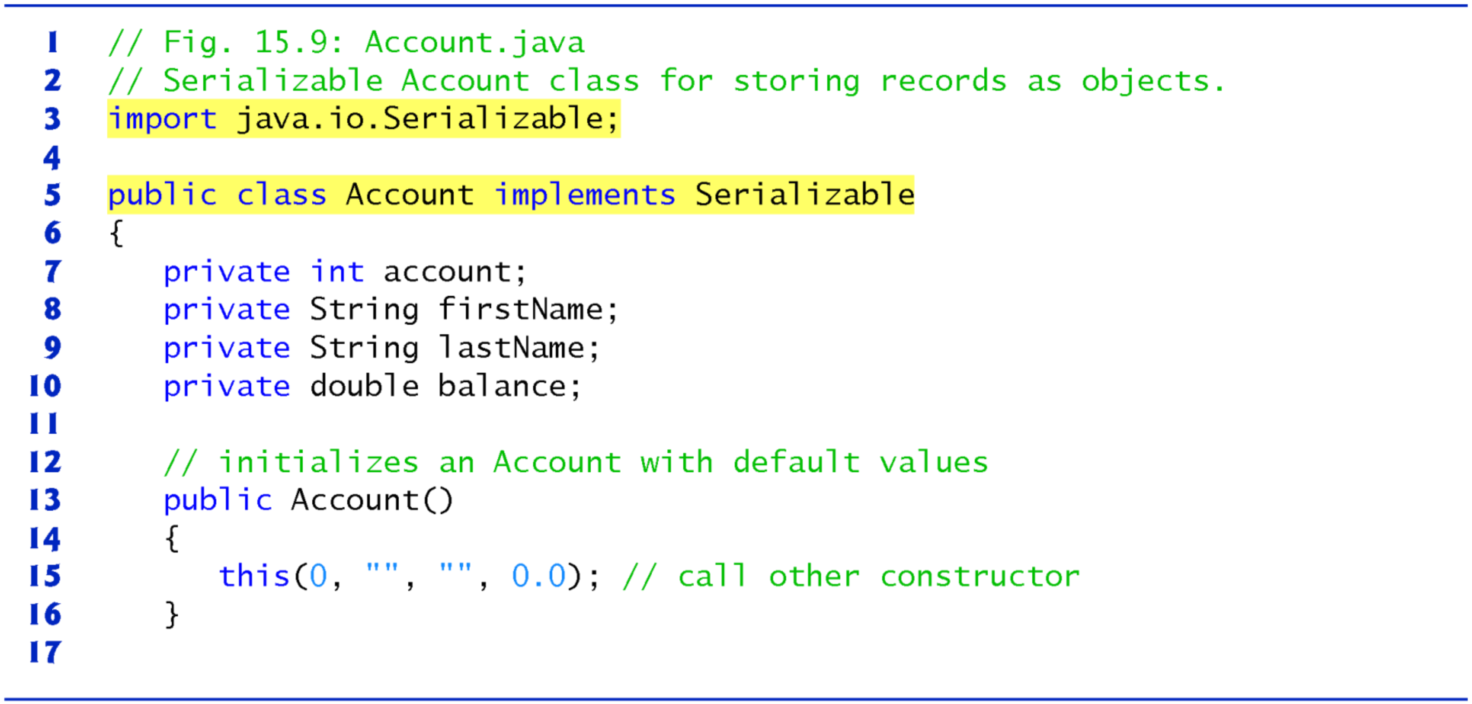
- Classes `ObjectInputStream` and `ObjectOutputStream` (package `java.io`). which respectively implement the **`ObjectInput`** and **`ObjectOutput`** interfaces, enable entire objects to be read from or written to a stream.
- To use serialization with files, initialize `ObjectInputStream` and `ObjectOutputStream` objects that read from and write to files.

15.5 Object Serialization (cont.)

- ObjectOutput interface method **writeObject** takes an Object as an argument and writes its information to an OutputStream.
- A class that implements ObjectOuput (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.

15.5.1 Creating a Sequential-Access File Using Object Serialization

- Objects of classes that implement interface `Serializable` can be *serialized* and *deserialized* with `ObjectOutputStream`s and `ObjectInputStream`s.
- 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.

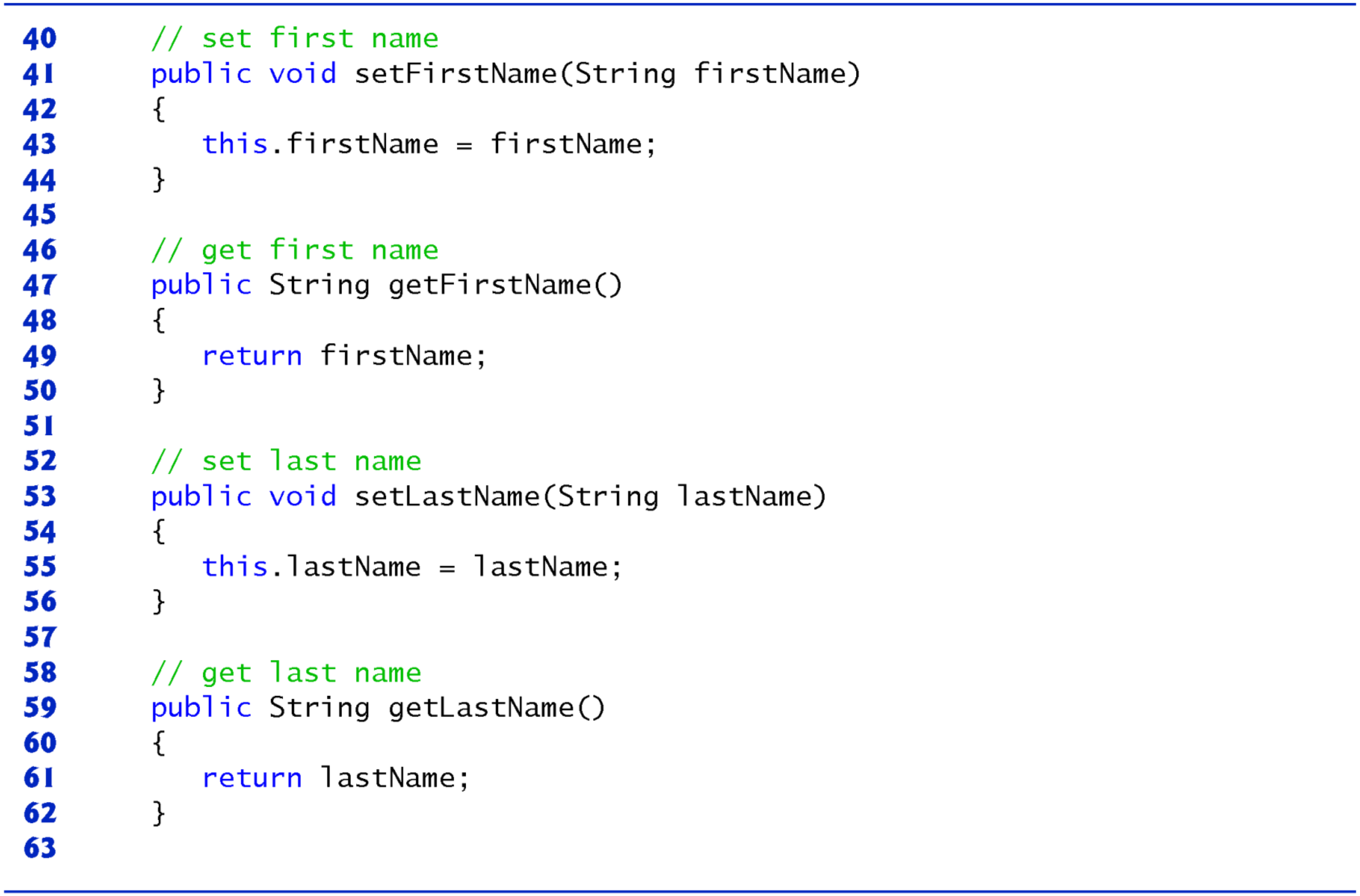


```java
// Fig. 15.9: Account.java
// Serializable Account class for storing records as objects.
import java.io.Serializable;

public class Account implements Serializable
{
   private int account;
   private String firstName;
   private String lastName;
   private double balance;

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

```

Fig. 15.9 | Account class for serializable objects. (Part 1 of 4.)

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

   // set account number
   public void setAccount(int acct)
   {
      this.account = account;
   }

   // get account number
   public int getAccount()
   {
      return account;
   }

```

Fig. 15.9 | Account class for serializable objects. (Part 2 of 4.)



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

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

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

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

```

Fig. 15.9 | Account class for serializable objects. (Part 3 of 4.)

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

   // get balance
   public double getBalance()
   {
      return balance;
   }
} // end class Account
```

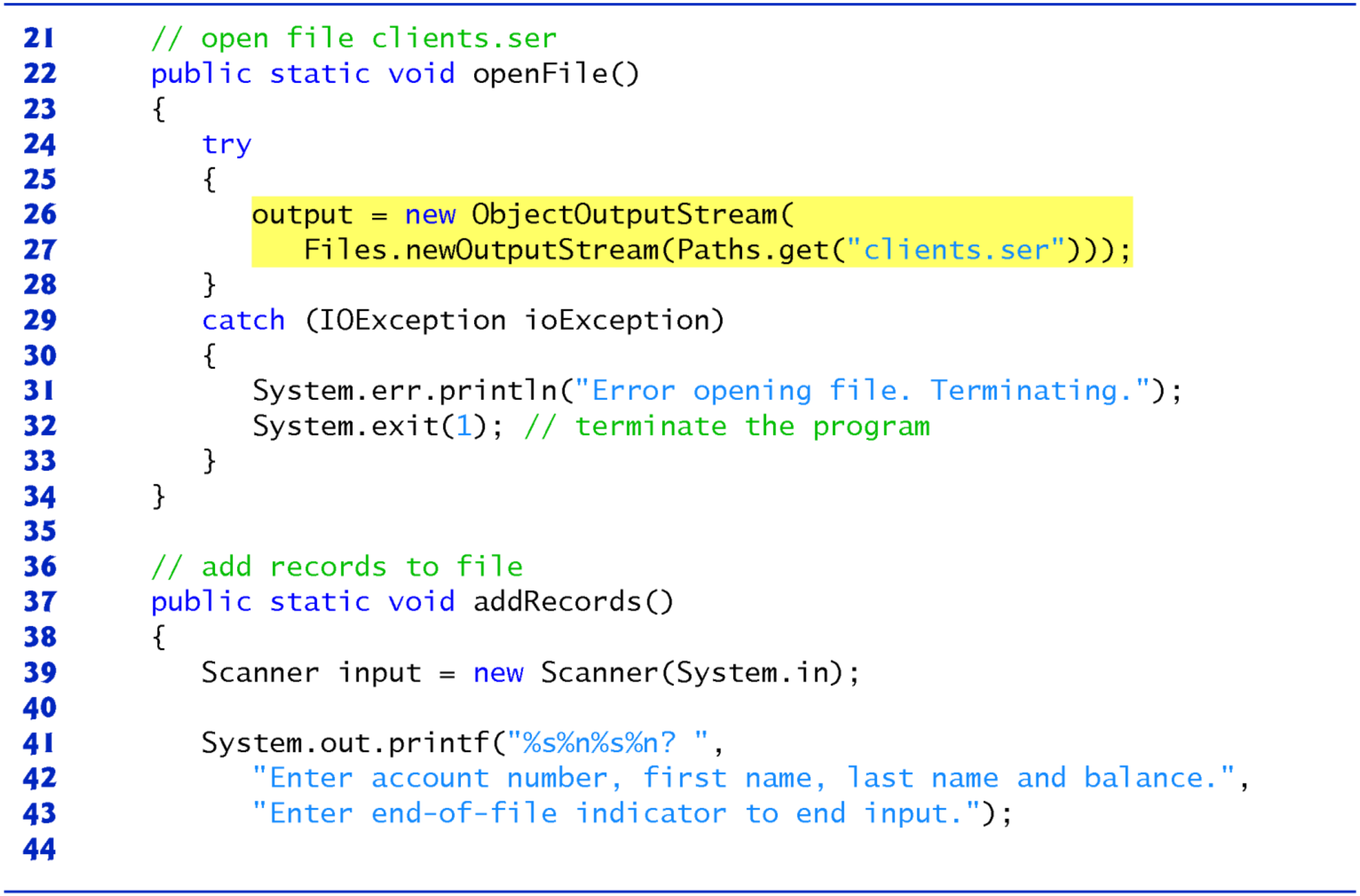
Fig. 15.9 | Account class for serializable objects. (Part 4 of 4.)

15.5.1 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`.

```java
// Fig. 15.10: CreateSequentialFile.java
// Writing objects sequentially to a file with class ObjectOutputStream.
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.NoSuchElementException;
import java.util.Scanner;

public class CreateSequentialFile
{
   private static ObjectOutputStream output; // outputs data to file

   public static void main(String[] args)
   {
      openFile();
      addRecords();
      closeFile();
   }

```

Fig. 15.10 | Sequential file created using `ObjectOutputStream`. (Part 1 of 5.)

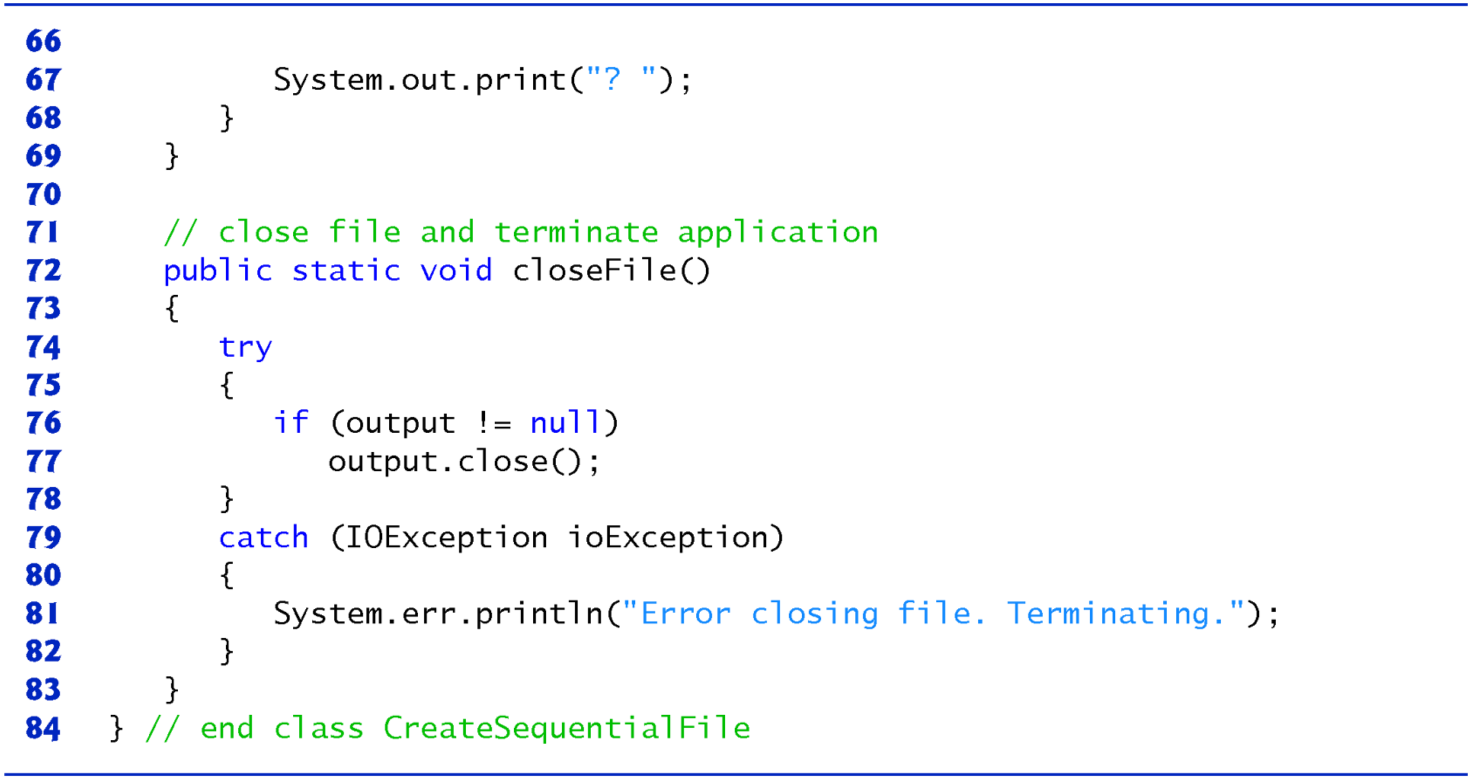


```
    // open file clients.ser
    public static void openFile()
    {
       try
       {
          output = new ObjectOutputStream(
             Files.newOutputStream(Paths.get("clients.ser")));
       }
       catch (IOException ioException)
       {
          System.err.println("Error opening file. Terminating.");
          System.exit(1); // terminate the program
       }
    }

    // add records to file
    public static void addRecords()
    {
       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. 15.10 | Sequential file created using `ObjectOutputStream`. (Part 2 of 5.)

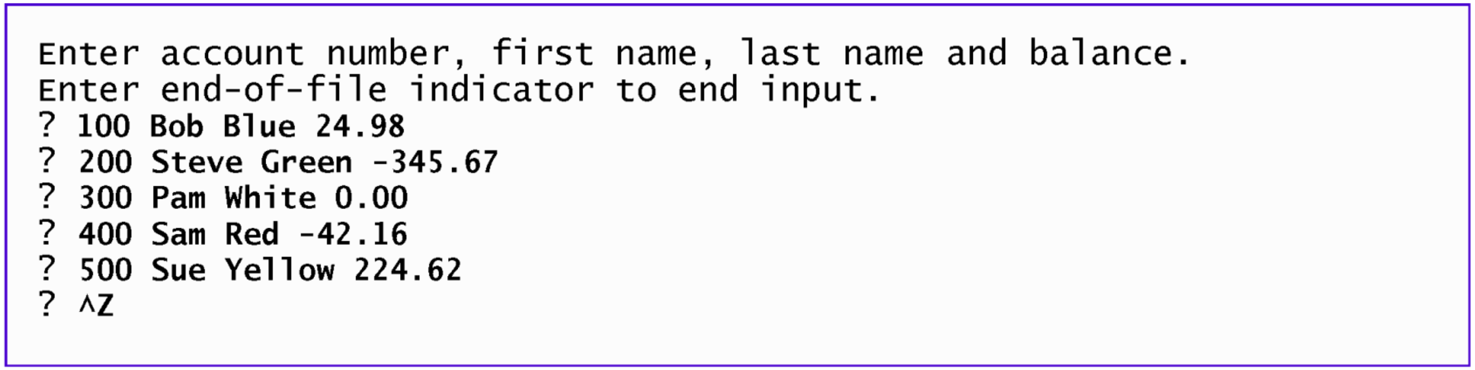
```java
      while (input.hasNext()) // loop until end-of-file indicator
      {
         try
         {
            // create new record; this example assumes valid input
            Account record = new Account(input.nextInt(),
               input.next(), input.next(), input.nextDouble());

            // serialize record object into file
            output.writeObject(record);
         }
         catch (NoSuchElementException elementException)
         {
            System.err.println("Invalid input. Please try again.");
            input.nextLine(); // discard input so user can try again
         }
         catch (IOException ioException)
         {
            System.err.println("Error writing to file. Terminating.");
            break;
         }
```

Fig. 15.10 | Sequential file created using `ObjectOutputStream`. (Part 3 of 5.)



```java

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

   // close file and terminate application
   public static void closeFile()
   {
      try
      {
         if (output != null)
            output.close();
      }
      catch (IOException ioException)
      {
         System.err.println("Error closing file. Terminating.");
      }
   }
} // end class CreateSequentialFile
```

Fig. 15.10 | Sequential file created using `ObjectOutputStream`. (Part 4 of 5.)

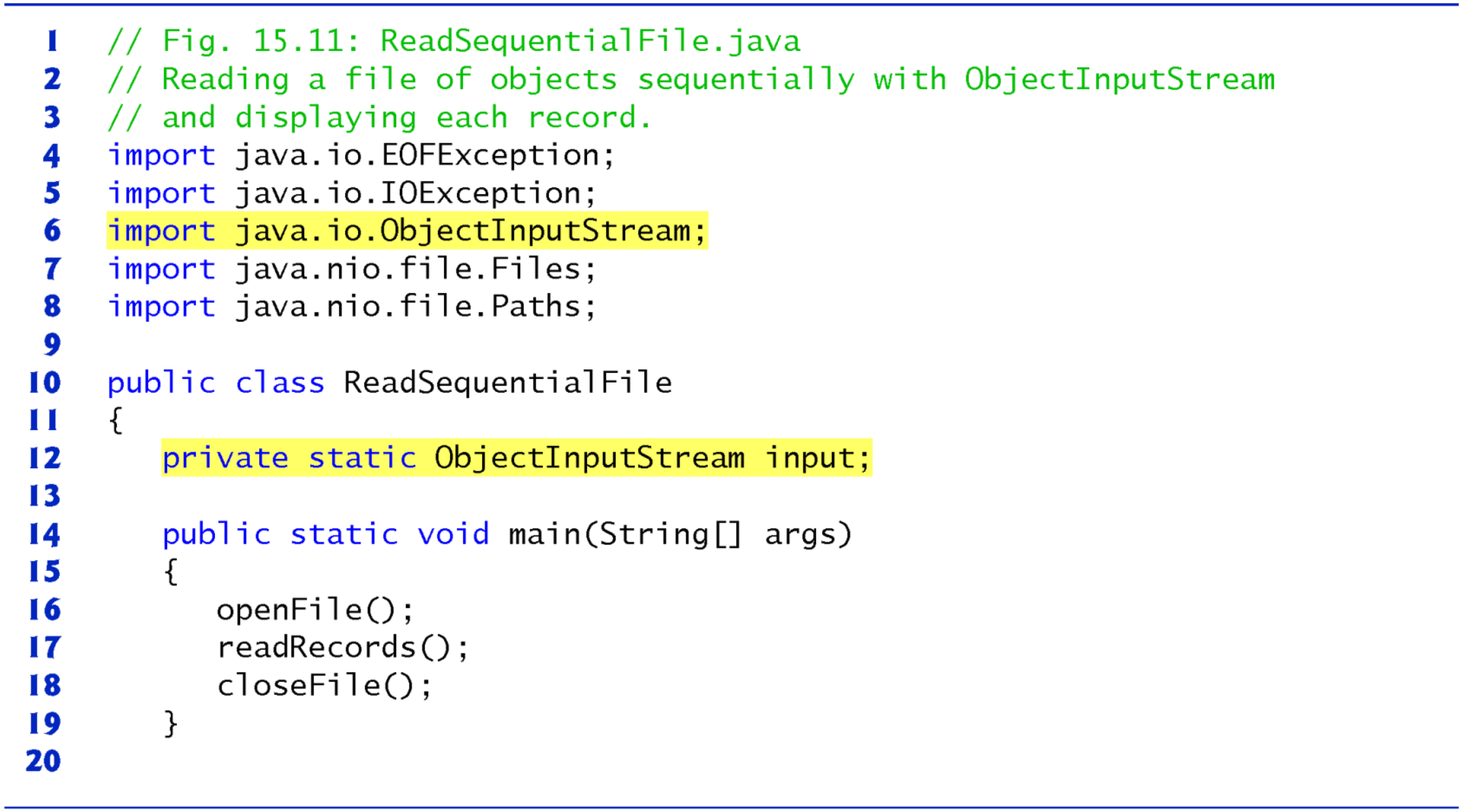


```
Enter account number, first name, last name and balance.
Enter end-of-file indicator to end input.
? 100 Bob Blue 24.98
? 200 Steve Green -345.67
? 300 Pam White 0.00
? 400 Sam Red -42.16
? 500 Sue Yellow 224.62
? ^Z
```

Fig. 15.10 | Sequential file created using `ObjectOutputStream`. (Part 5 of 5.)

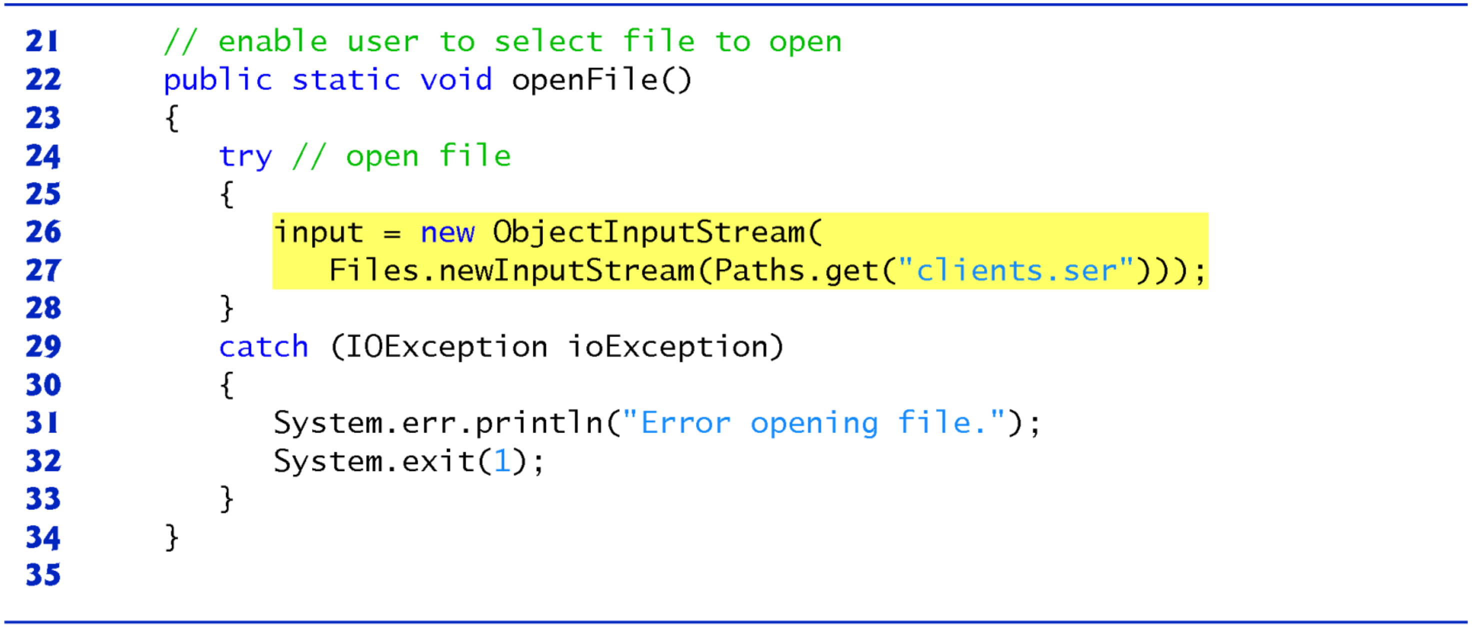
15.5.2 Reading and Deserializing Data from a Sequential-Access File

- The program in Fig. 15.11 reads records from a file created by the program in Section 15.5.1 and displays the contents.



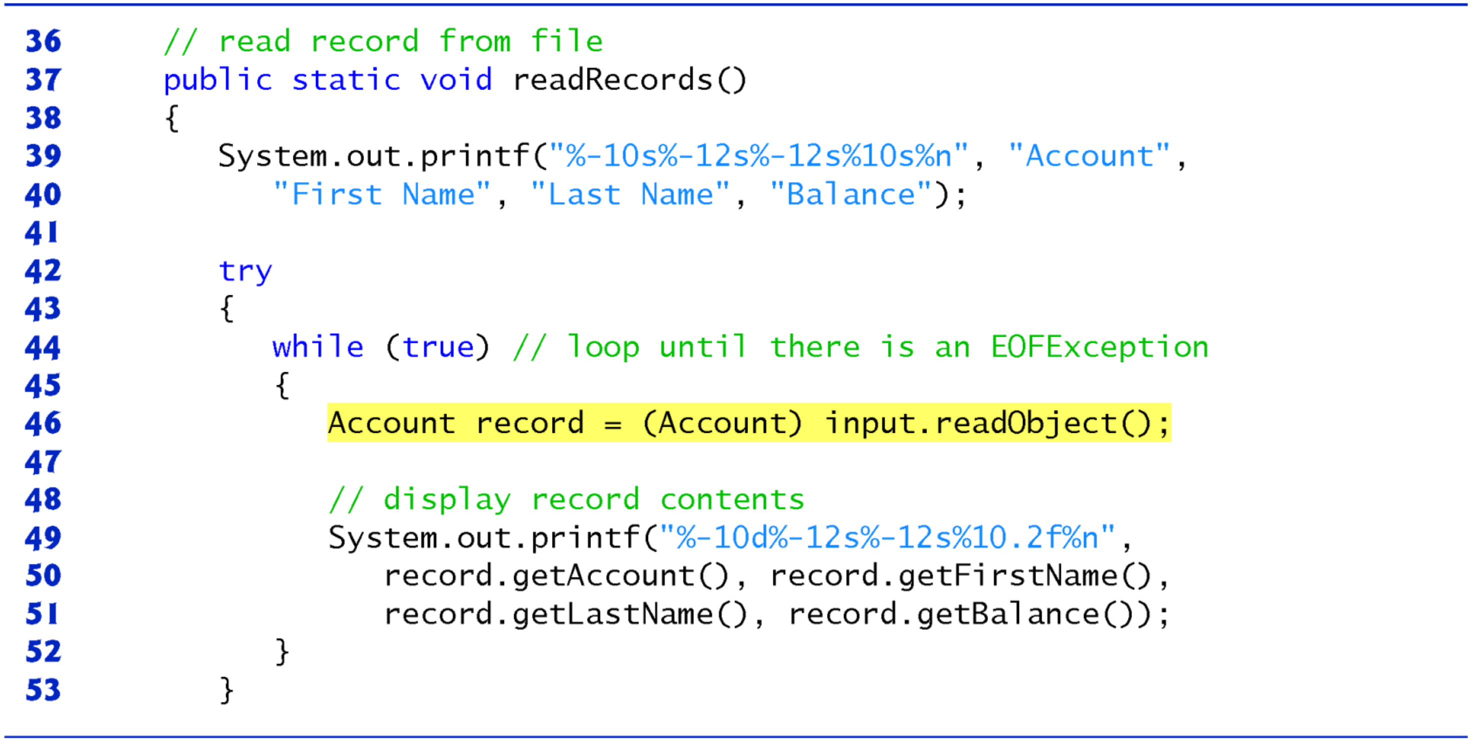
```java
// Fig. 15.11: ReadSequentialFile.java
// Reading a file of objects sequentially with ObjectInputStream
// and displaying each record.
import java.io.EOFException;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.nio.file.Files;
import java.nio.file.Paths;

public class ReadSequentialFile
{
   private static ObjectInputStream input;

   public static void main(String[] args)
   {
      openFile();
      readRecords();
      closeFile();
   }

```

Fig. 15.11 | Reading a file of objects sequentially with `ObjectInputStream` and displaying each record. (Part 1 of 6.)



```java
    // enable user to select file to open
    public static void openFile()
    {
       try // open file
       {
          input = new ObjectInputStream(
             Files.newInputStream(Paths.get("clients.ser")));
       }
       catch (IOException ioException)
       {
          System.err.println("Error opening file.");
          System.exit(1);
       }
    }

```

Fig. 15.11 | Reading a file of objects sequentially with `ObjectInputStream` and displaying each record. (Part 2 of 6.)



```java
   // read record from file
   public static void readRecords()
   {
      System.out.printf("%-10s%-12s%-12s%10s%n", "Account",
         "First Name", "Last Name", "Balance");

      try
      {
         while (true) // loop until there is an EOFException
         {
            Account record = (Account) input.readObject();

            // display record contents
            System.out.printf("%-10d%-12s%-12s%10.2f%n",
               record.getAccount(), record.getFirstName(),
               record.getLastName(), record.getBalance());
         }
      }
```

Fig. 15.11 | Reading a file of objects sequentially with `ObjectInputStream` and displaying each record. (Part 3 of 6.)

```
      catch (EOFException endOfFileException)
      {
         System.out.printf("%No more records%n");
      }
      catch (ClassNotFoundException classNotFoundException)
      {
         System.err.println("Invalid object type. Terminating.");
      }
      catch (IOException ioException)
      {
         System.err.println("Error reading from file. Terminating.");
      }
   } // end method readRecords

```

Fig. 15.11 | Reading a file of objects sequentially with `ObjectInputStream` and displaying each record. (Part 4 of 6.)

```java
   // close file and terminate application
   public static void closeFile()
   {
      try
      {
         if (input != null)
            input.close();
      }
      catch (IOException ioException)
      {
         System.err.println("Error closing file. Terminating.");
         System.exit(1);
      }
   }
} // end class ReadSequentialFile
```

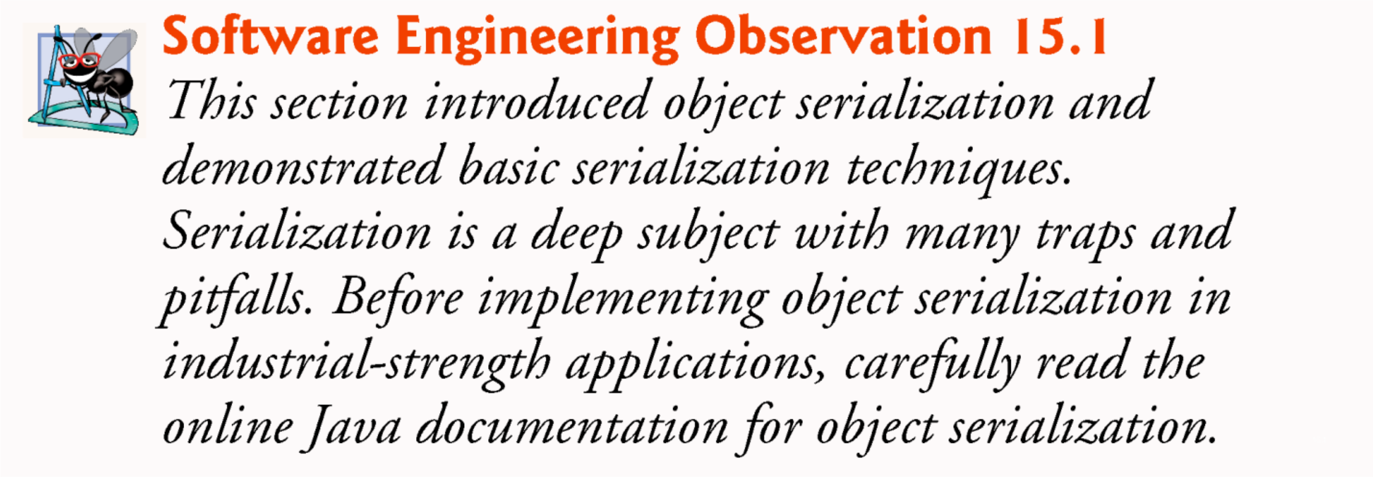
Fig. 15.11 | Reading a file of objects sequentially with `ObjectInputStream` and displaying each record. (Part 5 of 6.)

```
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. 15.11 | Reading a file of objects sequentially with `ObjectInputStream` and displaying each record. (Part 6 of 6.)

15.5.2 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.



Software Engineering Observation 15.1

This section introduced object serialization and demonstrated basic serialization techniques. Serialization is a deep subject with many traps and pitfalls. Before implementing object serialization in industrial-strength applications, carefully read the online Java documentation for object serialization.

15.6 Opening Files with JFileChooser

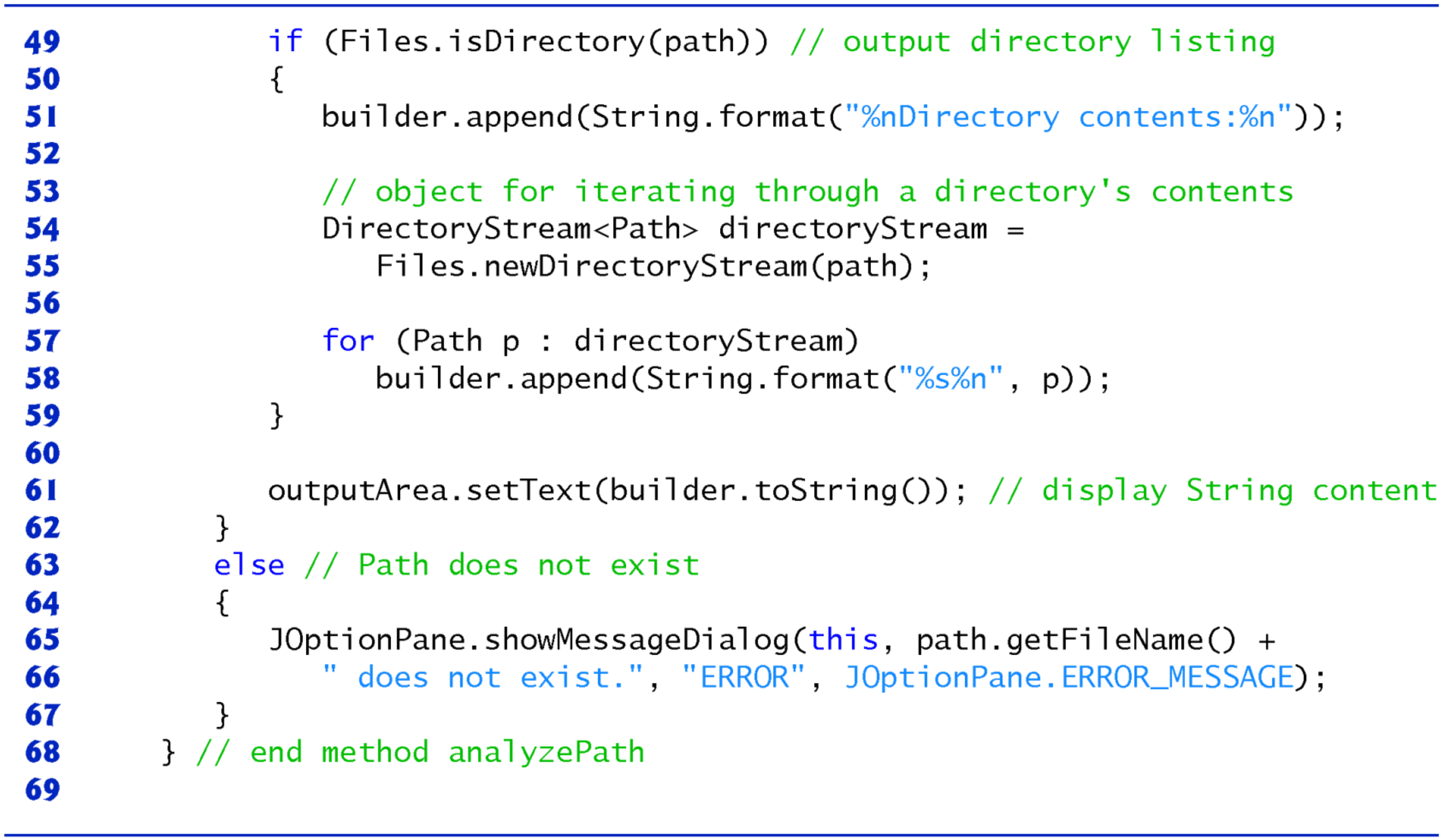
- Class **JFileChooser** displays a dialog that enables the user to easily select files or directories.
- To demonstrate JFileChooser, we enhance the example in Section 15.3, as shown in Figs. 15.12–15.13.
- Call method **setFileSelectionMode** specifies what the user can select from the fileChooser. For this program, we use JFileChooser static constant **FILES_AND_DIRECTORIES** to indicate that files and directories can be selected. Other static constants include **FILES_ONLY** (the default) and **DIRECTORIES_ONLY**.

```java
// Fig. 15.12: JFileChooserDemo.java
// Demonstrating JFileChooser.
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 javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

public class JFileChooserDemo extends JFrame
{
   private final JTextArea outputArea; // displays file contents

   // set up GUI
   public JFileChooserDemo() throws IOException
   {
      super("JFileChooser Demo");
      outputArea = new JTextArea();
      add(new JScrollPane(outputArea)); // outputArea is scrollable
      analyzePath(); // get Path from user and display info
   }
```

Fig. 15.12 | Demonstrating JFileChooser. (Part 1 of 4.)

```java
   // display information about file or directory user specifies
   public void analyzePath() throws IOException
   {
      // get Path to user-selected file or directory
      Path path = getFileOrDirectoryPath();

      if (path != null && Files.exists(path)) // if exists, display info
      {
         // gather file (or directory) information
         StringBuilder builder = new StringBuilder();
         builder.append(String.format("%s:%n", path.getFileName()));
         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()));
```

Fig. 15.12 | Demonstrating JFileChooser. (Part 2 of 4.)



```java
         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));
         }

         outputArea.setText(builder.toString()); // display String content
      }
      else // Path does not exist
      {
         JOptionPane.showMessageDialog(this, path.getFileName() +
            " does not exist.", "ERROR", JOptionPane.ERROR_MESSAGE);
      }
   } // end method analyzePath

```

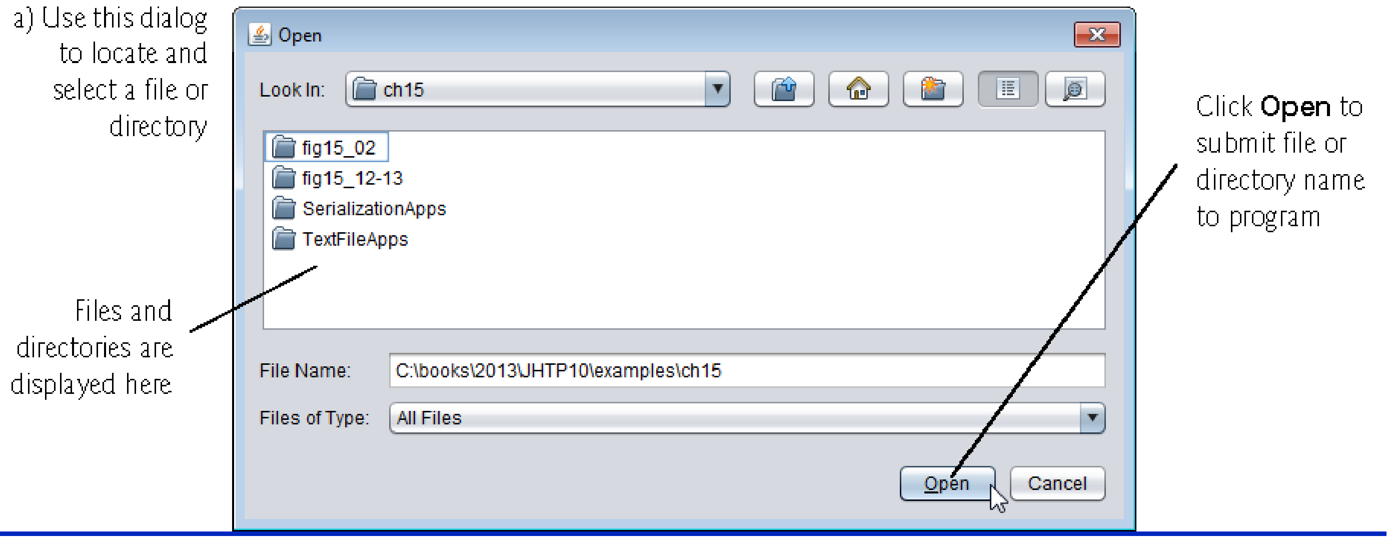
Fig. 15.12 | Demonstrating `JFileChooser`. (Part 3 of 4.)

```
   // allow user to specify file or directory name
   private Path getFileOrDirectoryPath()
   {
      // configure dialog allowing selection of a file or directory
      JFileChooser fileChooser = new JFileChooser();
      fileChooser.setFileSelectionMode(
         JFileChooser.FILES_AND_DIRECTORIES);
      int result = fileChooser.showOpenDialog(this);

      // if user clicked Cancel button on dialog, return
      if (result == JFileChooser.CANCEL_OPTION)
         System.exit(1);

      // return Path representing the selected file
      return fileChooser.getSelectedFile().toPath();
   }
} // end class JFileChooserDemo
```

Fig. 15.12 | Demonstrating `JFileChooser`. (Part 4 of 4.)

```java
// Fig. 15.13: JFileChooserTest.java
// Tests class JFileChooserDemo.
import java.io.IOException;
import javax.swing.JFrame;

public class JFileChooserTest
{
   public static void main(String[] args) throws IOException
   {
      JFileChooserDemo application = new JFileChooserDemo();
      application.setSize(400, 400);
      application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      application.setVisible(true);
   }
} // end class JFileChooserTest
```

Fig. 15.13 | Testing class `FileDemonstration`. (Part 1 of 3.)



a) Use this dialog to locate and select a file or directory

Files and directories are displayed here

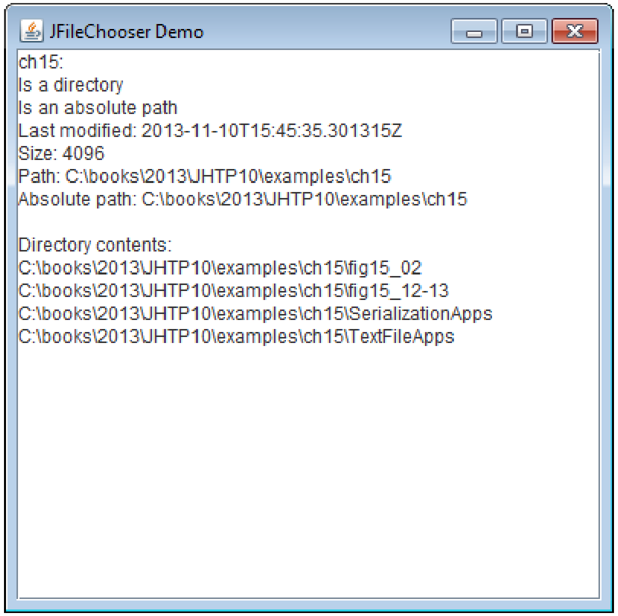
Click **Open** to submit file or directory name to program

File Name: C:\books\2013\JHTP10\examples\ch15

Files of Type: All Files

Fig. 15.13 | Testing class `FileDemonstration`. (Part 2 of 3.)

b) Selected file's or directory's information; if it's a directory, the contents of that directory are displayed



```
ch15:
Is a directory
Is an absolute path
Last modified: 2013-11-10T15:45:35.301315Z
Size: 4096
Path: C:\books\2013\JHTP10\examples\ch15
Absolute path: C:\books\2013\JHTP10\examples\ch15

Directory contents:
C:\books\2013\JHTP10\examples\ch15\fig15_02
C:\books\2013\JHTP10\examples\ch15\fig15_12-13
C:\books\2013\JHTP10\examples\ch15\SerializationApps
C:\books\2013\JHTP10\examples\ch15\TextFileApps
```

Fig. 15.13 | Testing class `FileDemonstration`. (Part 3 of 3.)

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.)

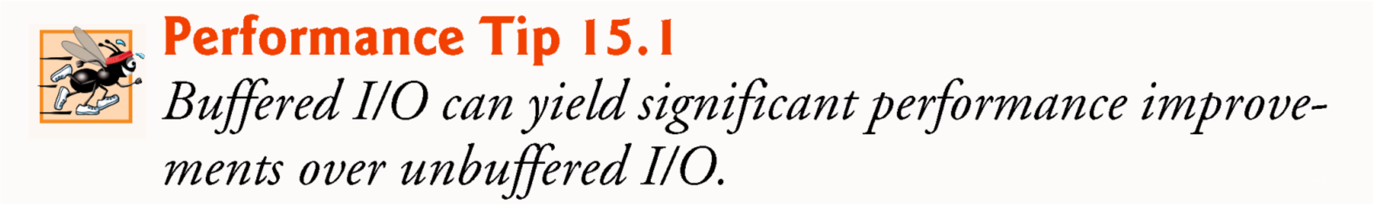
- Usually, programs read data as aggregates of bytes that form `int`s, `float`s, `double`s and so on.
- Java programs can use several classes to input and output data in aggregate form.
- Interface `DataInput` describes methods for reading primitive types from an input stream.
- Classes **`DataInputStream`** 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 `InputStream`s
- 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 OuputStream can be converted to a Writer via class **OutputStreamWriter**.
- Class File-Reader and class FileWriter read characters from and write characters to a file.
- Class **PipedReader** and class **PipedWriter** implement piped-character streams for transfering data between threads.
- Class **StringReader** b**StringWriter** read characters from and write characters to Strings.
- A PrintWriter writes characters to a stream.