Lesson 11 – Part I
Files, Streams and Object Serialization

Assoc. Prof. Marenglen Biba

In this Chapter you'll learn:

- What files are and how they are used to retain application data between successive executions.
- To create, read, write and update files.
- To use class `File` to retrieve information about files and directories.
- The Java input/output stream class hierarchy.
- The differences between text files and binary files.
- Sequential-access file processing.
- To use classes `Scanner` and `Formatter` to process text files.
- To use classes `FileInputStream` and `FileOutputStream` to read from and write to files.
- To use classes `ObjectInputStream` and `ObjectOutputStream` to read objects from and write objects to files.
- To use a `JFileChooser` dialog.

17.1 Introduction

17.2 Data Hierarchy

17.3 Files and Streams

17.4 Class `File`

17.5 Sequential-Access Text Files

17.5.1 Creating a Sequential-Access Text File
17.5.2 Reading Data from a Sequential-Access Text File
17.5.3 Case Study: A Credit-Inquiry Program
17.5.4 Updating Sequential-Access Files

17.6 Object Serialization

17.6.1 Creating a Sequential-Access File Using Object Serialization
17.6.2 Reading and Deserializing Data from a Sequential-Access File

17.7 Additional `java.io` Classes

17.7.1 Interfaces and Classes for Byte-Based Input and Output
17.7.2 Interfaces and Classes for Character-Based Input and Output

17.8 Opening Files with `JFileChooser`

17.9 Wrap-Up

17.1 Introduction

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

17.2 Data Hierarchy: Character Set

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

17.2 Data Hierarchy: fields

- **Fields** are composed of characters or bytes.

- A field is a **group of characters** or **bytes** that conveys meaning.

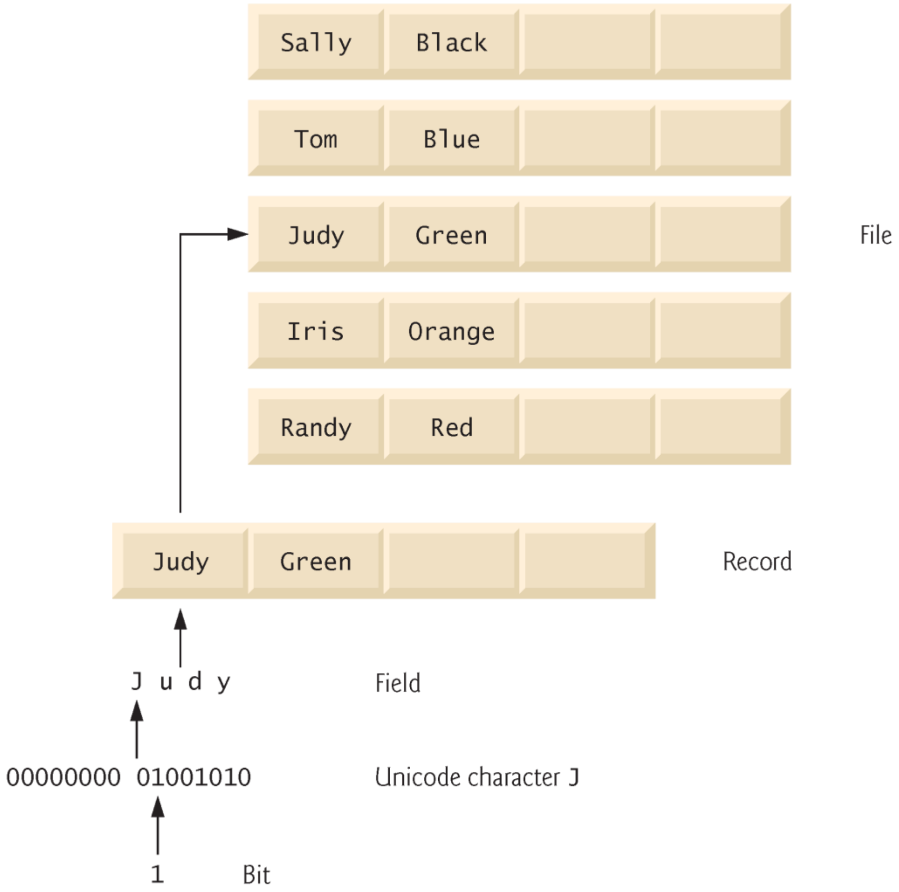
- Data items processed by computers form a **data hierarchy** that becomes larger and more complex in structure as we progress from bits to characters to fields, and so on.

17.2 Data Hierarchy (cont.)

- Typically, several fields compose a **record** (implemented as a `class` in Java).

- A record is a group of related fields.

- A **file** is a group of related records.



Sally Black

Tom Blue

Judy Green

Iris Orange

Randy Red

File

Judy Green

Record

J u d y

Field

00000000 01001010

Unicode character J

1

Bit

Fig. 17.1 | Data hierarchy.

17.3 Files and Streams

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

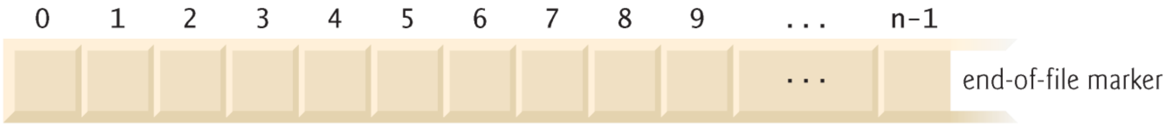


Fig. 17.2 | Java's view of a file of *n* bytes.

17.3 Files and Streams (cont.)

- **File streams** can be used to input and output data as bytes or characters.

- Streams that input and output bytes are known as **byte-based streams**, representing data in its binary format.

- Streams that input and output characters are known as **character-based streams**, representing data as a sequence of characters.

17.3 Files and Streams (cont.)

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

17.3 Files and Streams (cont.)

- A Java program **opens** a file by creating an object and associating a stream of bytes or characters with it.

- Java creates three stream objects when a program begins executing
  - `System.in` (the standard input stream object) normally inputs bytes from the keyboard
  - `System.out` (the standard output stream object) normally outputs character data to the screen
  - `System.err` (the standard error stream object) normally outputs character-based error messages to the screen.

- Class `System` provides methods **`setIn`**, **`setOut`** and **`setErr`** to **redirect** the standard input, output and error streams, respectively.

17.3 Files and Streams (cont.)

- Java programs perform file processing by using classes from package **`java.io`**.

- Includes definitions for stream classes
  - **`FileInputStream`** (for byte-based input from a file)
  - **`FileOutputStream`** (for byte-based output to a file)
  - **`FileReader`** (for character-based input from a file)
  - **`FileWriter`** (for character-based output to a file)

- You open a file by creating an object of one these stream classes. The object's **constructor** opens the file.

17.3 Files and Streams (cont.)

- Java can perform input and output of objects or variables of primitive data types without having to worry about the details of converting such values to byte format.

- To perform such input and output, objects of classes `ObjectInputStream` and `ObjectOutputStream` can be used together with the byte-based file stream classes `FileInputStream` and `FileOutputStream`.

- The complete hierarchy of classes in package `java.io` can be viewed in the online documentation at
  - http://docs.oracle.com/javase/8/docs/api/java/io/package-tree.html

17.3 Files and Streams (cont.)

- Class **`File`** provides information about files and directories.

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

17.4 Class File

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

17.4 Class `File` (cont.)

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

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

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

Method	Description
`String getParent()`	Returns a `String` with the parent directory of the file or directory (i.e., the directory in which the file or directory is located).
`long length()`	Returns the length of the file, in bytes. If the `File` object represents a directory, an unspecified value is returned.
`long lastModified()`	Returns a platform-dependent representation of the time at which the file or directory was last modified. The value returned is useful only for comparison with other values returned by this method.
`String[] list()`	Returns an array of `Strings` representing a directory's contents. Returns `null` if the `File` object does not represent a directory.

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

```java
// Fig. 17.4: FileDemonstration.java
// File class used to obtain file and directory information.
import java.io.File;
import java.util.Scanner;

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

      System.out.print( "Enter file or directory name: " );
      analyzePath( input.nextLine() );
   } // end main

   // display information about file user specifies
   public static void analyzePath( String path )
   {
      // create File object based on user input
      File name = new File( path );

```

Associates a file or directory with a `File` object.

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

```java
      if ( name.exists() ) // if name exists, output information about it
      {
         // display file (or directory) information
         System.out.printf(
            "%s%s\n%s\n%s\n%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s",
            name.getName(), " exists",
            ( name.isFile() ? "is a file" : "is not a file" ),
            ( name.isDirectory() ? "is a directory" :
               "is not a directory" ),
            ( name.isAbsolute() ? "is absolute path" :
               "is not absolute path" ), "Last modified: ",
            name.lastModified(), "Length: ", name.length(),
            "Path: ", name.getPath(), "Absolute path: ",
            name.getAbsolutePath(), "Parent: ", name.getParent() );

         if ( name.isDirectory() ) // output directory listing
         {
            String[] directory = name.list();
            System.out.println( "\n\nDirectory contents:\n" );

            for ( String directoryName : directory )
               System.out.println( directoryName );
         } // end if
      } // end outer if
```

Determines if the file or directory exists.

Returns an array of `Strings` containing the directory's contents.

Fig. 17.4 | `File` class used to obtain file and directory information. (Part 2 of 5.)

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

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

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

Directory contents:

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

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

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

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

17.4 Class `File` (cont.)

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

Common Programming Error 17.1

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

17.5 Sequential-Access Text Files

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

17.5.1 Creating a Sequential-Access Text File

- Java imposes no structure on a file

- Notions such as records **do not exist** as part of the Java language.

```java
// Fig. 17.5: AccountRecord.java
// AccountRecord class maintains information for one account.
package com.deitel.ch17; // packaged for reuse

public class AccountRecord
{
   private int account;
   private String firstName;
   private String lastName;
   private double balance;

   // no-argument constructor calls other constructor with default values
   public AccountRecord()
   {
      this( 0, "", "", 0.0 ); // call four-argument constructor
   } // end no-argument AccountRecord constructor

```

Fig. 17.5 | `AccountRecord` class maintains information for one account. (Part 1 of 4.)

```
   // initialize a record
   public AccountRecord( int acct, String first, String last, double bal )
   {
      setAccount( acct );
      setFirstName( first );
      setLastName( last );
      setBalance( bal );
   } // end four-argument AccountRecord constructor

   // set account number
   public void setAccount( int acct )
   {
      account = acct;
   } // end method setAccount

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

```

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

```
   // set first name
   public void setFirstName( String first )
   {
      firstName = first;
   } // end method setFirstName

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

   // set last name
   public void setLastName( String last )
   {
      lastName = last;
   } // end method setLastName

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

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

```java

   // set balance
   public void setBalance( double bal )
   {
      balance = bal;
   } // end method setBalance

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

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

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

- `Formatter` outputs formatted `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. 17.6: 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;

import com.deitel.ch17.AccountRecord;

public class CreateTextFile
{
   private Formatter output; // object used to output text to file

   // enable user to open file
   public void openFile()
   {
      try
      {
         output = new Formatter( "clients.txt" ); // open the file
      } // end try
```

Used to output text to a file.

Opens the file `clients.txt`.

Fig. 17.6 | Writing data to a sequential text file with class `Formatter`. (Part 1 of 5.)

```java
      catch ( SecurityException securityException )
      {
         System.err.println(
            "You do not have write access to this file." );
         System.exit( 1 ); // terminate the program
      } // end catch
      catch ( FileNotFoundException fileNotFoundException )
      {
         System.err.println( "Error opening or creating file." );
         System.exit( 1 ); // terminate the program
      } // end catch
   } // end method openFile

   // add records to file
   public void addRecords()
   {
      // object to be written to file
      AccountRecord record = new AccountRecord();

      Scanner input = new Scanner( System.in );

```

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

```
      System.out.printf( "%s\n%s\n%s\n%s\n\n",
         "To terminate input, type the end-of-file indicator ",
         "when you are prompted to enter input.",
         "On UNIX/Linux/Mac OS X type <ctrl> d then press Enter",
         "On Windows type <ctrl> z then press Enter" );

      System.out.printf( "%s\n%s",
         "Enter account number (> 0), first name, last name and balance.",
         "? " );

      while ( input.hasNext() ) // loop until end-of-file indicator
      {
         try // output values to file
         {
            // retrieve data to be output
            record.setAccount( input.nextInt() ); // read account number
            record.setFirstName( input.next() ); // read first name
            record.setLastName( input.next() ); // read last name
            record.setBalance( input.nextDouble() ); // read balance

            if ( record.getAccount() > 0 )
            {
```

Has end-of-file been reached?

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

```
               // write new record
               output.format( "%d %s %s %.2f\n", record.getAccount(),
                  record.getFirstName(), record.getLastName(),
                  record.getBalance() );
            } // end if
            else
            {
               System.out.println(
                  "Account number must be greater than 0." );
            } // end else
         } // end try
         catch ( FormatterClosedException formatterClosedException )
         {
            System.err.println( "Error writing to file." );
            return;
         } // end catch
         catch ( NoSuchElementException elementException )
         {
            System.err.println( "Invalid input. Please try again." );
            input.nextLine(); // discard input so user can try again
         } // end catch

```

Writes text to the file associated with `output`.

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

```java
         System.out.printf( "%s %s\n%s", "Enter account number (>0),",
            "first name, last name and balance.", "? " );
      } // end while
   } // end method addRecords

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

Closes the file.

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

```java
// Fig. 17.8: CreateTextFileTest.java
// Testing the CreateTextFile class.

public class CreateTextFileTest
{
   public static void main( String[] args )
   {
      CreateTextFile application = new CreateTextFile();

      application.openFile();
      application.addRecords();
      application.closeFile();
   } // end main
} // end class CreateTextFileTest
```

Fig. 17.8 | Testing the `CreateTextFile` class. (Part 1 of 2.)

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

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

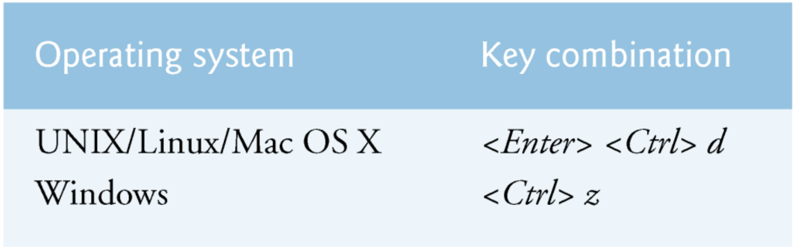
Fig. 17.8 | Testing the `CreateTextFile` class. (Part 2 of 2.)

Sample data			
100	Bob	Jones	24.98
200	Steve	Doe	-345.67
300	Pam	White	0.00
400	Sam	Stone	-42.16
500	Sue	Rich	224.62

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

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

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



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

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

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

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

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

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

17.5.2 Reading Data from a Sequential-Access Text File

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

```java
// Fig. 17.10: ReadTextFile.java
// This program reads a text file and displays each record.
import java.io.File;
import java.io.FileNotFoundException;
import java.lang.IllegalStateException;
import java.util.NoSuchElementException;
import java.util.Scanner;

import com.deitel.ch17.AccountRecord;

public class ReadTextFile
{
   private Scanner input;

   // enable user to open file
   public void openFile()
   {
      try
      {
         input = new Scanner( new File( "clients.txt" ) );
      } // end try
```

Opens clients.txt for reading.

Fig. 17.10 | Sequential file reading using a `Scanner`. (Part 1 of 4.)

```
      catch ( FileNotFoundException fileNotFoundException )
      {
         System.err.println( "Error opening file." );
         System.exit( 1 );
      } // end catch
   } // end method openFile

   // read record from file
   public void readRecords()
   {
      // object to be written to screen
      AccountRecord record = new AccountRecord();

      System.out.printf( "%-10s%-12s%-12s%10s\n", "Account",
         "First Name", "Last Name", "Balance" );

      try // read records from file using Scanner object
      {
         while ( input.hasNext() )
         {
            record.setAccount( input.nextInt() ); // read account number
            record.setFirstName( input.next() ); // read first name
            record.setLastName( input.next() ); // read last name
            record.setBalance( input.nextDouble() ); // read balance
```

Has end-of-file been reached?

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

```java

               // display record contents
               System.out.printf( "%-10d%-12s%-12s%10.2f\n",
                  record.getAccount(), record.getFirstName(),
                  record.getLastName(), record.getBalance() );
            } // end while
         } // end try
         catch ( NoSuchElementException elementException )
         {
            System.err.println( "File improperly formed." );
            input.close();
            System.exit( 1 );
         } // end catch
         catch ( IllegalStateException stateException )
         {
            System.err.println( "Error reading from file." );
            System.exit( 1 );
         } // end catch
      } // end method readRecords

```

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

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

Closes the file.

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

```java
// Fig. 17.11: ReadTextFileTest.java
// Testing the ReadTextFile class.

public class ReadTextFileTest
{
   public static void main( String[] args )
   {
      ReadTextFile application = new ReadTextFile();

      application.openFile();
      application.readRecords();
      application.closeFile();
   } // end main
} // end class ReadTextFileTest
```

```
Account    First Name   Last Name       Balance
100        Bob          Jones             24.98
200        Steve        Doe             -345.67
300        Pam          White              0.00
400        Sam          Stone            -42.16
500        Sue          Rich             224.62
```

Fig. 17.11 | Testing the `ReadTextFile` class.

17.5.3 Reading Data from a Sequential-Access Text File

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

17.5.4 Case Study: A Credit-Inquiry Program

- To **retrieve** data sequentially from a file, programs start from the beginning of the file and read all the data consecutively **until** the desired information is found.

- It might be necessary to process the file sequentially **several times** (from the beginning of the file) during the execution of a program.

- Class `Scanner` **does not allow repositioning** to the beginning of the file.
  - The program must close the file and reopen it.

```java
// Fig. 17.12: MenuOption.java
// Enumeration 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

   MenuOption( int valueOption )
   {
      value = valueOption;
   } // end MenuOptions enum constructor

   public int getValue()
   {
      return value;
   } // end method getValue
} // end enum MenuOption
```

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

```java
// Fig. 17.13: 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.File;
import java.io.FileNotFoundException;
import java.lang.IllegalStateException;
import java.util.NoSuchElementException;
import java.util.Scanner;

import com.deitel.ch17.AccountRecord;

public class CreditInquiry
{
   private MenuOption accountType;
   private Scanner input;
   private final static MenuOption[] choices = { MenuOption.ZERO_BALANCE,
      MenuOption.CREDIT_BALANCE, MenuOption.DEBIT_BALANCE,
      MenuOption.END };

   // read records from file and display only records of appropriate type
   private void readRecords()
   {
```

Fig. 17.13 | Credit-inquiry program. (Part 1 of 6.)

```java
      // object to store data that will be written to file
      AccountRecord record = new AccountRecord();

      try // read records
      {
         // open file to read from beginning
         input = new Scanner( new File( "clients.txt" ) );

         while ( input.hasNext() ) // input the values from the file
         {
            record.setAccount( input.nextInt() ); // read account number
            record.setFirstName( input.next() ); // read first name
            record.setLastName( input.next() ); // read last name
            record.setBalance( input.nextDouble() ); // read balance

            // if proper acount type, display record
            if ( shouldDisplay( record.getBalance() ) )
               System.out.printf( "%-10d%-12s%-12s%10.2f\n",
                  record.getAccount(), record.getFirstName(),
                  record.getLastName(), record.getBalance() );
         } // end while
      } // end try
```

Opens clients.txt for reading.

Fig. 17.13 | Credit-inquiry program. (Part 2 of 6.)

```java
      catch ( NoSuchElementException elementException )
      {
         System.err.println( "File improperly formed." );
         input.close();
         System.exit( 1 );
      } // end catch
      catch ( IllegalStateException stateException )
      {
         System.err.println( "Error reading from file." );
         System.exit( 1 );
      } // end catch
      catch ( FileNotFoundException fileNotFoundException )
      {
         System.err.println( "File cannot be found." );
         System.exit( 1 );
      } // end catch
      finally
      {
         if ( input != null )
            input.close(); // close the Scanner and the file
      } // end finally
   } // end method readRecords

```

Fig. 17.13 | Credit-inquiry program. (Part 3 of 6.)

```java
   // use record type to determine if record should be displayed
   private boolean shouldDisplay( 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 method shouldDisplay

   // obtain request from user
   private MenuOption getRequest()
   {
      Scanner textIn = new Scanner( System.in );
      int request = 1;

```

Fig. 17.13 | Credit-inquiry program. (Part 4 of 6.)

```java
      // display request options
      System.out.printf( "\n%s\n%s\n%s\n%s\n%s\n",
         "Enter request", " 1 - List accounts with zero balances",
         " 2 - List accounts with credit balances",
         " 3 - List accounts with debit balances", " 4 - End of run" );

      try // attempt to input menu choice
      {
         do // input user request
         {
            System.out.print( "\n? " );
            request = textIn.nextInt();
         } while ( ( request < 1 ) || ( request > 4 ) );
      } // end try
      catch ( NoSuchElementException elementException )
      {
         System.err.println( "Invalid input." );
         System.exit( 1 );
      } // end catch

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

```

Fig. 17.13 | Credit-inquiry program. (Part 5 of 6.)

```java
   public void processRequests()
   {
      // get user's request (e.g., zero, credit or debit balance)
      accountType = getRequest();

      while ( accountType != MenuOption.END )
      {
         switch ( accountType )
         {
            case ZERO_BALANCE:
               System.out.println( "\nAccounts with zero balances:\n" );
               break;
            case CREDIT_BALANCE:
               System.out.println( "\nAccounts with credit balances:\n" );
               break;
            case DEBIT_BALANCE:
               System.out.println( "\nAccounts with debit balances:\n" );
               break;
         } // end switch

         readRecords();
         accountType = getRequest();
      } // end while
   } // end method processRequests
} // end class CreditInquiry
```

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

```java
// Fig. 17.14: CreditInquiryTest.java
// This program tests class CreditInquiry.

public class CreditInquiryTest
{
   public static void main( String[] args )
   {
      CreditInquiry application = new CreditInquiry();
      application.processRequests();
   } // end main
} // end class CreditInquiryTest
```

Fig. 17.14 | Testing the `CreditInquiry` class.

```
Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run

? 1

Accounts with zero balances:

300       Pam         White             0.00

Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run

? 2

Accounts with credit balances:
200       Steve       Doe            -345.67
400       Sam         Stone           -42.16
```

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

```
Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run

? 3

Accounts with debit balances:
100        Bob         Jones              24.98
500        Sue         Rich              224.62

? 4
```

Fig. 17.15 | Sample output of the credit-inquiry program in Fig. 17.14. (Part 2 of 2.)

17.5.5 Updating Sequential-Access Files

- The data in many sequential files cannot be modified without the risk of destroying other data in the file.
- If the name "`White`" needed to be changed to "`Worthington`," the old name cannot simply be overwritten, because the new name **requires more space**.
- Fields in a text file—and hence records—can vary in size.
- Records in a sequential-access file are not usually updated in place. Instead, the entire file is usually rewritten.
- Rewriting the entire file is uneconomical to update just one record, but reasonable if a substantial number of records need to be updated.

17.6 Object Serialization

- To read an entire object from or write an entire object to a file, Java provides **object serialization**.

- A **serialized object** is represented as a sequence of bytes that includes the object’s data and its type information.

- After a serialized object has been written into a file, it can be read from the file and **deserialized** to recreate the object in memory.

Software Engineering Observation 17.1

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

17.6 Object Serialization (cont.)

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

17.6 Object Serialization (cont.)

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

17.6.1 Creating a Sequential-Access File Using Object Serialization

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

```java
// Fig. 17.16: AccountRecordSerializable.java
// AccountRecordSerializable class for serializable objects.
package com.deitel.ch17; // packaged for reuse

import java.io.Serializable;

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

   // no-argument constructor calls other constructor with default values
   public AccountRecordSerializable()
   {
      this( 0, "", "", 0.0 );
   } // end no-argument AccountRecordSerializable constructor

```

Objects of this class can be serialized.

Fig. 17.16 | `AccountRecordSerializable` class for serializable objects. (Part 1 of 4.)

```java
   // four-argument constructor initializes a record
   public AccountRecordSerializable(
      int acct, String first, String last, double bal )
   {
      setAccount( acct );
      setFirstName( first );
      setLastName( last );
      setBalance( bal );
   } // end four-argument AccountRecordSerializable constructor

   // set account number
   public void setAccount( int acct )
   {
      account = acct;
   } // end method setAccount

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

```

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

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

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

   // set last name
   public void setLastName( String last )
   {
      lastName = last;
   } // end method setLastName

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

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

```

   // set balance
   public void setBalance( double bal )
   {
      balance = bal;
   } // end method setBalance

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

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

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

- In a class that implements `Serializable`, every variable must be `Serializable`.

- Any one that is not must be declared **`transient`** so it will be ignored during the serialization process.

- All primitive-type variables are serializable.

- For reference-type variables, check the class's documentation (and possibly its superclasses) to ensure that the type is `Serializable`.

```java
// Fig. 17.17: CreateSequentialFile.java
// Writing objects sequentially to a file with class ObjectOutputStream.
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.util.NoSuchElementException;
import java.util.Scanner;

import com.deitel.ch17.AccountRecordSerializable;

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

   // allow user to specify file name
   public void openFile()
   {
      try // open file
      {
         output = new ObjectOutputStream(
            new FileOutputStream( "clients.ser" ) );
      } // end try
```

Associates an ObjectOutputStream with a file on disk.

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

```java
      catch ( IOException ioException )
      {
         System.err.println( "Error opening file." );
      } // end catch
   } // end method openFile

   // add records to file
   public void addRecords()
   {
      AccountRecordSerializable record; // object to be written to file
      int accountNumber = 0; // account number for record object
      String firstName; // first name for record object
      String lastName; // last name for record object
      double balance; // balance for record object

      Scanner input = new Scanner( System.in );

      System.out.printf( "%s\n%s\n%s\n%s\n\n",
         "To terminate input, type the end-of-file indicator ",
         "when you are prompted to enter input.",
         "On UNIX/Linux/Mac OS X type <ctrl> d then press Enter",
         "On Windows type <ctrl> z then press Enter" );

```

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

```
      System.out.printf( "%s\n%s",
         "Enter account number (> 0), first name, last name and balance.",
         "? " );

      while ( input.hasNext() ) // loop until end-of-file indicator
      {
         try // output values to file
         {
            accountNumber = input.nextInt(); // read account number
            firstName = input.next(); // read first name
            lastName = input.next(); // read last name
            balance = input.nextDouble(); // read balance

            if ( accountNumber > 0 )
            {
               // create new record
               record = new AccountRecordSerializable( accountNumber,
                  firstName, lastName, balance );
               output.writeObject( record ); // output record
            } // end if
```

Outputs an object to the file on disk.

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

```
            else
            {
               System.out.println(
                  "Account number must be greater than 0." );
            } // end else
         } // end try
         catch ( IOException ioException )
         {
            System.err.println( "Error writing to file." );
            return;
         } // end catch
         catch ( NoSuchElementException elementException )
         {
            System.err.println( "Invalid input. Please try again." );
            input.nextLine(); // discard input so user can try again
         } // end catch

         System.out.printf( "%s %s\n%s", "Enter account number (>0),",
            "first name, last name and balance.", "? " );
      } // end while
   } // end method addRecords

```

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

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

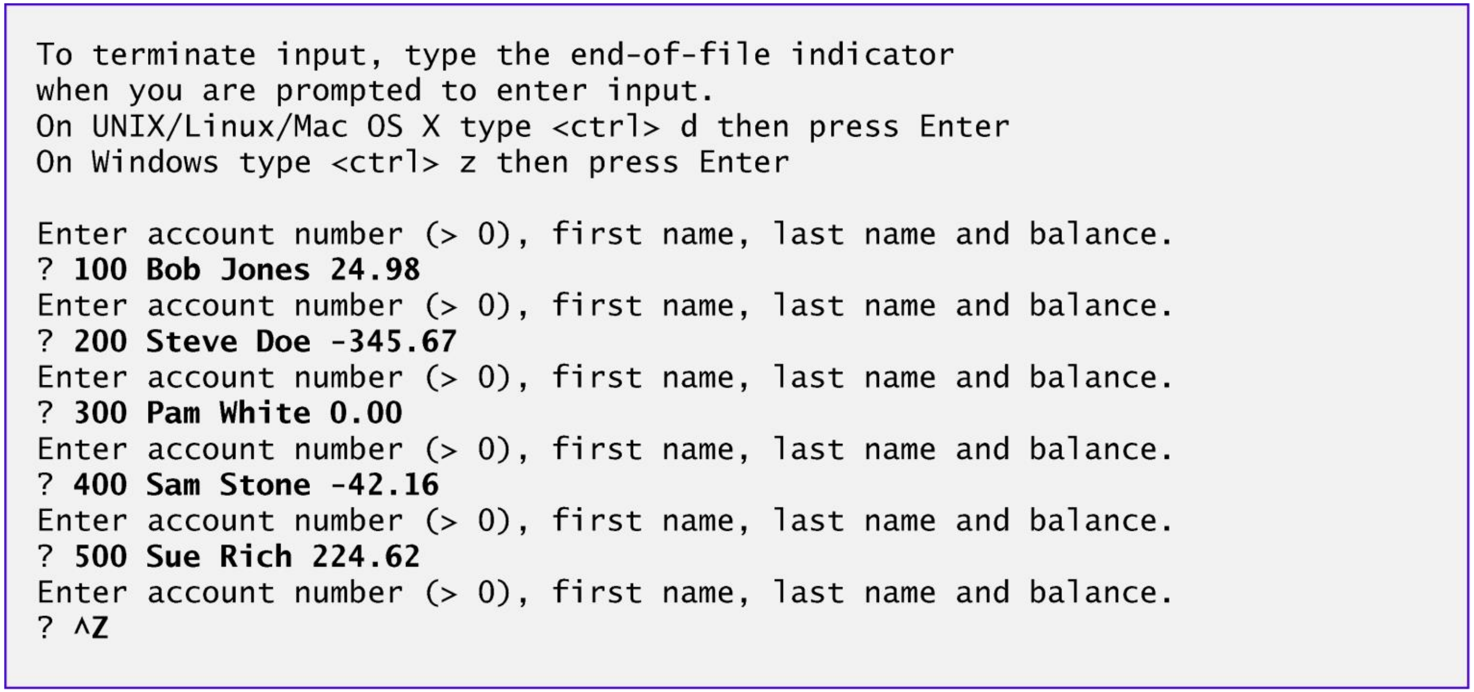
Fig. 17.17 | Sequential file created using `ObjectOutputStream`. (Part 5 of 5.)

```java
// Fig. 17.18: CreateSequentialFileTest.java
// Testing class CreateSequentialFile.

public class CreateSequentialFileTest
{
   public static void main( String[] args )
   {
      CreateSequentialFile application = new CreateSequentialFile();

      application.openFile();
      application.addRecords();
      application.closeFile();
   } // end main
} // end class CreateSequentialFileTest
```

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



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

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

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

Common Programming Error 17.2

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

17.6.2 Reading and Deserializing Data from a Sequential-Access File

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

```java
// Fig. 17.19: ReadSequentialFile.java
// Reading a file of objects sequentially with ObjectInputStream
// and displaying each record.
import java.io.EOFException;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

import com.deitel.ch17.AccountRecordSerializable;

public class ReadSequentialFile
{
   private ObjectInputStream input;

   // enable user to select file to open
   public void openFile()
   {
      try // open file
      {
         input = new ObjectInputStream(
            new FileInputStream( "clients.ser" ) );
      } // end try
```

Associates an `ObjectInputStream` with a file on disk.

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

```
      catch ( IOException ioException )
      {
         System.err.println( "Error opening file." );
      } // end catch
   } // end method openFile

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

      try // input the values from the file
      {
         while ( true )
         {
            record = ( AccountRecordSerializable ) input.readObject();

```

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

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

```java
            // display record contents
            System.out.printf( "%-10d%-12s%-12s%10.2f\n",
               record.getAccount(), record.getFirstName(),
               record.getLastName(), record.getBalance() );
         } // end while
      } // end try
      catch ( EOFException endOfFileException )
      {
         return; // end of file was reached
      } // end catch
      catch ( ClassNotFoundException classNotFoundException )
      {
         System.err.println( "Unable to create object." );
      } // end catch
      catch ( IOException ioException )
      {
         System.err.println( "Error during read from file." );
      } // end catch
   } // end method readRecords

```

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

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

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

```java
// Fig. 17.20: ReadSequentialFileTest.java
// Testing class ReadSequentialFile.

public class ReadSequentialFileTest
{
   public static void main( String[] args )
   {
      ReadSequentialFile application = new ReadSequentialFile();

      application.openFile();
      application.readRecords();
      application.closeFile();
   } // end main
} // end class ReadSequentialFileTest
```

```
Account   First Name  Last Name      Balance
100       Bob         Jones            24.98
200       Steve       Doe            -345.67
300       Pam         White             0.00
400       Sam         Stone           -42.16
500       Sue         Rich            224.62
```

Fig. 17.20 | Testing class `ReadSequentialFile`.

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

- `ObjectInputStream` method `readObject` reads an `Object` from a file.

- Method `readObject` throws an **`EOFException`** if an attempt is made to read beyond the end of the file.

- Method `readObject` throws a `ClassNotFoundException` if the class for the object being read cannot be located.

17.8 Opening Files with JFileChooser

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

```java
// Fig. 17.21: FileDemonstration.java
// Demonstrating JFileChooser.
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class FileDemonstration extends JFrame
{
   private JTextArea outputArea; // used for output
   private JScrollPane scrollPane; // used to provide scrolling to output

   // set up GUI
   public FileDemonstration()
   {
      super( "Testing class File" );

      outputArea = new JTextArea();
```

Fig. 17.21 | Demonstrating `JFileChooser`. (Part 1 of 5.)

```java

      // add outputArea to scrollPane
      scrollPane = new JScrollPane( outputArea );

      add( scrollPane, BorderLayout.CENTER ); // add scrollPane to GUI

      setSize( 400, 400 ); // set GUI size
      setVisible( true ); // display GUI

      analyzePath(); // create and analyze File object
   } // end FileDemonstration constructor

   // allow user to specify file or directory name
   private File getFileOrDirectory()
   {
      // display file dialog, so user can choose file or directory to open
      JFileChooser fileChooser = new JFileChooser();
      fileChooser.setFileSelectionMode(
         JFileChooser.FILES_AND_DIRECTORIES );

      int result = fileChooser.showOpenDialog( this );

```

Creates a `JFileChooser` for selecting files and directories.

Displays the `JFileChooser` centered over the parent window.

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

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

      File fileName = fileChooser.getSelectedFile(); // get File

      // display error if invalid
      if ( ( fileName == null ) || ( fileName.getName().equals( "" ) ) )
      {
         JOptionPane.showMessageDialog( this, "Invalid Name",
            "Invalid Name", JOptionPane.ERROR_MESSAGE );
         System.exit( 1 );
      } // end if

      return fileName;
   } // end method getFile

   // display information about file or directory user specifies
   public void analyzePath()
   {
      // create File object based on user input
      File name = getFileOrDirectory();

```

Retrieves the selected file or directory name.

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

```java
         if ( name.exists() ) // if name exists, output information about it
         {
            // display file (or directory) information
            outputArea.setText( String.format(
               "%s%s\n%s\n%s\n%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s",
               name.getName(), " exists",
               ( name.isFile() ? "is a file" : "is not a file" ),
               ( name.isDirectory() ? "is a directory" :
                  "is not a directory" ),
               ( name.isAbsolute() ? "is absolute path" :
                  "is not absolute path" ), "Last modified: ",
               name.lastModified(), "Length: ", name.length(),
               "Path: ", name.getPath(), "Absolute path: ",
               name.getAbsolutePath(), "Parent: ", name.getParent() ) );

            if ( name.isDirectory() ) // output directory listing
            {
               String[] directory = name.list();
               outputArea.append( "\n\nDirectory contents:\n" );

               for ( String directoryName : directory )
                  outputArea.append( directoryName + "\n" );
            } // end else
         } // end outer if
```

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

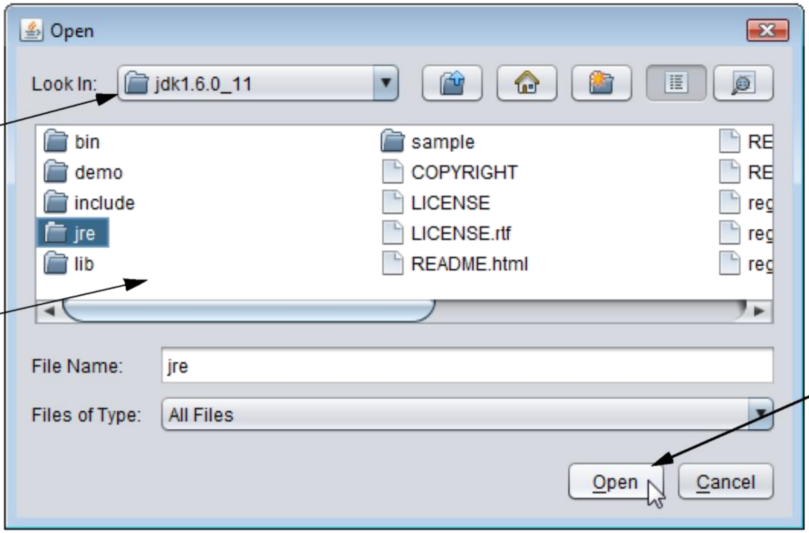
```
      else // not file or directory, output error message
      {
         JOptionPane.showMessageDialog( this, name +
            " does not exist.", "ERROR", JOptionPane.ERROR_MESSAGE );
      } // end else
   } // end method analyzePath
} // end class FileDemonstration
```

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

```java
// Fig. 17.22: FileDemonstrationTest.java
// Testing class FileDemonstration.
import javax.swing.JFrame;

public class FileDemonstrationTest
{
   public static void main( String[] args )
   {
      FileDemonstration application = new FileDemonstration();
      application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
   } // end main
} // end class FileDemonstrationTest
```

Fig. 17.22 | Testing class `FileDemonstration`. (Part 1 of 3.)



File Name: jre

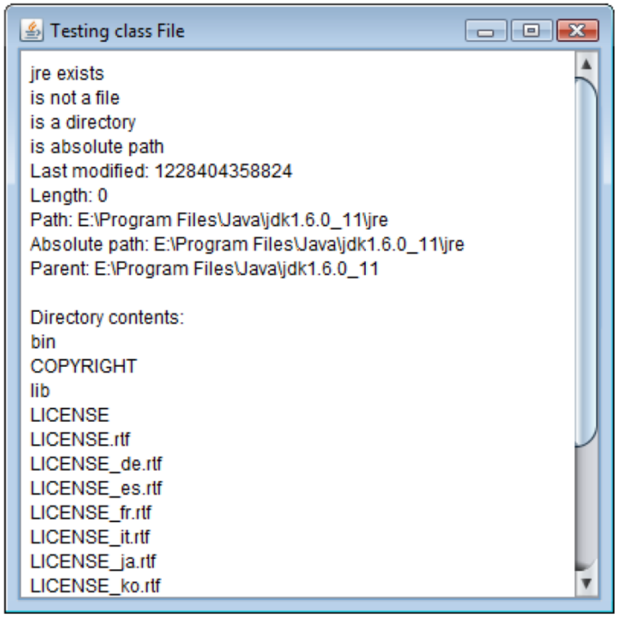
Files of Type: All Files

Select location of file or directory here

Files and directories are displayed here

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

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



```
jre exists
is not a file
is a directory
is absolute path
Last modified: 1228404358824
Length: 0
Path: E:\Program Files\Java\jdk1.6.0_11\jre
Absolute path: E:\Program Files\Java\jdk1.6.0_11\jre
Parent: E:\Program Files\Java\jdk1.6.0_11

Directory contents:
bin
COPYRIGHT
lib
LICENSE
LICENSE.rtf
LICENSE_de.rtf
LICENSE_es.rtf
LICENSE_fr.rtf
LICENSE_it.rtf
LICENSE_ja.rtf
LICENSE_ko.rtf
```

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

17.9 Opening Files with JFileChooser (cont.)

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

End of Part I

- Reading
  - Chapter 17