Java Programming

File Processing

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.

Files and Streams

- Java views each file as a sequential **stream of bytes**
- 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.
- 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.
- 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.

Files and Streams

- 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`.

Files and Streams

- 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.

Class `File`

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.

Fig. 17.2 | `File` methods. (Part 1 of 2.)

Class `File`

Method	Description
`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.
`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 `String`s representing a directory's contents. Returns `null` if the `File` object does not represent a directory.

Fig. 17.2 | `File` methods. (Part 2 of 2.)

Class `File`

```java
// Fig. 17.3: 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

```

Fig. 17.3 | `File` class used to obtain file and directory information. (Part 1 of 5.)

Class File

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

      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() );

```

Fig. 17.3 | File class used to obtain file and directory information. (Part 2 of 5.)

Class File

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

Class `File`

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

Class `File`

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

Opening Files with JFileChooser

- Class **JFileChooser** displays a dialog that enables the user to easily select files or directories.
- JFile-Chooser method **setFile-SelectionMode** 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.

Opening Files with `JFileChooser`

```java
// Fig. 17.20: 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

```

Fig. 17.20 | Demonstrating `JFileChooser`. (Part 1 of 5.)

Opening Files with JFileChooser

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

      outputArea = new JTextArea();

      // 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

```

Fig. 17.20 | Demonstrating `JFileChooser`. (Part 2 of 5.)

Opening Files with `JFileChooser`

```java
   // 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 );

      // 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

```

Fig. 17.20 | Demonstrating `JFileChooser`. (Part 3 of 5.)

Opening Files with `JFileChooser`

```java
      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();

      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() ) );

```

Fig. 17.20 | Demonstrating `JFileChooser`. (Part 4 of 5.)

Opening Files with JFileChooser

```java
            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
         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. 17.20 | Demonstrating `JFileChooser`. (Part 5 of 5.)

Opening Files with `JFileChooser`

```java
// Fig. 17.21: 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. 17.21 | Testing class `FileDemonstration`. (Part 1 of 3.)

Opening Files with `JFileChooser`

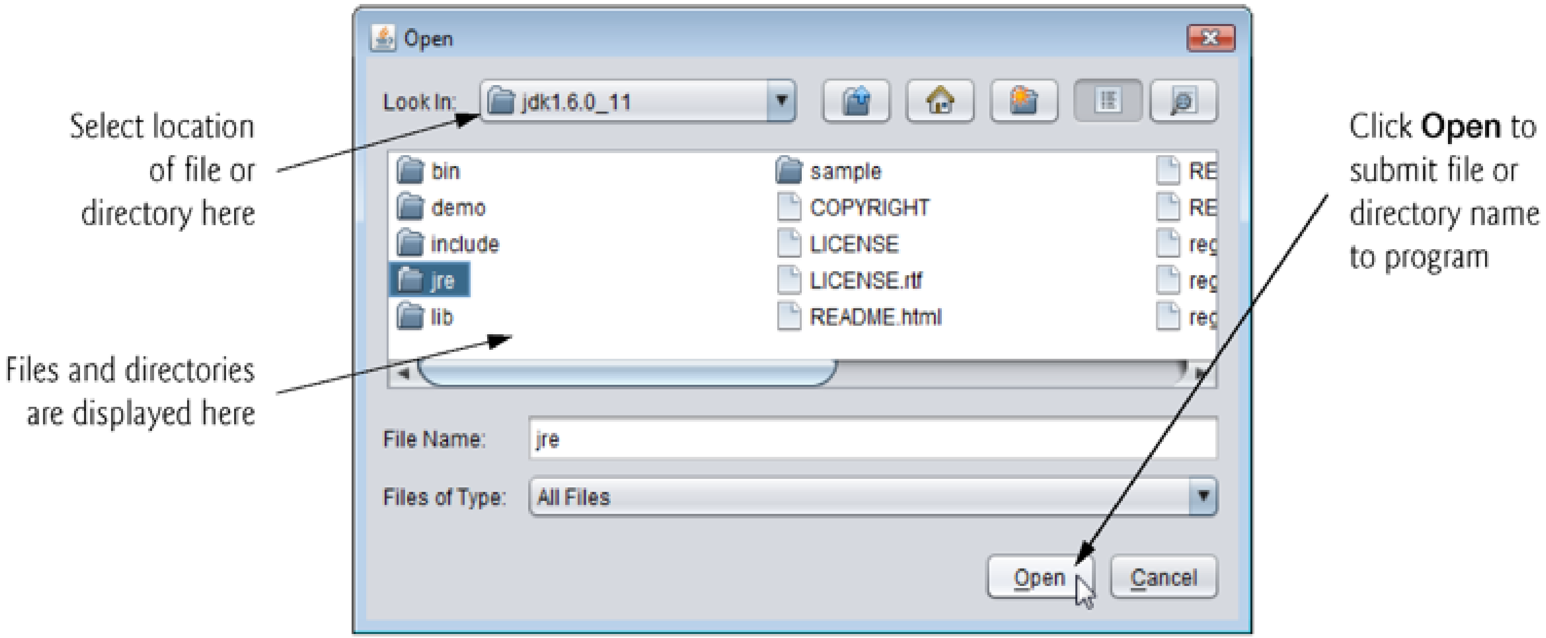
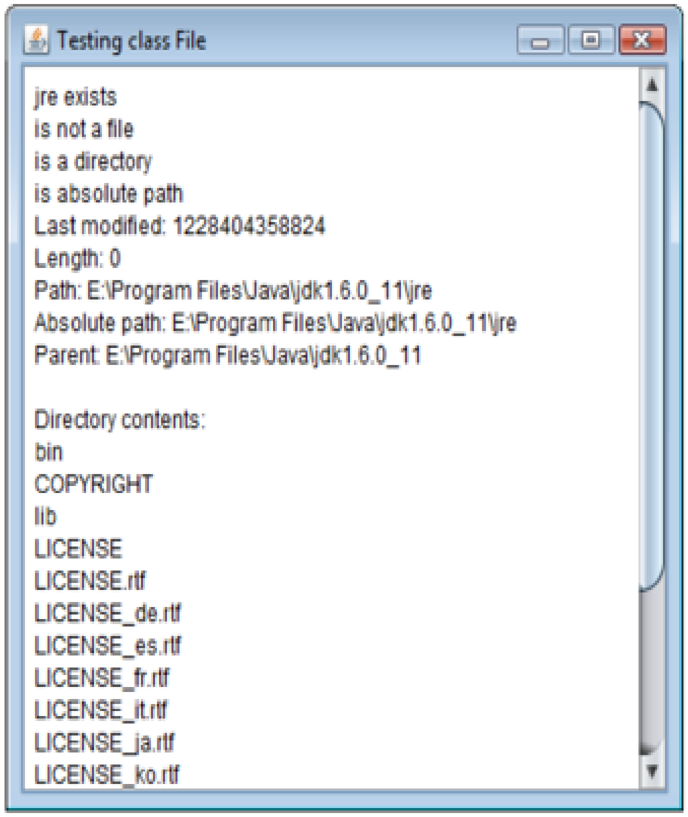

Select location of file or directory here
Files and directories are displayed here
Click **Open** to submit file or directory name to program
Open
Look In: jdk1.6.0_11
bin
demo
include
jre
lib
sample
COPYRIGHT
LICENSE
LICENSE.rtf
README.html
File Name: jre
Files of Type: All Files
Open
Cancel

Fig. 17.21 | Testing class `FileDemonstration`. (Part 2 of 3.)

Opening Files with `JFileChooser`



```
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. 17.21 | Testing class `FileDemonstration`. (Part 3 of 3.)

Sequential-Access Text Files

- Sequential-access files store records in order by the record-key field.

- Text files are human-readable files.

- 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.

Sequential-Access Text Files

```java
// Fig. 17.4: 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. 17.4 | AccountRecord class maintains information for one account. (Part 1 of 4.)

Sequential-Access Text Files

```
   // 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. 17.4 | AccountRecord class maintains information for one account. (Part 2 of 4.)

Sequential-Access Text Files

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

Sequential-Access Text Files

```

    // 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. 17.4 | AccountRecord class maintains information for one account. (Part 4 of 4.)

Sequential-Access Text Files

```java
// Fig. 17.5: 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

```

Fig. 17.5 | Writing data to a sequential text file with class `Formatter`. (Part 1 of 5.)

Sequential-Access Text Files

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

```

Fig. 17.5 | Writing data to a sequential text file with class `Formatter`. (Part 2 of 5.)

Sequential-Access Text Files

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

       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" );

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

```

Fig. 17.5 | Writing data to a sequential text file with class `Formatter`. (Part 3 of 5.)

Sequential-Access Text Files

```java
         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 )
               {
                  // 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
```

Fig. 17.5 | Writing data to a sequential text file with class `Formatter`. (Part 4 of 5.)

Sequential-Access Text Files

```
            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

            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
```

Fig. 17.5 | Writing data to a sequential text file with class `Formatter`. (Part 5 of 5.)

Sequential-Access Text Files

```java
// Fig. 17.7: 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. 17.7 | Testing the `CreateTextFile` class. (Part 1 of 2.)

Sequential-Access Text Files

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

Reading Data Sequential-Access Text Files

```java
// Fig. 17.9: 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;

```

Fig. 17.9 | Sequential file reading using a `Scanner`. (Part 1 of 4.)

Reading Data Sequential-Access Text Files

```
   // enable user to open file
   public void openFile()
   {
      try
      {
         input = new Scanner( new File( "clients.txt" ) );
      } // end try
      catch ( FileNotFoundException fileNotFoundException )
      {
         System.err.println( "Error opening file." );
         System.exit( 1 );
      } // end catch
   } // end method openFile

```

Fig. 17.9 | Sequential file reading using a Scanner. (Part 2 of 4.)

Reading Data Sequential-Access Text Files

```java
   // 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

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

Fig. 17.9 | Sequential file reading using a Scanner. (Part 3 of 4.)

Reading Data Sequential-Access Text Files

```
        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

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

Fig. 17.9 | Sequential file reading using a Scanner. (Part 4 of 4.)

Reading Data Sequential-Access Text Files

```java
// Fig. 17.10: 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. 17.10 | Testing the `ReadTextFile` class.

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.

Case Study: A Credit-Inquiry Program

```
// Fig. 17.11: 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 );

```

Fig. 17.11 | Enumeration for the credit-inquiry program's menu options. (Part 1 of 2.)

Case Study: A Credit-Inquiry Program

```java
   private final int value; // current menu option

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

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

Fig. 17.11 | Enumeration for the credit-inquiry program's menu options. (Part 2 of 2.)

Case Study: A Credit-Inquiry Program

```java
// Fig. 17.12: 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. 17.12 | Credit-inquiry program. (Part 1 of 6.)

Case Study: A Credit-Inquiry Program

```
      // 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
```

Fig. 17.12 | Credit-inquiry program. (Part 2 of 6.)

Case Study: A Credit-Inquiry Program

```
      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. 17.12 | Credit-inquiry program. (Part 3 of 6.)

Case Study: A Credit-Inquiry Program

```
   // 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. 17.12 | Credit-inquiry program. (Part 4 of 6.)

Case Study: A Credit-Inquiry Program

```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. 17.12 | Credit-inquiry program. (Part 5 of 6.)

Case Study: A Credit-Inquiry Program

```
    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. 17.12 | Credit-inquiry program. (Part 6 of 6.)

Case Study: A Credit-Inquiry Program

```java
// Fig. 17.13: 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. 17.13 | Testing the `CreditInquiry` class.

Case Study: A Credit-Inquiry Program

```
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. 17.14 | Sample output of the credit-inquiry program in Fig. 17.13. (Part 1 of 2.)

Case Study: A Credit-Inquiry Program

```
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. 17.14 | Sample output of the credit-inquiry program in Fig. 17.13. (Part 2 of 2.)