Lecture 11c

Files and Streams

OBJECTIVES

In this lecture you will learn:

- 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 and random-access file processing.
- To use classes `Scanner` and `Formatter` to process text files.
- To use the `FileInputStream` and `FileOutputStream` classes.
- To use a `JFileChooser` dialog.
- To use the `ObjectInputStream` and `ObjectOutputStream` classes

Outline

Outline

14.1 Introduction

- **Storage of data in variables and arrays is temporary**
- **Files used for long-term retention of large amounts of data, even after the programs that created the data terminate**
- **Persistent data – exists beyond the duration of program execution**
- **Files stored on secondary storage devices**
- **Stream – ordered data that is read from or written to a file**

14.2 Data Hierarchy

- Computers process all data items as combinations of zeros and ones
- Bit – smallest data item on a computer, can have values 0 or 1
- Byte – 8 bits
- Characters – larger data item
  - Consists of decimal digits, letters and special symbols
  - Character set – set of all characters used to write programs and represent data items
    - Unicode – characters composed of two bytes
    - ASCII

14.2 Data Hierarchy

- **Fields – a group of characters or bytes that conveys meaning**
- **Record – a group of related fields**
- **File – a group of related records**
- **Data items processed by computers form a data hierarchy that becomes larger and more complex from bits to files**
- **Record key – identifies a record as belonging to a particular person or entity – used for easy retrieval of specific records**
- **Sequential file – file in which records are stored in order by the record-key field**
- **Database – a group of related files**
- **Database Management System – a collection of programs designed to create and manage databases**

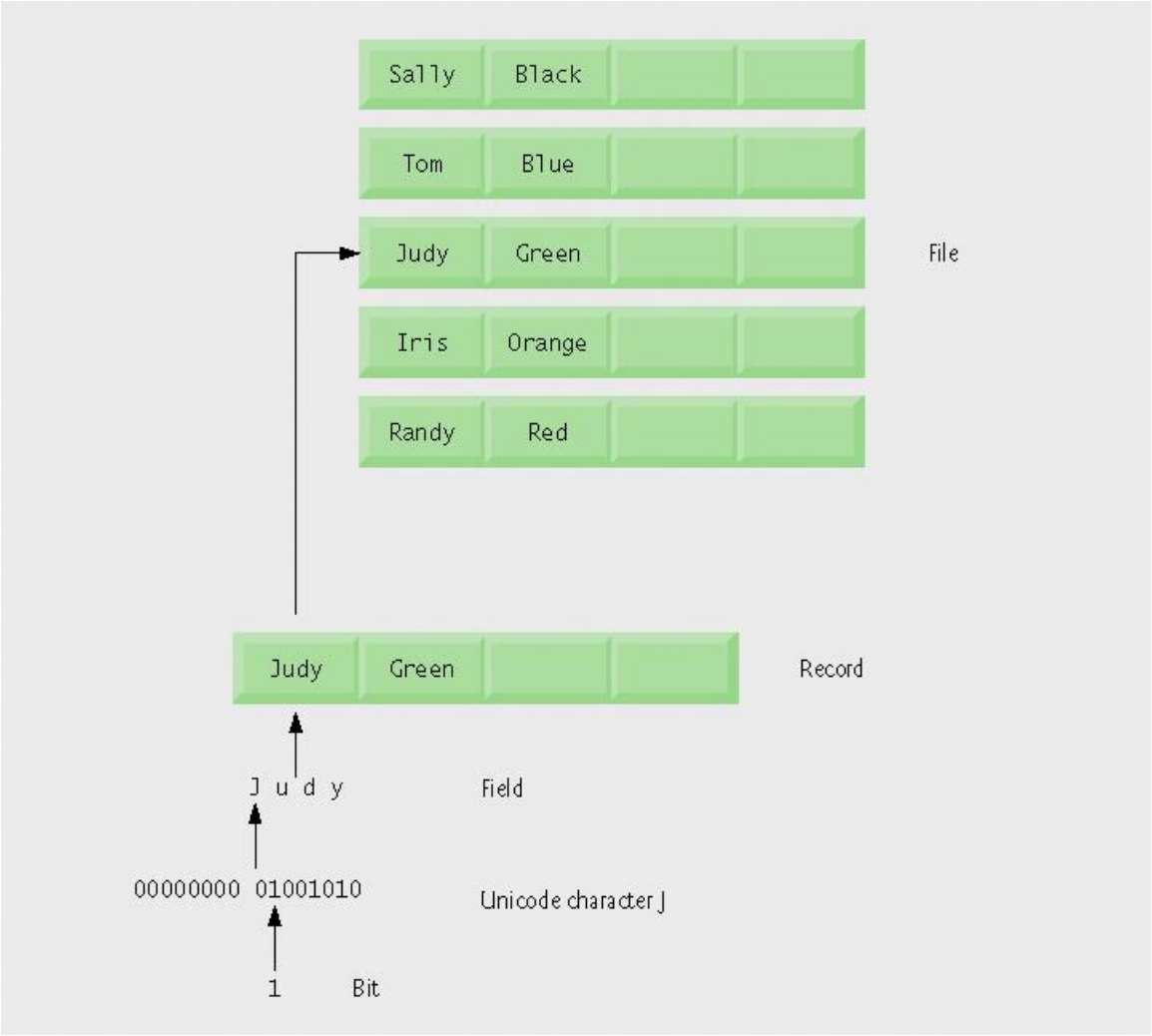


Fig. 14.1 | Data hierarchy.

14.3 Files and Streams

- **Java views each files as a sequential stream of bytes**
- **Operating system provides mechanism to determine end of file**
  - **End-of-file marker**
  - **Count of total bytes in file**
  - **Java program processing a stream of bytes receives an indication from the operating system when program reaches end of stream**

14.3 Files and Streams

- **File streams**
  - **Byte-based streams – stores data in binary format**
    - **Binary files – created from byte-based streams, read by a program that converts data to human-readable format**
  - **Character-based streams – stores data as a sequence of characters**
    - **Text files – created from character-based streams, can be read by text editors**
- **Java opens file by creating an object and associating a stream with it**
- **Standard streams – each stream can be redirected**
  - `System.in` **– standard input stream object, can be redirected with method** `setIn`
  - `System.out` **– standard output stream object, can be redirected with method** `setOut`
  - `System.err` **– standard error stream object, can be redirected with method** `setErr`

14.3 Files and Streams

- `java.io` classes
  - `FileInputStream` and `FileOutputStream` – byte-based I/O
  - `FileReader` and `FileWriter` – character-based I/O
  - `ObjectInputStream` and `ObjectOutputStream` – used for input and output of objects or variables of primitive data types
  - `File` – useful for obtaining information about files and directories
- Classes `Scanner` and `Formatter`
  - `Scanner` – can be used to easily read data from a file
  - `Formatter` – can be used to easily write data to a file

Fig. 14.2 | Java's view of a file of *n* bytes.

14.4 Class `File`

- Class `File` useful for retrieving information about files and directories from disk
- Objects of class `File` do not open files or provide any file-processing capabilities

Creating `File` Objects

- Class `File` provides four constructors:
  1. Takes `String` specifying name and path (location of file on disk)
  2. Takes two `String`s, first specifying path and second specifying name of file
  3. Takes `File` object specifying path and `String` specifying name of file
  4. Takes `URI` object specifying name and location of file
- Different kinds of paths
  - Absolute path – contains all directories, starting with the root directory, that lead to a specific file or directory
  - Relative path – normally starts from the directory in which the application began executing

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 name specified as the argument to the `File` constructor is a file or directory in the specified path; `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. 14.3 | File methods. (Part 1 of 2)

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 can be found).
`long length()`	Returns the length of the file, in bytes. If the `File` object represents a directory, `0` 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 the contents of a directory. Returns `null` if the `File` object does not represent a directory.

Fig.14.3 | File methods. (Part 2 of 2)

Error-Prevention Tip 14.1

Use `File` method `isFile` to determine whether a `File` object represents a file (not a directory) before attempting to open the file.

Demonstrating Class `File`

- Common `File` methods
  - `exists` – return `true` if file exists where it is specified
  - `isFile` – returns `true` if `File` is a file, not a directory
  - `isDirectory` – returns `true` if `File` is a directory
  - `getPath` – return file path as a string
  - `list` – retrieve contents of a directory
- Separator character – used to separate directories and files in a path
  - Windows uses \
  - UNIX uses /
  - Java process both characters, `File.pathSeparator` can be used to obtain the local computer's proper separator character

```java
// Fig. 14.4: FileDemonstration.java
// Demonstrating the File class.
import java.io.File;

public class FileDemonstration
{
   // display information about file user specifies
   public void analyzePath( String path
   {
      // create File object based
      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
            name.getName(), " exists",
            ( name.isFile() ? "is a file
            ( name.isDirectory() ? "is a d
               "is not a directory" ),
            ( name.isAbsolute() ? "is absc
               "is not absolute path" ),
            name.lastModified(), "Length:
            "Path: ", name.getPath(), "Absolute path: ",
            name.getAbsolutePath(), "Parent: ", name.getParent() );

```

Create new `File` object; user

Returns `true` if file or directory specified exists

Retri

Returns `true` if `name` is a directory, not a file

Returns `true` if `path` was an absolute path

Retrieve time file or dir

Retrieve length of file in bytes

Retrieve path entered as a string

Retrieve absolute path of file or direc

Retrieve parent directory (path where `File` object's file or directory can be found)

```java
         if ( name.isDirectory() ) // output directory listing
         {
            String directory[] = name.list();
            System.out.println( "\n\nDirector
            for ( String directoryName : directory )
               System.out.printf( "%s\n", directoryName );
         } // end else
      } // 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
```

Returns true if `File` is a directory, not a file

Retrieve and display contents of directory

```java
// Fig. 14.5: FileDemonstrationTest.java
// Testing the FileDemonstration class.
import java.util.Scanner;

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

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

```
Enter file or directory name here: C:\Program Files\Java\jdk1.5.0\demo\jfc
jfc exists
is not a file
is a directory
is absolute path
Last modified: 1083938776645
Length: 0
Path: C:\Program Files\Java\jdk1.5.0\demo\jfc
Absolute path: C:\Program Files\Java\jdk1.5.0\demo\jfc
Parent: C:\Program Files\Java\jdk1.5.0\demo

Directory contents:

CodePointIM
FileChooserDemo
Font2DTest
Java2D
Metalworks
Notepad
SampleTree
Stylepad
SwingApplet
SwingSet2
TableExample
```

```
Enter file or directory name here:
C:\Program Files\Java\jdk1.5.0\demo\jfc\Java2D\readme.txt
readme.txt exists
is a file
is not a directory
is absolute path
Last modified: 1083938778347
Length: 7501
Path: C:\Program Files\Java\jdk1.5.0\demo\jfc\Java2D\readme.txt
Absolute path: C:\Program Files\Java\jdk1.5.0\demo\jfc\Java2D\readme.txt
Parent: C:\Program Files\Java\jdk1.5.0\demo\jfc\Java2D
```

Common Programming Error 14.1

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.

14.5 Sequential-Access Text Files

- **Records are stored in order by record-key field**
- **Can be created as text files or binary files**

14.5.1 Creating a Sequential-Access Text File

- Java imposes no structure on a file, records do not exist as part of the Java language
- Programmer must structure files
- `Formatter` class can be used to open a text file for writing
  - Pass name of file to constructor
  - If file does not exist, will be created
  - If file already exists, contents are truncated (discarded)
  - Use method `format` to write formatted text to file
  - Use method `close` to close the `Formatter` object (if method not called, OS normally closes file when program exits)

14.5.1 Creating a Sequential-Access Text File

- **Possible exceptions**
  - `SecurityException` – occurs when opening file using `Formatter` object, if user does not have permission to write data to file
  - `FileNotFoundException` – occurs when opening file using `Formatter` object, if file cannot be found and new file cannot be created
  - `NoSuchElementException` – occurs when invalid input is read in by a `Scanner` object
  - `FormatterClosedException` – occurs when an attempt is made to write to a file using an already closed `Formatter` object

```java
// Fig. 14.6: AccountRecord.java
// A class that represents one record of information.
package com.deitel.jhtp7.ch14; // 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

   // 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

   // 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

   // 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
```

```java
// Fig. 14.7: CreateTextFile.java
// Writing data to a 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.jhtp7.ch14.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" );
      } // end try
      catch ( SecurityException securityException )
      {
         System.err.println(
            "You do not have write access to this file." );
         System.exit( 1 );
      } // end catch
```

Used for writing data to file

Used for retrieving input from user

Object used to output data to file

Open file `clients.txt` for writing

```
29      catch ( FileNotFoundException filesNotFoundException )
30      {
31         System.err.println( "Error creating file." );
32         System.exit( 1 );
33      } // end catch
34   } // end method openFile
35
36   // add records to file
37   public void addRecords()
38   {
39      // object to be written to file
40      AccountRecord record = new AccountRecord();
41
42      Scanner input = new Scanner( System.in );
43
44      System.out.printf( "%s\n%s\n%s\n%s
45         "To terminate input, type the e
46         "when you are prompted to enter
47         "On UNIX/Linux/Mac OS X type <ctrl> d then press Enter",
48         "On Windows type <ctrl> z then press Enter" );
49
50      System.out.printf( "%s\n%s",
51         "Enter account number (> 0), first name, last name and balance.",
52         "? " );
53
```

Create `AccountRecord` to be filled with user input

Create `Scanner` to retrieve input from user

```
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
   catch ( FormatterClosedException formatterClosedException )
   {
      System.err.println( "Error writing to file." );
      return;
   } // end catch
```

Loop while user is entering input

Retrieve input, store data in `AccountRecord`

Write `AccountRecord` information to file

File closed while trying to write to it

```
82          catch ( NoSuchElementException elementException )
83          {
84             System.err.println( "Invalid input. Please try again." );
85             input.nextLine(); // discard input
86          } // end catch
87
88          System.out.printf( "%s %s\n%s", "Enter account number (>0),",
89             "first name, last name and balance.", "? " );
90       } // end while
91    } // end method addRecords
92
93    // close file
94    public void closeFile()
95    {
96       if ( output != null )
97          output.close();
98    } // end method closeFile
99 } // end class CreateTextFile
```

Error with input entered by user

Close file

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

Fig.14.8 | End-of-file key combinations for various popular operating systems.

```java
// Fig. 14.9: 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
```

```
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
```

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.14.10 | **Sample data for the program in Fig. 14.7.**

14.5.2 Reading Data from a Sequential-Access Text File

- Data is stored in files so that it may be retrieved for processing when needed
- `Scanner` object can be used to read data sequentially from a text file
  - Pass `File` object representing file to be read to `Scanner` constructor
  - `FileNotFoundException` occurs if file cannot be found
  - Data read from file using same methods as for keyboard input – `nextInt`, `nextDouble`, `next`, etc.
  - `IllegalStateException` occurs if attempt is made to read from closed `Scanner` object

```java
// Fig. 14.11: 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.jhtp7.ch14.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
      catch ( FileNotFoundException fileNotFoundException )
      {
         System.err.println( "Error opening file." );
         System.exit( 1 );
      } // end catch
   } // end method openFile

```

Open file `clients.txt` for reading

```
29    // read record from file
30    public void readRecords()
31    {
32       // object to be written to screen
33       AccountRecord record = new AccountRecord();
34
35       System.out.printf( "%-10s%-12s%-12s%
36          "First Name", "Last Name", "Balar
37
38       try // read records from file using Scanner object
39       {
40          while ( input.hasNext() )
41          {
42             record.setAccount( input.nextInt() ); // read account number
43             record.setFirstName( input.next() ); // read first name
44             record.setLastName( input.next() ); // read last name
45             record.setBalance( input.nextDouble() ); // read balance
46
47             // display record contents
48             System.out.printf( "%-10d%-12s%-12s%10.2f\n",
49                record.getAccount(), record.getFirstName(),
50                record.getLastName(), record.getBalance() );
51          } // end while
52       } // end try
```

Create `AccountRecord` to store input from file

While there is data to be read from file

Read data from file, store in `AccountRecord`

Display `AccountRecord` contents

```
      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
```

Close file

```java
// Fig. 14.12: ReadTextFileTest.java
// This program test class ReadTextFile.

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
```

14.5.3 Case Study: A Credit-Inquiry Program

- **To retrieve data sequentially from a file, programs normally start reading from beginning of the file and read all the data consecutively until desired information is found**
- **Class `Scanner` provides no way to reposition to beginning of file**
- **Instead, file is closed and reopened**

```java
// Fig. 14.13: MenuOption.java
// Defines an 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

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

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

```java
// Fig. 14.14: 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.jhtp7.ch14.AccountRecord;

public class CreditInquiry
{
   private MenuOption accountType;
   private Scanner input;
   private 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()
   {
      // object to be written to file
      AccountRecord record = new AccountRecord();

```

`Scanner` used to read data from file

`AccountRecord` stores record being read from file

```
27      try // read records
28      {
29         // open file to read from beginning
30         input = new Scanner( new File
31
32         while ( input.hasNext() ) // input the values from the file
33         {
34            record.setAccount( input.nextInt() ); // read account number
35            record.setFirstName( input.next() ); // read first name
36            record.setLastName( input.next() ); // read last name
37            record.setBalance( input.nextDouble() ); // read balance
38
39            // if proper acount type, display
40            if ( shouldDisplay( record.getBalance() ) )
41               System.out.printf( "%-10d%-12s%-12s%10.2f\n",
42                  record.getAccount(), record.getFirstName(),
43                  record.getLastName(), record.getBalance() );
44         } // end while
45      } // end try
46      catch ( NoSuchElementException elementExc
47      {
48         System.err.println( "File improperly formed." );
49         input.close();
50         System.exit( 1 );
51      } // end catch
```

Open file `clients.txt` for reading

While there is data to read from file

Check if record is of requested type

Retrieve input, store data in `AccountRecord`

Display record data to screen

Close `Scanner`

```
      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

   // use record type to determine if record should be display
   private boolean shouldDisplay( double balance )
   {
      if ( ( accountType == MenuOption.CREDIT_BALANCE )
         && ( balance < 0 ) )
         return true;

      else if ( ( accountType == MenuOption.DEBIT_BALANCE )
         && ( balance > 0 ) )
         return true;

```

Close file

Method determines if record is of proper type

```java
      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;

      // 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
```

Loop until user enters valid request

Retrieve request entered

```
   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

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
```

Read file, display proper records

```java
// Fig. 14.15: 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
```

```
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

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
```

14.5.4 Updating Sequential-Access Files

- **Data in many sequential files cannot be modified without risk of destroying other data in file**
- **Old data cannot be overwritten if new data is not same size**
- **Records in sequential-access files are not usually updated in place. Instead, entire file is usually rewritten.**

14.6 Object Serialization

- With text files, data type information lost
- Object serialization – mechanism to read or write an entire object from a file
- Serialized object – object represented as sequence of bytes, includes object's data and type information about object
- Deserialization – recreate object in memory from data in file
- Serialization and deserialization performed with classes `ObjectInputStream` and `ObjectOutputStream`, methods `readObject` and `writeObject`

14.6.1 Creating a Sequential-Access File Using Object Serialization: *Defining the `AccountRecordSerializable` Class*

- `Serializable` interface – programmers must declare a class to implement the `Serializable` interface, or objects of that class cannot be written to a file
- To open a file for writing objects, create a `FileOutputStream` wrapped by an `ObjectOutputStream`
  - `FileOutputStream` provides methods for writing `byte`-based output to a file
  - `ObjectOutputStream` uses `FileOutputStream` to write objects to file
  - `ObjectOutputStream` method `writeObject` writes object to output file
  - `ObjectOutputStream` method `close` closes both objects

```java
// Fig. 14.17: AccountRecordSerializable.java
// A class that represents one record of information.
package com.deitel.jhtp7.ch14; // 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

   // 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

```

Interface `Serializable` specifies that `AccountRecordSerializable` objects can be written to file

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

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

   // 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

   // 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
```

```java
// Fig. 14.18: 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.jhtp7.ch14.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
      catch ( IOException ioException )
      {
         System.err.println( "Error opening file." );
      } // end catch
   } // end method openFile

```

Class used to create byte-based output stream

Class used to create output object data to byte-based stream

Open file `clients.ser` for writing

```
   // 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" );

      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( accoun
               firstName, lastName, balance );
            output.writeObject( record ); // output rec
         } // end if
         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

```

Write record object to file

Create `AccountRecord` based on user input

```
   // 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
```

```java
// Fig. 14.19: 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
```

```
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
```

Common Programming Error 14.2

It is a logic error to open an existing file for output when, in fact, the user wishes to preserve the file.

14.6.2 Reading and Deserializing Data from a Sequential-Access File

- To open a file for reading objects, create a `FileInputStream` wrapped by an `ObjectInputStream`
  - `FileInputStream` provides methods for reading `byte`-based input from a file
  - `ObjectInputStream` uses `FileInputStream` to read objects from file
  - `ObjectInputStream` method `readObject` reads in object, which is then downcast to proper type
    - `EOFException` occurs if attempt made to read past end of file
    - `ClassNotFoundException` occurs if the class for the object being read cannot be located
  - `ObjectInputStream` method `close` closes both objects

```java
// Fig. 14.20: ReadSequentialFile.java
// This program reads a file of objects sequentially
// and displays each record.
import java.io.EOFException;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

import com.deitel.jhtp7.ch14.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
      catch ( IOException ioException )
      {
         System.err.println( "Error opening file." );
      } // end catch
   } // end method openFile

```

Class used to create byte-based input stream

Class used to read input object data to byte-based stream

Open file `clients.ser` for reading

```java
   // 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();

            // 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
```

Read record from file

Output record information to screen

```
      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

   // 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
```

Close file

```java
// Fig. 14.21: ReadSequentialFileTest.java
// This program test 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
```

14.7 Additional `java.io` Classes: *Interfaces and Classes for Byte-Based Input and Output*

- `InputStream` and `OutputStream` classes
  - `abstract` classes that declare methods for performing byte-based input and output
- `PipedInputStream` and `PipedOutputStream` classes
  - Establish pipes between two threads in a program
  - Pipes are synchronized communication channels between threads
- `FilterInputStream` and `FilterOutputStream` classes
  - Provides additional functionality to stream, such as aggregating data byte into meaningful primitive-type units
- `PrintStream` class
  - Performs text output to a specified stream
- `DataInput` and `DataOutput` interfaces
  - For reading and writing primitive types to a file
  - `DataInput` implemented by classes `RandomAccessFile` and `DataInputStream`, `DataOutput` implemented by `RandomAccessFile` and `DataOuputStream`
- `SequenceInputStream` class enables concatenation of several `InputStream`s – program sees group as one continuous `InputStream`

Interfaces and Classes for Byte-Based Input and Output

- **Buffering is an I/O-performance-enhancement technique**
  - **Greatly increases efficiency of an application**
  - **Output (uses `BufferedOutputStream` class)**
    - **Each output statement does not necessarily result in an actual physical transfer of data to the output device – data is directed to a region of memory called a buffer (faster than writing to file)**
    - **When buffer is full, actual transfer to output device is performed in one large physical output operation (also called logical output operations)**
    - **Partially filled buffer can be forced out with method `flush`**
  - **Input (uses `BufferedInputStream` class)**
    - **Many logical chunks of data from a file are read as one physical input operation (also called logical input operation)**
    - **When buffer is empty, next physical input operation is performed**
- **`ByteArrayInputStream` and `ByteArrayOutputStream` classes used for inputting from `byte` arrays in memory and outputting to `byte` arrays in memory**

Performance Tip 14.1

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

Interfaces and Classes for Character-Based Input and Output

- `Reader` and `Writer` abstract classes
  - Unicode two-byte, character-based streams
- `BufferedReader` and `BufferedWriter` classes
  - Enable buffering for character-based streams
- `CharArrayReader` and `CharArrayWriter` classes
  - Read and write streams of characters to character arrays
- `LineNumberReader` class
  - Buffered character stream that keeps track of number of lines read
- `PipedReader` and `PipedWriter` classes
  - Implement piped-character streams that can be used to transfer information between threads
- `StringReader` and `StringWriter` classes
  - Read characters from and write characters to `String`s

14.8 Try-Catch With Resources

The `try-with-resources` statement is a `try` statement that declares one or more resources. A `resource` is an object that must be closed after the program is finished with it. The `try-with-resources` statement ensures that each resource is closed at the end of the statement.

Any object that implements `java.lang.AutoCloseable`, which includes all objects which implement `java.io.Closeable`, can be used as a resource.

Exceptions: class `Formatter` is not `AutoCloseable`

14.8 Try-Catch With Resources

```
static String readFirstLineFromFile(File file) throws IOException {
    try (Scanner br = new Scanner(file)) {
        return br.nextLine();
    }
}
```

```
static String readFirstObjectFromFile(File file) throws IOException {
    try (ObjectInputStream br =
                new ObjectInputStream(new FileInputStream(file))) {
        return (String) br.readObject();
    } catch (ClassNotFoundException ex) {
        System.out.println("ClassNotFoundException:...");
    }
    return null;
}
```

```
static void writeObjectToFile(File file, Serializable obj) throws IOException {
    try (ObjectOutputStream br =
                new ObjectOutputStream(new FileOutputStream(file))) {
        br.writeObject(obj);
    }
}
```

14.8 Try-Catch With Resources

In these examples, the `resource` declared in the `try-with-resources` statement are `Scanner`, `ObjectInputStream`, `ObjectOutputStream`. The declaration statements appears within parentheses immediately after the `try` keyword. These classes implement the `interface java.lang.AutoCloseable`.

Because the instances of these classes are declared in a `try-with-resource` statement, they will be closed regardless of whether the `try` statement completes normally or abruptly (for example, as a result of the method `ObjectInputStream.readObject()` throwing an `IOException`).

14.8 Try-Catch With Resources

Prior to Java SE 7, you can use a `finally` block to ensure that a resource is closed regardless of whether the `try` statement completes normally or abruptly. The following example uses a finally block instead of a `try-with-resources` statement:

```
static void writeObjectWithFinally(File file, Serializable obj) throws IOException {
    ObjectOutputStream ous = null;
    try {
        ous = new ObjectOutputStream(new FileOutputStream(file));
        ous.writeObject(new Player("A", "Ateam"));
    } catch (IOException ex) {
        System.out.println("catch IOException...");
    } finally {
        try {
            if (ous != null) {
                ous.close();
            }
        } catch (IOException ex) {
            System.out.println("Finally IOException...");
        }
    }
}
```

14.8 Try-Catch With Resources

However, in the `writeObjectToFileWithFinally` example, if the methods `writeObject` and `close` both `throw` exceptions, then the method `someMethod` throws the exception thrown from the `finally` block and the exception thrown from the `try` block is suppressed.

In contrast, in the example `writeObjectToFile`, if exceptions are thrown from both the `try` block and the `try-with-resources` statement, then the method `writeObjectToFile` throws the exception thrown from the `try` block and the exception thrown from the `try-with-resources` block is suppressed

14.8 Try-Catch With Resources

Note:

A `try-with-resources` statement can have `catch` and `finally` blocks just like an ordinary try statement. In a `try-with-resources` statement, any `catch` or `finally` block is run after the resources declared have been closed.

14.9 Opening Files with `JFileChooser`

- `JFileChooser` – class used to display a dialog that enables users to easily select files
  - Method `setFileSelectionMode` specifies what user can select from `JFileChooser`
    - `FILES_AND_DIRECTORIES` constant indicates files and directories
    - `FILES_ONLY` constant indicates files only
    - `DIRECTORIES_ONLY` constant indicates directories only
  - Method `showOpenDialog` displays `JFileChooser` dialog titled Open, with Open and Cancel buttons (to open a file/directory or dismiss the dialog, respectively)
    - `CANCEL_OPTION` constant specifies that user click Cancel button
  - Method `getSelectedFile` retrieves file or directory user selected

```java
// Fig. 14.22: FileDemonstration.java
// Demonstrating the File class.
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();

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

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

```

Class for display `JFileChooser` dialog

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

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

   // allow user to specify file name
   private File getFile()
   {
      // display file dialog, so user can choose file 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

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

```

Create `JFileChooser`

Allow user to select both files and directories

Display dialog

User clicked Cancel

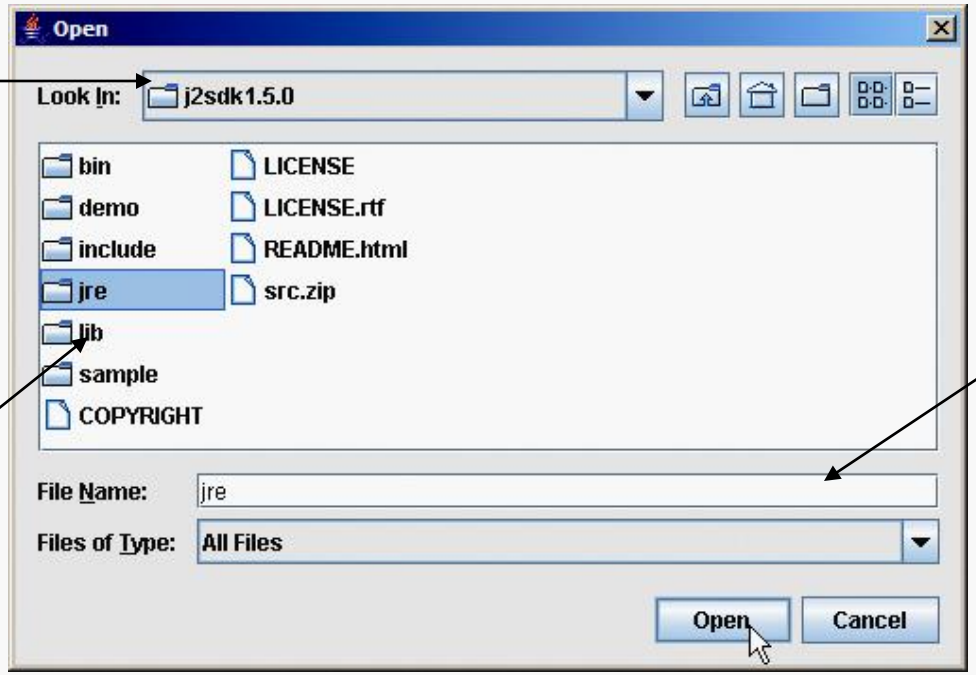
Retrieve file or directory selected by user

```java
      return fileName;
   } // end method getFile

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

      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() ) );

```

Display information about file

```
         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
```

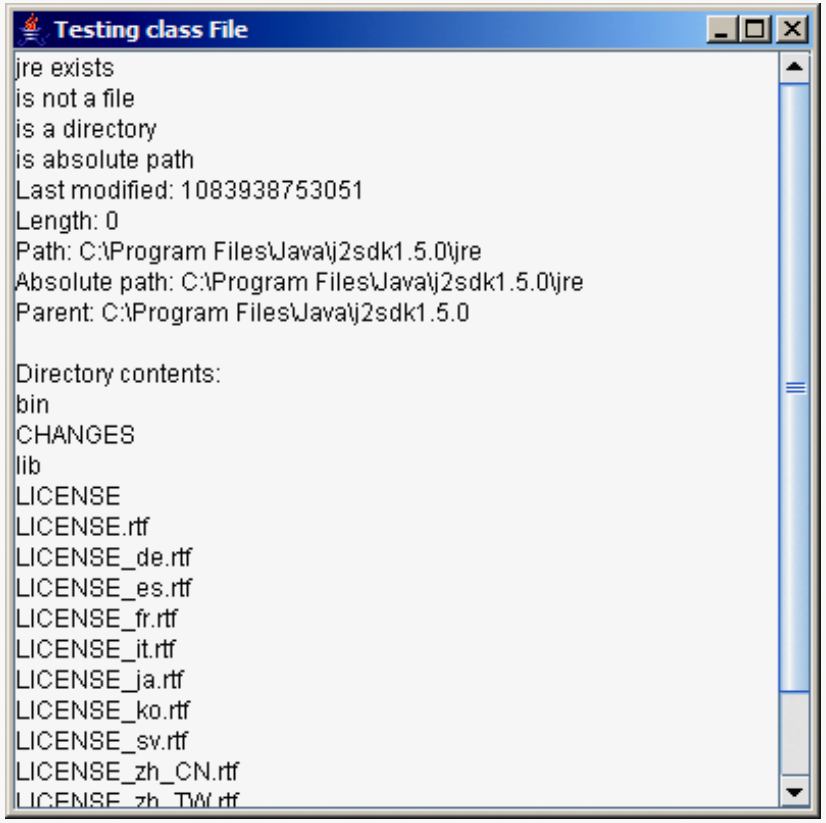
```java
// Fig. 14.23: FileDemonstrationTest.java
// Testing the FileDmonstration class.
import javax.swing.JFrame;

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



Select location for file here

Files and directories are displayed here

Click **Open** to submit new file name to program


Testing class File
jre exists
is not a file
is a directory
is absolute path
Last modified: 1083938753051
Length: 0
Path: C:\Program Files\Java\j2sdk1.5.0\jre
Absolute path: C:\Program Files\Java\j2sdk1.5.0\jre
Parent: C:\Program Files\Java\j2sdk1.5.0
Directory contents:
bin
CHANGES
lib
LICENSE
LICENSE.rtf
LICENSE_de.rtf
LICENSE_es.rtf
LICENSE_fr.rtf
LICENSE_it.rtf
LICENSE_ja.rtf
LICENSE_ko.rtf
LICENSE_sv.rtf
LICENSE_zh_CN.rtf