Streams, File I/O, and Networking

Chapter 10

*JAVA: An Introduction to Problem Solving & Programming, 7*th Ed. By Walter Savitch
ISBN 0133862119

Objectives

- Describe the concept of an I/O stream
- Explain the difference between text and binary files
- Save data, including objects, in a file
- Read data, including objects, in a file

Overview: Outline

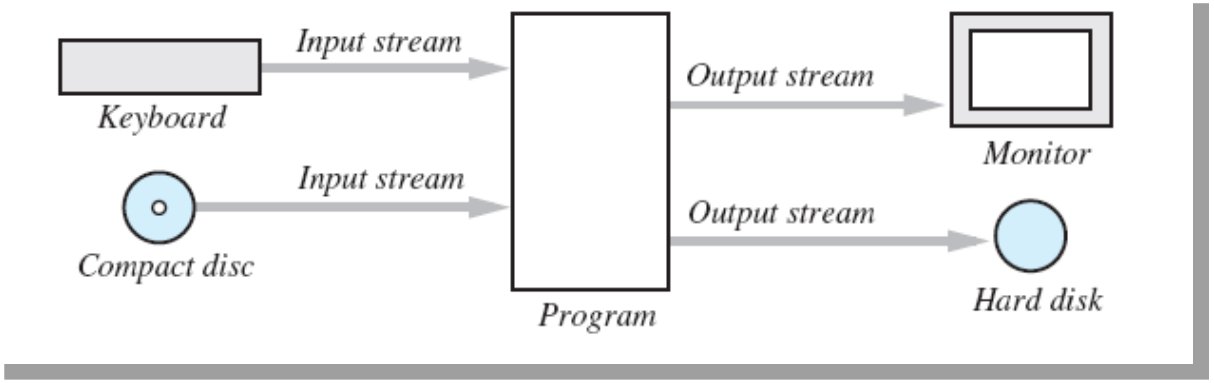
- The Concept of a Stream
- Why Use Files for I/O?
- Text Files and Binary Files

The Concept of a Stream

- Use of files
  - Store Java classes, programs
  - Store pictures, music, videos
  - Can also use files to store program I/O
- A *stream* is a flow of input or output data
  - Characters
  - Numbers
  - Bytes

The Concept of a Stream

- Streams are implemented as objects of special stream classes
  - Class **`Scanner`**
  - Object **`System.out`**
- Figure 10.1 I/O Streams



Why Use Files for I/O

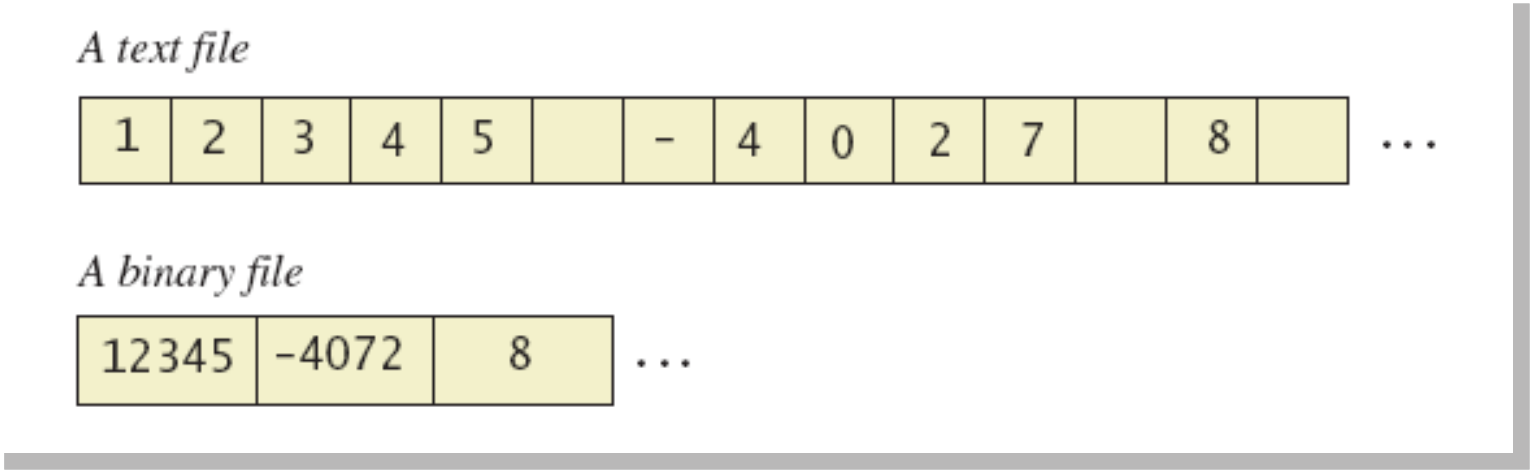
- Keyboard input, screen output deal with temporary data
  - When program ends, data is gone
- Data in a file remains after program ends
  - Can be used next time program runs
  - Can be used by another program

Text Files and Binary Files

- All data in files stored as binary digits
  - Long series of zeros and ones
- Files treated as sequence of characters called *text files*
  - Java program source code
  - Can be viewed, edited with text editor
- All other files are called *binary files*
  - Movie, music files
  - Access requires specialized program

Text Files and Binary Files

- Figure 10.2 A text file and a binary file containing the same values



Text-File I/O: Outlline

- Creating a Text File
- Appending to a text File
- Reading from a Text File

Creating a Text File

- Class **`PrintWriter`** defines methods needed to create and write to a text file
  - Must import package **`java.io`**
- To open the file
  - Declare *stream variable* for referencing the stream
  - Invoke **`PrintWriter`** constructor, pass file name as argument
  - Requires **`try`** and **`catch`** blocks

Creating a Text File

- File is empty initially
  - May now be written to with method **`println`**
- Data goes initially to memory buffer
  - When buffer full, goes to file
- Closing file empties buffer, disconnects from stream

Creating a Text File

- View sample program, listing 10.1

`class TextFileOutput`

```
Enter three lines of text:
A tall tree
in a short forest is like
a big fish in a small pond.
Those lines were written to out.txt
```

Sample screen output

Resulting File

```
1 A tall tree
2 in a short forest is like
3 a big fish in a small pond.
```

You can use a text editor to read this file.

Creating a Text File

- When creating a file
  - Inform the user of ongoing I/O events, program should not be "silent"
- A file has two names in the program
  - File name used by the operating system
  - The stream name variable
- Opening, writing to file overwrites pre-existing file in directory

Appending to a Text File

- Opening a file new begins with an empty file
  - If already exists, will be overwritten
- Some situations require appending data to existing file
- Command could be

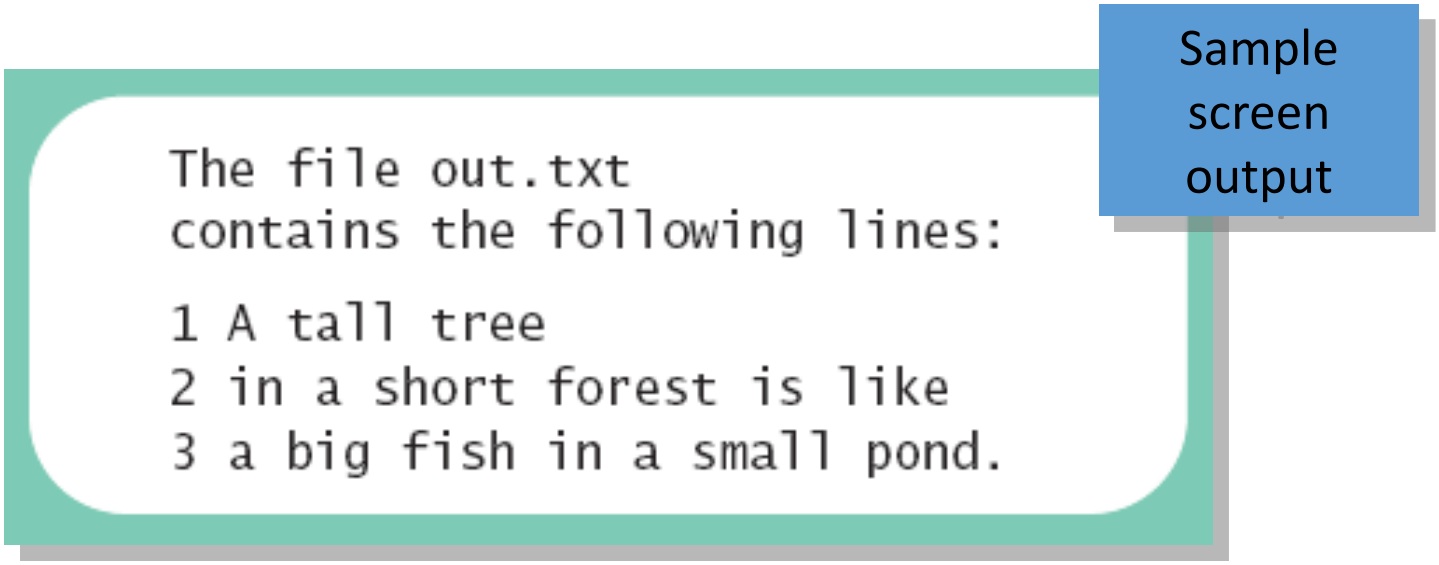
```
outputStream =
    new PrintWriter(
    new FileOutputstream(fileName, true));
```

- Method **println** would append data at end

Reading from a Text File

- Note text file reading program, listing 10.2
  `class TextFileInputDemo`
- Reads text from file, displays on screen
- Note
  - Statement which opens the file
  - Use of **`Scanner`** object
  - Boolean statement which reads the file and terminates reading loop

Reading from a Text File



Reading from a Text File

- Figure 10.3 Additional methods in class **`Scanner`**

Scannner_Object_Name`.hasNext()` Returns true if more input data is available to be read by the method `next`.
Scannner_Object_Name`.hasNextDouble()` Returns true if more input data is available to be read by the method `nextDouble`.
Scannner_Object_Name`.hasNextInt()` Returns true if more input data is available to be read by the method `nextInt`.
Scannner_Object_Name`.hasNextLine()` Returns true if more input data is available to be read by the method `nextLine`.

Techniques for Any File

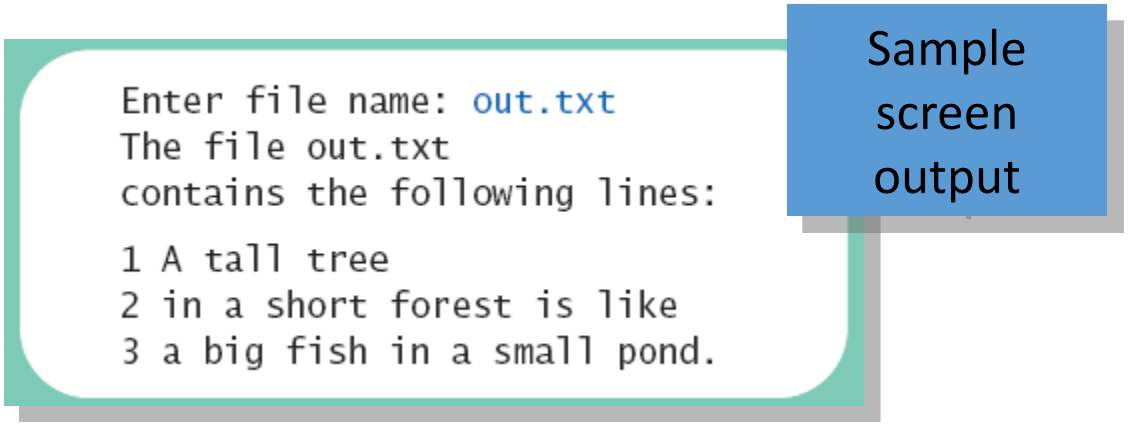
- The Class **`File`**
- Programming Example: Reading a File Name from the Keyboard
- Using Path Names
- Methods of the Class **`File`**
- Defining a Method to Open a Stream

The Class `File`

- Class provides a way to represent file names in a general way
  - A `File` object represents the name of a file
- The object
  ```
  new File ("treasure.txt")
  ```
  is not simply a string
  - It is an object that *knows* it is supposed to name a file

Programming Example

- Reading a file name from the keyboard
- View sample code, listing 10.3
  `class TextFileInputDemo2`



Using Path Names

- Files opened in our examples assumed to be in same folder as where program run
- Possible to specify path names
  - Full path name (absolute path)

    new File (“data\data1.txt”) ;

    f:\grandgrandma\grandma\mom\

    /home/someone/parent/child/.../

  - Relative path name

    “../../uncle/cousin1/../cousin2”

- Be aware of differences of pathname styles in different operating systems

Methods of the Class File

- Recall that a **`File`** object is a system-independent abstraction of file's path name
- Class **`File`** has methods to access information about a path and the files in it
  - Whether the file exists
  - Whether it is specified as readable or not
  - Etc.

Methods of the Class File

- Figure 10.4 Some methods in class **`File`**

`public boolean canRead()` Tests whether the program can read from the file.
`public boolean canWrite()` Tests whether the program can write to the file.
`public boolean delete()` Tries to delete the file. Returns true if it was able to delete the file.
`public boolean exists()` Tests whether an existing file has the name used as an argument to the constructor when the `File` object was created.
`public String getName()` Returns the name of the file. (Note that this name is not a path name, just a simple file name.)
`public String getPath()` Returns the path name of the file.
`public long length()` Returns the length of the file, in bytes.

Defining a Method to Open a Stream

- Method will have a **`String`** parameter
  - The file name
- Method will return the stream object
- Will throw exceptions
  - If file not found
  - If some other I/O problem arises
- Should be invoked inside a **`try`** block and have appropriate **`catch`** block

Defining a Method to Open a Stream

- Example code

```
public static PrintWriter openOutputTextFile(String fileName)
                    throws FileNotFoundException, IOException
{
    PrintWriter toFile = new PrintWriter(fileName);
    return toFile;
}
```

- Example call

```
PrintWriter outputStream = null;
try
{
    outputStream = openOutputTextFile("data.txt");
}
< appropriate catch block(s) >
```

Case Study
Processing a Comma-Separated Values File

- A comma-separated values or CSV file is a simple text format used to store a list of records
- Example from log of a cash register's transactions for the day:

```
SKU,Quantity,Price,Description
4039,50,0.99,SODA
9100,5,9.50,T-SHIRT
1949,30,110.00,JAVA PROGRAMMING TEXTBOOK
5199,25,1.50,COOKIE
```

Example Processing a CSV File

- View program that calculates total sales, listing 10.4 **`class TransactionReader`**
- Uses the split method which puts strings separated by a delimiter into an array

```
String line = "4039,50,0.99,SODA"
String[] ary = line.split(",");
System.out.println(ary[0]);          // Outputs 4039
System.out.println(ary[1]);          // Outputs 50
System.out.println(ary[2]);          // Outputs 0.99
System.out.println(ary[3]);          // Outputs SODA
```