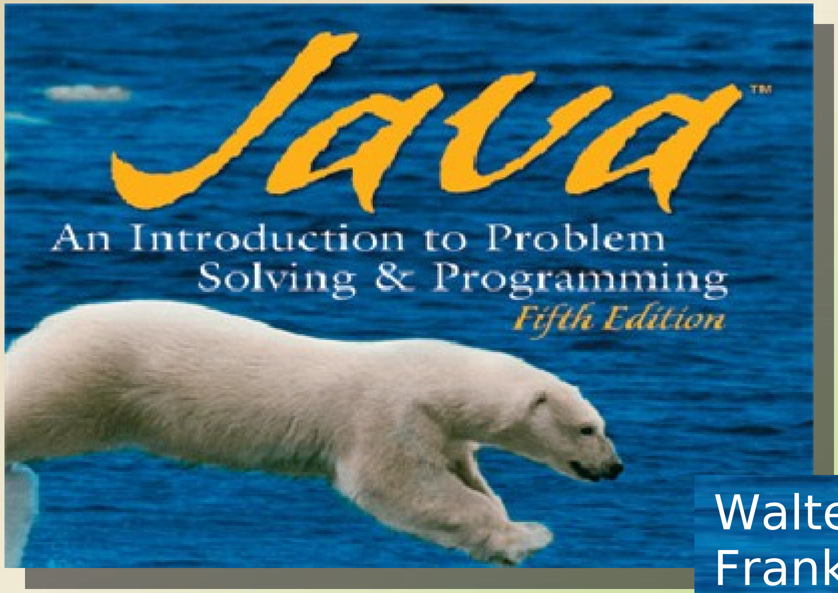


Walter Savitch
Frank M. Carrano

Streams and File I/O

Chapter 10

Objectives

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

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

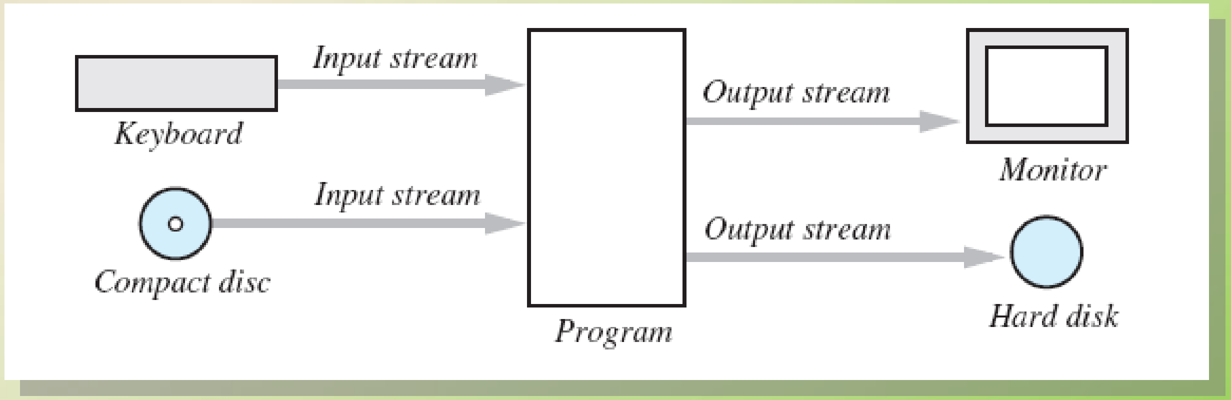
- A stream is a flow of data (characters, numbers, etc.).
- *input stream*
  - *Data flowing into a program*
  - Used for **reading** data
- *output stream*
  - *Data flowing out of a program*
  - Used for **writing** data

The Concept of a Stream

- A stream is implemented as an object.
  - Output streams deliver data to a destination such as a file or the screen
    - System.out
  - Input streams take data from a source such as a file or the keyboard, and deliver it to a program.
    - Scanner

The Concept of a Stream

- Figure 10.1
  I/O Streams



Why Use Files for I/O

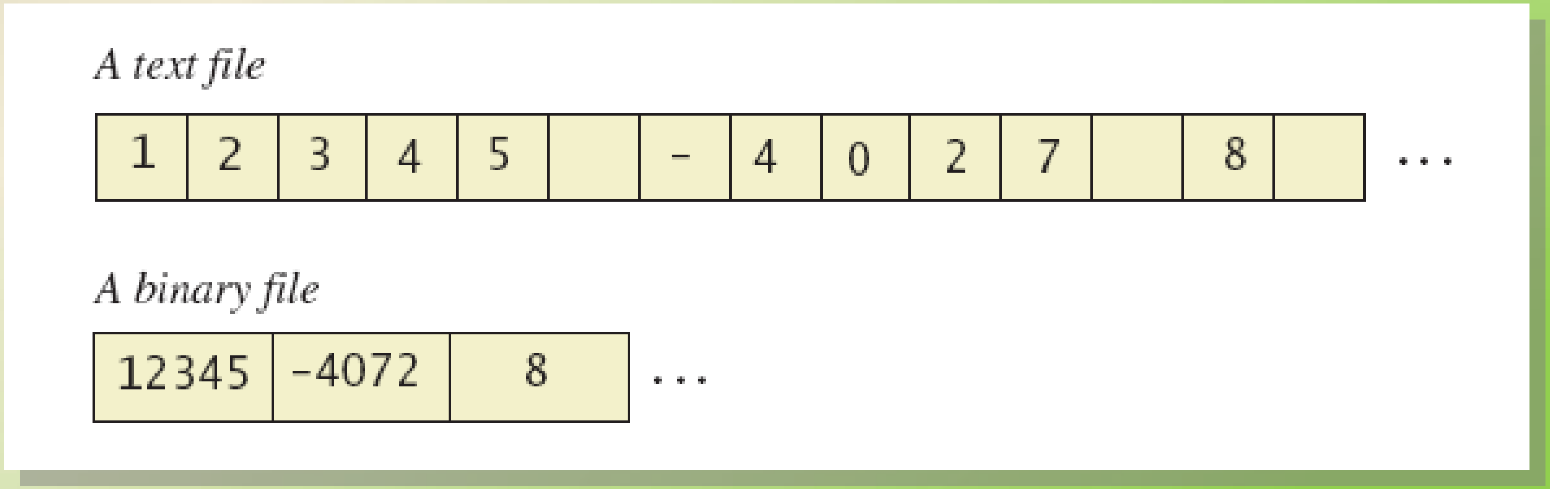
- Keyboard input, screen output deal with temporary data
  - When program ends, data is gone
- Data in a file remains after program ends
  - Can be used next time program runs
  - Can be used by another program

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
- Download from SavitchSrc link ch10
    - TextFileOutputDemo.java
    - TextFileInputDemo.java
    - TextFileInputDemo2.java

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

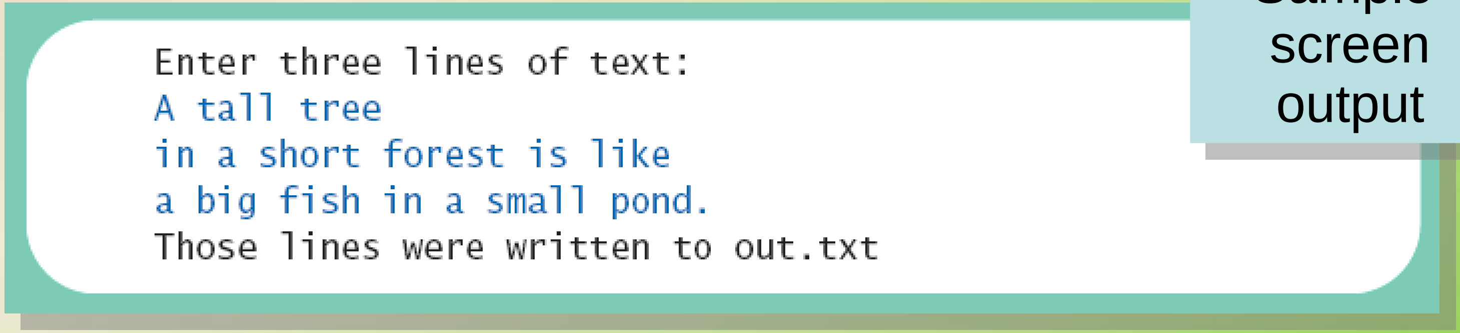
Creating a Text File

- 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

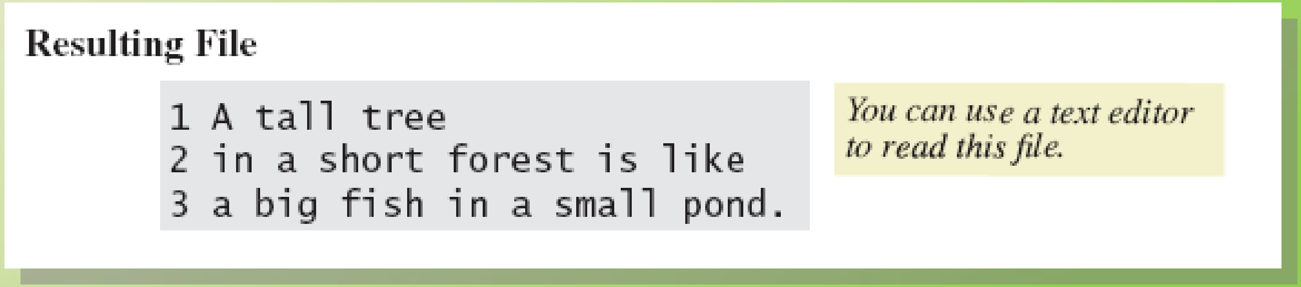
Creating a Text File

- View **`TextFileOutputDemo.java`**

Sample screen output



```
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
```



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

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

- Method `println` would append data at end

Reading from a Text File

- View `TextFileInputDemo.java`
- 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
- Prints file created by TextFileOutputDemo

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`**

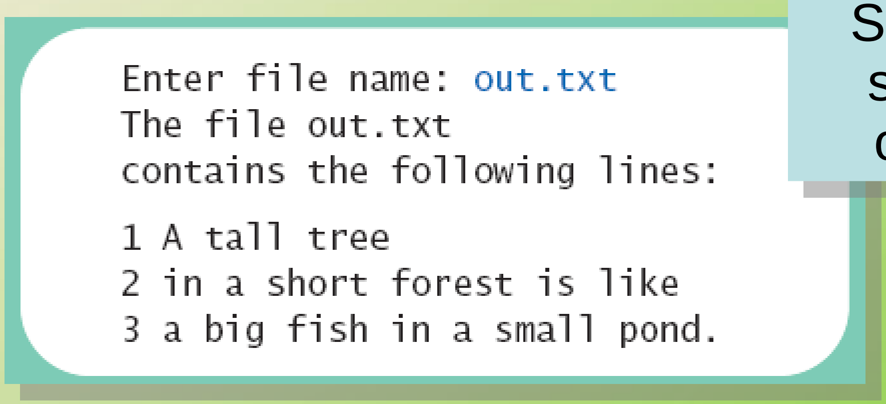
- Class provides a way to represent file names in a general way
  - A **`File`** object represents the name of a file
- The object

  ```
  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 **`TextFileInputDemo2`**



```
Enter file name: out.txt
The file out.txt
contains the following lines:

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

Sample screen output

Using Path Names

- Files opened in our examples assumed to be in same folder as where the program is being run
- Possible to specify path names
  - Full path name
  - Relative path name
- Be aware of differences of pathname styles in different operating systems

Using Path Names

- Safest to always use Unix-style pathnames, even under Windows:

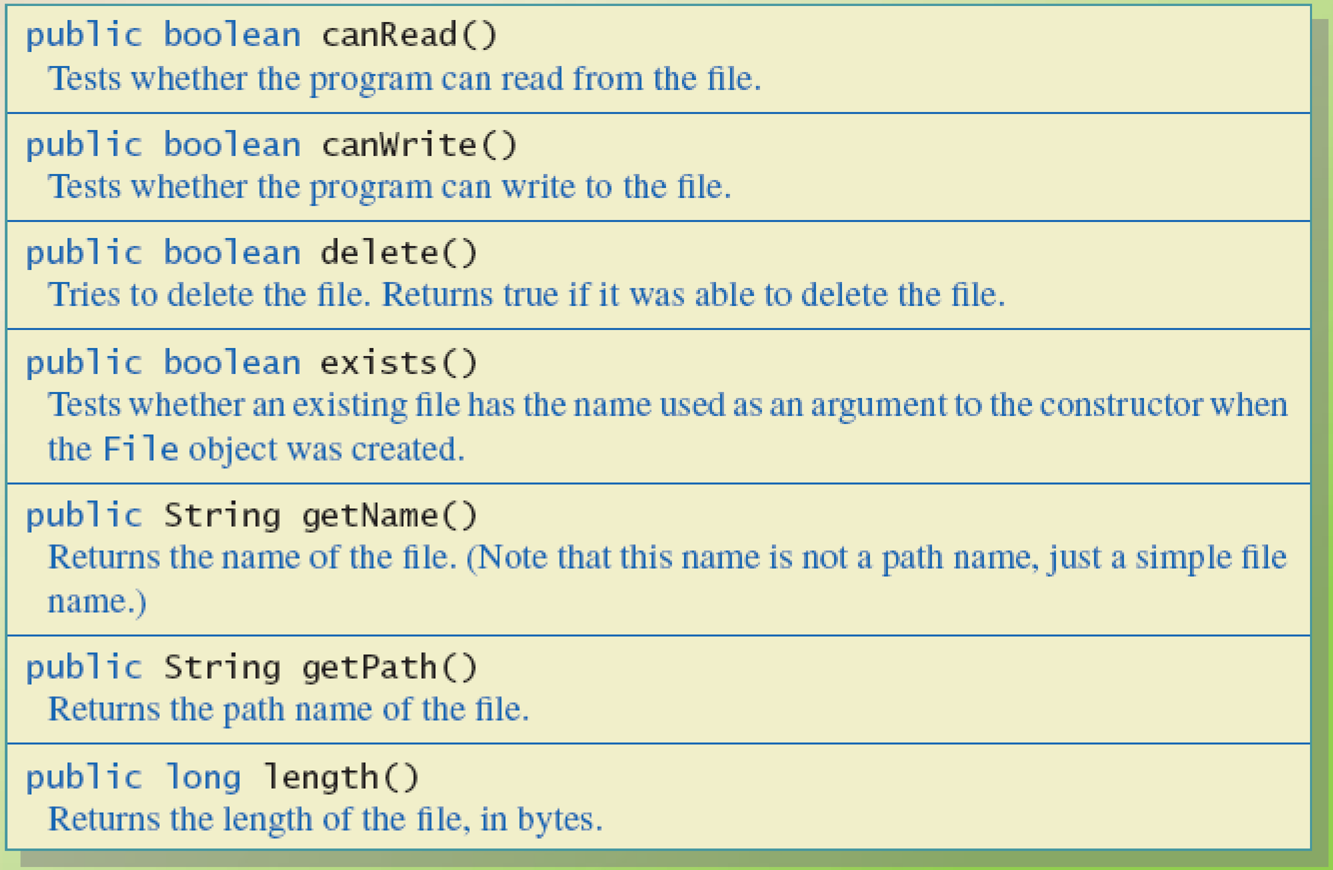
```
Scanner fileScan = new Scanner(new
  File(“D:/homework/hw1/data.txt”));
```

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.

FileUtils.java

- Download source files from Examples I/O
- Scanner is very slow
  - not appropriate for reading large data sets
- When writing a file, best to have control over encoding and overwrite/append options
- FileUtils has static methods for creating PrintWriter and BufferedReader objects with various parameters

FileUtils.java

```
PrintWriter openPrintWriter(String fileName)
PrintWriter openPrintWriter(File aFile)
PrintWriter openPrintWriter(String fileName, boolean append)
PrintWriter openPrintWriter(File aFile, boolean append)

BufferedReader openBufferedReader(String fileName)
BufferedReader openBufferedReader(File aFile)
BufferedReader openBufferedReader(String fileName, String encoding)
BufferedReader openBufferedReader(File aFile, String encoding)
```

Using PrintWriter

```java
PrintWriter dest;
String text = "Hi Mom!";
int num = 42;

try
{
   dest = FileUtils.openPrintWriter("tmp.txt");
   dest.println(text + " My favorite number is " + num);
   dest.close(); // must close file
}
catch (UnsupportedEncodingException e)
{
   System.out.println("Bad encoding. " + e.getMessage());
   System.exit(0);
}
catch (FileNotFoundException e)
{
   System.out.println(e.getMessage());
   System.exit(0);
}
```

Using BufferedReader

```
BufferedReader src;
String line;
String[] fields;

try {
    src = FileUtils.openBufferedReader("input.txt");
    while ((line = src.readLine()) != null) {
        line = line.trim(); // must trim before splitting
        fields = line.split("\\s+");
        // process fields
    }

    src.close(); // must close file

} catch (UnsupportedEncodingException e) {
    System.out.println("Bad encoding. " + e.getMessage());
    System.exit(0);
} catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
    System.exit(0);
} catch (IOException e) {
    System.out.println(e.getMessage());
    System.exit(0);
}
```