



# 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 in a file

# The Concept of a Stream

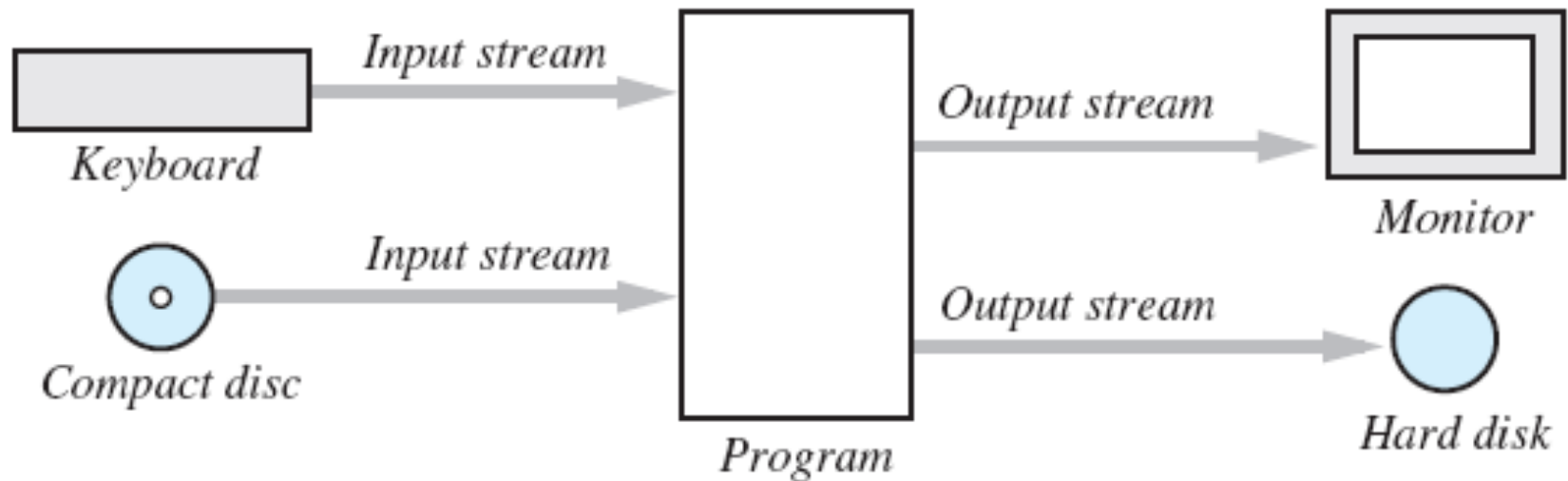
- Use of files
  - Store Java classes, programs
  - Store pictures, music, videos
  - Can also use files to store program I/O
- *A stream* is a flow of input or output data
  - Characters
  - Numbers
  - Bytes
- A stream is implemented as an object.

# The Concept of a Stream

- Input streams take data from a source such as a file or the keyboard, and deliver it to a program.
  - *Data flowing into a program*
  - Used for **reading** data
  - Example: **Scanner**
- Output streams deliver data to a destination such as a file or the screen.
  - *Data flowing out of a program*
  - Used for **writing** data
  - Examples: **System.out**, **System.err**

# 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-File I/O: Outline

- Creating a Text File
- Appending to a text File
- Reading from a Text File
- Download from SavitchSrc link ch10
  - **TextFileOutputDemo.java**
  - **AppendTextFile.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:  
**PrintWriter outputStream;**
  - Invoke **PrintWriter** constructor, pass file name as argument:  
**outputStream = new PrintWriter(fileName) ;**
  - Requires **try** and **catch** blocks

# Creating a Text File

- File is empty initially
- May now be written to with methods `println` and `print` which work the same for writing text to a file than `System.out.println` and `System.out.print` works for writing to the screen:

```
outStream.println("Line 1");  
outStream.print("Line 2");
```

# Creating a Text File

- Data goes initially to memory buffer
  - When buffer full, goes to file
- Closing file empties buffer, disconnects from stream:  
`outStream.close();`

# Creating a Text File

- View `TextFileOutputDemo.java`

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 and 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));
```
- Methods `println` and `print` would append data at the end
- View `AppendTextFile.java`

# 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.java`



# Reading from a Text File

Sample  
screen  
output

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.
```

# Reading from a Text File

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

*Scanner\_Object\_Name*.hasNext()

Returns true if more input data is available to be read by the method next.

*Scanner\_Object\_Name*.hasNextDouble()

Returns true if more input data is available to be read by the method nextDouble.

*Scanner\_Object\_Name*.hasNextInt()

Returns true if more input data is available to be read by the method nextInt.

*Scanner\_Object\_Name*.hasNextLine()

Returns true if more input data is available to be read by the method nextLine.

# Techniques for Any File: Outline

- 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.java**

```
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 program is being run
- Possible to specify path names
  - Full path name:  
**"D:/homework/hw1/data.txt"**
  - Relative path name  
**"../data.txt"**

# Using Path Names

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

`"/home/username/hw/hw1/data.txt"`

`"D:/homework/hw1/data.txt"`

- 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.
- View **FileClassDemo.java**

# 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.

# Examples: File I/O

- **Scanner** is very slow
  - not appropriate for reading large data sets
- Use **PrintWriter** and **BufferedReader** instead
- Download source files from **Examples/File I/O**
  - **UTF8FileInputDemo.java**
  - **UTF8FileOutputDemo.java**
  - **UTF8CopyFile.java**

# Arguments to a Java Program

- **UTF8FileInputDemo** expects one argument:

```
public static void main(String[] args) {  
    // make sure number of arguments is correct  
    if (args.length != 1) {  
        System.out.println("Usage: java " +  
                           UTF8FileInputDemo <filename>");  
        System.exit(0);  
    }  
    ...  
}
```

- Call the program with an argument (filename) behind the class name – e.g. on command line:

```
java UTF8FileInputDemo file.txt
```

# FileUtils.java

- When writing a file, best to have control over encoding and overwrite/append options
- Download **FileUtils.java** from **Examples/File I/O**
- **FileUtils.java** has static methods for creating **PrintWriter** and **BufferedReader** objects with various parameters
- Also see **UTF8CopyFile2.java**

# Methods in **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 fName, String encoding)**

**BufferedReader openBufferedReader(File aFile, String encoding)**

# Using **PrintWriter**

```
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);  
}
```

# Case Study

## Processing a CSV 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 `TransactionReader.java`
- 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
```

# Try with resources

`try` statement that declares resources.

Resource is an object that must be closed after use. (A file, a network connection....)

Resource is an object that must implement the `AutoCloseable` interface.

Resource is automatically closed at the end of the `try` block no matter whether the try block completes successfully or throws an error. No need for a `finally` statement.

# Try with resource Syntax

```
try (  
    <declare and open 1 or more  
    resources separated by ;>  
) {  
    Work with resources  
}  
  
catch () {  
}
```

# Try with resources example

```
try (  
    BufferedReader br = new  
        FileReader("/tmp/test.txt"))  
{  
    String line;  
    while ((line = br.readLine()) != null)  
    {  
        System.out.println(line);  
    }  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

# Using several resources

```
try ( InputStream in = new
      FileInputStream(src) ;
      OutputStream out = new
      FileOutputStream(dest) )
{
    // Do something with in and out
}
```



# Classes implementing Autoclosable

- Classes from `java.io` like
  - `InputStream` and `OutputStream` and their subclasses
  - `Reader` and `Writer` and their subclasses.
- The class in `java.util.zip.Zipfile`.
- Class `java.net.ServerSocket` and its subclasses.
- Classes in `java.sql` like `Statement` and `Connection`

---

# AutoCloseable interface

```
public interface AutoCloseable
{
    void close() throws Exception;
}
```

To use your own classes with try with resources they must implement the Autoclosable interface.

Visit the [Tutorial](#) at the Oracle Java site to see more examples.