

# File I/O



covering

- \*\* Saving Data: Serialisation *versus* Using a Text File
- \*\* Java I/O Connection and Chain Streams
- \*\* Reading from/Writing to a Text File: Java Classes
- \*\* **File** objects (`java.io.File`)



Chapter 16 – “Head First Java” book

Chapters 14,19 (sections 14.10, 19.1-19.3) – “Introduction to Java Programming” book

Chapter 10 – “Java in a Nutshell” book

# Saving Data/State (1/2)


- Data stored in variables, arrays, objects is temporary: once a program has finished executing, information is lost!
  - Example: Java program that *counts the number of characters and words in a line of text*.
    - Once program has run and displayed statistics, that output is lost if not saved somewhere!



Can you think of other examples?

- Saving data requires information to be stored in a file on a disk/CD.
  - How a program's data is stored depends on what the user intends to do with the data!

# Saving Data/State (2/2)

- There are two ways of saving data:
  - Using serialisation  Out of scope in this course!
  - The data stored will only be used by the Java program that generated it.
  - Example: A program wants to save its current state so that it can be loaded at a later date.
- Using a file (such as a plain text file)
  - The data stored in the file needs to be used by other programs.
  - Example: A .csv (comma separated values) file can be read by spreadsheet programs (such as Excel).

# What is I/O?

- Computer programs need to **interact** with the world:
  - *Bring in* information from an external source;
  - *Send out* information to an external destination.
- This interaction is what we refer to as **Input/Output**:
  - *Input*: to bring in information (*read*)
  - *Output*: to send out information (*write*)
- Information for **Input/Output** can be:
  - *anywhere*: memory, disk, in a file, over the network, in another program ...
  - *of any type* (any object): Text, Image, Audio, Video ...

# Examples: I/O Devices

*To be completed in class ...*

- Monitor output
- Printer output
- Scanner input
- Speaker output
- Hard disk input / output
- Keyboard input
- Mouse input



Which are **input** devices?  
Which are **output** devices?



There are many other  
examples of I/O devices ...

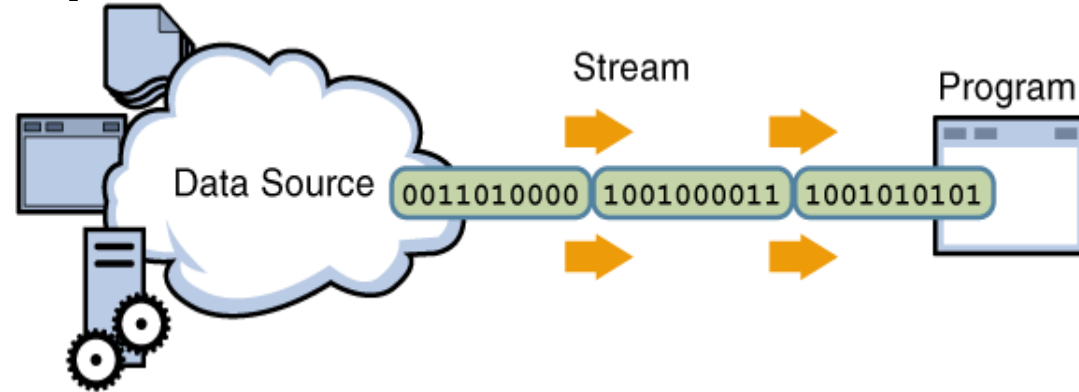
# Streams

- Java input/output makes use of *streams*:
  - A stream is a **connection to a source of data** or **to a destination for data** (sometimes both).
  - **Streams can represent any data**, so a stream is a sequence of bytes that flow from a source to a destination.
- In a program, we *read information* from an input stream and *write information* to an output stream.
- A program can manage multiple streams simultaneously.

# Input (*reading*) & Output (*writing*)

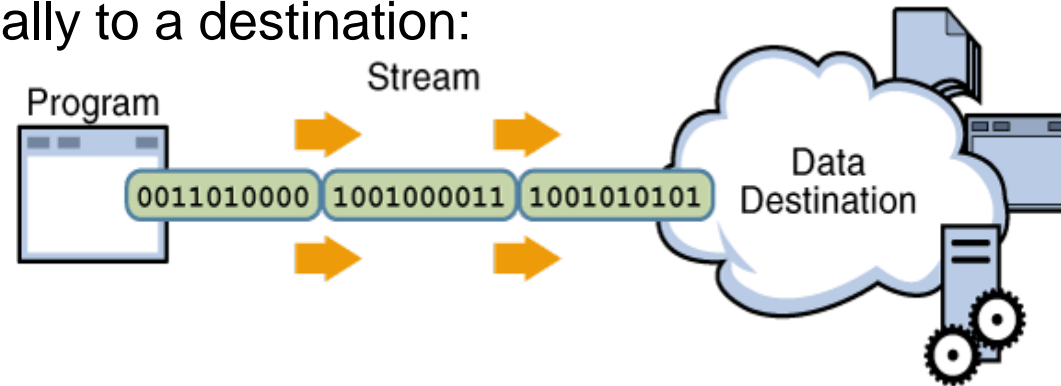
Program *reads a stream* sequentially from a source:

1. Open the stream.
2. Use the stream:  
while more information (data)  
*read* information (data)
3. Close the stream.



Program *writes a stream* sequentially to a destination:

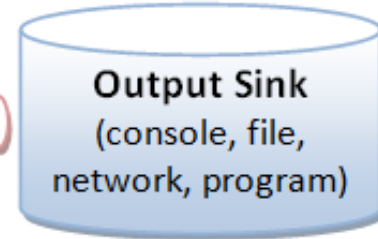
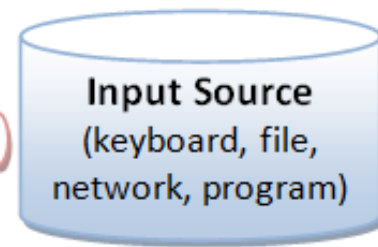
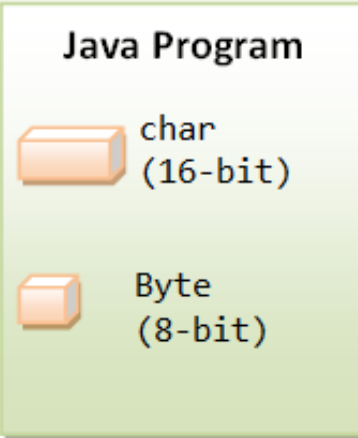
1. Open the stream.
2. Use the stream:  
while more information (data)  
*write* information (data)
3. Close the stream.



# Streams (again ...)

“Character” Streams  
(Reader/Writer)

“Byte” Streams  
(InputStream/  
OutputStream)



Internal Data Formats:

- Text (char): UCS-2
- int, float, double, etc.

External Data Formats:

- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

- Java has two broad categories of streams:
  - **byte streams**, for **machine-formatted data**
    - **InputStream**
    - **OutputStream**
  - **character streams** (textual), for **human-readable data**
    - **Reader**
    - **Writer**



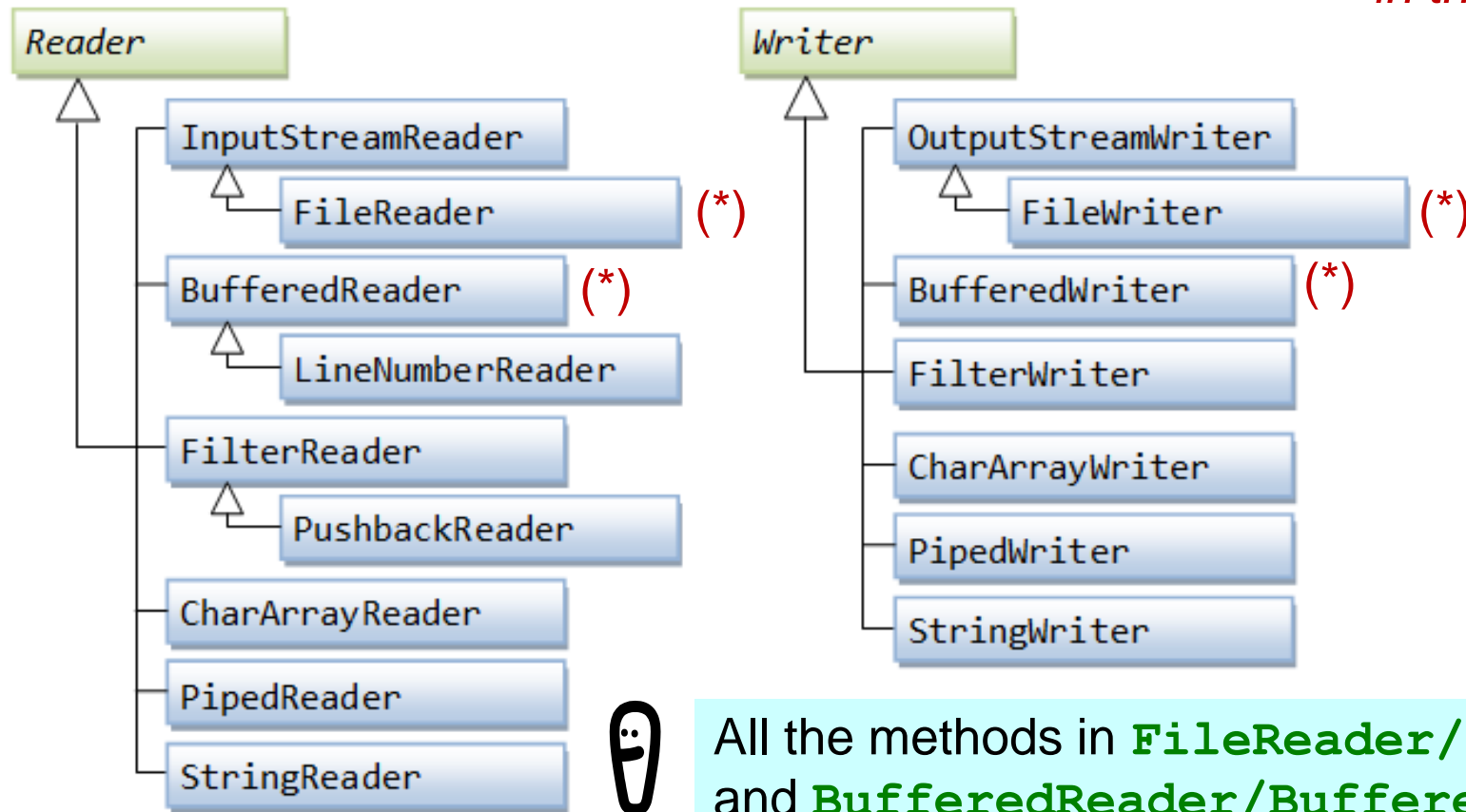
We do not cover  
binary I/O.

- **Text files** contain **data** represented in **human-readable form**.
  - Example: **.java files**.
- **Binary files** contain **data** represented in **binary form**.
  - Designed to be read by programs, but more efficient to process than text files. Example: **.class files**.



# I/O Classes

(\*) I/O classes used in this course.



All the methods in **FileReader/FileWriter** and **BufferedReader/BufferedWriter** are inherited from its superclasses.

# java.io.File Class (1/2)

- **Files** live in directories within the file system.
  - Complete file name (represented by a **String**) consists of the **path** + **name of file**.

**Example:**

```
c:\Work\JavaPrograms\MyFirstJavaProgram.java
```

directory path

file name

- **java.io.File**: contains **methods to obtain file properties**, for **renaming** and **deleting files**.
  - A wrapper class for a file's name and directory path: **represents an abstract pathname**.
  - It hides file system differences.
  - No exception is thrown **if file does not exist**.

# java.io.File Class (2/2)

- Constructors and methods in **File**:

**File**(String pathname): creates file with specified pathname

boolean **exists**() / boolean **isDirectory**() / boolean **isFile**()

boolean **canRead**() / boolean **canWrite**()

boolean **delete**(): returns **true** if file successfully deleted

String **getAbsolutePath**(): returns complete absolute  
file/directory name

boolean **renameTo**(File dest): returns **true** if operation successful

long **length**(): returns length of the file in bytes

String[] **list**(): returns an array of strings containing the list of files in  
this directory

boolean **mkdir**()



**java.io.File** in Java SE6, but  
**java.nio.file.Path** from Java SE7.

# Example: Using the File Class

```
import java.io.*;

public class TestFileClass {
    public static void main(String[] args) {
        File file = new File("Examples\\badger.jpg");
        System.out.println("Does it exist? " + file.exists());
        System.out.println("Can it be read? " + file.canRead());
        System.out.println("Can it be written? " + file.canWrite());
        System.out.println("What is its absolute path?" + file.getAbsolutePath());
        System.out.println("What is its name?" + file.getName());
        System.out.println("What is its path?" + file.getPath());
    }
}
```

Output is ...

```
> java TestFileClass
Does it exist? true
Can it be read? true
Can it be written? false
What is its absolute path? C:\\EBU4201\\Examples\\badger.jpg
What is its name? badger.jpg
What is its path? Examples\\badger.jpg
```

# Steps: Reading from / Writing to files

## 1. *Open file*

- Needs the file's name and maybe its location (path).
- Open file by creating an instance of an appropriate stream class.

## 2. *Perform operations*

- Read from and/or write to the file.
- Call instance methods that belong to the stream object's class.

## 3. *Close file*

- Any class from `InputStream`, `OutputStream`, `Reader` and `Writer` has a `close()` method.
- File I/O can cause a large number of *exceptions* to be thrown.

# Reading a text file // Writing a text file

- **FileReader:**
  - To represent a file on the file system.
  - The file containing character data.
- **BufferedReader:**
  - Improves efficiency.
  - Provides useful methods.
- The **FileReader** and **BufferedReader** together create a convenient *text file input stream*.

- **FileWriter:**
  - To represent a file on the file system.
  - The methods are limited.
- **BufferedWriter:**
  - Improves efficiency.
  - Provides useful methods.
- The **FileWriter** and **BufferedWriter** together create a convenient *text file output stream*.

# FileReader *versus* FileWriter

- **FileReader**: A `java.io.FileNotFoundException` will occur if you attempt to create a **FileReader** with a nonexistent file.

<pre>public <b>FileReader</b>(String filename) public <b>FileReader</b>(File file) int <b>read</b>()</pre>		<pre>int <b>read</b>(char[] cbuf) int <b>read</b>(char[] cbuf,int off,int len) void <b>close</b>()</pre>
------------------------------------------------------------------------------------------------------------	--	----------------------------------------------------------------------------------------------------------

- **FileWriter**: If the file doesn't exist, a new file will be created.

<pre>public <b>FileWriter</b>(String filename) public <b>FileWriter</b>(String filename, boolean append) public <b>FileWriter</b>(File file, boolean append)</pre>	<pre>public <b>FileWriter</b>(File file)</pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------

```
void write(int c)
void write(byte[] cbuf)
void write(char[] cbuf,int off,int len)
void write(String str)
void write(String str, int off,int len)
void close()
```

# Example 1: Reading a text file (with 1 line)

```
import java.io.*;

public class FileReadTest {
    public static void main(String args[]){
        String fileName = "input.txt";
        String contents = "";
        try {
            FileReader fileReader = new FileReader(fileName);
            BufferedReader bufferedReader = new BufferedReader(fileReader);
            contents = bufferedReader.readLine();
            bufferedReader.close();
            fileReader.close();
        }
        catch (IOException e) {
            System.out.println("Errors occurred");
            System.exit(1);
        }
        System.out.println(contents);
    }
}
```



This is the 'lazy' approach to catching IO exceptions, because this example can generate at least 2 different types of exceptions: **FileNotFoundException** and **IOException**.



## Example 2: Reading a text file (with several lines)

```
// other code ...
try {
    FileReader fileReader = new FileReader(fileName);
    BufferedReader bufferedReader = new BufferedReader(fileReader);
    String oneLine = bufferedReader.readLine();
    while (oneLine != null) {
        contents = contents + oneLine;
        oneLine = bufferedReader.readLine();
    }
    bufferedReader.close();
    fileReader.close();
}
// other code ...
```

## Example 3: Reading a text file (with numbers) + processing data

```
// other code ...
int sum = 0;
String fileName = "input.txt";
try {
    FileReader fileReader = new FileReader(fileName);
    BufferedReader bufferedReader = new BufferedReader(fileReader);
    String oneLine = bufferedReader.readLine();
    while (oneLine != null) {
        sum = sum + Integer.parseInt(oneLine);
        oneLine = bufferedReader.readLine();
    }
    bufferedReader.close();    fileReader.close();
}
catch (IOException e) {
    System.out.println("Errors occurred");    System.exit(1);
}
System.out.println(sum);
// other code ...
```

# Example: Writing a string to a text file

```
// other code ...  
String contents = "Welcome to BUPT.";  
String fileName = "output.txt";  
try {  
    FileWriter fileWriter = new FileWriter(fileName);  
    BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);  
    bufferedWriter.write(contents);  
    bufferedWriter.close();  
    fileWriter.close();  
}  
catch (IOException e) {  
    System.out.println("Errors occurred");  
    System.exit(1);  
}  
// other code ...
```

# Practice Exercise 1

1. Will the following code compile correctly?

```
File file = new File("temp.txt");  
FileReader in = new FileReader(file);
```

---

2. Does constructing a **File** object automatically create a disk file?
- 

3. What method ensures that data from previous calls to **write()** is sent to disk and leaves the file open?
- 

4. What does the following constructor do?

```
FileWriter fw = new FileWriter("myFile.txt");
```