Book Report **Name o**f the book

short line

Your Name  
4th September, 20XX

I/O Stream

Different operating systems use different file systems to manage their data. But the JVM will automatically connect to the local file system, allowing you to perform the same operations across multiple platforms. Different operating systems vary in their format of pathnames. For example, Unix‐based systems use the forward slash, /, for paths, whereas Windows‐based systems use the backslash \ character. Java offers two options to retrieve the local separator character: a system property and a static variable defined in the File class

**System.out.println(System.getProperty("file.separator"));**

**System.out.println(java.io.File.separator);**

java.io.File class is used to read information about

* existing files and directories,
* list the contents of a directory,
* create/delete files and directories.

The File class cannot read or write data within a file, although it can be passed as a reference to many stream classes to read or write data

File operator usually takes a String which can be either an absolute path or relative path to the current directory. There are 3 ways to create files.

**File file1 = new File("text.txt"); Pass the relative or absolute path**

**File file2 = new File("test.txt", "new.txt"); Pass the path of a parent file to create child file**

**File file3 = new File(file1, "new.txt"); Pass the reference of parent file**

Note: Creating file object doesn't create a file in system

**System.out.println(file1.exists()); //Without file will be false**

# FileMethods

**file1.exists(); Returns true if file/directory exists**

**file1.delete(); Returns true if file/directory deleted**

**file1.getAbsolutePath(); Returns String path of file/directory**

**file1.getName(); Returns string Name of file/directory**

**file1.getParent(); Returns parent folder directory returns null if passed like this "test.txt"**

**file1.isDirectory(); Returns true if it is directory**

**file1.isFile(); Returns true if it is file**

**file1.lastModified(); Returns long timeStamp**

**file1.length(); Return long file size**

**file1.listFiles(); Returns list of files in directory**

**file1.mkdir();**

**file1.mkdirs();**

**File dest = new File("new.txt");**

**System.out.println(file1.renameTo(dest));**

Its important to notice from where we are running the java command in case of relative paths. For example if my class with below code is in folder src-->javatest-->Test.java. It also has a file test.txt. Now if I run file.exists() from src as java .\javaTest\Test.java it will return false as it will check for test.txt in src. But if we run it from javatest then it will find the file and return true

# I/O Stream

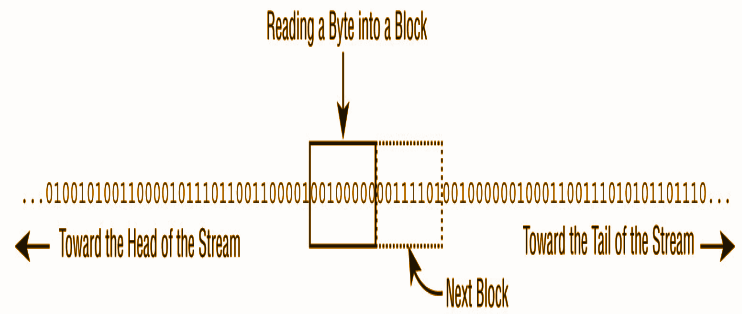
The contents of a file may be accessed or written via a stream, which is a list of data elements presented sequentially. Streams are like long, nearly never‐ending “streams of water” with data presented one “wave” at a time. The stream is so large that once we start reading it, we have no idea where the beginning or the end is. We just have a pointer to our current position in the stream and read data one block at a time.

Each type of stream segments data into a “wave” or “block” in a particular way. For example, some stream classes read or write data as

* Individual bytes.
* Individual characters or strings of characters
* Larger groups of bytes or characters at a time.

Irrespective of how the data is read or written nearly all are built on top of reading or writing an individual byte or an array of bytes at a time.

The reason higher-level streams exist is for convenience, as well as performance. For example, writing a file one byte at a time is time-consuming and slow in practice because the round-trip between the Java application and the file system is relatively expensive. By utilizing a BufferedOutputStream, the Java application can write a large chunk of bytes at a time, reducing the round-trips and drastically improving performance.



Files can be big like 1 TB. What stream does is helps reading chunks of data thus allowing systems with smaller specifications to be able to read data.

# Stream Types

## Byte vs. Character Streams

Stream can be classified as

| **Byte Stream** | **Character streams** |
| --- | --- |
| Which read/write binary data ( 0s and 1s) | Which read/write text data |
| have class names that end in InputStream or OutputStream | have class names that end in Reader or Writer |
| The byte streams are primarily used to work with binary data, such as an image or executable file | character streams are used to work with text files |
| FileInputStream | FileReader |

Even though character streams do not contain the word Stream in their class name, they are still I/O streams

Byte stream classes can write all types of binary data, including strings. So one might think that Character streams like reader are not necessary. However Character streams specifically focused on managing character and string data so that one doesn’t worry about the underlying character encoding of the file.

Java supports a wide variety of character encodings

**Charset usAsciiCharset = Charset.forName("US-ASCII");**

**Charset utf8Charset = Charset.forName("UTF-8");**

**Charset utf16Charset = Charset.forName("UTF-16");**

## Input VS Output Streams

Most InputStream stream classes have a corresponding OutputStream class, and vice versa. For example, the FileOutputStream class writes data that can be read by a FileInputStream. If you understand the features of a particular Input or Output stream class, you should naturally know what its complementary class does

It follows, then, that most Reader classes have a corresponding Writer class. For example, the FileWriter class writes data that can be read by a FileReader

**Exception**

**PrintWriter** no reader **PrintStream** has no Input stream

## Low vs High level Stream

A low‐level stream connects directly with the source of the data, such as a file, an array, or a String. Low‐level streams process the raw data or resource and are accessed in a direct and unfiltered manner

A high‐level stream is built on top of another stream using wrapping. Wrapping is the process by which an instance is passed to the constructor of another class

**try (var reader = new BufferedReader(new FileReader("text.txt"))) {**

**System.out.println(reader.readLine());**

**} catch (IOException e) {**

**e.printStackTrace();**

**}**

In this example, FileReader is the low‐level stream reader, whereas BufferedReader is the high‐level stream that takes a FileReader as input. Many operations on the high‐level stream pass through as operations to the underlying low‐level stream, such as read() or close(). Other operations override or add new functionality to the low‐level stream methods. The high‐level stream may add new methods, such as readLine(), as well as performance enhancements for reading and filtering the low‐level data.

High‐level streams can take other high‐level streams as input

**try (var ois = new ObjectInputStream(new BufferedInputStream(new FileInputStream("text.txt")))) {**

**System.out.print(ois.readObject());**

**} catch (ClassNotFoundException e) {**

**e.printStackTrace();**

**} catch (IOException e) {**

**e.printStackTrace();**

**}**

Buffered classes read or write data in groups, rather than a single byte or character at a time. The performance gain from using a Buffered class to access a low‐level file stream cannot be overstated. Unless you are doing something very specialized in your application, you should always wrap a file stream with a Buffered class in practice.

The java.io library defines four abstract classes that are the parents of all stream classes defined within the API: **InputStream, OutputStream, Reader, and Writer**. The constructors of high‐level streams often take a reference to the abstract class. For example, BufferedWriter takes a Writer object as input, which allows it to take any subclass of Writer.

**InputStream (Abstract) --> FileInputStream --> BufferedInputStream**

**OutputStream(Abstract) --> FileOutputStream--> BufferedOutputStream**

**Reader(Abstract)--> FileReader --> BufferedReader**

**Writer(Abstract)-->FileWriter -->BufferedWriter**

**new BufferedInputStream(new FileReader("z.txt")); *// DOES NOT COMPILE Reader wrapped in Input Stream***

**new BufferedWriter(new FileOutputStream("z.txt")); *// DOES NOT COMPILE Same***

**new ObjectInputStream( new FileOutputStream("z.txt")); *// DOES NOT COMPILE output wrapped in input***

**new BufferedInputStream(new InputStream()); *// DOES NOT COMPILE Input Stream is abstract***

# Summary

| **Class Name** | **Type** | **Description** |
| --- | --- | --- |
| InputStream | Abstract | Abstract class for all input byte streams |
| OutputStream | Abstract | Abstract class for all output byte streams |
| Reader | Abstract | Abstract class for all input character streams |
| Writer | Abstract | Abstract class for all output character streams |
| FileInputStream | Low | Reads file data as bytes |
| FileOutputStream | Low | Writes file data as bytes |
| FileReader | Low | Read file data as character |
| FileWriter | Low | Write file data as character |
| BufferedInputStream | High | Reads byte data from an existing InputStream in a buffered manner, which improves efficiency and performance |
| BufferedOutputStream | High | Writes byte data from an existing OutputStream in a buffered manner, which improves efficiency and performance |
| BufferedReader | High | Reads character data from an existing Reader in a buffered manner, which improves efficiency and performance |
| BufferedWriter | High | Writes character data to an existing Writer in a buffered manner, which  improves efficiency and performance |
| ObjectInputStream | High | Deserializes primitive Java data types and graphs of Java objects from an existing InputStream |
| ObjectOutputStream | HIgh | Serializes primitive Java data types and graphs of Java objects to an existing OutputStream |
| PrintStream | High | Writes formatted representations of Java objects to a binary stream |
| PrintWriter | High | Writes formatted representations of Java objects to a character stream |

# I/O Stream Methods

I/O Stream uses read and write method to read and write data to file

**public int read() throws IOException*// InputStream and Reader***

**public void write(int b) throws IOException *// OutputStream and Writer***

The int represents the bytecode value when it is -1 It represents the end of Stream.

# I/O Stream Operations

## Using FileInputStream and FileOutputStream

**File fileIn = new File("in.txt");**

**File fileOut = new File("out.txt");**

**try (var in = new FileInputStream(fileIn);**

**var out = new FileOutputStream(fileOut)) {**

**int b;**

**while ((b = in.read()) != -1) {**

**System.out.println(b);**

**out.write(b);**

**}**

**} catch (IOException e) {**

**e.printStackTrace();**

**}**

The FileOutputStream class includes overloaded constructors that take a boolean append flag. When set to true, the output stream will append to the end of a file if it already exists

**var out = new FileOutputStream(fileOut, true)**

If the source file does not exist, a ***FileNotFoundException***, which inherits IOException, will be thrown. If the destination file is not existing then it will be created otherwise if it already exists, this implementation will overwrite it, since the append flag was not sent. The copy() method copies one byte at a time until it reads a value of ‐1.

There are other overloaded implementation of read

**byte[] buffer = new byte[1024];**

**while ((b = in.read(buffer)) != -1) //data is read in buffer size provided for faster execution**

**while ((b = in.read(buffer, 2, 3)) != -1)// 2 bytes skipped and 3 bytes limit which is read**

Same for write methods

**byte[] buffer = new byte[1024];**

**int data;**

**while ((data = in.read()) != -1) {**

**out.write(data); *// byte data received during read. During read buffer should not be passed***

**while ((data = in.read(buffer)) != -1) {**

**out.write(buffer); *// buffer data received during read.***

**while ((data = in.read(buffer)) != -1) {**

**out.write(buffer, 0, data);**

Previous implementations perform poorly on large files. As discussed earlier, that's because there is a cost associated with each round‐trip to the file system. We can easily enhance our implementation using BufferedInputStream and BufferedOutputStream.