Table of Contents

1 Introduction

Java IO is one of the most important concept for day to day programming in java. To store small amount of data we use files, and it's better to user some database to store large amount of data.

The java.io package provides an extensive library of classes for dealing with input and output. Java provides streams as a general mechanism for dealing with data I/O. Streams implement sequential access of data.

There are two kinds of streams: **byte streams and character streams** (a.ka. binary streams and text streams, respectively).

Byte stream examples –

Video files
Audio files
Image files
PDF files
Doc files
Objects

Character stream examples –

Text files
Configuration files like json, properties file.

An input stream is an object that an application can use to read a sequence of data, and an output stream is an object that an application can use to write a sequence of data. An input stream acts as a source of data, and an output stream acts as a destination of data.

The following entities can act as both input and output streams:

- an array of bytes or characters
- a file
- a pipe (a mechanism by which a program can communicate data to another program during execution)
- a network connection
- System.in, System.out, System.error

Streams can be chained with filters to provide new functionality. In addition to dealing with bytes and characters, streams are provided for input and output of Java primitive values and objects. The java.io package also provides a general interface to interact with the file system of the host platform.

2 File

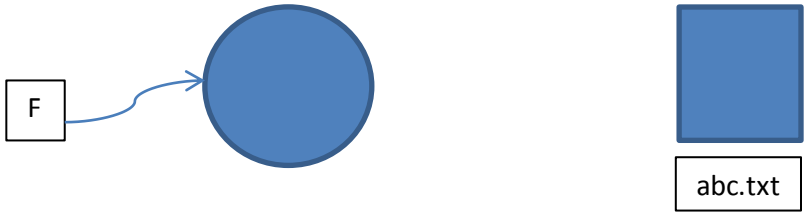
File is class useful to perform most common operation on file, like create a file, rename, etc.

File f = new File("abc.txt");

This line **won't create any physical files**, first it will check whether any physical file already available with the name of "abc.txt" or not.

If it is not already avalilable this line won't create any physical file and just it creates a java file object to represent the name abc.txt

File f = new File("abc.txt"); -------→ This line creates only **java file Object**
System.out.println(f.exists());
f.createNewFile(); ----------→ this line will create physical file
System.out.println(f.exists());



Output:

1st Run	2nd Run
False	True
True	True

We can use java file object to represent directory also.

File f = new File("abc.txt");
System.out.println(f.exists());
f.mkdir();----------→ this line will create a file
System.out.println(f.exists());

Note: Java file IO concept is implemented based on UNIX system and in UNIX everything is treat as file. Hence we can use java file object to represent both files and directories.

2.1 File Separators

Member variables in file class–
public static final char **separatorChar**
public static final String **separator**
Defines the character or string that separates the directory and the file components in a pathname. This separator is '/', '\' or ':' for Unix, Windows, and Macintosh, respectively.

public static final char **pathSeparatorChar**
public static final String **pathSeparator**

Defines the character or string that separates the file or directory names in a "path list." This character is ':' or ';' for Unix and Windows, respectively.

2.2 File class Constructors

Below are the constructors of File class.

1. **File f = new File(String name)**
   Creates a java file object to represent name of specified file or directory present in current working directory.

   "/book/chapter1" – absolute pathname of a file
   File chap1 = new File(File.separator + "book" + File.separator + "chapter1");

   "draft/chapters" – relative pathname of a directory
   File draftChapters = new File("draft" + File.separator + "chapters");

2. **File f = new File(String subdir, String name)**
   Creates a java file object to represent name of the file or directory present in specific directory.

   "/book/chapter1" – absolute pathname of a file
   File updatedChap1 = new File(File.separator + "book", "chapter1");

3. **File f = new File(String subdir, String name);**
   - Below is the code to create a file named with abc.txt in current working directory

     ```
     File f = new File(abc.txt);
     f.createNewFile();
     ```

   - Below is the code to create a directory nameed with "naren" in current working directory and under that create a file named with "abc.txt".

     ```
     File f = new File("naren");
     f.mkdir();
     File f1 = new File("naren", "abc.txt");
     File f2 = new File(f, "abc.txt");
     f.createNewFile();

     // "chapter13" – relative pathname of a file
     File parent = null;
     File chap13 = new File(parent, "chapter13");
     // "draft/chapters/chapter13" – relative pathname of a file
     File draftChapters = new File("draft" + File.separator + "chapters");
     File updatedChap13 = new File(draftChapters, "chapter13");
     ```

2.3 Querying the File System

The File class provides a number of methods for obtaining the platform-dependent representation of a pathname and its components.

String getName() - Returns the name of the file entry, excluding the specification of the directory in which it resides

On Unix, the name part of "/book/chapters/one" is "one".
On Windows platforms, the name part of "c:\java\bin\javac" is "javac".
On the Macintosh, the name part of "HD:java-tools:javac" is "javac".

The strings "." and ".." generally designate the current directory and the parent directory in pathnames, respectively.

String getPath() –
The method returns the (absolute or relative) pathname of the file represented by the File object.

String getAbsolutePath() -
If the File object represents an absolute pathname, this pathname is returned, otherwise the returned pathname is constructed by concatenating the current directory pathname, the separator character and the pathname of the File object.

String getCanonicalPath() throws IOException
Also platform-dependent, the canonical path usually specifies an absolute pathname in which all relative references have been completely resolved.

For example, if the File object represented the absolute pathname "c:\book\chapter1" on Windows, this pathname would be returned by these methods.

On the other hand, if the File object represented the relative pathname "..\book\chapter1" and the current directory had the absolute pathname "c:\documents",
the pathname returned by the getPath(), getAbsolutePath(), and getCanonicalPath() methods would be

"..\book\chapter1",
"c:\documents\..\book\chapter1" and
"c:\book\chapter1", respectively.

String getParent()
The parent part of the pathname of this File object is returned if one exists, otherwise the null value is returned.
On Unix, the parent part of "/book/chapter1" is "/book", whose parent part is "/", which in turn has no parent.
On Windows platforms, the parent part of "c:\java-tools" is "c:\", which in turn has no parent.

boolean isAbsolute()
Whether a File object represents an absolute pathname can be determined using this method.

long lastModified()
The modification time returned is encoded as a long value, and should only be compared with other values returned by this method.

long length()
Returns the size (in bytes) of the file represented by the File object.

boolean equals(Object obj)
This method just compares the pathnames of the File objects, and returns true if they are identical. On Unix systems, alphabetic case is significant in comparing pathnames; on Windows systems it is not.

2.4 File or Directory Existence

boolean exists() - Returns true if specified physical file or directory exists in the system
boolean isFile() - return true if the file Object pointing to physical file.
boolean isDirectory() - return true if the file Object pointing to physical Directory.

2.5 File and Directory Permissions

Write, read and execute permissions can be set by calling the following methods. If the first argument is true, the operation permission is set; otherwise it is cleared. If the second argument is true, the permission only affects the owner; otherwise it affects all users.

These methods throw a SecurityException if permission cannot be changed. It should be noted that the exact interpretation of these permissions is platform dependent. To check whether the specified file has write, read, or execute permissions, the following methods can be used.

boolean setReadable(boolean readable)
boolean setReadable(boolean readable, boolean owner)
boolean setWritable(boolean writable)
boolean setWritable(boolean writable, boolean owner)
boolean setExecutable(boolean executable)
boolean setExecutable(boolean executable, boolean owner)

To check whether the specified file has write, read, or execute permissions, the following methods can be used.

They throw a SecurityException if general access is not allowed, i.e., the application is not even allowed to check whether it can read, write or execute a file.

boolean canWrite()
boolean canRead()
boolean canExecute()

2.6 Listing Directory Entries

The entries in a specified directory can be obtained as an array of file names or abstract pathnames using the following list() methods. The current directory and the parent directory are excluded from the list.

```
String[] list()
String[] list(FilenameFilter filter)

File[] listFiles()
File[] listFiles(FilenameFilter filter)
File[] listFiles(FileFilter filter)
```

These methods **return null if the abstract pathname does not denote a directory**, or if an I/O error occurs.
The filter argument can be used to specify a filter that determines whether an entry should be included in the list.
A filter is an object of a class that implements either of these two interfaces:

```
interface FilenameFilter {
        boolean accept(File currentDirectory, String entryName);
}
interface FileFilter {
        boolean accept(File pathname);
}
```

The list() methods call the accept() methods of the filter for each entry to determine whether the entry should be included in the list.

2.7 Creating New Files and Directories

The File class can be used to create files and directories. A file can be created whose pathname is specified in a File object using the following method:

boolean createNewFile() throws IOException

It creates a new, empty file named by the abstract pathname if, and only if, a file with this name does not already exist.
The returned value is **true if the file was successfully created**, ***false if the file already exists***. Any I/O error results in an IOException.

```
boolean mkdir()
boolean mkdirs()
```

The mkdirs() method creates any intervening parent directories in the pathname of the directory to be created.

2.8 Renaming Files and Directories

A file or a directory can be renamed, using the following method which takes the new pathname from its argument. **It throws a SecurityException if access is denied.**

```
boolean renameTo(File dest)
```

2.9 Deleting Files and Directories

A file or a directory can be deleted using the following method. In the case of a directory, it must be empty before it can be deleted. It throws a SecurityException if access is denied.

boolean delete()

2.10 Important Methods of File class

1. boolean exists() - Returns true if specified physical file or directroy exists in the system

2. boolean createNewFile() - First this method will check whether the physical file already available or not.

If already available then this method returns false without creating any physical file.

if it is not already available then this method creates new file and returns true

3. boolean mkdir()

4. boolean isFile() - return true if the file Object pointing to physical file.

5. boolean isDirectory() - return true if the file Object pointing to physical Directory.

6. String[] list() - returns the names of all the files and directories present in specified directroy

7. long length() - returns the number of characters present in the specified file.

8. boolean delete() - to delete specified file or directroy

3 Java IO Class Overview

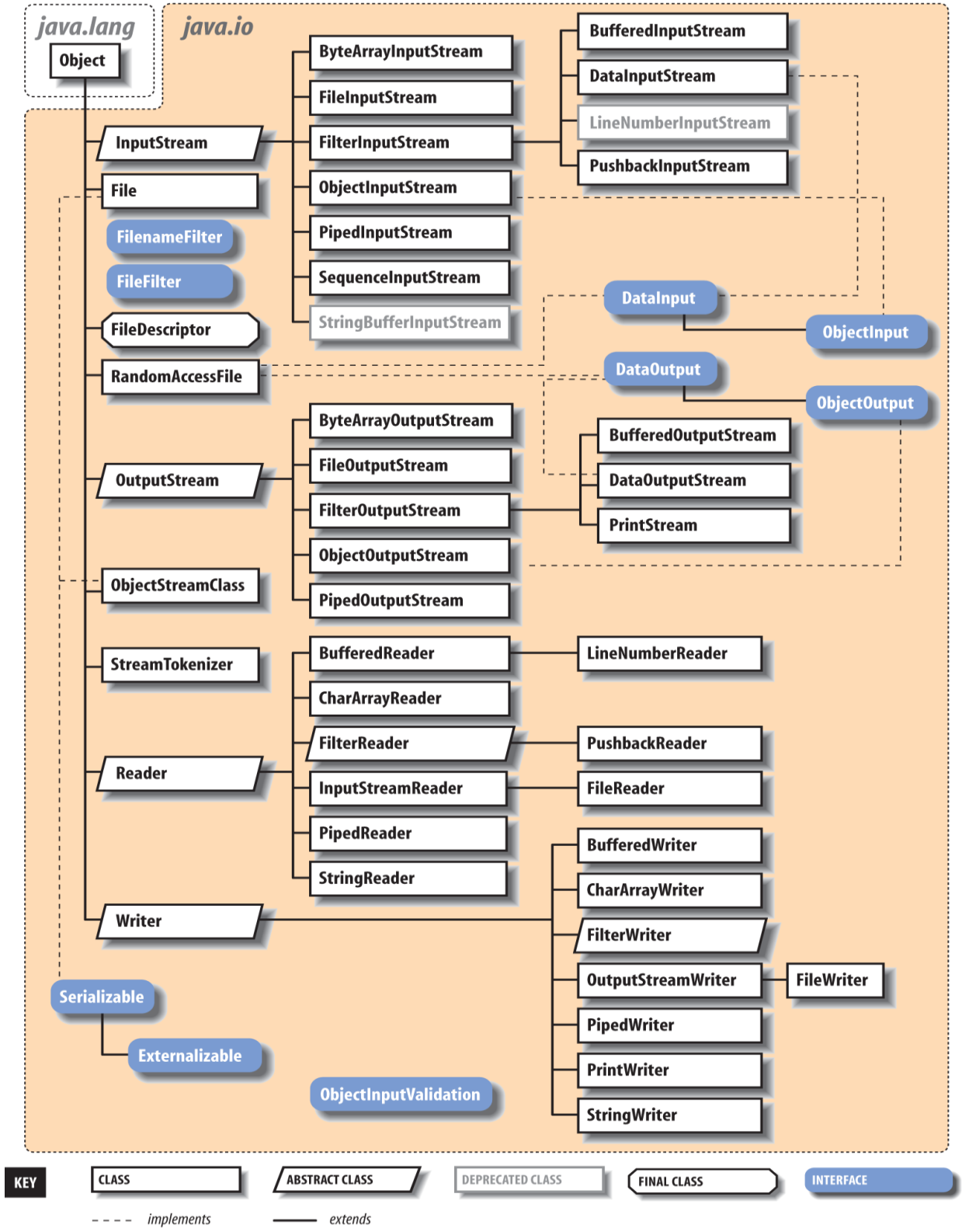
Java IO contains many subclasses of the InputStream, OutputStream, Reader and Writer classes. The reason is, that all of these subclasses are addressing various different purposes. That is why there are so many different classes. The purposes addressed are summarized below:

- File Access
- Network Access
- Internal Memory Buffer Access
- Inter-Thread Communication (Pipes)
- Buffering
- Filtering
- Parsing
- Reading and Writing Text (Readers / Writers)
- Reading and Writing Primitive Data (long, int etc.)
- Reading and Writing Objects

Here is a table listing most (if not all) Java IO classes divided by input, output, being byte based or character based, and any more specific purpose they may be addressing, like buffering, parsing etc.

Purpose	Byte Based		Character Based	
	Input	***Output***	***Input***	***Output***
Basic	InputStream	OutputStream	Reader InputStreamReader	Writer OutputStreamWriter
Arrays	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
Files	FileInputStream RandomAccessFile	FileOutputStream RandomAccessFile	FileReader	FileWriter
Pipes	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
Buffering	BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
Filtering	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
Parsing	PushbackInputStream StreamTokenizer		PushbackReader LineNumberReader	
Strings			StringReader	StringWriter
Data	DataInputStream	DataOutputStream		
Data - Formatted		PrintStream		PrintWriter
Objects	ObjectInputStream	ObjectOutputStream		
Utilities	SequenceInputStream			

3.1 Class diagram of IO Streams



4 Byte Streams

Byte streams are basically for reading and writing binary files, like video, audio, document, image files. However it can be used for text file as it a subset of binary files. But to read/write text/character files better use character streams as they use character encoding for which makes better character conversion.

- The abstract classes InputStream and OutputStream are the root of the inheritance hierarchies for handling the reading and writing of bytes.
- Their subclasses, implementing different kinds of input and output streams. override methods from the InputStream and OutputStream classes to customize the reading and writing of bytes.
- Read and write operations on streams are synchronous (blocking) operations, i.e., a call to a read or write method does not return before a byte has been read or written.
- Many methods in the classes contained in the java.io package throw the checked IOException.

4.1 Basic Streams

4.1.1 InputStream Class

An InputStream is typically always connected to some data source, like a file, network connection, pipe etc.
The InputStream class has following methods, As its an abstract class instance of this class usually created by its child classes

```
int read() throws IOException
int read(byte[] b) throws IOException
int read(byte[] b, int off, int len) throws IOException
void close() throws IOException
```

- Note that the first read() method reads a byte, but returns an int value.
- The byte read resides in the eight least significant bits of the int value, while the remaining bits in the int value are zeroed out.
- The read() methods return the value –1 when the end of the stream is reached.

Ex –

```
InputStream inputstream = new FileInputStream("c:\\data\\input-text.txt");
int data = inputstream.read();
while(data != -1) {
        //do something with data...
        doSomethingWithData(data);
        data = inputstream.read();
}
inputstream.close();
```

read()

The read() method of an InputStream returns an int which contains the byte value of the byte read. Here is an InputStream read() example:
You can case the returned int to a char like this:

```
int data = inputstream.read();
```

char aChar = (char) data;

Subclasses of InputStream may have alternative read() methods. For instance, the DataInputStream allows you to read Java primitives like int, long, float, double, boolean etc. with its corresponding methods readBoolean(), readDouble() etc.

End of Stream

If the read() method returns -1, the end of stream has been reached, meaning there is no more data to read in the InputStream.
That is, -1 as int value, not -1 as byte or short value. There is a difference here!
When the end of stream has been reached, you can close the InputStream.

read(byte[])

The InputStream class also contains two read() methods which can read data from the InputStream's source into a byte array. These methods are:
Reading an array of bytes at a time is much faster than reading one byte at a time, so when you can, use these read methods instead of the read() method.

int read(byte[])
int read(byte[], int offset, int length)

The read(byte[]) method will attempt to read as many bytes into the byte array given as parameter as the array has space for. The read(byte[]) method returns an int telling how many bytes were actually read. In case less bytes could be read from the InputStream than the byte array has space for, the rest of the byte array will contain the same data as it did before the read started. Remember to inspect the returned int to see how many bytes were actually read into the byte array.
The read(byte[], int offset, int length) method also reads bytes into a byte array, but starts at offset bytes into the array, and reads a maximum of length bytes into the array from that position. Again, the read(byte[], int offset, int length) method returns an int telling how many bytes were actually read into the array, so remember to check this value before processing the read bytes.

For both methods, if the end of stream has been reached, the method returns -1 as the number of bytes read.

Ex to use the InputStream's read(byte[]) method:

```
InputStream inputstream = new FileInputStream("c:\\data\\input-text.txt");
byte[] data     = new byte[1024];
int    bytesRead = inputstream.read(data);
while(bytesRead != -1) {
        doSomethingWithData(data, bytesRead);
        bytesRead = inputstream.read(data);
}
inputstream.close();
```

First this example creates a byte array. Then it creates an int variable named bytesRead to hold the number of bytes read for each read(byte[]) call, and immediately assigns bytesRead the value returned from the first read(byte[]) call.

Inside the while loop the doSomethingWithData() method is called, passing along the data byte array as well as how many bytes were read into the array as parameters. At the end of the while loop data is read into the byte array again.
It should not take much imagination to figure out how to use the **read(byte[], int offset, int length)** method instead of read(byte[]). You pretty much just replace the read(byte[]) calls with read(byte[], int offset, int length) calls.

mark() and reset()

The InputStream class has two methods called mark() and reset() which subclasses of InputStream may or may not support.

If an InputStream subclass supports the mark() and reset() methods, then that subclass should override the markSupported() to return true.
If the markSupported() method returns false then mark() and reset() are not supported.

The mark() sets a mark internally in the InputStream which marks the point in the stream to which data has been read so far.
The code using the InputStream can then continue reading data from it.
If the code using the InputStream wants to go back to the point in the stream where the mark was set, the code calls reset() on the InputStream.
The InputStream then "rewinds" and go back to the mark, and start returning (reading) data fro m that point again.
This will of course result in some data being returned more than once from the InputStream.
The methods mark() and reset() methods are typically used when implementing parsers.
Sometimes a parser may need to read ahead in the InputStream and if the parser doesn't find what it expected, it may need to rewind back and try to match the read data against something else.

4.1.2 OutputStream Class

An OutputStream is typically always connected to some data destination, like a file, network connection, pipe etc.
The OutputStream class has following methods, As its an abstract class instance of this class usually created by its child classes

void write(int b) throws IOException
void write(byte[] b) throws IOException
void write(byte[] b, int off, int len) throws IOException
void close() throws IOException
void flush() throws IOException Only for OutputStream

- The first write() method takes an int as argument, but truncates it down to the eight least significant bits before writing it out as a byte.
- A stream should be closed when no longer needed, to free system resources.
- Closing an output stream automatically flushes the stream, meaning that any data in its internal buffer is written out.
- An output stream can also be manually flushed by calling the second method.

write(byte)

The write(byte) method is used to write a single byte to the OutputStream. The write() method of an OutputStream takes an int which contains the byte value of the byte to write. Only the first byte of the int value is written. The rest is ignored.

Subclasses of OutputStream may have alternative write() methods. For instance, the DataOutputStream allows you to write Java primitives like int, long, float, double, boolean etc. with its corresponding methods writeBoolean(), writeDouble() etc.

Here is an OutputStream write() example:

```
OutputStream output = new FileOutputStream("c:\\data\\output-text.txt");
while(hasMoreData()) {
        int data = getMoreData();
         output.write(data);
}
output.close();
```

This OutputStream write() example first creates a FileOutputStream to which the data will be written. Then the example enters a while loop. The condition to exit the while loop is the return value of the method hasMoreData(). The implementation of hasMoreData() is not shown, but imagine that it returns true if there is more data to write, and false if not.
Inside the while loop the example calls the method getMoreData() to get the next data to write to the OutputStream, and then writes that data to the OutputStream. The while loop continues until hasMoreData() returns false.

write(byte[])

The OutputStream class also has a write(byte[] bytes) method and a write(byte[] bytes, int offset, int length) which both can write an array or part of an array of bytes to the OutputStream.

The write(byte[] bytes) method writes all the bytes in the byte array to the OutputStream.

The write(byte[] bytes, int offset, int length) method writes length bytes starting from index offset from the byte array to the OutputStream.

flush()

The OutputStream's flush() method flushes all data written to the OutputStream to the underlying data destination. For instance, if the OutputStream is a FileOutputStream then bytes written to the FileOutputStream may not have been fully written to disk yet. The data might be buffered in memory somewhere, even if your Java code has written it to the FileOutputStream. By calling flush() you can assure that any buffered data will be flushed (written) to disk (or network, or whatever else the destination of your OutputStream has).

close()

Once you are done writing data to the OutputStream you should close it. You close an OutputStream by calling its close() method. Since the OutputStream's various write() methods may throw an IOException, you should close the OutputStream inside a finally block. Here is a simple OutputStream close() example:

```
OutputStream output = null;
try{
        output = new FileOutputStream("c:\\data\\output-text.txt");
        while(hasMoreData()) {
```

```
                int data = getMoreData();
                output.write(data);
            }
        } finally {
            if(output != null) {
                output.close();
            }
        }
```

This simple example calls the OutputStream close() method inside a finally block. While this makes sure that the OutputStream is closed.

4.2 File Streams

4.2.1 FileInputStream Class

The FileInputStream class makes it possible to read the contents of a file as a stream of bytes. The FileInputStream class is a subclass of InputStream. This means that you use the FileInputStream as an InputStream

FileInputStream Example :

```
InputStream input = new FileInputStream("c:\\data\\input-text.txt");
int data = input.read();
while(data != -1) {
  //do something with data...
  doSomethingWithData(data);
  data = input.read();
}
input.close();
```

FileInputStream Constructors

The FileInputStream class has a three different constructors you can use to create a FileInputStream instance.

1. The first constructor takes a String as parameter. This String should contain the path in the file system to where the file to read is located. Here is a code example:

   ```
   String path = "C:\\user\\data\\thefile.txt";
   FileInputStream fileInputStream = new FileInputStream(path);
   ```

2. The second FileInputStream constructor takes a File object as parameter. The File object has to point to the file you want to read. Here is an example:

   ```
   String path = "C:\\user\\data\\thefile.txt";
   File   file = new File(path);
   FileInputStream fileInputStream = new FileInputStream(file);
   ```

Which of the constructors you should use depends on what form you have the path in before opening the FileInputStream. If you already have a String or File, just use that as it is. There is no particular gain in converting a String to a File, or a File to a String first.

Below methods are same as InputStream

read()
read(byte[])
close()

4.2.2 FileOutputStream Class

The FileOutputStream class makes it possible to write a file as a stream of bytes. The FileOutputStream class is a subclass of OutputStream
The FileOutputStream class makes it possible to write a file as a stream of bytes. The FileOutputStream class is a subclass of OutputStream meaning you can use a FileOutputStream as an OutputStream.

FileOutputStream Example:-

```
OutputStream output = new FileOutputStream("c:\\data\\output-text.txt");
while(moreData){
        int data = getMoreData();
        output.write(data);
}
output.close();
```

FileOutputStream Constructors

1. The FileOutputStream class contains a set of different useful constructors. I will cover the most commonly used constructors here.
   The first constructor takes a String which contains the path of the file to write to.
   Here is an example:

```
String path = "C:\\users\\jakobjenkov\\data\\datafile.txt";
FileOutputStream output = new FileOutputStream(path);
```

2. The second FileOutputStream constructor takes a File object which points to the file in the file system. Here is an example:

```
String path = "C:\\users\\jakobjenkov\\data\\datafile.txt";
File   file = new File(path);
FileOutputStream output = new FileOutputStream(file);
```

Overwriting vs. Appending the File

When you create a FileOutputStream pointing to a file that already exists, you can decide if you want to overwrite the existing file, or if you want to append to the existing file. You decide that based on which of the FileOutputStream constructors you choose to use.
This constructor which takes just one parameter, the file name, will overwrite any existing file:

```
OutputStream output = new FileOutputStream("c:\\data\\output-text.txt");
```

3. There is a constructor that takes 2 parameters too: The file name and a boolean. The boolean indicates whether to append or overwrite an existing file. Here are two examples:

   OutputStream output = new FileOutputStream("c:\\output-text.txt", true); //appends to file
   OutputStream output = new FileOutputStream("c:\\output-text.txt", false); //overwrites file

flush()
When you write data to a FileOutputStream **the data may get cached internally in the memory** of the computer and written to disk at a later time. For instance, every time there is X amount of data to write, or when the FileOutputStream is closed.
If you want to **make sure that all written data is written to disk without having to close the FileOutputStream you can call its flush() method**. Calling flush() will make sure that all data which has been written to the FileOutputStream so far, is fully written to disk too.

Below methods same as outputStream :
close()
write()

4.3 Filtering Streams

A filter is a high-level stream that provides additional functionality to an underlying stream to which it is chained. The data from the underlying stream is manipulated in some way by the filter.

The FilterInputStream and FilterOutputStream classes, together with their subclasses, define input and output filter streams.

- **The subclasses BufferedInputStream and BufferedOutputStream implement filters that buffer** input from and output to the underlying stream, respectively.
- **The subclasses DataInputStream and DataOutputStream implement filters that allow binary representation of Java primitive values to be read and written**, respectively, to and from an underlying stream.

4.3.1 FilterInputStream Class

The FilterInputStream is a base class for implementing your own filtering input streams.

Basically **it just overrides all methods in InputStream and passes all calls to any method on the FilterInputStream** onto a wrapped InputStream.

The wrapped InputStream is passed to the FilterInputStream in its constructor, like this:

```
FilterInputStream inputStream = new FilterInputStream(new FileInputStream("c:\\myfile.txt"));
```

The FilterInputStream class does not have any special behaviour. It is intended to be a base class for your own subclasses, but in my opinion you might as well just subclass InputStream directly.

I think no sensible purpose for this class. I cannot see that this class actually adds or changes any behaviour in InputStream except that it takes an InputStream in its constructor.

4.3.2 FilterOutputStream Class

The FilterOutputStream is a base class for implementing your own filtering output streams. Basically it just overrides all methods in OutputStream.

I see no sensible purpose for this class. I cannot see that this class actually adds or changes any behaviour in OutputStream except that it takes an OutputStream in its constructor. If you choose to extend this class you might as well extend the OutputStream class directly, and avoid the extra class in the hierarchy

4.4 Buffering Streams

4.4.1 BufferedInputStream Class

The BufferedInputStream class provides buffering to your input streams. Buffering **can speed up IO** quite a bit. Rather than read one byte at a time from the network or disk, the BufferedInputStream **reads a larger block at a time into an internal buffer**.

When you read a byte from the BufferedInputStream you are therefore reading it from its internal buffer of BufferedInputStream.

When the buffer is fully read, the BufferedInputStream reads another larger block of data into the buffer. This is typically much faster than reading a single byte at a time from an InputStream, especially for disk access and larger data amounts.

BufferedInputStream Example

```
InputStream input = new BufferedInputStream(new FileInputStream("c:\\data\\input-file.txt"));
```

The BufferedInputStream creates a byte array internally, and attempts to fill the array by calling the InputStream.read(byte[]) methods on the underlying InputStream.

Setting Buffer Size of a BufferedInputStream

You can set the buffer size to use internally by the BufferedInputStream. You provide the buffer size as a parameter to the BufferedInputStream constructor, like this:

```
int bufferSize = 8 * 1024;
InputStream input = new BufferedInputStream(new FileInputStream("c:\\data\\input-file.txt"),
bufferSize);
```

This example sets the internal buffer used by the BufferedInputStream to 8 KB.

It is best to use buffer sizes that are multiples of 1024 bytes. That works best with most built-in buffering in hard disks etc.

Optimal Buffer Size for a BufferedInputStream

You should make some experiments with different buffer sizes to find out which buffer size seems to give you the best performance on your concrete hardware.

The optimal buffer size may depend on whether you are using the BufferedInputStream with a disk or network InputStream.

With both disk and network streams, the optimal buffer size may also depend on the concrete hardware in the computer.

If the hard disk is anyways reading a minimum of 4KB at a time, it's stupid to use less than a 4KB buffer.

It is also better to then use a buffer size that is a multiple of 4KB. For instance, using 6KB would be stupid too. Even if your disk reads blocks of e.g. 4KB at a time, it can still be a good idea to use a buffer that is larger than this.

A disk is good at reading data sequentially - meaning it is good at reading multiple blocks that are located after each other. Thus, using a 16KB buffer, or a 64KB buffer (or even larger) with a BufferedInputStream may still give you a better performance than using just a 4KB buffer.

Also keep in mind that some hard disks have a read cache of some megabytes. If your hard disk anyways reads, say 64KB, of your file into its internal cache, you might as well get all of that data into your BufferedInputStream using one read operation, instead of using multiple read operations.

Multiple read operations will be slower, and you risk that the hard disk's read cache gets erased between read operations, causing the hard disk to re-read that block into the cache.

To find the optimal BufferedInputStream buffer size, find out the block size your hard disk reads in, and possibly also its cache size, and make the buffer a multiple of that size.

You will definitely have to experiment to find the optimal buffer size. Do so by measuring read speeds with different buffer sizes.

mark() and reset()

An interesting aspect to note about the BufferedInputStream is that is supports the mark() and reset() methods inherited from the InputStream.

Not all InputStream subclasses support these methods. In general you can call the markSupported() method to find out if mark() and reset() are supported on a given InputStream or not.

4.4.2 BufferedOutputStream Class

The BufferedOutputStream class provides buffering to your output streams. Buffering can speed up IO quite a bit. Rather than write one byte at a time to the network or disk, you write a larger block at a time. This is typically much faster, especially for disk access and larger data amounts.

To add buffering to your OutputStream's simply wrap them in a BufferedOutputStream. Here is how that looks:

```
OutputStream output = new BufferedOutputStream(
                                        new FileOutputStream("c:\\data\\output-file.txt"));
```

Setting Buffer Size of a BufferedOutputStream

```
int bufferSize = 8 * 1024;
OutputStream output = new BufferedOutputStream(
        new FileOutputStream("c:\\data\\output-file.txt"), bufferSize);
```

4.5 Array Streams

Byte and char arrays are often used in Java to temporarily store data internally in an application. As such arrays are also a common source or destination of data.

You may also prefer to load a file into an array, if you need to access the contents of that file a lot while the program is running. And you can access these arrays directly by indexing into them.

But what if you have a component that is designed to read some specific data from an InputStream and not an array?

4.5.1 ByteArrayInputStream

Reading Arrays via InputStream –

To make such a component read from the data from an array, you will have to wrap the byte or char array in anByteArrayInputStream or CharArrayReader. This way the bytes or chars available in the array can be read through the wrapping stream or reader.

Here is a simple example:

```
byte[] bytes = new byte[1024];
//write data into byte array...
InputStream input = new ByteArrayInputStream(bytes);
//read first byte
int data = input.read();
while(data != -1) {
//do something with data
        //read next byte
        data = input.read();
}
```

4.5.2 ByteArrayOutputStream

Writing to Arrays via OutputStream -

It is also possible to write data to an ByteArrayOutputStream or CharArrayWriter. All you have to do is to create either a ByteArrayOutputStream or CharArrayWriter, and write your data to it, as you would to any other stream or writer.

Once all the data is written to it, simply call the method toByteArray() ortoCharArray, and all the data written is returned in array form.

Here is a simple example:

```
ByteArrayOutputStream output = new ByteArrayOutputStream();
output.write("This text is converted to bytes".getBytes("UTF-8"));
byte[] bytes = output.toByteArray();
```

4.6 Data Streams

4.6.1 DataInput and DataOutput Interfaces

Reading and Writing Binary Values : -

The java.io package **contains the two interfaces DataInput and DataOutput**, that streams can implement to allow reading and writing of binary representations of Java primitive values (boolean, char, byte, short, int, long, float, double).

The filter streams DataOutputStream and DataInputStream implement DataOutput and DataInput interfaces, respectively, and can be used to read and write binary representations of Java primitive values to and from an underlying stream

- The methods for writing binary representations of Java primitive values are named **writeX**, where X is any Java primitive data type.
- The methods for reading binary representations of Java primitive values are similarly named readX.

A file containing binary values (i.e., binary representation of Java primitive vales) is usually called a binary file.

Note the methods provided for reading and writing strings. Whereas the methods readChar() and writeChar() handle a single character, the methods **readLine() and writeChars() handle a string** of characters.

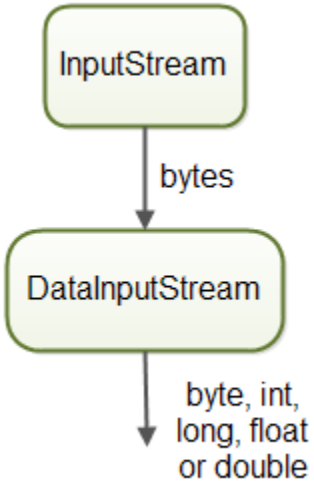
The methods readUTF() and writeUTF() also read and write characters, but use the UTF-8 character encoding.

4.6.2 DataInputStream

The Java DataInputStream class enables you to read Java primitives (int, float, long etc.) from an InputStream instead of only raw bytes.

You wrap an InputStream in a DataInputStream and then you can read Java primitives from the DataInputStream.

That is **why it is called DataInputStream - because it reads data (numbers) instead of just bytes**. Data – Formatted Streams



The DataInputStream is handy if the data you need to read consists of Java primitives larger than one byte each, like int, long, float, double etc.

DataInputStream Example :

```
DataInputStream dataInputStream = new DataInputStream(
                    new FileInputStream("binary.data"));
int    aByte   = input.read();
int    anInt   = input.readInt();
float  aFloat  = input.readFloat();
double aDouble = input.readDouble();
//etc.
nput.close();
```

First a DataInputStream is created with a FileInputStream as source for its data. Second, Java primitives are read from the DataInputStream.

Using a DataInputStream With a DataOutputStream

As mentioned earlier, the DataInputStream class is often used together with a DataOutputStream.

Therefore I just want to show you an example of first writing data with a DataOutputStream and then reading it again with a DataInputStream.

Here is the example Java code:

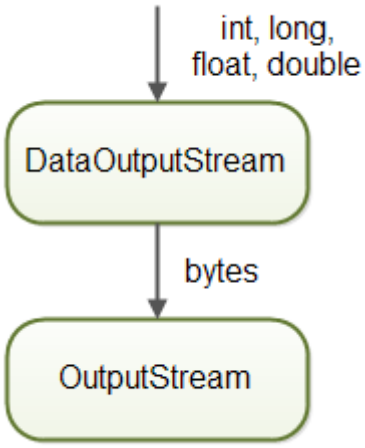
```
import java.io.*;
public class DataInputStreamExample {
        public static void main(String[] args) throws IOException {
            DataOutputStream dataOutputStream = new DataOutputStream(
                    new FileOutputStream("data/data.bin"));
            dataOutputStream.writeInt(123);
            dataOutputStream.writeFloat(123.45F);
            dataOutputStream.writeLong(789);
            dataOutputStream.close();
            DataInputStream dataInputStream = new DataInputStream(
                    new FileInputStream("data/data.bin"));
            int  int123    = dataInputStream.readInt();
            float float12345 = dataInputStream.readFloat();
            long  long789   = dataInputStream.readLong();

            dataInputStream.close();
            System.out.println("int123    = " + int123);
            System.out.println("float12345 = " + float12345);
            System.out.println("long789    = " + long789);
          }
        }
```

This example first creates a DataOutputStream and then writes an int, float and a long value to a file. Second the example creates a DataInputStream which reads the int, float and long value i n from the same file.

4.6.3 DataOutputStream

The Java DataOutputStream class enables you to write Java primitives to OutputStream's instead of only bytes. You wrap an OutputStream in a DataOutputStream and then you can write primitives to it.

That is why it is called a DataOutputStream - because you can write int, long, float and double values to the OutputStream, and not just raw byte



Here is a Java DataOutputStream example:

```
DataOutputStream dataOutputStream = new DataOutputStream(
                    new FileOutputStream("binary.data"));
dataOutputStream.write(45);          //byte data
dataOutputStream.writeInt(4545);     //int data
dataOutputStream.writeDouble(109.123); //double data
dataOutputStream.close();
```

4.7 Data – Formatted Streams

4.7.1 PrinrStream Class

The Java PrintStream class (java.io.PrintStream) enables you to write formatted data to an underlying OutputStream. The PrintStream class can format primitive types like int, long etc. formatted as text, rather than as their byte values. That is why it is called a PrintStream, because it formats the primitive values as text - like they would look when printed to the screen (or printed to paper).

Here is a simple Java PrintStream example:

```
PrintStream printStream = new PrintStream(outputStream);
printStream.print(true);
printStream.print((int) 123);
printStream.print((float) 123.456);
printStream.close();
```

This example first creates a PrintStream which is connected to an OutputStream. Second, the example prints three primitivate values to the PrintStream. Third, the example closes the PrintStream.

System.out and System.err are PrintStreams

You may be familiar with these two well-known PrintStream instances in Java: System.out and System.err . If you have every used any of these two streams, you have already used a PrintStream.

printf()

The Java PrintStream class contains the powerful format() and printf() methods (they do exactly the same, but the name "printf" is more familiar to C-programmers). These methods allow you to mix text and data in very advanced ways, using a formatting string.

Here is a simple Java printf() example:

```
PrintStream printStream = new PrintStream(outputStream);
printStream.printf(Locale.UK, "Text + data: %1$", 123);
printStream.close();
```

4.8 Objects Streams

4.8.1 ObjectInputStream class

The Java ObjectInputStream class (java.io.ObjectInputStream) enables you to read Java objects from an InputStream instead of just raw bytes. You wrap an InputStream in a ObjectInputStream and then you can read objects from it. Of course the bytes read must represent a valid, serialized Java object. Otherwise reading objects will fail.

ObjectInputStream Example

```
ObjectInputStream objectInputStream =
                        new ObjectInputStream(new FileInputStream("object.data"));
MyClass object = (MyClass) objectInputStream.readObject();
//etc.
objectInputStream.close();
```

For this ObjectInputStream example to work the object you read must be an instance of MyClass, and must have been serialized into the file "object.data" via an ObjectOutputStream. Before you can serialize and de-serialize objects the class of the object must implement java.io.Serializable.

Ex -

```
import java.io.*;
public class ObjectInputStreamExample {
   public static class Person implements Serializable {
      public String name = null;
      public int   age  =  0;
```

```
    }
        public static void main(String[] args) throws IOException,
ClassNotFoundException {
    ObjectOutputStream objectOutputStream =
      new ObjectOutputStream(new FileOutputStream("data/person.bin"));
    Person person = new Person();
    person.name = "Jakob Jenkov";
    person.age  = 40;

    objectOutputStream.writeObject(person);
    objectOutputStream.close();

    ObjectInputStream objectInputStream =
      new ObjectInputStream(new FileInputStream("data/person.bin"));
    Person personRead = (Person) objectInputStream.readObject();
    objectInputStream.close();

    System.out.println(personRead.name);
    System.out.println(personRead.age);
  }
}
```

4.8.2 ObjectOutputStream class

The Java ObjectOutputStream class (java.io.ObjectOutputStream) enables you to write Java objects to an OutputStream instead of just raw bytes. You wrap an OutputStream in a ObjectOutputStream and then you can write objects to it.

ObjectOutputStream Example

```
ObjectOutputStream objectOutputStream =
        new ObjectOutputStream(new FileOutputStream("object.data"));
MyClass object = new MyClass();
output.writeObject(object);
output.close();
```

Before you can serialize and de-serialize objects the class of the object must implement java.io.Serializable.

4.9 Pipes Streams

Pipes in Java IO provides the ability for two threads running in the same JVM to communicate. Therefore pipes can also be sources or destinations of data.

You cannot use a pipe to communicate with a thread in a different JVM (different process). The pipe concept in Java is different from the pipe concept in Unix / Linux, where two processes running in different address spaces can communicate via a pipe. In Java, the communicating parties must be running in the same process, and should be different threads.

Creating Pipes via Java IO

Creating a pipe using Java IO is done via the PipedOutputStream and PipedInputStream classes. A PipedInputStream should be connected to a PipedOutputStream. The data written to the PipedOutputStream by one thread can be read from the connected PipedInputStream by another thread.

Pipes and Threads

Remember, when using the two connected pipe streams, pass one stream to one thread, and the other stream to another thread. The read() and write() calls on the streams are blocking, meaning if you try to use the same thread to both read and write, this may result in the thread deadlocking itself.

4.9.1 PipedInputStream

The PipedInputStream class makes it possible to read the contents of a pipe as a stream of bytes. Pipes are communication channels between threads inside the same JVM.

```
import java.io.IOException;
import java.io.PipedInputStream;
import java.io.PipedOutputStream;

public class PipeExample {

  public static void main(String[] args) throws IOException {
    final PipedOutputStream output = new PipedOutputStream();
    final PipedInputStream  input  = new PipedInputStream(output);

    Thread thread1 = new Thread(new Runnable() {
      @Override
      public void run() {
        try {
          output.write("Hello world, pipe!".getBytes());
        } catch (IOException e) {
        }
      }
    });
    Thread thread2 = new Thread(new Runnable() {
      @Override
      public void run() {
```

```
            try {
                int data = input.read();
                while(data != -1){
                    System.out.print((char) data);
                    data = input.read();
                }
            } catch (IOException e) {
            }
        }
    });
    thread1.start();
    thread2.start();
  }
}
```

4.9.2 PipedOutputStream

The PipedOutputStream class makes it possible to write to a Java pipe as a stream of bytes. Pipes are communication between threads running in the same JVM.

4.10 Parsing Streams

4.10.1 PushbackInputStream

The PushbackInputStream is intended to be used when you parse data from an InputStream. Sometimes you need to read ahead a few bytes to see what is coming, before you can determine how to interpret the current byte. The PushbackInputStream allows you to do that. Well, actually it allows you to push the read bytes back into the stream. These bytes will then be read again the next time you call read().

PushbackInputStream Example

```
PushbackInputStream input = new PushbackInputStream(
                        new FileInputStream("c:\\data\\input.txt"));
int data = input.read();
input.unread(data);
```

The call to read() reads a byte just like from an InputStream. The call to unread() pushes a byte back into the PushbackInputStream. The next time read() is called the pushed back bytes will be read first. If you push back multiple bytes into the PushbackInputStream, the latest byte pushed back will be returned first from read(), just like on a stack.

Setting the Push Back Limit of a PushbackInputStream

You can set the number of bytes you should be able to unread in the constructor of the PushbackInputStream. Here is how to set the push back limit via the PushbackInputStream constructor:

```
int pushbackLimit = 8;
```

```
PushbackInputStream input = new PushbackInputStream(
        new FileInputStream("c:\\data\\input.txt"),pushbackLimit);
```

This example sets an internal buffer of 8 bytes. That means you can unread at most 8 bytes at a time, before reading them again.

4.10.2 StreamTokenizer

The Java StreamTokenizer class (java.io.StreamTokenizer) can tokenize the characters read from a Reader into tokens.

For instance, in the string "Mary had a little lamb" each word is a separate token.

When you are parsing files or computer languages it is normal to break the input into tokens, before further processing them.

This process is also called "lexing" or "tokenizing".

Using a Java StreamTokenizer you can move through the tokens in the underlying Reader. You do so by calling the nextToken() method of the StreamTokenizer inside a loop. After each call to nextToken() the StreamTokenizer has several fields you can read to see what kind of token was read, it's value etc. These fields are:

ttype The type of token read (word, number, end of line)

sval The string value of the token, if the token was a string (word)

nval The number value of the token, if the token was a number.

Example –

```
StreamTokenizer streamTokenizer = new StreamTokenizer(
      new StringReader("Mary had 1 little lamb..."));

while(streamTokenizer.nextToken() != StreamTokenizer.TT_EOF){
   if(streamTokenizer.ttype == StreamTokenizer.TT_WORD) {
      System.out.println(streamTokenizer.sval);
   } else if(streamTokenizer.ttype == StreamTokenizer.TT_NUMBER) {
      System.out.println(streamTokenizer.nval);
   } else if(streamTokenizer.ttype == StreamTokenizer.TT_EOL) {
      System.out.println();
   }
}
streamTokenizer.close();
```

4.11 Utilities

4.11.1 SequenceInputStream

The Java SequenceInputStream combines two or more other InputStream's into one.

First the SequenceInputStream will read all bytes from the first InputStream, then all bytes from the second InputStream.

That is the reason it is called a SequenceInputStream, since the InputStream instances are read in sequence. Two InputStream instances combined with a SequenceInputStream

SequenceInputStream Example

```
InputStream input1 = new FileInputStream("c:\\data\\file1.txt");
InputStream input2 = new FileInputStream("c:\\data\\file2.txt");
SequenceInputStream sequenceInputStream =
                        new SequenceInputStream(input1, input2);
int data = sequenceInputStream.read();
while(data != -1){
        System.out.println(data);
        data = sequenceInputStream.read();
}
```

This Java code example first creates two FileInputStream instances. The FileInputStream extends the InputStream class, so they can be used with the SequenceInputStream.

Second, this example creates a SequenceInputStream . The SequenceInputStream is given the two FileInputStream instances as constructor parameters.

This is how you tell the SequenceInputStream to combine two InputStream instances.

The two InputStream instances combined with the SequenceInputStream can now be used as if it was one coherent stream.

When there is no more data to read from the second InputStream, the SequenceInputStream read() method will return -1, just like any other InputStream does.

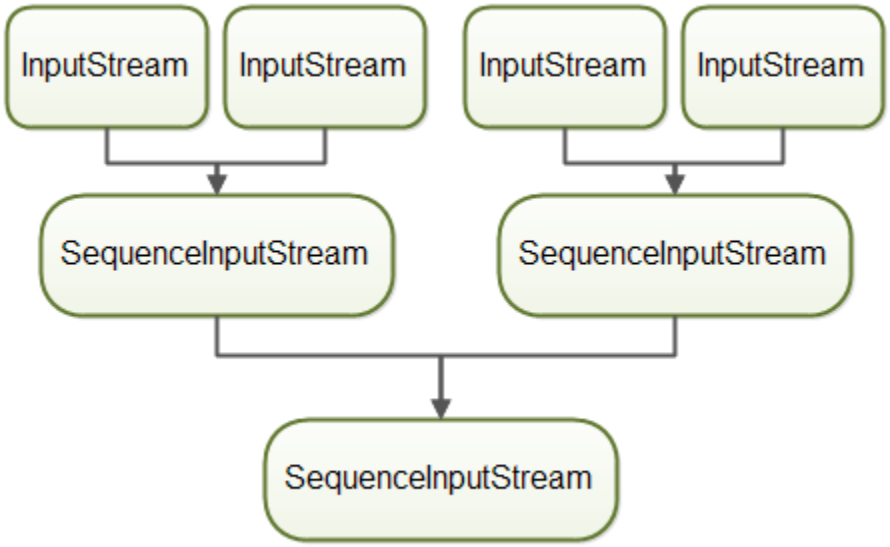
Combining More Than Two InputStreams

```
InputStream input1 = new FileInputStream("c:\\data\\file1.txt");

InputStream input2 = new FileInputStream("c:\\data\\file2.txt");
InputStream input3 = new FileInputStream("c:\\data\\file3.txt");
Vector<InputStream> streams = new Vector<>();
streams.add(input1);
streams.add(input2);
```

```
streams.add(input3);

SequenceInputStream sequenceInputStream =
   new SequenceInputStream(streams.elements()))

int data = sequenceInputStream.read();
while(data != -1){
   System.out.println(data);
   data = sequenceInputStream.read();
}
sequenceInputStream.close();
```

Closing a SequenceInputStream

When you are finished reading data from the SequenceInputStream you should remember to close it. Closing a SequenceInputStream will also close the InputStream instances which the SequenceInputStream is reading.



5 Character Streams

A character encoding is a scheme for representing characters.

- Java programs represent values of the char type internally in the 16-bit Unicode character encoding,
- The host platform might use another character encoding to represent and store characters externally.

For example, the ASCII (American Standard Code for Information Interchange) character encoding is widely used to represent characters on many platforms. However, it is only one small subset of the Unicode standard.

The abstract classes Reader and Writer are the roots of the inheritance hierarchies for streams that read and write Unicode characters using a specific character encoding.

A reader is an input character stream that reads a sequence of Unicode characters.
A writer is an output character stream that writes a sequence of Unicode characters.

Character encodings are used by readers and writers to convert between **external encoding and internal Unicode characters.**

Readers use the following methods for reading Unicode characters:
Note that the read() methods read the character as an int in the range 0 to 65535 (0x0000–0xFFFF). The value –1 is returned if the end of the stream has been reached.

int read() throws IOException
int read(char cbuf[]) throws IOException
int read(char cbuf[], int off, int len) throws IOException
long skip(long n) throws IOException - A reader can skip over characters using the skip() method.

Writers use the following methods for writing Unicode characters:

void write(int c) throws IOException
The write() method takes an int as argument, but writes only the least significant 16 bits.
void write(char[] cbuf) throws IOException
void write(String str) throws IOException
void write(char[] cbuf, int off, int length) throws IOException
void write(String str, int off, int length) throws IOException

These methods write the characters from an array of characters or a string.

void close() throws IOException
void flush() throws IOException

- Character stream should be closed when no longer needed to free system resources. Closing a character output stream automatically flushes the stream.
- A character output stream can also be manually flushed.
- Many methods of the character stream classes throw an IOException

5.1 Basic Character Streams

5.1.1 Reader Class

The Java Reader class (java.io.Reader) is the base class for all Reader subclasses in the Java IO API. A Reader is like an InputStream except that it is character based rather than byte based. In other words, a Java Reader is intended for reading text, whereas an InputStream is intended for reading raw bytes.

Characters in Unicode

Today, many applications use UTF (UTF-8 or UTF-16) to store text data. It may take one or more bytes to represent a single character in UTF-8. In UTF-16 each character takes 2 bytes to represent.

Therefore, when reading text data, a single byte in the data may not correspond to one character in UTF. If you just read one byte at a time of UTF-8 data via an InputStream and try to convert each byte into a char, you may not end up with the text you expected.

To solve this problem we have the Reader class. The Reader class is capable of decoding bytes into characters. You need to tell the Reader what character set to decode. This is done when you instantiate the Reader.

Reading Characters With a Reader

The read() method of a Reader returns an int which contains the char value of the next character read. If the read() method returns -1, there is no more data to read in the Reader, and it can be closed. That is, -1 as int value, not -1 as byte or char value. There is a difference here!

5.1.2 Writer Class

The Java Writer class (java.io.Writer) is the base class for all Writer subclasses in the Java IO API. A Writer is like an OutputStream except that it is character based rather than byte based. In other words, a Writer is intended for writing text, whereas an OutputStream is intended for writing raw bytes

5.1.3 InputStreamReader Class

The Java InputStreamReader class (java.io.InputStreamReader) is intended to wrap an InputStream, thereby turning the byte based input stream into a character based Reader.

The Java InputStreamReader is often used to read characters from files (or network connections) where the bytes represents text. For instance, a text file where the characters are encoded as UTF-8. You could use an InputStreamReader to wrap a FileInputStream in order to read such a file.

```
InputStream inputStream      = new FileInputStream("c:\\data\\input.txt");
Reader     inputStreamReader = new InputStreamReader(inputStream);
int data = inputStreamReader.read();
while(data != -1){
        char theChar = (char) data;
        data = inputStreamReader.read();
```

```
}
inputStreamReader.close();
```

The read() method of an InputStreamReader returns an int which contains the char value of the char read.
Here is a Java InputStreamReader read() example:

```
int data = inputStreamReader.read();
char aChar = (char) data;
```

Character Encoding Constructors
The Java InputStreamReader has a set of alternative constructors that allow you to specify the character set (ISO-Latin1, UTF-8, UTF-16 etc.) to use to interpret the bytes in the underlying InputStream.
Here a character encoding to the constructor:

```
InputStream inputStream      = new FileInputStream("c:\\data\\input.txt");
Reader     inputStreamReader = new InputStreamReader(inputStream, "UTF-8");
```

This InputStreamReader will now interpret the bytes from the underlying InputStream as UTF-8 encoded characters. The InputStreamReader will thus decode the UTF-8 encoded characters one by one, and return each character as a single char value (inside the returned int).

5.1.4 OutputStreamReader Class

The Java OutputStreamWriter class (java.io.OutputStreamWriter) is intended to wrap an OutputStream, thereby turning the **byte based output stream into a character based Writer**.

The Java OutputStreamWriter is useful if you need to write characters to a file, encoded as e.g. UTF-8 or UTF-16. You can then write the characters (char values) to the OutputStreamWriter and it will encode them correctly and write the encoded bytes to the underlying OutputStream.

OutputStreamWriter Example:

```
OutputStream outputStream      = new FileOutputStream("c:\\data\\output.txt");
Writer  outputStreamWriter = new OutputStreamWriter(outputStream);
outputStreamWriter.write("Hello World");
outputStreamWriter.close();
```

Character Encoding Constructors

The Java OutputStreamWriter also has alternative constructors that allow you to specify the character set (ISO-Latin1, UTF-8, UTF-16 etc.) to use to convert the written characters to the bytes written to the underlying OutputStream. Here is a Java OutputStreamWriter example showing the use of one of these constructors:

```
OutputStream outputStream      = new FileOutputStream("c:\\data\\output.txt");
Writer      outputStreamWriter = new OutputStreamWriter(outputStream, "UTF-8");
```

This example creates an OutputStreamWriter that will convert all characters written to it to UTF-8 encoded characters (one or more bytes per character) and write the UTF-8 encoded bytes to the underlying OutputStream

5.2 File Character Streams

5.2.1 FileReader Class

The Java FileReader class (java.io.FileReader) makes it possible to read the contents of a file as a stream of characters. It works much like the FileInputStream except the FileInputStream reads bytes, whereas the FileReader reads characters. The FileReader is intended to read text, in other words. One character may correspond to one or more bytes depending on the character encoding scheme.

FileReader Example

```
Reader fileReader = new FileReader("c:\\data\\input-text.txt");
int data = fileReader.read();
while(data != -1) {
        //do something with data...
        doSomethingWithData(data);
        data = fileReader.read();
}
fileReader.close();
```

This example first creates a FileReader which is connected directly to the file pointed to by the file path passed as parameter to the FileReader constructor. Second, this example reads all characters one char at a time from the FileReader. Third, the FileReader is closed

5.2.2 FileWriter Class

The Java FileWriter class (java.io.FileWriter) makes it possible to write characters to a file. In that respect it works much like the FileOutputStream except that a FileOutputStream is byte based, whereas a FileWriter is character based. The FileWriter is intended to write text, in other words. One character may correspond to one or more bytes, depending on the character encoding scheme in use.

FileWriter Example

```
Writer fileWriter = new FileWriter("data\\filewriter.txt");
fileWriter.write("data 1");
fileWriter.write("data 2");
fileWriter.write("data 3");
fileWriter.close();
```

Overwriting vs. Appending the File

The FileWriter constructor taking just one parameter, the file name, will overwrite any existing file:

FileWriter has a constructor that takes 2 parameters too: The file name and a boolean. The boolean indicates whether to append or overwrite an existing file. Here are two Java FileWriter examples showing that:

```
Writer fileWriter = new FileWriter("c:\\data\\output.txt", true);  //appends to file
Writer fileWriter = new FileWriter("c:\\data\\output.txt", false); //overwrites file
```

5.3 Buffering Character Streams

5.3.1 BufferedReader Class

The Java BufferedReader class (java.io.BufferedReader) provides buffering to your Reader instances. Buffering can speed up IO quite a bit. Rather than read one character at a time from the network or disk, the BufferedReader reads a larger block at a time. This is typically much faster, especially for disk access and larger data amounts.

The Java BufferedReader is similar to the BufferedInputStream but they are not exactly the same. The main difference between BufferedReader and BufferedInputStream is that BufferedReader reads characters (text), whereas the BufferedInputStream reads raw bytes.

BufferedReader Example:

```
BufferedReader bufferedReader = new BufferedReader(new FileReader("c:\\data\\input-file.txt"));
```

BufferedReader Buffer Size

BufferedReader same kind of logic for buffering as BufferedInputStream

5.3.2 BufferedWriter Class

The Java BufferedWriter class (java.io.BufferedWriter) provides buffering to Writer instances. Buffering can speed up IO quite a bit. Rather than write one character at a time to the network or disk, the BufferedWriter writes a larger block at a time. This is typically much faster, especially for disk access and larger data amounts.

BufferedWriter Example

```
BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter("c:\\data\\output-file.txt"));
```

BufferedWriter Buffer Size

BufferedWriter same kind of logic for buffering as BufferedOutputStream

```
int bufferSize = 8 * 1024;
BufferedWriter bufferedWriter =  new BufferedWriter(
        new FileWriter("c:\\data\\output-file.txt"), bufferSize);
```

5.4 Arrays Character Streams

5.4.1 CharArrayReader Class

The Java CharArrayReader class (java.io.CharArrayReader) enables you to read the contents of a char array as a character stream.

The Java CharArrayReader is handy when you have data in a char array, but need to pass that data to some component which can only read from a Reader (or a Reader subclass). Simply wrap the char array in a CharArrayReader and pass it to that component.

CharArrayReader Example

```
char[] chars = "123".toCharArray();
CharArrayReader charArrayReader = new CharArrayReader(chars);
int data = charArrayReader.read();
while(data != -1) {
        //do something with data
        data = charArrayReader.read();
}
charArrayReader.close();
```

This example first creates a char array from a Java string. Second, the example creates a CharArrayReader instance, passing the char array as parameter to the CharArrayReader constructor. Third, the example reads the chars one by one from the CharArrayReader, and finally the CharArrayReader is closed.

Creating a CharArrayReader From Part of a char Array

It is possible to create a Java CharArrayReader from just part of a char array. Here is an example showing how to create a CharArrayReader that reads only part of a char array:

```
char[] chars = "0123456789".toCharArray();
int offset = 2;
int length = 6;
CharArrayReader charArrayReader = new CharArrayReader(chars, offset, length);
```

5.4.2 CharArrayWriter Class

The Java CharArrayWriter class (java.io.CharArrayWriter) makes it possible to write characters via the Writer methods (CharArrayWriter is a subclass of Writer) and convert the written characters into a char array.

The Java CharArrayWriter class is handy when you have a component that can only write characters to a Writer, but you need the characters as a char array. Simply pass that component a CharArrayWriter and when all characters are written to it, call toCharArray() on the CharArrayWriter.

CharArrayWriter Example

```
CharArrayWriter charArrayWriter = new CharArrayWriter();
charArrayWriter.write("CharArrayWriter");
char[] chars1 = charArrayWriter.toCharArray();
charArrayWriter.close();
```

Setting Initial char Array Size

The Java CharArrayWriter has a constructor that lets you set the initial size of the char array used internally to store the written characters.
Setting the initial size does not prevent the CharArrayWriter from storing more characters than the initial size. If the number of characters written to the CharArrayWriter exceed the size of the initial char array, a new char array is created, and all characters are copied into the new array.

```
int initialSize = 1024;
CharArrayWriter charArrayWriter = new CharArrayWriter(initialSize);
```

5.5 Data – Formatted Character Streams

5.5.1 PrintWriter Class

The capabilities of the OutputStreamWriter and the InputStreamReader classes are limited, as they primarily write and read characters.

In order to write a text representation of Java primitive values and objects, a PrintWriter should be chained to either a writer, a byte output stream, a File, or a String file name, using one of the following constructors:

In that way the PrintWriter is a bit different from other Writer subclasses which tend to have mostly constructors that can take other Writer instances as parameters (except for a few, like OutputStreamWriter).

```
PrintWriter(Writer out)
PrintWriter(Writer out, boolean autoFlush)
PrintWriter(OutputStream out)
PrintWriter(OutputStream out, boolean autoFlush)
PrintWriter(File file)
PrintWriter(File file, String charsetName)
PrintWriter(String fileName)
PrintWriter(String fileName, String charsetName)
```

The autoFlush argument specifies whether the PrintWriter should be flushed when any println() method of the PrintWriter class is called.

The Java PrintWriter class (java.io.PrintWriter) enables you to write formatted data to an underlying Writer. For instance, writing int, long and other primitive data formatted as text, rather than as their byte values.

The Java PrintWriter is useful if you are generating reports (or similar) where you have to mix text and numbers.

The PrintWriter class has all the same methods as the PrintStream except for the methods to write raw bytes. Being a Writer subclass the PrintWriter is intended to write text.

PrintWriter Example

```
FileWriter  writer     = new FileWriter("d:\\data\\report.txt");
PrintWriter printWriter = new PrintWriter(writer);
printWriter.print(true);
printWriter.print((int) 123);
printWriter.print((float) 123.456);
printWriter.printf(Locale.UK, "Text + data: %1$", 123);
printWriter.close();
```

5.6 Pipes Character Streams

5.6.1 PipedReader Class

The Java PipedReader class (java.io.PipedReader) makes it possible to read the contents of a pipe as a stream of characters. As such it works very much like a PipedInputStream except the PipedInputStream is byte based, not character based. The PipedReader is intended to read text, in other words.

A Java PipedReader must be connected to a PipedWriter. Often, the PipedReader and PipedWriter are used by different threads. Only one PipedReader can be connected to the same PipedWriter.

PipedReader Example

```
PipedWriter pipedWriter = new PipedWriter();
PipedReader pipedReader = new PipedReader(pipedWriter);
int data = pipedReader.read();
while(data != -1) {
        //do something with data...
        doSomethingWithData(data);
        data = pipedReader.read();
}
pipedReader.close();
```

5.6.2 PipedWriter Class

The Java PipedWriter class (java.io.PipedWriter) makes it possible to write to a Java pipe as a stream of characters. In that respect the PipedWriter works much like a PipedOutputStream except that a PipedOutputStream is byte based, whereas a PipedWriter is character based. The PipedWriter is intended for writing text, in other words.

Normally a Java PipedWriter is connected to a PipedReader. And often the PipedWriter and the PipedReader are used by different threads.

PipedWriter Example

```
PipedWriter pipedWriter = new PipedWriter();
while(moreData()) {
  int data = getMoreData();
  pipedWriter.write(data);
}
pipedWriter.close();
```

write()

The write() method of a PipedWriter takes an int which contains the byte value of the byte to write. There are also versions of the write() method that take a String, char array etc.

5.7 Filtering Character Streams

5.7.1 FilterReader Class

The FilterReader is a base class for implementing your own filtering readers. Basically it just overrides all methods in Reader.

Like with FilterInputStream, I see no sensible purpose for this class. I cannot see that this class actually adds or changes any behaviour in Reader except that it takes a Reader in its constructor. If you choose to extend this class you might as well extend the Reader class directly, and avoid the extra class in the hierarchy.

5.7.2 FilterWriter Class

The FilterWriter is a base class for implementing your own filtering Writer's. Basically it just overrides all methods in Writer.

Like with FilterOutputStream, I see no sensible purpose for this class. I cannot see that this class actually adds or changes any behaviour in Writer except that it takes a Writer in its constructor. If you choose to extend this class you might as well extend the Writer class directly, and avoid the extra class in the hierarchy.

5.8 Parsing Character Streams

5.8.1 PushbackReader Class

The Java PushbackReader class (java.io.PushbackReader) is intended to be used when you parse data from a Reader. Sometimes you need to read ahead a few characters to see what is coming, before you can determine how to interpret the current character. The PushbackReader allows you to do that. Well, actually it allows you to push back the read characters into the Reader. These characters will then be read again the next time you call read().

The Java PushbackReader works much like the PushbackInputStream except that the PushbackReader works on characters, whereas the PushbackInputStream works on bytes.

PushbackReader Example

```
PushbackReader pushbackReader=new PushbackReader(new FileReader("c:\\data\\input.txt"));
int data = pushbackReader.read();
pushbackReader.unread(data);
```

The call to read() reads a character from the PushbackReader just like from any other Reader.

The call to unread() pushes a character back into the PushbackReader.

The next time read() is called the pushed back characters will be read first. If you push back multiple characters into the PushbackReader, the latest pushed back character will be returned first from the read() method, just like with a stack.

Setting the Push Back Limit of a PushbackReader

You can set the number of characters you should be able to unread in the constructor of the PushbackReader. Here is how to set the pushback limit using the PushbackReader constructor:

```
int pushbackLimit = 8;
PushbackReader rd = new PushbackReader(new FileReader("c:\\data\\input.txt"),
pushbackLimit);
```

This example sets an internal buffer of 8 characters in the PushbackReader.

That means you can unread at most 8 characters at a time, before reading them again.

5.8.2 LineNumberReader Class

The Java LineNumberReader class (java.io.LineNumberReader is a BufferedReader that keeps track of line numbers of the read characters. Line numbering begins at 0. Whenever the LineNumberReader encounters a line terminator in the characters returned by the wrapped Reader, the line number is incremented.

You can get the current line number from the LineNumberReader by calling the getLineNumber() method. You can also set the current line number, should you need to, by calling the setLineNumber() method.

LineNumberReader Example

```
LineNumberReader lineNumberReader = new LineNumberReader(new
FileReader("c:\\data\\input.txt"));
int data = lineNumberReader.read();
while(data != -1){
   char dataChar = (char) data;
   data = lineNumberReader.read();
   int lineNumber = lineNumberReader.getLineNumber();
}
lineNumberReader.close();
```

This example first creates a LineNumberReader, and then shows how to read all the characters from it, and also shows how to get the line number (for each character read, in fact, which may be a bit more than you need).

Line Numbers in Parsing

Line number can be handy if you are parsing a text file that can contain errors. When reporting the error to the user, it is easier to correct the error if your error message includes the line number where the error was encountered.

5.9 Strings Character Streams

5.9.1 StringReader Class

The Java StringReader class enables you to turn an ordinary String into a Reader. This is useful if you have data as a String but need to pass that String to a component that only accepts a Reader.

StringReader Example

```
String input = "Input String... ";
StringReader stringReader = new StringReader(input);
int data = stringReader.read();
while(data != -1) {
        //do something with data...
        doSomethingWithData(data);
        data = stringReader.read();
}
stringReader.close();
```

5.9.2 StringWriter Class

The Java StringWriter class (java.io.StringWriter) enables you to obtain the characters written to a Writer as a String. The StringWriter is useful if you have a component that only can write data to a Writer but you need that data as a String.

StringWriter Example

```
StringWriter stringWriter = new StringWriter();
//write characters to writer.
stringWriter.write("This is a text");
String       data       = stringWriter.toString();
StringBuffer dataBuffer = stringWriter.getBuffer();
stringWriter.close();
```

Third the characters written to the StringWriter are obtained via the two methods toString() and getBuffer().

You only need to use one of these two methods, but both are showed so you know they both exist.

The method toString() returns the characters written to the StringWriter as a String.

The method getBuffer() returns the StringBuffer used by the StringWriter to build the string from the written characters.