WOLDIA UNIVERSITY
TECHNOLOGY COLLEGE
SCHOOL OF COMPUTING

DEPARTMENT OF SOFTWARE
ADVANCED PROGRAMMING

CHAPTER TWO
File I/O and Streams

By Demeke G.
AY-2017

Outline

- Introduction to streams
- File Management
- Reading and Writing text files
- Reading and Writing binary files
- Object Streams

Introduction

- A file is a unit of digital storage used for organizing and storing information on a computer
- It is a collection of data or information that is saved under a specific name, referred to as the filename
- It is stored in a storage medium such as a hard drive, SSD, USB drive, or cloud storage.
- Files are the most important mechanism for storing data permanently on mass-storage devices.
- The file system is the structure by which computers manage files on their hard drives.
- Most programs cannot accomplish their goals without accessing external data

Cont...

- The **java.io** package contains classes to perform input and output
- **Files can contain:**
  - data in a format that can be interpreted by programs, but not easily by humans (*binary files*);
  - alphanumeric characters, codified in a standard way (e.g., using ASCII or Unicode), and directly readable by a human user (*text files*.
  - Text files are normally organized in a sequence of lines, each containing a sequence of characters and ending with a special character (usually the *newline character*). for example, a Java program stored in a file on the hard-disk.
- Each file is characterized by a name and a directory in which the file is placed
- File has Filename, Extension, Path, Size and Permission

Exceptions

- File operations can typically cause **unexpected** situations that the program is not capable of handling.
- For example, if we try to open a file for reading, and specify a filename that does not exist. Such situations are called **exceptions**.
- the methods for opening a file for reading or writing can generate an exception of type **IOException**

```
public static void main(String[] args) throws IOException {
        //or
            try{   body    }
                        catch(IOException ex){
            System.out.print(ex);
            }
             }
```

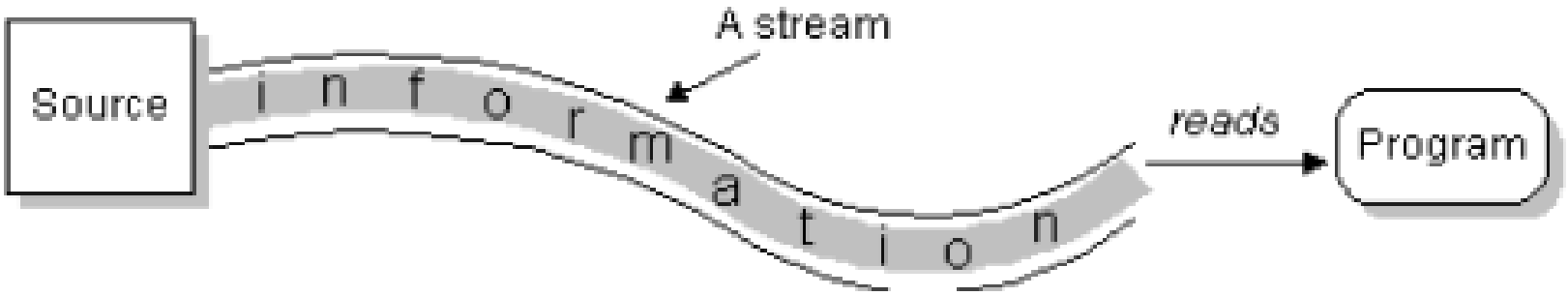
Stream Classes

- A **stream** represents a flow of data, or a channel of communication with a writer at one end and a reader at the other.
- A stream can be defined as a sequence of data.
- When you are working with terminal input and output, reading or writing files, or communicating through sockets in Java, you are using a stream
- Java's stream-based I/O is built upon four abstract classes:
  InputStream, **OutputStream**, **Reader**, and **Writer**.
- **InputStream** and **OutputStream** are designed for byte streams.
- **Reader** and **Writer** are designed for character streams.
- Streams have no idea of the structure or meaning of your data
- There are two kinds of Streams
  - **InPutStream**: used to read data from a source.
  - **OutPutStream**: used for writing data to a destination

Cont..

Input Stream

- To bring in information, a program opens a stream on an information source (a file, memory, a socket) and reads the information sequentially, as shown here:
- The **InputStream** is used to read data from a source.

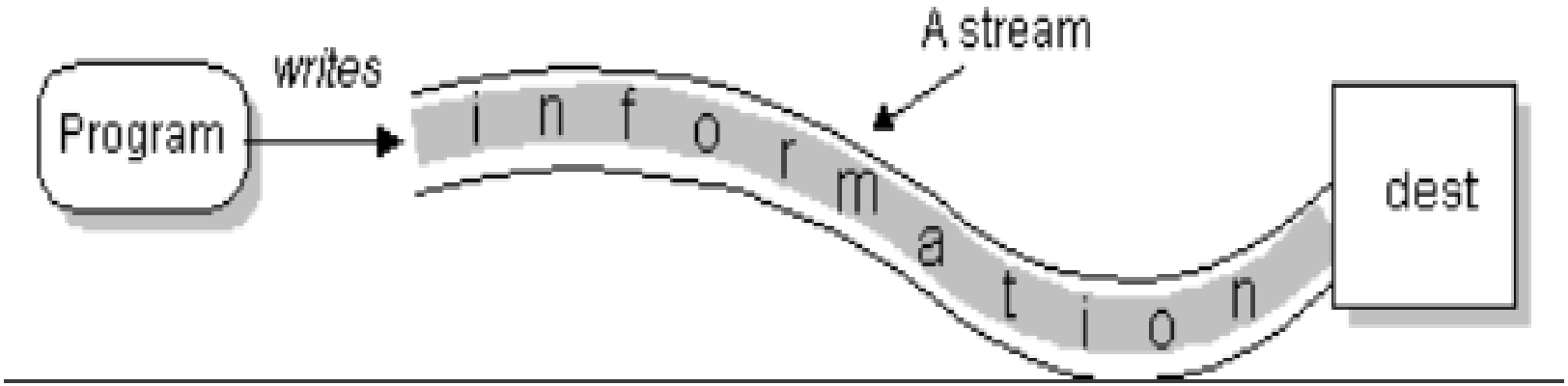


- **InputStream** is an abstract super class for all classes representing an **inputStream** of bytes –
- Input can be from keyboard, File and memory.

Cont..

Output Stream

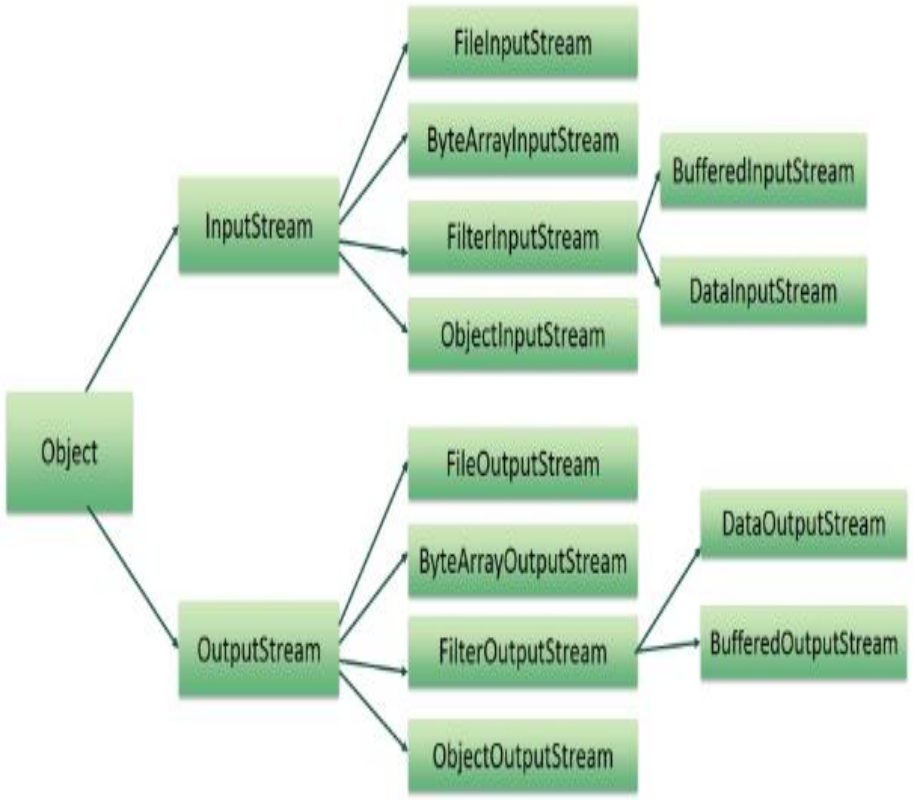
- A program can send information to an external destination by opening a stream to a destination and writing the information out sequentially
- **OutputStream** is used for writing data to a destination.



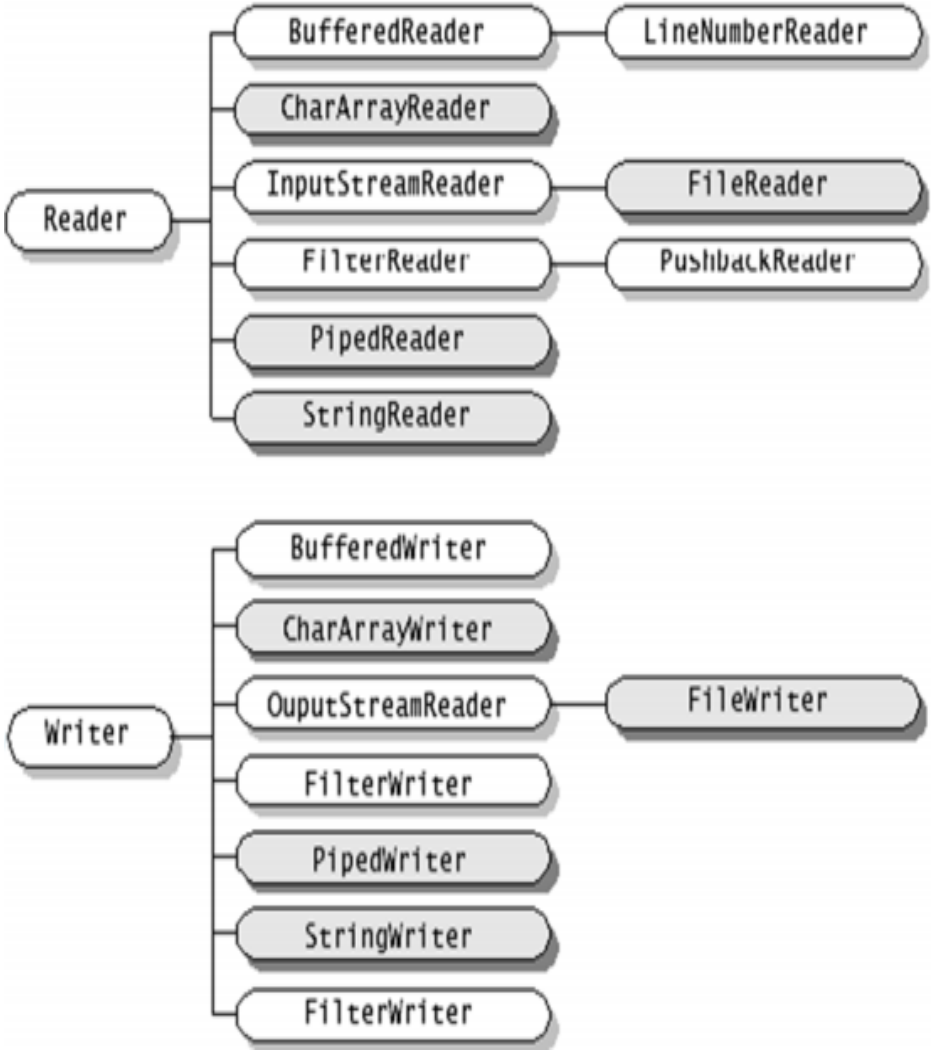
- **OutputStream** is also an abstract super class for all classes representing an output stream of bytes.
- Output can be from monitor, memory or File.

cont..

classes of binary stream writer and reader



Character Stream writer and reader classes



Cont..

- Java stream I/O divided into two

1. Character streams : read and **write** text characters that represent strings.

- Text files may use delimiters (special characters) to separate data elements
- i.e- A comma-delimited file uses commas to separate individual fields of data.

2. Binary streams

- Binary streams **read** and **write** individual bytes that represent primitive data types.
- You can connect a binary stream to a binary file to store binary data on disk .
- Programs that work with file streams include an **import java.io.*** statement.
- i.e. primitive data types, objects, or images. Video

File management

- The ***File*** class is your key to processing files and directories.
- A File object represents a single file or directory
- Java uses a single class to represent both files and directories because a directory is a special type of file.
- The ***File*** class is in the ***java.io*** package

❖ **main constructors and methods of the *File* class**

- ❖ File(String pathname)
- ❖ boolean canRead() :Determines whether the file can be read.
- ❖ boolean canWrite() : Determines whether the file can be written.

Cont..

Methods and description

- ***boolean createNewFile() :***Creates the file on disk if it doesn't already exist.Returns true if the file was created, false if the file already existed.
- ***boolean delete():*** Deletes the file or directory. Returns true if the file was successfully deleted.
- ***boolean exists():*** Returns true if the file exists on disk, false if the file doesn't exist
- ***String getCanonicalPath():*** Returns the complete path to the file, including the drive letter if run on a Windows system. Throws ***IOException***.
- ***String getName() :***Gets the name of this file.
- ***String getParent():*** Gets the name of the parent directory of this file or directory.

Cont..

Method and description

- ***boolean isDirectory():*** Returns true if this ***File*** object is a directory, false if it is a file.
- ***boolean isFile() :*** Returns true if this ***File*** object is a file, false if it is a directory.
- ***boolean isHidden():*** Returns true if this file or directory is marked by the operating system as hidden.
- ***long length() :***Returns the size of the file in bytes.
- ***String[] list()*** :Returns an array of String objects with the name of each file and directory in this directory. Each string is a simple filename, not a complete path. If this File object is not a directory, returns null.

Cont..

- ***File[] listFiles()*** :Returns an array of File objects representing each file and directory in this directory. If this File object is not a directory, returns null.
- ***String toString():*** Returns the pathname for this file or directory as a string.
- ***boolean mkdir() :***Creates a directory on disk from this File object. Returns true if the directory was successfully created.
- ***boolean mkdirs():*** Creates a directory on disk from this File object, including any parent directories that are listed in the directory path but don't already exist. Returns true if the directory was successfully created.
- ***boolean renameTo(File dest) :*** Renames the File object to the specified destination File object. Returns true if the rename was successful.
- ***boolean setReadOnly():*** Marks the file as read-only. Returns true if the file was successfully marked.

Example

- To create a File object, you can call the File constructor, passing a string representing the filename of the file as a parameter.

  File f = new File("D://hits.log");

- To find out if the file/ directory exists, use the exists method

  if (!f.exists())
  System.out.println("The input file does not exist!");

- To create a new file on disk

  if (f.createNewFile())
  System.out.println("File created.");
  else
  System.out.println("File could not be created.");

- To get just the filename, use the ***getName*** method.This method returns a string the filename, not the complete path.
- To get the full path for a file use the ***getCannonicalPath*** method.

Cont..

- To lists the name of every file in a directory

```
File dir = new File(path);
if (dir.isDirectory()) {
File[] files = dir.listFiles();
for (File f : files)
System.out.println(f.getName());
}
```

- To change the name of a file named ***hits.log*** to ***savedhits.log***:

```
File f = new File("D://hits.log");
if (f.renameTo(new File("savedhits.log")))
System.out.println("File renamed.");
else
System.out.println("File not renamed.");
```

- ***To delete a file call the delete method***

```
 if (f.delete())
System.out.println("File deleted.");
else
System.out.println("File not deleted.");
```

Reading Character Streams

- To read a text file through a character stream, usually work with the following classes:

1. **File:** The ***File*** class, represents a file on disk. In file I/O applications, the main purpose of the **File** class is to ***identify the file you want to read from or write to.***
2. **FileReader**

- The **FileReader** class provides basic methods for reading data from a character stream
- It provides methods that let you read data ***one character at a time.***
- Create a **FileReader** object to connect your program to a file, and then pass that object to the constructor of the BufferedReader class, ***which provides more efficient access to the file.***

3. **BufferedReader**

- This class “wraps” around the **FileReader** class to provide more efficient input.
- This class adds a buffer to the input stream that allows the input to be read from disk in large chunks rather ***than one byte at a time***.
- This can result in a huge improvement in performance.
- The **BufferedReader** class lets you read data one character at a time or a line at a time.
- Use Java’s string-handling features to break the line into individual fields.

BufferedReader and FileReader classes Constructor

Constructors	Description
BufferedReader (Reader in)	Creates a buffered reader from any object that extends the Reader class. you pass this constructor a **FileReader** object.
FileReader(File file)	Creates a file reader from the specified File object. Throws FileNotFoundException if the file doesn't exist or if it is a directory rather than a file.
FileReader(String path)	Creates a file reader from the specified pathname. Throws FileNotFoundException if the file doesn't exist or if it is a directory rather than a file.

BufferedReader and FileReader methods

Methods	Description
void close()	Closes the file. Throws IOException.
int read()	Reads a single character from the file and returns it as an integer. Returns –1 if the end of the file has been reached. Throws ***IOException***.
String readLine()	Reads an entire line and returns it as a string. Returns null if the end of the file has been reached.

Example

```
class ReadCStream{
public static void main(String[]args){
File file= new File("D://student.txt");
FileReader fr= new FileReader(file);
BufferedReader br= new BufferedReader(fr);
String str=br.readLine();
while(str!=null){
str=br.readLine();
System.out.println(str);
}}}
```

Writing Character Streams

- To write data to a text file; use the ***PrintWriter*** class.
- To connect a print writer to a text file. you work with ***three classes***

1. ***FileWriter:*** This class connects to a **File** object but provides only rudimentary writing ability**.**
2. ***BufferedWriter:*** This class connects to a ***FileWriter*** and provides output buffering.
   - Without the buffer, data is written to disk one character at a time.
   - This class lets the program accumulate data in a buffer and writes the data only when the buffer is filled up or when the program requests that the data be written.

3. PrintWriter

- This class connects to a Writer, which can be a **BufferedWriter,** a **FileWriter,** or any other object that extends the abstract ***Writer*** class. Most often, you connect this class to a ***BufferedWriter.***

❖ **Most important constructors and methods of this class**

Constructors	Description
PrintWriter(Writer out)	Creates a print writer for the specified output writer
PrintWriter(Writer out, boolean flush)	Creates a print writer for the specified output writer. If the second parameter is true, the buffer is automatically flushed whenever the println method is called.
BufferedWriter(Writer out)	Creates a buffered writer from the specified writer. Typically, you pass this constructor a FileWriter object.

Cont..

Construcrtors	Description
FileWriter(File file)	Creates a file writer from the specified File object. Throws ***IOException*** if an error occurs
FileWriter(File file, boolean append)	Creates a file writer from the specified File object. Throws ***IOException*** if an error occurs. If the second parameter is true, data is added to the end of the file if the file already exists.
FileWriter(String path)	Creates a file writer from the specified pathname. Throws ***IOException*** if an error occurs.

PrintWriter Methods

Methods	Description
void close()	Closes the file.
void flush()	Writes the contents of the buffer to disk.
void print(value)	Writes the value, which can be any primitive type or any object. If the value is an object, the object's ***toString()*** method is called.
void println(value)	Writes the value, which can be any primitive type or any object. If the value is an object, the object's ***toString()*** method is called. A line break is written following the value.

1.Example

- create a **File** object for the file
- call the ***PrintWriter*** constructor to create a ***PrintWriter*** object you can use to write to the file.
- This constructor wraps around a ***BufferedWriter*** object, which in turn wraps around a **FileWriter** object like this:

```
File file = new File("studentlist.txt");
PrintWriter out = new PrintWriter(
new BufferedWriter(
new FileWriter(file)
) );
```

Cont..

- At the bottom is the ***FileWriter*** class, which has the **ability to write characters to a file**.
- ***BufferedWriter*** class adds buffering to the mix, saving data in a buffer until it makes sense to write it all out to the file.
- The ***PrintWriter*** class adds basic formatting capabilities, like adding line endings at the end of each line and converting primitive types to strings.
- Both the ***FileWriter*** and the ***PrintWriter*** classes have an optional **boolean parameter** add extra capabilities to the file stream. If you specify **true** in the ***FileWriter*** constructor, the file is appended if it exists.

Con..

- i.e. Here's a **PrintWriter** constructor that appends data to its file:

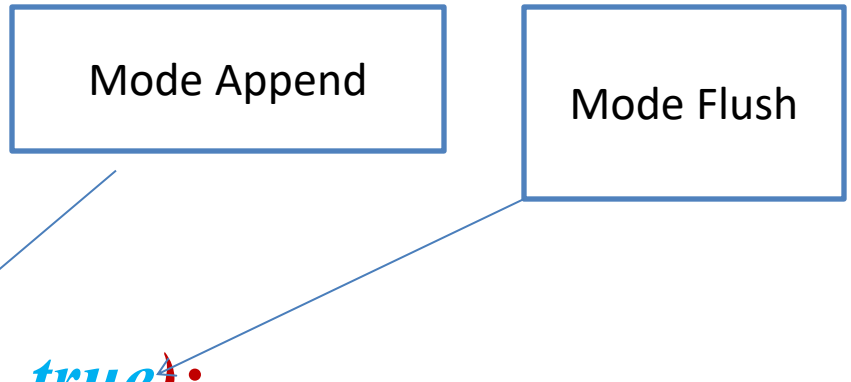
```
File file = new File("movies.txt");
PrintWriter out =new PrintWriter( new BufferedWriter (
new FileWriter(file, true )))// append mode
```

- If you specify **false** instead of **true**, or if you leave this parameter out altogether, an existing file is deleted, and its data is lost.
- The **boolean** parameter in the **PrintWriter** tells the **BufferedWriter** class to flush its buffer whenever you use the **println** method to write a line of data.

Cont..

- The code for specifying this option looks

```
File file = new File("studentlist.txt");
PrintWriter out =
new PrintWriter(
new BufferedWriter(
new FileWriter(file,true) ), true);
```



OR

```
FileWriter fw = new FileWriter(file, true);
BufferedWriter bw = new BufferedWriter(fw);
PrintWriter out = new PrintWriter(bw, true);
```

Example

```
Class WriteCStream{
Public static void main(String[]args){
Try{
    FileWriter fw = new FileWriter(file, true);//append
    BufferedWriter bw = new BufferedWriter(fw);
    PrintWriter out = new PrintWriter(bw, true); //flush
    out.println("writting");// write with new line
    out.print("writing");//without newline

}catch(Exception ex){}

}
}
```

Reading Binary Streams

- Binary streams are a bit tougher to read than character streams b/c you need to know exactly the type of each item that was written to the file.
- If any incorrect data is in the file, the program won't work.

❖ **To read a binary file use following classes:**

- ***File*:** use the File class to represent the file itself.
- ***FileInputStream:*** it connects the input stream to a file.
- **BufferedInputStream:** This class adds buffering to the basic **FileInputStream**, which improves the stream's efficiency.

Cont..

- **DataInputStream:** This class used to read data from the stream.
  - ❖This class knows how to read basic data types, including primitive types and strings.
- **constructors and methods of these classes are:** ***BufferedInputStream(InputStream in)*** Creates a buffered input stream from any object that extends the InputStream class. you pass this constructor a **FileInputStream** object.

Cont..

DataInputStream (InputStream in) :Creates a data input stream from any object that extends the **InputStream** class. typically, you pass this constructor a **BufferedInputStream** object.

FileInputStream (File file) :Creates a file input stream from the specified File object. Throws ***FileNotFoundException*** if the file doesn't exist or if it is a directory rather than a file

Cont..

DataInputStream Methods	Description
boolean readBoolean()	Reads a boolean value from the input stream.
byte readByte()	Reads a byte value from the input stream. Throws EOFException and IOException.
char readChar()	Reads a char value from the input stream. Throws EOFException and IOException.
readDouble()	Reads a double value from the input stream.
float readFloat()	Reads a float value from the input stream.
int readInt()	Reads an int value from the input stream.
String readUTF()	Reads a string stored in UTF format from the input stream. Throws EOFException, IOException, and UTFDataFormatException.

Creating a DataInputStream

- To read data from a binary file, you want to connect a ***DataInputStream*** object to an input file. To do that use,
- A ***File*** object to represent the file,
- A ***FileInputStream*** object that represents the *file as an input stream*,
- **A *BufferedInputStream*** object that adds buffering to the mix
- A ***DataInputStream*** object to provide the methods that read various data type.
- The constructor looks like this:

```
File file = new File("movies.dat");
DataInputStream in = new DataInputStream(
new BufferedInputStream(
new FileInputStream(file) ) );
```

Reading from a data input stream

- With binary files, you don't read an entire line into the program and parse it into individual fields.
- Instead, you use the various read methods of the ***DataInputStream*** class to read the fields one at a time.
- To do that, you have to know the exact sequence in which data values appear in the file.

```
File file = new File("C:\\javastreams\\totalmark.txt");
in = new DataInputStream(  new BufferedInputStream(
        new FileInputStream(file)));
      st = in.readUTF();
       st2 = in.readUTF();
       b = in.readInt();
       c = in.readDouble();
```

Writing Binary Streams

- To write data to a binary file, you use the following classes
- **FileOutputStream:** This class connects to a ***File*** object and creates an output stream that can write to the file. However, this output stream is limited in its capabilities: It can write only *raw bytes to the file*. In other words, it doesn't know how to write values such as ***ints, doubles, or strings.***
- **BufferedOutputStream**: This class connects to a **FileOutputStream** and adds output buffering.
- **DataOutputStream:** This class adds the ability to write primitive data types and strings to a stream.
- The following table lists the essential constructors and methods of these classes.

Cont..

Constructors	Description
DataOutputStream (OutputStream out)	Creates a data output stream for the specified output stream.
BufferedIOutputStream (OutputStream out)	Creates a buffered output stream for the specified stream. Typically, you pass this constructor a **FileOutputStream** object.
FileOutputStream (File file)	Creates a file writer from the file. Throws ***FileNotFoundException*** if an error occurs.
FileOutputStream(File file, boolean append)	Creates a file writer from the file. Throws FileNotFoundException if an error occurs. If the second parameter is true, data is added to the end of the file if the file already exists.

DataOutputStream Methods	Description
void close()	Closes the file.
void flush()	Writes the contents of the buffer to disk.
int size()	Returns the number of bytes written to the file.
void writeBoolean (boolean value)	Writes a boolean value to the output stream. Throws ***IOException***.
void writeByte(byte value)	Writes a byte value to the output stream. Throws IOException.
void writeChar(char value)	Writes a char value to the output stream. Throws ***IOException***.
void writeUTF(String value)	Writes a string stored in UTF format to the output stream. Throws ***EOFException, IOException***, and ***UTFDataFormatException***.
void writeInt(int value)	Writes an int value to the output stream. Throws IOException.
Read about writeDouble, writeFloat, writeLong and writeShor????	

Creating a DataOutputStream

- Creating a ***DataOutputStream*** object requires yet another one of those crazy nested constructor things:

```
File file = new File(name);
DataOutputStream out = new DataOutputStream(
new BufferedOutputStream(
new FileOutputStream(file) ) );
```

- If you prefer, you can unravel the constructors like this:

```
File file = new File(name);
FileOutputStream fos = new FileOutputStream(file);
BufferedOutputStream bos = new
BufferedOutputStream(fos);
DataOutputStream out = new DataOutputStream(bos);
```

Cont..

- The ***FileOutputStream*** class has an optional boolean parameter you can use to indicate that the file should be ***appended*** if it exists.
- To use this feature, call the constructors like this:

```
File file = new File(name);
DataOutputStream out = new DataOutputStream(
new BufferedOutputStream(
new FileOutputStream(file, true) ) );
```

- If you specify false instead of true or leave the parameter out altogether,an existing file is *deleted and its data is lost.*

Writing to a binary stream

- After you successfully connect a ***DataOutputStream*** to a file, writing data to it is simply a matter of calling the various write methods to write different data types to the file. i.e,

```
dout.writeUTF(“Computer");
 dout.writeUTF(“Science");
 dout.writeInt(667);
 dout.writeDouble(123.7);
```

- Of course, these methods throw **IOException**. As a result, you have to enclose them in a ***try/catch*** block.

Object Streams

- Writing objects to a stream
- object streams support I/O of objects
- The object stream classes are **ObjectInputStream** and **ObjectOutputStream**.
- All the primitive data I/O methods covered in Data Streams are also implemented in object streams
- An object stream can contain a mixture of primitive and object values
- **ObjectInputStream** class used to deserialize the primitive data and objects which are written by using **ObjectOutputStream**
- The class should implements the **Serializable** interface
- oos.writeObject(myObject); to write an object
- ois.readObject() : To read an object

Example

- The class should implement the Serializable interface
- Serializable interface found in **java.io** package

```
public class Person  implements Serializable{
    private String username;
    private int    id;
    public String getUserName() { return username; }
    public void setUserName(String str) { username = str;}
    public int getID() { return id; }
    public void setID(int ID) { id = ID; }
}
```

Write an object of Person to file

```
public class JavafileRW {
    public static void main(String[] args) {
try {
          var  fos= new   FileOutputStream("H://fileobject.txt");
            var oos= new ObjectOutputStream(fos);
            var obj= new Person();
            obj.setID(200);
            obj.setUserName("Aster");
            oos.writeObject(obj);
            oos.flush();
} catch (Exception e) {
      e.printStackTrace();
        } }}
```

Reading an object of Person from file

```
public class Javafilerw2 {
    public static void main(String[] args) {
try {
          var fis= new  FileInputStream("H://fileobject.txt");
            var ois= new ObjectInputStream(fis);
            var object=(Person) ois.readObject();
            System.out.println(object.getID());
            System.out.println(object.getUserName());
            oos.flush();
} catch (Exception e) {
      e.printStackTrace();
         }
 }}
```

END

THANK YOU!!!