Chapter 3

Input and Output (I/O)

Java offers a fairly consistent API for reading and writing streams of bytes. A Stream is a sequence of bytes or characters that may be read from some source (eg a keyboard, a file, arrays memory, a network resource, or another program) or written to some destination (eg a file) The process for reading and processing bytes from a file on the disk is not different from reading from a keyboard, or over the network. They are treated about equally. The difference lies in the object (and constructor) used to interact with the source of bytes.

There are two major categories of streams: InputStream and OutputStream. As their names imply, one is for reading from, and the other is for writing to. Most streams are treated as byte streams, but it can be convenient to treat streams as character streams. Java thus provides two additional streams: Reader and Writer. Unlike the Streams which operate on 8-bit bytes, character streams work on 16-bit Unicode characters.

File I/O

Reading from a File:

To read from a file, import the java.io.* packages.
Assuming a file “Data.txt” contains some text, it can be opened as a file input stream. The read function returns one byte at a time if it reads successfully. In the case of Listing 2, it can be seen that each byte read is printed out on the screen. The byte must however be cast as a character so that the characters corresponding to the ASCII value read (in bytes) is printed. Otherwise ‘A’ will be printed as the number 65, and ‘a’ will print out as the number 97. (those are the ASCII equivalent byte values.) Text is actually stored as the binary equivalent of the ASCII representation.

The following piece of code in Listing 2 assumes that a file named “Data.txt” exists and contains the text in Listing 1.

```
-rw-r--r--   1 namanquah  staff     0 Sep 16 13:29 data.txt
-rw-r--r--   1 namanquah  staff  1784 Sep 16 13:28 dirListerv2.class
-rw-r--r--@  1 namanquah  staff   950 Sep 16 13:21 dirListerv2.java
-rw-r--r--   1 namanquah  staff  1925 Sep 16 13:29 fileIO.class
-rw-r--r--@  1 namanquah  staff  1331 Sep 16 13:29 fileIO.java
```

Listing 1 Sample Data text file content

This code in Listing 2 reads and displays on screen exactly what is Listing 1.

```
public static void readFileAsByteStream () throws IOException
{
        InputStream in1 =new FileInputStream("Data.txt");
             int x;
             while ((x=in1.read()) != -1)
               {
                      System.out.print((char)x);
               }
                    in1.close();
}
```

Listing 2 Reading a file as byte stream and displaying on console

The code in Listing 2 can throw an IOException, and it must be caught, or declared to be thrown. The latter has been done in this case in the function header. It is always important to close the streams that are opened, to free up system resources. A good strategy is to place statement to close the file/stream in a finally block, to ensure it is always closed. It is entirely possible that the stream was not created successfully, hence the check to ensure that it is not null, before calling a close() method on it. Failing to check first could result in an error if it was not created. Listing 3 illustrates this at the end of the code.

```
public static void readFileAsCharStream() throws IOException
{
      FileReader in1 =null;

      try
      {
            in1=new FileReader("Data.txt");
            int x;
            while ((x=in1.read()) != -1)
            {
                  System.out.print((char)x);

            }
      }finally
      {
            if (in1!=null)
            in1.close();
      }
}
```

Listing 3 Character Oriented FileIO, also showing how to close file resource.

Listing 3 is a character oriented version of Listing 2. The output for the two pieces of code is identical. The difference is that Listing 3 treats the value read as a 16-bit value (thus supporting more than the basic ASCII set of characters) while the former treats it as an 8-bit value. The real advantage in using character streams is when a file is processed line by line rather than character by character. InputStreams are the primitives for dealing with low level IO and process input byte by byte. This is ideal for dealing with binary data such as images and zipped files. Most streams are byte streams.

Input can be buffered, meaning that a read request reads the desired character plus surrounding characters into memory, so that when another read request is made, no additional expensive disk access occurs, but is returned from memory, speeding up the process. A wrapper class to provide the buffering functionality is BufferedReader. It is much quicker to user a buffer. This code snippet reads a while line at once.

```
public static void readingWithBuffering () throws IOException
{

      FileReader in1 =null;
      BufferedReader br= null;
      try
      {
            in1=new FileReader("Data.txt");
            br= new BufferedReader(in1);
            String x;
            while ((x=br.readLine()) != null)
            {
                  System.out.println(x);

            }
      }finally
      {
            if (br !=null)
                  br.close();
      }
}
```

Listing 4 Buffering Character Oriented File IO,

In Listing 4 a comparison is made with null rather than -1. The readLine() function returns a string or null, whereas the read() function returns a byte or -1 if unsuccessful. Because the basic InputStreams and Readers (and corresponding output versions) all propagate close and flush to their wrapped streams, there is no need to close or flush the inner objects. It is sufficient to close the outermost wrapper class as was done in Listing 4. It was not necessary to close `in1`.

There are a number of classes that can be wrapped around the basic InputStream and Reader classes. (as well as their corresponding output versions). Figure

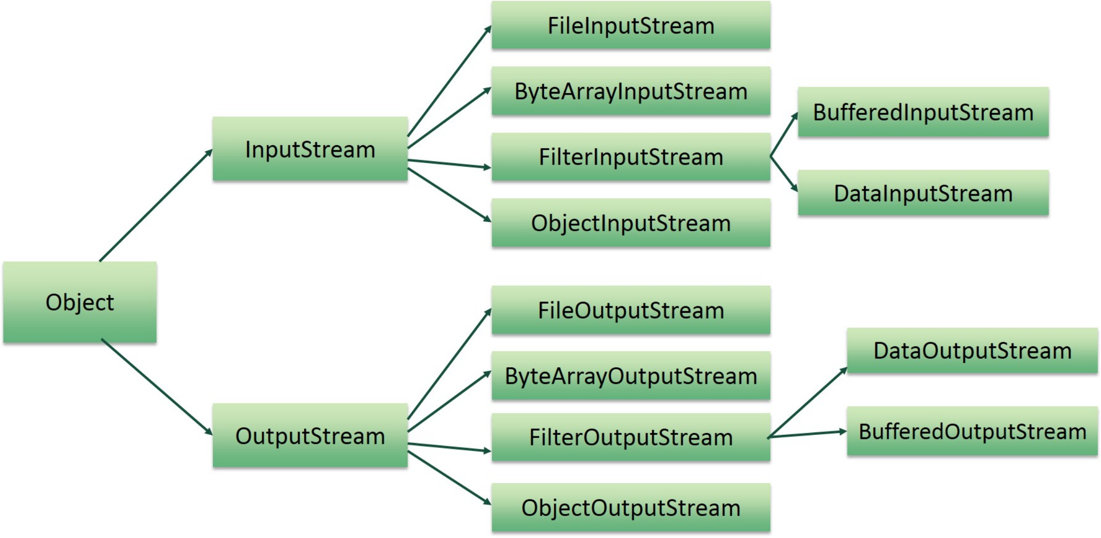
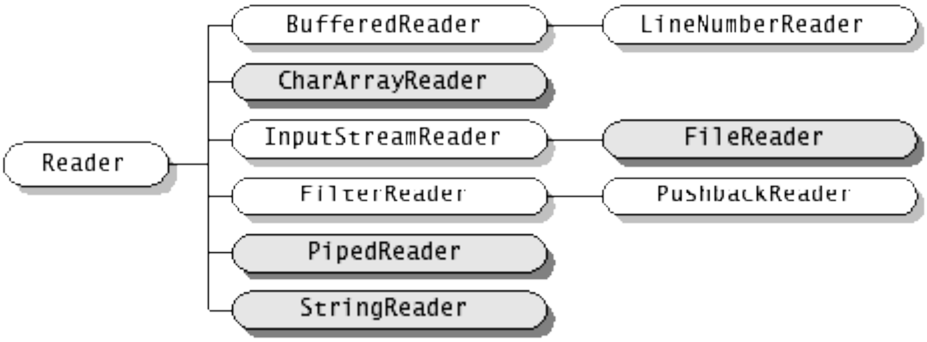


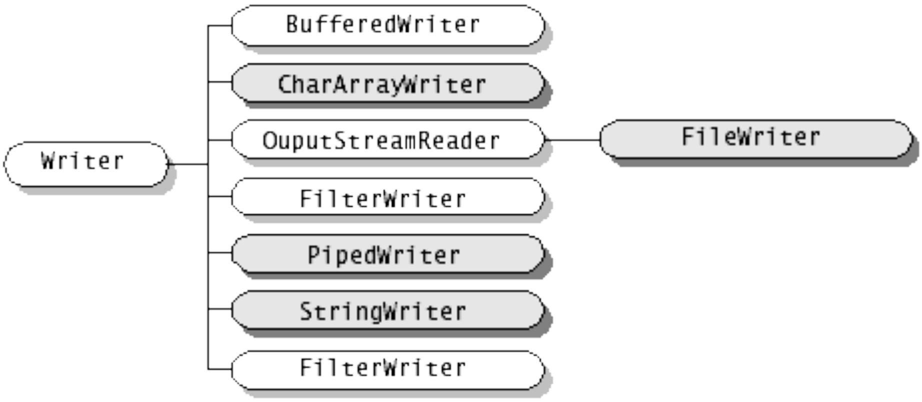
Figure 1 Hierarchy of InputStream Objects,

Source: tutorialspoint.com[1]

Figure xx shows the hierarchy for Reader and Writer Character Streams. These are 16 bit unicode streams.



[1] See the official java.io hierarchy at https://docs.oracle.com/javase/7/docs/api/java/io/package-tree.html



There are two sets of hierarchies. The Streams can be wrapped in other streams and so can the character streams be wrapped in other character streams in order to treat them differently. For example, to make use of a PrintWriter, because it has the println() function, but to write to a file, one could create

```
            FileWriter fw = new FileWriter("testfw.txt");
            PrintWriter pw = new PrintWriter(fw);
```

Or in one step

```
      PrintWriter pw=new PrintWriter(new FileWriter("testfw.txt"));
```

Two classes are particularly noteworthy. The InputStreamReader and the corresponding OutputStreamWriter. These two classes convert between the two distinct hierarchies.

Table xx lists the Character and Byte streams in alphabetic order and gives a brief description of their typical use cases.

Class Name	**Purpose**	
InputStream, OutputStream	Basic Type	Examples are System.in. This is the ancestor class for most input streams.
FileInputStream, FileOuputStream	File IO	Create a file for input or output as byte streams
ByteArrayInputStream, ByteArrayOutputStream		
RandomAccessFile		
BufferedInputStream, BufferedOutputStream		
FilterInputStream, FilterOutputStream		
PipedInputStream PipedOutputStream		
DataInputStream, DataOutputStream		
*PrintStream		

ObjectInputStream, ObjectOutputStream		
~~*SequenceInputStream~~ ~~*PushbackINpuStream~~ ~~*StringTokenizer~~		

Class Name	Description
Reader, Writer	
InputStreamReader, OutputStreamWriter	
CharArrayReader CharArrayWriter	
FileReader, FileWriter	
StringReader, StringWriter	
PipedReader PipedWriter	
BufferedReader BufferedWriter	
PrintWriter*	
FilterReader FilterWriter	
PushbackReader LineNumberReader	

Table xx highlights the key methods provided by the InputStream and Reader classes as well as some of their descendant objects.

Object	Frequently Used Method	Description

**Also categorize what objects write to files

Writing to a file

A number of classes can write directly to a file.

- The FileOutputStream can write to a file, byte by byte, using write() functions
- The FileWriter can write character streams byte by byte, or an array of bytes
- The PrintWriter can write to files using the familiar print() and println() methods. It can open a file directly if given a filename in its constructor

This example uses a PrintWriter class:

```
import java.io.*;
class FileIo{
    public static void main(String argv[]){
      PrintWriter pw=null;
      try{
            pw = new PrintWriter("test.txt");
            pw.println("done");
            pw.println("done");
            pw.close();
      }catch(Exception e){
            System.out.println(e.toString());
      }

    }
}
```

The println() method adds a newline character corresponding to the platform. Unix and Windows use a slightly different combination for new line characters.

The following example illustrates writing to a file using the file writer. It is wrapped in a BufferedWriter to improve its efficiency. The write method writes to the output stream immediately and is thus not efficient. The BufferedWriter class also has a newline() method that uses the appropriate new line character depending on the platform. A call to the flush() method sends the output to the file at once.

```
import java.io.*;
class FileIo{
    public static void main(String argv[]){
      FileWriter fw=null;
      BufferedWriter bw=null;
      try{
            bw = new BufferedWriter(new FileWriter("test.txt"));
            bw.write("done");
            bw.newLine();
            bw.write("done");
            bw.flush();
            bw.close();
      }catch(Exception e){
            System.out.println(e.toString());
      }

    }
}
```

In both cases, the file can be opened in append mode by adding a Boolean parameter set to true, when opening the file eg
bw = new BufferedWriter(new FileWriter("test.txt", **true**));

Some differences between PrintWriter and FileWriter:[2]

- FileWriter throws IOException in case of any IO failure, but none of PrintWriter methods throws IOException, instead they set a boolean flag which can be checked using checkError().
- PrintWriter has on optional constructor that can be used to enable auto-flushing when methods such as println(), print and format are called (rather then just when a newline character is encountered. FileWriter does not have that option.

[2] http://stackoverflow.com/questions/5759925/printwriter-vs-filewriter-in-java

- PrintWriter has a richer range of functions including print, printf println, write and format, but FileWriter mostly has write methods.
- When writing to files, FileWriter has an optional constructor which allows it to append to the existing file when the "write()" method is called.

Reading from the keyboard

The default input stream for a computer system is the keyboard. In java, it is represented as System.in. It is a byte stream. It can be read one byte at a time. However, it is move useful to buffer the reading and read whole lines (up to a carriage return) at a time. However, BufferedReader is a character stream while System.in is a byte stream. The InputStreamReader converts between the byte stream and the character stream. Listing 5 shows how this is accomplished.

```
public void KeyboardBasic()
{

try(BufferedReader       br       =       new       BufferedReader(new
InputStreamReader(System.in)) )
      {
            String inputText=null;
            inputText=br.readLine();
            System.out.println("what was entered was .."+inputText);


      }catch (Exception e){
            System.out.println("Error        -Other        Exception
"+e.toString());
      }
System.out.println("finished");
}
```

Listing 5 Reading Keyboard input with BufferedReader

The Scanner Class

It can be quick and convenient to parse text using the Scanner class. Scanner itself is not a stream, but can parse primitive types and strings that are separated by some delimiter. The default separator is white space. (Regular expressions may also be used). To use the scanner class, import java.util.Scanner.

Any InputStream can be read and processed in a similar manner by wrapping the steam source in an appropriate class. (that includes files, a network resource etc)

For example, Listing 6 makes use of the Scanner class and the InputStream System.in directly.

```
public static void KeyboardWithScanner() //throws IOException
{
  try(Scanner sc = new Scanner(System.in) )
      {
      String inputText=null;
      inputText=sc.next();
      System.out.println("what was entered was .."+inputText);
  }catch (Exception e){
      System.out.println("Error -Other Exception "+e.toString());
    }
    System.out.println("finished");
}
```

Listing 6 Reading Keyboard Input with Scanner class

The Scanner class has a number of functions for reading tokenized input, as illustrated in Listing 7. Table 1 lists some commonly used scanner methods.

```
      Scanner sc = new Scanner(System.in);
      int i = sc.nextInt();
      System.out.println(i);
      String s = sc.next();
      System.out.println(s);
      while (sc.hasNextInt()) {
            int anInt = sc.nextInt();
            ...
            }
```

Listing 7 Some common Scanner methods

Method	**Description**
String next()	Returns the entire string including tokens
String next(Pattern p)	Returns a token if it matches the pattern p
int nextInt()	Returns the next integer, similar functions exit for nextFloat(), nextBoolean() etc each returning the corresponding type eg float and boolean in this case.
boolean hasNextInt()	Returns true if the next token is an int. similar functions exist for other primitive types eg hasNextFloat(), hasNextLong() etc.
String nextLine()	Returns the entire line and advances to the next line

Table 1 Common Scanner functions

You can use the Scanner class with FileInputStream or FileReader as well. Eg
Scanner sc= new Scanner(new File("filename.txt"));
The following example illustrates reading with a FileWriter

```
import java.io.*;
import java.util.*;

class FileIo{

    public static void main(String argv[]){
      Scanner sc=null;
      try{
            sc= new Scanner(new FileReader("test.txt"));
            while (sc.hasNext())
            System.out.println(sc.next());

            sc.close();
      }catch(Exception e){
            System.out.println(e.toString());
      }

    }
}
```

Reading from or writing to other streams

Two Streams are particularly useful when writing objects to storage that can accept only key value pairs (eg certain mobile storage formats), or when communicating over the networks (eg via HTTP POST or retrieving data from an HTTP GET request. It is generally useful to create to a byte array output stream. They array of bytes can then simply be returned by calling toByteArray(). The DataOutputStream provide convenient methods for also writing bytes to a stream. The ObjectOutputStream writes primitive data types and objects that implement the Serializable interface to an OutputStream.

ByteArrayInputStream/ ByteArrayOutputStream

Data can be held in an array and parsed as an array of bytes of data. To read from such a stream, the ByteArrayInputStream can be used to read from the memory based array of bytes. The converse is to use ByteArrayOutputStream to write out data into an array. There are many use cases for this. The purpose may be to write to a resource that only accepts byte data, and perhaps stores it as a key value pair. All the data can be written together into an array of bytes before commiting to disk as one entity.

The following example shows how to use the ByteArrayOutputStream and its counterpart. It may be useful when it is necessary to save an object as an array of bytes. After the stream is created, it can be treated like any other stream. BufferedReader could be have been wrapped around it (with the help of OutputStreamReader) for example.

```
public static byte[] convertToBytes(Object obj) throws IOException {
    try(ByteArrayOutputStream baos = new ByteArrayOutputStream()){
        try(ObjectOutputStream oos = new ObjectOutputStream(baos)){
            oos.writeObject(obj);
        }
        return baos.toByteArray();
    }
}

public static Object bytesToObject(byte[] bytes) throws IOException,
ClassNotFoundException {
    try(ByteArrayInputStream bais = new ByteArrayInputStream(bytes)){
        try(ObjectInputStream ois = new ObjectInputStream(bais)){
            return ois.readObject();
        }
    }
}
```

Listing 8 Using the ByteArray Streams

Note that the ByteAraryOutputStream and its counterpart only write to an array held in memory.

ObjectOutputStream

If data need not be accessed in a text editor, then saving as a binary file is efficient (in terms of space used and access speed). The ObjectOutputStream and its complement have methods to write primitive types and to write objects. The methods include writeBoolean, writeInt, writeFloat, writeDouble etc which write the corresponding primitive types to the output stream. writeUTF writes a string to the stream using the UTF-8 encoding. UTF-8 supports more characters than the ASCII code does. ObjectOuputStream itself does not have a constructor that takes a file stream, hence the FileOutputStream class is employed.

This is an example from the API documentation:

```
      FileOutputStream fos = new FileOutputStream("t.tmp");
      ObjectOutputStream oos = new ObjectOutputStream(fos);

      oos.writeInt(12345);
      oos.writeObject("Today");
      oos.writeObject(new Date());

      oos.close();
```

Serialization

Objects cannot ordinarily be saved directly to a file. However, when the object implements the java.io.Serializable interface, the entire object can be written do a stream in one unit. It may be saved to disk or may be written to the network. A caveat is that its member variables must all be serializable also. Generally, primitive types are serializable.

Using the ObjectOutputStream, the object o can thus be written:

```
ObjectOutputStream oos = new ObjectOutputStream(someOutputStream);
                  out.writeObject(o);
```

To read back from the stream, use

```
ObjectInputStream ois = new ObjectInputStream(someInputStream);
     o = (SpecificObjectType) ois.readObject();
```

It is important to cast the variable back to the expected type, and to try to catch a ClassNotFoundException during the catch. Listing 9 illustrates how to read and write serializable data. The output stream is a file in this case, but it could have been a network resource.

```
//imports
public class tester
{
    static class Student implements java.io.Serializable{
        String surname;
        int age;
        float GPA;
    }

    public  void writeSerializableData() throws Exception
    {
        Student s1=null;

        s1= new Student();
        s1.surname="Mensah";
        s1.age=44;
        s1.GPA=4;

        try
        (FileOutputStream fos =
         new FileOutputStream("serialized.txt")){

            try(ObjectOutputStream out =
                   new ObjectOutputStream(fos)){
             out.writeObject(s1);
```

```
                System.out.printf("Saved!");
            }catch(IOException e)
            {
                System.out.println(e.toString());
                e.printStackTrace();
            }

        }catch(Exception e)
        {
            System.out.println(e.toString());
            e.printStackTrace();
        }
    }

    public void readBack(){
      Student s=null;
      try(   FileInputStream fis =
                new FileInputStream("serialized.txt")){
       try(ObjectInputStream ois = new ObjectInputStream(fis)){
        s = (Student) ois.readObject();

    }catch(IOException e){
        System.out.println(e.toString());
        return;
    }

   }catch(ClassNotFoundException c){
    System.out.println("Student class not found\n"+c.toString());
    return;
   }catch(Exception e){
       System.out.println(e.toString());
       return;
   }
   System.out.println("Student Info is ");
   System.out.println("Name:  " + s.surname);
   System.out.println("Age:  " + s.age);
   System.out.println("GPA:  " + s.GPA);

   }

   public static void main (String args[]) throws Exception
   {
       tester d= new tester();
       t.writeSerializableData();
       t.readBack();

   }
   }
```

Listing 9 Reading and Writing Serialized Objects

DataInputStream/DataOutputStream

This is a convenience class for reading and writing mostly primitive types to a stream, just like ObjectOutputStream and its complement do. (The ObjectOutput/Input streams appear to be a better option.)

```
DataOutputStream ds=
            new DataOutputStream(new FileOutputStream("myFile.txt"));
ds.writeBytes("---starting lines---");
for (int i=0; i<10; i++)
    ds.writeBytes("\n"+i + " Some repeated data");
ds.flush();
ds.close();
```

Other DataOutputStream methods include writeBoolean, writeInt, writeFloat, writeDouble etc which write the corresponding primitive types to the output stream. writeUTF writes a string to the stream using the UTF-8 encoding. writeBytes as above writes the string given as a sequence of bytes to the stream.

Standard Input, Output and Err

On a computer system, the standard InputStream is the keyboard, and the standard output is the monitor. Error output can be sent to another output stream "standard error". Standard Error is written to as System.err.println("This will print on the screen always, even if standard out is redirected elsewhere");

Error Handling

A good strategy to handle errors related to FileIO is to follow a simple scheme.

```
acquireResource();
try {
    use();
} finally {
    release();
}
```

Do not put the acquireResource() within the try block, nor place additional statements between the acquire() and try (other than simple assignment statements). Also at the end of the code, do not release multiple resources in a single finally block. Do handle the errors that may be thrown at a different level from where the exception occurred. The code in listing **below** in demonstrates catching additional kinds of errors.

```
public void goodPractice()
{

      try{
            FileWriter fw= new FileWriter("outputext.txt");
            try{
                  PrintWriter pw = new PrintWriter(fw);
                  for (int i=0; i<10; i++)
                        pw.println("this is line "+i);

                  }finally{
                        fw.close();
                  }
            }catch (FileNotFoundException e){
                  System.out.println("Error       finding       file
"+e.toString());
            }catch (IOException e){
                  System.out.println("Error       IO       Exception
"+e.toString());
            }catch (Exception e){
                  System.out.println("Error     -Other     Exception
"+e.toString());
            }
}
```

Listing 10 Error handling using nested try statements

An alternative strategy is demonstrated in Listing 4 where a finally block is used. One must always check if the resource is not null before closing it. This is different from the structure in Listing 10.

Try-With-Resource

Starting with Java 7, the try-with-resources that simplifies error handling has been introduced. The format is

```
try (resource) {
…
}
```

Any resources open in the try() will be closed automatically after the block ends, whether or not the try block finishes properly. It means that a finally block is no longer needed.

**Suppressed errors[3]

```
public void java7AndAbove()
{
try(PrintWriter pw = new PrintWriter(new FileWriter("testfw2.txt")))
{
      for (int i=0; i<12; i++)
            pw.println("this is line "+i);
}catch (FileNotFoundException e){
      System.out.println("Error finding file "+e.toString());
}catch (IOException e){
      System.out.println("Error IO Exception "+e.toString());
}catch (Exception e){
      System.out.println("Error -Other Exception "+e.toString());
}
}
```

Listing 11 Try-with-recource example

Examples

The following code snippet makes use of Scanner to read and display content from a file. Either the commented line with FileInputStream or the next line with FileReader will work perfectly. One or the other can be commented out.

```
import java.io.*;
import java.util.Scanner;
class TT{
      public static void main(String a[]){
       try{
            //FileInputStream ff=new FileInputStream("TT.java");
            FileReader ff=new FileReader("TT.java");
            Scanner s= new Scanner(ff);
            while (s.hasNext()){
                  System.out.println(s.nextLine());
```

[3] https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

```
            }
        }catch (Exception e){
                  System.out.println(e.toString());
        }
      }
}
```

Listing 12 Reading from a file using Scanner

The code snippet in Listing 13 reads and the bytes of a file and displays them with buffered reader (readFile), next the file is read and displayed but no use of buffering is employed(unbuffIn), and then it uses printWriter to write to a file (iofxn)

```
import java.*;
import java.io.*;
public class fileIO
{
      /**
      * read and display from a file onto console
      */
      public static void readFile() throws Exception
      {
            System.out.println("Reading from Delme.java");
            BufferedReader in1 =
            new BufferedReader(new InputStreamReader(new 
FileInputStream("Data.txt")));
            String x=null;
            while ((x=in1.readLine()) !=null)
            {
                  System.out.println(x);
            }
      }


      //also reading a file, one byte at a time, not buffering
      public static void unbuffIn() throws IOException
      {
            InputStream in1 =new FileInputStream("Data.txt");

            System.out.println("size ="+in1.available());
            int x;
            while ((x=in1.read()) != -1)
            {
                  System.out.print((char)x);
            }
            in1.close();
      }

      //wrirint a string repeatedly into a file, a line number is 
prepended to the written string
      public static void iofxn() throws IOException
      {
            try {
                  String s= "My test string";
                  PrintWriter out1 =  new PrintWriter(new 
BufferedWriter( new FileWriter("Output.txt")));

                  int lineCount = 1;

                  for (int i=0;i<10; i++)
                        out1.println(lineCount++ + ": " + s);
```

```
                out1.close();
            } catch(EOFException e) {
                System.err.println("End of stream");
            }

        }


        public static void main (String args[])
        {
            try
            {
                readFile();
                iofxn();
                unbuffIn();
            }catch (Exception e)
            {
                System.out.println(e.toString());
            }
        }
}
```

Listing 13 A variety of file I/O interactions

The File Object

The File object is in the java.io package. The File object is not the content of the file itself but provides information about a file, a path or a directory. Instantiating the File object does not create a physical file one can write to. It can however be used in conjuction with other classes such as FileWriter to create a physical file on disk. For example instead of passing a filename (as a string) to FileWriter, a File Object can be passed instead.

Eg

```
BufferedReader br=
      new BufferedReader(new FileReader(new File (“inputfile.txt”)));
```

The program in Listing 14 explores the file system using the attributes of the FILE object. The methods calls are quite self explanatory. For example, canRead() returns true if the current user has read permissions on the file or folder in question. Length() returns the file size if the File object is a file or the number of files enclosed if it is a folder. Calling list() on the object, if it is folder returns the array of File Objects for each file or directory within the folder.

```
import java.io.*;
class dirLister
{
public static void main(String args[])
{
      //this program prints file or directory attributes
      File dir =new File("sub");
      try
      {
            System.out.println(
                  "name= "+dir.getName()+ "\n"+
                  "canRead "+ dir.canRead() +"\n"+
                  "canWrite "+ dir.canWrite()+ "\n"+
                  "exists "+ dir.exists() +"\n"+
                  "getAbsolutePath "+ dir.getAbsolutePath() +"\n"+
                  "getCanonicalPath "+ dir.getCanonicalPath()+"\n"+
                  "getName "+ dir.getName() +"\n"+
                  "getParent "+ dir.getParent() +"\n"+
                  "getPath "+ dir.getPath() +"\n"+
                  "isAbsolute "+ dir.isAbsolute() +"\n"+
                  "isDirectory "+ dir.isDirectory()+ "\n"+
                  "isFile "+ dir.isFile() +"\n"+
                  "lastModified "+ dir.lastModified() +"\n"+
                  "length "+ dir.length()+"\n"
                  );


            if (dir.isDirectory())
            {
                  String dirList[]= dir.list();
                  for (int i=0;i<dirList.length;i++)
                        System.out.println(dirList[i]);
            }

      }catch(IOException e)
      {     System.out.println("error "+e.toString());}

}
}
```

Listing 14 File and Directory properties using the File Object

Absolute Path and Cannonical Path deserve some comment. Canonical path to a file is the single, unique actual path. There may be many absolute paths to a file (depending on the system). Eg on UNIX, /usr/local/../bin is identical to /usr/bin. The CanonicalPath is not ambiguous. It is /usr/local

Paths are specified generally with a forward slash on both windows and UNIX type systems. Eg “C:/users/myname/myfolder/myfile.txt”. Alternatively, for windows a double backslash can be used eg
“C\\users\\myname\\myfolder\\myfile.txt”. To make the application platform independent, consider using File.separator instead eg
“C:”+ File.separator+ “users” + File.separator + “myname” + File.separator +
“myfolder” + File.separator +”myfile.txt”

Network I/O

Opening the URL InputStream

The URL object in the java.net package enables the creation of a URL object from a string version of the URL. Calling the openStream() method on the URL object returns an InputStream which can be read from as any other InputStream. Listing 15 illustrates opening and printing out in the console the content at a given URL. It wraps the opened InputStream in a BufferedReader.

```
import java.io.*;
import java.net.*;

public class OpenAndCopyURL
{
  public static void main(String[] args) {

    try {
      URL url =
      new URL("http://www.ashesi.edu.gh");

      InputStream urlIN = url.openStream();
      BufferedReader br = new BufferedReader(new
InputStreamReader(urlIN));

      String line;
      while ((line = br.readLine()) != null)
        System.out.println(line );
    } catch(Exception ex) {
      System.out.println(ex.toString());
    }
  }
}
```

Listing 15 Opening and reading from a URL InputStream

The output of this piece of code will be identical to what will be obtained if the webpage at the given address is open and the user "views the source" of the page. This is an option that can typically be accessed in a browser by right clicking on the page and choosing view source. It represents the HTML code for forming the page. Hence one will not see images for example, but the HTML tags that embed images.

Using a URLConnection instance

The openConnection() methods of the URL object returns an instance of the URLConnection object. URLConnection is itself an Abstract ancestor class. The object returned allows the opening of both an InputStream and an OutputStream by calling getInputStream() or getOutputStream() on that returned object.

```
import java.io.*;
import java.net.*;
import java.util.*;

public class UsingURLConnectionAndProperties
{
  public static void main(String[] args) throws Exception {

    try {
      URL url =
      new URL("http://www.ashesi.edu.gh");

      URLConnection urlCon = url.openConnection();
      InputStream in= urlCon.getInputStream();
      BufferedReader br =
            new BufferedReader(new InputStreamReader(in));
      String line;
      while ((line = br.readLine()) != null)
        System.out.println(line );
      br.close();


    } catch(Exception ex) {
      System.out.println(ex.toString());
    }
  }
}
```

Properties of a connection

Just like files, URLs also have properties that can be accessed. First of all, the URL object has methods that allow the user to split the given URL into its component parts including the host name, the port number, the protocol and the file. Other functions enable the printing of the full URL format. The latter may be useful if a constructor was used that accepts the individual elements of the URL.

```
import java.net.*;
public class urlAttribs
{
public static void main(String args[]) throws Exception {

URL mURL = new URL("http://campus:80/home.htm");
        System.out.println("protocol = " + mURL.getProtocol());
        System.out.println("host = " + mURL.getHost());
        System.out.println("filename = " + mURL.getFile());
        System.out.println("port = " + mURL.getPort());
        System.out.println("ref = " + mURL.toExternalForm());
            //returns the string representation of URL

}
}
```

Listing 16 Obtaining properties of a URL

The URL connection too has properties such as the type of size of file to “downloaded”, the date it was last modified etc. Content length will be -1 if file size is unknown.

```
System.out.println("ContentType " + urlCon.getContentType());
Date x= new Date(urlCon.getDate());
System.out.println("Date " + x);
System.out.println("**Date " + urlCon.getDate());  //returns long
x= new Date(urlCon.getLastModified());
System.out.println("Last Modified " + x);
System.out.println("Content Length"+ urlCon.getContentLength());
```

Listing 17 Obtaining properties of a URLConnection

Example output of this snippet is

```
ContentType text/html; charset=utf-8
Date Thu Jun 23 09:07:23 GMT 2016
**Date 1466672843000
Last Modified Thu Jun 23 09:07:24 GMT 2016
Content Length-1
```

Figure 2 Sample output for URLConnection Properties.

*If content type is text, getContent() can return the full text of the html source as a String object

Random Access Files

The files previously considered are sequential in nature. In read mode, you can only read line 2 after reading line 1. There is no option to read line 100 even if you do not need the first 99 lines. Also you cannot read and write at the same time. Random access files remove that limitation. You can read any section of the file at any time, and the file can be open in read/write mode. The constraint however is that the file created is not a text file and is thus not human readable. A random access file can be considered like a large byte array on disk.

A file is open in read only, read-write, or rw with immediate write to storage. The RandomAccessFile class can be found in the java.io package. The constructor is of the form

```
      RandomAccessFile(String name, String mode)
```

With random access files, you "seek" a position (in bytes from the start of the file) in the file and read or write at that point. The current position can be obtained by getFilePointer(). The initial position after opening a file is position 0. Writing to the file starts at the current file pointer location. It will extend the file if the amount of data exceeds the current file size. Reading a larger block of data starting from a file pointer than there are bytes available will result in an EOFException. The total length of a file can be determined by the length() method.

With RandomAccessFile, one can write bytes, and primitive types directly. Strings are accessed with readUTF() and writeUTF(); Also bytes can be written with write(int b) or write(byte b[]); An exception will be thrown if the data cannot be read correctly. Endeavour to read back data in the same format as was originally written. Listing 18 shows how to write to a file, read back from it, seek a specific position to make a change, and then proceeds to read and display the entire file. The writeToFile() method looks for the end of the file and adds on to it always. The modifyFile() only looks for a particular record at the beginning and changes that one.

```
import java.*;
import java.io.*;
public class randomFileIO
{


  public static void writeToFile() throws Exception
  {
    System.out.println("writing to randIO.txt");
    RandomAccessFile raf = new RandomAccessFile("randIO.txt", "rw");
    raf.seek(raf.length());
    System.out.println("current length = "+raf.length());
    raf.writeInt(100);
    raf.writeUTF("This is intermediate Programming");
    raf.writeDouble(14.4);
    raf.writeInt(20);
    raf.close();
  }


//also reading a file, not buffering
  public static void readFromFile() throws IOException
  {
    System.out.println("Reading from randIO.txt");
    RandomAccessFile raf = new RandomAccessFile("randIO.txt", "rw");
    int i= raf.readInt();
    String s= raf.readUTF();
    double d= raf.readDouble();

    raf.close();
    System.out.printf("int is %d string is %s and double is
%3.2f\n\n", i, s, d);
  }

//also reading the whole file
  public static void readAllFromFile() throws IOException
  {
    System.out.println("Reading entire file from randIO.txt");
    RandomAccessFile raf = new RandomAccessFile("randIO.txt", "rw");

    try{
     while( true ){
      int i= raf.readInt();
      String s= raf.readUTF();
      double d= raf.readDouble();
      int ii = raf.readInt();  //has not been used
      System.out.printf("int is %d string is %s and double is
%3.2f\n\n", i, s, d);
    }
  }
  catch(EOFException eof){
    System.out.printf("Reached file end\n");
  }
  finally{
    raf.close();
  }

}


public static void modifyFile() throws IOException
{
  RandomAccessFile raf = new RandomAccessFile("randIO.txt", "rw");
  raf.seek(0);
  raf.writeInt(3333);
  raf.close();
  System.out.println("modifying randIO.txt");


}


public static void main (String args[])
{
  try
  {

    writeToFile();
    readFromFile();
    modifyFile();
    readFromFile();
    readAllFromFile();

  }catch (Exception e)
  {
    System.out.println(e.toString());
  }


}

}
```
Listing 18 Random Access File manipulations

Exercises and Labs

FileIO

Lab 1.
AshesiMain has a point of sale program that records transactions into the *sales.dat* file. This is a plain text file. The sales file contains the product name (a string) on one line followed by the sale value (a number) on the next line. Management wants to use excel to get a sum of the sales. However, since the string are interspersed with numbers, excel will not handle this well. The plan is to create a new file which has only the numbers in it. Write a **java** program that reads the sales file (*sales.dat*) and writes out only the sale amounts in a second file (named *onlynumbers.txt*) Add a blank line after the last entry. Next add the sum of the sale values. There will thus be no need for excel.

Sample contents of *sales.dat*: [note, file length is unknown]
Rice
2.0
Jam
4.0
Sugar
1.2
....
Restrictions: you must make use of FileWriter and FileInputStream classes

Lab 2.
Ashesi Kindergarten needs software to keep records on its pupils. Students take only 3 subjects: Art, Reading and Writing. There can be only up to a 100 pupils.

The program has a simple menu that allows the teacher to add a new student (including their grades), print out all student records, and print out the averages for each of the subjects. The last menu option exits the program. For the program to be useful to the teacher, it must be able to save the records previously entered, so that when the program is started again another time, the old data will still be available in the program.

Write a program to accomplish this. Hint: You need to save the data in a file before the program exists, and you need to read the file when the program starts. No user intervention will be required. Look at the **File** object (in the java documentation) to find out how to determine if a file exists already.

You may add extra functions beyond these for extra credit. Extra credit will only be awarded if the required functions have already been implemented.

Lab 3:
Write a program that lists all the files in a given folder. If any are folders, it should recursively list the contents of those folders. The output of the program should be saved in a file. You may format the display/data as a tree, or use another appropriate means to clearly show the folders and their contents hierarchically.

Further work:
Enable to user to provide command line parameters to

- Specify which folder's content are to be accessed.
- Specify whether folders should recursively be accessed or not.
- Allow user to print out only filenames of .java files.
- Allow user to specify output file name,
- Allow user to specify screen printing only, or screen printing and file output, or file output only.
- Allow user to append to existing file or overwrite existing file
- Allow user to make a backup copy of an existing file before creating a new output file. (you can save the existing file with a .bak extention eg oldfilename.bak).
- Allow user to create a folder (if it does not exist) and store the output file in that folder.

Lab 4.
Ashesi academic records are exported for the FOCUS software in a format as follows:

```
20161001
Addo, John Sule
Year Group: 2016,Current GPA: 3.15
Courses Taken:
Text And Meaning: 82.5
Calculus 2: 77.2
Physics II :  33.5
======

10251002
Bola, Mary Adu
Year Group: 2020,Current GPA: 3.51
Courses Taken:
French: 82.5
Calculus 2: 77.2
Intermediate Computer Programming: 97
African Literature :  33.5
Competitive Strategy:  83.5
FDE :  63.5
```

```
======

20201002
Mansah-Musa, Yaa Asantewa
Year Group: 2019,Current GPA: 3.75
Courses Taken:
Text And Meaning: 82.5
Physics II :  33.5
======
```

The registry needs to use Excel to analyse some data. They would like to open a CSV file containing data in the following order:
FirstName OtherNames Surname, IndexNo, GPA, NumberOfCoursesTaken.
The CSV file should have a .csv extention, and each item should be placed within quotes and separated by commas. The sample data above will consequently be saved as follows:

```
“Name”,”Index Number”,”GPA”, “No. of Course”
“John Sule Addo”, “20161001”, “3.15”, “3”
“Mary Adu Bola”, “10251002”,  “3.51”, “6”
“Yaa Asantewa Mansah Musah”, “20201002”, “3.75”, “2”
```

As the newly hired IT person, analyse the source data and the desired report format and write a program to create the desired report from the source data.

Network IO

Lab 1.

In order to optimize online time by student computers, you have been tasked to write software that can download the pages of a website given its home address. The first step of the process is to determine all the links on that first page.

Actions:

i. write a function which, when given a url can download the web page and save the page as a local file. You may save the file with its original name or use a new name if one is provided. (ie if **`localFileName`** is not null). The function name should be
`public void downloadPage(String URL, String localFileName);`

ii. write a function that parses the saved file and extracts all the URLs in that file and appends the URLs to a file of URLS. See the API for how to append.

`public void parseFile(String localFileName, String urlFile);`

iii. write a function that displays the content of the saved file of URLs

`public void displayFile(File localFileName);`

iv. write a program that makes calls to these functions
Extra Credit:
Extend the program so that all web files that are local, relative to the index page are downloaded and saved by their original names (that will make a copy of the site). No need to process more than the index page.

Hints: take a look at the source for the page you wish to download. Locate href=..