BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
CS F213
LAB 8

AGENDA

TIME: 02 Hrs

- Byte and Character streams
- Predefined streams (System in, out, err)
- File input and output

8.1 Byte and Character streams:

In our previous labs, we have mostly worked with hardcoded data in the programs. But most of the real-world applications are interactive, that is, they take some input from the user through input devices like keyboard, files, etc. and after processing display the output on monitors, speakers, files, etc. Considering the above point, we would learn about byte and character streams in this section. Let us understand each of them in detail.

8.1.1 What are streams?

A stream is an object that enables the sequential flow of data between a program and some I/O device or file. Different streams behave similarly even if the input/output devices may be different, thus providing ease of access to the user. Streams are broadly classified into two categories:

- If the data flow into a program, then the stream is called an input stream
- If the data flow out of a program, then the stream is called an output stream

In our second lab session, we learned about reading and writing data to the console. The data was read from the keyboard using input stream - System.in and written back to the console using output stream - System.out

```
Scanner in = new Scanner(System.in);
```

```
System.out.println("Hello world!");
```

8.1.2 Byte streams

Byte streams are used to perform input and output of 8-bit bytes. It is suitable for processing raw data like binary files, video, audio, etc.

All the classes related to the byte streams are available in the java.io package. There are two abstract classes provided namely:

- InputStream: for byte-based input operations
    - Various classes inherit the above class and override its methods - BufferedInputStream, FileInputStream, DataInputStream, ByteArrayInputStream
- OutputStream: for byte-based output operations
    - Various classes available - BufferedOutputStream, FileOutputStream, DataOutputStream, ByteArrayOutputStream

Let us see how we can read and write to a local file.

Program 8.1.a: Reading from a file using Byte stream

```
import java.io.*;
public class ByteReader{
	public static void main(String[] args)throws IOException{
		FileInputStream fileInputStream = new FileInputStream(new
File("input.txt"));//enter appropriate file location
		BufferedInputStream input = new
BufferedInputStream(fileInputStream);
		try {
			char c = (char)input.read();
			System.out.println("Read:' " + c + " 'from the file");
		}
		catch(Exception e) {
			System.out.println(e);
		}
		finally {
			input.close();
		}
	}
}
```

Program 8.1.b: Writing to a file using Byte stream

```
import java.io.*;
public class ByteWriter{
	public static void main(String[] args) throws IOException {
		String data = "We are learning to use byte streams.";
```

```
            BufferedOutputStream out = null;
            try {
                  // If a file does not exist at the specified location,
                  // a new file would be automatically created
                  // else, the content of the existing file would be
overridden
                  FileOutputStream fileOutputStream = new
FileOutputStream(new File("output.txt"));
                  out = new BufferedOutputStream(fileOutputStream);
                  out.write(data.getBytes());
            }
            catch(Exception e) {
                  System.out.println(e);
            }
            finally {
                  out.close();
            }
      }
}
```

Note:
It is always a good practice to close the streams if they are not required further to prevent any memory/resource leaks.

8.1.3 Character streams

In Java, characters are stored using Unicode conventions. Each character corresponds to a specific numeric value. Character stream allows us to read and write 16-bit Unicode characters by automatically translating the Unicode set to and from the local character set. They are useful for processing text files character by character.

Similar to the byte streams, all the classes related to the character streams are available in the java.io package. There are two abstract classes provided namely:

- Reader: for 16-bit character stream-based input operations
    - Various classes inherit the above class and override its methods - BufferedReader, FileReader, InputStreamReader, CharArrayReader
- Writer: for 16-bit character stream-based output operations
    - Various classes available - BufferedWriter, FileWriter, OutputStreamWriter, CharArrayWriter

In our previous labs, we have already seen reading inputs using InputStreamReader.

```
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader in = new BufferedReader(isr);
String name = "";
name = in.readLine(); //read a string
```

Now, let us see how we can use character streams instead of byte streams to read/write a file.

Program 8.1.c: Reading from a file using Character stream

```
import java.io.*;
public class CharacterReader{
	public static void main(String[] args) throws IOException {
		Reader input = null;
		try {
			input = new FileReader("input.txt");
			char c = (char)input.read();
			System.out.println("Read: '" + c + "' from the file");
		}
		catch(Exception e) {
			System.out.println(e);
		}
		finally {
			input.close();
		}
	}
}
```

Program 8.1.d: Writing to a file using Character stream

```
import java.io.*;
public class CharacterWriter{
	public static void main(String[] args) throws IOException {
		Writer out = null;
		String data = "We are learning to use character streams.";
		try {
			out = new FileWriter("output.txt");
			out.write(data);
		}
		catch(Exception e) {
			System.out.println(e);
```

```
            }
            finally {
                  out.close();
            }
      }
}
```

From the above examples, the read/write classes for byte and character streams are almost similar. The main difference is that the byte stream classes use FileInputStream/FileOutputStream while the character stream classes use FileReader/FileWriter. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time. Thus, character streams are often considered "wrappers" for byte streams. The character stream uses the byte stream to perform the physical I/O, while the character stream handles the translation between characters and bytes.

Note:
An easy way to identify the stream being used in a program is:

- Byte streams typically end with InputStream/OutputStream
- Character streams end with Reader/Writer

8.1.4 Exercises

1. Write a program to read user data from the console and write it to a file using byte streams.
2. Write a program to copy the content of an existing file to another file using character streams.
   Hint: You would have to read the content of the source file character by character and write similarly. read() call on the reader object would return -1 when the end of the file is reached.

8.2 Predefined Standard Streams

In the *java.lang* package, a **System** class is defined which provides several functionalities related to the runtime environment such as **standard streams**; loading files and libraries; access to externally defined properties and environment variables; and utility methods. Our focus here is the three predefined stream variables : **in, out and err** available in System class.

Java's JVM (Java Virtual Machine) **automatically initializes all three predefined streams** on startup. Hence these streams are already open at the start of the program, and are ready to supply input data or accept output data. Since **predefined streams are byte streams**, we need to wrap them in character-based streams for character based IO operations in Java.

Some questions to think about:-

- **Ex 8.2.a** : Why predefined streams are byte streams and not character streams? (Hint: think whether character streams were supported in original specification of Java)

- **Ex 8.2.b** : Can you instantiate System class? Why or Why not?

8.2.1 Standard Input Stream

1) The standard input stream , ***System.in*** , typically corresponds to keyboard input or another input source specified by the host environment or user. By default, it is keyboard input.

```
public static final InputStream in

// wrapping standard input stream in BufferedReader object to get a
character based stream
BufferedReader br = new BufferedReader (new InputStreamReader (System.in)
);
```

2) **setIn** method in System class is used to redirect the standard input stream.

```
public static void setIn ( InputStream in) throws SecurityException
```

8.2.2 Standard Output Stream

1) The standard output stream , ***System.out*** , typically corresponds to display output or another output destination specified by the host environment or user. By default, this is the console.

```
public static final PrintStream out

// as you are already aware, standard way to write a line of output data:
System.out.println(data)
```

2) **setOut** method in System class is used to redirect the standard output stream.

```
public static void setOut(PrintStream out)  throws SecurityException
```

8.2.3 Standard Error Output Stream

1) The standard error output stream , ***System.err*** , typically corresponds to display output or another output destination specified by the host environment or user. By default, this is the console.

   The idea behind a standard error output stream, apart from standard output stream, is that the information such as error messages **that require immediate attention of the user** can be displayed via this stream. Particularly useful in a scenario where System.out has been redirected to a destination not continuously monitored by the user.

```
public static final PrintStream err


// typical way to use System.err in a program

try {

// Code for execution

}

catch ( Exception e) {

System.err.println ( "Following error in the code:- " + e );

}
```

2) **setErr** method in System class is used to redirect the standard error output stream.

```
public static void setErr(PrintStream err)  throws SecurityException
```

Note: Please read about SecurityException if interested.

Program 8.2.a

Let's write a Java program step by step, for taking integer inputs as long as ‘q’ (for quit) character is not entered. Then write the sum of all these numbers in an output file by redirecting standard output. Then we will again redirect the standard output.

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

   public static void main(String args[]) throws IOException {

// 8.2.a.1 declare a buffered reader object and initialise it to null


      int sum = 0 ;
      String temp_str;

      try {
      // 8.2.a.2 initialise br with standard input stream


      System.out.println("Enter integers, 'q' to quit.");

     // 8.2.a.3 read string in temp_str

     // 8.2.a.4 in while condition : continue as long as string is not
equal to "q".

      while( _________________ ){

       // 8.2.a.5 add parsed integer value in sum variable

       // 8.2.a.6 read next line in temp_str

      }


      }

      catch ( Exception e) {
      // 8.2.a.7 in case of any errors, print the error in standard error
output stream
```

```
      }


finally {

      PrintStream op = new PrintStream(new File("A.txt"));

      PrintStream console;
      // 8.2.a.8 Store current System.out in a PrintStream object before
assigning a new value


      // 8.2.a.9 Assign op to standard output stream


      // 8.2.a.10 Print the sum in A.txt

      // 8.2.a.11 Use stored value for output stream

      System.out.println("Accomplished the task!");

      if (br != null) {
           // 8.2.a.12 close the buffered reader : good practise !!

      }
      }
   }
}

/*
 Answers :
8.2.a.1 BufferedReader br = null;
8.2.a.2 br = new BufferedReader(new InputStreamReader(System.in));
8.2.a.3 temp_str = br.readLine();
8.2.a.4 ! temp_str.equals("q")
8.2.a.5 sum += Integer.parseInt( temp_str );
8.2.a.6 temp_str = br.readLine();
8.2.a.7 System.err.println ( "Following error in the code while doing IO:-
" + e );
8.2.a.8 PrintStream console = System.out;
8.2.a.9 System.setOut(op);
```

```
8.2.a.10 System.out.println("The sum of the numbers is : " + sum );
8.2.a.11 System.setOut(console);

8.2.a.12 br.close();
*/
```

8.3 File I/O

As we have already seen, the distinction between byte and characters streams, and explored the predefined streams objects in Java. In this section, we'll be looking at other I/O streams which will come in handy in developing some real-world applications.

8.3.1 PrintWriter

This PrintWriter can be used to write It is used to print the formatted representation of objects to the text-output stream.

In the below program, we have a Student class and PrintWriterExample class, which tries to use PrintWriter to print the details of students in a text file "output1.txt".

In line1, you could see that, PrintWriter class instantiation which accepting **System.out** as the Output Stream. In line2, we could also observe that it can also be used to link its stream with the highly used output stream for text-based data i.e. "**FileWriter**".

Program 8.3.a:

```
import java.io.PrintWriter;
import java.io.FileWriter;
class Student{
    String name;
    int year, age;
    Student(String name,int year,int age){
        this.name = name;
        this.year = year;
        this.age = age;
    }
}
```

```
public class PrintWriterExample {
    public static void main(String[] args) throws Exception {
        PrintWriter pw = new PrintWriter(System.out); //line1
        pw.write("Hey JavaBeans!\n");
        pw.flush();
        Student s1 = new Student("Teja",2020,22);
        pw.close();
        PrintWriter pw1 =null;
        pw1 = new PrintWriter(new FileWriter("output1.txt",true)); //line2
        pw1.write("BITS Pilani Students info \n");
        pw1.flush();
        //check the file contents
        Student s2 = new Student("Harshith",2020,22);
        pw1.write(s1.name+","+s1.year+","+s1.age+"\n"); //line3
        pw1.write(s2.name+","+s2.year+","+s2.age); //line4
      //pw1.print(s1); //line5
        pw1.flush();
        pw1.close();


        /*
        After running the above program!
        The contents of the output file will be of the following form:

                BITS Pilani Students info
                Teja,2020,22
                Harshith,2020,22



   Questions:
   1) What is the importance of a boolean argument "true" in creating
      an instance of the FileWriter in line2? Try to remove it and
      Run, What changes do you observe in the file contents?
   2) Replace the "write" method in the line3,4 with the "print" method.
      What changes do you see?
   3) Comment out line5, see what it is doing or adding to the
      File contents?
   4) After answering the 2nd and 3rd questions, as of now how many
      different signatures are there for the "print" method?

 */
```

```
    }
}
```

8.3.2 Scanner

In this example, we are trying to explore ways on how we can use **Scanner** to read inputs not only from the keyboard but also through other streams. In order to read data from a file, we can use a highly used input stream for text data i.e. "**FileReader**"

Program 8.3.b:

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

public class ScannerExample{
    public static void main(String[] args) throws Exception{
        Scanner sc = new Scanner(new FileReader("input2.txt")); //line1
        Scanner sc1 = null;
        PrintWriter pw = new PrintWriter(new FileWriter("output2.txt"));
        String newLine=null;
        String s="";
        while(sc.hasNextLine()){
         newLine = sc.nextLine();
         sc1 = new Scanner(newLine); //line2
         sc1.useDelimiter("/"); //line3
         s+= "Name:" + sc1.next() + "\n";
         s+= "Year: "+ sc1.nextInt()+"\n";
         s+= "Age:" + sc1.nextInt()+"\n";
         sc1.close();
         pw.write(s);
         pw.flush(); //line4
         s=""; //line5
        }
        sc.close();
        pw.close();

        /*
        For the above program, the contents of the input2.txt are
                Teja/2020/22
                Harshit/2020/22
                Surya/2020/24
                Krishna/2020/23
```

```
        Contents of output2.txt will be:
                Name:Teja
                Year: 2020
                Age:22
                Name:Harshit
                Year: 2020
                Age:22
                Name:Surya
                Year: 2020
                Age:24
                Name:Krishna
                Year: 2020
                Age:23



      Questions:
       1) What changes you can observe in the line1 and line2, i.e
          While creating Scanner instances?
       2) What is the importance of line3 and 4?
       3) How line5 is being useful in generating correct output?

         */
    }
}
```

8.3.3 Console

The **Console** class in java is used to get the inputs directly from the console. The java.io.Console class is attached to the system console internally.

The instance of the Console class is created using the static method called "**console**" in the System class.
In the below example, we are trying to gather the student names and ids in two separate files using Console class and PrintWriter classes.

Program 8.3.c:

```
import java.io.Console;
import java.io.FileOutputStream;
import java.io.FileWriter;
```

```
import java.io.IOException;
import java.io.PrintWriter;
public class ConsoleExample {
    public static void main(String args[]) throws IOException{
        Console c = System.console();  //line1
        PrintWriter studNameWriter = new PrintWriter(new
FileWriter("studname.txt"));
        PrintWriter studIdWriter = new PrintWriter(new
FileOutputStream("studId.txt"));
        System.out.println("Enter no. of students: ");
        int num = Integer.parseInt(c.readLine());
        for(int i =1;i<=num;i++){
            System.out.format("Enter Student%d name: ",i);
            studNameWriter.write(c.readLine()+"\n");

            System.out.format("Enter Student%d ID: ",i);
            studIdWriter.write(c.readLine()+"\n");

            studNameWriter.flush();
            studIdWriter.flush();
        }
        studIdWriter.close();
        studNameWriter.close();

        }
}

/*
Sneak peek:
There are some more available methods in the Console Class, some of them
are really helpful while you are trying to create "Login" functionality
from the command line and you don't want to display the typing password to
the user(same as when you are trying to give the password for sudo access
in your system through the terminal).

*/
```

8.3.4 SequenceInputStream

It is a special class, which helps us to read inputs from multiple streams one after the other, here we are reading inputs from “stduname.txt” and “studId.txt” which we created and populated

the data in the above program. In line1, we could see the syntax of how one can read inputs from two different input streams. (which is one of the available constructor's definitions)

Program 8.3.d:

```
import java.io.SequenceInputStream;
import java.io.FileInputStream;

public class SeqInputExample {
    public static void main(String args[]) throws Exception{
        FileInputStream input1=new FileInputStream("studname.txt");
        FileInputStream input2=new FileInputStream("studId.txt");
        //line1
        SequenceInputStream inst=new SequenceInputStream(input1, input2);
        int j;
        while((j=inst.read())!=-1){
         System.out.print((char)j);
        }
        inst.close();
        input1.close();
        input2.close();
       }
}

/*
  Let's say the contents in studname.txt are:
     Geetika
     Teja
     Arjun

 In studId.txt are:
     2020H1030112P
     2020H1030113P
     2020H1030114P

  The output on the console is:
      Geetika
      Teja
      Arjun
      2020H1030112P
      2020H1030113P
      2020H1030114P
```

```
Sneak peek:
We can also read inputs from more than two input streams using
SequenceInputStream, but using Enumerations, Look into this aspect by
yourselves to learn more. By default, we can hardcode only two input
streams using the available constructor of the class.

*/
```

8.3.5 ByteArrayOutputStream

This class will become handy when we are trying to write common data to multiple streams. The ByteArrayOutputStream class internally uses a **byte Array** (which it creates itself and is dynamic in nature). The data that needs to be written to multiple streams are first buffered in this byte array using the "**write**" method and the buffered data can be streamed into multiple output streams using the "**writeTo**" method.

Program 8.3.e:

```
import java.io.*;
public class BAOExample {
public static void main(String args[])throws Exception{
      FileOutputStream fout1=new FileOutputStream("output3.txt");
      fout1.write("Hey! I'm File1\n".getBytes());
      FileOutputStream fout2=new FileOutputStream("output4.txt");
       fout2.write("Hello! I'm File2\n".getBytes());
      ByteArrayOutputStream bout=new ByteArrayOutputStream();
      String s = "Hi this is annoucement";
      byte buf[]=s.getBytes();
      bout.write(buf);    //line1
      bout.writeTo(fout1);   //line2
      bout.writeTo(fout2);

      bout.flush();
      bout.close();
      fout1.close();
```

```
      fout2.close();
    }
  }
```

8.3.6 Exercises

1) Modify the code of **program 8.3.a** in an efficient way, such that the contents of the "output1.txt" file after the execution should be of the below form:

```
BITS Pilani Students info
Student1 Details:
Name: Teja
Year: 2020
Age: 22
-----------------
Student2 Details:
Name: Harshith
Year: 2020
Age: 22
-----------------
```

2) In the **program 8.3.d**, we could see the output when we are trying to read two files one after the other using SequenceInputStream. Now, write a program, which reads the data inside from "**studname.txt**" and "**studId.txt**" parallelly, and merge those data and store it in "**studDetails.txt**".

   The contents of "studDetails.txt" should be of the below form:

```
Geetika - 2020H1030112P
Teja - 2020H1030113P
Arjun - 2020H1030114P
```