**National University of Computer & Emerging Sciences, Karachi**

**Computer Science Department**

**Fall 2022, Lab Manual - 03**

|  |  |
| --- | --- |
| **Course Code: SL3001** | **Course: Software Development and construction** |
| **Instructor:** | **Miss Nida Munawar** |

**Lab # 03**

**I/O, Try-with-Resources**

**Streams**

Java programs perform I/O through streams. A *stream* is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system.

**Byte Streams and Character Streams**

Java defines two types of streams: **byte and character**.

***Byte streams***provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data.

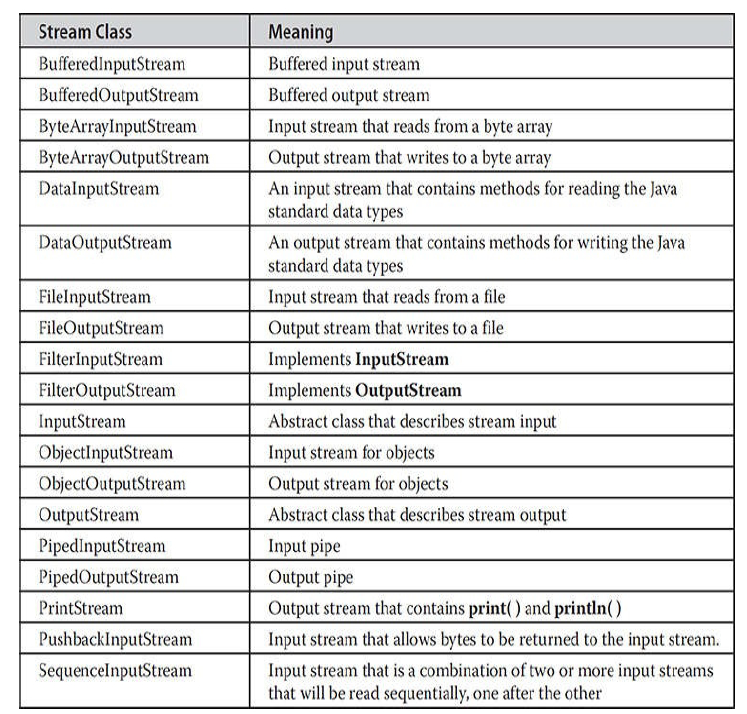
***Character streams*** provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized. Also, in some cases,character streams are more efficient than byte streams.

**The Byte Stream Classes**

Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream**.

Each of these abstract classes has several concrete subclasses that handle the differences among various devices, such as disk files, network connections, and even memory buffers.

**Remember, to use the stream classes, you must import java.io.**



**Among all of these concrete classes we will be using FileInputStream and FileOutputStream**

1. Java FileOutputStream Class

Java FileOutputStream is an output stream used for writing data to a [file](https://www.javatpoint.com/java-file-class)

.If you have to write primitive values into a file, use FileOutputStream class. You can write byte-oriented as well as character-oriented data through FileOutputStream class.





**Example 1:**

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**public** **class** FileOutputforint {

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

// **TODO** Auto-generated method stub

FileOutputStream o = **new** FileOutputStream("new.txt" , **true**);

o.write(65);

o.close();

System.***out***.println("hello");

}}

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, application

Description automatically generated

**Example 2: converting string to bytes**

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**public** **class** Fileoutputforstring {

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

FileOutputStream o = **new** FileOutputStream("new.txt" , **true**);

String s = "i am nida";

**byte**[] b = s.getBytes();

o.write(b);

o.close();

System.***out***.println("hello");

}

}

# Java PrintWriter class

Java PrintWriter class is the implementation of [Writer](https://www.javatpoint.com/java-writer-class)

class. It is used to print the formatted representation of [objects](https://www.javatpoint.com/object-and-class-in-java)

to the text-output stream.

Although using System.out to write to the console is acceptable, the recommended method of writing to the console when using Java is through a PrintWriter stream. PrintWriter is one of the character-based classes. Using a character-based class for console output makes internationalizing your program easier.

PrintWriter defines several constructors. The one we will use is shown here:

**PrintWriter(OutputStream outputStream, boolean flushingOn)**

Here, outputStream is an object of type OutputStream, and flushingOn controls whether Java flushes the output stream every time a println( ) method (among others) is called. If flushingOn is true, flushing automatically takes place. If false, flushing is not automatic. PrintWriter supports the print( ) and println( ) methods.

**Writing on console**

**Example 1:**

**import** java.io.File;

**import** java.io.PrintWriter;

**public** **class** PrintWriterconsole {

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

//Data to write on Console using PrintWriter

PrintWriter writer = **new** PrintWriter(System.***out***,**true**);

writer.println("Javatpoint provides tutorials of all technology.");

writer.println(2);

}

}

Graphical user interface, text, application

Description automatically generated

**Writing on File**

**Example 2:**

**import** java.io.FileNotFoundException;

**import** java.io.FileOutputStream;

**import** java.io.PrintWriter;

**public** **class** Fileoutputprintwriter {

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

FileOutputStream f = **new** FileOutputStream("new.txt" , **true**);

//another way to pass file object directly in constructor

//PrintWriter p = new PrintWriter(new FileOutputStream("new.txt" , true));

PrintWriter p = **new** PrintWriter(f);

p.println("hello this is me");

System.***out***.println("written");

p.close();

}

}

.

1. Java FileInputStream Class

Java FileInputStream class obtains input bytes from a [file](https://www.javatpoint.com/java-file-class).





**Example 1:** taking input from file using scanner

**import** java.io.FileInputStream;

**import** java.io.IOException;

**import** java.util.Scanner;

**public** **class** FileInputusingscanner {

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

// **TODO** Auto-generated method stub

FileInputStream f = **new** FileInputStream("new.txt");

Scanner s = **new** Scanner(f);//taking input from file

**while**(s.hasNext()) {

System.***out***.println(s.nextLine());

}}}

**Example 2:** taking input from file using read(raw bytes)

**import** java.io.FileInputStream;

**import** java.io.IOException;

**public** **class** Fileinputusingread {

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

FileInputStream f = **new** FileInputStream("new.txt");

**int** i = 0;

**while**((i=f.read())!= -1) {

System.***out***.print((**char**)i);

}

f.close();

}}

**The Character Stream Classes**

Character streams are defined by using two class hierarchies. At the top are two abstract classes: Reader and Writer. These abstract classes handle Unicode character streams. Java has several concrete subclasses of each of these. The character stream classes in java.io are shown in Table

Table

Description automatically generated

**Among all of these concrete classes we will be using BufferedReader and BufferedWriter**

## Reading data from console by InputStreamReader and BufferedReader

**In this example, we are connecting the BufferedReader stream with the [InputStreamReader](https://www.javatpoint.com/Input-from-keyboard-by-InputStreamReader) stream for reading the line by line data from the keyboard, for numeric data we have to parse it.**

**package com.javatpoint;**

**import java.io.\*;**

**public class BufferedReaderExample{**

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

**InputStreamReader r=new InputStreamReader(System.in);**

**BufferedReader br=new BufferedReader(r);**

**System.out.println("Enter your name");**

**String name=br.readLine();**

**System.out.println("Welcome "+name);**

**}**

**}**

## Reading data from file by FileReader and BufferedReader

**import** java.io.\*;

**public** **class** Bufferedreader {

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

// **TODO** Auto-generated method stub

//System.out.println("enter value");

BufferedReader f = **new** BufferedReader(**new** FileReader( "new.txt"));;

String s;

**while**((s=f.readLine())!=**null**) {

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

}

}}

# Java BufferedWriter Class

Java BufferedWriter class is used to provide buffering for Writer instances. It makes the performance fast. It inherits [Writer](https://www.javatpoint.com/java-writer-class) class. The buffering characters are used for providing the efficient writing of single [arrays](https://www.javatpoint.com/array-in-java), characters, and [strings](https://www.javatpoint.com/java-string).

**import** java.io.\*;

**public** **class** Bufferedwritter {

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

// **TODO** Auto-generated method stub

BufferedWriter w = **new** BufferedWriter(**new** FileWriter("new.txt" , **true**));

w.write("hello");

w.close();}}

**Try-with-Resources**

**explicit calls to close( )**

**Question: when you handle the exception through try and catch where to put close() method?**

**Inside try?(if exception occurs rest of code is ignored and control switches to catch)**

**or**

**Inside catch? (if exception does not happen catch will never execute)**

**So we call close inside the finally block(exception occurs or not finally will execute)**

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.IOException;

**import** java.util.Scanner;

**public** **class** Trywithresources {

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

FileInputStream f = **null**;

**try** {

f = **new** FileInputStream("new.txt");

Scanner s = **new** Scanner(f);

**while**(s.hasNext()) {

System.***out***.println(s.nextLine());

}}

**catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**finally** {

f.close();

}}}

**Automatically Closing a File(automatic resource management, or ARM)**

In the preceding example, the example programs have made explicit calls to close( ) to close a file once it is no longer needed. As mentioned, this is the way files were closed when using versions of Java prior to JDK 7. Although this approach is still valid and useful, JDK 7 added a feature that offers another way to manage resources, such as file streams, by automating the closing process. This feature, sometimes referred to as automatic resource management, or ARM for short, is based on an expanded version of the try statement. The principal advantage of automatic resource management is that it prevents situations in which a file (or other resource) is inadvertently not released after it is no longer needed. As explained, forgetting to close a file can result in memory leaks, and could lead to other problems. Automatic resource management is based on an expanded form of the try statement. Here is its general form: Typically, resource-specification is a statement that declares and initializes a resource, such as a file stream. It consists of a variable declaration in which the variable is initialized with a reference to the object being managed. When the try block ends, the resource is automatically released. In the case of a file, this means that the file is automatically closed. (Thus, there is no need to call close( ) explicitly.) Of course, this form of try can also include catch and finally clauses. This form of try is called the try-with-resources statement.

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.IOException;

**import** java.util.Scanner;

**public** **class** Trywithresources {

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

**try** (FileInputStream f = **new** FileInputStream("new.txt")){

Scanner s = **new** Scanner(f);

**while**(s.hasNext()) {

System.***out***.println(s.nextLine());

}}

**catch** (FileNotFoundException e) {

e.printStackTrace();

}

}}

**NOTE Beginning with JDK 9, it is also possible for the resource specification of the try to consist of a variable that has been declared and initialized earlier in the program. However, that variable must be effectively final, which means that it has not been assigned a new value after being given its initial value.**

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.IOException;

**import** java.util.Scanner;

**public** **class** Trywithresources {

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

FileInputStream f = **new** FileInputStream("new.txt");

//passing reference in try

**try** (f){

Scanner s = **new** Scanner(f);

**while**(s.hasNext()) {

System.***out***.println(s.nextLine());

}}

**catch** (FileNotFoundException e) {

e.printStackTrace();

}

}}