National University of Computer & Emerging Sciences, Karachi
Computer Science Department
Spring 2022, Lab Manual - 05

Course Code: SL3001	**Course: Software Development and construction**
Instructor(s):	**Miss Nida Munawar, Miss Abeeha Sattar**

Lab # 05

I/O, Try-with-Resources

Streams

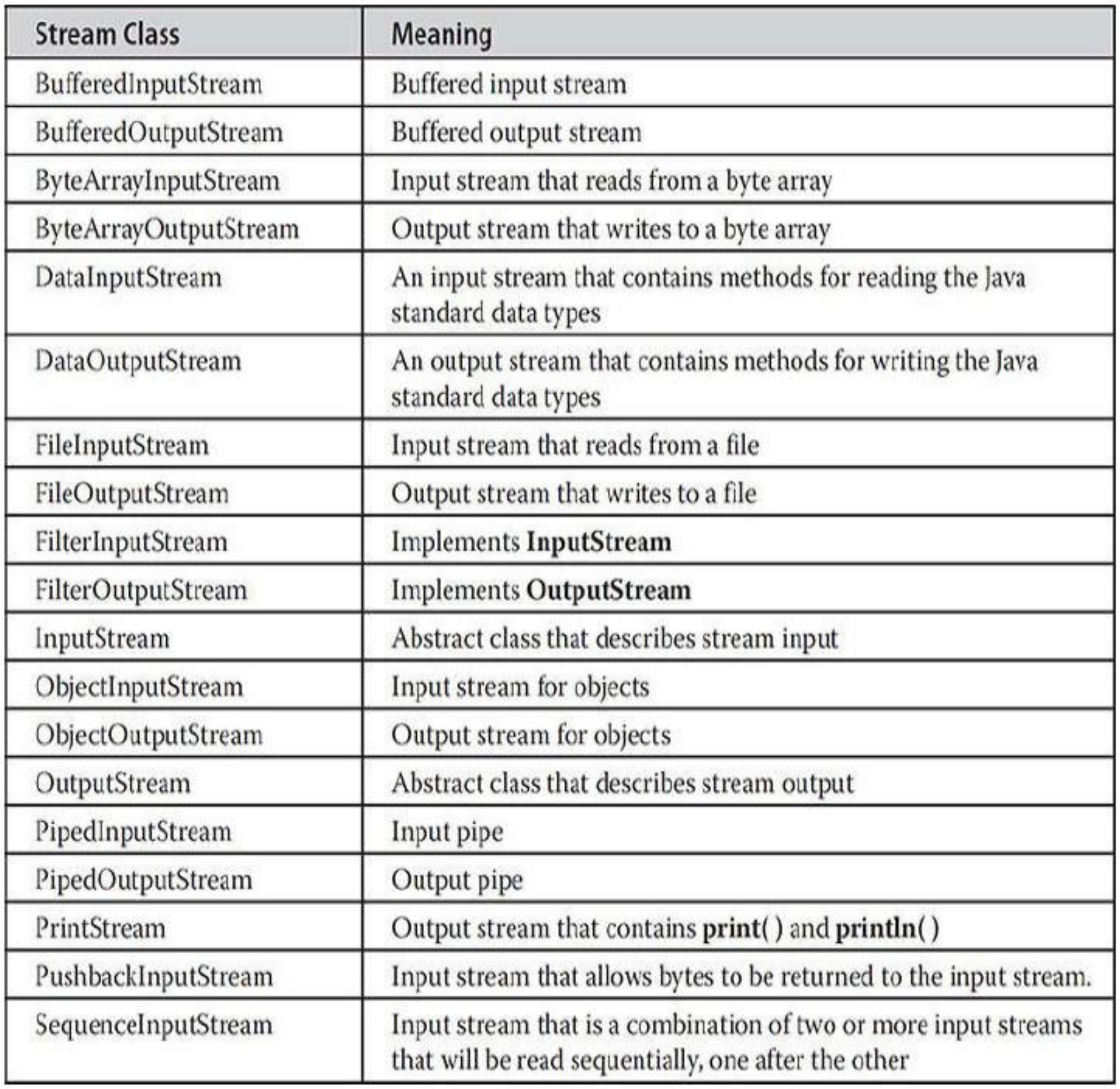
Java programs perform I/O through streams. A *stream* is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system.

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.



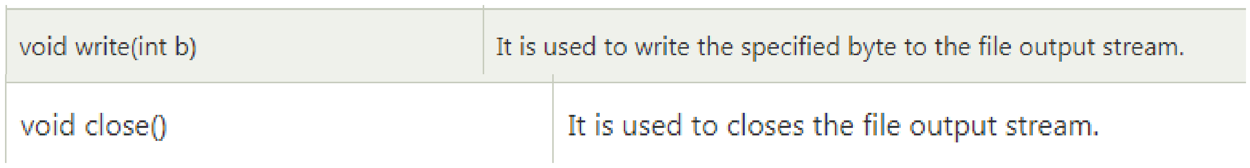
Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains **print( )** and **println( )**
PushbackInputStream	Input stream that allows bytes to be returned to the input stream.
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

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

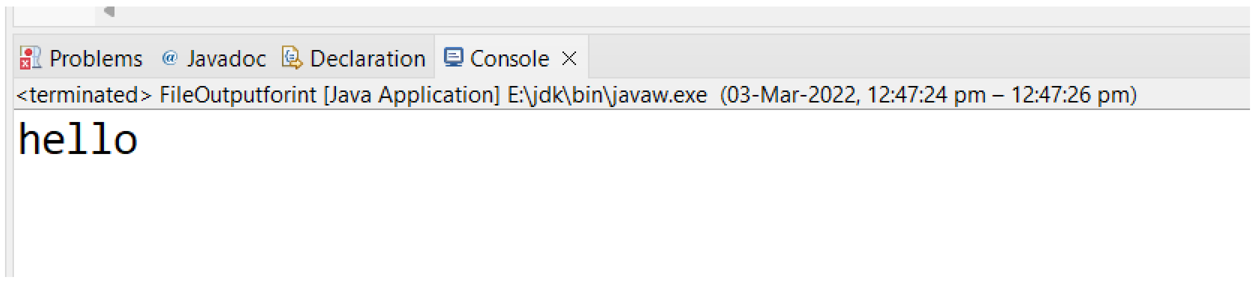
.If you have to write primitive values into a file, use FileOutputStream class. You can write byte-oriented as well as character-oriented data through FileOutputStream class.



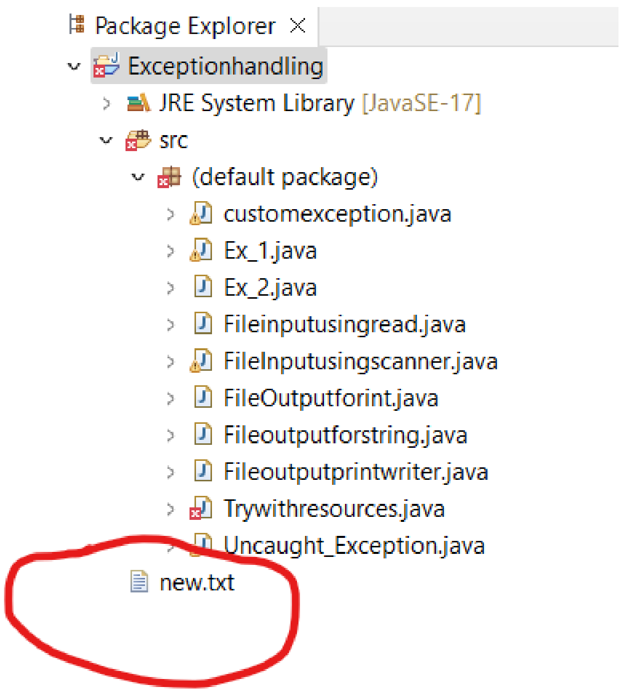
void write(int b)	It is used to write the specified byte to the file output stream.
void close()	It is used to closes the file output stream.

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");
}}
```



```
<terminated> FileOutputforint [Java Application] E:\jdk\bin\javaw.exe  (03-Mar-2022, 12:47:24 pm – 12:47:26 pm)
hello
```


Package Explorer

Exceptionhandling
- JRE System Library [JavaSE-17]
- src
  - (default package)
    - customexception.java
    - Ex_1.java
    - Ex_2.java
    - Fileinputusingread.java
    - FileInputusingscanner.java
    - FileOutputforint.java
    - Fileoutputforstring.java
    - Fileoutputprintwriter.java
    - Trywithresources.java
    - Uncaught_Exception.java
- new.txt

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");
    }
}
```

2. Java PrintWriter class

Java PrintWriter class is the implementation of Writer

class. It is used to print the formatted representation of objects

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)

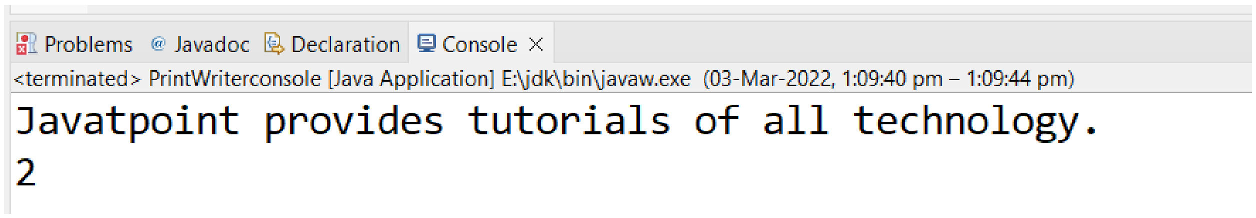
Here, outputStream is an object of type OutputStream, and flushingOn controls whether Java flushes the output stream every time a println( ) method (among others) is called. If flushingOn is true, flushing automatically takes place. If false, flushing is not automatic. PrintWriter supports the print(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

```
    }
}
```


Problems Javadoc Declaration Console

```
<terminated> PrintWriterconsole [Java Application] E:\jdk\bin\javaw.exe  (03-Mar-2022, 1:09:40 pm – 1:09:44 pm)
Javatpoint provides tutorials of all technology.
2
```

Writing on File

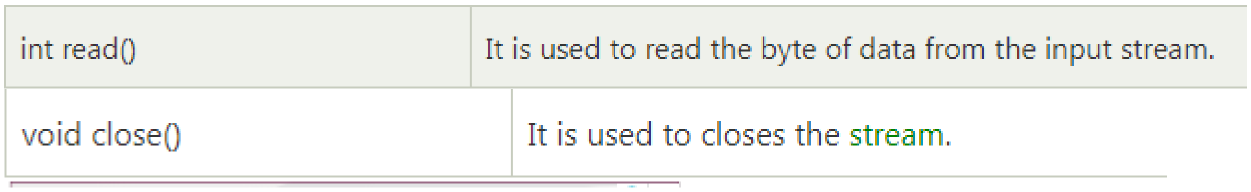
Example 2:

```
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();
}
}
```

.

3. Java FileInputStream Class

Java FileInputStream class obtains input bytes from a file.



int read()	It is used to read the byte of data from the input stream.
void close()	It is used to closes the stream.

Example 1: taking input from file using scanner

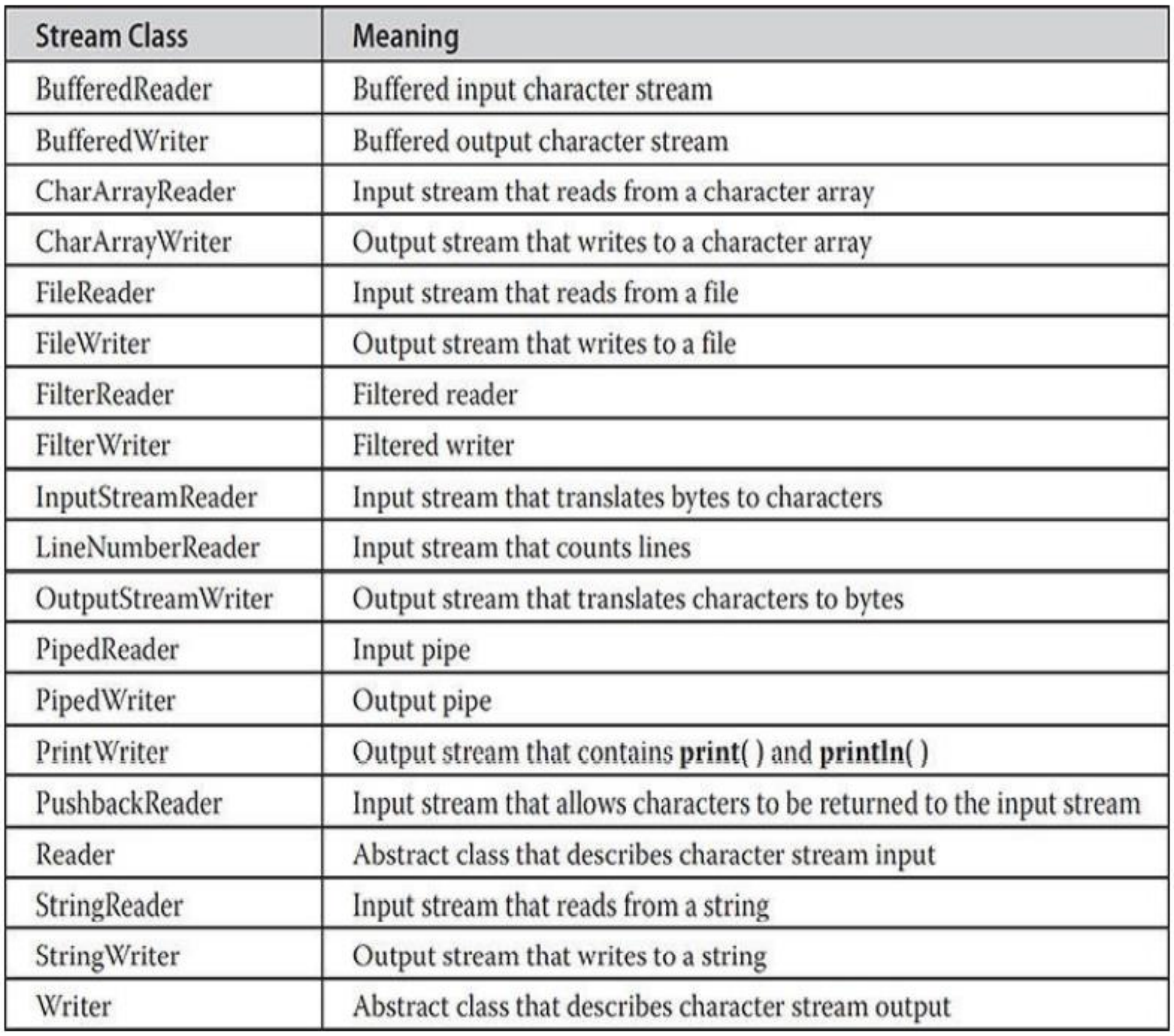
```
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Scanner;
public class FileInputusingscanner {
public static void main(String[] args) throws
IOException {
        // TODO Auto-generated method stub
        FileInputStream f = new
FileInputStream("new.txt");
        Scanner 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



Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

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 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);
        }
}}
```

4. Java BufferedWriter Class

Java BufferedWriter class is used to provide buffering for Writer instances. It makes the performance fast. It inherits Writer class. The buffering characters are used for providing the efficient writing of single arrays, characters, and strings.

```
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.

```java
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();
        }
            }}
```