UNIT 11: FILE HANDLING

UNIT STRUCTURE

11.1 Learning Objectives

11.2 Introduction

11.3 Java I/O

11.4 I/O Basics

11.4.1 Stream and Stream Classes

11.5 Reading Console Input

11.6 Writing Console Output

11.7 Reading and Writing Files

11.8 Let Us Sum Up

11.9 Further Reading

11.10 Answer to Check Your Progress

11.11 Model Questions

11.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- explain the concept of I/O (Input-Output)
- define stream
- explain the various streams for I/O
- describe the file and the different file operations
- explain reading and writing console input and console output
- describe how to read and write files.

11.2 INTRODUCTION

In the earlier unit, we learned about the concept of exception handling. We have learned about java exception, its various types and their differences, and how to handle the exception. In this unit, you will be introduced to the concepts related to file handling.

Topics like Java I/O, I/O basics, and different stream classes are discussed in detail in this unit. In addition to these, concepts related to

reading console input, writing console output, and reading and writing files along with different file operations are also described in this unit. In the next unit, we will explore topics related to Java applets.

11.3 JAVA I/O

Java I/O (Input-Output) is a standard mechanism that processes the input and generates the output. The package "java.io" contains the methods to perform all the input and output operations. To perform I/O operations faster, Java uses the concept of streams. A stream can be defined as a sequence of data consisting of bytes. A stream refers to a logical representation of the flow of data. In this unit, the background of I/O along with its related term Stream will be discussed. Further, the way of reading and writing files in Java will also be explained.

11.4 I/O BASICS

Programs read inputs from data sources (e.g., keyboard, file, network, memory buffer, or another program) and write outputs to a data sink (e.g., display console, file, network, memory buffer, or another program). In Java standard I/O, inputs and outputs are handled by the so-called streams. A stream is a sequential and logical representation of a one-way flow of data. It is important to mention that Java does not differentiate between the various types of data sources in the stream I/O. They are all treated as a sequential flow of data. Input and output streams can be established from/to any data source/sink, such as files, network, keyboard/console, or another program. The Java program receives data from a source by opening an input stream and sends data to a sink by opening an output stream. If a program needs to perform both input and output operations, we have to open two streams - an input stream and an output stream.

Stream I/O operations involve three steps:

1. Open an input/output stream associated with a physical device (e.g., file, network, console/keyboard), by constructing an appropriate I/O

stream instance.

2. Read from the opened input stream until "end-of-stream" is encountered, or write to the opened output stream (and optionally flush the buffered output).
3. Close the input/output stream.

Java internally stores characters (char type) in the 16-bit UCS-2 character set. But the external data source/sink could store characters in another character set (e.g., US-ASCII, ISO-8859-x, UTF-8, UTF-16, and many others), in a fixed length of 8-bit or 16-bit, or variable length of 1 to 4 bytes.

On the other hand, in Java, a File is known as a named location used to store related information. There are several File Operations like creating a new File, getting information about a File, writing into a File, reading from a File, and deleting a File.

To understand the File operations, it is required to know the Stream.

11.4.1 Stream and Stream Classes

A stream is simply a logical representation of a sequence of data. It is also defined as a sequence of bytes. It acts as an object that either delivers data to its destination or that takes data from a source. The source and destination can be disk files, arrays, text files, etc. The input stream reads or retrieves data from a source, whereas the output stream writes data to a destination. Based on the data type, there are two types of streams: Byte Stream and Character Stream.

Byte Stream:

Byte Stream is mainly involved with byte data. A file-handling process with a byte stream is a process in which input is provided and executed with the byte data. Byte streams perform input and output of 8-bit bytes. They read and write data one byte at a time. Other stream types are built on top of byte streams.

Character Stream:

Character Stream is mainly involved with character data. A file-handling process with a character stream is a process in which an input is processed with the character data.

Figure 1 depicts these two types of streams along with basic classes under each category.

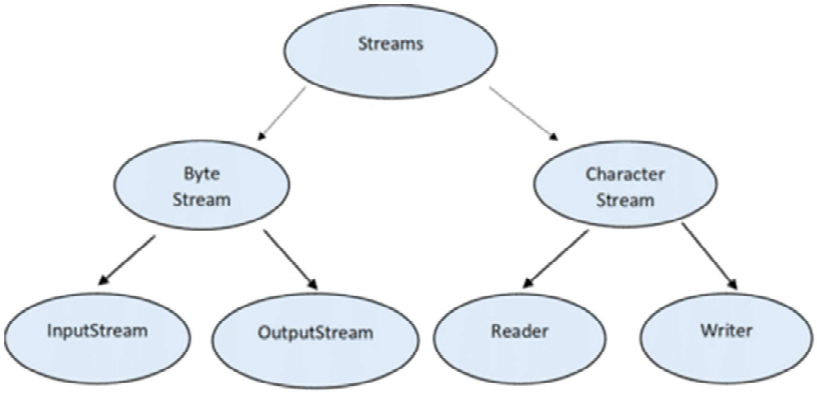


Figure11. 1: Classification of I/O streams based on data type

Again, depending on the type of operations, byte streams can be classified into two main groups: InputStream and OutputStream.

InputStream: This stream is used to read data that must be taken as an input from a source array or a file or any peripheral device. For example, FileInputStream, BufferedInputStream, ByteArrayInputStream, etc. are some predefined input stream classes as shown in figure 2.

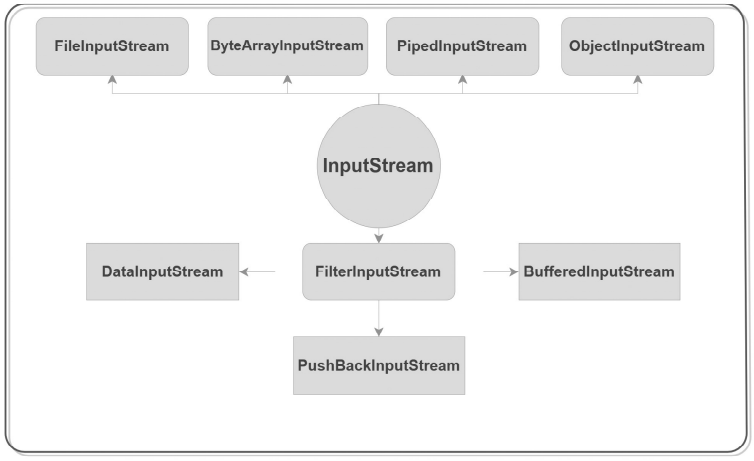


Figure11. 2: Classes under InputStream

OutputStream: This stream is used to write data as outputs into an array or file or any output peripheral device. For example, FileOutputStream, BufferedOutputStream, ByteArrayOutputStream etc. are predefined streams under this group. Figure 3 presents all the built-in classes under the OutputStream group.

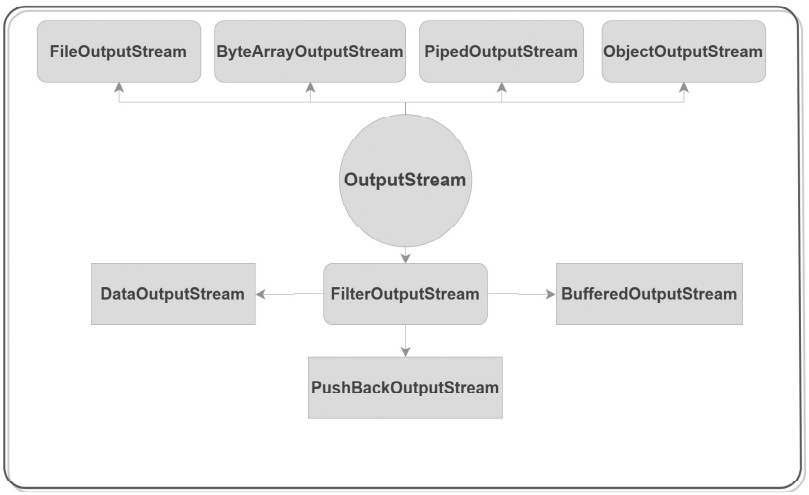


Figure11. 3: Classes under OutputStream

Java brings various Streams with its I/O package that helps the user to perform all the input-output operations. These streams support all the types of objects, data types, characters, files, etc. to execute the I/O operations. The read operation and write operation through Java application are depicted in figure 4.



Figure11. 4: The flow of read-and-write operations

Java I/O stream is the flow of data that you can either read from, or you can write to. It is used to perform read and write operations in files permanently. Java uses streams to perform these tasks. Java I/O stream is also called File Handling, or File I/O. This Java I/O stream is available in the java.io package. Java.io package provides classes for system input and output through files, network

streams, memory buffers, etc.

Example 11.1: Program to read contents of a file

```
import java.io.*;
class ReadHello{
     public static void main(String args[])
{
        try     {
          FileInputStream     fin=new
FileInputStream("hello.txt");
          inti=0;
          while((i=fin.read())!=-1)  {
          System.out.println((char)i);
          }
          fin.close();
          }
        catch(Exception e)  {
          System.out.println(e);  }
               }
          }
     }
```

Example 11.2: Program to read from a file

```
import java.io.*;
     public class ReadFile {
public static void main(String[] args) {
BufferedReader br1 = null;
BufferedReader br2 = null;
    try    {
     br1  =  new  BufferedReader(new
FileReader("D:\\myfile.txt"));
     String content = br1.readLine();
     while (content != null) {
          System.out.println(content);
```

```
                content = br1.readLine();
      }
            }
      catch (IOExceptionioe)
      {
      ioe.printStackTrace();
      } }
            }
```

Example 11.3: Program in Java to write in a file

```
      import java.io.*;
            public class Hello{
            public static void main(String args[]){
             try  {
            F i l e O u t p u t s t r e a m f o = n e w
FileOutputStream("hello.txt");
            String i="Hello World ";
            byte  b[]=i.getBytes();/  /converting
string into byte array
            fo.write(i);
            fo.close();
                 }
                     catch(Exception  e)     {
system.out.println(e);  }
               }
            }
      This will create a file hello.txt.
```

Output:

Hello World

Example 11.4: Program to write in a file

```
            import java.io.*;
            public class writeFirst     {
            public static void main(String args[])
```

```
{
        File file=new File("visiters.txt");
        try   {
              PrintWriter      output=new
PrintWriter(file);
              output.println("Hello Visiters!
Welcome to Intellipaat");
              output.close();
                    }
     catch(IOException        ex)        {
System.out.println("Error:"+ex);}
       }
     }
```

There are 3 standard or default streams System.in, System.out and System.err that Java provides for I/O operations which are shown in figure 5.

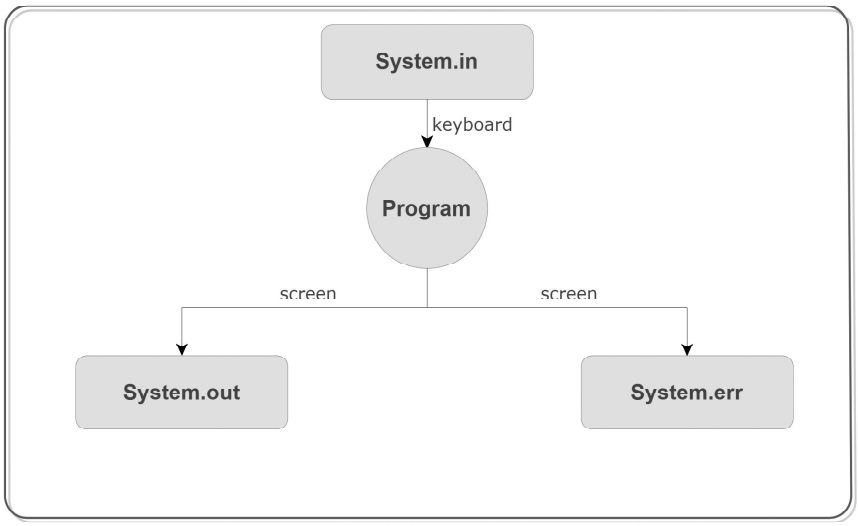


Figure11. 5: Standard Java streams

1. **System.in:** This is the standard input stream that is used to read characters from the keyboard or any other standard input device.
2. **System.out:** This is the standard output stream that is used to produce the result of a program on an output device like the

computer screen.

3. **System.err:** This is the standard error stream that is used to output all the error data that a program might throw, on a computer screen or any standard output device. This stream can also use all the 3 above-mentioned functions to output the error data.

The various print functions associated with System.out are discussed here.

print(): This method in Java is used to display a text on the console. The text is passed as the parameter to this method in the form of String. While displaying the text on the console, the cursor remains at the end of the text at the console.

Syntax: `System.out.print(parameter);`

Example 11.5: Program to show use of print()

```
// Java code to illustrate print()
import java.io.*;
class Demo_print {
    public static  void main(String[] args)
    {
        System.out.print("Welcome! ");
        System.out.print("Welcome! ");
        System.out.print("Welcome! ");
    }
}
```

Output:

Welcome! Welcome! Welcome!

println(): This method in Java is also used to display text on the console. However, it prints the text on the console, and then the cursor moves to the start of the next line at the console.

Syntax: `System.out.println(parameter);`

Example 11.6: Program to show the use of println()

```
// Java code to illustrate println()
import java.io.*;
```

```
class Demo_print {
    public static void main(String[] args)
    {
        System.out.println("Hello ");
        System.out.println("Hello ");
        System.out.println("Hello ");
    }
}
```

Output:

Hello
Hello
Hello

printf(): This is the easiest of all methods as this is similar to printf in C. Note that System.out.print() and System.out.println() take a single argument, but printf() may take multiple arguments. This is used to format the output in Java.

Example 11.7: Program to show working of printf()

```
// A Java program to demonstrate working of
// printf() in Java
class JavaFormat {
    public static void main(String args[])
    {       intx = 100;
System.out.printf("Printing simple"+"integer:x=%d\n",x);
      // this will print it upto 2 decimal places
System.out.printf("Formatted with"+"precision:PI = %.2f\n",
                  Math.PI);
                floatn = 5.2f;
// automatically appends zero to the rightmost
// part of decimal
System.out.printf("Formatted to "+ "specific width: n = %.4f\n",n);
```

```
                n = 2324435.3f;
        // here number is formatted from right margin
// and occupies 20 characters
         System.out.printf("Formatted to " + "right
margin: n = %20.4f\n", n);
            }
        }
```

Output:

Printing simple integer: x = 100

Formatted with precision: PI = 3.14

Formatted to specific width: n = 5.2000

Formatted to right margin: n = 2324435.2500

Stream Classification based on type of files: Depending on the types of files, Streams can be classified into two primary classes which can be further divided into other classes as shown in the figure 6.

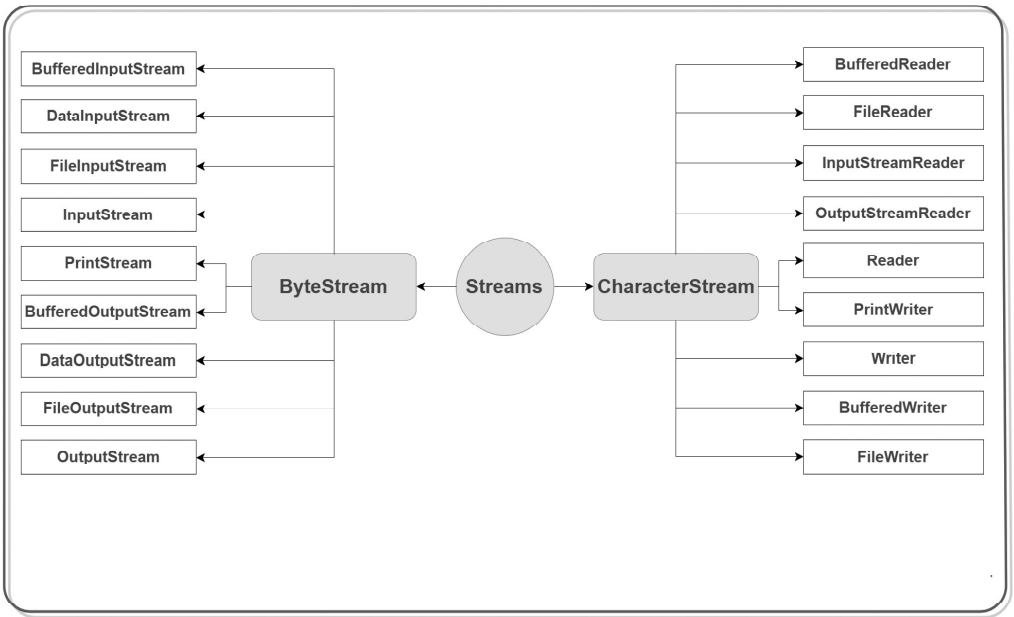


Figure 11. 6: Various classes under ByteStream and CharacterStream

ByteStream: This is used to process data byte by byte. Although ByteStream has many classes, the FileInputStream and the FileOutputStream are the mostly used classes. The FileInputStream

is used to read from the source and FileOutputStream is used to write to the destination. A list of various ByteStream Classes is given in Table 11.1.

Table 11.1: Various Classes under ByteStream

Stream class	Description
BufferedInputStream	It is used for Buffered Input Stream.
DataInputStream	It contains method for reading java standard datatypes.
FileInputStream	This is used to reads from a file
InputStream	This is an abstract class that describes stream input.
PrintStream	This contains the most used print() and println() method
BufferedOutputStream	This is used for Buffered Output Stream.
DataOutputStream	This contains method for writing java standard data types.
FileOutputStream	This is used to write to a file.
OutputStream	This is an abstract class that describe stream output.

Example 11.8: Program to show use of Byte Stream

```
// A Java Program to illustrating the Byte
Stream to copy the contents // of one file to
another file.
import java.io.*;
public class BYTEStream {
```

```
            publicstaticvoidmain( String[] args)
throwsIOException
          {
              FileInputStream sourceStream = null;
              FileOutputStreamtargetStream = null;
              try{
      source = newFileInputStream("File1.txt");
      target = new File Output Stream
("targetfile.txt");

      // Reading source file and writing content
to target file byte by byte
                  int temp;
                  while((temp = source.read()) !=
-1)
                      target.write((byte)temp);
              }
              finally{
                  if(source!=null)
                      source.close();
                  if(target!=null)
                     target.close();      }     }
}
```

CharacterStream: In Java, characters are stored using Unicode conventions where CharacterStream allows us to read/write data character by character. Among various classes, the FileReader and the FileWriter are the most common ones. FileReader reads from the source and FileWriter writes to the destination respectively. Table 2 presents a list of various CharacterStream Classes.

Table 11.2: Various CharacterStream Classes

Stream class	Description
BufferedReader	Used to handle buffered input stream.
FileReader	This is an input stream that reads from file.
InputStreamReader	Used to translate byte to character.
OutputStreamReader	This output stream is used to translate character to byte.
Reader	This is an abstract class that define character stream input.
PrintWriter	It contains the most used print() and println() method
Writer	This is an abstract class that define character stream output.
BufferedWriter	Used to handle buffered output stream.
FileWriter	Used to output stream that writes to file.

File Operations: The following operations on a file can be performed.

- Create a File
- Retrieve File Information
- Write to a File
- Read from a File
- Delete a File

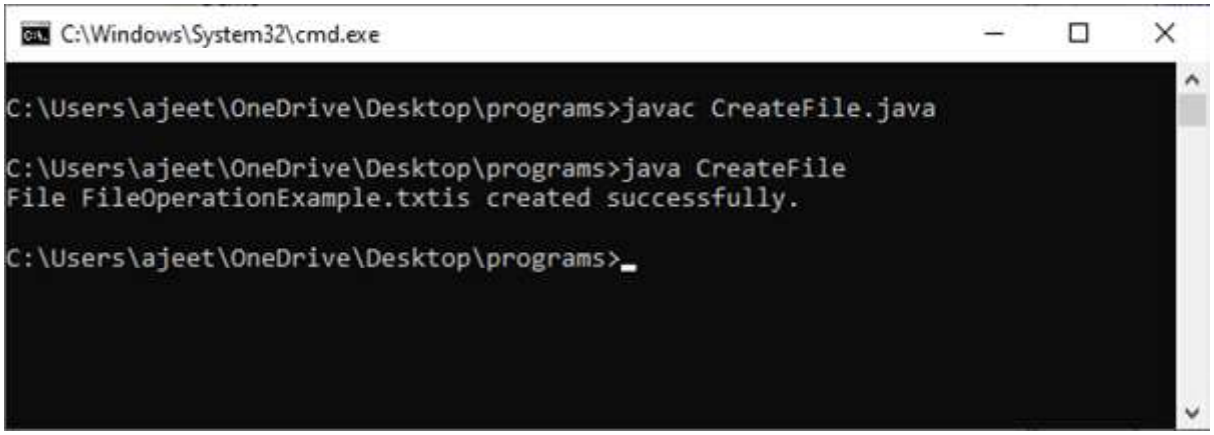
Create a File:

To create a File, we use the createNewFile() method of file. The createNewFile() method returns true if it successfully creates a new file and returns false when the file already exists. Following program creates a new file.

Example 11.9: Program to show how to create a file

```
// Importing File class
import java.io.File;
// Importing the IOException class for handling
errors
import java.io.IOException;
class CreateFile {
    public static void main(String args[]) {
        try {
    // Creating an object of a file
File f1 = new File("D:FileOperationExample.txt");
            if (f1.createNewFile()) {
System.out.println("File " + f1.getName() + " is
created successfully.");
       }
else {
 System.out.println("File is already exist in the
directory.");
                }
            }  catch (IOException exception) {
                System.out.println("An unexpected
error is occurred.");
                  exception.printStackTrace();
             }
       }
     }
```

Output:



Writing data to a file:

The write operation on a file means storing the data in a file. To perform write operations on a file, we have to use the write() method along with the FileWriter class. We should use the close() method to close the stream to retrieve the allocated resources.

Example 11.10: Program to write data to a file

```
importjava.io.*;
publicclassFileHandling
{
  publicstaticvoidmain(String []arr) throwsIOException
{
     // creating a new file and writing data to a file
     FileWriterfw=newFileWriter("demo.txt");
     String s="Welcome to Java File Handling.";
     fw.write(s);
     // closing a file
     fw.close();
   }
}
```

Reading from a file:

The read operation can access or retrieve the data stored from a file. To perform read operation on a file, we will use Scanner class along with has NextLine() and nextLine() methods to retrieve data

from a file. We should use the close() method to close the stream.

Example 11.11: Program to read data from a file

```
importjava.io.*;
importjava.util.Scanner;
publicclassFileHandle{
     publicstaticvoidmain(String []arr)
throwsIOException{
     File f=newFile("demo.txt");
     Scanner sc=newScanner(f);
     while(sc.hasNextLine())
     {
       String str=sc.nextLine();
       System.out.println(str);
     }
       sc.close();     }   }
```

Deleting a file:

In Java file handling, a file can be deleted using the delete() method of the File class.

There is no need to close the file using close() because neither the FileWriter nor the Scanner classes are used to delete a file.

Example 11.12: Program to delete a file

```
import java.io.*;
public class FileHandle  {
     public  static  void  main(String  []arr)
throwsIOException
    {
      File f=new File("DEMO.txt");
      Boolean result=f.delete();
      if(result)
         System.out.print(f+" deleted successfully.");
      else
```

```
        System.out.format("%s","File cannot be deleted
due to some error.");
   }
}
```

Retrieving file information:

There are few methods in Java that I have discussed earlier at methods of file class to get the information of the file.

Example 11.13: Program to retrieve file information file information

```
importjava.io.*;
publicclassFileHandle{
  publicstaticvoidmain(String []arr) throwsIOException{
     File file=newFile("demo.txt");
     file.createNewFile();
     String filename=file.getName();
     System.out.println("File Name is "+filename);
     System.out.println("Absolute path of "+filename+"
: "+file.getAbsolutePath());
        System.out.print("length of "+filename+" :
"+file.length());
        System.out.println("Is "+filename+" readable?
"+file.canRead());
        System.out.println("Is "+filename+" writable?
"+file.canWrite());
        System.out.println("Is "+filename+" exists?
"+file.exists());
  }
}
```

11.5 READING CONSOLE INPUT

Java input console is accomplished by reading from System.in. To obtain a character based stream that is attached to the console, we have to wrap System.in in a BufferedReader object, to create a character stream.

The syntax to obtain an object of BufferedReader is:

```
BufferedReaderbr = new BufferedReader(new
InputStreamReader(System.in));
```

Once BufferedReader is obtained, we can use read() method to reach a character or readLine() method to read a string from the console.

Reading Characters from Console

To read a character from a BufferedReader, we would use read( ) method whose syntax is as follows: int read( ) throws IOException

Each time that read( ) is called, it reads a character from the input stream and returns it as an integer value. As you can see, it can throw an IOException. The following program demonstrates read( ) by reading characters from the console until the user types a "q":

Example 11.14: Program to show use of read()

```
// Use a BufferedReader to read characters from the
console.
import java.io.*;
public class BRRead
{ public static void main(String args[]) throws
IOException
{ char c; // Create a BufferedReader using System.in
BufferedReaderbr = new BufferedReader(new
InputStreamReader(System.in)); System.out.println("Enter
characters, 'q' to quit."); // read characters
do { c = (char) br.read(); System.out.println(c); }
while(c != 'q');
} }
```

Reading Strings from Console

To read a string from the keyboard, we have to use readLine( ) that is a member of the BufferedReader class. Its general form is:

```
String readLine( ) throws IOException
```

The following program demonstrates BufferedReader and the readLine( ) method. The program reads and displays lines of text until you enter the word "end":

Example 11.15: Program to show use of readLine()

```
// Read a string from console using a BufferedReader.
import java.io.*;
public class BRReadLines
{ public static void main(String args[]) throws
IOException
{ // Create a BufferedReader using System.in
BufferedReaderbr = new BufferedReader(new
InputStreamReader(System.in));
String str;
System.out.println("Enter lines of text.");
System.out.println("Enter 'end' to quit.");
do { str = br.readLine(); System.out.println(str);
} while(!str.equals("end"));
}
}
```

There are several ways to read data given by the user at the console. They are:

- Scanner class
- BufferedReader class
- Console class

Scanner Class:

This is the most common and preferred method to take input. It is used to read input from the console. However, it was initially brought up for the purpose of parsing strings and primitive data types. With this feature of parsing the tokenized input using regex and a series of methods that can be used to parse primitive data, makes it a popular choice to read input.

Example 11.16: Program to show use of Scanner class

```
import java.util.Scanner;
public class ScannerTest
{
    public static void main(String args[])
    {
```

```
            // Using Scanner for Getting Input using
System.in[ Default Configuration is Console ]
        Scanner in = new Scanner(System.in);
System.out.println("Please enter some string: ");
        String myString = in.nextLine(); //This parses
to string

System.out.println("Please enter some integer: ");
intmyInt = in.nextInt(); //This parses to integer

System.out.println("Please enter some float: ");
        float myFloat = in.nextFloat();  //This parses
to float

System.out.println("*******Validation Part*******");
System.out.println("Your input string: "+myString);
System.out.println("Your input int: "+myInt);
System.out.println("Your input float: "+myFloat);
    }
}
```

Output:

```
Please enter some string: Arunabha Das
Please enter some integer: 60
Please enter some float: 25.87
*******Validation Part*******
Your input string: Arunabha Das
Your input int: 60
Your input float: 25.87
```

BufferedReader Class

This is the classical way to take input, which was part of JDK's first release. Here, we instantiate the BufferedReader class by wrapping an InputStreamReader which wraps the standard console input stream. It buffers the input for efficient reading. It is accompanied by static methods

of Wrapper classes to read corresponding data types.

Program 11.17: Program to show use of BufferedReader class

```
import java.io.*;
public class BufferedReaderTest
{
      public static void main(String args[]) throws
IOException
    {
      // Using Scanner for Getting Input using System.in
BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));
System.out.println("Givea string: ");
        String myString = in.readLine(); //This parses
to string

System.out.println("Please enter some integer: ");
intmyInt = Integer.parseInt(in.readLine()); //This parses
to integer

System.out.println("Enter some float value: ");
       float myFloat = Float.parseFloat(in.readLine());
//This parses to float

System.out.println("*******Validation Part*******");
System.out.println("Your input string: "+myString);
System.out.println("Your input int: "+myInt);
System.out.println("Your input float: "+myFloat);
    }
}
```

Output:

```
Give a string: Reading console input
Please enter some integer: 45
Enter some float value: 15.67
```

```
*******Validation Part*******
Your input string: Reading from console
Your input int: 45
Your input float: 15.67
```

Console Class

It is another for reading the user's input from the command line being synchronized. In addition, it has the feature of maintaining secrecy while reading password-like input without echoing the input. The string can also be formatted.

Program 11.18: Program to show use of Console class

```
public class MainConsoleTest
{
    public static void main(String args[])
    {
                            String   myString   =
System.console().readLine("Please enter username: ");
char[] myPassword = System.console().readPassword("Please
enter password: ");

System.out.println("*******Validation Part*******");
System.out.println("Your input username: "+myString);
System.out.print("Your input pasword: ");
for(char x:myPassword){
System.out.print(x);
        }
    }
}
```

Output:

```
Please enter username: Robert Perry
Please enter password: *****
*******Validation Part*******
Your input username: Robert Perry
Your input pasword: 12345
```

11.6 WRITING CONSOLE OUTPUT

Console output is most easily accomplished with print() and println() methods which are defined by the class PrintStream which is the type of the object referenced by System.out. Because PrintStream is an output stream derived from OutputStream, it also implements the low?level method write(). Thus, write() can be used to write to the console. The simplest form of write() defined by PrintStreamis :

void write(intbyteval);

This method writes to the stream the byte specified by byteval. Although byteval is declaredas an integer, only the low?order eight bits are written.

A simple example demonstrating write( ) to output the character "A" followed by a newline to the screen is given below:

```
import java.io.*; // Demonstrate System.out.write().
public class WriteDemo{ public static void main(String args[])
{ int b;
b = 'A';
System.out.write(b);
System.out.write('\n');
 }
}
```

There are two ways to write at the console. Both the ways belong to class PrintStream(Character-based class).

- **print and.println methods:** Here, the methods are defined by class Printstream and is referenced by System.out.
- **write method:** It is also defined by Printstream class. It is not recommended for use.

Let us look at the following example.

Program 11.19: Program to show use of write().

```
public class Main
{
```

```
        public static void main(String args[])
        {
    System.out.println("System.out.print will print the
passed string");
    for(inti=1;i<=5;i++){
    System.out.print(i);
          }

    System.out.println("\nSystem.out.printlnprints the
passed string and end with a new line");
    for(inti=1;i<=5;i++)
    {
    System.out.println(i);
          }
    System.out.println("\nSystem.out.write accepts the
intbyteval as input parameter ");
    for(int x=65;x<=70;x++){
    System.out.write(x);
    System.out.write('\n');
          }
        }
    }
```

Output:

```
    System.out.print will print the passed string.
    12345
    System.out.println prints the passed string and
end with a new line
    1
    2
    3
    4
    5
```

```
System.out.writeaccepts the intbyteval as input
parameter
A
B
C
D
E
F
```

11.7 READING AND WRITING FILES

To read from or write to files, three important streams FileInputStream, ByteArrayInputStream and DataInputStreamare usedfor inputs and three streams FileOutputStream, ByteArrayOutputStream and DataOutputStream are used for outputs.

File Input Stream

This stream is used for reading data from the files. Objects can be created using the keyword new and there are several types of constructors available.

A constructor of FileInputStreamtakes a file name as a string to create an input stream object to read the file as shown below.

```
InputStream f = new FileInputStream("C:/java/
hello");
```

Another constructor of File class takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows:

```
File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);
```

Once you have InputStream object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

Byte Array Input Stream

The ByteArrayInputStream class allows a buffer in the memory to

be used as an InputStream. The input source is a byte array. There are following forms of constructors to create ByteArrayInputStream objects.

By taking a byte array as the parameter:

```
ByteArrayInputStreambArray = new ByteArrayInputStream(byte [] a);
```

Another form takes an array of bytes, and two integers as arguments, where off is the first byte to be read and len is the number of bytes to be read.

```
ByteArrayInputStreambArray = new ByteArrayInputStream(byte []a, int off, intlen);
```



CHECK YOUR PROGRESS

Q 1. Which of these class contains the methods used to write in a file?

a) FileStream b) FileInputStream c) BufferedOutputStream d) FileBufferStream?

Q 2. Which of these methods are used to read in from file?

a) get() b) read() c) scan() d) readFileInput()

Q 3. Which of these exception is thrown by close() and read() methods?

a) IOException b) FileException c) FileNotFoundException d) FileInputOutputException

Q 4. Which of these classes is used for file handling?

a) InputStream b) PrintStream c) FileWriter d) None of these

Q 5. Which of these classes is not a member of java.io package?

a) File b) Writer c) String d) StringReader

Q 6. Which method is used to write a byte to the current output stream?

a) public void write(int) throws IOException

b) public void write(byte[]) throws IOException

c) public void flush() throws IOException

d) public void close()throws IOException

Q 7. Which class in Java is used to take input from the user?

a) Scanner b) Input c) FileInput

d) None of the above

Q 8. It is important to close the stream _____ after writing to a file is finished.

a) Object b) Reader c) File d) Directory

Q 9. The abstract class _____ defines the functionality that is available to all character input streams.

a) Object b) Reader c) File d) Directory

Q 10. _____ class converts a stream of bytes into a stream of characters.

a) Writer b) Reader c) String d) None of these



11.8 LET US SUM UP

- In Java standard I/O, inputs and outputs are handled by the so-called streams.
- A stream is a sequential and logical representation of one-way flow of data.
- Stream I/O operations involve three steps:
  - Open an input/output stream associated with a physical device (e.g., file, network, console/keyboard), by constructing an appropriate I/O stream instance.
  - Read from the opened input stream until "end-of-stream" encountered, or write to the opened output stream (and optionally flush the buffered output).
  - Close the input/output stream.
- Based on the data type, there are two types of streams: Byte Stream and Character Stream.
- Byte Stream is mainly involved with byte data. A file handling process with a byte stream is a process in which an input is provided and executed with the byte data.

- Character Stream is mainly involved with character data. A file handling process with a character stream is a process in which an input is processed with the character data.
- Depending on the type of operations, byte streams can be classified into two main groups: InputStream and OutputStream.
- InputStream is used to read data that must be taken as an input from a source array or a file or any peripheral device. For example, FileInputStream, BufferedInputStream, ByteArrayInputStream, etc. are some predefined input stream classes.
- OutputStream is used to write data as outputs into an array or file or any output peripheral device. For example, FileOutputStream, BufferedOutputStream, ByteArrayOutputStream etc. are predefined streams under this group.
- There are 3 standard or default streams System.in, System.out and System.err that Java provides for I/O operations.
- Five operations on a file can be performed- Creating a file, Reading from a file, Writing to a file, Deleting a file and Retrieving file operation.



11.9 FURTHER READING

1) Schildt, H. (2014). Java: The Complete Reference, Oracle Press.
2) Rao, R. N. (2006). Core and Advanced Java Black Book, Dreamtech Press



11.10 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q No 1. b

Ans to Q No 2. b

Ans to Q No 3. a

Ans to Q No 4. a

Ans to Q No 5. c

Ans to Q No 6. a

Ans to Q No 7. a

Ans to Q No 8. a

Ans to Q No 9. b

Ans to Q No 10. b



11.11 MODEL QUESTIONS

Q 1. Discuss about basics of I/O in Java.

Q 2. What is stream meant for? What are the streams used for I/O?

Q 3. Explain about three Standard I/O streams.

Q 4. How do you implement Reading Console input and Writing Console output in Java?

Q 5. Write a Java program to create a file "Welcome.txt" and write one line of text into that file.

Q 6. What are the operations be performed on files? Discuss various Predefined streams for file operations in Java.

*** ***** ***