EXPERIMENT NO 9

Q1. What is stream?

Ans. 1. A ***Stream*** can be defined as a sequence of data.

2.In Java 8, the Stream API is used to process collections of objects. A ***Stream*** is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

3. The features of Java stream are –

- A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
- Streams don't change the original data structure, they only provide the result as per the pipelined methods.
- Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

4. Different Operations On Streams- e.g. **map, filter, sorted, forEach etc**

5. import java.io.*;

Q2. What are the different streams?

Ans. All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen. If you are aware of C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similarly, Java provides the following three standard streams −

- **Standard Input** − This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.
- **Standard Output** − This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**.
- **Standard Error** − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**.

Following is a simple program, which creates **InputStreamReader** to read standard input stream until the user types a "q" –

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

   public static void main(String args[]) throws IOException {
      InputStreamReader cin = null;

      try {
         cin = new InputStreamReader(System.in);
         System.out.println("Enter characters, 'q' to quit.");
         char c;
         do {
            c = (char) cin.read();
            System.out.print(c);
         } while(c != 'q');
      }finally {
         if (cin != null) {
            cin.close();
         }
      }
   }
}
```

Q3. What is String ? and to perform different String operations, which are the possible classes are available in java?

Ans. 1. Java provides ***String class*** to perform string operations.

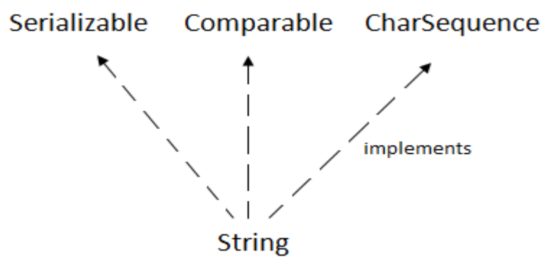
2.In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string.

For example:

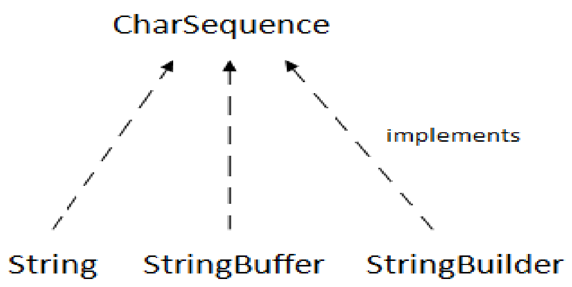
Char [] ch={'j','a','v','a','t','p','o','i','n','t'};
String s=new String(ch);
is same as:
String s="javatpoint";

3. **Java String** class provides a lot of methods to perform operations on string such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

4.The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces.



CharSequence Interface:
The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in java by using these three classes.



The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For ***mutable*** strings, you can use ***StringBuffer*** and ***StringBuilder classes.***

Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

No.	Method	Description
1	char charAt(int index)	returns char value for the particular index
2	int length()	returns string length
3	static String format(String format, Object... args)	returns a formatted string.
4	static String format(Locale l, String format, Object... args)	returns formatted string with given locale.
5	String substring(int beginIndex)	returns substring for given begin index.
6	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index.

7	boolean contains(CharSequence s)	returns true or false after matching the sequence of char value.
8	static String join(CharSequence delimiter, CharSequence... elements)	returns a joined string.
9	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	returns a joined string.
10	boolean equals(Object another)	checks the equality of string with the given object.
11	boolean isEmpty()	checks if string is empty.
12	String concat(String str)	concatenates the specified string.
13	String replace(char old, char new)	replaces all occurrences of the specified char value.
14	String replace(CharSequence old, CharSequence new)	replaces all occurrences of the specified CharSequence.
15	static String equalsIgnoreCase(String another)	compares another string. It doesn't check case.
16	String[] split(String regex)	returns a split string matching regex.
17	String[] split(String regex, int limit)	returns a split string matching regex and limit.
18	String intern()	returns an interned string.
19	int indexOf(int ch)	returns the specified char value index.
20	int indexOf(int ch, int fromIndex)	returns the specified char value index starting with given index.
21	int indexOf(String substring)	returns the specified substring index.
22	int indexOf(String substring, int fromIndex)	returns the specified substring index starting with given index.
23	String toLowerCase()	returns a string in lowercase.
24	String toLowerCase(Locale l)	returns a string in lowercase using

25	String toUpperCase()	returns a string in uppercase.
26	String toUpperCase(Locale l)	returns a string in uppercase using specified locale.
27	String trim()	removes beginning and ending spaces of this string.
28	static String valueOf(int value)	converts given type into string. It is an overloaded method.

Q4. What are the different approaches are available to read and write data into file using java ? List out the classes from Character-oriented and byte-oriented approaches.

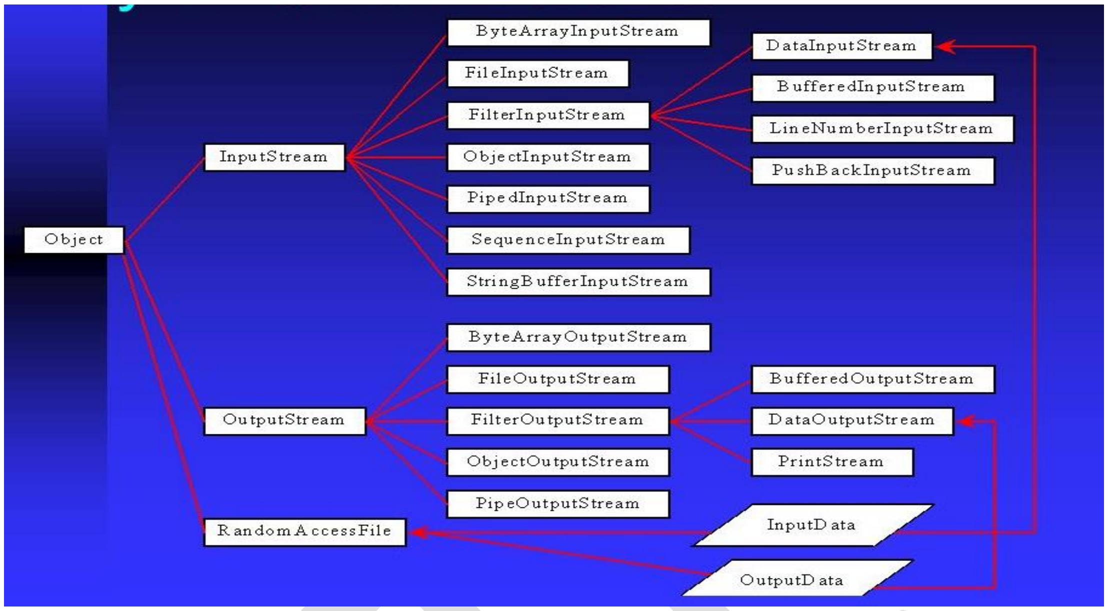
Ans. There are two approaches provided in java for read and write into file.

- **1.Byte Streams**
- **2.Character Streams**

- **1.Byte Streams**

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**.

Following is an example which makes use of these two classes to copy an input file into an output file –



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

   public static void main(String args[]) throws IOException {
      InputStreamReader cin = null;

      try {
         cin = new InputStreamReader(System.in);
        import java.io.*;
public class CopyFile {

   public static void main(String args[]) throws IOException {
      FileInputStream in = null;
      FileOutputStream out = null;

      try {
         in = new FileInputStream("input.txt");
         out = new FileOutputStream("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

- **2.Character Stream**

Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**. Though internally FileReader uses FileInputStream and

FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

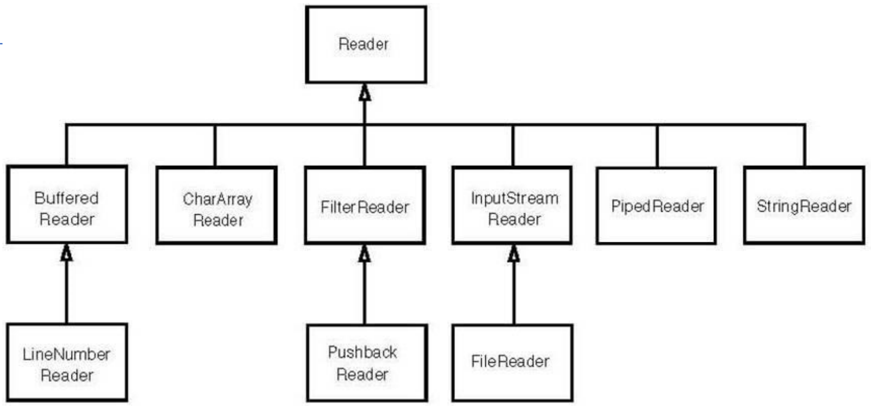
We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file –

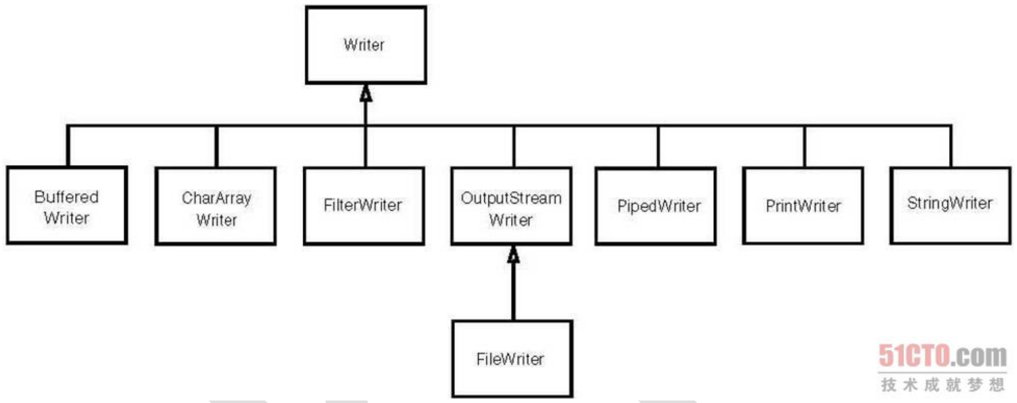
```
import java.io.*;
public class CopyFile {

   public static void main(String args[]) throws IOException {
      FileReader in = null;
      FileWriter out = null;

      try {
         in = new FileReader("input.txt");
         out = new FileWriter("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```





Q5. What is use of BufferedInputStream and BufferedOutputStream class?

Ans. Java.io.BufferedInputStream class in Java

1.A BufferedInputStream adds functionality to another input stream-namely, the ability to buffer the input and to support the mark and reset methods.

2.When the BufferedInputStream is created, an internal buffer array is created. As bytes from the stream are read or skipped, the internal buffer is refilled as necessary from the contained input stream, many bytes at a time.

Methods:

- **int available() :** Returns an estimate of the number of bytes that can be read (or skipped over) from this input stream without blocking by the next invocation of a method for this input stream.
- **void close() :** Closes this input stream and releases any system resources associated with the stream.
- **void mark(int readlimit) :** Marks the current position in this input stream.

Parameters:
readlimit - the maximum limit of bytes that can be read
before the mark position becomes invalid.

- **boolean markSupported() :** Tests if this input stream supports the mark and reset methods.
- **int read(byte[] b, int off, int len) :** Reads bytes from this byte-input stream into the specified byte array, starting at the given offset.

Parameters:
b - destination buffer.
off - offset at which to start storing bytes.
len - maximum number of bytes to read.

- **void reset() :** Repositions this stream to the position at the time the mark method was last called on this input stream.
- **long skip(long n) :**Skips over and discards n bytes of data from this input stream

Parameters:
n - the number of bytes to be skipped.

Program:

```
import java.io.BufferedInputStream; // Java program to demonstrate working of BufferedInputStream
import java.io.FileInputStream;
import java.io.IOException;
class BufferedInputStreamDemo   // Java program to demonstrate BufferedInputStream methods
{
        public static void main(String args[]) throws IOException
    {
         FileInputStream fin = new FileInputStream("file1.txt"); // attach the file to FileInputStream
         BufferedInputStream bin = new BufferedInputStream(fin);
         System.out.println("Number of remaining bytes:" + bin.available());    // illustrating available method

         boolean b=bin.markSupported(); // illustrating markSupported() and mark() method
         if (b)
         bin.mark(bin.available());

         bin.skip(4);           // illustrating skip method
         System.out.println("FileContents :"); // read characters from FileInputStream and
         int ch;
         while ((ch=bin.read()) != -1)
                System.out.print((char)ch);

         bin.reset(); // illustrating reset() method
         while ((ch=bin.read()) != -1)
                System.out.print((char)ch);
         fin.close();         // close the file
    }
}

Output:
Number of remaining bytes:47
FileContents :
 is my first line
This is my second line
This is my first line
This is my second line
```

Java BufferedOutputStream Class

1.Java BufferedOutputStream class is used for buffering an output stream.

2.It internally uses buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

3.For adding the buffer in an OutputStream, use the BufferedOutputStream class.

4.Let's see the syntax for adding the buffer in an OutputStream:

OutputStream os= new BufferedOutputStream(new FileOutputStream("D:\\IO Package\\testout.txt"));

Methods:

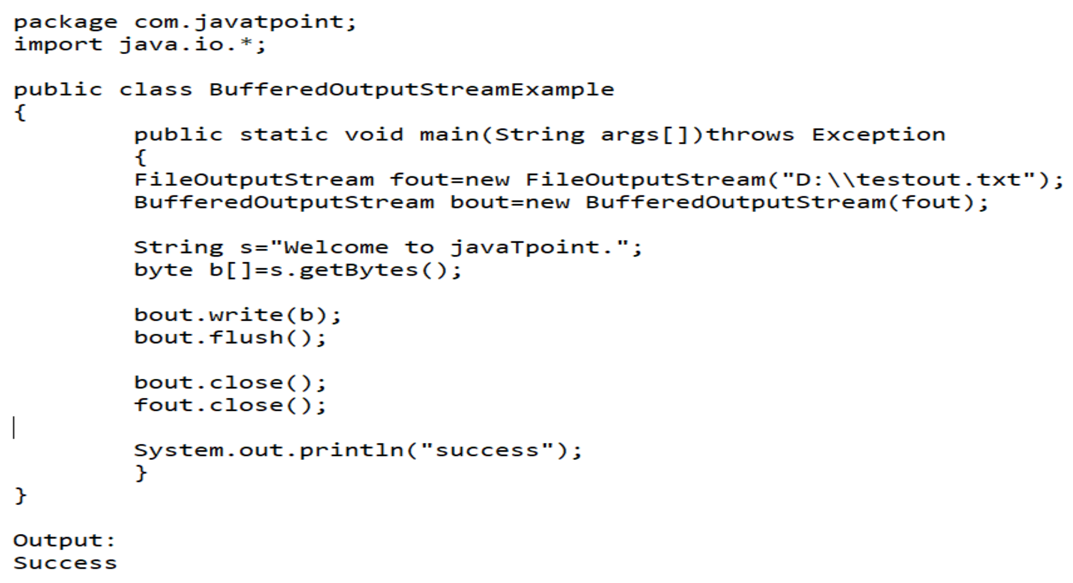
- **void write(int b):** It writes the specified byte to the buffered output stream.

- **void write(byte[] b, int off, int len):** It write the bytes from the specified byte-input stream into a specified byte array starting with the given offset.

- **void flush():** It flushes the buffered output stream.

Example of BufferedOutputStream class:

In this example, we are writing the textual information in the BufferedOutputStream object which is connected to the FileOutputStream object. The flush() flushes the data of one stream and send it into another. It is required if you have connected the one stream with another.



```
package com.javatpoint;
import java.io.*;

public class BufferedOutputStreamExample
{
        public static void main(String args[])throws Exception
        {
        FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
        BufferedOutputStream bout=new BufferedOutputStream(fout);

        String s="Welcome to javaTpoint.";
        byte b[]=s.getBytes();

        bout.write(b);
        bout.flush();

        bout.close();
        fout.close();

        System.out.println("success");
        }
}

Output:
Success
```