UNIT 2: WORKING WITH FILES

Unit Structure

2.0 Learning Objectives

2.1 Introduction

2.2 I/O Streams

2.3 Streams

2.4 Reading Console Input

2.5 Writing Console Output

2.6 Reading and Writing Files

2.7 Serialisation

2.8 Let Us Sum Up

2.9 Suggested Answer for Check Your Progress

2.10 Glossary

2.11 Assignment

2.12 Activities

2.13 Case Study

2.14 Further Readings

2.0 Learning Objectives

After learning this unit, you will be able to:

- Define i/o streams
- Describe reading console input and writing console output
- Explain reading and writing files
- Discuss serialization

2.1 Introduction

The Java I/O is said as Java Input /Output and is a part of java.io package. Now package has an Input Stream and Output Stream. The Java Input Stream is meant for reading the stream, byte stream and array of byte stream. This can be used for memory allocation. The Output Stream is used for writing byte and array of bytes.

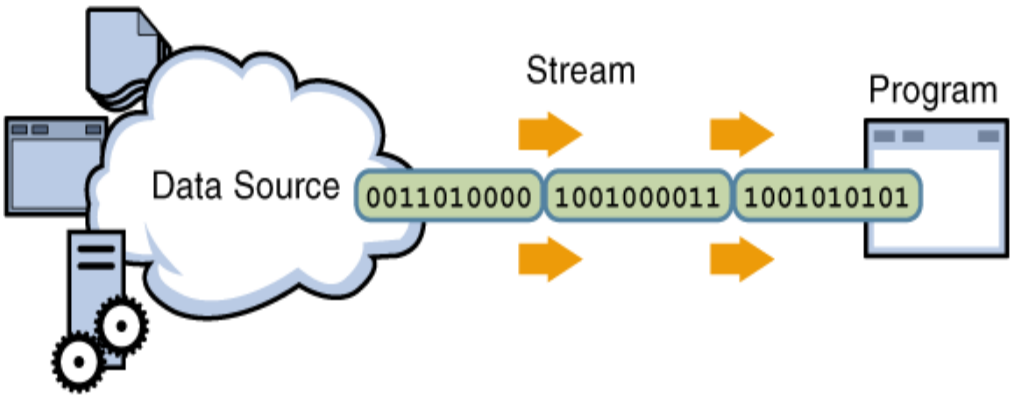
In this lesson, we will learn and understand streams that can handle all kinds of data, from primitive values to advanced objects.

2.2 I/O Streams

An input source or an output target is represented (shown) by an I/O Stream. A stream can represent disk files, devices, other programs and memory arrays and also as many different kinds of sources and destinations target.

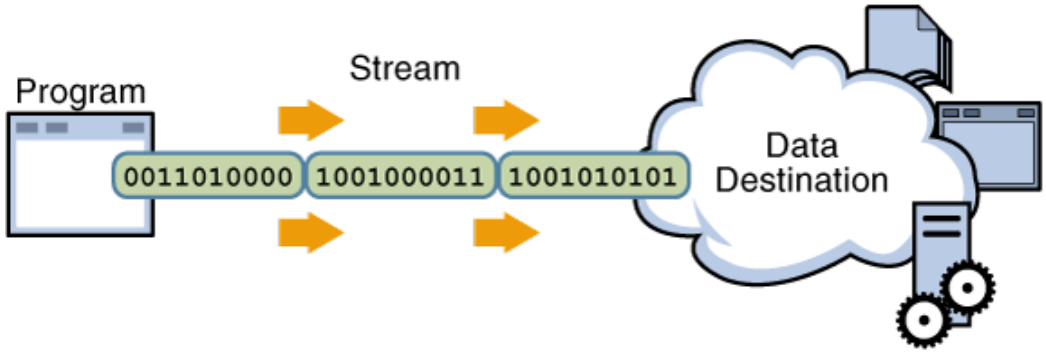
Streams support simple bytes, primitive data types, localized characters and objects as well as many different kinds of data. Some streams simply pass on data while the others manipulate and transform the data in useful ways.

No matter how they work internally, all streams present the same simple model to programs that use them: A stream is a sequence of data. A program uses an input stream to read data from a source, one item at a time:



Reading Information into a Program

A program uses an output stream to write data to a destination, one item at time:



Writing Information from a Program

The data source and data destination pictured in the figure can be anything that stores, generates or consumes data. Obviously, this comprises of disk files but a source or destination can also be another program, a peripheral device, a network socket or an array.

Check your progress 1

1. Explain I/O stream.

.....

.....

.....

.....

.....

.....

.....

.....

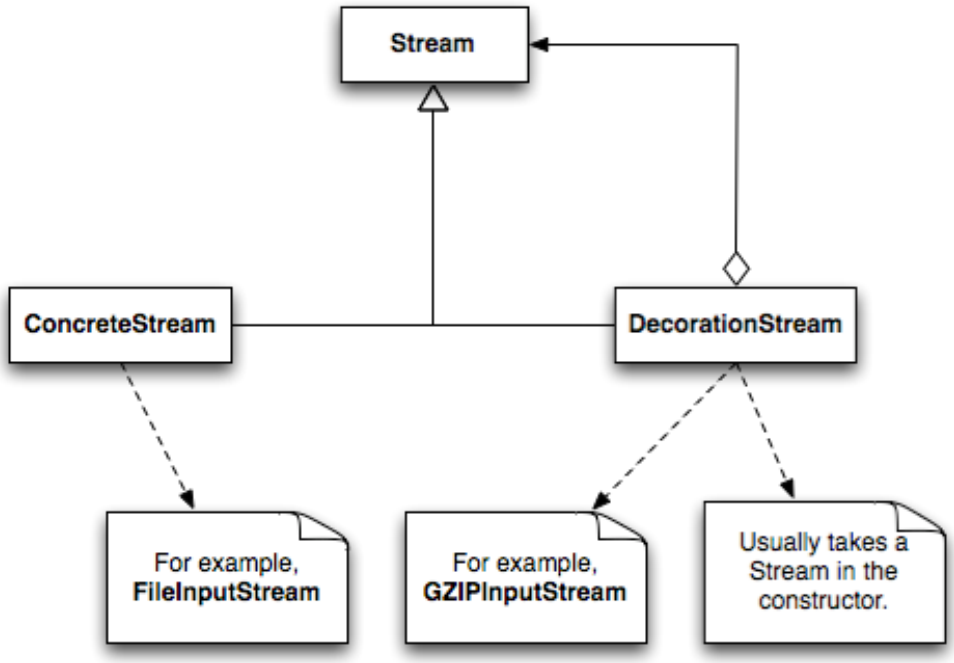
.....

.....

.....

.....

2.3 Streams



Byte Streams:

The Byte streams are used by programs for the purpose of performing input and output of 8-bit bytes. Here all byte stream classes are descended from Input Stream and Output Stream.

We must understand that there are many byte stream classes. To demonstrate how byte streams work, we'll focus on the file I/O byte streams, File Input Stream and File Output Stream. Now other kinds of byte streams are used in a lot the same way; they differ mainly in the way they are constructed or made.

Using Byte Streams:

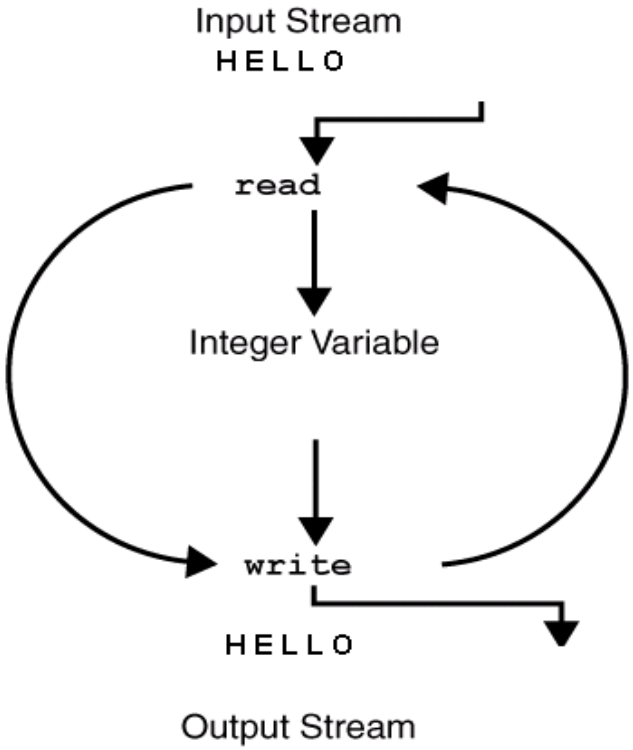
Now we will search File Input Stream and File Output Stream by examining an example program named Copy Bytes, which uses byte streams to copy hello.txt, one byte at a time.

import java.io. File Input Stream

```
import java.io. File Output Stream
import java.io.IO Exception

public class Copy Bytes {
public static void main (String[] args) throws IO Exception {
File Input Stream in = null
File Output Stream out = null
try {
in = new File Input Stream("hello.txt")
out = new File Output Stream("outagain.txt")
int c
while ((c = in.read()) != -1) {
out.write(c)
}
      } finally {
         if (in != null) {
            in close()
         }
         if (out != null) {
            out. close()
         }
      }
   }
}
```

Copy Bytes spends most of its time in a simple loop that reads the input stream and writes the output stream, one byte at a time, as shown in the following figure.



Simple Byte Stream Input and Output

Notice that read () returns an int value. If the input is a stream of bytes, why does not read () return a byte value? Using an int as a return type allows read () to use -1 to indicate that it has reached the end of the stream.

Always Close Streams:

It is very important to close a stream when it is no longer needed. Copy Bytes uses a finally block to guarantee that both streams will be closed even if an error occurs. This practice helps to avoid serious resource leaks.

Copy Bytes being unable to open one or both files is one of the possible errors found. When that happens, the stream variable corresponding to the file never changes from its initial null value. For this reason, Copy Bytes makes sure that each stream variable contains an object reference before invoking close.

When Not to Use Byte Streams:

Copy Bytes seems like a normal program but it actually a representation of a kind of low-level I/O that you should avoid. Since hello.txt contains character

data, the best approach is to use character streams, as discussed in the next section. There are also streams available for more complicated data types. Byte streams should only be used for the most primitive I/O.

So why talk about byte streams? As all other stream types are built on byte streams.

Character Streams:

The character values are stored by Java platformusing Unicode conventions. Character stream I/O automatically translates this internal format to and from the local character set. In Western locales, the local character set is usually an 8-bit superset of ASCII.

I/O with character streams is no more complicated than I/O with byte streams for most applications. Input and output done with stream classes automatically translates to and from the local character set.

Using Character Streams:

All character stream classes are descended from Reader and Writer. As with byte streams, there are character stream classes that specialise in file I/O: File Reader and File Writer. The Copy Characters example illustrates these classes.

```
import java.io. File Reader
import java.io. File Writer
import java.io.IO Exception;

public class Copy Characters {
public static void main(String[] args) throws IO Exception {
File Reader input Stream = null
File Writer output Stream = null
try {
input Stream = new File Reader("xanadu.txt")
```

```
output Stream = new File Writer("characteroutput.txt")

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

Copy Characters is very similar to Copy Bytes. The most important difference is that Copy Characters uses File Reader and File Writer for input and output in place of File Input Stream and File Output Stream. Notice that both Copy Bytes and Copy Characters use an int variable to read to and write from. However, in Copy Characters, the int variable holds a character value in its last 16 bits; in Copy Bytes, the int variable holds a byte value in its last 8 bits.

Character Streams that use Byte Streams:

Character streams are often "wrappers" for byte streams. In order to perform the physical I/O, the character stream uses the byte stream while the translation between characters and bytes is handled by the character stream. File Reader, for example, uses File Input Stream, while File Writer uses File Output Stream.

There are two general-purpose byte-to-character "bridge" streams: Input Stream Reader and Output Stream Writer. Use them to create character streams

when there are no prepackaged character stream classes that meet your needs. The sockets lesson in the networking trail shows how to create character streams from the byte streams provided by socket classes.

Line-Oriented I/O:

Let us modify the Copy Characters example to use line-oriented I/O. To do this, we have to use two classes we have not seen before, Buffered Reader and Print Writer. We will explore these classes in greater depth in Buffered I/O and Formatting. Right now, we're just interested in their support for line-oriented I/O.

The Copy Lines example invokes Buffered Reader. Read Line and Print Writer. println to do input and output one line at a time.

```
import java.io. File Reader
import java.io. File Writer
import java.io. Buffered Reader
import java.io. Print Writer
import java.io.IO Exception

public class Copy Lines {
public static void main(String[] args) throws IO Exception {
Buffered Reader input Stream = null
Print Writer output Stream = null

try {
input Stream =
new Buffered Reader(new File Reader("xanadu.txt"))
output Stream =
new Print Writer(new File Writer("characteroutput.txt"))
```

```
String l
while ((l = input Stream.read Line()) != null) {
output Stream. println(l)
}
} finally {
if (input Stream != null) {
input Stream.close()
}
if (output Stream != null) {
output Stream. close()
        }
      }
   }
}
```

Invoking read Line returns a line of text with the line. With the use of println Copy Lines outputs each line, which appends the line terminator for the current operating system. This might not be the same line terminator that was used in the input file.

There are many ways to structure text input and output beyond characters and lines. For more information, see scanning and Formatting.

Buffered Streams:

Most of the examples use unbuffered I/O. This means the underlying OS handles directly each read or write request. The result is that the program is made much less efficient as each such request often triggers disk access, network activity, or some other operation that is relatively expensive.

The Java platform implements buffered I/O streams in order to reduce this kind of overhead. Buffered input streams read data from a memory area known as a buffer; the native input API is called only when the buffer is empty. Similarly,

buffered output streams write data to a buffer and the native output API is called only when the buffer is full.

There are four buffered stream classes, which can be used to wrap unbuffered streams: Buffered Input Stream and Buffered Output Stream create buffered byte streams, while Buffered Reader and Buffered Writer create buffered character streams.

Flushing Buffered Streams:

It often makes sense to write out a buffer at critical points, without waiting for it to fill. This is known as flushing the buffer.

Some buffered output classes support autoflush, specified by an optional constructor argument. Certain key events cause the buffer to be flushed when autoflush is enabled. For example, an autoflush Print Writer object flushes the buffer on every invocation of println or format. In order to flush a stream manually, invoke its flush method. The flush method is valid on any output stream but has no effect unless the stream is buffered.

Scanning and Formatting:

To assist you with programming I/O that often involves translating to and from the neatly formatted data humans like to work with, the Java platform provides two APIs. The scanner API breaks input into individual tokens associated with bits of data and the formatting API assembles data into nicely formatted, human-readable form.

I/O from the Command Line:

A program is often run from the command line and interacts with the user in the command line environment. The Java platform supports this kind of interaction in two ways: through the Standard Streams and through the Console.

Standard Streams:

Standard Streams are a feature of many operating systems. By default, they read input from the keyboard and write output to the display. They also support I/O on files and between programs but that feature is controlled by the command line interpreter, not the program.

The Java platform supports the following three Standard Streams: Standard Input, accessed through System.in; Standard Output, accessed through System.out; and Standard Error, accessed through System.err. These objects are defined automatically and do not need to be opened. Standard Output and Standard Error are both for output; having error output separately allows the user to divert regular output to a file and still be able to read error messages. You might expect the Standard Streams to be character streams, but, for historical reasons, they are byte streams. System.out and System.err are defined as PrintStream objects. Although it is technically a byte stream, PrintStream utilizes an internal character stream object to emulate many of the features of character streams.

By contrast, System.in is a byte stream with no character stream features. To use Standard Input as a character stream, wrap System.in in InputStreamReader.

Input Stream Reader cin = new InputStreamReader(System.in);

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, Object, localized characters etc.

A stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.

Java does provide strong, flexible support for I/O as it relates to files and networks but this tutorial covers very basic functionality related to streams and I/O. We would see most commonly used example one by one:

Check your progress 2

1. Explain standard streams.
2. Write a program to illustrate the use of byte streams.

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

2.4 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, you wrap System.in in a Buffered Reader object, to create a character stream. Here is most common syntax to obtain Buffered Reader:

```
Buffered Reader br = new Buffered Reader(new
                        Input Stream Reader(System.in))
```

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

Reading Characters from Console

Now let us learn how to read a character from a BufferedReader, we would use read( ) method whose sytax is as follows:

int read( ) throws IOException

Every time that read ( ) is called, then it reads a character from the input stream and then returns it as an integer value. It returns .1 as soon as the end of the stream is encountered. As you can see, it can throw an IOException.

Reading Strings from Console

Now let us see to read a string from the keyboard, first use the version of readLine( ) which is a member of the BufferedReader class. Its general form is shown here:

String readLine( ) throws IOException

Now the time to understand the following program which demonstrates BufferedReader and the readLine( ) method. This program reads and displays lines of text until you enter the word "end":

```
// Read a string from console using a BufferedReader.
import java.io.*
class BRReadLines {
public static void main(String args[]) throws IOException
```

```
  {
    // Create a BufferedReader using System.in
    BufferedReader br = 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"));
  }
}
```

Here is a sample run:

```
Enter lines of text
Enter 'end' to quit
This is line one
This is line one
This is line two
This is line two
end
end
```

Check your progress 3

1. Write a program to read characters from console.
2. Write the syntax to obtain Buffered Reader

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

2.5 Writing Console Output

To write Console output which is most easily accomplished (done) with print( ) and println( ). These methods are defined by the class Print Stream that is the type of the object referenced by System.out. Even if System.out is a byte stream, using it for simple program output is still acceptable.

Because Print Stream is an output stream resultant(derived) from Output Stream, which also implements the low-level method write(). hence, write() can be used to write to the console. The easiest form of write( ) defined by Print Stream is shown as bellow here:

void write(int byteval)

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

Example:

Here is a short example that uses write( ) to output the character "A" followed by a newline to the screen:

```
import java.io.*


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

This would produce simply 'A' character on the output screen.

Note: You will not often use write( ) to perform console output because print( ) and println( ) are substantially easier to use.

Check your progress 4

1. Write the general form of write() defined by Print Stream.
2. How is console output accomplished?

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

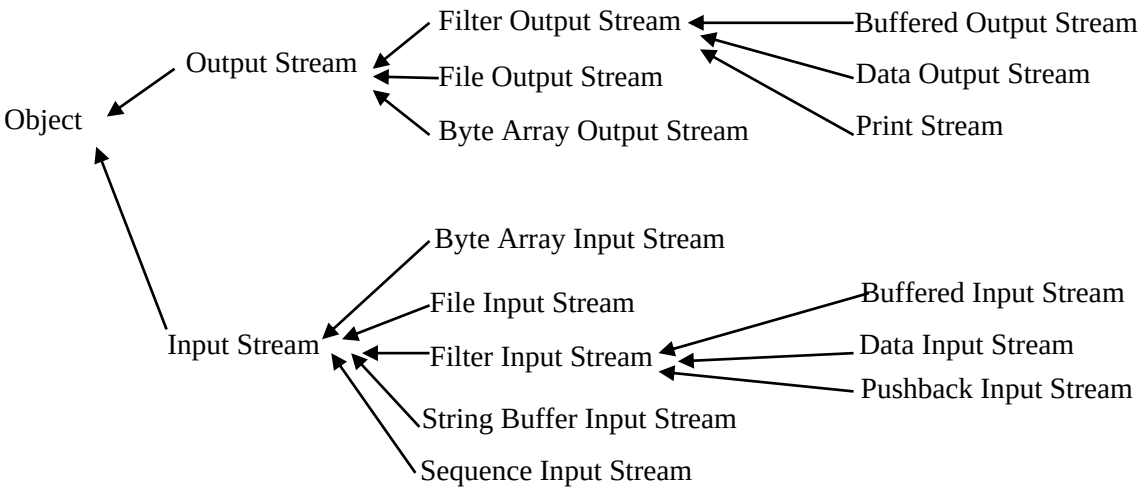
.....

.....

2.6 Reading and Writing Files

Now we explain about, a stream can be defined as a sequence(string) of data. The InputStream which is used to read data from a source and the OutputStream is then used for writing data to a destination (output).

Here, is a hierarchy of classes to deal with Input and Output streams.



Hierarchy of classes to deal with Input and Output streams

The two important streams are File Input Stream and File Output Stream which would be discussed in this section:

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.

Following constructor takes a file name as a string to create an input stream object to read the file:

Input Stream f = new File Input Stream("C:/java/hello");

Following constructor 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");

Input Stream f = new File Input Stream(f)

Once you have Input Stream 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.

Table 10.1: List of helper methods

SN	Methods with Description
1	public void close() throws IOException{} This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	protected void finalise()throws IOException {} This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	public int read(int r)throws IOException{} This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's end of file.
4	public int read(byte[] r) throws IOException{} This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If end of file -1 will be returned.
5	public int available() throws IOException{} Gives the number of bytes that can be read from this file input stream. Returns an int.

There are other important input streams available:

- Byte Array Input Stream
- Data Input Stream

File Output Stream:

File Output Stream is used to create a file and write data into it. The stream would create a file, if it does not already exist, before opening it for output.

Here are two constructors that can be used to create a File Output Stream object.

Following constructor takes a file name as a string to create an input stream object to write the file:

Output Stream f = new File Output Stream("C:/java/hello")

Following constructor takes a file object to create an output stream object to write the file. First we create a file object using File() method as follows:

File f = new File("C:/java/hello")

Output Stream f = new File Output Stream(f)

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

SN	Methods with Description
1	public void close() throws IOException{} This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	protected void finalize()throws IOException {} This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	public void write(int w)throws IOException{} This methods writes the specified byte to the output stream.
4	public void write(byte[] w) Writes w.length bytes from the mentioned byte array to the OutputStream.

There are other important output streams available:

- Byte Array Output Stream
- Data Output Stream

Example:

Following is the example to demonstrate Input Stream and Output Stream:

```
import java.io.*

public class file StreamTest{

public static void main(String args[]){

try{
byte bWrite [] = {11,21,3,40,5}
Output Stream os = new File Output Stream("C:/test.txt")
for(int x=0; x < bWrite.length ; x++){
os.write( bWrite[x] ); // writes the bytes
}
os.close()

InputStream is = new File Input Stream("C:/test.txt");
int size = is.available()

for(int i=0; i< size; i++){
System.out.print((char)is.read() + "  ")
}
is.close()
}catch(IOException e){
System.out.print("Exception")
    }
   }
}
```

The above code would create file test.txt and would write given numbers in binary format. Same would be output on the std out screen.

File Navigation and I/O

There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.

- File Class
- File Reader Class
- File Writer Class

Creating Directories in Java:

There are two useful File utility methods which can be used to create directories:

- The mkdir( ) method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.
- The mkdirs() method creates both a directory and all the parents of the directory.

Following example creates "/tmp/user/java/bin" directory:

```
import java.io.File


class CreateDir {

public static void main(String args[]) {

String dirname = "/tmp/user/java/bin"

File d = new File(dirname)

// Create directory now.

d.mkdirs()

  }

}
```

Compile and execute above code to create "/tmp/user/java/bin".

Note: Java automatically takes care of path separators on UNIX and Windows as per conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly.

Reading Directories:

A directory is a File that contains a list of other files and directories. When you create a File object and it is a directory, the isDirectory( ) method will return true.

You can call list( ) on that object to extract the list of other files and directories inside. The program shown here illustrates how to use list( ) to examine the contents of a directory:

```
import java.io.File


class DirList {
public static void main(String args[]) {
String dirname = "/java"
File f1 = new File(dirname)
if (f1.isDirectory()) {
System.out.println( "Directory of " + dirname)
String s[] = f1.list()
for (int i=0; i < s.length; i++) {
File f = new File(dirname + "/" + s[i])
if (f.isDirectory()) {
System.out.println(s[i] + " is a directory")
} else {
System.out.println(s[i] + " is a file")
}
}
```

```
} else {
System.out.println(dirname + " is not a directory")
  }
 }
}
```

This would produce following result:

```
Directory of /mysql
bin is a directory
lib is a directory
demo is a directory
test.txt is a file
README is a file
index.html is a file
include is a directory
```

Check your progress 5

1. Write the hierarchy of classes to deal with Input and Output streams.
2. Write a note on File Input Stream.

.....

.....

.....

.....

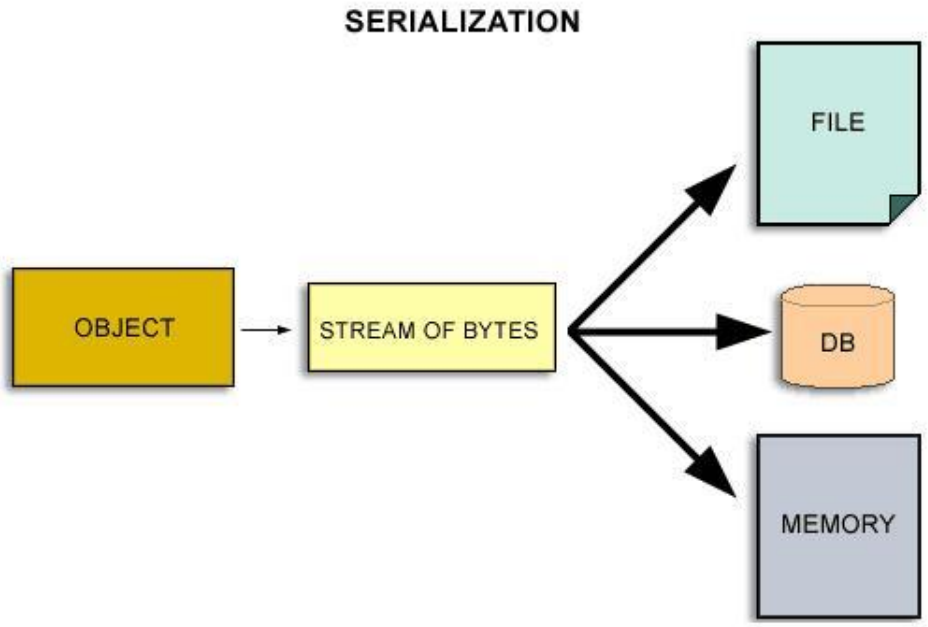
.....

.....

.....

.....

2.7 Serialization



Let us understand that Java provides a mechanism, called object serialization where an object can be represented (shown) as a sequence(string) of bytes that includes the object's data and also information about the object's type and the types of data stored in the object.

After a serialised object has been written into a file, it can be read from the file and deserialised, i.e., the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialised on one platform and deserialised on an entirely different platform.

Classes ObjectInputStream and ObjectOutputStream are high-level streams that contain the methods for serialising and deserialising an object.

The ObjectOutputStream class contains many write methods for writing various data types but one method in particular stands out:

public final void writeObject(Object x) throws IOException

The above method serialises an Object and sends it to the output stream. Similarly, the ObjectInputStream class contains the following method for deserialising an object:

public final Object readObject() throws IOException,

ClassNotFoundException

This method retrieves the next Object out of the stream and deserializes it. The return value is Object, so you will need to cast it to its appropriate data type.

To demonstrate how serialization works in Java, I am going to use the Employee class that we discussed early on in the book. Suppose that we have the following Employee class, which implements the Serialisable interface:

```
public class Employee implements java.io.Serialisable
{
public String name
public String address
  public int transient SSN
  public int number
  public void mailCheck()
  {
 System.out.println("Mailing a check to " + name
                    + " " + address)
  }
}
```

Notice that for a class to be serialised successfully, two conditions must be met:

- The class must implement the java.io.Serialisable interface.
- All of the fields in the class must be serialisable. If a field is not serialisable, it must be marked transient.

If you are curious to know if a Java Satandard Class is serialisable or not, check the documentation for the class. The test is simple: If the class implements java.io.Serialisable, then it is serialisable; otherwise, it is not.

Serializing an Object:

The Object Output Stream class is used to serialize an Object. The following Serialize Demo program instantiates an Employee object and serializes it to a file.

When the program is done executing, a file named employee.ser is created. The program does not generate any output but study the code and try to determine what the program is doing.

Note: When serialising an object to a file, the standard convention in Java is to give the file a .ser extension.

Desterilizing an Object:

The following Deserialise Demo program deserialises the Employee object created in the Serialise Demo program. Study the program and try to determine its output:

```
import java.io.*;
  public class DeserialiseDemo
  {
     public static void main(String [] args)
     {
       Employee e = null
       try
       {
         FileInputStream fileIn =
                    new FileInputStream("employee.ser")
         ObjectInputStream in = new
ObjectInputStream(fileIn)
         e = (Employee) in.readObject()
         in.close()
         fileIn.close()
```

```
        }catch(IOException i)
        {
           i.printStackTrace()
           return;
        }catch(ClassNotFoundException c)
        {
           System.out.println(.Employee class not found.);
           c.printStackTrace()
           return
        }
        System.out.println("Deserialized Employee...")
        System.out.println("Name: " + e.name)
        System.out.println("Address: " + e.address)
        System.out.println("SSN: " + e.SSN)
        System.out.println("Number: " + e.number)
    }
}
```

This would produce following result:

```
Deserialized Employee...
Name: Reyan Ali
Address:Phokka Kuan, Ambehta Peer
SSN: 0
Number:101
```

Check your progress 6

1. Which are the streams that contain methods for serialising and deserialising an object?
2. What do you mean by serialisation?

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

2.8 Let Us Sum Up

This Unit No.2 has got importance because details regarding the files and its Management like Input and output. Java I/O is said as Java Input/ Output and is a part of java.io package. Now package has a Input Stream and Output Stream. The Java Input Stream is meant for reading the stream, byte stream and array of byte stream. This can be used for memory allocation. The Output Stream is used for writing byte and array of bytes.

An input source or an output target is represented (shown) by an I/O Stream. A stream can represent disk files, devices, other programs and memory arrays and also as many different kinds of sources and destinations target. BYTE STREAMS: The Byte streams are used by programs for the purpose of performing input and output of 8-bit bytes. Here all byte stream classes are descended from Input Stream and Output Stream. The character values are stored by Java platformusing

Unicode conventions. Character stream I/O automatically translates this internal format to and from the local character set. In Western locales, the local character set is usually an 8-bit superset of ASCII.

There is learning related to use line-oriented I/O, unbuffered I/O, flushing the buffer, Scanning and formatting and standard stream. Next thing came in learning related to read Console. To write Console output which is most easily accomplished (done) with print( ) and println( ). These methods are defined by the class Print Stream that is the type of the object referenced by System.out. Even even if System.out is a byte stream, using it for simple program output is still acceptable.Now futher we explain how to read and write files.

The Input Stream which is used to read data from a source and the Output Stream is then used for writing data to a destination (output).

Serialization where an object can be represented as a sequence (string) of bytes that includes the object's data and also information about the object's type and the types of data stored in the object. Next we have understood about Serializing an Object which is nothing but the Object Output Stream class is used to serialize an Object and Deserialising an Object.

2.9 Suggested Answer for Check Your Progress

Check your progress 1

Answers: See Section 2.2

Check your progress 2

Answers: See Section 2.3

Check your progress 3

Answers: See Section 2.4

Check your progress 4

Answers: See Section 2.5

Check your progress 5

Answers: See Section 2.6

Check your progress 6

Answers: See Section 2.7

Check your progress 7

Answers: See Section 2.8

2.10 Glossary

1. **Stream** - A stream can represent disk files, devices, other programs and memory arrays and also as many different kinds of sources and destinations target.
2. **Byte streams -** are used by programs for the purpose of performing input and output of 8-bit bytes.
3. **Standard Streams -** Standard Streams are a feature of many operating systems. By default, they read input from the keyboard and write output to the display. They also support I/O on files and between programs but that feature is controlled by the command line interpreter, not the program.
4. **Serialization** - where an object can be represented (shown) as a sequence(string) of bytes that includes the object's data and also information about the object's type and the types of data stored in the object

2.11 Assignment

Write a program to create a file student to store the roll no., name and marks of 3 subjects of 5 students using the features of Java.

2.12 Activities

1. Explain how to create directories in Java.
2. Explain the methods that can be used to read to stream.

2.13 Case Study

Read a file of sums and products (one per line) and write the values of the expressions to another file. The two file names are expected to be given as command-line arguments

2.14 Further Reading

1. Core Java 2, 2 volumes, Cay S. Horstmann, Gary Cornell, The Sun Microsystems Press, 1999, Indian reprint 2000
2. Java 2, The Complete Reference, Patrick Naughton and Herbert Schildt, Tata McGraw Hill, 1999
3. Programming with Java, Ed. 2,E. Balagurusamy, Tata McGraw Hill, 1998, reprint, 2000
4. The Java Tutorial, Ed. 2, 2 volumes, Mary Campione and Kathy Walrath, Addison Wesley Longmans, 1998
5. The Java Language Specification, Ed. 2, James Gosling, Bill Joy, Guy Steele & Gilad Bracha, (Ed. 1 1996, Ed. 2 2000), Sun Microsystems, 2000
6. Using Java 2, Joseph L. Weber, Prentice Hall, Eastern Economy Edition, 2000