Module: Core Java

Session 24: Ad. Java IO

* File Streams
* Character Streams
* Sources and sinks
* Buffering
* Serialization
* Externalizable
* StreamTokenizer

**Objective**

At the end of this chapter you will be able to:

* Learn more about File streams and Character streams
* Know how is the process of work in Sources and sinks
* Learn how Buffering action is done
* Know about object serialization, Externalizable, and StreamTokenizer

#### File Streams

Most of the examples in this chapter have used the streams System.in and System.out. These are convenient for examples, but in real life, you will more commonly attach streams to data sources like files and network connections. You will use the java.io.FileInputStream and java.io.FileOutputStream classes, which are concrete subclasses of InputStream and OutputStream, to read and write files.

### Reading Files:

FileInputStream is a pure subclass of InputStream that provides an input stream connected to a particular file.

public class FileInputStream extends InputStream

FileInputStream has all the usual methods of input streams, such as read(), available(), skip(), and close().These are used exactly as they are for any other input stream. There are three FileInputStream() constructors:

public FileInputStream(String fileName) throws IOException

public FileInputStream(File file) throws FileNotFoundException

public FileInputStream(FileDescriptor fd)

The following program reads “Sample.txt” file and prints it back.

import java.io.\*;

public class ReadBytes

{

public static void main(String[] args) {

try {

FileInputStream fis = new FileInputStream("C:/Sample.txt");

int n;

while ((n = fis.available()) > 0) {

byte[] b = new byte[n];

int result = fis.read(b);

if (result == -1) break;

String s = new String(b);

System.out.print(s);

} // End while

} // End try

catch (IOException e)

{

System.err.println(e);

}

System.out.println();

}}

### Writing Files:

The FileOutputStream class is a concrete subclass of OutputStream that provides output streams connected to files.

public class FileOutputStream extends OutputStream

This class has all the usual methods of output streams, such as write(), flush(), and close().They are used exactly as they are for any other output stream. There are three main FileOutputStream() constructors:

public FileOutputStream(String filename) throws IOException

public FileOutputStream(File file) throws IOException

public FileOutputStream(FileDescriptor fd)

The following program creates a file called “Sample.txt” and writes a string into the file.

import java.io.\*;

public class WriteBytes

{

public static void main(String[] args) {

try

{

File f=new File("C:/Sample.txt");

FileOutputStream fos = new FileOutputStream(f);

String str="HAPPY B DAY TO U";

byte []b=str.getBytes();

for(int n=0;n<b.length;n++)

{

fos.write(b[n]);

}

}

catch (IOException e)

{

System.err.println(e);

System.out.println();

}

}

}

#### Character Streams

### Reader class

You know that Readers are character based input streams that can read Unicode characters.

**The read()** method reads a single character and also returns a character that can be read as an integer in the range from 0 to 65535 or it can be -1 if the end of the stream is reached.

**The abstract read()** method reads characters into a portion of an array that starts at the offset up to length number of characters. It returns the number of characters read or –1 only if the end of the stream is reached.

### Character input streams

Java.io package has different character input streams. They are:

If you look at the different character input streams in the java.io package, they are

* Strings
* Character arrays
* Pipes

As InputStreamReader is a character input stream, it uses a byte input stream as its data source, converting it into Unicode characters. LineNumberReader is a character input stream and a subclass of BufferedReader that tracks the number of lines of text that have been read from it. PushbackReader, is a character input stream and a subclass of FilterReader, that uses another input stream as its input source. It also adds the ability to push characters back onto the stream.

The following example used the InputStreamReader to read its source file.

import java.io.\*;

public class IsrDemo{

public static void main(String argv[]){

try{

FileInputStream in = new FileInputStream("c:/IsrDemo.java");

InputStreamReader isr = new InputStreamReader(in);

int ch=0;

while((ch = in.read())> -1){

StringBuffer buf = new StringBuffer();

buf.append((char)ch);

System.out.print(buf.toString());

}

} catch (IOException e){System.out.println(e.getMessage());}

}

}

### Writer class:

The writer class consists of character based output streams called writers that can write character bytes and turn Unicode into bytes. The methods that includes these methods are as follows:

* The void write() method – They take a character and writes single character in 16 low-order bits.
* The abstract void write() method – They take a character array and writes a portion of an array of characters

### Character output streams

In the java.io package, you will find many character output streams. There you can view branches of this inheritance tree. It has not only Readers, but also other important branches available.

The Writer output having Sinks can be:

* Strings
* CharArray
* Pipes

The destination for the data in OutputStreamWriter is used with the help of byte output stream. Buffering is employed by BufferedWriter to a character output stream. Therefore, it enhances the efficiency of the output by combining many small write requests into a single large request. An abstract class FilterWriter functions as a superclass for character output streams. The stream helps in filtering the data written to them before writing it to some other character output stream.

PrintWriter is a character output stream having print() and println()methods. These methods give textual representations of primitive values and objects as output.

#### Sources and sinks

### Introduction

Here you will know what are sources and sinks and how they function. There will be codes to helps you understand how some of these classes work.

* Byte arrays
* Pipes
* Sequences
* Char arrays

### Byte arrays

The creation of ByteArrayInputStream object is passed as an array. The bytes from this array can be read as an InputStream. ByteArrayOutputStream writes to a byte array. If bytes are written to the output stream, they are added to an internal array. It is obvious that you will think of the way of getting to the byte array. At any point in time The toByteArray() method returns you the byte array. The byte array in the form of a String is returned by the available toString() method.Therefore, the byte array can be either a source or a sink as in Fig. 3.



Fig. 3: Byte arrays as a source or a sink

### Pipes in Java code

The two threads of a Java program have the facility to communicate by means of a pipe. One of the threads can write to a piped output stream and the other thread reads from a piped input stream, connected to the piped output stream. The pipe work between two processes in UNIX and MS-DOS is same as the pipe work between two threads. With those operating systems, there can be output from one process piped to the input of another process. Take the example of MS-DOS with the command "dir | more". This command pipes the output of the "dir" command to the input of the "more" program.

### Sequences

SequenceInputStream aggregates input streams and this stream reads each input stream till its completion. SequenceInputStream can be used to read two or three streams considering them as one stream. There are various sources to concatenate streams. You can take two or more files. Enumeration, which represents a set of input streams, can help in constructing a SequenceInputStream. You can also specify two streams for the same work. A program in UNIX like the "cat" can be used to do something like this. SequenceInputStream starts reading from the first underlying input stream that is given to it. After the exhaustion of the first one, it opens the next input stream and reads the same. When the same gets exhausted, it reads the next, and so on. But in between, the users are unaware of when the input is transferred from one stream to the next stream. Another byte is simply read. After the reading of all the input streams, the entire input stream receives an EOF.

### Char arrays

To start with Readers and Writers classes, we have read and write methods to input and output a character respectively. If you recollect the function of ByteArrayOutputStream, it is easy to remember CharArrayWriter that can write characters to a char array in the same way. When a character is written to a CharArrayWriter object, it is added to an array of characters that automatically increments its size. Any time, you can get the character array filled up by you. The method called toCharArray() returns an array of characters. The character array is used as a source by a CharArrayReader. With a CharArrayWriter object, there is also an array that has been created. Take an alternate constructor, where you can specify the array, the position of starting in the array (an offset) and the number of bytes to read (a length) before you return an EOF character. See Fig.4:



Fig. 4: Chararcter arrays as a source or a sink

### InputStreamReader

InputStreamReader, which is discussed earlier, reads bytes from an InputStream and converts them to characters. An InputStreamReader uses the default character encoding for the bytes, which is usually ASCII. If the bytes that are being read are ASCII bytes, only a single byte at a time is used to form a character. If the bytes are not ASCII, such as Chinese or another language, you want its conversion to Unicode as well. Specific encoding of the byte stream is necessary, and the InputStreamReader converts it to Unicode. With an alternate constructor, you can specify the encoding of the bytes on the InputStream.

### OutputStreamReader

OutputStreamWriter is similar to InputStreamReader. The output characters, which are in Unicode, are converted to the underlying format of the machine using an OutputStreamWriter. The converted characters are written to an OutputStream. The underlying default format is typically ASCII. However, you can state a specific encoding scheme with an alternate constructor.

#### Buffering

The InputStream class also defines methods for reading several bytes of data in one step into an array of bytes. However, InputStream provides no convenient methods for reading other types of data, such as int or double, from a stream. This is not a problem because you'll never use an object of type InputStream itself. Instead, you'll use subclasses of InputStream that add more convenient input methods to InputStream's rather primitive capabilities. Similarly, the OutputStream class defines a primitive output method for writing one byte of data to an output stream, the method

The Reader and Writer classes provide very similar low-level read and write operations. But in these character-oriented classes, the I/O operations read and write char values rather than bytes. In practice, you will use sub-classes of Reader and Writer, as discussed below.

Reading and writing to a file, getting data one byte at a time is slow and painstaking. One way to speed up the process is to put a wrapper around the file and put a buffer on it .A buffer can increase efficiency by reducing the number of physical read or write operations that correspond to read( ) or write( ) method calls. You create a buffered stream with an appropriate input or output stream and a buffer size.

### How Buffering works:

Buffered input streams as shown in Fig.5,read more data than they initially need into a buffer (an internal array of bytes). When one of the stream's read( ) methods is invoked, data is removed from the buffer rather than from the underlying stream. When the buffer runs out of data, the buffered stream refills its buffer from the underlying stream. Likewise, buffered output streams store data in an internal byte array until the buffer is full or the stream is flushed; then the data is written out to the underlying output stream in one swoop. In situations where it is almost as fast to read or write several hundred bytes from the underlying stream, as it is to read or write a single byte, a buffered stream can provide a significant performance boost.

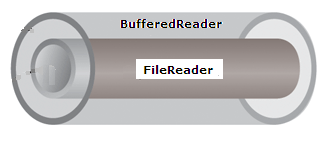


Fig. 5: Buffering of Streams

A program can convert an unbuffered stream into a buffered stream using the wrapping, where the unbuffered stream object is passed to the constructor for a buffered stream class, for example,

inputStream =

new BufferedReader(new FileReader("xanadu.txt"));

outputStream =

new BufferedWriter(new FileWriter("characteroutput.txt"));

This section introduces buffering and covers the following topics:

* BufferedInputStream and BufferedOutputStream
* BufferedReader and BufferedWriter
* New-line characters
* Buffering input and output streams

### BufferedInputStream

Our BufferedInputStream is going to put a buffer onto an InputStream that is specified in the constructor. The actual data source is what you pass it as an InputStream. The BufferedInputStream reads large chunks of data from the InputStream. Then you read individual bytes or small chunks of bytes from the BufferedInputStream. The default buffer size is 512 bytes, but there's a constructor that allows you to specify the buffer size if you want something different.

To improve your efficiency, you read from the object of BufferedInputStream instead of reading directly from the underlying InputStream. And you will not have to go back to the operating system to read individual bytes. See Fig. 6:



Fig. 6: Buffering

Here is an example of using the BufferedInputStream, note how similar it is with the previous example when InputStreamReader is replaced by BufferedInputStream:

import java.io.\*;

public class BufIn{

public static void main(String argv[]){

try{

FileInputStream fin = new FileInputStream("c:\BufIn.java");

BufferedInputStream bin = new BufferedInputStream(fin);

int ch=0;

while((ch=bin.read())> -1){

StringBuffer buf = new StringBuffer();

buf.append((char)ch);

System.out.print(buf.toString());

}

}catch(IOException e){System.out.println(e.getMessage());};

}

}

### BufferedOutputStream

BufferedOutputStream extends FilterOutputStream. When you apply it to an OutputStream, you have a buffered output stream. Instead of going to the operating system for every byte you write, you have the intermediary that provides a buffer and you write to that. When that buffer is full, it is written all at once to the operating system. And it is written automatically if the buffer gets full, if the stream is full, or if the flush() method is used. The flush() method forces any output buffered by the stream to be written to its destination. So for creating a BufferedOutputStream, you have to specify:

* The output stream you are going to use.
* The buffer size if you don't like the default.



Fig. 7: BufferedOutputStream

### BufferedReader and BufferedWriter

A BufferedReader and a BufferedWriter act like BufferedOutputStream and BufferedInputStream, except they deal with reading and writing characters. For a BufferedReader, you specify an underlying Reader and optionally a buffer size. For a BufferedWriter, you specify an underlying Writer and optionally a buffer size. BufferedReader has one additional method, called readLine(), which allows us to simply read an entire line of characters from the underlying Reader.

#### Serialization

Serialization of a class is enabled by the class implementing the java.io.Serializable interface. Classes that do not implement this interface will not have any of their state serialized or deserialized. All subtypes of a serializable class are themselves serializable. The serialization interface has no methods or fields and serves only to identify the semantics of being serializable. There are a couple of other features of a serializable class. First, it has to have a zero parameter constructor. When you read the object, it needs to be able to construct and allocate memory for an object, and it is going to fill in that memory from what it has read from the serial stream. The static fields, or class attributes, are not saved because they are not part of an object. If you do not want a data attribute to be serialized, you can make it transient.

An Example of a serialized class is provided below:

import java.io.\*;

public class Student implements Serializable

{

int roll;

String name;

transient String phone;

}

**The ObjectOutptStream class**

An ObjectOutputStream writes primitive data types and graphs of Java objects to an OutputStream. Only objects that support the java.io.Serializable interface can be written to streams. The default serialization mechanism for an object writes the class of the object, the class signature, and the values of all non-transient and non-static fields. References to other objects (except in transient or static fields) cause those objects to be written also. Multiple references to a single object are encoded using a reference sharing mechanism so that graphs of objects can be restored to the same shape as when the original was written.

For example to write an object of the student class into a file that can be read by the example in ObjectInputStream:

import java.io.\*;

public class ObjectWriter

{

public static void main(String []s) throws Exception

{

Student st=new Student();

st.roll=12;

st.name="Amitav";

st.phone="23467549";

File f=new File("C:/Demo.txt");

f.createNewFile();

FileOutputStream fos = new FileOutputStream(f);

ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeObject(st);

}

}

**ObjectInputStream:**

An ObjectInputStream deserializes primitive data and objects previously written using an ObjectOutputStream. The method readObject is used to read an object from the stream. Java's safe casting should be used to get the desired type. In Java, strings and arrays are objects and are treated as objects during serialization. When read they need to be cast to the expected type.

Sample code for reading the object stored in the previous example:

import java.io.\*;

public class ObjectReader

{

public static void main(String []s) throws Exception

{

File f=new File("C:/Demo.txt");

FileInputStream fis = new FileInputStream(f);

ObjectInputStream oos = new ObjectInputStream(fis);

Object ob=oos.readObject();

Student st=(Student)ob;

System.out.println("The roll is " + st.roll);

System.out.println("The name is " + st.name);

System.out.println("The phone number is " + st.phone);//transient member

}

}

#### Externalizable

Externalizable interface are implemented by a class to give the class complete control over the format and contents of the stream for an object and its supertypes. These methods must explicitly coordinate with the supertype to save its state. These methods supercede customized implementations of writeObject and readObject methods.

#### StreamTokenizer

StreamTokenizer breaks the input stream into tokens using *whitespace* as a delimiter. By default, Unicode characters \u0000 through \u0020 are considered whitespace. This encompasses things like space, tab, newline, etc. If you want to change this list, you need to invoke the method whitespaceChars(int low, int high); all characters having Unicode values between low and high will be considered whitespace, *in addition to* the default set.

You can call whitespaceChars() any number of times - each invocation will add to the list of whitespace characters. The only way to clear out the list is to set those characters to be something other than whitespace - you might use ordinaryChar(int ch), ordinaryChars(int low, int high), wordChars(int low, int high), or resetSyntax() to do this.

The following program is a very simple example of using StreamTokenizer to parse a text file into words, number, and characters. The file to be parsed is taken from the first argument; the second argument is a string containing all the characters to use as delimiters.

import java.io.\*;

public class TokenizeIt {

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

IOException {

FileReader file = new FileReader(args[0]);

BufferedReader in = new BufferedReader(file);

StreamTokenizer st = new StreamTokenizer(in);

char[] c = args[1].toCharArray();

for (int i=0; i<c.length; i++) {

System.out.println("Whitespace will include '" + c[i] + "'");

st.whitespaceChars(c[i], c[i]);

}

int tokval;

while ((tokval = st.nextToken()) != StreamTokenizer.TT\_EOF) {

switch (tokval) {

case StreamTokenizer.TT\_WORD:

System.out.println("Word token \"" + st.sval + "\"");

break;

case StreamTokenizer.TT\_NUMBER:

System.out.println("Number token \"" + st.nval + "\"");

break;

default:

System.out.println("Character '" + (char) tokval + "'");

break;

}

}

}

}

For example, if the input is delimited by commas and colons, you would run this using the command line:

java TokenizeIt ",:"

**Summary**

In this chapter you have learned to work with streams. We used character streams to handle text and byte streams for any other kind of data. The java.io package consists of all the I/O related classes. Streams are very powerful ways to extend the functionality of your java programs because they offer a connection to any kind of data you want to work with.

After going through this chapter you will know that:

* Serialization is the process for storing the object state into the persistent storage device. Object serialization is a technique that allows you to record the values of an object as a series of bytes that can be passed along the streams. Objects can be written to & read from streams with the help of java classes.
* Deserialization is the process for retrieving the object state from the file.
* Externalizable interface are implemented by a class to give the class complete control over the format and contents of the stream for an object and its supertypes.
* StreamTokenizer breaks the input stream into tokens using whitespace as a delimiter.