CS F213
Object Oriented Programming

BITS Pilani
Hyderabad Campus

Prof.R.Gururaj
CS&IS Dept.

Java Input/Output

Ch.21 of The Complete Reference- Java, 11th Edition, Herbert Schildt, Tata McGraw Hill Publishing.

And also refer to Class notes.

Content

1. Introduction to IO
2. Streams
3. Various facilities

Introduction

java.io package

Supports Java's basic Input and Output System, including File IO.

So far we have seen only *System.out.print()* and *println()*

Text based console I/O is not so important in Java.

All fundamental I/O in Java is based on *streams*.

A streams is an abstraction that either produces or consumes information.

A stream represents a flow of data , or a channel of communication (with at least conceptually) a writer at the one end and the reader at the other end.

A stream is linked to a physical device by the Java IO system.

All streams behave in the same way independent of the device it is associated with.

Thus the same IO classes and methods can be applied to any device.

An input stream can abstract many different devices like- keyboard, file, network socket etc.

Similarly an Output stream can abstract many different devices like- monitor, file, network connection etc.

Streams are clean way to deal with I/O without having every part of your code understand the difference between the devices.

Java implements streams within class hierarchies defined in

java.io package.

Java defines two types of streams:

Byte oriented Streams

Character oriented Streams (Java 1.1)

At the lowest level all I/O is byte-oriented.

Byte Stream classes

Abstract classes:

– InputStream

– OutputStream

Each of these have several concrete classes that handle the difference between devices.

Abstract classes InputStream and OutputStream define several key methods that other stream classes implement.

Ex: *read()* and *write()*

OutputStream
- ByteArrayOutputStream
- FileOutputStream
- FilterOutputStream
  - PrintStream
  - BufferOutputStream
  - DataOutputStream
- ObjectOutputSrtream
- PipedOutputStream

InputStream
- ByteArrayInputStream
- FileInputStream
- FilterInputStream
  - BufferedInputStream
  - DataInputStream
- ObjectInputSrtream
- PipedInputStream
- StringBufferInputStream

RandomAccessFile

File

FileDescripter

Character Stream classes

Abstract classes:

– Reader

– Writer

Each of these have several concrete classes that handle the difference between devices.

Abstract classes Reader and Writer define several key methods that other stream classes implement.

Ex: *read()* and *write()*

Reader

- BufferedReader
- CharArrayReader
- FiltereReader
- FileReader
- IntputSrtreamReader
  - FileReader
- PipedReader
- StringReader

Writer

- BufferedWriter
- CharArrayWriter
- FiltereWriter
- OutputSrtreamReader
  - FileWriter
- PipedWriter
- StringWriter
- PrintWriter

Predefined streams

The *System* class encapsulates several aspects of runtime environment.

We can query various properties and settings of the system.

It also contains three predefined stream variables.

1. in   refers to standard input stream - keyboard
2. out   refers to standard output stream - console
3. err   standard error stream- console

System.in is an object of type InputStream

System.out *and* *System.err* are objects of type PrintStream

Reading console input

```
import java.io.*;
class Demo
{     public static void main ( String args [ ])
       {
       BufferedReader  br= new BufferedReader (new InputStreamReader(System.in));
       String line ;
        try   {
                while   ( !( line = br. readLine( )).equals("stop"))
                            System.out.println  (line);
                }
        catch  (IOException  e)
                { System .out. println("Exception :  " +e);}
      }
}
```

```
C:\Users\admin\FS2019>java Demo
Hi
Hi
Hello
Hello
stop
```

```
import java.io.*;
public class MyEditor
{     public static void main ( String args [ ])
      {  BufferedReader  br= new BufferedReader (new InputStreamReader(System.in));
          String str[]=new String[100];
          System.out.println  (" Enter lines of Text: ");    System.out.println  (" Enter 'stop' to exit: ");
          for (int i=0; i<100; i++)
          {     try   {                 str[i] = br. readLine( );
                              if(str[i].equals("stop")) break;
                }
              catch  (IOException  e)
             { System .out. println("Exception :   " +e);}
           }
          System.out.println  (" \n Now printing the file you edited: ");
          for (int i=0; i<100; i++)
          {                   if(str[i].equals("stop")) break;    System.out.println(str[i]);
          }
    }
}
```

```
C:\Users\admin\FS2019>java MyEditor
 Enter lines of Text:
 Enter 'stop' to exit:
Hello
friends how are you doing
where are you now
Let us meet
stop

 Now printing the file you edited:
Hello
friends how are you doing
where are you now
Let us meet
```

Reading and writing to files

Byte oriented classes

Character oriented classes(jdk 1.1)

FileInputStream and FileOutputStream

- Create byte streams linked to files.

FileInputStream(String file) throws FileNotFoundException

FileOutputStream(String file) throws FileNotFoundException

close()

read() reads each byte as an integer value and –1 for EOF

File

File

Deals with file and file systems. This does not specify how info is stored and accessed in files.

This describes the file itself.   A directory is also treated as a file.

Constructors:

File(String path)

File(String dir, String file)

File (File dir, String file)

Methods:

boolean renameTo(File newName)

boolean delete()

boolean setReadOnly()

String getName()

String getParent()

boolean isFile()

boolean isDirectory()

String getAbsolutePath()

Other Methods:

long getFreeSpace()

long getTotalSpace()

boolean isHidden()

boolean setReadonly()

lastModified()

DATA STREAMS

DataInputStream and DataOutputStream

Java also supports non-text data files.

The DataOutputStream class has methods for writing primitive Java types to a stream in a portable way.

You can use the DataInputStream class to read them back .

DataInputStream and DataOutputStream are filtered streams that you read or write strings and primitive data types that comprise more than a single byte.

RandomAccessFiles

The input and output streams that we've been learning about so far in this lesson have been sequential access streams-- streams whose contents must be read or written sequentially.

Random access files, on the other hand , permit non-sequential, or random , access to the contents of a file. Random access Files are useful for many different applications.

The RandomAccessFile class implements both the DataInput and DataOutput interfaces and therefore can be used for both reading and writing .

This line of Java code creates a RandomAccessFile to read the file named data.txt.

```
new RandomAccessFile("data.txt","r");
```

And this one opens the same file for reading and writing (you have to be able to read a file in order to write it):

```
new RandomAccessFile("data.txt", "rw");
```

I/O methods that implicitly move the file pointer explicitly manipulating the file's file pointer.

skipBytes(int n)

Moves the fiel pointer forward the specified number of bytes.

seek(long pos)

Positions the file pointer just before the specified byte

getFilePointer( )

Returns the current location of the file pointer (in bytes).

To read from a file using Reader

```
 public class Demo
{    public static void main ( String args [ ])
      {
              FileReader  fr= new FileReader(“temp.txt”);
              BufferedReader br=new BufferedReader(fr);
              String s;
              while   ( ( s= br. readLine ( )) != null)
              {
              System.out.print (s);
              }
              br.close();
     }
}
```

Serialization

Is the process of writing the state of an object a byte stream.

This is useful when we want to store our program state to a persistent storage area.

If we serialize an object all dependent objects are recursively serialized.

Only an object that implements **Serializable** can be saves.

Whenever needed we can deserialize the object to get the original object.

Serializable interface defines no methods.

It just indicates that a class can be serialized.

If a class is serializable all its sub classes are serializable.

Ex: classes Hashtable, Vector etc. implement **Serializable** interface.

ObjectOutputStream extends **OutputStream** and implements **ObjectOutput** interface

ObjectInputStream extends **InputStream** and implements **ObjectInput** interface

ObjectOutputStream(OutputStream os)

final void writeObject(Object obj)

ObjectInputStream(InputStream os)

final Object readObject()

```
import java.io.*;
class ObjectStreamDemo
{
     public static void main(String org[])
    {   Box b1=new Box(10,20,30);
       try{
       FileOutputStream fos=new FileOutputStream("C:\\users\\Admin\\JavaPrograms\\obj.txt");
       ObjectOutputStream oos=new ObjectOutputStream(fos);
       oos.writeObject(b1);
       }catch(Exception e){}
     }
}
class Box implements Serializable
{int l,b,h;
 Box(int m, int n, int o)
 { l=m; b=n; h=o;}
}
```

```
class ObjectDemo1
{
    public static void main(String org[])
    {
        try{
      FileInputStream fis=new FileInputStream("
      C:\\users\\Admin\\JavaPrograms\\sample.txt ");
      ObjectInputStream ois=new ObjectInputStream(fis);
      Box b1=(Box)ois.readObject();
      System.out.println(" Object b1 L B H is :"b1.l+"  "b1.b+"  " +b1.h);
      }catch(Exception e){}
     }
}
```

Summary

- ❑ Concept of Streams in Java
- ❑ Byte / char oriented Streams
- ❑ Data streams
- ❑ File operations
- ❑ Random access files
- ❑ Object Input and Output Streams