File Handling

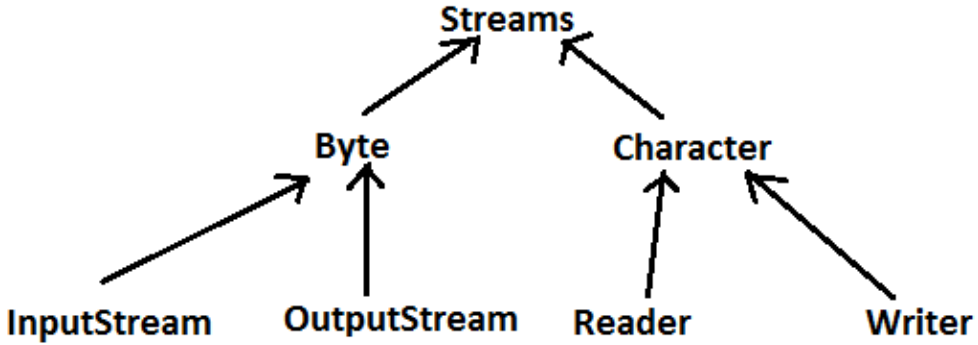
Contents

Module	Topic
Module 1	Introduction
Module 2	Byte & character streams
Module 3	Java IO class hierarchy
Module 4	File reading & writing
Module 5	Piped streams
Module 6	Sequence streams
Module 7	Handling primitive data
Module 8	Using RandomAccessFile
Module 9	Object Serialization

Introduction

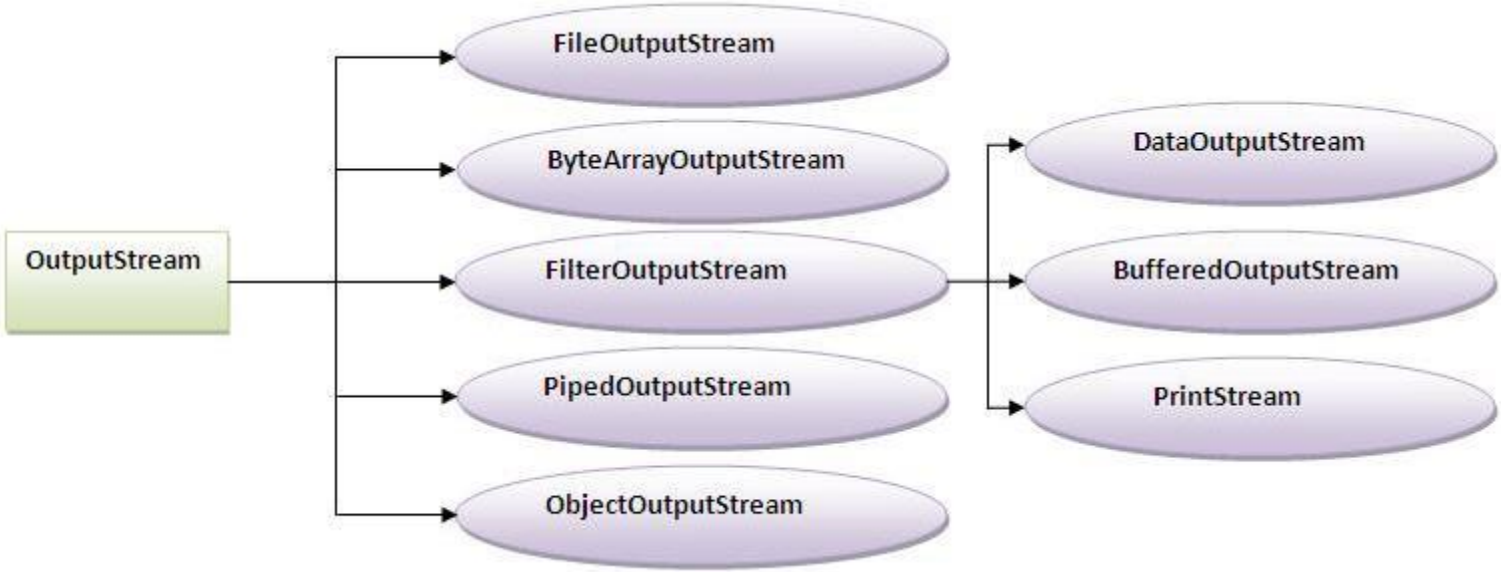
- File handling is used to read & manipulate different files on hard disk.
- Java introduces IO APIs for file handling & provides an important package called java.io.
- For IO operations, java uses concept called ‘Streams’. Stream is a sequence of data in byte format.
- Operation that reads the file stream is called as InputStream where as writing into file stream is called as OutputStream.

Byte & character streams

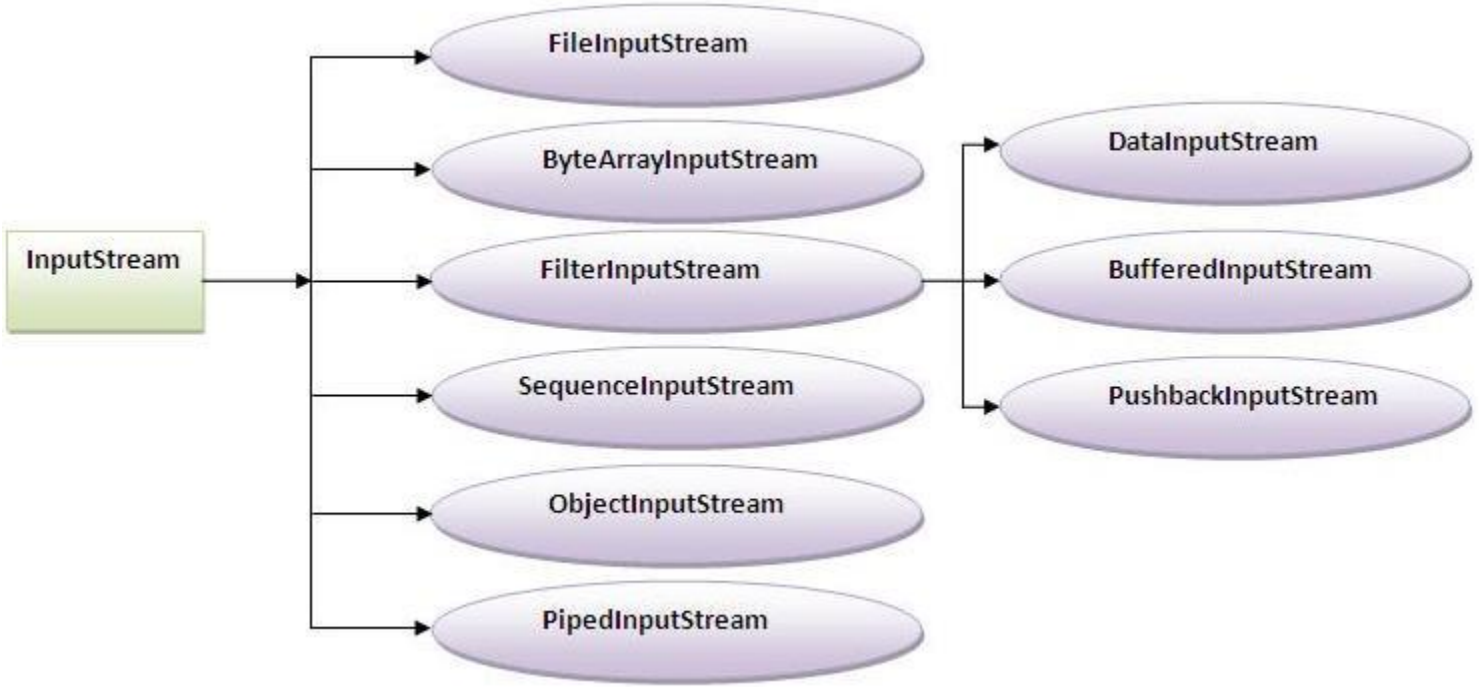


- Java offers us two types of stream classes. i.e. byte stream & character stream.
- Byte stream is a generic stream & useful for handling binary data where as character stream is used for handling text based data.
- Java IO APIs provide us four major classes i.e. two for byte stream (InputStream, OutputStream) & two for character stream (Reader, Writer).
- Note that all four classes are abstract class & hence we do not directly use them. Instead we use their sub classes.

OutputStream hierarchy



InputStream hierarchy



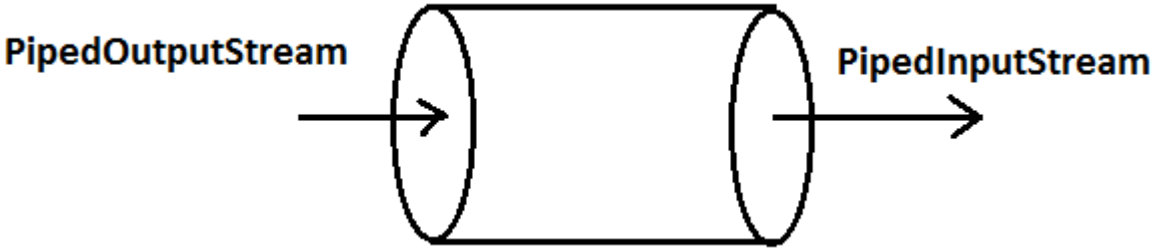
File writing operation

```
import java.io.*;

File file = new File(“c:/temp/abc.txt”);
FileOutputStream fos = new FileOutputStream(file);
String strData = “Hello Java”;
fos.write(strData.getBytes());
fos.flush();
fos.close();
```

File reading operation

```
File file = new File(“c:/temp/abc.txt”);
FileInputStream fis = new FileInputStream(file);
StringBuilder sb = new StringBuilder(“”);
int i = 0;
do {
        i = fis.read();
        if(i != -1)
                sb.append((char)i);
}
while(i != -1); // -1 represents end of file (EOF)
System.out.println(“File contents: “ + sb);
fis.close();
```

Piped streams



- Using piped streams, two threads can communicate with each other.

Piped streams continue…

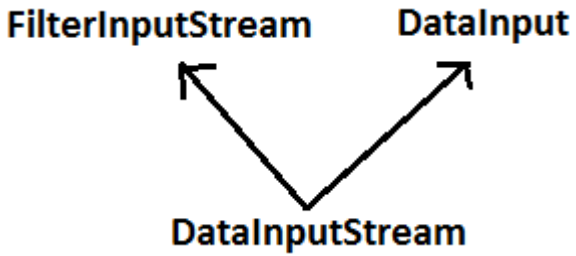
```
final PipedOutputStream output = new PipedOutputStream();
final PipedInputStream  input  = new PipedInputStream(output);

output.write("Hello Java".getBytes());

int i = 0;
do {
        i = input.read();
        if (i != -1)
                System.out.print((char)i);
}while(i != -1);
```

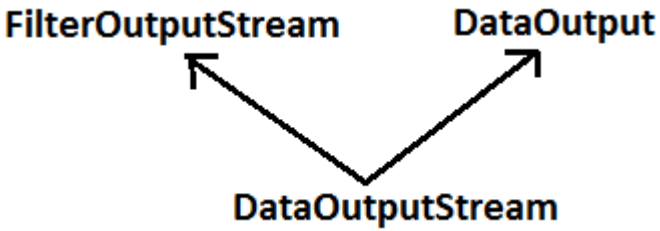
Sequence streams

- Sequence stream allows us to read data from multiple streams. Here is sample code:

```
FileInputStream fin1=new FileInputStream("abc.txt");
FileInputStream fin2=new FileInputStream("pqr.txt");
SequenceInputStream sis=new SequenceInputStream(fin1,fin2);
int i;
while((i=sis.read())!=-1){
   System.out.println((char)i);
}
sis.close();
fin1.close();
fin2.close();
```

Handling primitive data





- If you wish to read/write primitive data into a file then Java IO provides us two classes DataInputStream & DataOutputStream.
- DataInputStream & DataOutputStream extend filter stream classes & implement DataInput & DataOutput interfaces.
- These interfaces have required methods for handling primitive data types. For example readInt(), readDouble(), writeInt(), writeDouble() etc.

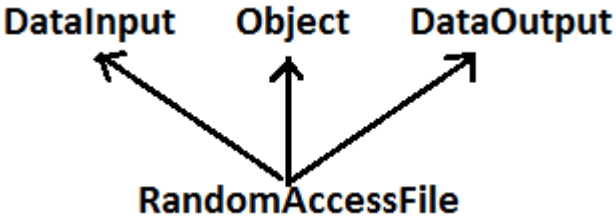
Writing primitive data

```
File file = new File("abc.txt");
FileOutputStream fos = new FileOutputStream(file);
DataOutputStream dos = new DataOutputStream(fos);
dos.writeInt(23);
dos.writeUTF("Tom");
dos.writeDouble(12000.85);
dos.flush();
fos.flush();
dos.close();
fos.close();
```

Reading primitive data

```
File file = new File("abc.txt");
FileInputStream fis = new FileInputStream(file);
DataInputStream dis = new DataInputStream(fis);
int id = dis.readInt();
String name = dis.readUTF();
double salary = dis.readDouble();
dis.close();
fis.close();
```

RandomAccessFile

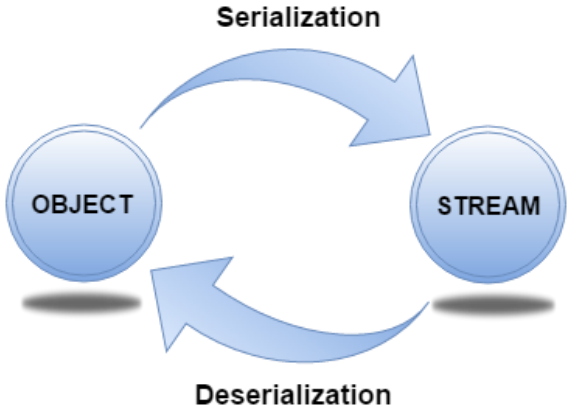


- In case of frequent read/write file operations, java developer needs a single stream class that can be used for both read/write purposes.
- Java IO APIs introduces an advanced class named 'RandomAccessFile' that is useful for both read & write operations.
- RandomAccessFile is also useful in case of handling primitive data.
- RandomAccessFile provides us file pointer that can be used for free navigation within a file.
- Note that RandomAccessFile does not extend InputStream or OutputStream classes.

RandomAccessFile continue…

```
File file = new File("abc.txt");
RandomAccessFile raf = new RandomAccessFile(file, "rw"); //rw means read & write both modes
raf.writeInt(23);
raf.writeUTF("Tom");
raf.writeDouble(12000.85);
long currentPosition = raf.getFilePointer(); //gives current file pointer position
raf.seek(0); //moves file pointer to required location within the file
int id = raf.readInt();
String name = raf.readUTF();
double salary = raf.readDouble();
raf.close();
```

Object Serialization



- Java supports object persistency in the form of ‘Object Serialization’. Serialization is a process in which current state of Object will be saved in stream of bytes.
- Java object is persistence into file system or to another java process is called as ‘Object Serialization’.
- Java object reading from file system or from another java process is called as ‘Object Deserialization’.
- In order to serialize any object, the object must implement Serializable interface.

Object Serialization code

```
File file = new File("abc.txt");
FileOutputStream fout = new FileOutputStream(file);
ObjectOutputStream out = new ObjectOutputStream(fout);
out.writeObject(new Order(1, "Chair purchase", 25000));
out.flush();
out.close();
```

Object Deserialization code

```
File file = new File("abc.txt");
FileInputStream fin = new FileInputStream(file);
ObjectInputStream in = new ObjectInputStream(fin);
Order order = (Order)in.readObject();
System.out.println("order = " + order);
```

Facts about Serialization

- If you do not wish to serialize a specific attribute of an object then declare it as ‘transient’. For example:

```
class Order {
   String title;
   transient double price;
}
```

- Static & transient part of an object is never serialized.
- If a serialized object maintains has a relationship with another object then contained object must also be serializable.
- If a base class implements Serializable interface then naturally all its sub classes become eligible to get serialized.
- If you wish to customize your serialization & deserialization process then implement Externalizable interface instead of Serializable.

SerialVersionUID

- *The serialVersionUID is a static final long number declared inside a class that gets serialized. For example:*

```
public class Employee implements Serializable
{
  public String firstName;
  private static final long serialVersionUID = 5462223600l;
}
```

- *Defining a serialVersionUID field in serializable class is not mandatory.*
- *If you do not declare serialVersionUID then coompiler will generate it automatically.*
- *If there is a difference between serialVersionUID of loaded reciever class and corresponding sender class then InvalidClassException will be thrown.*
- *You should use different serialVersionUID for different version of same class if you want to forbid serialization of new class with old version of same class.*

Thank You!

US – Corporate Headquarters

1248 Reamwood Avenue,
Sunnyvale, CA 94089
Phone: (408) 743 4400

343 Thornall St 720
Edison, NJ 08837
Phone: (732) 395 6900

UK

20 Broadwick Street
Soho, London
W1F 8HT, UK

89 Worship Street
Shoreditch,
London EC2A 2BF, UK
Phone: (44) 2079 938 955

India

Mumbai
4th Floor, Nomura
Powai , Mumbai 400 076

Pune
5th Floor, Amar Paradigm
Baner, Pune 411 045

Kolkata
2B, 12th Floor, Tower 'C'
Rajarhat, Kolkata 700 156

Bangalore
4th Floor, Kabra Excelsior,
80 Feet Main Road,
Koramangala 1st Block,
Bengaluru (Bangalore) 560034

Gurgaon
A/373rd Floor, Sigma Center
Gurgaon, Haryana 122 011s