Chapter 3

Java programs can communicate with different input/output devices such as standard input/output, files, pipes. The programming of this communication is based on the concept of streams. For example, they exist streams to read/write characters and bytes. The stream may enter the buffering mechanism and filters.

In this chapter we will present the applications working on the streams of data (characters, bytes, objects) and a number of additional features implemented by the filters.

An application (program) in Java

Running a Java application begins with the *main()* method located in the *main* class. This method should contain arguments. The following example uses the method *System.out.printl()* used to display the arguments and internal data. This method is overloaded and carries out an automatic conversion of values to strings.

```
public class ApplicationPrint{
public static void main(String[] args) {
String stringData = "Java Mania";
char[] charArrayData = { 'a', 'b', 'c' };
int integerData = 4;
long longData = Long.MIN_VALUE;
float floatData = Float.MAX_VALUE;
double doubleData = Math.PI;
boolean booleanData = true;
for(int i=0;i<args.length;i++)
  System.out.println(args[i]);
System.out.println(stringData);
System.out.println(charArrayData);
System.out.println(integerData);
System.out.println(longData);
System.out.println(floatData);
System.out.println(doubleData);
System.out.println(booleanData); }
}
```

The streams

The data streams are the devices through which communication takes place with the user (standard input/output) with the file system and the network. In a stream, data is written and read sequentially. The stream can be accessed by units (blocks) of arbitrary size: byte, word, line, etc.. To improve communication performance, a stream may use a buffer. The classes related to streams are grouped in the package *java.io*.

Flow of characters

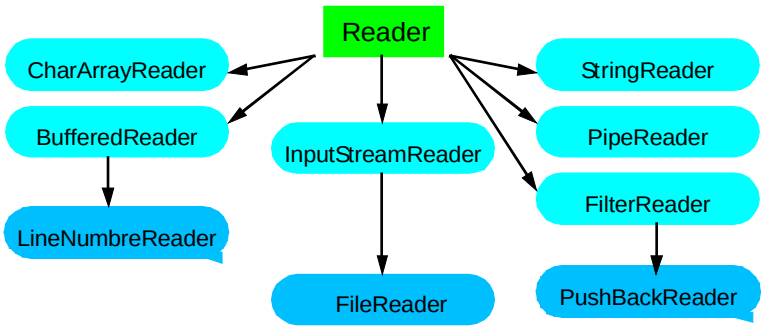
Reader and *Writer* are abstract super-classes defined for reading/writing streams of characters. *Reader* and *Writer* are implemented by subclasses that handle specific flows when reading and writing on different devices: the standard I/O, files, or network. Other classes used to implement specific processing of streams. The abstract class *Reader* contains methods type *read* () for reading characters and arrays of characters:

```
int read()
int read(char cbuf[])
int read(char cbuf[], int offset, int length)
```

The abstract class Writer contains methods *write*() for writing characters and arrays of characters:

```
int write(int c)
int write(char cbuf[])
int write(char cbuf[], int offset, int length)
```

FIGURE 2._ **Hierarchy of *Reader* classes to read the characters**



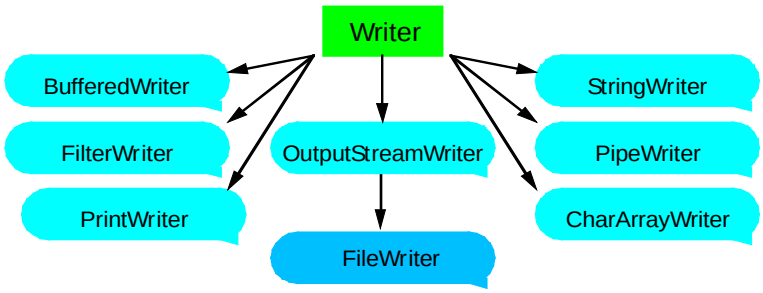
Example of direct reading (un-buffered) of strings from standard input:

```
import java.io.*;
public class AppliReadStream{
  public static void main(String[] args) throws IOException{
  int nClus;
  Reader stdIn =  new InputStreamReader(System.in);
  char[] userInput = new char[256];
  while((nClus = stdIn.read(userInput)) != -1)  System.out.println(userInput);  }
}
```

Example of reading buffer for strings coming from standard input:

```
import java.io.*;
public class AppliReadBuffered{
  public static void main(String[] args) throws IOException{
  BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
  String userInput;
  while((userInput = stdIn.readLine()) != null)    System.out.println(userInput);  }
}
```

FIGURE 3._ Hierarchy of Writer classes



Byte Streams

Java programs can read/write byte strings (bytes). For example, files that contain images and sound files are binary. The InputStream abstract class contains methods *read()* used for reading bytes and arrays of bytes:

```
int read();
int read(char cbuf[]);
int read(char cbuf[], int offset, int length);
```

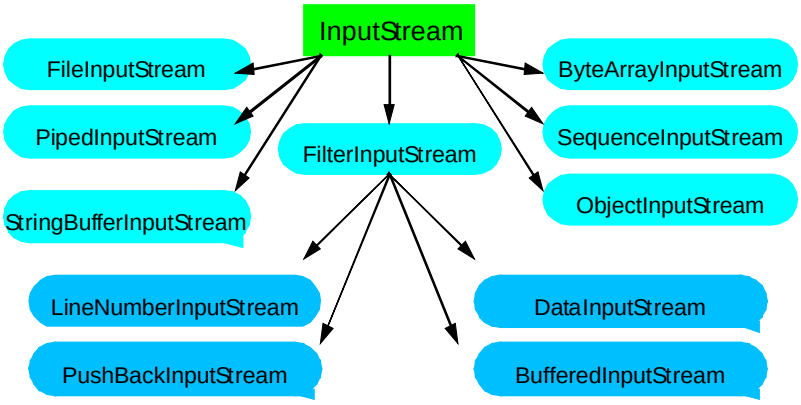
The OutputStream abstract class contains methods *write()* used to write the bytes and arrays of bytes:

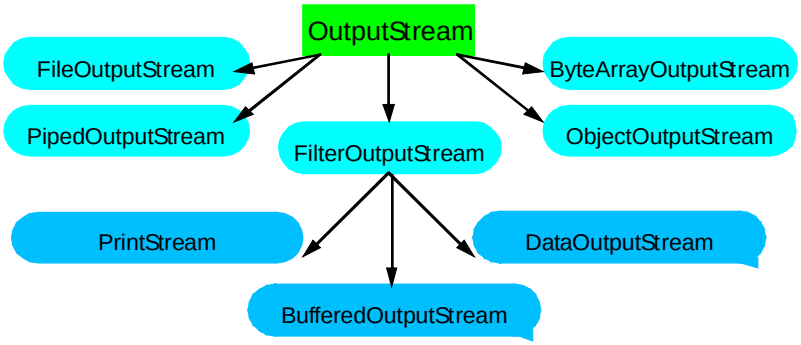
```
int write(int c);
int write(char cbuf[]);
int write(char cbuf[], int offset, int length);
```

The table below includes all methods for reading and writing streams on different storage devices.

device	**Character chains**	**Byte chains**
memory	CharArrayReader, CharArrayWriter StringReader, StringWriter	ByteArrayInputStream, ByteArrayOutputStream StringBufferInputStream
tube (pipe)	PipedReader, PipedWriter	PipedInputStream, PipedOutputStream
file	FileReader, FileWriter	FileInputStream, FileOutputStream

The figures below list the concrete subclasses of InputStream and OutputStream, prepared to read/write and process byte streams.





Examples:

Display of characters stored in the memory:

```
String str = “XYZ”;  // to write on screan
StringReader reader = new StringReader(str);
int ncl;
while(( ncl=reader.read()) != -1)     // reading characters one by one
  {
  System.out.println((char)ncl);        // writing characters one by one
  }
```

The storage in the memory of byte coming from keyboard:

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
int nol;  // number of bytes read
while((nol = System.in.read()) !=-1) { out.write(nol);
 }
byte[] btab = out.toByteArray();
out.reset();   // clear the buffer
```

Files

The stream classes on file are: FileInputStream, FileOutputStream, FileReader, FileWriter and RandomAccessFile.
The FileInputStream and FileOutputStream classes allow to create streams of bytes. The FileReader and FileWriter classes are derived classes form InputStreamReader and OutputStreamWriter.

The RandomAccessFile class allows you to move the file pointer to the position sought. Constructors of streams on files take as parameters the file name or the name of an object of class File.

```
FileReader in = new FileReader(“fichier.txt”);
FileReader = new FileReader(new File(“fichier.txt”));
```

Example: Copying of a text file:

```
import java.io.*;
public class CopyTextFile {
public static void main(String[] args) throws IOException {
File inputFile = new File("fichier_in.txt");
File outputFile = new File("fichier_out.txt");
FileReader in = new FileReader(inputFile);
FileWriter out = new FileWriter(outputFile);
int c;
while ((c = in.read()) != -1) out.write(c);
in.close();
out.close(); }
}
```

Copying of a binairy file:

```
import java.io.*;
public class CopyByteFile {
public static void main(String[] args) throws IOException {
FileInputStream in = new FileInputStream(args[0]);
FileOutputStream out = new FileOutputStream(args[1]);
tampon = new byte[1024];
int bl;
while ((bl = in.read(tampon)) != -1) out.write(tampon,0,bl);    // lecture/ecriture par blocs de 1024 octets
in.close();
out.close(); }
}
```

In cases where the data must be written at the end of an existing file, the File method should be modified as follows:

```
File outputFile = new File("fichier_out.txt", true);
```

Directory files

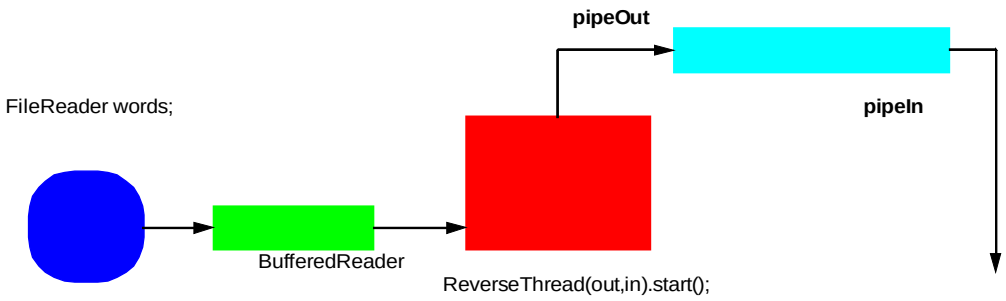
The list of files in a directory can be obtained by the method *list().*

To write an application like: *ls-l* command, simply open a file folder, extract the names of files stored in the directory with the method list () and display the parameters of the files.

```
import java.io.*;
public class ListDirectory{
public static void main(String[] args) throws FileNotFoundException {
File fin = new File(args[0]);
String nom=args[0];
if(!fin.isDirectory()){
  throw new FileNotFoundException();
  }
String chemin;
chemin = nom.endsWith(File.separator)?nom:nom+File.separator;
String[] ls = fin.list();
File fichier;
for(int i=0;i<ls.length;i++) {
  fichier = new File(chemin+ls[i]);
  System.out.println(fichier.isDirectory()?"d\t":"-\t");
  System.out.println(fichier.getName());
  System.out.println(fichier.length());
  System.out.println(fichier.canWrite()?"w\t":"-\t");
  }
}
}
```

Pipes

The tubes are used to channel the output of an input method to another method. Take for example a class “word processor” capable of reversing the order of registration in the strings. The string to be treated may be filed in the tube (pipe) and accessed immediately by another method without having to send data to a file.



Our application uses the methods *PipedWriter* and *PipedReader*

```
FileReader words = new FileReader("words.txt");
Reader ReverseWords = reverse(words);
public static Reader reverse(Reader source) {
   BufferedReader in = new BufferedReader(source);
   PipedWriter pipeOut = new PipedWriter();
   PipedReader pipeIn = new PipedReader(pipeOut);
   PrintWriter out = new PrintWriter(pipeOut);
   new ReverseThread(out, in).start();  // the thread that processes the data in the pipe
   return pipeIn;
}
```

Buffers and filters

A stream is buffered to fill a data buffer. When a program needs this data, it first looks in this buffer before returning to the original source of the stream. This technique is more effective than reading directly from the source. The byte stream to buffer is implemented by classes *BufferedInputStream* and *BufferedOutputStream*. The inflow can operate with two constructors: *BufferedInputStream (InputStream)* and BufferedInputStream (InputStream, int). The second method specifies the buffer size.

The easiest way to read the data from a stream buffer is to call its *read()* method without arguments, the method will normally return a byte - value between 0 and 255. At the end of the flow the method returns the value -1. The output stream can also use two constructors: *BufferedInputStream(InputStream)* and BufferedInputStream(InputStream, int). The second method specifies the buffer size. The method write (int) (limited to an integer value between 0 and 255) sends a byte to the stream output buffer.

Example:

```
import java.io.*;
public class BufCopyFile {
public static void main(String[] args) throws IOException {

FileInputStream fin = new FileInputStream(args[0]);
BufferedInputStream tampon_in = new BufferedInputStream(fin);

FileOutputStream fout = new FileOutputStream(args[1]);
BufferedOutputStream tampon_out = new BufferedOutputStream(fout);

int c;
while ((c = tampon_in.read()) != -1) tampon_out.write(c);

tampon_in.close();
tampon_out.close(); }
}
```

If you need to work with different data such as single bytes: int, long, float, .. we can use filters that read in a byte stream in requested data format. To create an input stream, use the method (filter) constructor *DataInputStream (InputStream*); the argument may be a direct stream or a stream from buffer. Similarly, we can create an output stream using the method *DataOutputStream(OutputStream)* coupled to an output stream.

Reading / writing to a stream of data is performed by the following methods:

```
readInt()/writeInt()
readFloat()/writeFloat()
readLong()/writeLong()
RradBoolean()/writeBoolean()
readDouble()/writeDouble()
readByte()/writeByte()
```

Example: reading a file through a filter of the integers

```
import java.io.*;
public class ReadIntFile {
public static void main(String[] args) throws IOException {


FileInputStream fin = new FileInputStream(args[0]);
BufferedInputStream tampon = new BufferedInputStream(fin);
DataInputStream data = new DataInputStream(tampon);


try {
while(true) {
   int in = data.readInt();
   System.out.print(in + " ");
   }
 }
catch(EOFException eof) { tampon.close(); }
}
}
```

Filters for objects

A stream of object allows to read/write Java objects as a stream. This technique, called serialization can be done by the filters *ObjectInputStream* and *ObjectOutputStream*.

The serialization of an object is done by calling the method *writeObject(Object object)* of a stream that implements the interface *ObjectOutput*. The de-serialization of an object is performed by the method *readObject()on* a stream that implements the interface *ObjectInput*.

The implementation of the interfaces *DataInput* and *DataOutput* is necessary to safeguard the fields of type *primitive*. The first time an object is saved to a stream, all objects that can be reached from that object are also serialized to a stream.

To save an object into a stream, the class of this object must implement the *Serializable* interface. The *Serializable* interface has no methods. When saving a serializable object we save a descriptor of the class, followed by the values of all non-static fields. The descriptor specifies the class name, version and the types and names of its fields. While reading a serialized object, this information allows the recovery of his class. The field values recorded in the stream are used to reconstruct an instance.

In the following example, two objects of class ReadSerPoint are saved in a temporary file and then replayed from that file. For each backup in the flow of object, an object reference is also saved.

```
import java.io.*;
public class ReadSerPoint implements Serializable {
int x;
int y;
public ReadSerPoint(int x, int y) {
this.x=x;
this.y=y;
}
public String toString() {
return "(" + x + "," + y + ")";
}

public static void main(String[] args) throws Exception {
ReadSerPoint p1 = new ReadSerPoint(3,4);
ReadSerPoint p2 = new ReadSerPoint(5,6);
File fichier = File.createTempFile("tempFichier","code");
fichier.deleteOnExit();  // fichier detruit a la fin d'execution
ObjectOutputStream ob_out = new ObjectOutputStream(new FileOutputStream(fichier));
ob_out.writeObject(p1);
ob_out.writeObject(p2);
ob_out.close();
ObjectInputStream ob_in = new ObjectInputStream(new FileInputStream(fichier));
ReadSerPoint p1s = (ReadSerPoint) ob_in.readObject();
ReadSerPoint p2s = (ReadSerPoint) ob_in.readObject();
ob_in.close();
System.out.println("p1 = " + p1);
System.out.println("p2 = " + p2);
System.out.println("p1s = " + p1s);
System.out.println("p2s = " + p2s);
System.out.println("p1 egale a p1s ? " + p1.equals(p1s));
System.out.println("p2 egale a p2s ? " + p2.equals(p2s));
}
}
```

Streams and Applets

The operation of applets will be presented in detail in a chapter dedicated to applets. In this chapter we present only an example applet that communicates with the files that are stored on the same server as the html file incorporating the applet. This constraint protects the system against local access to system resources through a web page.

The following example shows an applet file reader located in the same directory as the *html* page.

```
import java.awt.*;
import java.io.*;
import java.applet.*;
import java.awt.event.*;
public class ReadFileApplet extends Applet implements ActionListener {
TextArea text_out;                    // preparing the display area
Button but1;                          // declaration of a bouton
File textFile;                        // declaration of the file
FileReader in;                        // declaration of a reader
public void init() {                  // applet initialization
text_out = new TextArea(12,48);    // creation of display area
add(text_out);                        // vizualization of display area
but1 = new Button("Read File");    // creation of button
add(but1);                            // vizualization of button
but1.addActionListener(this);       // attachement of button to action
}
public void actionPerformed(ActionEvent e) {      //  ActionListener
lireDonnees();
}

void lireDonnees() {
char[] data = null;  int sizeoffile = 0;
try
 {
 try {
   textFile = new File("toto.res");
   in = new FileReader(textFile);
   int noc= 0;
   sizeoffile = (int)textFile.length();
   data = new char[sizeoffile];
   noc += in.read(data,noc,sizeoffile);       }
   catch(IndexOutOfBoundsException e) {}
 for(int i=0; i<sizeoffile;i++) text_out.append("" + data[i]);     // text display
 in.close(); }
catch(IOException e){ System.out.println("no file"); }}
}
```

Summary

In this chapter we studied the classes to read and write data at the input and output streams. In principle, Java allows to process separately the flow of characters and byte streams. The use of buffers allows for a more flexible and more efficient communication with the files. The stream may contain different types of data. The existence of specialized classes can provide the transfer of different "types" of data (int, float, etc..).

Finally, the serialization mechanism offers the possibility of transferring and storing objects. The developer can then manage the transfer and storage of complex data without knowing the details concerning the registration of such objects.