# Chapitre 3

Les programmes en Java peuvent communiquer avec différents dispositifs entrée/sortie tels que entrée/sortie standard, fichiers, tubes. La programmation de cette communication est basée sur la notion de flux (*streams*). Par exemple, ils existent des flux pour lire/écrire les caractères et les octets. Les flux peuvent intégrer le mécanisme de *bufferisation* et les filtres.

Dans ce chapitre nous allons présenter les applications travaillant sur les flux de données (caractères, octets, objets) et un certain nombre de fonctions supplémentaires implémentées par les filtres.

### Une application (programme) Java

L'exécution d'une application Java débute par la méthode *main()* située dans la classe principale. Cette méthode doit contenir des arguments.

L'exemple suivant utilise la méthode *System.out.println()* pour afficher les arguments et les données internes. Cette méthode est surchargée et effectué une conversion automatique des valeurs en chaînes de caractères.

```
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); }
}
```

## *Les flux*

Les flux de données constituent l'interface à travers laquelle s'effectue la communication avec l'utilisateur (entrée/sortie standard), avec le système de fichiers et avec le réseau. Dans un flux, les données sont écrites et lues de façon séquentielle. Le flux peut être accédé par des unités (blocs) de taille arbitraire: un octet, un mot, une ligne, etc. Pour améliorer les performances de communication, un

flux peut utiliser un tampon; on parle de flux à tampon Les classes consacrées aux flux sont regroupées dans le package *java.io*.

## Flux de caractères

*Reader* et *Writer* sont des super-classes abstraites définies pour la lecture/écriture des flux de caractères. *Reader* et *Writer* sont implémentés par les sous-classes qui traitent les flux spécifiques en lecture et en écriture sur différents dispositifs: les entrées/sorties standard, les fichiers, le réseau. D'autres classes permettent d'implémenter les traitements spécifiques sur les flux.

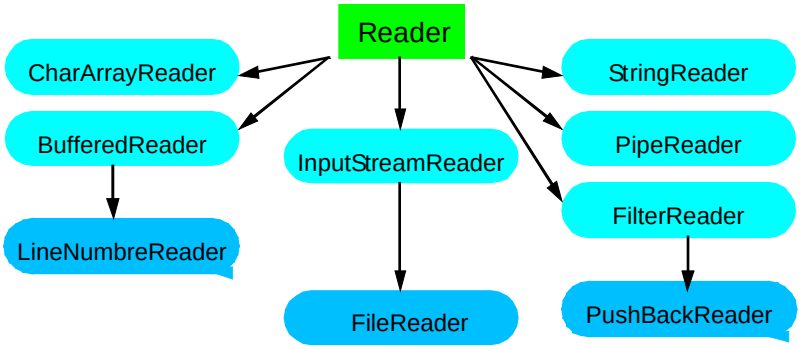
La classe abstraite *Reader* contient les méthodes *read()* permettant la lecture des caractères et *arrays* de caractères:

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

La classe abstraite *Writer* contient les méthodes *write()* permettant l'écriture des caractères et *arrays* de caractères:

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

FIGURE 2._ **Hiérarchie des classes *Reader* pour lecture de caractères**



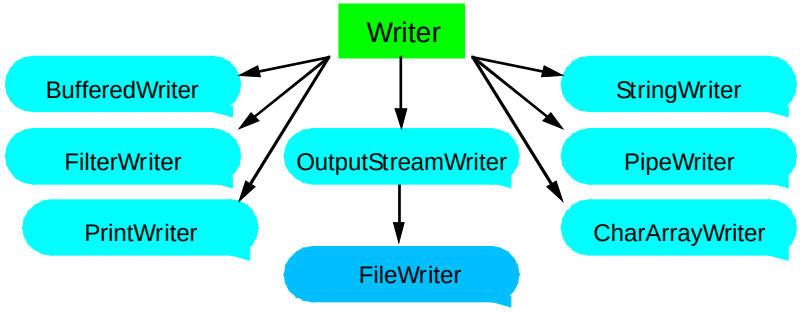

Exemple de lecture directe (sans tampon) des chaînes de caractères sur l'entrée standard:

```
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);
  }
}
```

Exemple de lecture à tampon des chaînes de caractères sur l'entrée standard:

```
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._ Hiérarchie des classes Writer




## Flux d'octets - *Byte Streams*

Les programmes Java peuvent lire/écrire les chaînes d'octets (*bytes*). Par exemple, les fichiers qui contiennent les images et le son sont des fichiers binaires. La classe abstraite *InputStream* contient les méthodes *read()* permettant la lecture des octets et *arrays* d'octets:

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

La classe abstraite *OutputStream* contient les méthodes *write()* permettant l'écriture des octets et *arrays* d'octets:

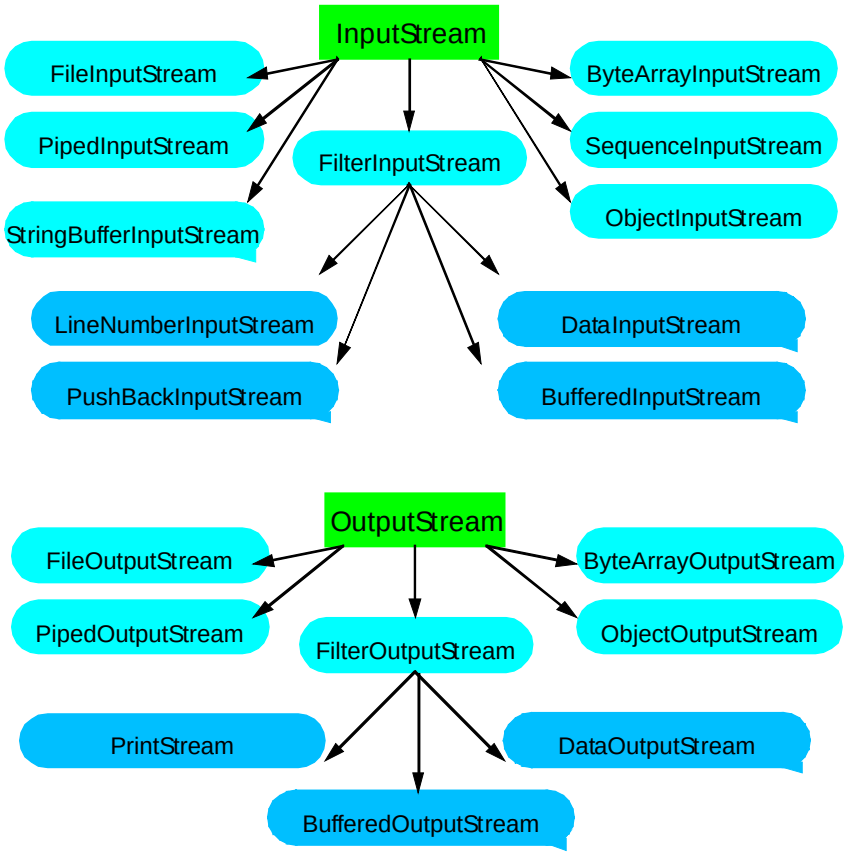
```
int write(int c);
int write(char cbuf[]);
int write(char cbuf[], int offset, int length);
```

La table ci-dessous regroupe l'ensemble des méthodes à utiliser pour la lecture et l'écriture des flux sur différents dispositifs de stockage.

| **dispositif** | **chaînes de caractères** | **chaînes d'octets** |
|---|---|---|
| mémoire | CharArrayReader, CharArrayWriter<br>StringReader, StringWriter | ByteArrayInputStream,<br>ByteArrayOutputStream<br>StringBufferInputStream |
| tube (pipe) | PipedReader,<br>PipedWriter | PipedInputStream, PipedOutputStream |

| | | |
|---|---|---|
| fichier | FileReader,<br>FileWriter | FileInputStream,<br>FileOutputStream |

Les figures ci-dessous répertorient les sous-classes concrètes de *InputStream* et *OutputStream*, permettant de lire/écrire et traiter les “byte streams”.




Exemples:

Affichage des caractères stockés dans la mémoire:

```
String str = “XYZ”;  // une chaine à écrire sur l’écran
StringReader reader = new StringReader(str);
int ncl;
L’enregistrement dans la mémoire des octets saisis sur clavier:while(( ncl=reader.read()) != -1)     // lecture caractère par caractère
  {
  System.out.println((char)ncl);        // écriture caractère par caractère
  }
```

L’enregistrement dans la mémoire des octets saisis sur clavier:

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
int nol;  // nombre d’octets lus
while((nol = System.in.read()) !=-1) { out.write(nol);  // la taille du tampon out augmente automatiquement
 }
byte[] btab = out.toByteArray();
out.reset();   // effacement du tampon
```

## Fichiers

Les classes de flux sur des fichiers sont: *FileInputStream*, *FileOutputStream*, *FileReader*, *FileWriter* et *RandomAccessFile*.

Les classes *FileInputStream* et *FileOutputStream* permettent de créer des flux d'octets. Les classes *FileReader* et *FileWriter* sont dérivées des classes *InputStreamReader* et *OutputStreamWriter*. La classe *RandomAccessFile* permet de déplacer le pointeur du fichier sur la position demandée.

Les constructeurs de flux sur des fichiers prennent en paramètre le nom du fichier ou le nom d'un objet de la classe *File*.

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

Exemple: Copie d'un fichier texte:

```
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(); }
}
```

Copie d'un fichier binaire:

```
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(); }
}
```

Dans le cas où les données doivent être écrites à la fin du fichier existant, la méthode *File* doit être modifiée comme suit:

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

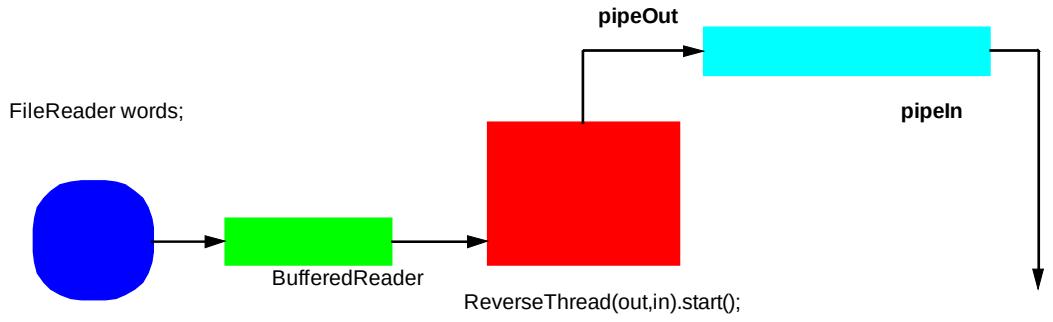
## Fichiers répertoires

La liste des fichiers d'un répertoire peut être obtenue par la méthode *list()*. Pour écrire une application type commande *ls -l* il suffit d'ouvrir un fichier répertoire, extraire les noms des fichiers enregistrés dans le répertoire par la méthode *list()*, et afficher les paramètres des fichiers.

```
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");
  }
}
}
```

## Tubes (*pipes*)

Les tubes sont utilisés pour canaliser la sortie d'une méthode vers l'entrée d'une autre méthode. Prenons comme exemple une classe de traitement de texte capable de renverser l'ordre d'enregistrement des chaînes de caractères. La chaîne à traiter peut être déposée dans le tube (*pipe*) et accédée immédiatement par une autre méthode sans qu'il soit nécessaire d'envoyer les données dans un fichier.



Notre application utilise les méthodes *PipedWriter* et *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();  // le thread qui traite des données et les envoie dans le pipe
   return pipeIn;
}
```

## *Tampons et filtres*

Un flux à tampon remplit un tampon de données. Quand un programme a besoin de ces données, il commence par regarder dans ce tampon avant de revenir à la source originale du flux. Cette technique est plus performante qu'une lecture directe à la source.

Le flux d'octets à tampon est implémenté par les classes *BufferedInputStream* et *BufferedOutputStream*.

Le flux d'entrée peut exploiter deux constructeurs: *BufferedInputStream(InputStream)* et *BufferedInputStream(InputStream, int)*. La deuxième méthode spécifie la taille du tampon.

La méthode la plus simple pour lire des données à partir d'un flux à tampon consiste à appeler sa méthode *read()* sans arguments; elle doit normalement retourner un octet - une valeur entre 0 et 255. A la fin du flux la méthode renvoie la valeur -1.

Le flux de sortie peut exploiter également deux constructeurs: *BufferedOutputStream(InputStream)* et *BufferedOutputStream(InputStream, int)*. La deuxième méthode spécifie la taille du tampon.

La méthode *write(int)* (un entier limité à une valeur entre 0 et 255) permet d'envoyer un octet dans le flux à tampon de sortie.

Exemple:

```
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(); }
}
```

Si on a besoin de travailler avec les données différentes de simples octets telles que: *int*, *long*, *float*, .., on peut utiliser les filtres qui lisent dans un flux d'octets les données au format demandé.

Pour crée un flux d'entrée, il faut utiliser la méthode (filtre) constructeur *DataInputStream(InputStream)*; l'argument à donner peut être un flux direct ou un flux à tampon. De la même façon, on peut créer un flux de sortie en utilisant le méthode *DataOutputStream(OutputStream)* associée à un flux de sortie.

La lecture/écriture dans un flux de données est réalisée par les méthodes suivantes:

readInt()/writeInt()

readFloat()/writeFloat()

readLong()/writeLong()

RradBoolean()/writeBoolean()

readDouble()/writeDouble()

readByte()/writeByte()

Exemple: lecture d'un fichier par un filtre des entiers

```
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(); }
}
}
```

## Filtres d'objets

Un flux d'objets permet de lire/écrire des objets Java dans un flux. Cette technique, appelée sérialisation, peut être réalisée par les filtres *ObjectInputStream* et *ObjectOutputStream*.

La sérialisation d'un objet se fait par l'appel de la méthode *writeObject(Object objet)* d'un flux qui implémente l'interface *ObjectOutput*. La dé-sérialisation d'un objet est effectuée par la méthode *readObject()* d'un flux qui implémente l'interface *ObjectInput*.

L'implémentation des interfaces *DataOutput* et *DataInput* est nécessaire pour la sauvegarde des champs de type primitif. La première fois qu'un objet est sauvegardé dans un flux, tous les objets qui peuvent être atteints depuis cet objet le sont également.

Pour sauvegarder un objet dans un flux, la classe de cet objet doit implémenter l'interface *Serializable*. L'interface *Serializable* ne possède aucune méthode. Lors de la sauvegarde d'un objet sérialisable on sauvegarde un descripteur de sa classe, suivi des valeurs de tous ses champs non statiques. Le descripteur de la classe spécifie son nom, sa version et les types et noms de ses champs. Pendant la lecture d'un objet sérialisé, ces informations permettent la récupération de sa classe. Les valeurs des champs enregistrées dans le flux permettent d'en reconstruire une instance.

Dans l'exemple suivant, deux objets de la classe *ReadSerPoint* sont enregistrés dans un fichier temporaire puis relus depuis ce fichier. A chaque sauvegarde dans le flux d'objet, une référence de l'objet est également sauvegardée.

```
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));
}
}
```

## Flux et Applets

Le fonctionnement des applets sera présenté en détail dans un chapitre dédié aux applets. Dans ce chapitre nous présentons seulement un exemple d'applet qui communique avec les fichiers qui se trouvent enregistrés sur le même serveur que le fichier *html* intégrant l'applet. Cette contrainte protège le système local contre un accès aux ressources système par le biais d'une page WEB.

L'exemple suivant illustre un applet lecteur de fichier localisé dans le même répertoire que la page *html*.

```
import java.awt.*;
import java.io.*;
import java.applet.*;
import java.awt.event.*;
public class ReadFileApplet extends Applet implements ActionListener {
TextArea text_out;                  // preparation d'une zone d'affichage
Button but1;                        // declaration d'un bouton
File textFile;                      // declaration d'un fichier
FileReader in;                      // declaration d'un lecteur sur fichier

public void init() {                // intialisation de l'applet
text_out = new TextArea(12,48);     // creation d'une zone d'affichage
add(text_out);                      // visualisation de la zone d'affichage
but1 = new Button("Read File");     // creation d'un bouton
add(but1);                          // visualisation d'un bouton
but1.addActionListener(this);       // attachement du bouton sur activateur des actions
}

public void actionPerformed(ActionEvent e) {      // methode de l'ActionListener
lireDonnees();
}
```

```
void lireDonnees() {
char[] data = null;
int sizeoffile = 0;
try
 {
 try
  {
  textFile = new File("toto.res");                    // ouverture d'un fichier
  in = new FileReader(textFile);                      // creation du lecteur sur le fichier ouvert
  int noc= 0;
  sizeoffile = (int)textFile.length();                // lecture de la taille du fichier
  data = new char[sizeoffile];                        // reservation de la zone de lecture
  noc += in.read(data,noc,sizeoffile);                // lecture dans la zone
  }
  catch(IndexOutOfBoundsException e) {}
 for(int i=0; i<sizeoffile;i++) text_out.append("" + data[i]);     // visualisation du texte
 in.close();
 }
 catch(IOException e){ System.out.println("no file"); }
}
}
```

## *Résumé*

Dans ce chapitre nous avons étudié les classes permettant de lire et d'écrire des données dans les flux d'entrée et de sortie. En principe, Java permet de traiter séparément les flux de caractères et les flux d'octets. L'utilisation de tampons permet une gestion plus souple et plus performante de la communication avec les fichiers. Les flux peuvent porter différents types de données. L'existence des classes spécialisées permet d'effectuer les transferts "types" de données (*int*, *float*, etc.).

Finalement, le mécanisme de sérialisation offre la possibilité de transfert et de mémorisation des objets. Le développeur peut donc gérer les transferts et la mémorisation des données complexes sans connaître les détails d'enregistrement de ces objets.