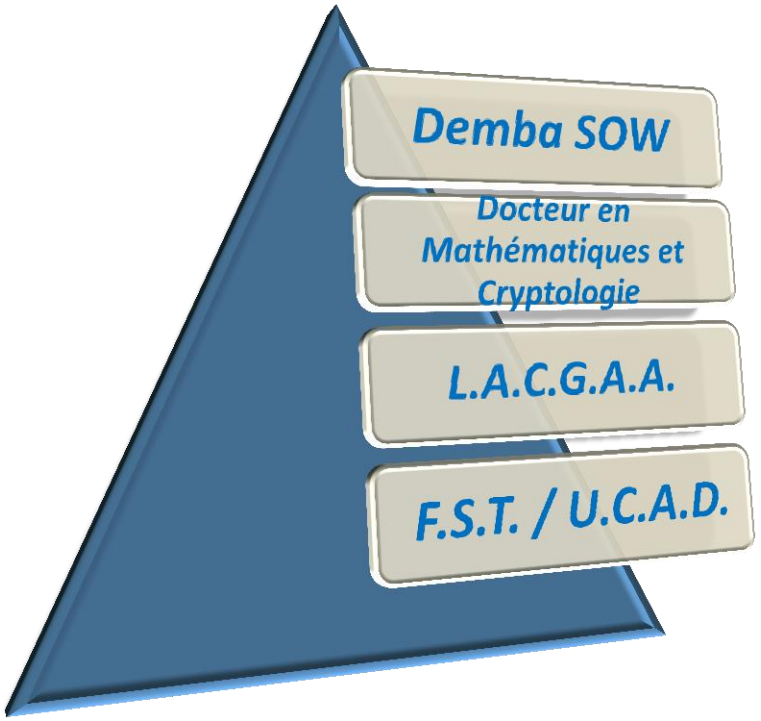
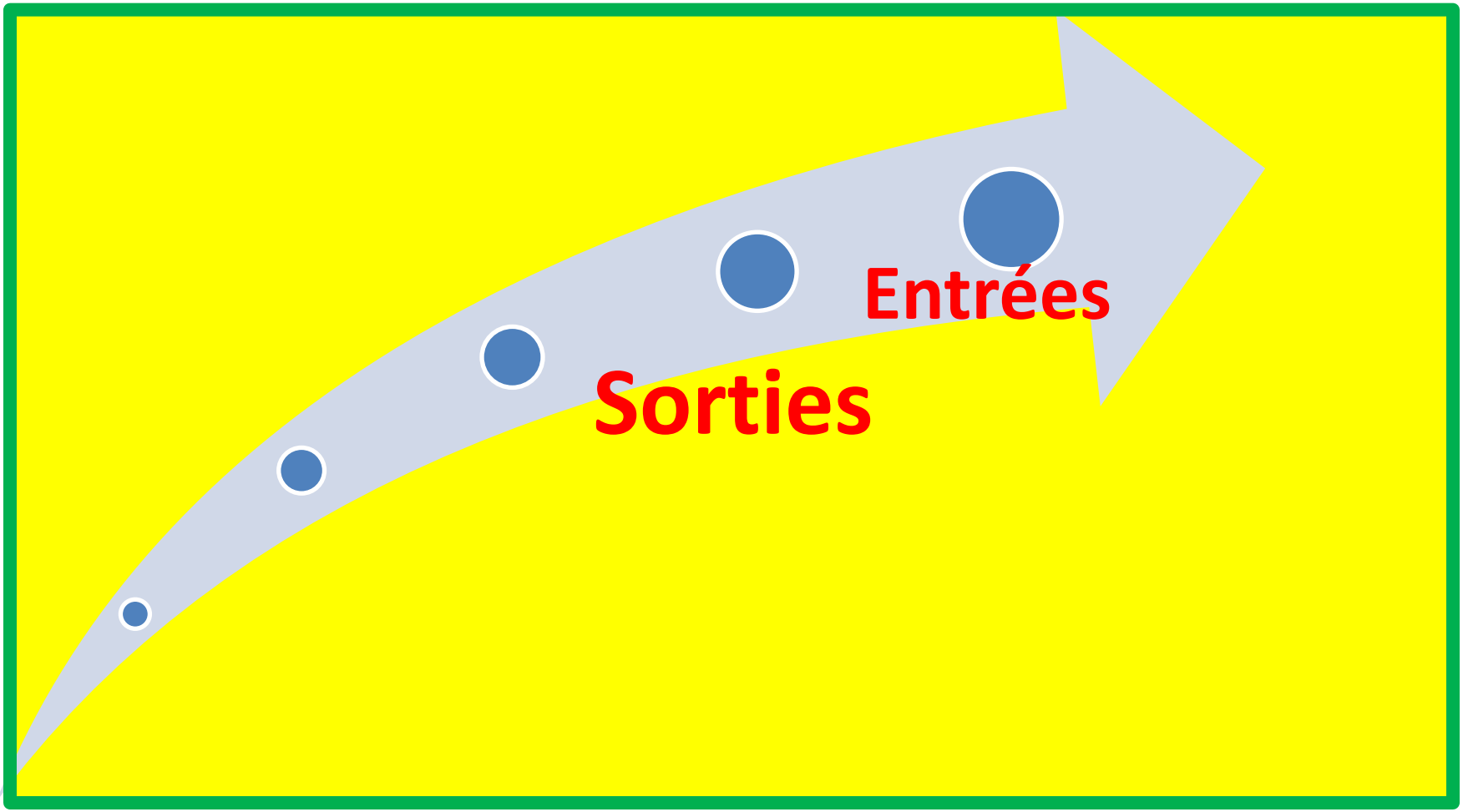
Cours de Programmation Orientée Objet JAVA




Entrées
Sorties

Entrées-Sorties: les flux

Un programme a souvent besoin d'échanger des informations pour recevoir d'une source ou pour envoyer des données vers un destinataire.

La source et la destination peuvent être de natures multiples: fichier, socket réseau, autre programme, etc.…

La nature des données échangées peut également être diverse: texte, images, son, etc.

***Entrée**: le programme lit (reçoit) des données de l'intérieur,*
***Sortie**: le programme écrit (envoie) des données vers l'extérieur.*

Vue générale sur les flux

Un flux (**stream**) permet d'encapsuler des processus pour l'envoi et la réception de données.
Un flux traite toujours les données de *façon séquentielle.*

Par rapport à *la **direction du flux**,* on peut ranger les flux en deux familles:

- les flux d'entrée (**input stream**)
- les flux de sortie (**output stream**)

par rapport à *la **nature des données*** manipulées, on peut diviser les flux en deux grandes catégories:

- les flux de caractères
- les flux d'octets (ou flux binaires)

➡Java définit des flux pour *lire* ou *écrire* des données mais aussi des classes pour traiter les données du flux. Ces classes doivent être associées à un flux de lecture ou d'écriture et sont considérées comme des **filtres**.

Vue générale sur les flux

Toutes les classes de manipulation de flux sont dans le paquetage java.io .

Les classes de flux

L'utilisation de ces classes n'est pas chose facile, vu leur nombre élevé et la difficulté de choisir la classe convenable à un besoin précis.

Pour bien choisir une classe adaptée à un traitement donné, il faut comprendre la dénomination des différentes classes.

Le nom d'une classe est toujours composé d'un **préfixe** et d'un **suffixe**.

Il existe **quatre suffixes possibles** selon le type du flux (flux de caractères ou flux d'octets) et le sens du flux (flux d'entrée ou flux de sortie).

Les classes de flux: suffixes

flux	octet	caractères
entrée	**In**putStream	Reader
sortie	**Out**putStream	Writer

Les classes de flux

Il existe donc quatre hiérarchies de classes qui encapsulent des flux particuliers. Ces classes peuvent être séparées en deux séries de deux catégories différentes:
les classes **de lecture et d'écriture** et les classes de manipulation **de caractères et d'octets.**

- les sous classes de **Reader** sont des types de flux en **lecture** sur les **caractères**,
- les sous classes de **Writer** sont des types de flux en **écriture** sur les **caractères**,
- les sous classes de **InputStream** sont des types de flux en **lecture** sur les **octets**,
- les sous classes de **OutputStream** sont des types de flux en **écriture** sur les **octets**.

Les flux de caractères

Ces flux transportent des données sous forme de caractères et utilisent le format *Unicode* qui code les informations sur 2 *octets.*

Les classes qui gèrent les flux de caractères héritent d'une des deux classes abstraites **Reader** et **Writer**.
Il existe beaucoup de classes dérivées de celles-ci qui permettent de traiter les flux de *caractères*.

Un fichier texte sera considéré comme une séquence de caractères organisés en ***lignes***.
Une fin de ligne est codée par:

- **"\n" sous Unix et Linux**
- **"\r\n" sous MSDOS et Windows**
- **System.getProperty ("line.separator") en Java.**

Les flux de caractères

Il y a plusieurs classes de lecture /écriture.

BufferedReader / BufferedWriter
CharArrayReader / CharArrayWriter
FileReader / FileWriter
InputStreamReader / OutputStreamWriter
LineNumberReader
PipedReader / PipedWriter
PushbackReader
StringReader / StringWriter

Les flux de caractères

Les flux dont le préfixe est:

File gère et manipule les fichiers,
CharArray gère et manipule le tableau de caractères en mémoire
String gère et manipule les chaînes de caractères
Pipe gère et manipule un pipeline entre deux threads
LineNumber: filtre pour numéroter les lignes contenues dans le flux
InputStream/OutputStream: filtre pour convertir des octets en caractères

La classe Reader

C'est une classe abstraite qui est la classe de base de toutes les classes qui gèrent des flux texte d'entrée.

quelques méthodes:

void close ( ) // ferme le flux et libère les ressources qui lui étaient associées
int read ( ) // renvoie le caractère lu et -1 si la fin du flux est atteinte
boolean ready( ) // indique si le flux est prêt à être lu
boolean markSupported ( ) // indique si le flux supporte la possibilité de
// marquer des positions
void mark (int) // permet de marquer une position dans le flux
void reset ( ) // retourne dans le flux à la dernière position marquée

La classe java.io.FileReader

Cette classe permet de gérer les fichiers texte d'entrée. Elle dispose des mêmes fonctionnalités que la classe abstraite Reader.

Mais les méthodes de cette classe sont rudimentaires, c'est pourquoi, il faut souvent la coupler à un objet de la classe **BufferedReader** qui possède des méthodes plus appropriées et complètes à la manipulation de fichiers textes en lecture.

La classe **BufferedReader** est doté d'un tampon et d'une méthode **String readLine ( )** pour la lecture d'une ligne.
Parmi les constructeurs de **FileReader**, on peut distinguer:
FileReader (File objectFile)
FileReader (String fileName)

Lecture d'un fichier texte

Voyons comment lire les informations contenues dans un fichier texte (d'extension .txt) situé sur disque.

Le fichier est situé dans le répertoire: **c:\demba\ essai.txt**

Attention

On commence par lier à un flux texte d'entrée:

```
FileReader   fichier = new FileReader (" c:/demba / essai.txt");
```

Pour utiliser des méthodes plus évoluées, on enroule ce flux dans un filtre:

```
BufferedReader buffer = new BufferedReader(fichier);
```

On parcourt le filtre pour afficher le contenu du fichier:

```
while (ligne_lue != null )
  {System.out .println (ligne_lue) ;
    ligne_lue = buffer.readLine ( ) ;
}
buffer.close ( ) ;
fichier.close( ) ;
```

Lecture d'un fichier texte

```
import java.io .*; // pour utiliser FileReader et BufferedReader
public class TestFileReader {
public static void main(String[] args) throws IOException{
 FileReader fichier = new FileReader ("c:/demba/essai.txt");
 BufferedReader buffer = new BufferedReader (fichier);
String ligne_lue = buffer.readLine( ) ;
 while (ligne_lue != null )
    { System.out .println (ligne_lue) ;
      ligne_lue = buffer.readLine ( ) ;
    }
   buffer.close( ) ; // on ferme le filtre
   fichier.close( ) ; // on ferme le flux
 }
}
```

Pour les éventuelles exceptions sur les classes d'I/O.

Remarques

Les deux instructions :

```
FileReader fichier = new FileReader ("c:/demba/essai.txt");
BufferedReader buffer = new BufferedReader (fichier);
```

peuvent être remplacées par l'unique instruction:

```
BufferedReader buffer = new BufferedReader (new FileReader ("c:/demba/essai.txt"));
```

Et dans ce cas on ne ferme que le filtre puisque contenant le flux, le faite de le fermer, ferme en même temps le flux associé.

On verra une autre façon de créer le flux en utilisant la classe **File**.

La classe java.io.File

La classe **File** permet la gestion des fichiers et des répertoires (création, suppression, renommage, copie, etc.....).

L'instruction **File monfichier = new File ("memoire.txt");**
crée un objet monfichier associé au fichier memoire.txt, mais elle ne crée pas le fichier texte (ie memoire.txt) en tant que tel.
On peut vérifier l'existence d'un fichier avec la méthode boolean exits ( ).
Si le fichier memoire.txt existe dans le système, on peut créer un objet de type File en mentionnant au constructeur un nom de répertoire:
File monfichier = new File ("c:/memoire.txt");
Si le fichier memoire.txt n'existait pas, il serait possible de le créer explicitement sur disque en utilisant la méthode createNewFile ( ) qui renvoie un booléen (true si la création a réussi, false sinon):

```
monfichier.createNewFile ( );
```

Exemple de création d'un Fichier

```
import java.io .*;
public class TestFile {
public static void main(String[] args) {
File file = new File ("c:/memoire.txt");
System.out .println ( file.exists ( ) ); // true car memoire.txt existe sur disque
File f = new File("c:/memoire2.txt");
System.out .println( f.exists ( ) ); // false car memoire2.txt n'existe pas encore
try {
boolean creation = f.createNewFile ( ); // memoire2.txt est créé sur disque
}
catch ( IOException ex)
{ex.printStackTrace() ;}
System.out .println (f.exists ( ) ); //true car memoire2.txt existe maintenant
}}
```

On protège la méthode createNewFile( ) des éventuelles erreurs de type IOException

Constructeur de File

Pour la création d'un fichier, on utilisera souvent l'un des deux constructeurs:

File (File fileName) // on transmet le nom simple du fichier

File (String pathName) // on transmet le répertoire complet du fichier

Si vous créez un fichier en spécifiant un nom de répertoire, il s'agit en fait d'un chemin qui peut être :

- ✓**absolu**: c-à-d spécifié intégralement depuis la racine du système de fichiers;
- ✓**relatif**: c-à-d se référer au répertoire courant.

Si vous créez un programme qui doit être déployé dans un environnement unique, utilisez les conventions en vigueur dans cet environnement:

- ***sous Windows***, les chemins absolus commencent par X (X étant le nom d'un lecteur, ex C:, D:, E:) ou par \
- ***sous Unix***, le chemin absolu commence par /

Constructeur de File

Si le programme doit être déployé dans un environnement multiplateforme, il faut tenir compte de la portabilité et pour cela il faut fournir le séparateur en vigueur dans chaque environnement concerné en utilisant la constante de type **String File.separator .**

NB: il existe aussi deux autres constructeurs de la classe File que vous pouvez souvent utiliser:

File (File repertoire, String simpleNomFichier)

File (String repertoire, String simpleNomFichier)

Exemples

```
import java.io .*;
public class TestFile {
public static void main(String [ ] args) {
File f = new File("\\memoire.txt");      1  // équivaut à "c:\\memoire.txt"
String sep = File.separator ; // pour la portabilité
File ficportable = new File(sep+"demba"+sep+"fic"+sep+"lettre.txt");   2
File fic = new File("\\demba\\fic\\lettre.txt");   3
}
}
```

1 *Chemin relatif par rapport au répertoire courant*

2 *On crée un fichier exploitable dans n'import quel environnement*

3 *Création valable dans l'environnement Windows*

Le fichier memoire.txt est situé dans C:\ et le fichier lettre.txt dans le répertoire absolu C:\demba\fic

Méthodes de la classe File

```
/*crée un nouveau fichier qui n'existait pas, renvoie true si la création réussie*/
                    boolean   createNewFile ( )
/* essaie de supprimer le fichier, renvoie true si la suppression réussie
renvoie false si le fichier n'existe pas*/
                    boolean   delete ( )
/*crée un répertoire ayant le nom spécifié, renvoie true si la création s'est
déroule correctement. Seul le dernier niveau du répertoire peut être créé*/
                    boolean   mkdir ( )
/* idem que mkdir mais avec possibilité de création d'éventuels
niveaux intermédiaires de répertoires*/
                    boolean   mkdirs ( )
/* renvoie true si le fichier correspondant existe*/
                    boolean   exist ( )
/*renvoie true si l'objet correspond à un nom de fichier (même si le fichier
en tant que tel n'existe pas*/
                    boolean   isFile ( )
```

Méthodes de la classe File

```
/*renvoie true si l'objet correspond à un nom de répertoire(même si le répertoire en tant que tel n'existe pas*/
                    boolean   isDirectory ( )
/* donne la longueur du fichier en octets ( 0 si le fichier n'existe pas ou est vide)*/
                    long   length ( )
/*fournit une chaîne contenant le nom du fichier (sans nom de chemin)*/
                    String   getName ( )
/* fournit true si l'objet spécifié correspond à un fichier caché*/
                    boolean   isHidden ( )
/*fournit true si le fichier est autorisé en lecture*/
                    boolean   canRead ( )
/* renvoie true si le fichier est autorisé en écriture*/
                    boolean   canWrite ( )
```

Méthodes de la classe File

Il est possible de connaître tous les répertoires et fichiers d'un répertoire donné en utilisant l'une des deux méthodes ci-dessous:

String [ ] list ( ) // fournit les éléments du répertoire sous forme d'un tableau de chaînes

File[ ] listFiles ( ) // fournit les éléments du répertoire sous forme d'un tableau de fichiers

java.io.StreamTokenizer

La classe **StreamTokenizer** prend un flux d'entrée et l'analyse en jetons (« tokens »), autorisant les jetons à être lus à un moment.
Cette classe réalise un ***analyseur lexical*** qui considère plusieurs types de *tokens:*

- nombre
- mot (délimité par des espaces)
- chaîne de caractères
- commentaires
- EOF (End Of File)
- EOL (End Of Line)

*La méthode **nextToken( )** permet d'appeler l'analyseur.*

java.io.StreamTokenizer: champs et méthodes

champs

```
/*si le token courant est un nombre, ce champ contient sa valeur*/
        double   nval
/*si le jeton courant est un mot, sval contient une chaîne représentant ce mot*/
        String   sval
/*EOF représente une constante indiquant que la fin du flux a été lue*/
        static  int TT_EOF
/*EOL représente une constante indiquant que la fin de la ligne a été lue*/
        static  int TT_EOL
/*ce champ représente une constante indiquant qu'un nombre a été lu*/
        static  int TT_NUMBER
/*ce champ représente une constante indiquant qu'un mot a été lu*/
        static  int TT_WORD
/*après un appel à la méthode nextToken, ce champ contient le type du jeton
venant d'être lu*/
        int  ttype
```

java.io.StreamTokenizer: champs et méthodes

Méthodes

*/*spécifie que l'argument de type char démarre un commentaire de ligne unique*/*
void commentChar (int ch)
*/*détermine si les fins de ligne doivent être traitées comme des jetons*/*
void eollsSignificant (boolean flag)
*/*retourne le numéro de ligne courante*/*
int lineno ( )
/ détermine si le mot doit être automatiquement converti en minuscule*/*
void lowerCaseMode (boolean b)
*/*analyse le prochain jeton à partir du flux d'entrée de l'objet StreamTokenizer*/*
int nextToken ( )
*/*spécifie que l'argument de type char est ordinaire dans l'objet StreamTokenizer*/*
void ordinaryChar (int ch)
*/*spécifie que tous les caractères dans l'intervalle [low, hi]sont ordinaire dans l'objet StreamTokenizer*/*
void ordinaryChars (int low, int hi)
/ retourne un objet String représentant l'objet StreamTokenizer courant*/*
String toString( )

Exercice StreamTokenizer

Ecrire une classe ArrayListStreamTokenizer qui permet de créer un vecteur dynamique (ArrayList) constitué de tous les mots d'un fichier texte (.txt).
Le ArrayList sera formé de tous les mots séparés par des espaces blancs, apparaissant dans le fichier spécifié. On prévoira:

- **une méthode ArrayList arrayListFromStreamTokenizer (File fic)**
  qui prend en argument un fichier et stocke dans un vecteur tous ses mots.
- **une méthode void printArray ( ) qui affiche le vecteur obtenu.**

Voici une classe de test:

```
public class TestStreamTokenizer {
public static void main (String[] args) throws FileNotFoundException, IOException {
    File fichier = new File ("C:\\demba\\memoire.txt");
    ArrayListStreamTokenizer essai = new ArrayListStreamTokenizer ( );
    essai.arrayListFromStreamTokenizer (fichier);
    essai.printArray ( ) ;
  }
}
```

Exercice StreamTokenizer : Corrigé (1/2)

```
public class AnalyseurStreamTokenizer {
ArrayList liste;
ArrayList arrayListFromStreamTokenizer (File fic) throws FileNotFoundException,
IOException
  { liste = new ArrayList ( );
    BufferedReader br = new BufferedReader( new FileReader (fic));
    StreamTokenizer st = new StreamTokenizer (br);
    int unMotTrouve = StreamTokenizer.TT_WORD ; // un mot est lu (-3)
    while (unMotTrouve != StreamTokenizer.TT_EOF )
      { unMotTrouve =st.nextToken ( ) ; // jeton courant (si mot alors = -3)
        if (unMotTrouve == StreamTokenizer.TT_WORD )
          { String s = st.sval ;
            liste.add (s) ;
          }
       }
     return liste;
}
```

Exercice StreamTokenizer :Corrigé (2/2)

```
void printArray(){
    System.out .println (liste.toString ( ) ) ;
  }
} // fin de la classe
```

Remarque:

Dans l'entête
de la méthode arrayListFromStreamTokenizer , nous avons capturé par
throws FileNotFoundException, IOException
les éventuelles exceptions qui seraient générées respectivement par l'emploi de FileReader et par l'appel de la méthode nextToken( ).

Filtre sur les noms de fichiers

Un filtre est un objet qui permet de ne prendre en compte qu'un certain nombre de types fichiers.

Pour cela, on utilisera la méthode (de la classe File):
/*prend en argument un filtre et renvoie sous forme de tableau de chaînes tous les fichiers vérifiant les conditions du filtre*/
public String [ ] list (FilenameFilter filtre)

Toute classe qui définit un filtre sur un fichier doit implémenter l'interface FilenameFilter et donc redéfinir la seule méthode de cette dernière:
/*là on définit les conditions du filtre*/
public boolean accept (File rep, String nom).

Exemple de Filtre

```
import java.io .*;
class Filtre implements FilenameFilter {
public boolean accept (File rep, String nom) {
/*on définit les conditions du filtre*/
if (rep.isDirectory ( ) && nom.endsWith(".java")) return true;
else return false;
}}
public class FiltreTest {
public static void main (String args[] ){
new FiltreTest( ).affiche("."); // test: on filtre le répertoire racine ( .)
}
public void affiche(String rep) {
File fichier = new File (rep); // on veut filtrer ce répertoire
String nomFics [ ] = fichier.list ( new Filtre( ) ); // on lui associe donc un filtre
for (int i = 0; i < nomFics.length; i++)
System.out.println (nomFics [i]);}}
```

La classe Writer

C'est une classe abstraite qui est la classe de base de toutes les classes de flux de caractères en écriture.

Elle définit quelques méthodes rudimentaires:

/*ferme le flux et libère les ressources qui lui étaient associées*/

void close ( )

/*écrire le caractère en paramètre dans le flux*/

void write (int)

/* écrire le tableau de caractères dans le flux*/

void write (char [ ])

/*écrit le tableau de caractères tab, i = indice du premier caractère à écrire
j = le nombre de caractères à écrire*/

void write (char [ ], int i, int j)

/*écrire la chaîne de caractères en paramètre dans le flux*/

write (String)

/*écrire une chaîne à partir du caractère i , j étant le nombre de caractères à écrire*/

write (String, int i, int j)

La classe java.io.FileWriter

Cette classe gère les flux de caractères en ***écriture***. Elle possède plusieurs constructeurs qui permettent d'écrire un ou plusieurs caractères dans le flux:

*/*Si le nom spécifié n'existe pas alors le fichier sera créé. S'il existe et qu'il contient des données, celles-ci seront écrasées*/*

FileWriter (String nomfichier)

/ Idem que précédemment mais ici le fichier est précisé avec un objet de la classe File* /*

FileWriter (File nomfichier)

*/*le booléen permet de préciser si les données seront ajoutées au fichier (valeur true) ou écraseront les données existantes (valeur false)*/*

FileWriter (String, boolean)

La classe java.io.FileWriter

Pour améliorer les performances des flux d'écriture sur un fichier texte, la mise en tampon des données écrites permet de traiter un ensemble de caractères plutôt que traiter caractère par caractère. La mise en tampon entraîne un gain de temps.

La classe BufferedWriter permet de tamponner un flux en écriture sur un fichier texte.

```
File fichierOuEcrire = new File ( fileName);
FileWriter fw = new FileWriter (fichierOuEcrire);
BufferedWriter bw = new BufferedWriter (fw);
```

Et pour fermer les flux, on suit l'ordre inverse de leur création:

```
bw.close ( );
fw.close ( );
```

Ces trois instructions peuvent être rassemblées en une seule :

```
BufferedWriter bw =new BufferedWriter (new FileWriter (new File (fileName)));
```

Ici on ne ferme que le flux externe; ça suffit: `bw.close ( );`

Ecrire dans un fichier texte

```
public class TestFileWriter {
   public static void main(String[ ] args)throws IOException {
   File fichier = new File("c:\\demba\\essai.txt");
   FileWriter fw =new FileWriter( fichier.toString ( ) ,true);
   BufferedWriter bw = new BufferedWriter(fw); // flux tamponne
   int i = 0;
   while (i < 5)
    {
       bw.newLine ( ) ; // on crée une nouvelle ligne dans le fichier
       bw.write (" bonjour le monde") ; // et on écrit cette chaine
       i++;
  }
  bw.close ( ) ;
  fw.close ( ) ;
 }
}
```

La classe java.io.LineNumberReader

(cette classe permet de numéroter les lignes d'un flux de caractères)

```
/** Exemple de copie d'un fichier texte dans un autre fichier avec numérotation des
lignes. La première ligne porte le numéro 1. */

public class NumeroLigne {
public static void main (String [] st) throws IOException {
File src = new File ("c:\\ demba\\ fex.txt"); // on copie ce fichier
File dest = new File ("c:\\ demba\\ newT.txt"); // dans celui-ci
                                                    //avec des numéros de lignes
LineNumberReader in = new LineNumberReader (new FileReader (src)) ;
BufferedWriter out = new BufferedWriter (new FileWriter (dest));
            String s = in.readLine ( );
            while (s != null) {
            out.write (in.getLineNumber ( ) + ": " + s + "\r\n");
            s = in.readLine ( );
        }
    in.close ( );
    out.close ( );
}}
```

// pour marquer une fin de //ligne.

La classe java.io.PrintWriter

Cette classe permet d'écrire dans un flux des données *formatées*.
Le formatage autorise la manipulation simultanée *des données de types différents* dans un même flux.
Cette classe dispose de plusieurs constructeurs:

/*le paramètre fourni précise le flux, le tampon est automatiquement vidé*/
PrintWriter (Writer)
/*le booléen précise si le tampon doit être automatiquement vidé*/
PrintWriter (Writer, boolean)
/*le paramètre fourni précise le flux, le tampon est automatiquement vidé*/
PrintWriter (OutputStream)
/*le booléen précise si le tampon doit être automatiquement vidé*/
PrintWriter (OutputStream, boolean)

Cette classe s'applique à la fois pour les flux de sortie de caractères et d'octets .

La classe java.io.PrintWriter

La classe PrintWriter présente plusieurs méthodes print (anyType var) prenant un argument *var* de type *anyType* où *anyType* peut être n'importe quel type primitif (int, long, double, boolean, ….), type String, Object ou tableau de caractères (char [ ]) permettant d'envoyer des données formatées dans le flux.

La méthode println ( ) permet de terminer la ligne dans le flux en y écrivant un saut de ligne.

Il existe aussi plusieurs méthodes println (anyType var) faisant la même chose que la méthode print mais les données sont écrites avec *une fin de ligne*.

La classe PrintWriter présente une méthode spéciale void flush ( ) qui vide le tampon (sans que le flux ne soit fermé) en écrivant les données dans le flux.
Les méthode write (…) de cette classe hérite de Writer.

Exemple PrintWriter

```
public class TestPrintWriter {
String fichierDest;
public TestPrintWriter (String fichierDest) throws IOException {
this.fichierDest = fichierDest;
traitement ( );
}
  private void traitement ( ) throws IOException
    {   PrintWriter pw =new PrintWriter (new FileWriter (fichierDest));
        pw.println ("bonjour monsieur") ;
        pw.write (100) ; //écrire le caractère de code 100 c à d 'd'
        pw.println ( ) ; //mettre un saut de ligne
        pw.println ("votre solde est"+ 10000) ;
        pw.print ("nous sommes le:"+ new java.util.Date ( )) ;
        pw.close ( ) ; //fermer le flux pour que les données soit écrites
  }
public static void main (String[] args) throws IOException {
        TestPrintWriter tp = new TestPrintWriter("c:\\demba\\ecrit.txt");
}}
```

Exercice avec PrintWriter

Utiliser la classe PrintWriter pour réaliser une classe CopyFile qui permet de copier le contenu d'un fichier texte dans un autre fichier texte. On prévoira:
-une méthode void copieFichier (File src, File dest) permettant de copier le contenu du fichier *src* dans le fichier *dest*. La copie sera conforme (ie les données apparaîtront dans les deux fichiers exactement de la même manière)
on suppose que le fichier *dest peut ne pas exister, dans ce cas, il faut prévoir:*
-la méthode void createFile (File f) qui permet de créer le fichier f s'il n'existe pas encore.
Avant de réaliser la copie dans *copieFichier,* il faut s'assurer que le fichier *dest existe ou* pas en appelant *createFile* qui créera éventuellement ce fichier.

```
/*une classe de test pour cet exercice*/
    public class TestCopy {
        public static void main(String[] args) {
         File s = new File ("c:\\demba\\source.txt");
       File d = new File ("c:\\demba\\dest.txt"); // si dest.txt n'existe pas, il est alors créé
        new CopyFile ( ).copieFichier (s,d) ; // et la copie est effectuée ici
}}
```

Exercice avec PrintWriter : Corrigé

```
public class CopyFile {
public File createFile (File f )
{ if ( f.exists ( ) = = false)
{ try { f.createNewFile ( );
}
catch (IOException e)
{ e.printStackTrace ( ) ;}
}
return f;
}
```

```
public void copieFichier (File src, File dest)
{ dest = createFile(dest); // pour être sur que dest existera
try {
FileReader ficsrc = new FileReader (src);
BufferedReader b = new BufferedReader (ficsrc);
PrintWriter pw=new PrintWriter(new FileWriter (dest));
String lg = b.readLine ( ) ;
      while (lg!= null)
         { pw.write (lg) ; // on écrit la ligne courante lue
           pw.println ( ) ; // on met un saut de ligne
           lg = b.readLine ( ) ; // on récupère la ligne suivante
         }
      pw.close ( ) ;
      b.close ( ) ;
  }
catch (FileNotFoundException ee){ee.printStackTrace ( ) ;}
catch (IOException er){er.printStackTrace ( ) ; }
}
```

Les flux d' octets

Ce type de flux permet le transport de données sous forme d'octets.
Les flux de ce type sont capables de transporter toutes les données.

Les classes qui gèrent les flux d'octets héritent d'une des deux classes abstraites InputStream (flux binaire d'entrée) et OutputStream (flux binaire de sortie).
Ces deux classes abstraites disposent de méthodes rudimentaires:

Pour OutputStream

```
void write (int ) //écrit un octet
void write (byte [ ]) // écrit un
                      //tableau d'octet
void close ( ) //ferme le flux
void flush ( ) //vide le tampon
```

Pour InputStream

```
int read ( ) // lit un octet, -1 si fin du flux
int read (byte[ ]) // lit une suite d'octets
void close ( ) // ferme le flux
void skip (long n) // saute n octets dans
                   //le flux
```

Classe java.io.FileInputStream

Cette classe permet de gérer les flux binaires en *lecture* sur un fichier.
Cette classe possède plusieurs constructeurs qui peuvent tous lever une exception de type . **FileNotFoundException**

/*ouvre un flux en lecture sur un fichier de nom la chaîne fic*/

FileInputStream (String fic)

/*ouvre un flux en lecture sur un fichier dont le nom est un objet de type File*/

FileInputStream (File filename)

Cette classe hérite des méthodes de la classe *InputStream* mais elle définit aussi d'autres méthodes qui permettent de lire un ou plusieurs octets dans le flux.

La méthode int available ( ) permet de connaître le nombre d'octets qu'il est encore possible de lire dans le flux.

Exemple de lecture d'un fichier binaire

```
import java.io .*;
public class TestFileInputStream {
public static void main(String[] args) {
File f = new File("c:\\demba\\entree.dat"); // un objet File sur le fichier à lire
try { FileInputStream fis = new FileInputStream (f);// on associe un flux au fichier à lire
byte [ ] tab = new byte [(int) f.length ( ) ];
try { while (fis.read ( ) !=-1) // on s' arrête à la fin du fichier
fis.read (tab,0,(int) f.length ( ) -1 ); // on lit les octets qu'on stocke dans un tableau
fis.close( );
}
catch (IOException er){System.out .print("Erreur de lecture") ;}
for (int i = 0;i < tab.length ;i++)
System.out .println (tab [i]);// on affiche chaque octet lu
}
catch (FileNotFoundException ee) {System.out .println (ee.getMessage ( ) );}
}}
```

Exemple de lecture d'un fichier binaire: Notes

La méthode read ( ) peut lever une exception de type IOException, d'où la nécessité de capturer les éventuelles exceptions qu'elle peut générer par un gestionnaire try …. catch (IOException e).

Beaucoup de méthodes de lecture ou d'écriture dans un flux lèvent une exception de ce type, donc à capturer.

Il en est de même du constructeur FileInputStream qui peut lever une exception de type FileNotFoundException. Ici aussi, il faut la capturer.

Flux binaire en lecture tamponné.

On peut doter les flux binaires d'entrée d'un tampon qui permet d'avoir des opérations plus rapides puisque les données sont lues en rafales.
Pour ce faire, on utilise la classe BufferedInputStream.
Les octets sont lus dans un tableau tampon interne qui est crée simultanément avec la création de l'objet *BufferedInputStream*.
Il existe deux constructeurs de cette classe:

```
BufferedInputStream (InputStream in)
BufferedInputStream (InputStream in, int tailleTampon)
```

L'exemple précédent pouvait être amélioré en dotant du flux créé un tampon comme ceci:

```
File f = new File ("c: \\ demba\\entree.dat");
FileInputStream fis = new FileInputStream (f);
BufferedInputStream bis = new BufferedInputStream (fis);
```

Classe java.io.FileOutputStream

Cette classe permet de gérer les flux binaires en *écriture* sur un fichier.
Cette classe possède plusieurs constructeurs qui peuvent lever tous une exception de type : **FileNotFoundException**

/*ouvre un flux en écriture sur un fichier de nom la chaîne fic si le fichier n'existe pas, il sera créé, s'il existe et qu'il contient des données celles-ci seront écrasées*/

FileOutputStream (String fic)

/*ici le boolean précise si les données seront rajoutées au fichier (valeur true) ou écraseront les données existantes (valeur false)*/

FileOutputStream (String fic, boolean b)

/*ouvre un flux en écriture sur un fichier dont le nom est un objet de type File si le fichier n'existe pas, il sera créé. S'il existe et qu'il contient des données celles-ci seront écrasées*/

FileOutputStream (File filename)

Cette classe hérite des méthodes write (…) de la classe OutputStream.

Exemple de copy de fichier binaire

(on réalise la copie d'un fichier dans un autre fichier avec usage de tampon)

```
public class TestFileOutputStream {
void copyFile (File src, File dest){
try {
BufferedInputStream bis = new BufferedInputStream (new FileInputStream(src));
BufferedOutputStream bos =new BufferedOutputStream(new
FileOutputStream(dest ));
while (bis.available ( ) > 0) // tant qu'il reste des octets à lire
bos.write (bis.read ( ) ) ; // on écrit l'octet lu
bis.close ( ) ; bos.close ( ) ;
}
catch (FileNotFoundException e) { System.out .println ("Erreur sur les flux") ;}
catch (IOException er) {System.out .println("Erreur de read ou write") ;}
}
public static void main(String[] args) {
File dest = new File("c:\\demba\\ecrit.dat");
File src = new File("c:\\demba\\lecture.dat");
new TestFileOutputStream( ).copyFile (src,dest) ;
}}
```

Pour traiter un ensemble d'octets au lieu de traiter octet par octet. Le nombre d'opérations est alors réduit.

Exemple de copy de fichier binaire: Notes

Les classes de flux utilisées ici lèvent toutes une exception de type FileNotFoundException.
Les méthodes read ( ) et write(…) lèvent aussi une exception de type IOException.
Pour gérer ces exceptions, nous avons un bloc try suivi de deux blocs catch pour les deux types d'erreurs évoquées.
Pour gérer ces mêmes exceptions, au lieu de bloc try … catch, on pourrait Mentionner après l'entête
de la méthode:

throws FileNotFoundException, IOException.

Mais, il faut éviter souvent d'utiliser cette dernière possibilité pour des méthodes de l' API que vous redéfinissez. Chacune d'elles gère des exceptions spécifiques.
Par ailleurs, au lieu de deux blocs *catch, on pouvait n'utiliser qu'un* seul bloc catch de cette façon: catch (Exception er).

Classe java.io.DataInputStream
Classe java.io.DataOutputStream

Nous avons vu que la classe FileInputStream disposait de méthodes rudimentaires qui permettent de lire seulement un octet ou un tableau d'octet. Même associé à un tampon les performances se limitent tout simplement à une réduction du nombre d'opérations de lecture.
La classe DataInputStream dispose de méthodes plus évoluées (de lecture de tous les types de données) et l'un de ses constructeurs prend un argument de type FileInputStream.

Il en de même de la classe FileOutputStream qu'on peut lier dans un DataOutputStream pour disposer de méthodes plus évoluées d'écriture de données de n'importe quel type.

Ces deux classes améliorent donc considérablement la manipulation des fichiers binaires.

Exemple de copy de fichier binaire avec Data|In/Out|putStream

```
public class TestCopy {
void copyFile (File src, File dest){ // pour copier src dans dest
try {
DataInputStream bis = new DataInputStream (new FileInputStream(src));
DataOutputStream bos =new DataOutputStream(new FileOutputStream(dest ));
while (bis.available ( ) > 0) // tant qu'il reste des octets à lire
            { bos.writeChars (bis.readLine ( ) ) ; bos.writeChars("\n\r");}
   bis.close ( ) ; bos.close ( ) ;
 }
catch (FileNotFoundException e) { System.out .println ("Erreur sur les flux") ;}
catch (IOException er) {System.out .println("Erreur de read ou write") ;}
}
public static void main(String[] args) {
      File dest = new File("c:\\demba\\ecrit.dat");
      File src = new File("c:\\demba\\lecture.dat");
      new TestCopy( ).copyFile (src,dest) ;
}}
```

Classe java.io.RandomAcessFile

Cette classe gère les *fichiers à accès direct* permettant ainsi un accès rapide à un enregistrement dans un fichier binaire.
Il est possible dans un tel fichier de mettre à jour directement un enregistrement.
La classe RandomAccessFile enveloppe les opérations de *lecture/écriture* dans un fichier.
Elle implémente deux interfaces: DataInput et DataOutput.
Elle possède deux constructeurs:

RandomAccessFile (String nomfichier, String modeacces)

RandomAccessFile (File nomfichier, String modeacces)

Le mode d'accès est une chaîne qui est égale à:

+ "r" : mode lecture

+ "rw" : mode lecture/écriture .

Les constructeurs de cette classe lèvent tous les exceptions:

FileNotFoundException si le fichier n'est pas trouvé,

IllegalArgumentException si le mode n'est pas "r" ou "rw",

SecurityException si le gestionnaire de sécurité empêche l'accès aux fichiers dans le mode précisé.

Exemple de lecture/écriture dans un fichier

```
public class TestReadWrite {
void readWriting (String src, String dest ) throws FileNotFoundException,
IOException{
RandomAccessFile raf = new RandomAccessFile (src,"r");// en lecture seule
RandomAccessFile rw = new RandomAccessFile (dest,"rw"); // en lecture/écriture
while (raf.read ( ) != -1)
{
rw.writeChars (raf.readLine ( ) ) ;
}
raf.close ( ) ;
rw.close ( ) ;
}
public static void main(String [ ] args) throws FileNotFoundException,IOException {
        String fic1="c:\\demba\\lecture.dat"; // on copie ce fichier
        String fic2="c:\\demba\\ecrit.dat"; // dans celui-ci
        new TestReadWrite( ).readingWriting (fic1,fic2);
        }
}
```

Exemple d' accès direct aux enregistrements

```
public class TestWriteInt {
public static void main(String [ ] args) {
String fic = "c:\\demba\\destination.txt";
try { RandomAccessFile rw = new RandomAccessFile (fic,"rw");
/*ecriture dans le fichier dix entiers*/
for ( int i = 0;i < 10;i++)
{ rw.writeInt (i*100 ) ; }
/*lecture du fichier: accès aux enregistrements*/
for (int i = 0; i < 10; i++)
{ long pos = rw.getFilePointer ( ) ; // position courante du pointeur
rw.seek (4*i) ;
System.out .println (rw.readInt ( ) +" "+pos) ;
}
rw.close() ;
}
catch (Exception e){ }
}}
```

0	0
100	4
200	8
300	12
400	16
500	20
600	24
700	28
800	32
900	36

ième enregistrement.

Taille des données (un int est codé sur 4 octets)

Entrées/Sorties standards

Il s'agit de flux relatifs à l'utilisation de la classe System du paquetage java.lang. Cette classe contient plusieurs méthode utilitaires et ne peut être instanciée.

Les entrées-sorties associées à cette classe se traduisent par l'utilisation de variables définissant trois types de flux:

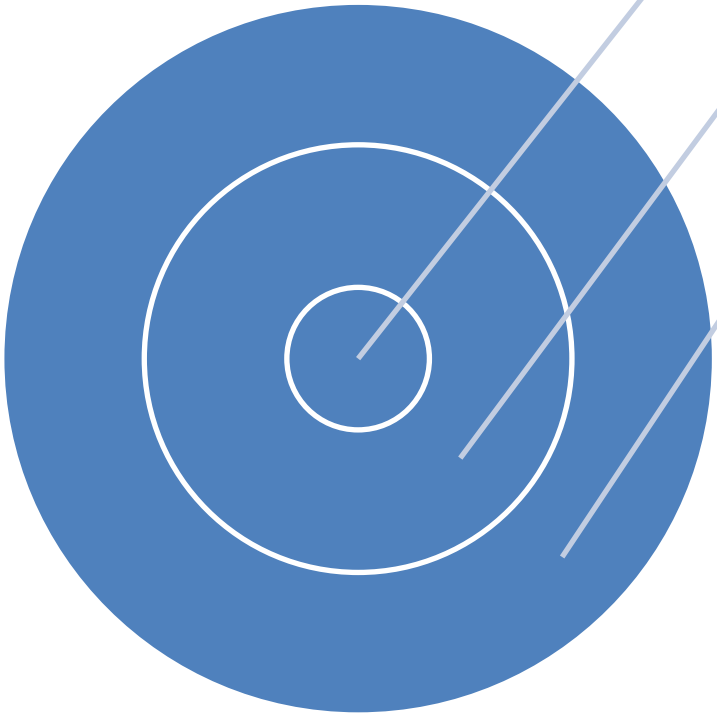
- **static PrintStream err pour le flux *d'erreur* standard,**
- **static InputStream in pour le flux *d'entrée* standard,**
- **static PrintStream out pour le flux *de sortie* standard.**

Le flux de sortie standard défini par la variable out est plus "connu" et est très facile à manier dans des instructions du genre:

```
System.out.println ("usage du flux de sortie pour l'affichage sur la console");
```

Exemple de flux standard

```
public class LectureClavier {
public static void main (String [ ] args) {
  InputStreamReader isr = new InputStreamReader (System.in );
BufferedReader dis = new BufferedReader (isr); // pour des méthodes plus évoluées
System.out .print ("saisir UN ENTIER") ;
String ligne_lue ="";
try{ ligne_lue = dis.readLine ( ) ; // récupérer ce qui est entré au clavier
try { int k = Integer.parseInt (ligne_lue) ; // et le convertir en entier
System. out .println("valeur entière lue ="+k) ;
}
catch (NumberFormatException g ){ // pour les exceptions de parseInt (…)
System. err .print ("Format de nombre incorrect") ;
}
catch (IOException e) { // pour les exceptions de readLine ( )
System. err .print ("Erreur lecture") ;}
}}
```



Fin

du

chapitre