prosper class: page intentionnally left blank to overcome an incompatibility bug between B. Gaulle ’french’ package and the seminar class.

Entrées/sorties en Java

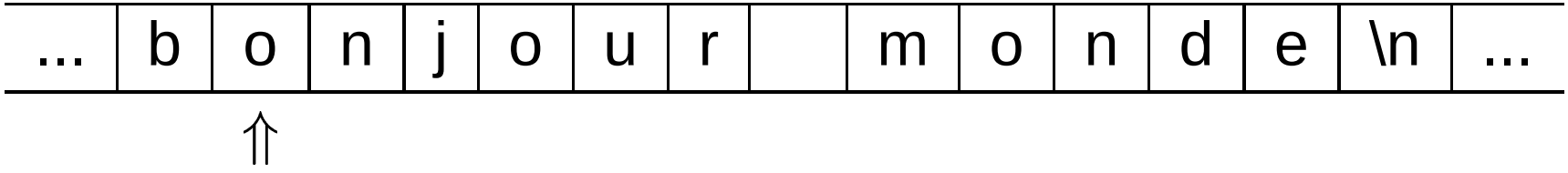
Serge Rosmorduc

rosmord@iut.univ-paris8.fr

Plan

- Notions fondamentales
  - flux , flux texte/binaire, accès direct et séquentiel ;
  - fichiers
- Classes orientées flux

Flux



Définition : Un flux est une suite de données, finie ou infinie, dotée d'un curseur. Il existe deux grands types de flux : les flux en lecture (auquel cas le curseur correspond à une tête de lecture) et les flux en écriture. Les lectures (resp. écritures) dans le flux se font à la position du curseur. L'opération de lecture (resp. d'écriture) avance le curseur.

Exemple de flux : canal de communication en réseau, fichier sur disque, entrée standard au clavier...

Flux Textes et binaires

Flux binaire un flux binaire est une suite d'*octets.* L'interprétation de ces octets est à la charge du programme qui les lit ou les écrit.

Flux texte un flux texte est une suite de *caractères.* Il est stocké selon un système de codage déterminé.

Note : En dernier ressort, un flux texte est toujours contruit au dessus d'un flux binaire.

Flux séquentiels et accès direct

flux séquentiel : un flux séquentiel est un flux dans lequel les caractères sont lus (ou écrits) les uns après les autres, du premier au dernier.

flux en accès direct : un flux en accès direct *(Random Access)* permet la lecture de n'importe quel caractère du flux. La tête de lecture (ou d'écriture) peut se placer à n'importe quel endroit du flux.

Notion de fichier

Définition : un fichier sera ici défini comme un flux fini enregistré sur un disque.

Un fichier peut être envisagé de deux manières : comme objet du système de gestion de fichiers, il a une taille, des droits, une date de création, il appartient à un répertoire... par ailleurs, il contient un flux.

Classes orientées flux

Exemple d'écriture

```
public static void testEcritureTexte (String fname)
        throws IOException {
  // On crée un objet Writer, ce qui ouvre le fichier
  Writer f= new FileWriter(fname);
  // On utilise cet objet pour écrire
  f.write("hello ");
  f.write("world.");
  f.write("It works !");
  // On ferme le flux
  f.close();
}
```

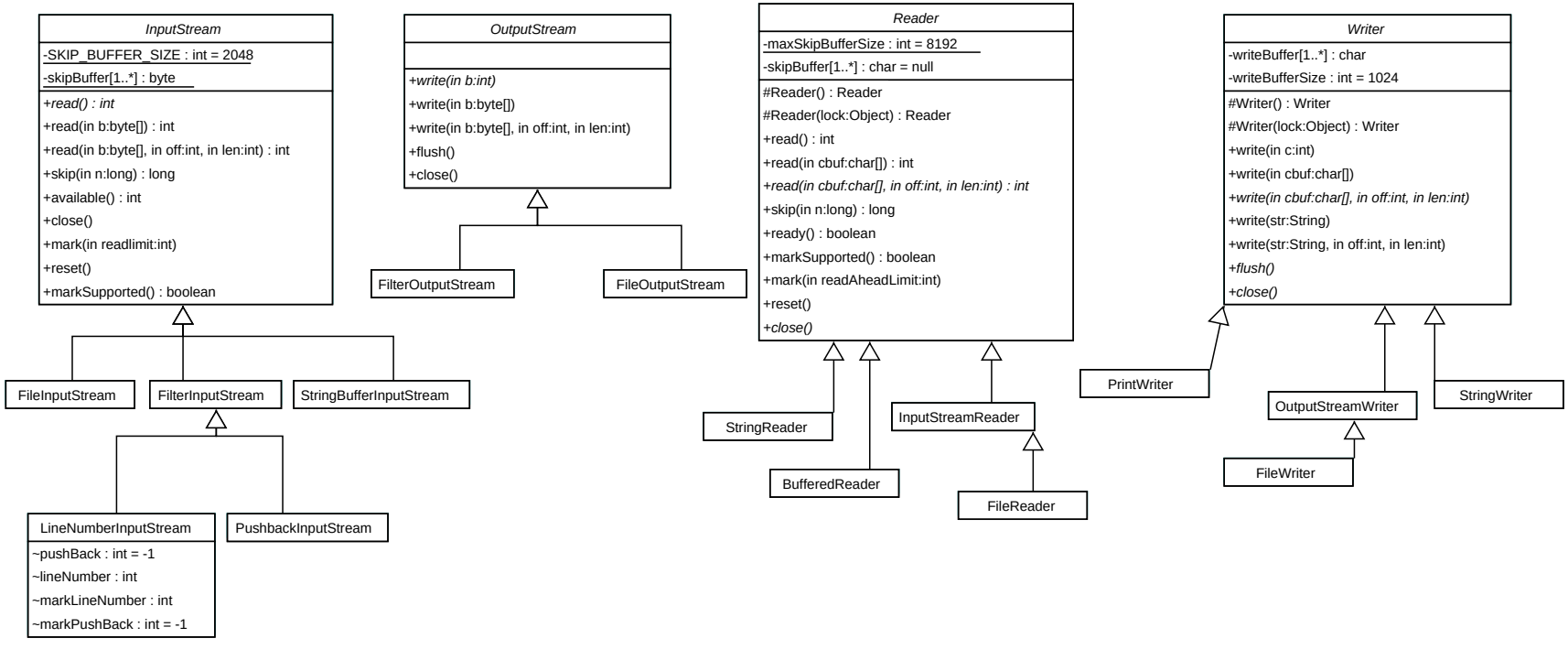
Exemple de lecture

```
public static void testLectureTexte (String fname)
        throws IOException {
  Reader f= new FileReader(fname);
  // Le résultat de read doit être un entier
  // (à cause du -1 qui est renvoyé en fin de fichier)
  int cc;
  cc= f.read();
  while (cc != -1)
  {
    // Pour ranger cc dans un char, il faut un cast :
    char c= (char)cc;
    ...
    cc= f.read(); // Lecture du suivant.
  }
  f.close();
}
```

Architecture des entrées/sorties en Java

		Lecture	Écriture
Flux séquentiels	Binaire	InputStream	OutputStream
	Texte	Reader	Writer

Flux à accès direct : `RandomAccessFile`.



Les classes

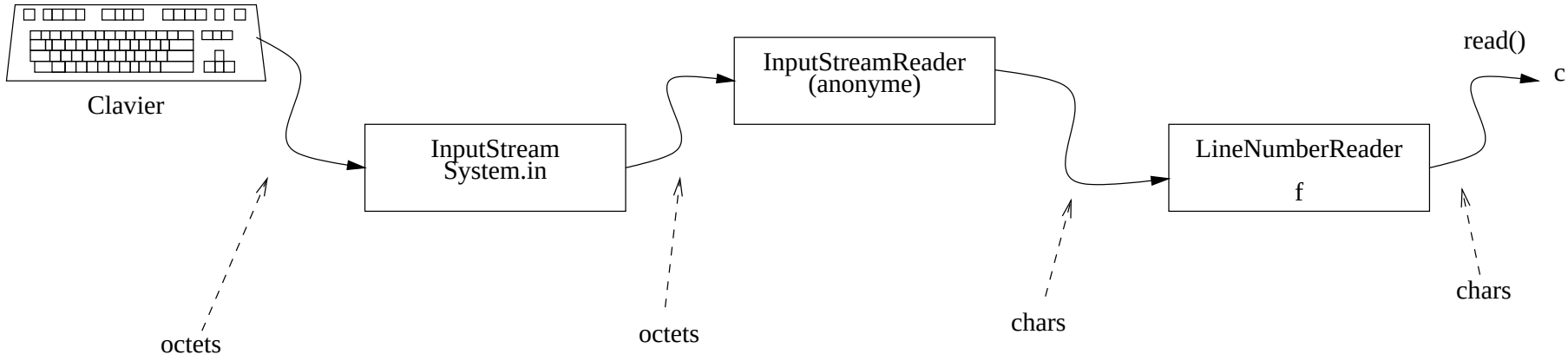
- Typiquement :

$$\left\{\begin{array}{l}\emptyset \\ \texttt{Buffered} \\ \texttt{StringBuffer/ByteArray} \\ \texttt{File} \\ \texttt{Filter} \\ \texttt{Piped}\end{array}\right. \quad \left\{\begin{array}{l}\texttt{InputStream} \\ \texttt{OutputStream} \\ \texttt{Reader} \\ \texttt{Writer}\end{array}\right.$$

- D'autres classes sont plus spécifiques : e.g. `PrintWriter`.

Filtrage

```
LineNumberReader f=
        new LineNumberReader(
          new InputStreamReader(System.in));
...
c= f.read();
```



La classe Writer, étude détaillée

```
void write (int c)
                throws IOException
```

Écrit le caractère dont le code est `c`. Notez que cette méthode fonctionne correctement que `c` soit un `char` ou un `int`.

```
void write (char [] s)
                throws IOException
```

écrit le contenu de `s`.

```
void write (char[] s, int off, int longueur)

                throws IOException
```

écrit les caractères de `s` compris entre l'indice `off` et `off + longueur`.

La classe Writer, étude détaillée (2)

```
void write (String s)
                throws IOException
```

écrit la chaîne s.

```
void flush ()
                throws IOException
```

Réalise effectivement la copie sur le support. C'est utile dans le cas de classes comme `BufferedWriter,` où les données sont stockées temporairement en mémoire.

```
void close ()
                throws IOException
```

ferme le `Writer`

La classe Reader : étude détaillée

```
int read ()
                throws IOException
```

lit un caractère sur l'entrée, et renvoie son code, ou -1 si la fin du fichier est atteinte.

```
int read (char [] buf)
                throws IOException
```

Remplit le tableau `buf` avec les caractères lus.
Attention, cette méthode n'alloue pas le tableau.
La méthode renvoie le nombre de caractères lus, ou -1 si la fin du fichier est atteinte.

La classe Reader : étude détaillée (2)

```
int read (char [] buf, int dep, int
    longueur)
                throws IOException
```

Copie les caractères lus dans `buf`, en commençant à l'indice `dep`, en en lisant au maximum `longueur` caractères.
La méthode renvoie le nombre de caractères lus, ou -1 si la fin du fichier est atteinte.

```
void close ()
                throws IOException
```

ferme le lecteur.

classes orientées fichiers : écriture

Classes `FileOutputStream, FileWriter`.

```
FileOutputStream (File file)
            throws IOException


FileOutputStream (String name)
            throws IOException


FileOutputStream (String name, boolean
   append)
            throws FileNotFoundException
```

classes orientées fichiers : lecture

Classes `FileReader, FileInputStream`

```
FileInputStream (File file)
                throws FileNotFoundException


FileInputStream (String name)
                throws FileNotFoundException
```

Classes orientées chaînes : écriture

CharArrayWriter, ByteArrayOutputStream, StringWriter

```
int c;
FileReader f= new FileReader(args[0]);
StringWriter sw= new StringWriter();
c= f.read();
while (c != -1) {
  // écrit c dans sw
  sw.write(c);
  c= f.read();
}
// Copie le contenu de sw dans s.
String s= sw.toString();
System.out.println(s);
```

Classes orientées chaînes : lecture

`CharArrayReader, ByteArrayInputStream, StringReader, StringBufferInputStream :`

```
Reader f= new StringReader("hello tout le monde"
```

Permet d'écrire d'utiliser des méthodes prévues pour des flux sur des chaînes de caractères.

Ponts entre flux binaires et textes

Les classes `InputStreamReader` et `OutputStreamWriter` permettent de construire un flux orienté caractères au dessus d'un flux d'octets.

Lecture Bufferisée

`BufferedReader` : utilise un *tampon (buffer)*.

- Lecture plus rapide ;
- retour en arrière :

```
void reset ()
            throws IOException

void mark (int limite)
            throws IOException
```

- `String readLine ()`

```
            throws IOException
```

Écriture formatée : la classe `PrintWriter`

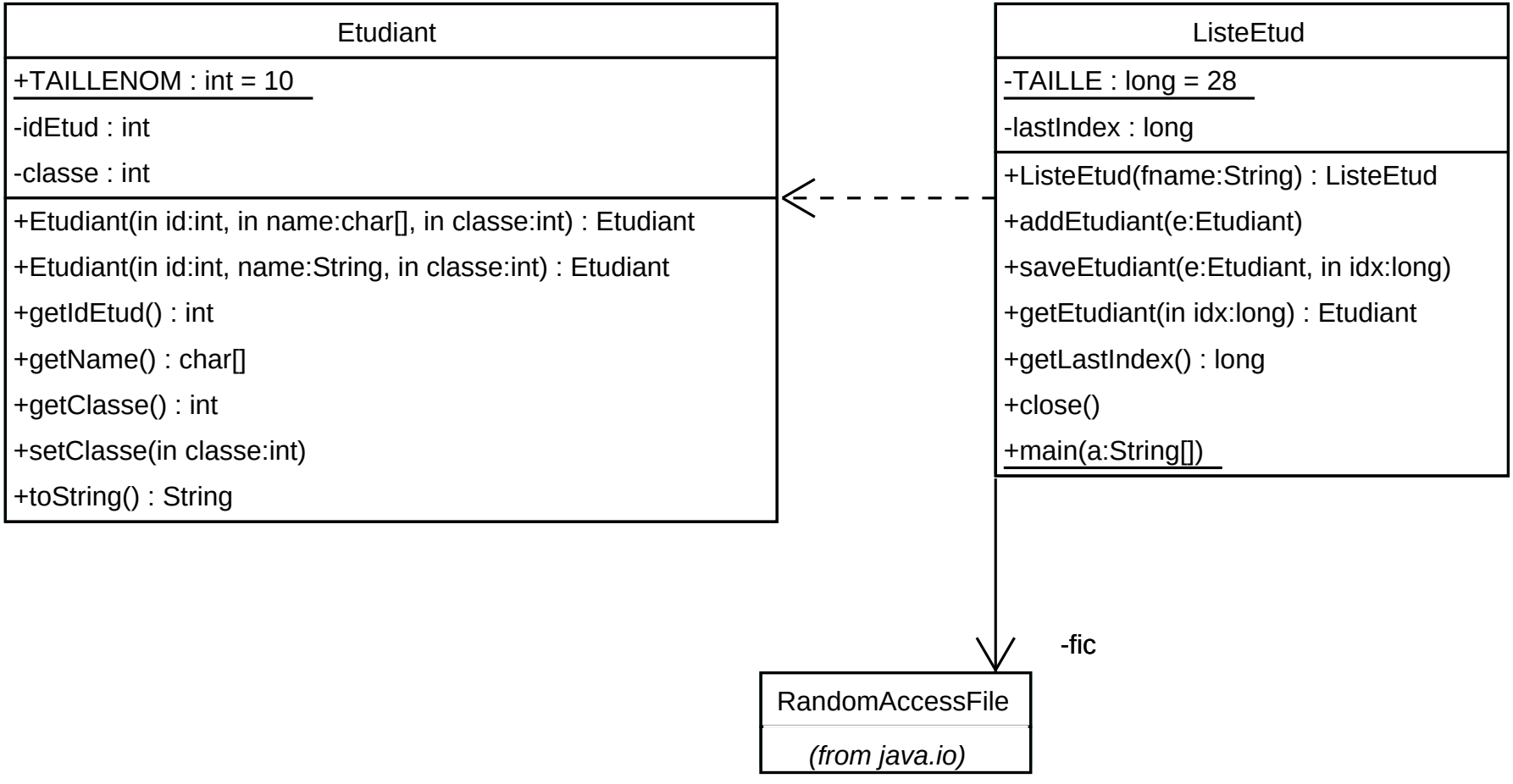
Classe qui dispose de méthodes pour afficher des données non textuelles :

- pas d'exceptions lancées ;
- méthodes `print` et `println` pour les types de base et les tableaux de char.

Fichiers en accès direct

- Fichiers dans lesquels il est possible de positionner la tête de lecture/écriture à n'importe quel endroit
- Pourquoi : Fichiers binaires formatés, avec recherche. Par exemple : bases de données.

Exemple de Fichier en Accès direct



Accès direct

```
class Etudiant {
    public static final int TAILLENOM= 10;
    private int idEtud;
    private char name[];
    private int classe;
    public Etudiant(int id, char name[], int classe)
    {
        this.idEtud= id;
        // Nom sur 10 caractères
        this.name= new char[Etudiant.TAILLENOM];
        for (int i= 0; i<name.length; i++)
            this.name[i]= name[i];
        for (int i= name.length; i< Etudiant.TAILLENOM; i++)
            this.name[i]= ' ';
        this.classe= classe;
    }
    ...
}
```

Accès Direct : Classe ListeEtud

```
public class ListeEtud {
  // Pour cette valeur, voir saveEtudiant.
  private static final long TAILLE= 28;

  private RandomAccessFile fic;
  private long lastIndex;

  public ListeEtud(String fname) throws FileNotFoundException
  {
      fic= new RandomAccessFile(fname, "rw");
  }

  public void close() throws IOException {
      fic.close();
  }
  ...
```

Accès Direct : ListeEtud, écriture

```
/**
 * ajoute un étudiant dans la base.
 */
public void addEtudiant(Etudiant e) throws IOException {
    // Se déplacer à la fin du fichier :
    long idx= fic.length() / TAILLE;
    saveEtudiant(e, idx);
}

public void saveEtudiant(Etudiant e, long idx) throws IOException {
    fic.seek(idx*TAILLE);
    // On écrit les données de l'étudiant :
    fic.writeInt(e.getIdEtud()); // 4 octet.
    for(int i= 0; i< Etudiant.TAILLENOM; i++)
        fic.writeChar(e.getName()[i]); // 20 octets
    fic.writeInt(e.getClasse()); // 4 octet.
    lastIndex= idx;
}
```

Accès Direct : ListeEtud, lecture

```
public Etudiant getEtudiant(long idx) throws IOException {
    lastIndex= idx;
    fic.seek(idx*TAILLE);

    int idEtud;
    char s[]= new char[Etudiant.TAILLENOM];
    int classe;

    idEtud= fic.readInt();
    for(int i=0; i< Etudiant.TAILLENOM; i++)
        s[i]= fic.readChar();
    classe= fic.readInt();
    Etudiant resultat= new Etudiant(idEtud, s, classe);
    return resultat;
}
```

Accès Direct : ListeEtud, démo

```
public static void main(String a[]) throws IOException {
    // Création de la base
    ListeEtud e= new ListeEtud("toto.db");
    e.addEtudiant(new Etudiant(23, "Turing", 1));
    e.addEtudiant(new Etudiant(28, "Babbage", 1));
    e.addEtudiant(new Etudiant(2, "Lovelace", 1));
    e.addEtudiant(new Etudiant(5, "Wirth", 1));
    e.addEtudiant(new Etudiant(10, "Meyer", 1));
    e.close();
```

Accès Direct : ListeEtud, démo

```
        // ... suite =>
        // On ré-ouvre la base :
        e= new ListeEtud("toto.db");
        // On récupère le 3eme étudiant :
        Etudiant etud= e.getEtudiant(3);
        // On l'affiche :
        System.out.println(etud.toString());
        // On le modifie :
        etud.setClasse(2);
        // On le sauve
        e.saveEtudiant(etud, 3);
        // On en cherche un autre :
        etud= e.getEtudiant(0);
        // On l'affiche :
        System.out.println(etud.toString());
        // Vérification de la modif :
        etud= e.getEtudiant(3);
        // On l'affiche :
        System.out.println(etud.toString());
    }
```

Taille des données

Type	Taille (en octets)
byte	1
short	2
char	2
int	4
long	8
float	4
double	8

Sérialisation (1)

- Technique très simple pour sauver des objets dans un fichier ;
- Fichiers dépendent de la JVM ;
- Les classe sauvées, *directement ou indirectement,* doivent implémenter `Serializable` ;

Sérialisation (2)

écriture

```
MaClasse s;
...
// Code qui remplit s
...
ObjectOutputStream o= new ObjectOutputStream(
                        new FileOutputStream("toto.sav"));
o.writeObject(s);
o.close();
```

lecture

```
ObjectInputStream f= new ObjectInputStream(
                        new FileInputStream("toto.sav"));
MaClasse x= (MaClasse)f.readObject();
f.close();
```

Technique de lecture de fichier (1)

- Principe : la structure du code rappelle celle du fichier.
- cas simples : une seule possibilité.

```
Tant que pas fin de fichier
  lire nom
  lire prenom
  lire age
  lire adresse
Fin tant que
```

- techique du look-ahead : on lit un « mot » à l'avance. On peut alors résoudre certaines ambiguïtés.
- Le découpage du texte en mots est délégué à une classe, le *tokenizer.* Le reste du programme raisonne en termes de mots.

Techniques de lecture de fichier (2)

Exemple : programme qui lit, soit des nombres, soit le mot « print », auquel cas il affiche la somme des nombres lus. Toute autre entrée donne lieu à un message d'erreur.

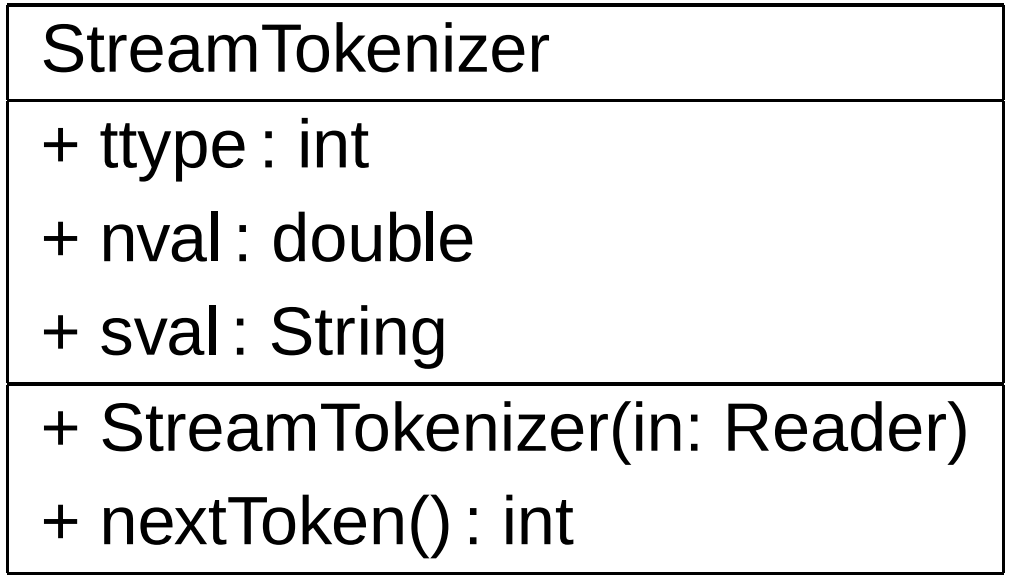
algorithme

```
somme= 0
tokenizer.avancer();
Tant que tokenizer.code != FIN_FICHIER
      Si tokenizer.code = ENTIER alors
              somme= somme + tokenizer.valeurEntiere;
      Sinon Si tokenizer.code = MOT
            Et tokenizer.valeurChaine égal à "print" Alors
                    afficher somme
              Fin Si
      Sinon
              afficher "mauvaise entrée " + tokenizer.valeurChaine
      Fin Si
      tokenizer.avancer();
Fin Tant que
```

La classe StreamTokenizer (1)

Classe extrêmement puissante et paramétrable pour lire un

fichier :



Exemple :

```
import java.io.*;
public class TestTok {
    public static void addition() throws IOException {
        double total= 0.0; int token;
        StreamTokenizer s=
            new StreamTokenizer(new InputStreamReader(System.in));
```

La classe StreamTokenizer (2)

```
s.nextToken();
while( s.ttype != StreamTokenizer.TT_EOF)
    {
        switch (s.ttype) {
        case StreamTokenizer.TT_NUMBER: total+= s.nval;
            break;
        case StreamTokenizer.TT_WORD:
            if (s.sval.equals("print"))
                System.out.println("total= " + total);
            else
                System.out.println("inconnu " + s.sval);
            break;
        default:
            System.out.println("chaîne inattendue: "
                    + (new Character((char)token)));
        }
        s.nextToken();
    }}}
```

La classe File

- Permet de gérer les aspects « extérieurs des fichiers » ;
- Un objet de classe `File` représente un *fichier* ou un *répertoire*, existant ou non.
- Les méthodes applicables à un objet f de type `File` permettent entre autres :
  - de savoir s'il existe (`exists()`) ;
  - de connaître sa taille (`length()`) ;
  - de savoir si c'est un répertoire (`isDirectory()`)
  - de connaître les droits qui s'y appliquent (`canRead()` et `canWrite()`) ;
  - de le détruire (`delete()`)...

Rappel sur les fichiers

nom : un fichier a un nom. Dans un répertoire donné, un seul fichier peut porter un nom donné. Par contre, il peut y avoir plusieurs fichiers portant le même nom dans des répertoires différents.

chemin d'accès : le *path* (chemin d'accès) identifie véritablement un fichier. Exemple : `/home/Profs/rosmord/TP/fichier.java`

***path* relatif** quand un *path* est donné à partir du répertoire *courant,* il est dit « relatif ». Exemple : je suis dans `/home/Profs/rosmord/TP1`. Le fichier précédent peut être désigné par : `../TP/fichier.java`. Si j'étais dans `TP`, alors le path relatif `fichier.java` suffirait.

File : constructeurs

`public File (String pathname)`

Crée un nouvel objet `File`, correspondant au chemin (path) indiqué. Par exemple, avec

```
File f= new File(".");
```

`f` désignera le répertoire courant. Le chemin peut correspondre à un fichier ou à un répertoire.

`public File (File parent, String child)`

Crée un nouvel objet `File`, correspondant au chemin (path) composé en ajoutant `child` à `parent` :

```
File homedir= new File("/home/rosmord");
File f= new File(homedir, "MonProg.java");
```

File : quelques méthodes

`boolean` **`canRead`** `()`
Renvoie vrai si le fichier est lisible.

`boolean` **`canWrite`** `()`
Renvoie vrai si le fichier est autorisé en écriture.

`long` **`lastModified`** `()`
Retourne la date de dernière modification.

`boolean` **`isDirectory`** `()`
Renvoie vrai si le fichier est un répertoire.

`boolean` **`isFile`** `()`
Renvoie vrai si le fichier est un fichier normal.

`long` **`length`** `()`
Retourne la longueur du fichier

File : quelques méthodes

`boolean` **`exists`** `()`
Renvoie vrai si le fichier existe.

`String` **`getName`** `()`
Retourne le nom du fichier.

`String` **`getPath`** `()`
Retourne le chemin du fichier (nom inclus).

`boolean` **`delete`** `()`
Détruit le fichier ou le répertoire correspondant.

`boolean` **`renameTo`** `(File dest)`
Renomme un fichier.

`boolean` **`setReadOnly`** `()`
Place un fichier en lecture seule.

File : Méthodes orientées répertoires

`boolean` **`mkdir`** `()`

Crée le répertoire correspondant à `this`.

`String[]` **`list`** `()`

Renvoie la liste des noms des fichiers contenus dans `this`, qui doit bien entendu être un répertoire.

`File[]` **`listFiles`** `()`

Renvoie la liste des fichiers contenus dans `this`.

`File[]` **`listFiles`** `(FilenameFilter filter)`

Idem, mais ne concerne que les fichiers sélectionnés par filter. Voir l'exemple de code en introduction.

File : exemples

Clean.java

```
1  import java.io.*;
2
3  public class Clean implements FilenameFilter {
4
5      public boolean accept(File dir, String name) {
6          if (name.charAt(name.length() -1) == '~') return true;
7          else return false;
8      }
9
10     static public void main(String args[]) {
11         Clean filter= new Clean();
12         File dir= new File (args[0]);
13         if (dir.isDirectory()) {
14             File [] fichs= dir.listFiles(filter);
15             for (int i= 0; i< fichs.length; i++)
16                 fichs[i].delete();
17         }
18     }
19 }
```