Entrées/Sorties

FIP ING 39

Serge Rosmorduc
serge.rosmorduc@lecnam.net
Conservatoire National des Arts et Métiers

2016-2017

Technique de lecture de fichier texte

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

```
1       Tant que pas fin de fichier
2         lire nom
3         lire prenom
4         lire age
5         lire adresse
6       Fin tant que
```

- technique 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 (analyseur lexical).* Le reste du programme raisonne en termes de mots.

Notion de « grammaire »

- Quand on veut décrire précisément un format, on écrit sa *grammaire* en décrivant son *lexique* et sa *syntaxe.*
- bases théoriques solides, algorithmes multiples (mais dépassent le cadre de ce cours) (voir ANTLR pour un outil) ;

Exemple : pseudo-java

```
instructionif ::=
  'if' '(' expression ')'
    instruction ('else' instruction)? ;
expression ::=  subExpression '==' subExpression;
subExpression::=  IDENT |  VAL;
instruction::=  IDENT '=' expression ';'
  |  IDENT'()' ';'
  |  '' (instruction)* ''
 ;
VAL  ::=  ['0'-'9']+;
IDENT  ::=  ['a'-'z"A'-'Z']['a'-'z"A'-'Z"0'-'9']*;
```

pas si simple à écrire : ambiguïtés, etc.

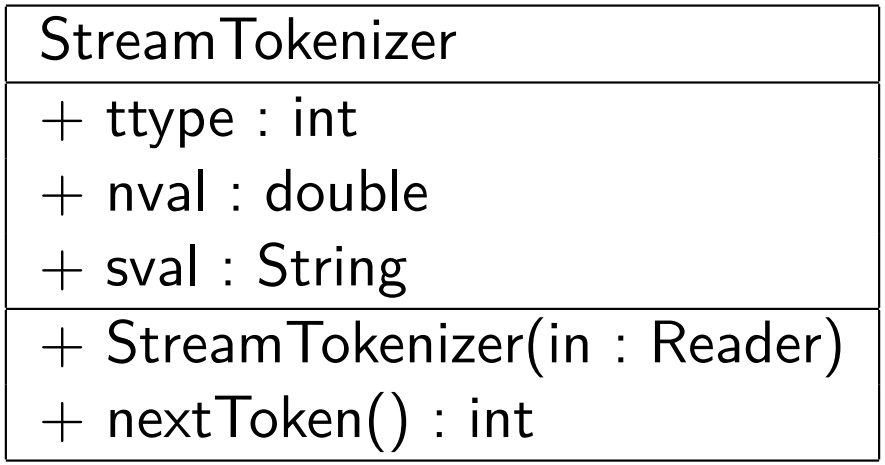
Classe d'analyse lexicale

Le but : découper un texte en unités (*tokens* ou *lexèmes*) correspondant aux mots et autres symboles utilisés par le langage analysé (pour java, par exemple : identifiants, opérateurs comme « == », nombres, ponctuations...)
Un *lexer* ou *analyseur lexical* ou *tokenizer* est un objet qui

- conserve le dernier « token/lexème » lu ;
- permet de savoir de quel type (nombre, texte, etc..) est ce token ;
- permet d'avancer et de se placer sur le token suivant.

La classe StreamTokenizer

Classe extrêmement puissante et paramétrable pour lire un fichier :



- `nextToken()` : passe au token suivant (à appeler dès le départ) et retourne son « type » ;
- `ttype` : type (« texte », « nombre »...) du dernier token lu ;
- nval : valeur numérique du dernier token lu, si c'est un nombre ;
- sval : dernier mot lu, si le token est un mot.

StreamTokenizer, algo

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.

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

```
double total= 0.0; int token;
StreamTokenizer s=
     new StreamTokenizer(new InputStreamReader(System.in));
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)s.ttype)));
  }
  s.nextToken();
}
```

StreamTokenizer

Saute les espaces.

- si un nombre a été lu, sa valeur est dans `nval` ;
- si un *mot* a été lu (`TT_WORD`), sa valeur est dans `sval`
- si une chaîne entre guillemets a été lue (voir `quoteChar`), `ttype` vaut *le code du caractère de guillemets utilisé*, et le contenu est dans `sval` ;
- si autre chose a été lu, c'est un caractère isolé et son code est dans `ttype`.
- pour tous les opérateurs, parenthèses, etc... on utilise donc `ttype` directement.

Dans tous les cas, `tokenizer.toString()` renvoie le token lu.

Paramétrage de StreamTokenizer

À appeler *avant le début de* la lecture.

- `eolIsSignificant(boolean)` : la fin de ligne est elle un espace « normal » ?
- `lowerCaseMode(boolean)` : passe tout en minuscule si vrai ;
- `parseNumbers()` : demande d'analyser les nombres (appelée par défaut) ;
- `slashSlashComments(boolean)` : commentaires « // »
- `slashStarComment(boolean)` : commentaires « /*....*/ »
- `resetSyntax()` remet la syntaxe à 0

Paramétrage de StreamTokenizer

- `commentChar(char)` : permet de fixer un caractère de commentaires ;

```
1    tok.commentChar('#');
```

- `wordChars(int low, int hi)` définit des caractères comme composant des *mots* ;
- `ordinaryChars(int low, int hi)`
- `ordinaryChar(int c)` définit des caractère comme des symboles isolés ;
- `whitespaceChars(int low, int hi)` : définit des caractères comme étant des espaces ;
- `quoteChar(int c)` : utilise c comme guillemet

Paramétrage de StreamTokenizer

- Paramétrage par défaut de `StreamTokenizer` peu satisfaisant.
- Bonne base pour le redéfinir :

```
StreamTokenizer tok= new StreamTokenizer(....);
// remise à zéro...
tok.resetSyntax();
// on fixe les caractères des mots
tok.wordChars('a', 'z');
tok.wordChars('A', 'Z');
tok.wordChars(128 + 32, 255); // accents usuels
// on fixe l'espace comme espace...
tok.whitespaceChars(0, ' ');
// en option : analyse des nombres
tok.parseNumbers();
```

Autre exemple

Fichier décrivant une image. Grammaire :

```
fichier ::= cercle*
cercle ::=  'CERCLE' '(' INT INT INT ')'
```

Code :...

```
import java.io.*;

public class LireCercles {
 public static void error(String e) {
  throw new RuntimeException("erreur de syntaxe "+ e);
 }

  public static void main(String[] args) throws IOException {
    double cx, cy, r; // pour stocker les données
    StreamTokenizer t= new StreamTokenizer(
                    new InputStreamReader(System.in));
    t.nextToken(); // On lit le premier token !!!
```

```
while (t.ttype != StreamTokenizer.TT_EOF) {
   if (t.ttype != StreamTokenizer.TT_WORD
       || ! "CERCLE".equals(t.sval))
     error(" 'CERCLE' attendu"); // On veut voir
   t.nextToken();
   if (t.ttype != '(') error("'(' attendue");
   t.nextToken();
   if (t.ttype != StreamTokenizer.TT_NUMBER)
      error("nombre attendu");
   cx= t.nval; t.nextToken();
   if (t.ttype != StreamTokenizer.TT_NUMBER)
      error("nombre attendu");
   cy= t.nval; t.nextToken();
   if (t.ttype != StreamTokenizer.TT_NUMBER)
      error("nombre attendu");
   r= t.nval; t.nextToken();
   System.out.println("lu cercle " + cx + ","+ cy + ","+ r);
   if (t.ttype != ')') error("'(' attendue");
     t.nextToken();
}}}
```

Scanner : Alternative à StreamTokenizer ?

- plus simple ;
- lit plus de types (int, boolean...) ;
- mais utilise des *séparateurs*. Ne sais pas lire « 3+4 » comme « 3 » « + » « 4 », par exemple ;
- néanmoins très utile ;
- voir annexes.

Entrées/Sorties binaires

Flux d'entrées/sorties binaires

- Cette fois, l'unité de base n'est plus le caractère mais **l'octet (byte)** : un nombre entre 0 et 255.
- on utilise des `InputStream` pour la lecture et des `OutputStream` pour l'écriture.
- les classes réellement utilisées descendent de ces deux classes.
- *tout* ce qui est manipulé par un ordinateur est, en dernier ressort, binaire !

Le tableau qu'il faut connaître

	Lecture	**Écriture**
Binaire	`InputStream`	`OutputStream`
Texte	`Reader`	`Writer`

InputStream

Classe abstraite, ancêtre de toutes les classes de flux d'entrée binaires.

A priori, toutes les méthodes qui suivent lèvent IOException.

Méthodes

int read()

avance, puis renvoie le code de *l'octet* sous la tête de lecture, ou -1 si on est à la fin du fichier.

void close()

ferme le flux (important !)

FileInputStream

InputStream qui lit dans un fichier.

Constructeurs

FileInputStream(String fileName)

Ouvre un inputStream sur un fichier dont on fournit le nom (en fait, le chemin d'accès comme `/home/rosmord/toto.png`, `toto.dat` ou `C:\Data\toto.txt`)

FileInputStream(File file)

Ouvre un inputStream sur le fichier file.

(pas de méthode spécifique à FileInputStream)

FileInputStream (2)

Exemple

```
FileInput r= new FileInputStream("toto.dat");
int c= r.read(); int nb= 0;
while (c!= -1) {
  nb++;
  c= r.read();
}
r.close();
System.out.println("taille␣"+ nb + "␣octets");
```

ByteArrayInputStream

Flux lisant dans un tableau d'octets... pratique pour écrire des tests.

Constructeur

ByteArrayInputStream(byte[] bytes)

Crée un InputStream qui lira le contenu de bytes.

DataInputStream

décorateur qui ajoute à un InputStream des méthodes permettant de lire des données primitives (int, double...) en format binaire.

Constructeur

DataInputStream(InputStream in)

Crée un DataInputStream qui lira le contenu de in.

OutputStream

Flux binaire ouvert en écriture. Classe abstraite. *Comme pour les InputStreams, les méthodes lèvent IOException.*

Méthodes

void write(int c)

écrit l'octet c (entre 0 et 255) sur le flux.

void close()

Ferme le flux. Important ! ! !

FileOutputStream

Flux d'écriture dans un fichier binaire.

Constructeurs

FileOutputStream(String nomFichier)

Crée un flux binaire qui écrit dans le fichier `nomFichier`. Le fichier est créé, et, s'il existe déjà, *vidé.*

FileOutputStream(File file)

Crée un flux binaire qui écrit dans le fichier `file` (voir ci-dessus).

FileOutputStream (2)

Exemple simple

```
1 FileOutputStream w= new FileOutputStream("test.dat");
2 w.write(10);
3 w.write(11);
4 w.write(200);
5 w.close();
```

ByteArrayOutputStream

OutputStream qui écrit en mémoire (utile pour les tests).

Constructeur

ByteArrayOutputStream()

crée un ByteArrayOutputStream.

Méthodes

byte[] toByteArray()

permet de récupérer les octets écrits dans le ByteArrayOutputStream.

Codage des flux texte

Codage des flux textuels

- En réalité, l'ordinateur ne travaille que sur des données binaires ;
- un flux texte est donc construit à partir d'un flux binaire ;
- mais comment ?

Flux textes et codage des caractères

Approche simple (anciennement) :

- un fichier est une suite de nombre entre 0 et 255 ;
- le codage d'un flux texte associe à chaque nombre un caractère ;
- le codage dépend généralement du système et de sa configuration.

Exemple

caractère	code ASCII	code latin-1	code MacRoman
espace	32	32	32
0 (zéro)	48	48	48
1 (un)	49	49	49
a	97	97	97
A	65	65	65
é	(non)	233	142
œ	(non)	**(non)**	207

Unicode

- Un codage pour *tous* les caractères $\rightarrow$ dépasse l'octet !
- Les codes vont de 0 à 0x10FFFF (1.114.111)
- comment les représenter concrètement : UTF-8, UTF-16 (BE/LE), UTF-32(BE/LE)
  - pour les fichiers : UTF-8 (plus compact, pas d'ambiguïté BE/LE) ;
  - en interne, un char java est en UTF-16.

texte	u	n		é	t	é
Latin-1	0x75	0x6E	0x20	0xE9	0x74	0xE9
UTF-8	0x75	0x6E	0x20	0xC3 0xA9	0x74	0xC3 0xA9

Fins de lignes

trois codages différents

- unix : saut de ligne, '\n', code 10
- mac : retour chariot, '\r', code 13
- dos/windows : '\r\n'

Attention : un fichier créé par windows et lu sous unix comportera probablement le saut de lignes windows !
BufferedReader traite correctement les fins de ligne, quel que soit leur type.

InputStreamReader

Pont entre binaire et texte

Reader (donc texte) qui prend ses données dans un flux binaire.

Constructeur

InputStreamReader(InputStream in, String charsetName)

Crée un reader qui prendra ses données dans `in`, dans le codage `charsetName` (ISO-8859-1 ou UTF-8 par exemple).

InputStreamReader(InputStream in)

Crée un reader qui prendra ses données dans `in`, interprété avec le codage par défaut pour cet ordinateur

```
1 Reader r= new InputStreamReader(
2                   new FileInputStream("toto.txt"),
3                   "UTF-8");
```

OutputStreamWriter

Writer qui écrit ses données dans un flux binaire.

Constructeur

OutputStreamWriter(OutputStream out, String charsetName)

Crée un writer qui écrira ses données dans `out`, dans le codage `charsetName` (ISO-8859-1 ou UTF-8 par exemple).

OutputStreamWriter(OutputStream out)

Crée un writer qui écrira ses données dans `out`, dans le codage par défaut pour cet ordinateur (MACROMAN pour Mac, souvent UTF-8 sous linux...)

Classes orientées Fichiers

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()`)...
- en java 1.7, remplacement possible : la classe `java.nio.file.Path`.

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

```
1       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` :

```
1       File homedir= new File("/home/rosmord");
2       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

`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

```
import java.io.*;
public class Clean implements FilenameFilter {

    public boolean accept(File dir, String name) {
        if (name.charAt(name.length() -1) == '~')
        return true;
        else return false;
    }

    static public void main(String args[]) {
        Clean filter= new Clean();
        File dir= new File (args[0]);
        if (dir.isDirectory()) {
            File [] fichs= dir.listFiles(filter);
            for (int i= 0; i< fichs.length; i++)
                fichs[i].delete();
        }
    }
}
```

« Path » et « Paths »

- `File` : pas de support des fonctionnalités spécifiques des systèmes d'exploitation (liens, droits complexes).
- `Path` (interface) représente un chemin ;
- `Paths` « factory » de `Path`
- `Files` classe utilitaire contient des méthodes statiques pour manipuler des objets de classe `Path`

Path

- accès facile aux éléments du path ;
- comparaison de chemins ;
- possibilité d'être averti quand le fichier est modifié.

Files

- contient la plupart des méthodes de manipulation de path ;
- création de dossier ;
- copie, déplacement, destruction de fichier ;
- création de liens symboliques, de dossiers ou fichiers temporaires ;
- gestion des droits des fichiers ;
- tests de droits, d'existence des fichiers ;
- création d'objets flux ;
- visiteurs pour parcourir une arborescence ;
- méthodes simplifiées pour la lecture et l'écriture `readAllLines...`

Annexes (hors programme)

Annexes

Dans le cours, nous avons choisi de nous concentrer sur un certain nombre de points importants pédagogiquement parlant.
Mais les entrées/sorties en Java sont un domaine très vaste (et remanié à plusieurs reprises).
Ces annexes, hors programme, pointent sur quelques classes et bibliothèques utiles.

Mise en forme

Tout ce qui concerne la mise en forme du texte est dans le package `java.text`. Exemples :

- formatages divers : `DecimalFormat, NumberFormat`....

```
1  DecimalFormat format= new DecimalFormat("00000.00");
2  double x= 1.0/3;
3  String s= format.format(x);
```

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.

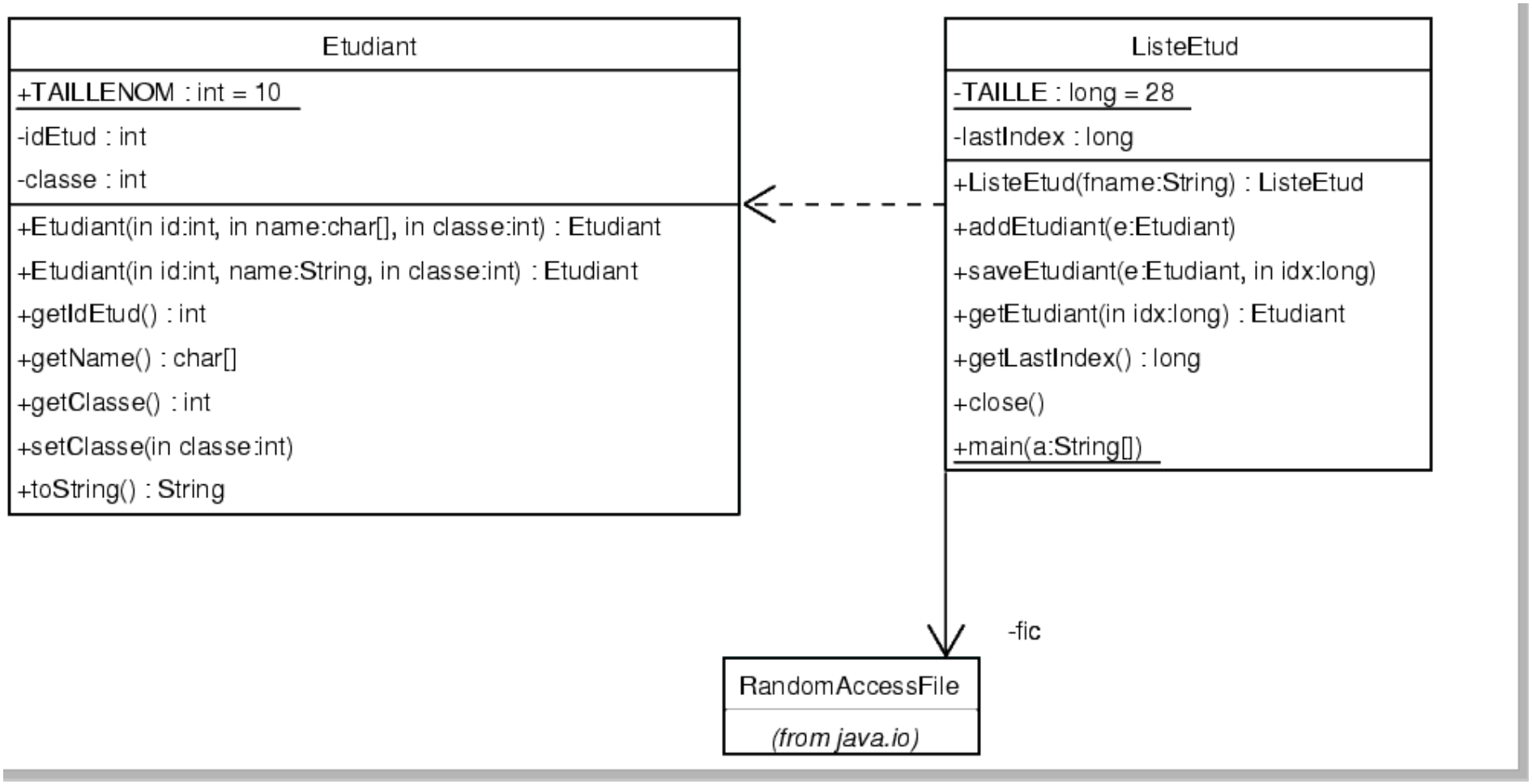
flux en accès direct : un flux en accès direct *(Random Access)* permet la lecture de n'importe quel caractère du flux. Le programmeur peut déplacer la tête de lecture (ou d'écriture) à n'importe quel endroit du flux.

En java, l'accès direct se fait grâce à `RandomAccessFile`.

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 : classe Etudiant

```
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

```java
/**
 * 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

```
1  public Etudiant getEtudiant(long idx) throws IOException {
2      lastIndex= idx;
3      fic.seek(idx*TAILLE);
4
5      int idEtud;
6      char s[]= new char[Etudiant.TAILLENOM];
7      int classe;
8
9      idEtud= fic.readInt();
10     for(int i=0; i< Etudiant.TAILLENOM; i++)
11         s[i]= fic.readChar();
12     classe= fic.readInt();
13     Etudiant resultat= new Etudiant(idEtud, s, classe);
14     return resultat;
15 }
```

Accès Direct : ListeEtud, démo

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

```
        // ... 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

New Input Output : java.nio.*

« Nouvelles » (depuis 10 ans...) bibliothèques permettant l'accès aux capacités avancées des systèmes d'exploitation sous-jacents à la JVM, en particulier :

- des entrées/sorties plus efficaces, par l'intermédiaire de buffers ;
- la possibilité de projeter une partie de fichier en mémoire *Memory-mapped I/O*
- la gestion de verroux pour garantir qu'un fichier n'est ouvert qu'une fois.
- la gestion d'entrées/sorties asynchrones

Bibliographie :
http ://www.cs.brown.edu/courses/cs161/papers/j-nio-ltr.pdf

Channels et Buffers

Channel abstraction d'un canal de communication. Sous interfaces : ReadableByteChannel, WritableByteChannel, InterruptibleChannel.

Buffer bloc de données lu ou écrit dans un Channel.

Création d'un channel

- À partir d'un objet de classe RandomAccessFile, FileOutputStream, FileInputStream, Socket, ServerSocket ou DatagramSocket par la méthode `getChannel()`
- La classe Channels fournit des méthodes statiques pour créer des channels à partir d'InputStream, d'OutputStream, de Reader et de Writer (et réciproquement).

```
try (
  FileInputStream in = new FileInputStream("entree.data");
  FileOutputStream out = new FileOutputStream("sortie.data");
) {
 FileChannel channelEntree = in.getChannel();
 FileChannel channelSortie = out.getChannel();

 ByteBuffer buffer= ByteBuffer.allocateDirect(1024*16);

 int success= channelEntree.read(buffer);

 while (success != -1) {
   // prépare le buffer pour l'écriture :
   //   position revient en début de buffer
   //   la limite est placée à la fin de ce qui a été lu
   buffer.flip();
   // écrit
   channelSortie.write(buffer);
   // prépare le buffer pour la lecture
   buffer.clear();
   success= channelEntree.read(buffer);
 }
} // tout est fermé (try with resources)
```

Données associées à un buffer

position position de la tête de lecture/écriture dans le buffer ;

limit valeur maximale (exclusive) de position ;

capacity valeur maximale de limite.

`clear()` met `position` à 0 et `limit` à `capacity`.
`flip()` met `limit` à la valeur courante de `position`, et `position` à 0.

Quelques méthodes de ByteBuffer

- Méthodes `get()` pour lire et `put()` pour écrire ;
- spécialisées pour les types de base :
  - float getFloat()
  - float getFloat(int index)
  - void putFloat(float f)
  - void putFloat(int index, float f)
- ordre des octets (BIG ENDIAN ou LITTLE ENDIAN) paramétrable par la méthode `order(ByteOrder)`

Memory-mapped I/O

Projette un fichier en mémoire. Les écritures dans le Buffer obtenu sont automatiquement répercutées sur le fichier. Le fichier peut être partagé.
Méthode de FileChannel :

```
MappedByteBuffer map (MapMode mode, long position, long
    size)
            throws NonReadableChannelException,
NonWritableChannelException,IllegalArgumentException,
IOException
```

MapMode est :

MapMode.READ_ONLY lecture seule ;
MapMode.READ_WRITE lecture et écriture ;
MapMode.PRIVATE : lecture et écriture, mais en cas d'écriture, les données sont détachées du fichier.

Le `MappedByteBuffer` est un `Buffer`, mais il dispose aussi d'une méthode `force()` qui garantit l'écriture. La méthode `array()`, si elle fonctionne, permet d'obtenir le tableau d'octets sous-jacent (utiliser `hasArray()` pour savoir si `array()` fonctionne.

Verroux (lock)

Système permettant d'interdire la lecture ou l'écriture de tout ou partie d'un fichier par d'autres programmes.
Principe simplifié (verroux exclusif) :

1. on demande un verroux sur une zone d'un fichier. Si aucun autre programme ne verrouille la zone en question, le verroux est acquis.
2. tant que le verroux n'est pas relâché, aucun autre programme ne peut verrouiller la zone en question.

Remarques importantes :

- c'est un mécanisme collaboratif. Si un programme écrit dans le fichier sans demander de verroux, il peut le faire ;
- seul le support de verroux exclusifs sur la totalité des fichiers est garanties dans toutes les JVM.

Verroux exclusifs

Adaptés à l'écriture dans une zone.

- Pour obtenir un verrou exclusif, il faut qu'aucun verrou ne soit placé sur la zone demandée ;
- Quand un verroux exclusif est en place, aucun autre verroux ne peut être obtenu sur la zone protégée.

Verroux non exclusifs

Adaptés à la lecture dans une zone.

- Plusieurs processus peuvent détenir des verroux non exclusifs sur une même zone ;
- Par contre, le verroux non exclusif est incompatible avec le verroux exclusif.

C'est logique : plusieurs lectures simultanées ne posent pas de problème *a priori.* Par contre, une écriture empêche toute lecture.

Mécanisme de verrouillage

Méthodes bloquantes

(Méthodes de FileChannel)

`FileLock` **`lock`** `()`
`throws` **`ClosedChannelException, OverlappingFileLockException, IOException`**

tente d'acquérir un verrou exclusif sur un fichier. Bloque en attendant.

`FileLock` **`lock`** `(long position, long size, boolean shared)`
`throws` **`ClosedChannelException, OverlappingFileLockException, IOException`**

tente d'acquérir un verrou sur une partie d'un fichier. Pour un accès exclusif, `shared` doit être à faux.

Mécanisme de verrouillage

Méthodes non bloquantes

`FileLock` **`tryLock`** `()`
`throws` **`ClosedChannelException`**`,`
`OverlappingFileLockException``,` **`IOException`**

Idem `lock()`, mais l'appel n'est pas bloquant. Si le verrou ne peut être obtenu, retourne `null`.

`FileLock` **`tryLock`** `(long position, long size, boolean shared)`
`throws` **`ClosedChannelException`**`,`
`OverlappingFileLockException``,` **`IOException`**

Appel non bloquant.

Pour obtenir un verrou exclusif, le fichier concerné *doit être ouvert en écriture.*

Mécanisme de verrouillage

Méthodes de FileLock

```
void release ()
                throws ClosedChannelException, IOException
```

libère le verrou.

Exemple

```
// Verrou non exclusif sur un fichier entier
FileLock lock = channel.Lock(0, Long.MAX_VALUE, true);
if (lock != null) {
   ... code ...
 lock.release();
}
```

- IO asynchrones : la lecture ou l'écriture ne bloquent plus le programme indéfiniment.
- sélecteur : système permettant de se mettre en attente sur plusieurs flux. Intéressant pour des serveurs.

La classe Scanner

Plus simple que le StreamTokenizer, plus riche pour les type lus.
On lit des « mots » (tokens) **séparés par un séparateur** (par défaut les espaces).

- Pour lire un entier (un double...) , on utilise `val= scanner.nextInt();` (`scanner.nextDouble()`...) ;
- `scanner.next()` retourne le prochain « token », sous forme de `String` ;
- `scanner.hasNextInt()`, `scanner.hasNext()` : permettent de savoir si un int (resp. un token quelconque) a été lu ;
- `scanner.nextLine()` passe à la ligne (et retourne le texte lu, sauf le saut de ligne) ;

Paramétrage

- `scanner.useDelimiter(separateurs);` fixe le délimiteur (expression régulière)
- `scanner.useRadix(base)` fixe la base à utiliser pour les entiers

Utilisation de scanner

```
double val= 0;
while (scan.hasNext()) {
  if (scan.hasNextDouble()) {
    val= val+ scan.nextDouble();
  } else {
    String texte= s.next();
    if ("print".equals(texte)) {
        System.out.println("somme " + val);
    } else {
System.out.println("Texte inattendu " + texte);
    }
  }
}
```

Utilisation de scanner

Attention, l'utilisation de séparateurs est une contrainte importante :

```
1 int a= scanner.readInt();
2 String op= scanner.next();
3 int b= scanner.readInt();
```

lit correctement :

```
3 + 4
```

mais pas

```
3+4
```

(3+4 sans espace est *un seul token*)

Scanner et expressions régulières

L'objet scanner peut aussi reconnaître des éléments décrits par des *expressions régulières.*

- `boolean hasNext(String pattern`
- `String next(String pattern)`...

```
String heurePattern= "[0-9][0-9]h[0-9][0-9]";
if (scanner.hasNext(heurePattern)
  String h= scanner.next(heurePattern);
  ...
```

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` ;

Exemple

```
1  public class Joueur implements Serializable {
2    private int numero;
3    private String nom;
4    private int score;
5    ... accesseurs, etc...
6  }
7
8
9  public class Equipe implements Serializable {
10   private ArrayList<Joueur> joueurs;
11   ....
12 }
```

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

Lecture...

```
ObjectInputStream f= new ObjectInputStream(
                           new FileInputStream("equipe.sav"));
Equipe x= (Equipe)f.readObject();
f.close();
```