Sommaire

I. Introduction

1. Objectifs

Ce chapitre introduit brièvement les entrées/sorties en Java. En Java, à part les notions fondamentales sur le langage, il est important de savoir manipuler les entrées/sorties. Nous allons parcourir l'API d'e/s et les classes concernées, ensuite les illustrer à partir de quelques exemples.

2. Contexte

La plupart des fonctionnalités d'entrée/sortie supportées par Java sont dans le paquetage *java.nio.file*. Cependant, un tel API possède plusieurs classes, mais seulement certaines d'entre elle permettent d'interagir avec cet API, ce qui rend l'utilisation facile.

3. Système de fichiers

Un fichier est une abstraction du système d'exploitation pour un stockage générique. La partie du système d'exploitation responsable de la gestion des fichiers s'appelle « Système de Fichier » (*file system)*.

De façon général, les systèmes de fichiers organisent, stockent et nomment (identification par nom) les fichiers stockés dans un périphérique de stockage persistant comme les disques durs, les clés USB, les DVD, etc.

De nos jours, il existe des systèmes de fichiers populaires tels que le « etx3 » pour les systèmes d'exploitation Linux, « NTFS » pour les systèmes d'exploitation basés sur Windows NT comme Windows XP, 7, 8 et « ISO9660 » ou « UDF » pour les support optiques tels que les CD, DVD. Cependant, bien que chaque système de fichier offre sa propre vision de comment les données sont stockées et les gère de sa propre manière, tous les systèmes de fichiers partagent certains aspects communs :

- Les fichiers sont souvent organisés dans des structures hiérarchiques (arbres) de dossiers (répertoires). Les dossiers sont des fichiers qui permettent l'organisation des données sur le périphérique de stockage.
- Le nom de fichier est lié à sa position dans le répertoire qui le contient, ce qui permet non seulement de l'identifier de façon unique mais aussi de le retrouver par son nom.

- Les fichiers sont souvent associés à des métadonnées telles que la date de création, la date de la dernière modification, le propriétaire, les différentes permissions accordées autres utilisateurs sur eux (lecture, écriture, etc). Cela convertit le système de fichier à une base de données avec laquelle on peut faire des recherches sur les fichiers qu'il gère.

II. Chemins: Nom de fichier

Un fichier est identifié par son chemin dans le système de fichier. Par exemple, le chemin (le chemin est le nom complet du fichier) du fichier dans lequel j'écris est /home/dcarrena/Les_entrées_sorties_en_java.docx.

Dans mon ordinateur, il ne peut y avoir un autre fichier ayant le même chemin, il peut y avoir d'autre fichier ayant le même nom « Les_entrées_sorties_en_java.docx ».

Bien sûr, le chemin du fichier change selon le système d'exploitation. Par exemple sous Linux, tous les fichiers contiennent le répertoire *root* (/) tandis que dans le système NTFS, les fichiers sont stockés dans des volumes (identifiés par des lettres alphabétiques). Les volumes contiennent des fichiers et des répertoires. Traditionnellement sous Windows, un fichier possède une extension («. » et trois lettres) qui permet d'identifier le type de fichier. Dans le deux systèmes toutes les informations permettant d'identifier le fichier se trouvent dans la chaine de caractères représentant le chemin. Par exemples :

- Linux: /home/dcarrena/mon_fichier: Dans le répertoire root (/), il y a un répertoire /home/, dans lequel se trouve le répertoire /home/dcarrena/ ("dcarrena"), dans lequel nous avons le fichier /home/dcarrena/mon_fichier.
- Windows 7: C:\Mes documents\ficher.txt: Dans le volume C, il y a un répertoire C:\Mes documents\, dans lequel se trouve le fichier C:\Mes documents\ficher.txt, qui a une vielle extension de trois lettres propre à Windows ("txt").

Il est important de noter aussi que chaque système d'exploitation possède des restrictions sur la nomenclature du chemin. Par exemple sur le système de fichier FAT (utilisé par le système MS-DOS) ne fait pas la différence entre le majuscule et le minuscule (il est insensible à la casse), ainsi les fichiers C:\FiChEr.TXT and C:\ficher.txt sont les mêmes. Toujours sur FAT, le nom du fichier ne doit pas dépasser 8 caractères et ne doit pas avoir de caractères spéciaux. Toutes ces restrictions ont été corrigées dans les nouveaux systèmes de fichiers.

1. Chemins Relatifs et chemins absolus

Jusqu'à présent nous avons vu des exemples de chemins absolus, ils identifient sans ambiguïté un fichier et son emplacement.

Un chemin relatif donne le chemin du fichier à partir du répertoire courant.

Exemples:

- Si le repertoire courant est /home/dcarrena alors le chemin relative mon_fichier identifie le fichier /home/dcarena/mon_fichier.
- Si le répertoire courant est /home/dcarrena/ alors le chemin relatif mon_projet/mon_fichier identifie the file /home/dcarrena/mon_projet/mon_fichier.

Le chemin relatif ne commence pas par la racine du système de fichier (/) pour Linux ou le nom du volume sous Windows tel que c:\

2. Chemins en Java

L'interface *java.nio.file.Path* représente le chemin et une classe implémentant cette interface peut être utilisée pour localiser un fichier dans le système de fichier.

La méthode la plus simple pour créer un objet qui implémente une telle interface est d'utiliser la classe *java.nio.file.Paths.* Elle possède une méthode statique qui retourne les objets Path basés sur la représentation en chaine de caractère (String) du chemin.

Par exemple: Path p = Paths.get("/home/dcarrena/mon_fichier");

Pour créer ces objets, il n'est pas obligatoire que les fichiers correspondant existent sur le disque dur. La représentation et la gestion des chemins en Java ne sont pas liées à l'existence des fichiers ou répertoires dans le système de fichier.

L'interface Path déclare plusieurs méthodes pour gérer les chemins telles que la méthode pour avoir le nom du fichier, pour retourner le répertoire du fichier, pour retrouver le chemin relatif, etc.

Il faut garder en tête que la gestion des chemins n'a rien à voir avec la gestion du contenu des fichiers. Par exemple, la modification du contenu d'un fichier n'est pas liée à son emplacement dans le système de fichier.

3. Exemple d'utilisation

```
1
2   import java .nio. file. Path;
3   import java .nio. file. Paths ;
4
5   class ExemplePath {
6       public static void main( String args []) {
7           Path path = Paths.get("C:\\Users\\dcarrena\\mon_ficher") ;
8           System .out. println("Le Chemin = " + path );
9           System .out. println(" Est-il absolu ? = " + path. isAbsolute());
10          System .out. println(" Le nom du fichier = " + path. getFileName());
11          System .out. println(" Le nom du repertoire parent = " + path. getParent());
12          System .out. println(" uri = "+ path. toUri ());
13      }
14  }
```

```
D:\>javac ExemplePath.java

D:\>java ExemplePath
Le Chemin = C:\Users\dcarrena\mon_ficher
 Est-il absolu ? = true
 Le nom du fichier = mon_ficher
 Le nom du repertoire parent = C:\Users\dcarrena
 uri = file:///C:/Users/dcarrena/mon_ficher

D:\>
```

III. Fichiers

La classe *java.nio.fileFile* permet de gérer les fichiers sur le disque en Java. Cette classe possède une méthode statique pour gérer les fichiers, permettant de créer, de supprimer fichiers et répertoires, de vérifier si le fichier existe sur le disque, de vérifier les permissions (lecture, écriture..), de déplacer les fichiers, de lire et d'écrire le contenu des fichiers.

Voyons comment faire les exemples opérations ci-dessus à partir de quelques exemples.

1. Vérification des permissions

```
1   import java .nio. file. Path;
2   import java .nio. file. Paths ;
3   import java .nio. file. Files ;
4
5   class VerificationExistence {
6   public static void main( String args []) {
7           Path path = Paths.get(".\\VerificationExistence.java") ;
8           System .out. println(" Chemin = " + path);
9           System .out. println(" Existe ? = " + Files . exists (path ));
10          System .out. println(" Lecture ? = " + Files . isReadable( path));
11          System .out. println(" Ecriture = " + Files . isWritable( path));
12          System .out. println(" Executable = " + Files . isExecutable( path));
13
14          path = Paths .get ("\\ce_ficher_n_existe_pas");
15          System .out. println(" Chemin = " + path);
16          System .out. println(" Existe ? = " + Files . exists (path ));
17          System .out. println(" Lecture ? = " + Files . isReadable( path));
18          System .out. println(" Ecriture = " + Files . isWritable( path));
19          System .out. println(" Executable = " + Files . isExecutable( path));
20      }
21  }
```

```
D:\>javac VerificationExistence.java

D:\>java VerificationExistence
 Chemin = .\VerificationExistence.java
 Existe ? = true
 Lecture ? = true
 Ecriture = true
 Executable = true
 Chemin = \ce_ficher_n_existe_pas
 Existe ? = false
 Lecture ? = false
 Ecriture = false
 Executable = false

D:\>
```

2. Création et suppression de fichiers

```java
import java .nio. file. Path;
import java .nio. file. Paths ;
import java .nio. file. Files ;
import java .io. IOException;

// Creer un nouveau fichier ou le supprimer s'il existe deja
class CreationOuSuppression {

    private static void usage () {
        System.err.println(" java CreationOuSuppression <fichier >");
        System.err.println(" L'argument  <fichier > est obligatoire ");
        System.exit (1);
    }

    public static void main( String args []) {
        if ( args.length != 1)
           usage();
        Path path = Paths.get( args[0]) ;
        try {
            if ( Files.exists(path))
            Files.delete( path);
            else
            Files.createFile( path);
        } catch ( IOException e) {
            System.err.println(e);
            System.exit(1);
        }
    }
}
```

```
D:\>javac CreationOuSuppression.java

D:\>java CreationOuSuppression d:\\Test.txt

D:\>java CreationOuSuppression d:\\Test.txt

D:\>java CreationOuSuppression
 java CreationOuSuppression <fichier >
 L'argument  <fichier > est obligatoire

D:\>
```

IV. Lecture d'un fichier

Il existe plusieurs manières pour lire un fichier en Java. Un petit fichier peut être lu en une seule invocation et stocké dans un tableau d'octets ou sur une chaine de caractères.

Pour des fichiers un peu plus lourds, ce ne serait plus efficace de lire un fichier par une seule invocation à cause de l'utilisation des ressources mémoires, il serait plus préférable de lire ces types de fichiers par bloc de données et de traiter chacun bloc avant lire un autre. Cela peut être fait par l'utilisation des tampons, qui permettre à leur tour d'utiliser de façon efficiente le disque dur.

1. Caractère (charsets)

Le « charset » est une association entre les nombres et les caractères. Le but du « charset » est d'assigner un nombre à chaque lettre pour permettre le stockage des lettres en utilisant leur équivalent en chiffre et de les récupérer en faisant la translation chiffre-lettre.

Le « charset » est nécessaire parce que le disque dur ne stocke que des données binaires et pas de lettres. A partir de là, si on veut écrire un fichier texte sur le disque dur, on doit transformer chaque lettre en chiffre et stocker les chiffres à la place des lettres. Dans le passé le plus populaire des « charsets » était l'ASCII.

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL	32	SPACE	64	@	96	‘
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	’	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL
Dec	Char	Dec	Char	Dec	Char	Dec	Char

Il est à noter que l’ASCII ne comporte pas les caractères accentués. Il existe plusieurs « charsets », par exemple dans un passé récent le « charset » ISO-8859-1 était le plus utilisé dans le monde francophone, espagnol mais à ce jour on utilise le « charset » UTF-8.

Quand on stocke un fichier texte sur le disque sans préciser le « charset », la relecture du même fichier pourrait retourner d’autres caractères si un autre « charset » est utilisé lors de la lecture. La meilleure solution pour ce problème de «charset », on utilise le « charset » de Java.

La classe abstraite *java.nio.charset.Charset* est utilisé en Java pour gérer les « charset » qu’on veut utiliser dans les programmes. Comme elle est abstrait ; on ne peut pas la instancier directement mais elle permet de nommer les « charset » à utiliser dans le programme.

```java
import java.nio.charset.Charset ;

class ExempleCharset {
    public static void main( String args []) {
        // Trouver le charset par defaut
        System.out.println (" Charset par defaut= " + Charset . defaultCharset());

        // Utiliser Latin1 pour les e/s fichier au lieu du charset par defaut
        System.setProperty(" file.encoding ", "ISO-8859-1") ;
        System.out.println (" Encodage du fichier = " + System . getProperty(" file.encoding "));

         // Exempkle d'utilisation directe dóbjets charset
        Charset ascii = Charset.forName ("US-ASCII");
        System.out.println (" Charset standard des autres systemes = " + ascii );
    }
}
```

```
D:\>javac ExempleCharset.java

D:\>java ExempleCharset
 Charset par defaut= windows-1252
 Encodage du fichier = ISO-8859-1
 Charset standard des autres systemes = US-ASCII

D:\>
```

2. Lecture d'un tableau d'octets

Ceci est assez simple, on nomme un fichier et on met son contenu dans un tableau d'octets. Bien que cela peut résulter d'une utilisation intense de la mémoire.

De toutes les façons cette méthode de lecture des fichiers est très élégante pour les petits fichiers (binaire un texte).

```java
import java.nio.file.Path;
import java.nio.file.Paths ;
import java.nio.file.Files ;
import java.io.IOException;

 // affiche le contenu du fichier en utilisant un tableau d'octets
class Cat1 {
    private static void usage() {
        System.err.println(" java Cat1 <fichier >");
        System.err.println(" Le <fichier > en argument est obligatoire");
        System.exit(1);
    }

    public static void main( String args []) {
        if( args.length != 1)
        usage();
        Path path = Paths.get(args [0]);
        try {
        byte [] content = Files.readAllBytes( path);
        for(int i=0; i< content.length ; i++)
        System.out.print((char)content[i]);
        } catch( IOException e) {
            System.err.println(" ERREUR : " + e);
            System.exit (1);
        }
    }
}
```

```
D:\>javac Cat1.java

D:\>java Cat1 d:\\test.txt
Voici le contenu du fichier d:\t\test.txt
D:\>
```

Le programme suivant compte le nombre de lignes d'un fichier texte.

```java
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;

// Compte le nombre de lignes
class CompterLignes1 {
private final static char WINDOWS_NEWLINE = '\n';
private final static char WINDOWS_LINEFEED = '\n';

 private static void usage () {
     System.err.println (" java CompterLignes1 <file >");
     System.err.println (" Le <fichier > en argument est obligatoire");
     System.exit(1);
 }

 public static void main( String args []) {
         if ( args.length != 1)
         usage ();

         Path path = Paths.get( args [0]) ;
         long count = 0;
         try {
             byte [] content = Files.readAllBytes( path);
             for (int i=0; i< content.length ; i ++)
             if (( char) content [i] == WINDOWS_NEWLINE && (char) content [i] == WINDOWS_LINEFEED)
                count ++;
         } catch (IOException e) {
             System.err.println (" ERREUR : " + e);
             System.exit(1);
         }
         System.out.println( count );
     }
 }
```

```
D:\>javac CompterLignes1.java

D:\>java CompterLignes1 d:\\test.txt
1

D:\>
```

La méthode *java.nio.Files.readAllLines()* peut être très utile pour traiter du texte quelques fois.

3. Tampons

Un tampon est une structure de données qui permet de gérer de bloc de données d'une collection plus large de données.

Les tampons sont utilisés pour éviter de stocker de grande quantité de données en mémoire et pour rendre les applications beaucoup plus performante puis que le programme n'aura plus à s'occuper du meilleur moment pour écrire sur le disque. Le tampon s'occupe de cette gestion.

4. Lecture d'un tampon

La classe *java.io.BufferReader* est utilisée pour lire les fichiers texte et de leur traitement. Il permet de lire de façon plus efficace les caractères, les tableaux ou des lignes entières en chaîne de caractères.

Chaque lecture à partir de la classe *BufferReader* va retourner en ordre les données du fichier correspondant, la classe *BufferReader* connait la quantité de données déjà lue.

La méthode *readLine()* va lire toute une ligne de texte et le stocke dans une chaîne de caractères.

Exemple :

```java
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;
import java.nio.charset.Charset;
import java.io.BufferedReader;

 // Affiche le contenu d'un fihier en utilisant BufferedReader
class Cat2 {
    private static void usage () {
        System.err.println(" java Cat2 <fichier >");
        System.err.println(" Le <fichier > en argument est obligatoire");
        System.exit (1);
    }
    public static void main( String args []) {
        if ( args.length != 1)
        usage ();

        Path path = Paths.get(args [0]);
        try {
            BufferedReader reader = Files.newBufferedReader(path , Charset.defaultCharset());
            String line ;
            while ( (line = reader.readLine ()) != null )
               System.out.println(line);
            reader.close ();
        } catch ( IOException e) {
            System.err.println(" ERREUR : " + e);
            System.exit (1);
        }
    }
}
```

Ce programme est beaucoup plus performant que la classe Cat1, puisqu'on utilise les tampons maintenant.

```
D:\>javac Cat2.java

D:\>java Cat2 d:\\test.txt
Voici le contenu du fichier d:\t\test.txt
```

```java
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;
import java.nio.charset.Charset;
import java.io.BufferedReader;

// Compte le nombre de lignes d'un fichier
class CompterLignes2 {
    private static void usage () {
        System.err.println(" java CompterLignes2 <fichier >");
        System.err.println(" Le <fichier > en argument est obligatoire");
        System.exit (1);
    }

    public static void main( String args []) {
        if ( args.length != 1)
        usage ();

        Path path = Paths.get( args [0]) ;
        long count = 0;
        try {
            BufferedReader reader = Files.newBufferedReader(path , Charset.defaultCharset());
            while( reader.readLine() != null )
            count ++;
            reader.close();
        } catch ( IOException e) {
            System.err.println(" ERREUR : " + e);
            System.exit(1);
        }
        System.out.println( count );
    }
}
```

```
D:\>
D:\>
D:\>javac CompterLignes2.java

D:\>java CompterLignes2 d:\\test.txt
1

D:\>
```

V. Ecrire dans un fichier

1. Modes d'accès et les paramètres d'ouverture

Lorsqu'on écrit dans un fichier en Java, on peut appliquer des restrictions sur le fichier. Les systèmes d'exploitation apportent plus de convivialité pour la gestion des permissions.

Par exemple, si un utilisateur possède les droits de lecture et d'écriture sur un fichier et notre application Java voudrait écrire dans ce fichier, on peut l'ouvrir en lecture seule ce qui restreint comment l'application utilise le fichier.

Pour cette raison et d'autres fonctionnalités, il existe en Java à travers le paramètre ***OpenOptions*** des méthodes d'accès fichier. La manière la plus simple d'utiliser ce paramètre est d'utiliser l'énumération ***StandardOpenOptions*** qui a ces différentes valeurs :

- WRITE : permet d'écrire dans le fichier.
- APPEND : commence par écrire à la fin du fichier (en gardant le contenu précédent).
- CREATE_NEW : crée un nouveau fichier ou déclenche une exception au cas le fichier n'existe pas.
- CREATE : crée un nouveau fichier ou ouvre un fichier existant.
- TRUNCATE_EXISTING : si le fichier existe et a du contenu, il commence à écrire au début du fichier ; écrasant l'ancien contenu et supprime le reste de l'ancien contenu.

Les méthodes suivantes utilisent ces modes d'accès, on peut voir dans la documentation ce que chaque méthode fait lorsque les paramètres d'accès ne sont pas explicités lors de l'invocation de la méthode.

2. Ecrire d'un tableau d'octets

Cet exemple montre la méthode la plus simple pour écrire dans un fichier. On utilise la méthode *java.nio.file.Files.write().*

```java
import java.nio.file.Path;
import java.nio.file.Paths ;
import java.nio.file.Files ;
import java.io. IOException;
import java.nio.file.StandardOpenOption;

// Copier un fichier
class Cp1 {
private static void usage () {
        System.err.println (" java Cp1 <fichier en entree > < fichier en sortie >");
        System.err.println(" Les <fichier en entree > et < fichier en sortie > en argument sont obligatoires");
        System.exit (1);
    }

    public static void main( String args []) {
        if ( args.length != 2)
        usage ();

        Path inputFile = Paths.get(args [0]);
        Path outputFile = Paths.get(args [1]);

        try {
            byte [] contents = Files.readAllBytes( inputFile);
            Files.write (outputFile , contents ,StandardOpenOption.WRITE ,StandardOpenOption.CREATE ,StandardOpenOption.TRUNCATE_EXISTING);
        } catch ( IOException e) {
            System.err. println (" ERREUR : " + e);
            System.exit (1);
        }
    }
}
```

```
D:\>javac Cp1.java

D:\>java Cp1 d:\\test.txt d:\\test2.txt

D:\>
```

3. Ecrire d'un tampon

Comme pour lire un fichier, écrire dans un fichier en utilisant les tampons est beaucoup plus efficace. Le programme suivant copie des fichiers en utilisant des tampons pour lire et écrire ligne par ligne

```java
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io. IOException;
import java.nio.charset.Charset ;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.nio.file.StandardOpenOption;
 // Copie de fichier
class Cp2 {
   private static void usage () {
       System.err.println (" java Cp1 <fichier en entree > < fichier en sortie >");
       System.err.println(" Les <fichier en entree > et < fichier en sortie > en argument sont obligatoires");
       System .exit (1);
   }
   public static void main( String args []) {
        if ( args.length != 2)
        usage ();
        Path input = Paths.get( args [0]) ;
        Path output = Paths.get( args [1]) ;
        try {
            BufferedReader inputReader =
            Files.newBufferedReader(input,Charset.defaultCharset());
            BufferedWriter outputWriter = Files.newBufferedWriter(output,Charset.defaultCharset(),StandardOpenOption.WRITE,
            StandardOpenOption.CREATE,StandardOpenOption.TRUNCATE_EXISTING);

            String line;
            while ( ( line = inputReader.readLine()) != null ) {
                outputWriter.write(line , 0, line. length ());
                outputWriter.newLine() ;
            }
            inputReader.close();
            outputWriter.close();
        } catch ( IOException e){
            System.err.println(" ERREUR : " + e);
            System.exit (1);
        }
   }
}
```

```
D:\>javac Cp2.java

D:\>java Cp1 d:\\test.txt d:\\test3.txt

D:\>
```

4. Exemples

Ce programme Java lit un fichier; ignore les lignes ayant des caractères en minuscule et affiche le résultat sur la sortie standard.

```
import java.nio.file. Path;
import java.nio.file. Paths ;
import java.nio.file. Files ;
import java.io.IOException;
import java.nio.charset.Charset ;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.nio.file.StandardOpenOption;
 // Copie ligne en  majuscule
class CopieMajuscule {
    private static void usage () {
        System.err.println (" java CopieMajuscule <fichier en entree > < fichier en sortie >");
        System.err.println(" Les <fichier en entree > et < fichier en sortie > en argument sont obligatoires");
        System .exit (1);
    }
    public static void main( String args []) {
        if ( args.length != 2)
        usage ();
        Path input = Paths.get( args [0]) ;
        Path output = Paths.get( args [1]) ;
        try {
           BufferedReader inputReader =
           Files.newBufferedReader(input , Charset.defaultCharset());
           BufferedWriter outputWriter = Files.newBufferedWriter(output,Charset.defaultCharset(),StandardOpenOption.WRITE,StandardOpenOption.CREATE,StandardOpenOption.TRUNCATE_EXISTING);
           String line;
           while ( ( line = inputReader.readLine ()) != null ) {
                if (line.equals( line.toUpperCase())) {
                    outputWriter.write(line , 0, line.length () );
                    outputWriter.newLine ();
                }
           }
            inputReader.close ();
            outputWriter.close ();
        } catch ( IOException e) {
        System.err.println (" ERREUR : " + e);
        System.exit (1);
        }
    }
}
```

Ce programme affiche les lignes qui vérifient une certaine expression régulière.

```
1   import java.nio.file.Path;
2   import java.nio.file.Paths ;
3   import java.nio.file.Files ;
4   import java.io.IOException;
5   import java.nio.charset.Charset ;
6   import java.io.BufferedReader;
7
8   // Recherche de texte dans un fichier
9   class Grep {
10      private static void usage () {
11          System.err.println (" java Grep <input file > <pattern >");
12          System.exit (1);
13      }
14
15      public static void main( String args []) {
16          if ( args.length != 2)
17          usage ();
18          Path input = Paths.get( args [0]) ;
19          try {
20          BufferedReader inputReader =
21          Files.newBufferedReader(input , Charset.defaultCharset());
22
23          String ligne;
24          long ligneNumber = 1;
25          while ( ( ligne = inputReader.readligne ()) != null ) {
26              if (ligne.contains ( args [1]) )
27              System.out.println ( ligneNumber + ": " + ligne );
28              ligneNumber++;
29          }
30          inputReader. close ();
31          } catch ( IOException e) {
32              System.err. println (" ERREUR : " + e);
33              System.exit (1);
34          }
35      }
36  }
37
```

VI. Les fichiers binaires

Un fichier binaire est un fichier qui n'est pas un fichier texte.

1- Ecriture d'un fichier binaire

```
String fileName = "out.bin";

// Writing binary file
try {
    // Creating stream
    FileOutputStream fileOs = new FileOutputStream(fileName);
    ObjectOutputStream objectOs = new ObjectOutputStream(fileOs);

    // Writing native type
    objectOs.writeInt(2048);
    objectOs.writeDouble(3.25);

    // Writing object
    objectOs.writeObject("Today");
    objectOs.writeObject(new Date());

    // Writing UTF
    objectOs.writeUTF("Java Input/Output course at Benin Esgis University - April 2016");

    // Closing streams
    objectOs.close();
    fileOs.close();

} catch (FileNotFoundException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

2- Lecture d'un fichier binaire

```
// Reading binary file
try {
    // Creating stream
    FileInputStream fileIs = new FileInputStream(fileName);
    ObjectInputStream objectIs = new ObjectInputStream(fileIs);

    // Reading native type
    int x = objectIs.readInt();
    System.out.println(x);
    double d = objectIs.readDouble();
    System.out.println(d);

    // Reading object
    String today = (String) objectIs.readObject();
    System.out.println(today);
    Date date = (Date) objectIs.readObject();
    System.out.println(date);

    //Reading UTF
    String sentence = objectIs.readUTF();
    System.out.println(sentence);

    // Closing stream
    objectIs.close();
    fileIs.close();

} catch (FileNotFoundException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```