Introduction

Ce document traite de la manipulation des fichiers à l'aide du lanagage Java. Il ne traite pas du paquetage java.io dans la mesure ou il ne traite que des flux connectés au système de fichiers. Il n'est pas exhaustif et présente un choix restreint de classes et méthodes nécessaires aux tâches usuelles d'un développeur.

De plus en plus les données sont gérées par un SGBDR. Il est de plus en plus rare de stoker de grands volumes d'information dans des fichiers. Néanmoins ils restent le support privilégié pour mémoriser des préférences, des configurations et des traces (logs). Ils sont également très utiles pour des échanges de données entre applications.

Dans ce documents vous trouverez plusieurs exemples de code. On admettra qu'il existe une classe dont l'interface sera la suivante :

```
public class Data implements Serializable{
     public Data( int numero, String nom );
     public int getNumero();
     public String getNom();
}
```

1 Les flux

Un flux (stream) représente un canal de transmission de données à partir d'une source ou vers une destination.

Il existe deux types de flux :

- Les flux d'octets ou binaires : Ils transportent des données sous forme d'octets
- Les flux de caractères : Java utilise l'Unicode pour les caractères qui sont donc codés sur 2 octets

Chaque type de flux se décline en deux spécialisations :

- Les flux en entrée : Ils permettent au programme de lire des informations.
- Les flux en sortie : Ils permettent au programme d 'écrire des informations.

Les flux peuvent être connectés à différentes sources ou destinations

- Fichier
- Tableau de caractères
- Chaine de caractère
- Tableau de bytes
- Pipeline entre deux threads

1.1 Liste des flux

	Caractère	***Octet***
Lecture	**FileReader** InputStreamReader CharArrayReader	**FileInputStream** ByteArrayInputStream PipedInputStream

	Caractère	*Octet*
	StringReader	
Ecriture	**FileWriter** OutputStreamWriter CharArrayWriter StringWriter	**FileOutputStream** ByteArrayOututStream PipedOutputStream

Dans le cadre de ce cours nous ne traiterons que des flux fichiers (notés en gras)

1.2 Ouverture d'un flux

Lors de l'ouverture d'un flux on l'associera avec une source de données. Cette source de données sera spécifiée lors de l'instanciation.

Par exemple pour ouvrir un flux permettant la lecture d'un fichier texte (flux caractère) :

```
FileWriter flux = new FileWriter("c:/exemple.txt");
```

A remarquer : Un flux est orienté, c'est à dire qu'il permet soit l'écriture soit la lecture. Il n'est pas possible d'ouvrir un fichier en lecture et en écriture. Il existe une classe qui permet ce mode mixte. Il s'agit de la classe RandomAccessFile dont nous parlerons plus loin.

1.2.1 Wrappers

Le flux n'est qu'un canal de transmission n'offrant que des services de bas niveau.

- Lecture-écriture d'un caractère ou d'un tableau de caractères.
- Lecture-écriture d'un byte ou d'un tableau de bytes.

Il est possible (et conseillé) d'utiliser un wrapper (décorateur) qui va ajouter des fonctionnalités au flux.

Dans le cadre de ce cours nous n'utiliserons que les wappers suivants:

1.2.2 Décorateurs pour flux de caractères

BufferedReader : ajoute à un Reader la mise en tampon (optimisation) et la lecture d'une ligne

Scanner : ajoute à un Reader des méthodes permettant de lire le types de base depuis un fichier texte.

PrintWriter : ajoute à un Writer des méthode permettant de formater la sortie.

1.2.3 Décorateurs pour flux de bytes

DataInputStream : Ajoute à un InputStream des méthodes permettant de lire des types primitifs de manière indépendante de la plateforme

ObjectInputStream : Ajoute à un InputStream des méthodes permettant de désérialiser des objets.

DataOutputStream : Ajoute à un OutputStream des méthodes permettant d'écrire des types primitifs de manière indépendante de la plateforme

ObjectOutputStream : Ajoute à un OutputStream des méthodes permettant de sérialiser

des objets.

1.3 Ouverture d'un fichier

Lors de l'ouverture d'un fichier il faudra associer un décorateur à un flux (un wrapper à un stream).

Par exemple pour ouvrir un fichier texte en écriture :

```
FileWriter flux = new FileWriter("c:/exemple.txt");
PrintWriter fWrite = new PrintWriter( flux );
```

Cette déclaration peut se faire en une seule opération

```
PrintWriter fWrite = new PrintWriter( new FileWriter("c:/exemple.txt") );
```

1.4 Manipulations de base

Le traitement d'un fichier suit toujours le même principe de base :

1. Ouverture du fichier
2. Manipulation (lecture ou écriture)
3. Fermeture du fichier

Ces trois étapes existent dans tous les cas. Chaque étape est susceptible de rencontrer des problèmes.

Échec lors de l'ouverture : Le fichier n'existe pas lors d'une ouverture en lecture. Le répertoire est verrouillé en lecture seule lors de l'ouverture en écriture...

Échec lors de la manipulation : Plus rien à lire (fin de fichier), données incompatibles, disque plein...

Échec lors de fermeture : Des données sont dans le tampon et ne peuvent pas être flushées.

La quasi totalité des méthodes liées aux flux lèvent des exceptions qui doivent êtres traitées. Toute les méthodes de traitement d'un flux sont susceptibles de lever au moins une exception de type java.io.IOException.

Votre algorithme doit s'assurer qu'en cas d'exception la fermeture soit effectuée. Raison pour laquelle on utilise en principe un bloc try...finally

1.5 Ouverture en écriture

Lors de l'ouverture d'un fichier en écriture un nouveau fichier est créé. S'il existe un fichier de même nom il sera écrasé et remplacé par un nouveau fichier vide. Il est de la responsabilité du développeur de vérifier l'éventuelle existence du fichier avant son ouverture.

```
fWrite = new DataOutputStream( new FileOutputStream("c:/exemple.bin") );
```

1.6 Ouverture en ajout

Certains flux permettent une ouverture en mode ajout, c'est à dire que le fichier est ouvert et le pointeur de fichier est positionné à la fin. Si le fichier n'existe pas il est crée.

```
f = new DataOutputStream( new FileOutputStream("c:/exemple.bin",true) );
```

1.7 Ouverture en lecture

Lors de l'ouverture d'un fichier en lecture le fichier doit exister. Si ce n'est pas le cas une exception de type java.io.FileNotFoundException est levée.

```
ObjectInputStream f = null;
try {
    f = new DataOutputStream( new FileInputStream("c:/exemple.bin") );
} catch (FileNotFoundException e) {
    System.out.println("Fichier non trouvé.");
    System.exit(1);
}
```

1.8 Fermeture

Après utilisation un flux doit impérativement être fermé afin de libérer les ressources allouées. Lors de la fermeture un flush (vidage du tampon est effectué.

```
f.close();
```

1.9 Écriture

Les méthodes d'écriture dépendent du wrapper utilisé.

1.10 Lecture

Les méthodes de lecture dépendent du wrapper utilisé.

AVERTISSEMENT

Pour alléger le code dans la suite de ce document nous ne traiterons que des exception essentielles. Les autres seront simplement renvoyées

2 Flux d'octets

2.1 Flux d'objets

Le langage Java propose un mécanisme de sérialisation permettant d'enregistrer ou de récupérer des objets dans un flux. Cette caractéristique est très intéressante en particulier pour les applications nécessitant une certaine persistance des objets ou pour envoyer des objets sur le réseau.

La sérialisation d'un objet est effectuée lors de l'appel de la méthode writeObject() sur un objet implémentant l'interface ObjectOutput (par exemple un objet de la classe ObjectOutputStream).

La désérialisation d'un objet est effectuée lors de l'appel de la méthode readObject() sur un objet implantant l'interface ObjectInput (par exemple un objet de la classe ObjectInputStream).

Lorsqu'un objet est sauvé, toutes les objets qui peuvent être atteint depuis cet objet sont également sauvés. En particulier si l'on sauve le premier élément d'une liste tous les éléments sont sauvés.

Tout objet n'est sauvé qu'une fois grâce à un mécanisme de cache. Il est possible de sauver une liste circulaire.

Pour qu'un objet puisse être sauvé ou récupéré sans lever une exception java.io.NotSerializableException il doit implémenter l'interface Serializable ou l'interface Externalizable.

L'interface Serializable ne contient pas de méthode. Tout objet implantant l'interface Serializable peut être enregistré ou récupéré même si une version différente de sa classe (mais compatible) est présente. Le comportement par défaut est de sauvegarder dans le flux tous les champs qui ne sont pas static, ni transient. Des informations sur la classe (nom, version), le type et le nom des champs sont également sauvegardés afin de permettre la récupération de l'objet.

Le fichier généré n'est pas modifiable. Donc pas d'accès direct ni de mode ajout.

2.1.1 Création

```
public void creation()throws IOException,ClassNotFoundException{
     ObjectOutputStream f=null;
     try{
          f = new ObjectOutputStream( new FileOutputStream("c:/exemple.bin") );
          f.writeObject( new Data(1,"un") );
          f.writeObject( new Data(2,"deux") );
          f.writeObject( new Data(3,"trois") );
          f.writeObject( new Data(4,"quatre") );
     }finally{
          f.close();
     }
}
```

2.1.2 Lecture

Lors de la lecture la méthode readObject() est susceptible de lever une exception de type java.io.EOFException lorsque la fin de fichier est rencontrée.

D'autre part elle retourne un Object qu'il faut transtyper dans le type choisi pour notre application. Lors du transtypage une exception de type java.lang.ClassNotFoundException peut être levée.

Pour ces raisons il est plus simple d'encapsuler la lecture dans une méthode.

Notre méthode traite des exceptions correctement et retourne une référence null si la fin de fichier est rencontrée.

```
public Data read( ObjectInputStream f )throws IOException{
     Data data = null;
     try{
          data=(Data) f.readObject();
     } catch( EOFException err ){
          // Ce n'est pas une erreur. On retourne null
     }catch( ClassNotFoundException err ){
          System.out.println("Erreur");
```

```
            System.exit(1);
        }
        return data;
}
```

Dès lors notre algorithme de lecture devient :

```
public void lecture()throws IOException,ClassNotFoundException{
        ObjectInputStream f = null;
        Data data = null;
        try {
            f = new ObjectInputStream( new FileInputStream("c:/test.bin") );
        } catch (FileNotFoundException e) {
            System.out.println("Fichier non trouvé.");
            System.exit(1);
        }
        try{
            System.out.println("numero nom");
            data=read(f);
            while( data!=null ){
                System.out.printf("%6d %s%n,data.getNumero(),data.getNom());
                data=read(f);
            }
        }finally{
            f.close();
        }
}
```

2.1.3 Ajout

L'ajout n'est pas possible directement dans le fichier car lors de la première écriture un header est inséré. Lors de l'ouverture en ajout on insèrerait un deuxième header...

Dès lors il faut utiliser la méthode de mutation (fichier temporaire).

```
public void ajouter( Data newData )throws IOException{
        ObjectInputStream fRead=null;
        ObjectOutputStream fWrite=null;
        Data data;
        try {
            fRead = new ObjectInputStream( new FileInputStream("c:/exemple.bin"));
        } catch ( FileNotFoundException err ) {
            System.out.println("Fichier non trouvé.");
            System.exit(1);
        }catch ( IOException err )
            System.out.println("Erreur.");
            fRead.close();
            System.exit(1);
        }
        try {
            fWrite = new ObjectOutputStream( new FileOutputStream("c:/exemple.tmp"));
```

```
     } catch (FileNotFoundException err ) {
          System.out.println("Fichier non trouvé.");
          System.exit(1);
     }catch ( IOException err )
          System.out.println("Erreur.");
          fWrite.close();
          System.exit(1);
     }
     try{
          data=read(f);
          while( data!=null ){
               f.writeObject(data);
               data=read(f);
          }
          f.writeObject( newData );
     }finally{
          fRead.close();
          fWrite.close();
     }

     File fileOri=new File("c:/exemple.bin");
     File fileTmp=new File("c:/exemple.tmp");
     fileOri.delete();
     fileTmp.renameTo(fileOri);
}
```

2.1.4 Effacement

L'effacement doit toujours faire appel à la méthode de mutation. Il n'est pas possible d'avoir des trous dans un fichier. Principe : On ne copie que les enregistrements à conserver.

```
public void effacer( int numero )throws IOException{
     ObjectInputStream fRead=null;
     ObjectOutputStream fWrite=null;
     Data data;
     try {
          fRead = new ObjectInputStream( new FileInputStream("c:/exemple.bin"));
     } catch ( FileNotFoundException err ) {
          System.out.println("Fichier non trouvé.");
          System.exit(1);
     }catch ( IOException err )
          System.out.println("Erreur.");
          fRead.close();
          System.exit(1);
     }
     try {
          fWrite = new ObjectOutputStream( new FileOutputStream("c:/exemple.tmp"));
     } catch (FileNotFoundException err ) {
          System.out.println("Fichier non trouvé.");
```

```
            System.exit(1);
        }catch ( IOException err )
            System.out.println("Erreur.");
            fWrite.close();
            System.exit(1);
        }
        try{
            data=read(f);
            while( data!=null ){
                if( data.getNumero()!=numero )
                    f.writeObject(data);
                data=read(f);
            }
            f.writeObject( newData );
        }finally{
            fRead.close();
            fWrite.close();
        }

        File fileOri=new File("c:/exemple.bin");
        File fileTmp=new File("c:/exemple.tmp");
        fileOri.delete();
        fileTmp.renameTo(fileOri);
}
```

2.1.5 Modification

Principe : On effectue une copie modifée des enregistrements.

```
public void modifier( Data newData )throws IOException{
        ObjectInputStream fRead=null;
        ObjectOutputStream fWrite=null;
        Data data;
        try {
            fRead = new ObjectInputStream( new FileInputStream("c:/exemple.bin"));
        } catch ( FileNotFoundException err ) {
            System.out.println("Fichier non trouvé.");
            System.exit(1);
        }catch ( IOException err )
            System.out.println("Erreur.");
            fRead.close();
            System.exit(1);
        }
        try {
            fWrite = new ObjectOutputStream( new FileOutputStream("c:/exemple.tmp"));
        } catch (FileNotFoundException err ) {
            System.out.println("Fichier non trouvé.");
            System.exit(1);
        }catch ( IOException err )
```

```
            System.out.println("Erreur.");
            fWrite.close();
            System.exit(1);
        }
        try{
            data=read(f);
            while( data!=null ){
                if( data.getNumero()==newData.getNumero() )
                    f.writeObject(newData);
                else
                    f.writeObject(data);
                data=read(f);
            }
            f.writeObject( newData );
        }finally{
            fRead.close();
            fWrite.close();
        }

        File fileOri=new File("c:/exemple.bin");
        File fileTmp=new File("c:/exemple.tmp");
        fileOri.delete();
        fileTmp.renameTo(fileOri);
}
```

2.2 Flux de données ou binaires

Les wrappers DataOutputStream et DataInputStream offrent la possibilité à une application de sauvegarder et lire les types de base de manière indépendante de la plateforme.

2.2.1 Méthodes d'écriture d'un DataOutputStream

void write(byte[] b, int off, int len)
void write(int b)
void writeBoolean(boolean v)
void writeByte(int v)
void writeBytes(String s)
void writeChar(int v)
void writeChars(String s)
void writeDouble(double v)
void writeFloat(float v)
void writeInt(int v)
void writeLong(long v)
void writeShort(int v)
void writeUTF(String str)

2.2.2 Méthodes de lecture d'un DataInputStream

int read(byte[] b)
int read(byte[] b, int off, int len)
boolean readBoolean()
byte readByte()
char readChar()
double readDouble()
float readFloat()
void readFully(byte[] b)
void readFully(byte[] b, int off, int len)
int readInt()
long readLong()
short readShort()
int readUnsignedByte()
int readUnsignedShort()
String readUTF()
static String readUTF(DataInput in)
int skipBytes(int n)

2.2.3 Encapsulation de l'écriture et de la lecture.

Lorsqu'on désire sauver des objets dans un flux binaire le programmeur doit lui-même écrire le code qui écrit les données membres dans le flux. De la même manière il doit écrire le code qui extrait les données membres du flux et crée un objet en mémoire. Ces deux méthodes doivent travailler de manière parfaitement symétrique. Lors de la lecture il est possible qu'une exception de type java.io.EOFException soit levée. Il est donc préférable de factoriser ce code en écrivant deux méthodes.

```
public void write( DataOutputStream f, Data data )throws IOException{
     f.writeInt( p.getNumero() );
     f.writeUTF( p.getNom() );
}

// Lit un enregistrement Retourne null si EOF
public Data read( DataInputStream f )throws IOException{
     Data data = null;
     try{
          int numero = f.readInt();
          String nom = f.readUTF();
          data = new Data(numero,nom);
     } catch( EOFException e ){
          // Ce n'est pas une erreur. On retourne null
     }
     return data;
}
```

2.2.4 Création

```
public void creation()throws IOException{
     DataOutputStream f=null;
     try{
          f = new DataOutputStream( new FileOutputStream("c:/exemple.dat") );
          write(f, new Data(1,"un") );
```

```
            write(f, new Data(2,"deux") );
            write(f, new Data(3,"trois") );
            write(f, new Data(4,"quatre") );
        }finally{
            f.close();
        }
}
```

2.2.5 Lecture

```
public void lecture()throws IOException{
        DataInputStream f = null;
        Data data = null;
        try {
            f = new DataInputStream( new FileInputStream("c:/exemple.dat") );
        } catch (FileNotFoundException e) {
            System.out.println("Fichier non trouvé.");
            System.exit(1);
        }
        try{
            System.out.println("numero nom");
            data=read(f);
            while( data!=null ){
                System.out.printf("%6d %s%n,data.getNumero(),data.getNom());
                data=read(f);
            }
        }finally{
            f.close();
        }
}
```

2.2.6 Ajout

Il est possible d'utiliser le mode append. Ceci simplifie fortement le code.

```
public void ajouter( Data newData )throws IOException{
        DataOutputStream f = null;
        try{
            f = new DataOutputStream( new FileOutputStream("c:/exemple.dat",true) );
            write(f, newData);
        }finally{
            f.close();
        }
}
```

2.2.7 Effacement - Modification

L'effacement d'un enregistrement et la modification d'un enregistrement exigent d'utiliser la méthode de mutation (fichier intermédiaire) voir Effacement et Modification. Seules les classes de wrapper doivent être modifiées.

2.3 Flux en Accès direct

Les instances de cette classent supportent aussi bien la lecture que l'écriture sur un fichier en accès direct. Un fichier en accès direct se comporte comme un grand tableau de bytes stocké dans le système de fichiers. Il offre une sorte de curseur ou index sur le tableau appelé pointeur de fichier. L'opération de lecture lit des bytes à partir de la position du pointeur et avance cecui-ci après les bytes lus. Si le fichier en accès direct est ouvert en mode lecture/écritue (read/write) les opérations d'écriture sont également disponibles. Une opération d'écriture écrit des bytes à la position du pointeur de fichier et vance celui-ci après le dernier byte écrit. Les opérations d'écriture effectuée à la fin du tableau déclenchent un agrandissement de celui-ci. Le pointeur de fichier peut être lu par la méthode getFilePointer et modifié par la méthode seek.

Si lors d'une lecture la fin du fichier est rencontrée une exception de type java.io.EOFException est levée.

Le positionnement avec la méthode seek se fait en donnant la position souhaitée exprimée en bytes à partir du début du fichier.

Le lien de la relation entre l'enregistrement à lire ou écrire et sa position est du ressort du programmeur. La solution que nous allons retenir est de stocker des enregistrements de longueur fixe afin de pouvoir calculer la position à partir d'un numéro d'enregistrement. Donc l'enregistrement 0 sera à la position 0 et l'enregistrement n sera à la position n*taille d'un enregistrement.

Si l'enregistrement que vous voulez stocker sur disque contient des String il faudra penser aux problèmes suivants.

Un String à une longueur dynamique donc la taille de votre enregistrement va varier. Avant d'enregistrer la chaine nous allons la "padder" (compléter) avec des espaces pour qu'elle ait une longueur fixe. A la lecture il faudra la trimer pour retirer ces espaces inutiles. La chaine sera stocker sous forme de suite de chars.

Taille des types primitifs

boolean	1 byte
char	2 bytes
byte	1 byte
short	2 bytes
int	4 bytes
long	8 bytes
float	4 bytes
double	8 bytes

2.3.1 Particularité

Le fichiers en accès direct ne font pas partie de l'arbre d'héritage des flux. En fait un

RandomAccesFile utilise intensivement le fonctions de bas niveau du système d'exploitation. On ne trouve donc pas de canal de transmission.

2.3.2 Méthodes

void close()	int readUnsignedByte()
FileChannel getChannel()	int readUnsignedShort()
FileDescriptor getFD()	String readUTF()
long getFilePointer()	void seek(long pos)
long length()	void setLength(long newLength)
int read()	int skipBytes(int n)void write(byte[] b)
int read(byte[] b)	void write(byte[] b, int off, int len)
int read(byte[] b, int off, int len)	void write(int b)
boolean readBoolean()	oid writeBoolean(boolean v)
byte readByte()	void writeByte(int v)
char readChar()	void writeBytes(String s)
double readDouble()	void writeChar(int v)
float readFloat()	void writeChars(String s)
void readFully(byte[] b)	void writeDouble(double v)
void readFully(byte[] b, int off, int len)	void writeFloat(float v)
int readInt()	void writeInt(int v)
String readLine()	void writeLong(long v)
long readLong()	void writeShort(int v)
short readShort()	void writeUTF(String str)

2.3.3 Ouverture

Lors de l'ouverture on doit spécifier si on va travailler en lecture ou en lecture/écriture. Pour ce choix on parle de mode d'ouverture. On spécifie le mode par une chaine de caractères qui peut valoir "r" ou "rw". D'autres valeurs sont possibles mais ne sont pas traitées dans ce document (réglage de la synchronisation).

```
RandomAccessFile f = new RandomAccessFile( "c:/test.bin","rw" );
```

2.3.4 Encapsulation de l'écriture et de la lecture.

```
public static final int TAILLENOM = 30;
public static final long TAILLE   = 4 + 2*TAILLENOM;

public void write( RandomAccessFile f, Data data )throws IOException{
     f.writeInt( p.getNumero() );
```

```
     StringBuffer str = new StringBuffer(p.getNom());
     while( str.length()<TAILLENOM )
          str.append(' ');
     f.writeChars( str.toString() );
}

public Data read( RandomAccessFile f )throws IOException{
     Data data = null;
     char[] charsNom =new char[TAILLENOM];

     try{
          int numero = f.readInt();
          for( int i=0;i<TAILLENOM;i++ )
               charsNom[i] = f.readChar();
          String nom = new String(charsNom);
          data = new data(numero,nom.trim());
     } catch( EOFException e ){
          // Ce n'est pas une erreur. On retourne null
     }
     return data;
}
```

2.3.5 Création

```
public void creation()throws IOException{
     RandomAccessFile f=null;
     try{
          f = new RandomAccessFile( "c:/exemple.dat","rw") );
          write(f, new Data(1,"un") );
          write(f, new Data(2,"deux") );
          write(f, new Data(3,"trois") );
          write(f, new Data(4,"quatre") );
     }finally{
          f.close();
     }
}
```

2.3.6 Lecture

```
// Lit un enregistrement à la position pos qui exprime le numéro d'enregistrement.
// Retourne null si l'enregistrement n'existe pas
public Data read( long pos )throws IOException{
     Data data = null;
     RandomAccessFile f = null;
     try {
          f = new RandomAccessFile( "c:/exemple.dat","r") );
     } catch (FileNotFoundException e) {
          System.out.println("Fichier non trouvé.");
          System.exit(1);
```

```
        }

        try{
            f.seek(pos*TAILLE);
            data = read(f);
        }
        finally{
            f.close();
        }
        return data;
}
```

Un fichier en accès direct permet également un parcours séquentiel.

```
public void lecture()throws IOException{
        RandomAccessFile f = null;
        Data data = null;
        try {
            f = new RandomAccessFile( "c:/exemple.dat","r") );
        } catch (FileNotFoundException e) {
            System.out.println("Fichier non trouvé.");
            System.exit(1);
        }
        try{
            System.out.println("numero nom");
            data=read(f);
            while( data!=null ){
                System.out.printf("%6d %s%n,data.getNumero(),data.getNom());
                data=read(f);
            }
        }finally{
            f.close();
        }
}
```

2.3.7 Ajout

```
public void ajouter( Data newData )throws IOException{
        RandomAccessFile f = null;
        try {
            f = new RandomAccessFile( "c:/test.bin","rw" );
            f.seek(f.length()/TAILLE);
            write(f,newData);
        }finally{
            f.close();
        }
}
```

2.3.8 Modification

```
public void modifier( int pos, Data newData ) throws IOException{
     RandomAccessFile f = null;
     try {
          f = new RandomAccessFile( "c:/exemple.dat","rw") );
     } catch (FileNotFoundException e) {
          System.out.println("Fichier non trouvé.");
          System.exit(1);
     }
     try{
          f.seek(pos*TAILLE);
          write(f,newData);
     }finally{
          f.close();
     }
}
```

2.3.9 Effacement

L'effacement d'un enregistrement exige d'utiliser la méthode de mutation (fichier intermédiaire) voir Effacement. Dans ce cas il nous faudra travailler avec deux RandomAccesFile.

3 Flux de caractères

Pour manipuler des fichiers texte nous allons utiliser les flux FileWriter et FileReader. Ces classes sont spécialisées dans la transformation (encodage) des caractères.

3.1 Flux en écriture

Afin de simplifier notre travail nous n'allons utiliser qu'un wrapper pour l'écriture dans un fichier texte. Nous utiliserons des instances de la classe PrintWriter qui nous offre une interface très riche

3.1.1 Méthodes

Voici une liste abrégée des méthodes à votre disposition

void close()	void println(boolean x)
void flush()	void println(char x)
void print(boolean b)	void println(char[] x)
void print(char c)	void println(double x)
void print(char[] s)	void println(float x)
void print(double d)	void println(int x)
void print(float f)	void println(long x)

void print(int i)	void println(Object x)
void print(long l)	void println(String x)
void print(Object obj)	void write(char[] buf)
void print(String s)	void write(char[] buf, int off, int len)
PrintWriter printf(Locale l, String format, Object... args)	void write(int c)
PrintWriter printf(String format, Object... args)	void write(String s)
void println()	void write(String s, int off, int len)

La méthode printf (apparue depuis la version 1.5) est très puissante mais nécessite un apprentissage. Elle s'inspire de la fonction homonyme du langage C.

3.1.2 Création d'un fichier texte

```
public static void creation() throws IOException{
     PrintWriter f=null;
     try{
          f = new PrintWriter( new FileWriter("c:/exemple.txt") );

          f.println("1 un");
          f.println("2 deux");
          f.println("3 trois");
          f.println("4 quatre");
          f.println("5 cinq");

     }finally{
          f.close();
     }
}
```

3.1.3 Ajout

Si on lit la documentation on peut voir qu'il est possible de fournir directement le nom du fichier à ouvrir au constructeur de PrintWriter. C'est confortable mais on se coupe alors de la possibilité de mettre le flux en mode append.

```
public void ajouter( Data newData )throws IOException{
     PrintWriter f = null;
     try {
          f = new PrintWriter( new FileWriter("c:/exemple.txt",true) );
     } catch (FileNotFoundException e) {
          System.out.println("Fichier non trouvé.");
          System.exit(1);
     }
     try {
          f.println( newData.getNumero() + newData.getNom() );
     }finally{
```

```
            f.close();
      }
}
```

3.1.4 Effacement – Modification

L'effacement d'un enregistrement et la modification d'un enregistrement exigent d'utiliser la méthode de mutation (fichier intermédiaire) voir Effacement et Modification. Les classes de flux et de wrapper doivent être modifiées.

3.2 Flux en lecture

Il existe différentes manières de lire un fichier texte mais les plus utilisées sont la lecture par ligne et la lecture formatée.

3.2.1 Lecture ligne par ligne

Cette méthode consiste à lire le fichier ligne après ligne et a faire du traitement de chaîne pour extraire les données du string reçu. La méthode split de la classe String est bien adaptée à ce type de traitement.

Pour ce type de traitement nous utiliserons un flux FileReader et un wrapper BufferedReader qui offre la méthode readln(). Cette méthode retourne null si la lecture échoue (fin de fichier).

```
public void lecture()throws IOException{
      BufferedReader f = null;
      Data data = null;
      String ligne;
      try {
            f = new BufferedReader( new FileReader("c:/exemple.txt") );
      } catch (FileNotFoundException e) {
            System.out.println("Fichier non trouvé.");
            System.exit(1);
      }
      try{
            System.out.println("numero nom");
            ligne=f.readLine();
            while( ligne!=null ){
                  String[] tokens=line.split(" ");
                  data = new Data( Integer.parseInt(tokens[0],token[1]);
                  System.out.printf("%6d %s%n,data.getNumero(),data.getNom());
                  ligne=f.readLine();
            }
      }finally{
            f.close();
      }
}
```

3.2.2 Lecture formatée

Avec cette méthode on délègue le travail de conversion à une classe spécialisé. La classe Scanner est un wrapper qui entoure un FileReader.

Le nombre de méthodes de cette classe est très important (55). Je vous invite donc à lire son API dans la documentation de Sun.

A nouveau les méthodes de cette classe lèvent des exception sur la fin de fichier donc nous allons encapsuler la lecture.

```
public Data read( Scanner f )throws IOException{
     Data data = null;
     try{
          int numero = f.nextInt();
          String nom = f.next();
          f.nextLine();
          data = new Data( numero,nom);
     } catch( NoSuchElementException e ){
          // Ce n'est pas une erreur. On retourne null
     }
     return data;
}


public void lecture()throws IOException{
     Scanner f = null;
     Data data = null;
     try {
          f = new Scanner( new FileReader("c:/exemple.txt") );
     } catch (FileNotFoundException e) {
          System.out.println("Fichier non trouvé.");
          System.exit(1);
     }
     try{
          System.out.println("numero nom");
          data=read(f);
          while( data!=null ){
               System.out.printf("%6d %s%n,data.getNumero(),data.getNom());
               data=read(f);
          }
     }finally{
          f.close();
     }
}
```