TD5[1]

Entrées - sorties

1 Préalables

Ce TD aborde l'étude des bibliothèques (*library*) Java pour la gestion des entrées et sorties fichier ou réseau. Ces API sont rassemblées dans les paquetages `java.io` et `java.nio`.

Dans les pages qui suivent, l'accent est mis sur la manipulation des fichiers. Cependant, les techniques développées serviront quasiment telles quelles lors de l'étude des `Sockets` (dans le prochain TD). En effet, sous des dehors légèrement effrayants et apparemment complexes, les classes d'entrées / sorties Java, et plus particulièrement les flux (*streams*), offrent un approche largement unifiée. Quel que soit le canal de communication, la manière de le manipuler est toujours la même.

2 Les fichiers

2.1 En bref

La classe `File` encapsule l'accès aux informations caractérisant un fichier ou un répertoire. Par son entremise, il est possible d'obtenir des renseignements sur un fichier ou un répertoire, d'en créer ou détruire, mais *pas* de lire ou modifier le contenu d'un fichier. Pour réaliser ces dernières manipulations, attendez la suite du menu (section 3.2) !

Un des constructeurs de `File` accepte un `String` en argument. Cette chaîne de caractères représente le chemin d'accès, complet ou relatif, au fichier ou dossier. Deux remarques sont à signaler :

- ↪ ce constructeur — comme tous les constructeurs de `File` — ne lève aucune exception.

[1] Le texte de ce TD s'inspire très largement de ceux des ateliers logiciels écrits jadis par différents collègues. Parmi les TDs que comprend ce cours, *MCD*, *NVS*, *RFS*, *SMB* et *VAK* ont contribués de manière diverses. Certains ont écrits entièrement un TD, d'autres ont contribués ou coécrits un TD, d'autres encore ont fait du travail de relecture. Je (*PBT*) laisse mon empreinte et en profite pour remercier tous ceux qui aiment être remerciés.

  L'instanciation d'un tel objet fourni donc essentiellement un descripteur de fichier (*handle*) vers la ressource en question. Il est possible de tester l'existence effective de celle-ci à l'aide de la méthode `exists();`
- ↪ la chaîne de caractères représentant le chemin d'accès à un fichier ou à un répertoire dépend du système d'exploitation sous-jacent.
  Par exemple, le caractère utilisé comme séparateur de dossier est « \ » sous MS-WINDOWS[2] et « / » sur les systèmes UNIX ou MAC. La portabilité de Java est cependant assurée par le biais des attributs publics de `File`, tel `File.separatorChar` dans le cas qui nous occupe, ou, plus généralement, grâce à la méthode `System.getProperties()` et à celles qui lui sont apparentées.

2.2 Mise en pratique

Exercice 1 Écrivez un programme Java avec un argument transmis par la ligne de commande. Cet argument est censé représenter le chemin d'accès, absolu ou relatif [3], à une ressource du système de fichier. Si aucun argument n'est fourni ou si la ressource n'existe pas, le programme l'indique et s'interrompt. Si elle existe :

- ↪ son nom (chemin d'accès) complet, son éventuel répertoire parent et sa date de dernière modification (dans un format lisible par un être humain) sont affichés ;
- ↪ sa nature (fichier, répertoire, ni l'un ni l'autre) est renseignée ;
- ↪ s'il s'agit d'un fichier, sa taille est fournie ;

avant la fin du programme.

Que se passe-t-il, sous MS-WINDOWS, si le caractère « \ » apparaît dans l'argument ? Et si on le remplace par « / » ?

Exercice 2 Écrivez un programme à un argument transmis par la ligne de commande. Cet argument est censé représenter le chemin d'accès à un répertoire. Si ce dossier n'existe pas physiquement, le programme l'indique et se termine. Dans le cas contraire, le contenu du répertoire est listé, avec affichage de

[2] Et, pas de chance !, ce caractère est un caractère d'échappement dans un `String`.

[3] Pour modifier le répertoire courant lors de l'exécution via NetBeans, adaptez : Project Properties > Run > Working Directory. Pour le faire hors NetBeans, placez-vous simplement dans le répertoire désiré et invoquez la JVM depuis cet endroit. Si la variable d'environnement `CLASSPATH` est bien définie et que vos classes compilées se trouvent au bon endroit, tout se passe bien. Remarquez que l'option de compilation pour modifier le répertoire courant de l'utilisateur (`-Duser.dir=`*`valeur`*) ou la modification de cette propriété directement au sein du code (par l'appel `System.setProperty("user.dir","`*`valeur`*`")`) semblent sans réel effet. . .

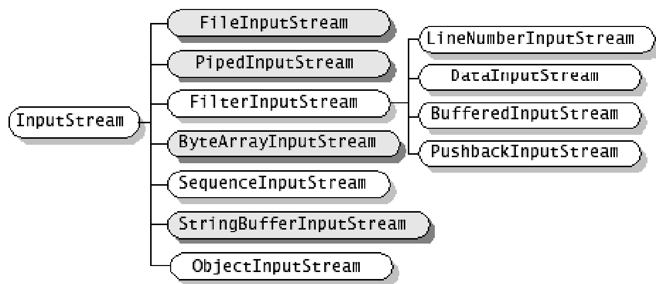

FIG. 1 – Flux binaires, c'est-à-dire d'octets, en entrée (source SUN).

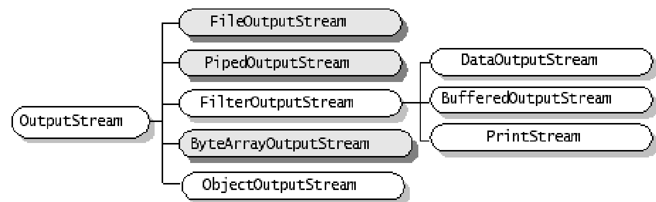

FIG. 2 – Flux binaires, c'est-à-dire d'octets, en sortie (source SUN).

la taille des fichiers. Si aucun argument n'est fourni, utilisez le répertoire courant.

Exercice 3 Écrivez un code qui liste les répertoires racine du système de fichiers de la machine hôte.

3 Les flux

3.1 Présentation générale

Vous avez probablement remarqué, en parcourant la documentation de la classe `File`, que cette classe offre des méthodes pour créer, renommer ou supprimer un fichier ou un répertoire ou encore pour modifier leurs attributs. Par contre, aucune méthode pour lire le contenu d'un fichier ou le modifier !

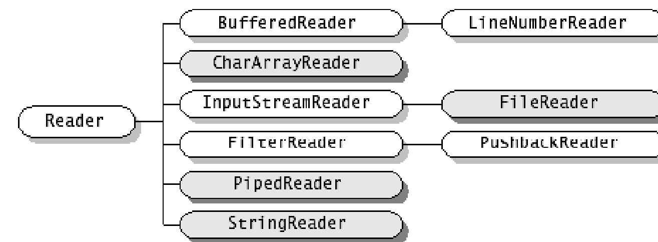

FIG. 3 – Flux texte, c'est-à-dire de caractères, en entrée (source SUN).

Pour réaliser de telles opérations, il faut ouvrir un flux [4] (*stream*) « de » ou « vers » le fichier en question. En Java, la plupart des opérations d'entrées / sorties se font via un ou des flux, que ces opérations soient une lecture clavier, un affichage à l'écran, une lecture dans un fichier ou encore l'envoi de données vers une machine distante…

Nous n'allons pas ici étudier exhaustivement l'ensemble des hiérarchies de classes de flux fournies dans le paquetage `java.io`. Celles-ci sont, en partie, représentées sur les FIG. 1-4.

Remarquons avant tout les quatre classes *abstraites*

↪ `InputStream`,
↪ `OuputStream`,
↪ `Reader` et
↪ `Writer`.

Les deux premières concernent des flux d'octets tandis que les deux dernières concernent des flux de caractères (deux octets en Java) avec support de l'UNICODE. Par ailleurs, les flux Java sont *uni-directionnels* : les flux dérivés de `InputStream` ou `Reader` sont des flux en entrée, ceux héritant de `OuputStream` ou `Writer` sont quant à eux des flux en sortie. Tous ces flux sont séquentiels.

Ils offrent tous — leurs classes dérivées instanciables en tous cas — la méthode `close()`, pour fermer le flux, c'est-à-dire libérer les ressources systèmes allouées pour la gestion du flux.

Les flux en sortie fournissent la méthode `flush()` [5]. Elle force le vidage dans le flux d'un tampon en écriture. Remarquez qu'il n'est pas garanti que la méthode `close()` d'un flux en sortie appelle `flush()`. Pour plus de sécurité, veillez donc à toujours appeler `flush()` avant de refermer un flux, sous peine

[4] Cfr. cours de langage Java de première
[5] Littéralement « tirer la chasse »

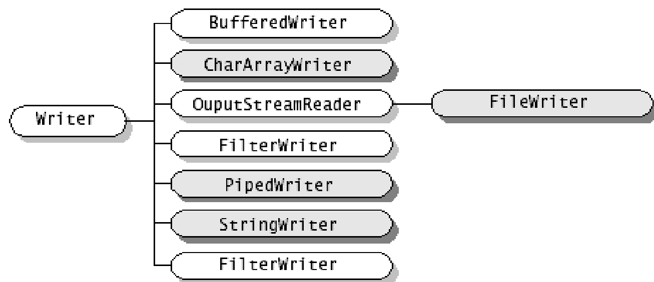

FIG. 4 – Flux texte, c'est-à-dire de caractères, en sortie (source SUN). *Attention* : la troisième classe à partir du haut est `OutputStreamWriter` et non `OutputStreamReader`; la classe `PrintWriter` pourrait également apparaître ici (sur fond blanc).

de possibles pertes de données ! D'autre part, dans le cadre d'applications réseau et d'un dialogue entre une application client et une application serveur, le recours à la méthode `flush()` est indispensable sous peine de blocages. Par exemple : un client attend la réponse d'un serveur, celui-ci l'a envoyée mais elle reste dans un *buffer* donc ne parvient pas au client...

Toute classe dérivant de `InputStream` doit implémenter la méthode `read()`. Celle-ci retourne un `int` dont l'octet de poids faible code l'octet lu dans le flux. Si la fin du fichier est atteinte, la valeur -1 est retournée. Cette méthode est bloquante.

La classe `Reader` est munie de la méthode `read()`. Celle-ci retourne un `int` dont les deux octets de poids faible codent le caractère lu dans le flux. Si la fin du fichier est atteinte, la valeur -1 est retournée. Un appel à cette méthode est bloquant.

Toute classe dérivant de `OutputStream` doit implémenter la méthode `write(int)`. Celle-ci écrit sur le flux le caractère stocké dans l'octet de poids faible de l'`int` en argument.

La classe `Writer` est munie de la méthode `write(int)`. Celle-ci écrit sur le flux le caractère stocké dans les deux octets de poids faible de l'`int` en argument.

Toutes les méthodes signalées jusqu'à présent lancent une `IOException` en cas d'erreur d'entrées / sorties.

Les FIG. 1-4 montrent des noms de classes sur fond gris, d'autres sur fond

blanc. Les classes sur fond gris sont des classes de flux permettant d'écrire ou lire vers des conteneurs de données : fichier, chaîne de caractères, etc. Ceux sur fond blanc sont des flux offrant des services supplémentaires : mise en tampon, manipulation de motifs binaires primitifs, sérialisation d'objet, etc.

Face à un problème d'entrées / sorties donné, le programmeur Java combine l'utilisation de divers flux, selon la nature du conteneur de données et les facilités d'accès à celles-ci, disponibles ou désirées. Cette combinaison se fait en *enveloppant* le flux lié au conteneur — classe sur fond gris des FIG. 1 à 4 — par un ou plusieurs flux de service — classe sur fond blanc dans ces mêmes figures. Il s'agit d'une mise en pratique du modèle de conception « *Décorateur* ». Ceci est appliqué dans la suite.

Comment choisir effectivement la combinaison de flux ? Nous n'allons pas répondre exhaustivement à cette question ici. Cependant, sur la base de quelques exemples, la manière de « faire son marché » aux flux devrait vous apparaître.

3.2 Mise en application

3.2.1 Premiers exemples : copie d'un fichier

Exemple Commençons par voir comment recopier le contenu d'un fichier dans un autre.

`$ cat CopieFichierBinaire.java`

```
package nvs.alg2ir;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

/**
 * Copie d'un fichier texte vers un autre à l'aide de
 * FileInputStream/OutputStream
 */
public class CopieFichierBinaire {

  public static void main(String[] args) {

    File inputFile = new File("original.txt");
    File outputFile = new File("copie.txt");

    try {
      FileInputStream in = new FileInputStream(inputFile) ;
      FileOutputStream out = new FileOutputStream(outputFile) ;
      int c;
```

```
      while ((c = in.read()) != -1) {
        out.write(c);
      }

      in.close();
      out.flush();
      out.close();
    } catch (FileNotFoundException fnfe) {
      System.err.println(fnfe);
    } catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

Quelques remarques :

↪ les flux utilisés ici sont des flux d'octets : `FileInputStream` et `FileOutputStream`. Le fichier original est donc copié octet par octet. Remarquez que le fichier, dans cet exemple, est un fichier texte. Pourquoi ne pas utilisez un flux de caractères ? Réponse ci-dessous ;

↪ notez la condition de boucle `while` « *à la* C » ;

↪ la gestion des exception n'est pas sans faille : un flux ouvert n'est pas refermé si un problème d'entrées / sorties se pose en cours de lecture ou d'écriture (cfr. `finally`).

Exemple Voici la copie de fichier, version flux de caractères.

`$ cat CopieFichierTexte.java`

```
package nvs.alg2ir;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

/**
 * Copie d'un fichier texte vers un autre à l'aide de FileReader/Writer
 */
public class CopieFichierTexte {

  public static void main(String[] args) {

    File inputFile = new File("original.txt");
    File outputFile = new File("copie.txt");

    FileReader in = null ;
    FileWriter out = null ;

    try {
      in = new FileReader(inputFile);
      out = new FileWriter(outputFile);
      int c;
      while ((c = in.read()) != -1) {
        out.write(c);
      }
```



```
    } catch (FileNotFoundException fnfe) {
      System.err.println(fnfe);
    } catch (IOException ioe) {
      System.err.println(ioe);
    } finally { // toujours exécuté : soit après try, soit après catch
        try {
        if (in != null ) {
          in.close();
        }
        if (out != null) {
          out.flush();
          out.close();
        }
      } catch (IOException ioe) {
        System.err.println("ioe");
      }
    } // fin du finally
  }
}
```

Quelques remarques :

↪ les flux utilisés maintenant sont des flux de caractères : `FileReader` et `FileWriter`. Cela semble cohérent puisque le fichier est un fichier texte. Cependant, un `char` en Java fait *deux* octets (UNICODE). Or, les fichiers textes sont, actuellement, constitués de caractères codés sur *un* octet ! Comment le lien entre le fichier texte — caractère d'un octet, multiples encodages possibles (ASCII étendu, ANSI, etc.) — et le flux texte — caractère sur deux octets (encodage UNICODE, UTF-16) — est-il réalisé ?

La réponse réside dans les FIG. 3 et 4. On y voit que la classe `FileReader` dérive de `InputStreamReader` et que `FileWriter` dérive de `OutputStreamWriter`. Ces deux classes mères sont les *ponts entre le monde des streams d'octets et celui des flux de caractères*. L'encodage utilisé ici, par `FileReader` et `FileWriter`, est l'encodage par défaut, celui de la machine hôte. Pour lire un fichier texte encodé selon un autre schéma d'encodage, il faut l'ouvrir via un `FileInputStream` puis utiliser les services d'un `InputStreamReader` dont on spécifie l'encodage.

Remarquez que, tout compte fait, la version de copie avec des flux binaires fait ici très bien l'affaire puisqu'on n'a pas à véritablement manipuler les données textuelles du fichier à recopier ;

↪ la gestion des exception est améliorée par l'utilisation du mot clé `finally`;

↪ les fichiers sont lu et écrit sans utiliser de tampon en lecture ou en écriture : à moins que le système d'exploitation y pallie, ce n'est vraiment pas efficace !

3.2.2 Affichage du contenu d'un fichier texte sur la sortie standard

Exemple Affichons le contenu d'un fichier texte, en *bufferisant* les accès au flux d'entrée. Au fait, quelle est la nature de `System.out` ?

```
$ cat AffichageFichierTexte.java
```

```
package nvs.alg2ir;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

/**
 * Affichage sur la sortie standard du contenu d'un fichier texte
 */
public class AffichageFichierTexte {

  public static void main(String[] args)
      throws FileNotFoundException, IOException{

    BufferedReader in ;
    int c ;

    in = new BufferedReader(new FileReader("src/nvs/alg2ir/" +
            "AffichageFichierTexte.java"));
    while ((c = in.read()) != -1) {
      System.out.print((char) c);  // notez le transtypage
    }
    in.close();
  }
}
```

Quelques remarques :

- ↪ le flux utilisé en entrée est un `BufferedReader` *enveloppant* (*wrapper*) un `FileReader`. Si on se réfère à la FIG. 3, on y remarque que `BufferedReader` y apparaît sur fond blanc : c'est donc une classe fournissant des fonctionnalités supplémentaires.
  - ˘ À qui ? Son constructeur requière un `Reader`.
  - ˘ Quel type de service ? La mise en tampon des accès au `Reader`.

  Ceci est utilisé dans l'exemple ci-dessus, avec la méthode `read()` : le contenu du fichier est amené dans un tampon dans lequel lit la méthode de lecture d'un caractère ;
- ↪ dans cet exemple, pas de `try catch`, les exceptions sont relancées par la méthode `main` ;
- ↪ un autre service rendu par un `BufferedReader` est la possibilité de réaliser une lecture ligne par ligne, celles-ci terminées par une marque de fin de ligne. Elle peut être mise en œuvre en remplaçant la boucle `while` par la suivante


```
String ligne ;
while ((ligne = in.readLine()) != null) {
  System.out.println(ligne);
}
```

  Selon la documentation, les marques de fin de ligne reconnues par `readLine()` sont `'\r'`, `'\n'` ou `"\r\n"`[6]. La fin du fichier est alors détectée par une valeur de retour `null` de la méthode `readLine()` ;
- ↪ la composition de flux, où l'un est utilisé comme argument de construction d'un autre, est typique de la gestion des flux d'entrées / sorties en Java.

3.2.3 Lecture sur l'entrée standard

L'entrée standard, `System.in`, est de type `InputStream`[7].

Exemple Voici comment lire des chaînes de caractères sur l'entrée standard, sans utiliser les classes `Scanner`, apparue avec le JDK 1.5., et `Console`, disponible dès le JDK 1.6., et les écrire sur la sortie standard et, simultanément, dans un fichier.

```
$ cat Clavier.java
```

```
package nvs.alg2ir;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;

/**
 * Lecture au clavier en mode console sans utiliser les classes
 * Scanner et Console
 */
public class Clavier {

  private static BufferedReader br = new BufferedReader(
        new InputStreamReader(System.in)) ; // System.in == InputStream

  public static String lireString() {

    String s = null ;
```

[6] Cependant, selon *Java™ Network Programming, Third Edition*, Elliotte Rusty Harold, O'Reilly, Sebastopol (CA), 2005, un retour chariot seul, \r n'est pas toujours reconnu comme marque de fin de chaîne, ce qui rend l'usage de `readLine()` hasardeux pour lire un flux ou un fichier issu d'un MAC OS 9. Ceci amène l'auteur du livre cité à déconseiller vivement l'utilisation de cette méthode lors du développement d'applications réseaux puisque, lors de la mise en réseau, il n'est pas (toujours) possible de contrôler les machines concernées

[7] Plus précisément d'une classe concrète, mais de nom inconnu, dérivant de `InputStream`.

```
    try {
      s = br.readLine() ;
    } catch (IOException ioe) {
      System.err.println("problème␣de␣lecture␣d'une␣chaîne␣:␣" + ioe);
    }
    return s ;
  }

  public static void main(String[] args) {

    PrintWriter pw = null ;

    try {
      pw = new PrintWriter(
              new BufferedWriter(
              new FileWriter("sortieFichier.txt")),true) ;  // autoflush
      String s ;

      System.out.print("Fournir␣une␣chaine␣de␣caracteres␣:␣");
      s = Clavier.lireString() ;
      while (s.length() > 0) {
        System.out.println("Chaine␣fournie␣:␣"+s);
        pw.println(s) ;
        System.out.print("Fournir␣une␣chaine␣de␣caracteres␣ou␣une␣" +
                "chaine␣vide␣pour␣finir␣:␣");
        s = Clavier.lireString() ;
      }
    } catch (IOException ioe) {
      System.err.println(ioe);
    } finally {
      if (pw != null) {
        pw.flush();
        pw.close();
      }
    } // fin du finally
  }
}
```

Quelques remarques :

- ↪ l'entrée standard `System.in` est convertie en flux de caractères par un `InputStreamReader` — avec évidemment l'encodage par défaut ! — lui-même *bufferisé* par un `BufferedReader`. En effet, on désire lire des caractères, constituant une `String` *et* on souhaite lire une chaîne de caractères d'une traite ;
- ↪ rien à signaler pour la sortie standard ;
- ↪ pour ce qui concerne la sortie fichier, on désire une sortie sous la forme d'un fichier texte, d'où le `FileWriter`. Pour des raisons de performances, on met en tampon les accès au fichier à l'aide d'un `BufferedWriter`. Pour finir, on désire écrire le plus facilement possible dans le fichier, d'où les services d'un `PrintWriter` dont on force le vidage de tampon à chaque fin de ligne[8] ;

[8] Cela n'est pas indispensable ici. Par contre, dans les applications réseau, c'est vivement conseillé pour éviter des transmissions partielles ! Notez cependant que l'utilisation de

- ↪ pour signifier la fin de la lecture, l'utilisateur est invité à fournir une chaîne vide de caractères.

Exemple Un boulot semblable à celui de l'exemple précedent mais en utilisant la classe `Scanner`.

```
$ cat TestScanner.java
```

```
package nvs.alg2ir;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner;

/**
 * Lecture au clavier en mode console en utilisant la classe Scanner
 */
public class TestScanner {

  public static void main(String[] args) {

    PrintWriter pw = null ;
    Scanner stdin = new Scanner(System.in) ;

    try {
      pw = new PrintWriter(
        new BufferedWriter(
        new FileWriter("sortieFichier.txt")),true) ;  // autoflush
      String s = null ;

      System.out.print("Fournir␣une␣chaine␣de␣caracteres␣:␣");
      while ( stdin.hasNext() ) {
        s = stdin.next() ;
        System.out.println("Chaine␣fournie␣:␣"+s);
        pw.println(s) ;
        System.out.print("Fournir␣une␣chaine␣de␣caracteres␣ou␣la␣" +
          "marque␣de␣fin␣de␣fichier␣␣pour␣finir␣:␣");
      }
    } catch (IOException ioe) {
      System.err.println(ioe);
    } finally {
      if (pw != null) {
        pw.flush();
        pw.close();
      }
    } // fin du finally
  }
}
```

Quelques questions comparant les exemples précédents :

- ↪ la fin des lectures au clavier n'est plus indiquée par une chaîne vide.

`println()` est très fortement déconseillée dans les flux d'une application réseau car la marque de fin de ligne insérée dans le flux est *celle de la machine hôte*. Elle diffère donc selon la machine exécutant le programme et ne correspond probablement pas systématiquement à celle du protocole réseau utilisé !

Pourquoi ?
↪ qu'en est-il des chaînes de caractères composées de plusieurs mots ?

Exemple À nouveau, une tâche assez semblable mais cette fois-ci en utilisant la classe `Console`[9].

`$ cat TestConsole.java`

```
package nvs.alg2ir;

import java.io.BufferedWriter;
import java.io.Console;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

/**
 * Lecture au clavier en mode console en utilisant la classe Scanner
 */
public class TestConsole {

  public static void main(String[] args) {

    PrintWriter pw = null ;
    Console console = System.console() ;

    if (console == null) {
      System.out.println("Système␣sans␣console...");
      System.exit(1);
    }

    try {
      pw = new PrintWriter(
        new BufferedWriter(
        new FileWriter("sortieFichier.txt")),true) ;  // autoflush
      String s = null ;

      console.printf("Fournir␣une␣chaine␣de␣caracteres␣:␣");
      while ( (s = console.readLine()) != null )  {
        console.printf("Chaine␣fournie␣:␣%s%n",s);
        pw.println(s) ;
        console.printf("Fournir␣une␣chaine␣de␣caracteres␣ou␣la␣" +
          "marque␣de␣fin␣de␣fichier␣␣pour␣finir␣:␣");
      }
    } catch (IOException ioe) {
      System.err.println(ioe);
    } finally {
      if (pw != null) {
        pw.flush();
        pw.close();
      }
    } // fin du finally
  }
}
```

Quelques remarques et questions :
↪ la classe `Console` est un singleton ;

[9] Vérifiez sa disponibilité en fonction du la version de JDK utilisée.

↪ notez les lectures *et* écritures via la console ;
↪ que se passe-t-il lorsqu'on exécute ce code dans NetBeans ?
↪ qu'en est-il des chaînes vides ?
↪ qu'en est-il des chaînes de caractères composées de plusieurs mots ?

3.2.4 Les fichiers binaires

Par opposition aux fichiers textes, c'est-à-dire ceux éditables dans un éditeur de texte, existent les fichiers binaires. Les informations textuelles y sont encodées comme dans les fichiers textes, à savoir qu'à chaque caractère correspond un motif binaire variant selon l'encodage. Par contre, les données numériques primitives n'y sont pas représentées sous la forme de chaînes de caractères. Elles y sont représentées par les mêmes motifs binaires que ceux qui les caractérisent en mémoire centrale[10]. Un `int`, par exemple, occupe toujours quatre octets dans un fichier binaire, l'algorithme de codage est celui du complément à deux et l'ordre de ses octets respecte le grand boutisme.

Exemple Voici un petit source. Découvrez par vous même son but et son fonctionnement !

`$ cat FichierBinaire`

```
package nvs.alg2ir;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.EOFException;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

/**
 * Création et lecture d'un fichier binaire
 */
public class FichierBinaire {

  public static void main(String[] args) {

    String nomFichier = "fruit.nvs" ;

    //
    // écriture
    //


    DataOutputStream out = null ;
```

[10] En machine virtuelle plutôt, ce qui n'est pas exactement la même chose (cf. endianisme, p. ex.).

```
    try {
      out = new DataOutputStream(
        new BufferedOutputStream(
        new FileOutputStream(nomFichier)));
    } catch (FileNotFoundException fnfe) {
      System.err.println(fnfe);
      System.exit(1) ;
    }

    double[] lesPrix = { 1.5, 1.75, 7.5, 12.0, 3.2 };
    int[] lesQuantités = { 2, 5, 1, 1, 4 };

    String[] lesFruits = { "coing", "pomme", "cerise", "groseille", "poire" } ;

    try {
      for (int i = 0; i < lesPrix.length; ++i) {
        out.writeDouble(lesPrix[i]);
        out.writeChar('\t');
        out.writeInt(lesQuantités[i]);
        out.writeChar('\t');
        out.writeChars(lesFruits[i]);
        out.writeChar('\n');  // choix explicite de la marque de fin de ligne
      }
      out.close();            // flush automatique (voir javadoc)
    } catch (IOException ioe) {
      System.err.println(ioe);
      System.exit(2);
    }

    //
    // lecture et affichage
    //
    DataInputStream in = null ;

    try {
      in = new DataInputStream(
        new BufferedInputStream(
        new FileInputStream(nomFichier)));
    } catch (FileNotFoundException fnfe) {
      System.err.println(fnfe);
      System.exit(3) ;
    }

    double prix;
    int quantité;
    StringBuffer fruit;
    double total = 0.0;

    try {
      while (true) {  // et la fin de fichier ?
        prix = in.readDouble();
        in.readChar();       // pour "passer" le "\t"
        quantité = in.readInt();
        in.readChar();       // pour "passer" le "\t"
        char chr;
        fruit = new StringBuffer(20);
        while ((chr = in.readChar()) != '\n')
          fruit.append(chr);
        System.out.printf("Tu␣as␣acheté␣%d␣kg␣de␣%s␣à␣%.2f␣\u20AC␣le␣kg\n",
          quantité,fruit,prix);
        total = total + quantité * prix;
      }
```



```
      } catch (EOFException eof) {
        // détection de la fin de fichier :
        //  pas vraiment dans l'esprit des exceptions !
      } catch (IOException ioe) {
        System.err.println(ioe);
        System.exit(4);
      }

      System.out.printf("\nTa␣note␣s'élève␣à␣%.2f␣\u20AC",total);

      try {
        in.close();
      } catch (IOException ioe) {
        System.out.println(ioe);
      }
    }
}
```

Quelques remarques :

- ↪ le fichier créé est un fichier binaire. Inutile donc d'y accéder par le biais d'un flux de caractère. Un `FileOutputStream` est parfait. Mettre les accès en tampon est toujours une bonne idée, d'où le `BufferedOutputStream`. L'écriture des données brutes binaires est largement facilitée par les services d'un `DataOutputStream`;
- ↪ pour ce qui concerne la lecture, il suffit de prendre les flux en sortie équivalents à ceux en entrée : un `DataInputStream` pour le décodage — il faut évidemment connaître la structure du fichier ! — et un `FileInputStream` pour effectivement accéder au fichier, un `BufferedInputStream` n'est pas obligatoire, mais bienvenu ;
- ↪ lors de la lecture, la fin de fichier est détectée par la capture d'une `EOFException`. Une telle utilisation d'une exception est assez choquante puisque atteindre la fin d'un fichier ne constitue pas une circonstance exceptionnelle ou particulièrement périlleuse. Cette approche se justifie par la difficulté de fournir une valeur de retour signalant la fin du fichier. Les valeurs utilisées par les méthodes de lecture d'autres flux (-1, `null`) constituent en effet des valeurs possibles de données binaires ;
- ↪ l'affichage console utilise la fonction `printf` apparue avec le JDK 1.5 ;
- ↪ la liste des codes UNICODE est disponible à l'adresse `http://unicode.org/charts/` Remarquez qu'ici on aurait pu encoder le caractère « € » immédiatement dans le source.

3.2.5 Sérialisation d'objet

Il est possible d'écrire dans un fichier ou d'envoyer sur un flux réseau un objet, brut et binaire. Cette opération s'appelle « sérialisation ». Un objet sérialisé peut ensuite être récupéré, lors d'une désérialisation. De telles opérations ne sont pas toujours triviales puisque les attributs d'un objet peuvent être eux-

même des instances de classe qui doivent également être sérialisées / désérialisées, etc.

Par défaut, une classe n'est pas sérialisable. Pour l'être, il lui suffit d'implémenter l'interface `Serializable`. Remarquez que cette interface est une interface de marquage. Aucune méthode ne fait partie du contrat ! Pour écrire un objet sur un flux, on utilise un `ObjectOutputStream`. Pour lire un objet sur un flux, la classe `ObjectInputStream` est disponible.

Exemple Voici un exemple suffisamment complexe illustrant la puissance de la sérialisation.

`$ cat Data.java`

```
package nvs.alg2ir;

import java.io.Serializable;

/**
 * Une classe sérialisable très simple...
 *
 * from Thinking in Java, 3rd ed. Revision 4.0,
 * Bruce Eckel, http://mindview.net/
 */
public class Data implements Serializable {

  private int n;
  private /*transient*/ int m ;

  public Data(int n) {
    this.n = n;
    m = 3 * n ;
  }

  public String toString() {
    return (Integer.toString(n) + Integer.toString(m));
  }
}
```

`$ cat Worm.java`

```
package nvs.alg2ir;

import java.io.Serializable;
import java.util.Random;

/**
 * Une classe sérialisable moyennement complexe (liste chaînée)
 *
 * from Thinking in Java, 3rd ed. Revision 4.0,
 * Bruce Eckel, http://mindview.net/
 */
public class Worm implements Serializable {

  private static Random rand = new Random();

  private Data[] d = { new Data(rand.nextInt(10)), new Data(rand.nextInt(10)),
          new Data(rand.nextInt(10)) };

```



```
  private Worm next;  // vaut null par défaut !

  private char c;

  // Value of i == number of segments
  public Worm(int i, char x) {
    //System.out.println("Worm constructor: " + i);
    c = x;
    if(--i > 0)
      next = new Worm(i, (char)(x + 1));
  }

  public Worm() {
    System.out.println("Default constructor");
  }

  public String toString() {
    String s = ":" + c + "(";
    for(int i = 0; i < d.length; i++) {
      s += d[i];
      if (i != d.length -1) s += " " ;
    }
    s += ")";
    if(next != null)
      s += next;      // appels récursifs...
    return s;
  }
}
```

`$ cat Test.java`

```
package nvs.alg2ir;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

/**
 * Test de sérialisation
 *
 * from Thinking in Java, 3rd ed. Revision 4.0,
 * Bruce Eckel, http://mindview.net/
 */
public class Test {

  public static void main(String[] args) {

    Worm w = new Worm(6, 'a');
    System.out.println("Objet original : w = " + w);

    try {
      ObjectOutputStream out = new ObjectOutputStream(
              new FileOutputStream("worm.out"));
      out.writeObject("Worm storage\n");
      out.writeObject(w);
      out.flush();
      out.close();

```

```
      ObjectInputStream in = new ObjectInputStream(
              new FileInputStream("worm.out"));
      System.out.println("Lecture␣du␣fichier␣:␣");
      String s = (String)in.readObject();   // notez le transtypage
      Worm w2 = (Worm)in.readObject();      // notez le transtypage
      System.out.println(s + "Objet␣désérialisé␣:␣w2␣=␣" + w2);
      in.close();
    } catch (ClassNotFoundException cnfe) {
      System.err.println(cnfe);
    } catch (FileNotFoundException fnfe) {
      System.err.println(fnfe);
    } catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

Quelques remarques :

↪ on aurait pu *bufferiser* les accès au fichier binaire ;

↪ l'attribut `m` de `Data` est précédé du mot clé `transient`, mis en commentaire. Retirez cette mise en commentaire et découvrez l'effet de ce mot réservé ;

↪ la méthode `readObject()` de `ObjectInputStream` lève une `ClassNotFoundException` lorsque la classe de l'objet sérialisé n'est pas disponible, c'est-à-dire lorsque le fichier de classe compilé (`.class`) de la classe de l'objet n'est pas dans un répertoire du `CLASSPATH`. Remarquez cependant que le type de l'objet désérialisé peut être déterminé par réflection, par un appel de `getClass()`, par exemple.

3.3 Exercices

Exercice 4 Ajoutez au code de l'Exemple 3.2.3 la méthode `lireInt()` permettant de lire un `int` sur l'entrée standard et la méthode `lireDouble()` pour y lire un `double`.

Exercice 5 Écrivez un petit programme à interface graphique, dans le genre de celle montrée à la FIG. 5, permettant d'éditer, sans possibilité de modification, le contenu d'un fichier texte choisi par l'utilisateur.

Pour le choix du fichier, le composant `JFileChooser` devrait vous combler. Un `JTextArea` dans un `JScrollPane` devraient faire l'affaire pour ce qui concerne l'affichage du contenu du fichier.

Exercice 6 Écrivez un code affichant sur la sortie standard et copiant dans un fichier le contenu d'une page WEB, comme `http://esi.magicrhesus.be/`, par exemple.

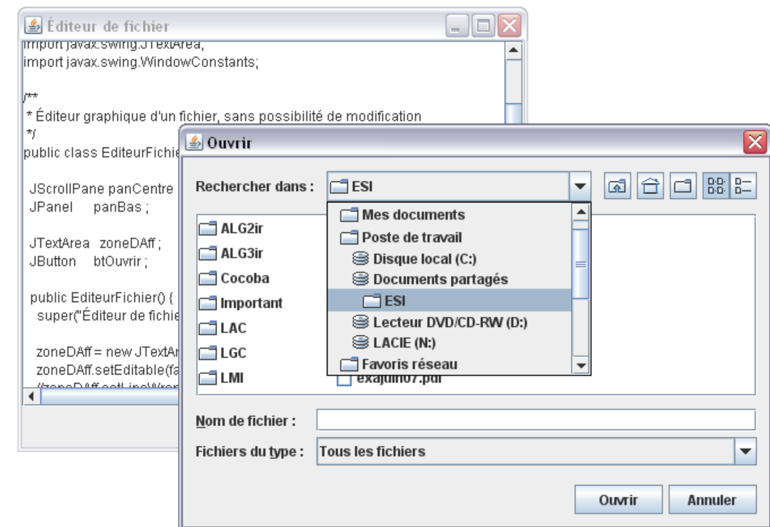

FIG. 5 – Interface graphique possible d'une solution à l'Ex 3.3
.

La classe `java.net.URL` et sa méthode `openStream()` vous seront d'une grande aide.

Exercice 7 Dans votre exercice du *poulailler*, ajouter la génération d'un journal (*log*) ; un fichier reprenant des informations du style *La poule « id » pond*.

Chaque *thread* écrit dans le même fichier journal.

Table des matières

1 Préalables **1**

2 Les fichiers **1**
2.1 En bref 1
2.2 Mise en pratique 2

3 Les flux **3**
3.1 Présentation générale 3
3.2 Mise en application 6
3.2.1 Premiers exemples : copie d'un fichier 6
3.2.2 Affichage du contenu d'un fichier texte sur la sortie standard 9
3.2.3 Lecture sur l'entrée standard 10
3.2.4 Les fichiers binaires 14
3.2.5 Sérialisation d'objet 16
3.3 Exercices 19