Part V

Les streams

Java I/O package

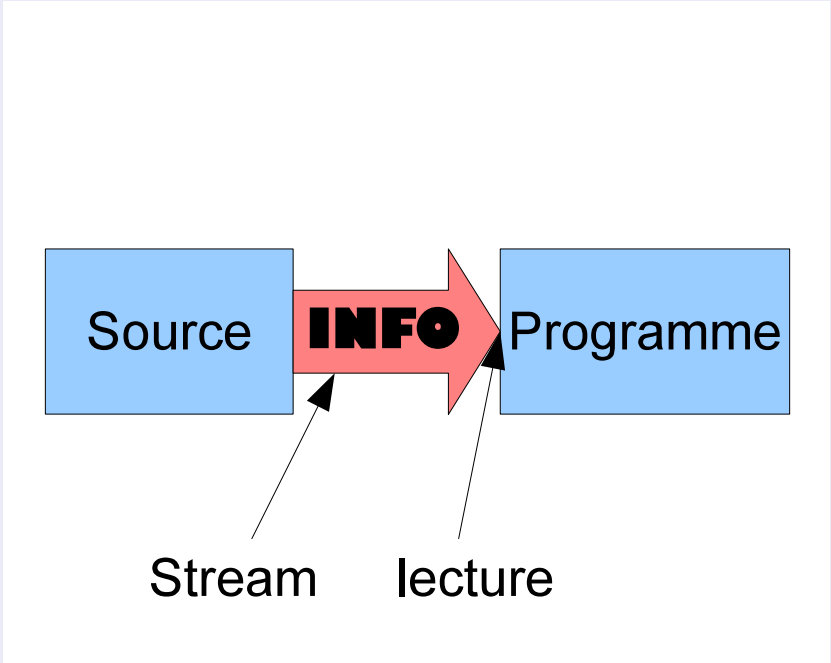
Definition (package java.io)

- Le package Java I/O (java.io) permet d'accéder aux ressources système sans que le programmeur n'ait à connaître les détails du Système.
- Il permet de traiter les entrées et sorties (I/O) en terme de "streams" (ou flux comme en C++).
- Les streams sont des objets pouvant être :
  - ouverts (à leur création)
  - fermés (en utilisant la méthode `close`)

Les streams : entrées

Definition (Stream)

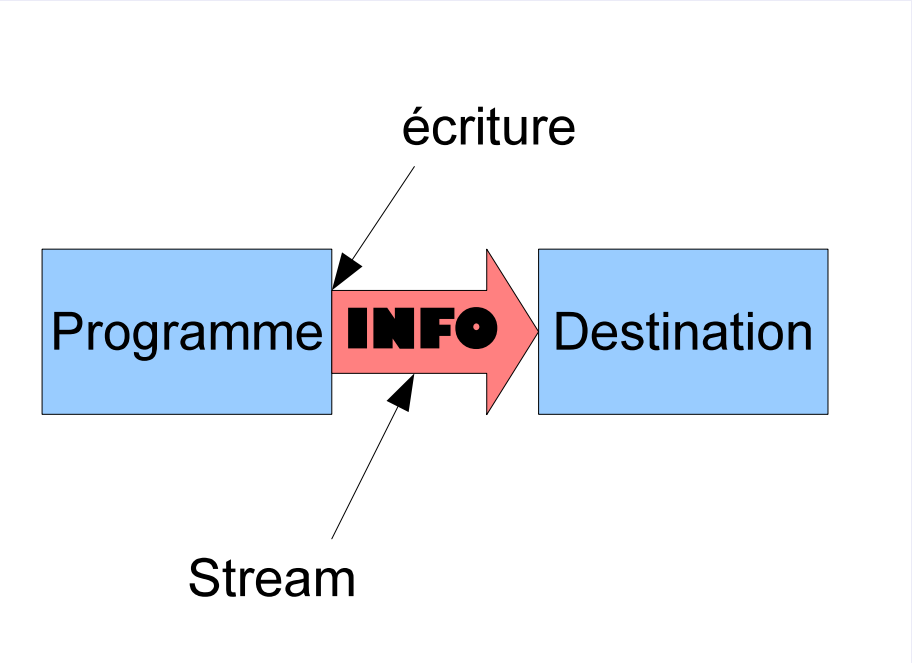
- Un stream est une séquence de données ayant une source et/ou une destination.
  - En entrée :
    - Pour obtenir de l'information, un programme ouvre un stream sur une source d'information (file, memory, socket,...) :



Les streams : sorties

Definition (Stream)

- Un stream est une séquence de données ayant une source et/ou une destination.
  - En sortie :
    - Le programme envoie de l'information sur une destination en ouvrant un stream et en envoyant l'information en sortie :

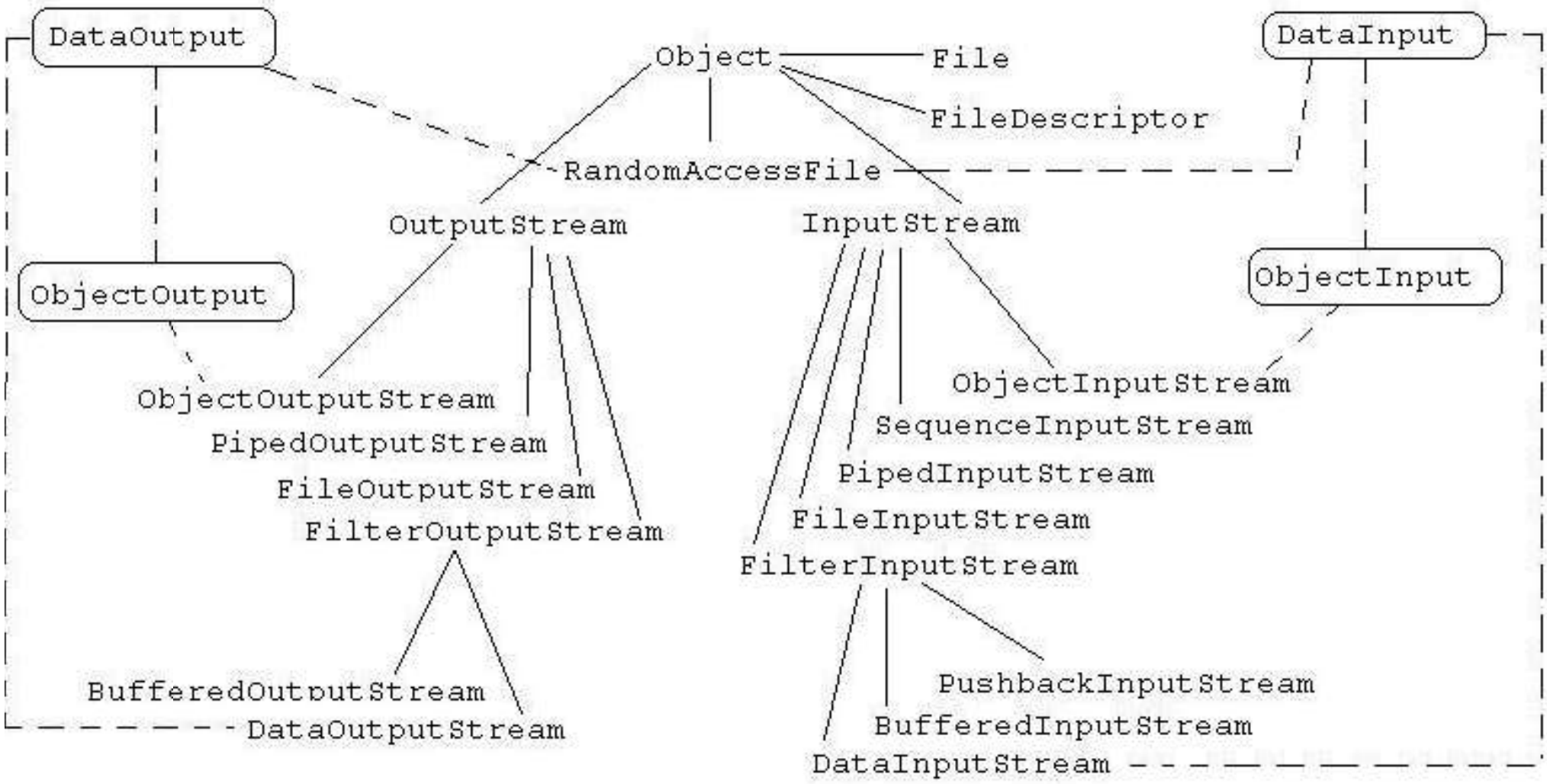


Les streams

Il existe deux grandes catégories de streams :

- les character streams :
  - données au format 16 bits (Unicode)
  - caractères
  - deux classes principales (super-classes abstraites) :
    - `Reader/Writer`
- les byte streams :
  - données au format 8 bits (ISO-Latin-1)
  - données différentes des caractères (image, son,...)
  - deux classes principales (super-classes abstraites) :
    - `InputStream/OutputStream`

Les byte streams



Les byte streams

Remarque

`System.out` et `System.in` sont paradoxalement des byte streams alors qu'ils traitent les caractères (pour des raisons historiques).

Example

Pour lire un caractère entré au clavier : `char c = (char)System.in.read();`

Pour lire un mot entier :

```
import java.io.*;
public class Test {
    public static void main(String[] args) throws IOException{
        char c;
        String chaine = "";
        System.out.println("enter some text :");
        while ((c=(char)System.in.read()) != '\n') {
            chaine = chaine + c; System.out.println(chaine);
        }
    }
}
```

La classe InputStream : quelques méthodes

- `public abstract int read() throws IOException`
  - lit un `byte` de données et le retourne sous forme d'un entier non signé (entre 0 et 255)
  - si il n'y a plus de `byte` à lire, la méthode retourne `-1`
- `public int read(byte[] buf, int offset, int count) throws IOException`
  - lit les bytes, les stocke entre `buf[offset]` et `buf[offset+count-1]` et retourne le nombre de bytes lus
  - retourne `-1` si la fin du stream est détectée
  - retourne 0 si `count==0` s'évalue à `true`
- `public int read(byte[] buf) throws IOException`
  - équivalent à `read(buf,0,buf.length)`
- `public int available() throws IOException`
  - retourne le nombre de bytes pouvant être lus
- `public void close() throws IOException`
  - ferme le stream

Exemple d'utilisation de InputStream

Example (Programme comptant le nombre de bytes dans un fichier)

```
import java.io.*;
public class CompteByte{
    public static void main(String[] args) throws IOException {
       InputStream in;
       in=new FileInputStream(args[0]);
       int total =0;
       while (in.read() != -1){
          total++;
       }
       System.out.println(total+'' bytes'');
    }
}
```

La classe OutputStream

- quelques méthodes de la classe abstraite `OutputStream` :
  - `public abstract void write(int b) throws IOException`
    - ★ écrit `b` sous forme de byte
  - `public void write(byte[] buf, int offset, int count) throws IOException`
    - ★ écrit les bytes de `buf[offset]` à `buf[offset+count-1]`
  - `public void write(byte[] buf) throws IOException`
    - ★ équivalent à `write(buf,0,buf.length)`
  - `public void close() throws IOException`
    - ★ ferme le stream

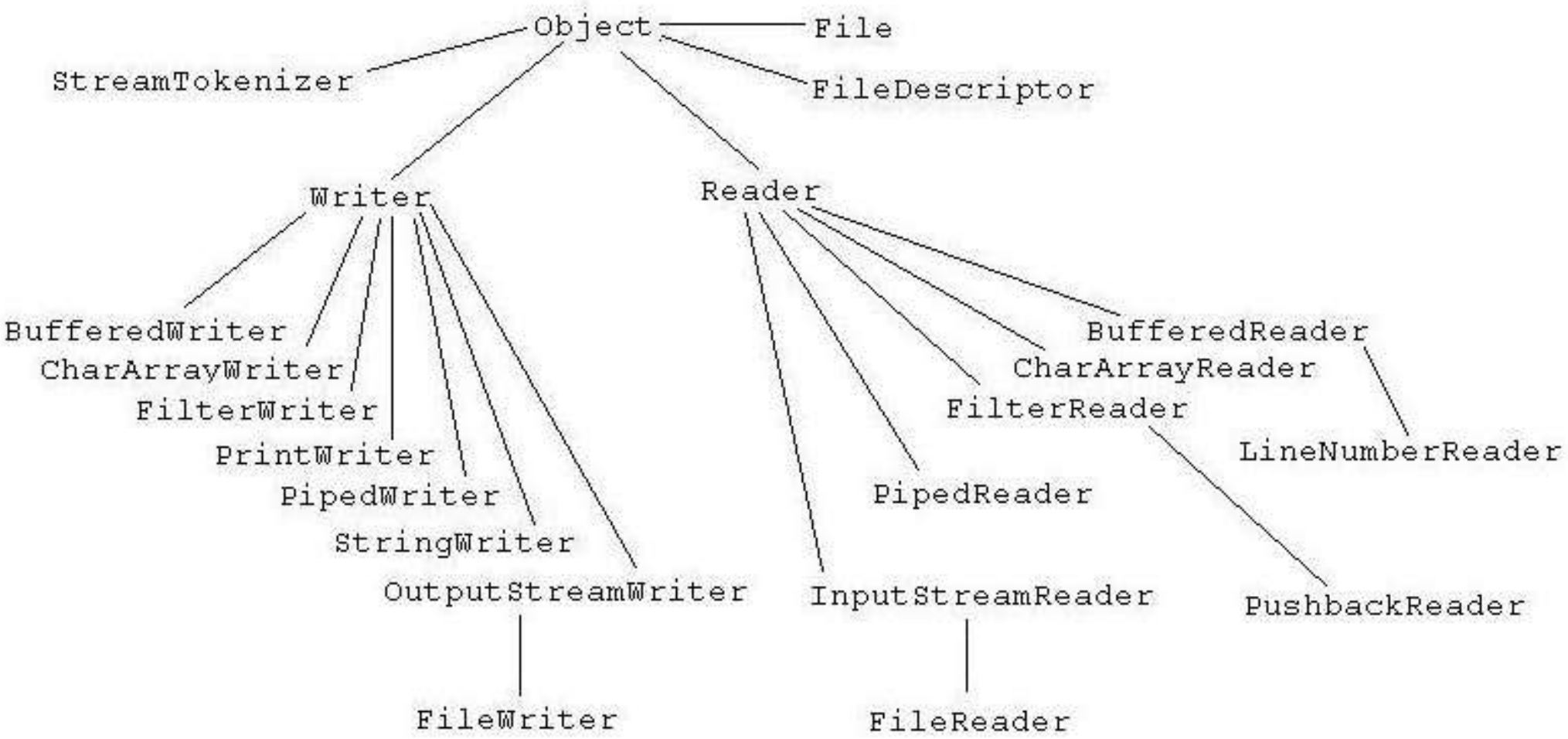
Exemple d'utilisation de OutputStream

Example (Programme copiant les bytes d'entrée vers les bytes de sorties après modification)

```
import java.io.*;
public class TranslateByte{
    public static void main(String[] args) throws IOException {
       byte avant = (byte) args[0].charAt(0);
       byte apres = (byte) args[1].charAt(0);
       int b;
       while ((b=System.in.read()) != -1){
          System.out.write(b==avant ?  apres :  b);
       }
    }
}
```

Java TranslateByte p m affiche mama si l'utilisateur saisit le texte papa.

Les character Streams



La classe Reader

- quelques méthodes de la classe abstraite `Reader` :
  - `public int read() throws IOException`
    - ★ lit un caractère et le retourne sous forme d'un entier (entre 0 et 65535)
    - ★ si il n'y a plus de caractère à lire, la méthode retourne `-1`
  - `public abstract int read(char[] buf, int offset, int count) throws IOException`
    - ★ lit les caractères, les stocke entre `buf[offset]` et `buf[offset+count-1]` et retourne le nombre de caractères lus
    - ★ retourne `-1` si la fin du stream est détectée
    - ★ retourne 0 si `count` est égal à 0
  - `public int read(char[] buf) throws IOException`
    - ★ équivalent à `read(buf,0,buf.length)`
  - `public boolean ready() throws IOException`
    - ★ retourne `true` si au moins un caractère peut être lu
  - `public abstract void close() throws IOException`
    - ★ ferme le stream

Exemple d'utilisation de Reader

Example (Programme comptant le nombre de caractères espace dans un fichier)

```
import java.io.*;
public class CompteEspace{
    public static void main(String[] args) throws IOException {
       Reader in;
       in=new FileReader(args[0]);
       int ch;
       int total;
       int spaces=0;
          for (total=0; (ch=in.read()) != -1;total++){
             if (Character.isWhitespace((char) ch)){
                spaces++;
             }
          }
       System.out.println(total+" chars, " +spaces+" spaces");
    }
}
```

La classe `Writer`

- quelques méthodes de la classe abstraite `Writer` :
  - `public void write(int ch) throws IOException`
    - ★ écrit `ch` sous forme de char (seuls les 16 premiers bits comptent)
  - `public void write(char[] buf, int offset, int count) throws IOException`
    - ★ écrit les caractères de `buf[offset]` à `buf[offset+count-1]` sur le stream
    - ★ variante avec `String buf` où les caractères écrits commencent à `buf.charAt(offset)`
  - `public void write(char[] buf) throws IOException`
    - ★ équivalent à `write(buf,0,buf.length)`
    - ★ variante avec `String buf` équivalente à `write(buf,0,buf.length)`
  - `public void close() throws IOException`
    - ★ ferme le stream

Conversion byte/character

Definition

Les classes `InputStreamReader` et `OutputStreamWriter` permettent des conversions byte stream / character stream :

- `public InputStreamReader(InputStream in, String encodage) throws UnsupportedEncodingException`
  - crée un stream character (`InputStreamReader`) à partir de l'`InputStream` et en utilisant le format `encodage`
  - possède une version utilisant l'encodage par défaut `InputStreamReader(InputStream in)`

Example (Conversion du format ISO 8859-6 (caractères Arabes) en caractères Unicode)

```
public Reader lecteurArabe(String file) throws IOException{
    InputStream fileIn = new FileInputStream(file);
    return new InputStreamReader(fileIn, “iso-8859-6”);
}
```

Conversion character/byte

Definition

Les classes `InputStreamReader` et `OutputStreamWriter` permettent des conversions byte stream / character stream :

- `public OutputStreamWriter(OutputStream out, String encodage) throws UnsupportedEncodingException`
  - crée un stream character (`OutputStreamWriter`) écrivant sur l'`OutputStream` en utilisant le format `encodage`
  - possède une version utilisant l'encodage par défaut `OutputStreamWriter(OutputStream out)`

Principales catégories de streams dans `java.io`

- les filtres : permettent d'effectuer des opérations de filtrage sur les données
  - `FilterReader, FilterInputStream, ...`
- les buffers : permettent le stockage temporaire des données en mémoire (évitent les recours abusifs au système de fichiers)
  - `BufferedWriter, BufferedOutputStream, ...`
- les pipes : sont des paires de streams permettant de diriger les sorties d'un thread vers l'entrée d'un autre.
  - `PipedReader, PipedWriter, ...`
- les mémoires : permettent de lire ou d'écrire directement sur des structures de données définies en mémoire
  - `charArrayReader, byteArrayOutputStream, StringReader, StringBufferInputStream, ...`

Principales catégories de streams dans `java.io`

- les prints : qui sont des input et output stream n'ayant pas de sortie, respectivement, d'entrée.
  - `PrintWriter`, `LineNumberReader`, `SequenceInputStream`, `ObjectInputStream`, ...
- les pushback : permettent de revenir en arrière lorsque la lecture est allée trop loin
  - `PushbackReader`, `PushbackInputStream`, ...
- les fichiers : permettent de manipuler (lecture/écriture) le système des fichiers
  - `FileWriter`, `FileInputStream`, ...
- les données : permettent de lire et écrire des données de types primitifs
  - `DataInputStream` et `DataOutputStream`

Les file streams

Definition

`FileReader/FileWriter` et `FileInputStream/FileOutputStream` :

- sont des classes permettant de lire ou d'écrire sur le système de fichiers
- possèdent toutes trois constructeurs :
  - un constructeur prenant un `String` nom de fichier
  - un constructeur prenant un objet `File` référant au fichier
  - un constructeur prenant un objet `FileDescriptor`
    - permet de faire référence à un fichier sans connaître son nom
    - la méthode `getFD` permet d'obtenir le `FileDescriptor` d'un file stream

Remarques

- La majorité des fichiers sont codés en 8 bits.
- Les `FileReader/FileWriter` encoderont les données en utilisant l'encodage par défaut du fichier :
  - peut être obtenu via `System.getProperty("fichier.extension");`

La classe `File`

Definition

- La class `File` permet de manipuler les noms de fichiers (chemins, ...) et possède 3 constructeurs :
  - `public File(String path)`
  - `public File(String dirName, String name)`
  - `public File(File filedir, String name)` :
    - équivalent à `File(filedir.getPath(), name)`
- Un fichier est séparé en répertoire et fichier par un caractère stocké dans l'attribut `static separatorChar` (ou `String separator`).
- La classe possède 5 méthodes permettant de connaître les composants du fichier (exemple : "`../test/Fichier.java`") :
  - `getName()` : "`Fichier.java`"
  - `getPath()` : "`../test/Fichier.java`"
  - `getAbsolutePath()` : "`/vob/java_prog/src/../test/Fichier.java`"
  - `getCanonicalPath()` : "`/vob/java_prog/src/test/Fichier.java`"
  - `getParent()` : "`../test`"

Tests de la classe `File`

Definition

- La classe `File` possède aussi des méthodes de test :
  - `exists` : `true` si le fichier existe dans le système
  - `canRead` : `true` si le fichier peut être lu
  - `canWrite`
  - `isFile` : `true` si ce n'est pas un répertoire ou un fichier spécial
  - `isDirectory` : `true` si c'est un répertoire
  - `isHidden` : `true` si le fichier est caché

Fonctionnalités de la classe `File`

Definition

- Quelques fonctionnalités (non exhaustives) :
  - `public long lastModified()` : valeur au format `long` du temps de dernière modification
  - `public long length()` : longueur en bytes
  - `public boolean delete()` : supprime le fichier et retourne true en cas de succès
  - `public boolean createNewFile()`
  - `public boolean mkdir()`
  - `public boolean mkdirs()` : crée tous les répertoire du chemin de l'objet `File`
  - `public boolean setReadOnly()`
  - `public void deleteOnExit()` : supprime le fichier lorsque la machine virtuelle termine

Exemple d'utilisation des file streams

Example

```
import java.io.*;
public class Copie{
    public static void main(String[] args) throws IOException{
       File input = new File(“toto.txt”);
       File output = new File(“raoul.txt”);
       FileReader in = new FileReader(input);
       FileWriter out = new FileWriter(output);
       int c;
       while ((c=in.read()) != -1){
          out.write(c);
       }
       in.close();
       out.close();
    }
}
```

Les random access file streams

Definition

Les objets `RandomAccessFile` :

- permettent d'avoir un accès aux données de manière non-séquentielle, contrairement aux autres streams.
- n'héritent pas de `InputStream/OutputStream` ni de `Reader/Writer`
- implémentent les interfaces `DataInput` et `DataOutput`.
- permettent à la fois la lecture et l'écriture via deux modes différents :
  - lecture seule "`r`"
  - lecture et écriture "`rw`"
- possèdent deux constructeurs :
  - `public RandomAccessFile (String chemin, String mode) throws FileNotFoundException`
  - `public RandomAccessFile (File fichier, String mode) throws FileNotFoundException`

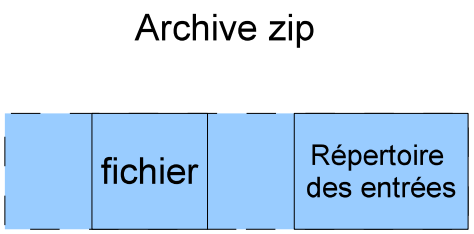
Les random access file streams

Definition

- se comportent comme des tableaux :
  - possèdent un pointeur, tête de lecture (et d'écriture dans le mode "`rw`")
  - en lecture (resp. écriture), lit (écrit) les bytes et avance le pointeur en fonction des bytes lus (écrits)
- 3 méthodes permettent de manipuler le pointeur :
  - `public long getFilePointer() throws IOException`
    - position actuelle du pointeur (en nombre de bytes) par rapport au début du fichier
  - `public void seek(long pos) throws IOException`
    - change la position du pointeur (si dépassement et écriture, la taille du fichier augmente)
  - `public int skipBytes(int count) throws IOException`
    - bouge la position du pointeur en fonction de l'entier `count` par rapport à la position courante.

Exemple de random access file streams

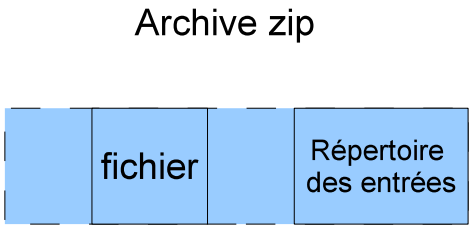
On désire extraire un fichier spécifique d'une archive ZIP :



Séquentiellement :

1. Ouvrir l'archive
2. Parcourir l'archive zip jusqu'à ce que l'on trouve le fichier que l'on désire extraire
3. Extraire le fichier
4. Fermer l'archive

Exemple de random access file streams



Avec `RandomAccessFile` :

1. Ouvrir l'archive
2. Chercher le fichier dans le répertoire des entrées (à la fin de l'archive)
3. Chercher (en arrière) la position du fichier à extraire
4. Extraire le fichier
5. Fermer l'archive

Cet algorithme est plus efficace car il ne lit que le répertoire des entrées et le fichier à extraire.

Les filter streams (filtres)

Definition

- `FilterReader/FilterWriter` et `FilterInputStream/FilterOutputStream` :
  - sont des classes abstraites définissant et implémentant partiellement les filtres.
  - ajoutent un constructeur prenant un stream à filtrer en argument.
  - permettent d'effectuer des opérations de filtrage sur les données.
- Quelques sous-classes de `FilterInputStream` et `FilterOutputStream` :
  - `DataInputStream` et `DataOutputStream` (convertisseur)
  - `BufferedInputStream` et `BufferedOutputStream` (buffers)
  - `PrintStream`, `LineNumberInputStream`, `PushbackInputStream`...
- Une seule sous-classe pour `FilterReader` :
  - `PushbackReader`

Créer ses propres filtres

Méthodologie :

1. Créer une nouvelle sous-classe des classes abstraites `FilterReader/FilterWriter` ou `FilterInputStream/FilterOutputStream`
   - Les filtres sont souvent par paire (filtre d'entrée, filtre de sortie)
2. Redéfinir les méthodes `read` et `write` (et autres si nécessaire)
3. Définir de nouvelles méthodes si nécessaire

Exemple de création de filtre

Example

```
public class ConvertisseurCaseHaute extends FilterReader{
    public ConvertisseurCaseHaute(Reader in){
        super(in);
    }
    public int read() throws IOException{
        int c=super.read();
        return (c==-1 ?  c :  Character.toUpperCase((char)c));
    }
    public int read(char[] buf, int offset, int count) throws IOException{
        int nbread = super.read(buf,offset,count);
        int last=offset+nbread;
        for (int i=offset;i<last;i++){buf[i]=Character.toUpperCase(buf[i]);}
        return nbread;
    }
    public static void main(String[] args) throws IOException{
        StringReader source = new StringReader(args[0]);
        FilterReader f = new ConvertisseurCaseHaute(source);
        int c;
        while ((c=f.read()) != -1){System.out.print((char) c);}
        System.out.println();
    }
}
```

Les data streams

Definition

- `DataInputStream` et `DataOutputStream` permettent de manipuler les données primitives.
- font parties des filtres et doivent donc toujours être associées à un stream d'entrée ou de sortie
- implémentent les interfaces `DataInput` et `DataOuput` incluant (de manière non-exhaustive) les méthodes abstraites :
  - `readTypePrimitif` et `writeTypePrimitif`
    - exemple : Pour les `short` : `readShort` et `writeShort`

Exemple de DataOutputStream

Example (Code créant un fichier stockant les valeurs de tableaux d'articles (prix, stock, nom))

```
DataOutputStream out = new DataOutputStream(new
FileOutputStream(“facture.txt”));
for( int i =0;i<prix.length;i++){
out.writeDouble(prices[i]);
out.writeChar('\t');
out.writeInt(units[i]);
out.writeChar('\t');
out.writeChars(descriptions[i]);
out.writeChar('\t');
```

Exemple de DataInputStream

Example (Code affichant les valeurs de tableaux d'articles)

```
DataInputStream in = new DataInputStream(new
FileInputStream(“facture.txt”));
try{
    while(true){
       prix=in.readDouble();
       in.readChar();//enleve la tabulation
       unit= in.readInt();
       in.readChar();//enleve la tabulation
       char ch;
       desc = new StringBuffer(20);
       char lineSep =System.getProperty(“line.separator”).charAt(0);
       while ((ch=in.readChar()) != lineSep){
          desc.append(ch);
       }
       System.out.println(“vous avez commande”+ unit +” unites de “+
desc “ a un prix de”+prix);
    }
}catch(EOFException e){\\...}
in.close();
```

Les buffer streams

Definition

`BufferedReader/BufferedWriter` et `BufferedInputStream/BufferedOutputStream` :

- sont des classes définissant les buffer streams
- sont généralement utilisées conjointement à des file streams
- permettent d'éviter les accès trop fréquents au système de fichiers en stockant les données en mémoire
- chacun de ces classes possèdent deux constructeurs :
  - un constructeur prenant un stream contenu et la taille du buffer.
    - ex : `new BufferedOutputStream( new FileOutputStream(chemin));`
  - un constructeur prenant un stream contenu qui donne une taille par défaut au buffer.

Les buffer streams

Remarque

Il faut garder une référence au stream contenu car il n'est pas possible d'accéder au stream contenu d'un buffer.

- si la méthode `read` est invoquée sur un stream buffer (`write` se comporte de manière similaire) :
  1. le programme invoque `read` sur le stream source
  2. remplit le buffer au maximum et retourne la donnée requise depuis le buffer
  3. les invocations suivantes de `read` retournent les données du buffer jusqu'à épuisement
  4. puis on recommence à l'étape 1 jusqu'à ce que les données du stream source soient épuisées.

Les piped streams (pipes)

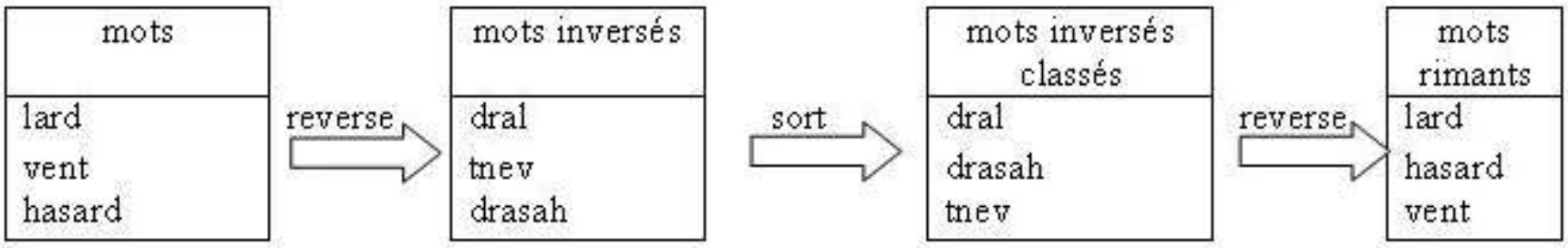
Definition

`PipedReader/PipedWriter` et `PipedInputStream/PipedOutputStream`

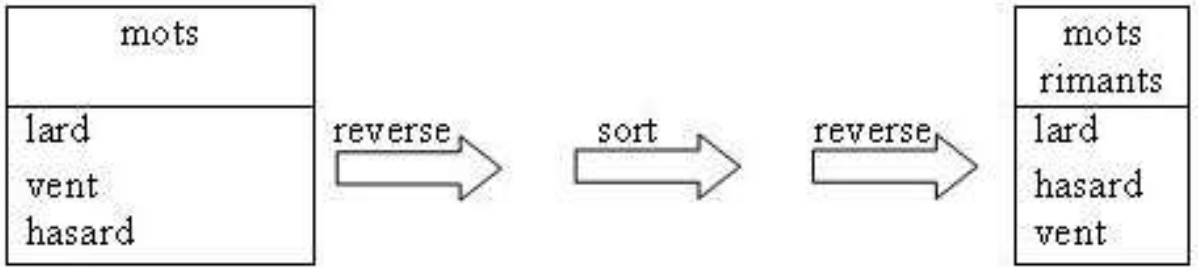
- fonctionnent par paire
- permettent de diriger la sortie d'un thread vers l'entrée d'un autre
- permettent donc d'effectuer des calculs sur les mots sans avoir besoin de stocker les résultats intermédiaires en mémoire
- stockent les données temporaires sur un buffer interne dont l'implémentation permet de lire et écrire à des vitesses différentes

Exemple de piped streams

Example (Obtenir une liste de mots rimants)



Algorithme avec stockage



Algorithme utilisant les pipes

Exemple de piped streams (suite)

Example

- Le contenu de la méthode `main` :

```
FileReader mots = new FileReader(“mots.txt”);
Reader motsrimants = reverse(sort(reverse(mots)));
```

- Le code de la méthode `reverse` :

```
public static Reader reverse(Reader src) throws IOException{
    BufferedReader in=new BufferedReader(src);
    PipedWriter pout=new PipedWriter();
    PipedReader pin=new PipedReader(pout);
    PrintWriter out = new PrintWriter(pout);
    new ReverseThread(out,in).start();
    return pin;
}
```

Connexion

Le code suivant permet la connexion entre le piped stream d'entrée et celui de sortie

```
PipedWriter pout=new PipedWriter();
PipedReader pin=new PipedReader(pout);
```

Remarques

- Les piped streams ne sont pas forcément connectés à leur création. Ils peuvent être connectés plus tardivement en utilisant la méthode `connect` :
  - `pipedReader.connect(pipedWriter);`
  - ou `pipedWriter.connect(pipedReader);`
- L'utilisation d'un pipe avant connection lance une `IOException`

Autre exemple de piped streams

Example (Méthode connectant un thread générant du texte à un thread lisant du texte)

```
public class Generateur extends Thread{
    private Writer out;
    public Generateur(Writer out){
       this.out=out;
    }
    public void run(){
       try{
          for(char c='a';c<='z';c++){
             out.write(c);
          }finally{
             out.close();
          }
       }catch(IOException e){
          getThreadGroup().uncaughtException(this,e);}
       }
    }
}
```

Autre exemple de piped streams (suite)

Example

```
class Pipe{
    public static void main(String[] args) throws IOException{
        PipedWriter out= new PipedWriter();
        PipedReader in= new PipedReader(out);
        Generateur donnee=new Generateur(out);
        donnee.start();
        while((ch=in.read())!=-1){
            System.out.print((char) c);
        }
        System.out.println();
    }
}
```

Streams mémoire : byteArray

`ByteArrayInputStream` utilise un tableau de bytes comme source de données :

- `public ByteArrayInputStream(byte[] buf)`
- `public ByteArrayInputStream(byte[] buf, int offset, int count)`

`ByteArrayOutputStream` construit un tableau dynamique stockant les données du ByteArray :

- `public ByteArrayOutputStream()`
- `public ByteArrayOutputStream(int size)`
- `public byte[] toByteArray()` : retourne une copie des bytes générés sur l'Ouput stream
- `public int size()` : retourne le nombre de bytes générés
- `public void reset()` : réinitialise le stream afin de réutiliser le buffer courant (effacé)

Streams mémoire : charArray

`CharArrayReader` utilise un tableau de caractères comme source de données :

- `public CharArrayReader(char[] buf)`
- `public CharArrayReader(char[] buf, int offset, int count)`

`CharArrayWriter` construit un tableau dynamique stockant les données du CharArray :

- `public CharArrayWriter()`
- `public CharArrayWriter(int size)`
- `public char[] toCharArray()` : retourne une copie des caractères générés sur le stream Writer
- `public int size()` : retourne le nombre de caractères générés
- `public void reset()` : réinitialise le stream afin de réutiliser le buffer courant (effacé)

Streams mémoire : `String` streams

Definition

- `StringReader` possède un unique constructeur construisant un stream à partir d'un `String`.
- `StringWriter` écrit son résultat sur un buffer (un `String` ou un `StringBuffer`) et possède les méthodes suivantes :
  - `public StringWriter()`
  - `public StringWriter(int size)`
  - `public StringBuffer getBuffer()` : retourne le buffer sous forme de `StringBuffer`
  - `public String toString()` : retourne le contenu du buffer sous forme de `String`

Streams mémoire : String streams

Example (Programme créant un String contenant les valeurs d'un tableau)

```
public static String arrayToString(Object[] obj){
    StringWriter sout= new StringWriter();
    PrintWriter out=new PrintWriter(sout);
    for (int i=0;i<obj.length;i++){
       out.println(i+": "+obj[i]);
    }
    return sout.toString();
}
```

Print streams

Definition

- Les classes `PrintStream` et `PrintWriter` :
  - permettent d'écrire des données dans un format lisible par le programmeur
  - fournissent des méthodes print, println sur les types char, char[], int, long, float, double, Object, String, boolean
- `PrintWriter` est préférable à `PrintStream` car elle traite les caractères
- Cependant, pour des raisons historiques les `PrintStream` `System.out` et `System.err` sont utilisés
- `PrintWriter` possède 4 constructeurs dont :
  - `public PrintWriter(OutputStream out)`
  - `public PrintWriter(Writer out)`

LineNumberReader streams

Definition

`LineNumberReader` permet de manipuler les lignes d'un texte en conservant des informations sur le numéro de ligne lors de sa lecture.

Example (Numéro de ligne de la première instance d'un caractère)

```
import java.io.*;
class RechercheChar {
    public static void main( String[] args) throws IOException{
if (args.length !=2){throw new IllegalArgumentException(“error”);}
        int match =args[0].charAt(0);
        FileReader file=new FileReader(args[1]);
        LineNumberReader in = new LineNumberReader(file);
        int ch;
        while((ch=in.read())!=-1){
if (ch==match){System.out.println(“le caractere ”+(char) ch+” a la ligne
”+in.getLineNumber());return;}
        }
System.out.println(“le caractere ”+(char) match+” n a pas ete trouve”);
    }
}
```

SequenceInputStream

Definition

`SequenceInputStream` :

- permet de créer un seul `InputStream` à partir de un ou plusieurs input streams
- lit le premier stream jusqu'à la fin, puis le suivant et ainsi de suite...
- possède deux constructeurs :
  - un constructeur pour deux `InputStream`
  - un constructeur pour une collection d'`InputStream`

SequenceInputStream

Example (Concaténation de fichiers donnés dans la ligne de commande)

```
import java.io.*; import java.util.*;
class Concat{
    public static void main( String[] args) throws IOException{
        InputStream in;
        if (args.length ==0){in=System.in;}
        else{
            InputStream filein, bufin;
            List inputs = new ArrayList(args.length);
            for(int i=0; i<args.length;i++){
                filein=new FileInputStream(args[i]);
                bufin=new BufferedInputStream(filein);
                inputs.add(bufin);
            }
            Enumeration files = Collection.enumeration(inputs);
            in =new SequenceInputStream(files);
        }
        int ch;
        while((ch =in.read())!=-1){System.out.write(ch);}
    }
}
```

Pushback streams

Definition

- `PushbackStream` et `PushbackReader` permettent de revenir en arrière après avoir lu un caractère.
- Cette faculté est utilisée par les analyseurs lexicaux qui ne connaissent la fin d'un mot qu'en ayant lu la première lettre du mot suivant.
- Ils possèdent deux constructeurs prenant comme argument :
  - le stream contenu et la taille du pushback buffer
  - le stream contenu
    - et ne pouvant stocker en mémoire qu'un seul byte ou caractère
- Ils lancent une `IOException` si on excède la capacité de mémorisation.

Pushback streams

- Le retour en arrière est effectué grâce à la méthode `unread` :
  - `public void unread(int c ) throws IOException`
    - ajoute le caractère `c` en tête du `PushbackStream`
  - `public void unread(char[] buf, int offset, int count) throws IOException`
    - ajoute les caractères entre `buf[offset]` et `buf[offset+count-1]` dans le `PushbackStream`
  - `public void unread(char[] buf) throws IOException`
    - équivalent à `unread(buf,0,buf.length)`

Pushback streams

Example (Programme retournant la plus longue séquence consécutive d'un bit dans un programme)

```
import java.io.*;
class CompteurSequence{
    public static void main( String[] args) throws IOException{
        InputStream i = new FileInputStream(args[0]);
        PushbackInputStream in = new PushbackInputStream(System.in);
        int max=0; //plus longue sequence trouvee
        int maxB=-1; //le byte de la séquence
        int b; //le byte courant en entrée
        do{
            int compte;
            int c = in.read(); //premier byte de la séquence
                for(compte=1; (b=in.read())==c;compte++){continue;}
                if(compte > max){
                max=compte;//memorisation de la plus longue séquence
                maxB=c;//memorisation du byte de la plus longue séquence
            }
            in.unread(b); //pushback
        }while( b!=-1)
        System.out.println(max+" bytes of "+maxB);
    }
}
```

Stream Tokenizer

Definition

- La classe `StreamTokenizer` permet de segmenter du texte :
  - Sa méthode `nextToken` permet de reconnaître le segment suivant.
- Il y a quatre types de segment pouvant être reconnus :
  1. un mot :
     - l'attribut `ttype` vaut `TT_WORD` et l'attribut `String sval` contient le mot reconnu.
  2. un nombre :
     - l'attribut `ttype` vaut `TT_NUMBER` et l'attribut `double nval` contient la valeur du nombre.
  3. une fin de ligne :
     - l'attribut `ttype` vaut `TT_EOL`
  4. une fin de fichier :
     - l'attribut `ttype` vaut `TT_EOF`

Exemple de stream Tokenizer

Example (Méthode calculant la somme des nombres contenus dans un Stream Reader)

```
static double sommeStream(Reader in) throws IOException{
    StreamTokenize nums= new StreamTokenizer(in);
    double resultat = 0.0;
    while (nums.nextToken() !=StreamTokenizer.TT_EOF){
       if(nums.ttype == StreamTokenize.TT_NUMBER){
          resultat +=nums.nval;
       }
    }
    return resultat;
}
```

La sérialisation

Definition

- La sérialisation (serialization) permet de sauvegarder des objets :
  - dans un byte stream transféré sur le réseau (ex RMI-communication par socket)
  - sur le disque (dans un fichier ou dans une BD)
- La sérialisation consiste en la conversion d'un objet vers un stream de bytes
- L'opération inverse s'appelle désérialisation
- Les streams `ObjectInputStream` et `ObjectOutputStream` permettent d'effectuer ces opérations via les méthodes par défaut :
  - `writeObject` et `readObject`

Exemples de sérialisation

Example (Exemple de sérialisation)

```
FileOutputStream out = new FileOutputStream(“temps”);
ObjectOutputStream s = new ObjectOutputStream(out);
s.writeObject(“Today”);
s.writeObject(new Date());
s.flush();
```

Example (Exemple de désérialisation)

```
FileInputStream in = new FileInputStream(“temps”);
ObjectInputStream s = new ObjectInputStream(in);
String today = (String) s.readObject();
Date date = (Date) s.readObject();
```

L'interface `Serializable`

Definition

- Un objet doit implémenter l'interface Serializable pour pouvoir être sérialisé :

```
package java.io;
public Interface Serializable{
//vide
}
```

- Dans le cas contraire, toute tentative de sérialisation lance une `NotSerializableException`.
- Les attributs `static` et `transient` ne sont pas sérialisés.
- Si un programme contient des données confidentielles, il faut donc les spécifier `transient`.
- La désérialisation préserve l'intégrité des références :
  - Si deux attributs pointent vers le même objet, l'objet désérialisé possèdent deux attributs pointant vers le même objet.

Sérialiser un singleton

Example (Singleton)

```
public class Elvis{
    public static final Elvis INSTANCE = new Elvis();
    private Elvis{...}
}
```

- Pour sérialiser un singleton, il n'est pas suffisant d'ajouter `implements Serializable` :
  - En effet, chaque désérialisation créera un nouvel objet
  - Dans l'exemple précédent, la désérialisation entrainerait la création de multiples `Elvis`

Example (Solution : ajouter une méthode readResolve (depuis Java 1.2))

```
private Object readResolve() throws ObjectStreamException{
    //retourne le VRAI Elvis
    return INSTANCE;
}
```

Part VI

L'interface utilisateur