Exceptions, Fichiers et entrées/sorties en JAVA

DESS CCI 2, 2001/2002

27 novembre 2003

Table des matières

Chapitre 1

Exceptions

Lorsqu'un programme traite des données, il peut arriver deux types de situations gênantes :

- on peut lui demander d'enlever un élément d'une liste vide. Il est possible de traîter ce problème tout de suite, en disant que le résultat, dans ce cas, est une liste vide ;
- on peut aussi demander la valeur du premier élément d'une liste vide. Dans ce cas, on ne peut pas répondre. La seule chose possible est de dire qu'il y a une erreur à cet endroit, et charge à d'autres d'essayer de réparer cette erreur.

Dans le premier type de situations, il est possible de modifier le code en séparant les cas (içi, liste vide ou liste non-vide), pour éliminer le problème.

Le deuxième cas est plus gênant, car on ne peut pas le traîter au niveau actuel. Il faut arrêter de faire ce qui était en cours, et signaler l'erreur. On appelle celà une **Exception**.

Prenons un autre exemple :

```
float division(float a, float b) {
  return a/b;
}
```

La méthode **division** ne semble pas poser de problèmes. On peut cependant remarquer qu'il peut y avoir un problème si $b = 0$. C'est d'autant plus gênant que cela n'embête pas Java, qui va rendre une valeur. Autrement dit, on va continuer le calcul avec une valeur qui n'a aucun sens, et sans le savoir. Dans le cas précédent, il n'y avait pas d'autre possibilité que de s'arrêter. Dans ce cas-çi, il est possible de continuer, mais ce n'est pas souhaitable. Il faut signaler qu'il y a eu un problème à cet endroit. Il est donc préférable, là encore, d'utiliser une **Exception**.

> Une **Exception** est un problème qu'il n'est pas possible de traîter immédiatement.

Il y a beaucoup de types d'exceptions déjà définis en Java. Ils dérivent tous de la class Exception. Le but est que la classe d'une exception soit suffisament précise pour que le programmeur puisse savoir, juste en connaissant cette classe, quel a été le problème. Il est aussi possible d'ajouter de nouvelles exceptions pour les classes qu'on défini. Dans l'exemple suivant, NullPointerException est une classe d'exception déjà définie, qui correspond au cas où on a un objet qui est vide alors qu'il ne devrait pas l'être.

```
// ajouter une exception pour le cas file_vide
class FileVideException extends NullPointerException  {
   public FileVideException () {
   }

   public FileVideException (String message) {
      super(message) ;
   }
}
```

Dans ce cas, on n'ajoute rien à la classe, mais le nom de la classe de l'exception donne plus de renseignement sur l'erreur.

1.1 Erreurs controlées et non-contrôlées

Une exception (=erreur) peut soit être non-contrôlée (donnée par une fonction qu'on ne contrôle pas), soit contrôlée (lors de l'écriture du programme, on decide qu'on est dans un cas d'erreur). Dans le premier cas, il faut se référer à un manuel Java pour savoir à quoi correspond l'erreur.

1.1.1 Création d'une exception

Pour indiquer un cas d'erreur, on utilise la commande throw :

```
// code ...
throw  new FileVideException("pas de valeur pour les listes vides !") ;
...
```

Un exemple standard est la division flottante par zéro. Ce n'est pas un problème pour java, qui renvoie juste une valeur NaN (Not a Number)[1]. Si on veut signaler une erreur dans ce cas, on peut utiliser :

```
class DivisionParZero extends Exception  {
   public float numerateur ;
   public DivisionParZero(){
   }

   public DivisionParZero(String message){
      super(message) ;
   }
}
```

On peut ensuite utiliser ce type d'exception dans un autre fichier :

[1] Dans le cas d'une division entière, Java signale une exception du type ArithmeticException

```
...
  if (b == 0 ) {
    throw  new DivisionParZero("le numerateur est : " + a) ;
  }
  else {
    c = a/b ;
  }
...
```

1.1.2 Déclaration des méthodes

Lorsqu'une méthode peut générer une erreur, elle *doit* le signaler. Pour cela, lors de la déclaration de la méthode, on doit mettre, par exemple :

```
// Déclarer que la méthode division peut renvoyer une exception DivisionParZzero
...
  public float division (float a,float b) throws DivisionParZero {
    if (b == 0){
      throw  new DivisionParZero () ;
    }
    return a/b ;
  }
...
```

Si une méthode déclare qu'elle peut générer une exception, il *faut* que cette exception soit rattrappée dans une des méthodes appelantes. Certaines méthodes définies dans les bibliothèques standard, par exemple celles de lecture/écriture de fichiers, déclarent qu'elles peuvent générer des exceptions. Lorsqu'on les utilise, il est donc obligatoire de pouvoir traiter ces exceptions.

1.2 Traitement des erreurs

Pour l'instant, dès qu'une exception a été lancée, le programme s'arrête. Cela peut être gênant, car le plus souvent, une erreur ne concerne qu'une partie du travail que le programme a à faire. Par exemple :

- une calculatrice, à qui on demande de calculer $5/0$;
- un programme qui maintient une liste de clients en attente, et deux programmes qui servent ces clients, si un de ces programmes clients reçoit un client vide.

Il faut donc pouvoir rattrapper les erreurs qui peuvent apparaitre, afin de limiter la portée de l'exception, et de pouvoir continuer le reste du programme. Pour cela, on utilise l'instruction `try` :

```
try {
  if (b == 0 ) {
    throw new DivisionParZero(“le numerateur est : ” + a) ;
  }
  else {
    c = a/b ;
    System.out.println (""“le resultat est : ” +c");
  }
}
catch (DivisionParZero e) {
  System.out.println (""“on a essaye de faire une division par 0”"");
}
System.out.println (""“coucou”"");
```

Il y a alors 2 cas :

1. si $b == 0$: on lance une exception, elle est rattrappee par le bloc catch , donc on execute ce bloc. Donc on affiche le message. Une fois qu'il est termine, on passe à la suite, on affiche “coucou”
2. si b !=0 : on effectue la division, on affiche le resultat, on passe sur le bloc catch , et on affiche juste “coucou”

Il n'est pas nécessaire que l'erreur soit rattrappée dans la méthode ! Tout l'intérêt de ce mécanisme est justement de pouvoir arrêter l'exécution d'une méthode et de pouvoir continuer d'autres calculs. Dans ce cas, lors de la déclaration de la méthode, il faut préciser tous les types d'exceptions qui peuvent être engendrés. Par exemple :

```
public float division(float a, float b) throws DivisionParZero {
  float c ;

  if (b == 0 ) {
    throw  new DivisionParZero(“le numerateur est : “ + a) ;
  }
  else {
    c = a/b ;
  }
  return c ;
} ;
```

Ensuite, pour utiliser cette méthode, il faudra rattrapper, à un moment ou à un autre, l'exception qu'a envoyée cette méthode. Par exemple :

```
try {
  c = division(a,b) ;
}
catch (DivisionParZero e) {
  System.out.println (""On a essaye une division par zero ;"");
  System.out.println ("e.getMessage()");
}
```

Ou encore, si on utilise la méthode division pour l'algorithme du pivot de Gauss :

```
try {
   P = PivotdeGauss(M);
}
catch (DivisionParZero e) {
   System.out.println (""on a utilise une division par zero"");
// ca peut vouloir dire que la matrice M n'est pas inversible...
}
```

exercice : Déclarer une classe ListeEntiers, utilisant des exceptions, et ne contenant que les méthodes estVide, prem et reste.

1.3 Gestion des exceptions

Dans l'exemple précédent, ca ne nous renseigne pas beaucoup de savoir qu'il y a eu une division par zéro à un endroit de l'algorithme. Il faut pouvoir gérer au fur et à mesure les exceptions pour qu'au moment où le problème peut être rattrapper, on dispose de suffisament d'informations sur l'erreur. Prenons l'exemple d'une liste, dans laquelle on lit les elements pour les afficher. on peut utiliser l'algorithme suivant :

```
try {
   while (true) {
      a = l.valeur();// peut lancer une exception
      l = l.suivant();
      System.out.println ("a");
   }
}
catch (ListeVideErreur e) {
   // on a termine, il n'y a plus rien a faire
}
```

On peut modifier cet algorithme pour afficher le n-ieme element de la liste :

```
try {
   while (n > 0) {
      a = l.valeur();// peut lancer une exception          l = l.suivant();
      n--;
   }
}
catch (ListeVideErreur e) {
   // dans ce cas, il faut prevenir que la liste a moins de n elements ! ...
}
System.out.println ("a");
```

Pour prevenir que la liste a moins de n elements, on peut lancer une nouvelle exception :

```
  try {
...
  }
  catch (ListeErreurVide e) {
    throw  new ErreurPasAssezdElements() ;
  }
```

Il est aussi possible de mettre plusieurs catch a la suite. Dans ce cas, c'est le premier dont la classe correspond à la classe de l'exception qui est utilisé :

```
  try {
...
  }
  catch (ErreurPasAssezdElements e) {
...
  }
  catch (DivisionParZero e) {
...
  }
...
```

La possibilité de créeer de nouvelles classes d'erreur alliée à la possibilité de relancer des erreurs permet d'avoir une description precise de l'erreur au niveau où au doit la traiter.

1.4 Conclusion sur les exceptions

– mécanisme tres utile ;
– permet de séparer le traitement general des cas particuliers qui engendrent des erreurs ;
– permet d'arreter a tout moment l'execution d'une boucle ou d'une methode

Elles peuvent aussi servir lors du développement d'un programme, pour savoir d'où peut venir une erreur (on génère une exception, et on la rattrappe en affichant la valeur des variables...)

Il ne faut pas utiliser les exceptions pour remplacer des tests. L'exemple avec la liste vide est mauvais, on passe beaucoup plus de temps avec ce programme qu'en faisant un test avant de rendre la valeur.

Chapitre 2

Les flux (streams)

2.1 C'est quoi les entrées/sorties d'un programme ?

exemple : on peut prendre le cas d'un programme qui calcule les nombres premiers inférieurs à N en utilisant le crible d'erathostene. On suppose qu'un autre programme a besoin des nombres entiers premiers inférieurs à N. Comment peut-on les faire communiquer ?

Dans le cas d'un programme qui communique avec l'utilisateur, il y a aussi besoin de manier les entrées (ce que donne l'utilisateur) et les sorties (ce que le programme affiche).

Java est tres lourd sur ce plan, car il distingue ces 4 cas comme cas de base :

1. lecture de données
2. ecriture de données
3. lecture de caractères affichables
4. écriture de caractère affichables

et les différencie en utilisant 4 classes de base pour les entrées/sorties. Ces 4 classes sont respectivement :

1. InputStream
2. OutputStream
3. Reader
4. Writer

Ces classes ne peuvent pas etre utilisées, on ne peut instancier que des classes dérivées de celles-ci, et il y en a 60 différentes en tout ! (Et il est hors de question de toutes les voir !)

2.2 Les flux (streams)

Pour comprendre les InputStream et les OutputStream, il peut être utile de savoir ce qu'est un stream. On se place dans le cas ou un programme lit des donnees.

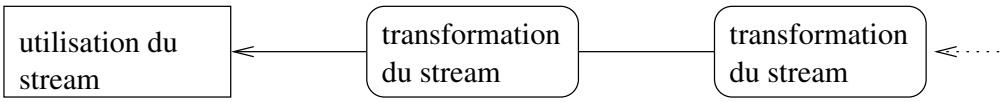

FIG. 2.1 – Les flêches indiquent la direction des données

Le programme n'a pas besoin de savoir d'ou viennent les données, il va les récupérer les unes après les autres. Le but, pour le programmeur, est de composer les transformations pour que, à partir du flux initial de données, il obtienne un flux de données qui soient utilisables dans le programme.

On peut représenter les streams comme des files[1] de données.

[1] premier arrivé, premier servi

flux ou stream = maniere d'acceder à un objet (périphérique, fichier, partie de la mémoire). Les flux sont à la base des systèmes UNIX.

2.3 En Java

Chaque stream est représenté par une classe Java. Celles-ci dépendent de la manière de lire ou d'écrire le stream, des objets qui sont sur le stream, ou de la signification de la suite d'objets du stream (si ça représente un fichier ZIP, par exemple).

Dans la suite, stream = binaire, et flux = texte.(completement arbitraire, flux est la traduction de stream !).

Chapitre 3

InputStream et OutputStream

3.1 Création d'un $stream$

3.1.1 Création d'un stream à partir d'un fichier

Une classe importante est la classe FileInputStream (ou FileOutputStream), qui est dérivée de la classe InputStream (resp. OutputStream), et qui permet de lire (resp. d'écrire) dans un fichier :

FileInputStream : $< fichier > \mapsto stream(octets)$
FileOutputStream : $< fichier > \mapsto stream(octets)$
FileOutputStream : $< fichier > \times boolean \mapsto stream(octets)$

où $< fichier >$ est donné soit par son nom, soit par un objet de type $FILE$. $stream(octets)$ est un stream dans lequel il n'est possible de lire que des octets. Si il n'y avait pas de transformations possibles, il ne serait donc pas possible de faire grand chose avec les fichiers. Il serait, par exemple, impossible de lire un fichier de nombres entiers.

La première version de FileOutputStream est à utiliser si on veut effacer un éventuel fichier déjà existant portant ce nom. Si on veut ajouter des données à un fichier déjà existant, il faut utiliser la deuxième version de FileOutputStream, avec la valeur du booléen à true.

Enfin, lorsqu'on a terminé d'utiliser un $stream$ flux, il faut le fermer en utilisant l'instruction :

```
...
   flux.close() ;
...
```

3.1.2 Création d'un stream à partir d'un String

la classe StringBufferInputStream

C'est une classe qu'il ne faut normalement plus utiliser. Il faut utiliser, à la place, StringReader. Mais c'est beaucoup plus simple si on veut lire une ligne qui contient des données (int ,...). Le constructeur de cette classe prend un String en argument.

StringTokenizer

L'utilisation est un peu differente, car il n'y a pas de méthode read/write. À la place, on donne en argument une chaîne de caractères contenant des séparateurs de texte. Cette classe définie les méthodes suivantes :

– constructeur (String,String delim) ou (String)
– hasMoreTokens() $\emptyset \rightarrow bool$
– countTokens() $\emptyset \rightarrow int$
– nextToken() $\emptyset \rightarrow String$
– nextToken(String delim) $String \rightarrow String$

L'utilité est qu'il est possible de lire les valeurs d'un objet, par exemple, champs par champs. Un séparateur souvent utilisé est "|".

3.1.3 Création d'un Stream à partir d'un tableau d'octets

Un tableau d'octets (ByteArray) peut contenir des données quelconques en retenant leur valeur en mémoire. Les deux classes ByteArrayOutput et ByteArrayInputStream permettent de créer un stream vers ou à partir de tels tableaux.

Une fois qu'on a terminé d'utiliser un stream de la classe ByteArrayOutput ou ByteArrayInputStream, on peut transformer ce stream en un tableau d'octets en utilisant la méthode toByteArray().

3.2 Manipulation des $streams$ – les filtres

3.2.1 Introduction

Heureusement, il est possible de combiner les streams pour faire des opérations plus complexes. En effet, la plupart des classes des stream sont créées avec un constructeur prenant un stream en argument. Par exemple :

```
DataInputStream( InputStream )
```

DataInputStream est une classe dérivée de InputStream, et dont le constructeur prend en argument un objet d'une classe dérivée de InputStream. L'objet $stream$ qu'on obtient en sortie permet de lire les types basiques de Java (int, $float$, ...).

Il est alors facile des lire un fichier "toto.int" contenant des entiers :

```
DataInputStream fichierentier = new DataInputStream( new FileInputStream("toto.int")) ;
```

qu'on utilise ensuite avec :

```
a = fichierentier.readInt() ;
```

On peut représenter cette construction avec le diagramme suivant :

3.2.2 InputStream et OutputStream

classes abstraites

InputStream : read()

permet de lire *un* octet ou un tableau d'octets.

OutputStream : read()

permet d'écrire *un* octet ou un tableau d'octets.

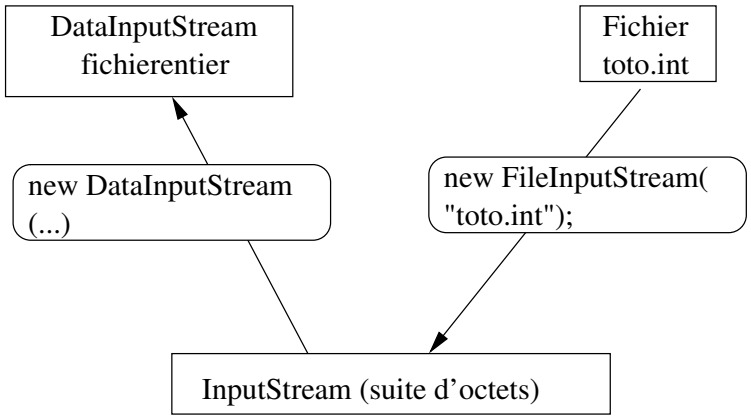

FIG. 3.1 – On commence par transformer le fichier en un flux d'entrées InputStream avec un objet FileInputStream, puis on transforme ce flux d'entrées en un flux de données...

3.2.3 BufferedInputStream et BufferedOutputStream

explication de ce qu'est un buffer
plus rapide
pas d'autres fonctionnalitées, permet d'avoir des entrees/sorties plus rapides.

3.2.4 DataInputStream et DataOutputStream

=> lire des données (entier, flottant,...)
read(Boolean|Byte|Char|Double|Float|Int|Line|Long|Short|UTF)()
DataOutputStream => pareil, mais avec write !

3.3 Lire et écrire des objets – l'interface Serializable

3.3.1 Le problème

* 2 pointeurs sur le même objet *
– écriture (3 fois) ;
– lecture (3 fois)
conclusion : après lecture et écriture, on n'a plus les mêmes listes !
solution Il est aussi possible de stocker des objets dans un fichier, pour pouvoir les réutiliser lors d'une session suivante. Pour des objets de la classe **toto**, on opère de la façon suivante :
on commence par préciser que la classe **toto** est une classe d'objets qui peuvent être enregistrés dans un fichier. Pour cela, il faut légèrement modifier la déclaration de la classe **toto** :

```
class toto implements Serializable {...}
```

C'est la seule modification à apporter à la classe **toto** ;

3.3.2 Utilisation

on peut ensuite lire et écrire les objets de la classe **toto** à l'aide d'une des deux classes suivantes, dont les constructeurs sont :

```
ObjectInputStream(InputStream in);
ObjectOutputStream(OutputStream out);
```

Exercice :
Faire une méthode qui prend en entrée un objet de la classe **toto** et et un nom de fichier, et qui écrit l'objet dans le fichier.

3.3.3 Les versions

problème : différentes versions d'un programme.
On veut que même si la définition d'une classe change, on puisse continuer à lire un objet d'une ancienne classe.
Ce n'est pas possible par defaut (codage de la classe !).
On peut ajouter, à la definition d'une classe, une constante

static final long serialVersionUID

Lorsqu'on lit des objets, c'est légèrement plus compliqué, puisqu'il faut préciser la classe des objets lus avant de faire une affectation. Un très grand avantage de Java est qu'il est possible de lire et d'utiliser des objets lus dans un fichier et dont on ne connait pas la classe.

On va cependant se limiter au cas où on connait la classe (**toto**, encore) de l'objet lu. On utilise presque le même code que pour écrire l'objet :

```
public toto LitObjet (String fichier) {
   toto objet;
   ObjectInputStream in =  new ObjectInputStream(
         new FileInputStream(fichier));
   objet = (toto)in.readObject();
   in.close();
   return objet;
}
```

3.3.4 Disgression : le clonage

différence clonage/copie
interface Cloneable
methode clone()
une implémentation possible :

```
class SerialCloneable implements Cloneable, Serializable {
  public Object clone () {
    try {
      ObjectOutputStream oot ;
      ByteArrayOutputStream baot
```

```
      ByteArrayInputStream bain;
      ObjectInputStream ois;
     Object ret;

     // on commence par ecrire l'objet dans un tableau

      baot = new  ByteArrayOutputStream ();
     oot = new ObjectOutputStream(baot);
      out.writeObject(this);
      out.close();

     // puis on relie l'objet a partir du tableau

      bain = new
                        ByteArrayInputStream(baot.toByteArray());
       ois = new
                        ObjectInputStream (bain);
       ret = ois.readObject();
       ois.close();

       // on a maintenant une copie exacte dans ret !
       return ret;
    }
    catch (Exception e) {
      return null;
    }
  }
}
```

Chapitre 4

Les classes Reader et Writer

4.1 Unicode

stream = données
maintenant : texte (lisible par l'utilisateur)

Le problème, lors d'un affichage, est d'être sûr d'afficher des caractères lisibles par l'utilisateur. Il est donc intéressant de savoir comment fait un programme normal (par exemple, un terminal) pour afficher des caractères.

L'exemple le plus simple est d'afficher le contenu d'un fichier (avec cat, par exemple). Un fichier est une suite d'octets (byte), qu'on va écrire comme une suite d'entiers entre 0 et 255. Pour afficher, le terminal :

1. regarde l'octet qu'il doit afficher, par exemple 80 ;
2. utilise un tableau de conversion pour les caractères qui dépend en général du pays (le codage ascii), et qui donne un numéro de lettre, par exemple celui de 'p' ;
3. affiche cette lettre.

Le problème est que certaines lettres ne sont pas définies dans le tableau, comme Ü, par exemple. Dans ce cas, elles n'apparaissent pas.

Le codage unicode, de son côté, associe à chaque caractère un numéro sur 2 octets. Il ne sert pas directement à l'affichage, mais est très utile pour communiquer entre des ordinateurs différents (c'est le même codage partout !). Le deuxième octet désigne le carctère, et on ajoute un premier octet qui désigne le tableau dans lequel ce carctère doit être interprêté.

4.2 Unicode et Java

En interne, Java utilise le codage Unicode (UTF) qui associe à chaque caractère un numéro particulier, mais qui est sur 2 octets (penser au grec,japonais, vietnamien...). Il faut donc, lors des entrées/sorties, convertir le codage du caractère tel qu'il est stocké par Java (sur deux octets) en un codage qui peut être affiché par le terminal (et donc sur un octet).

Les classes InputStream et OutputStream servent à sauvegarder ou à lire des *données* dans un fichier. L'affichage est beaucoup plus complexe, puisque le code du caractère qu'on désire afficher dépend de la langue utilisée par la personne utilisant le programme. Il faut penser à un russe qui voudrait lire des caractères cyrilliques. Il existe un standard qui, pour chaque langue, donne un numéro à un caractère. Les classes dérivées de Reader et Writer servent de traductrices entre la représentation Unicode utilisée par Java et la représentation des caractères sur chaque ordinateur.

> Chaque fois qu'on veut que des données puissent être écrites ou lues par un utilisateur, il faudra utiliser une classe dérivée de la classe Reader ou Writer

Par défaut, Java considère que le format des données de InputStream ou de OutputStream est celui de l'ordinateur courant. On peut spécifier un autre langage avec un paramètre optionnel. Par exemple, pour écrire des caractères allemands tels que Ü,

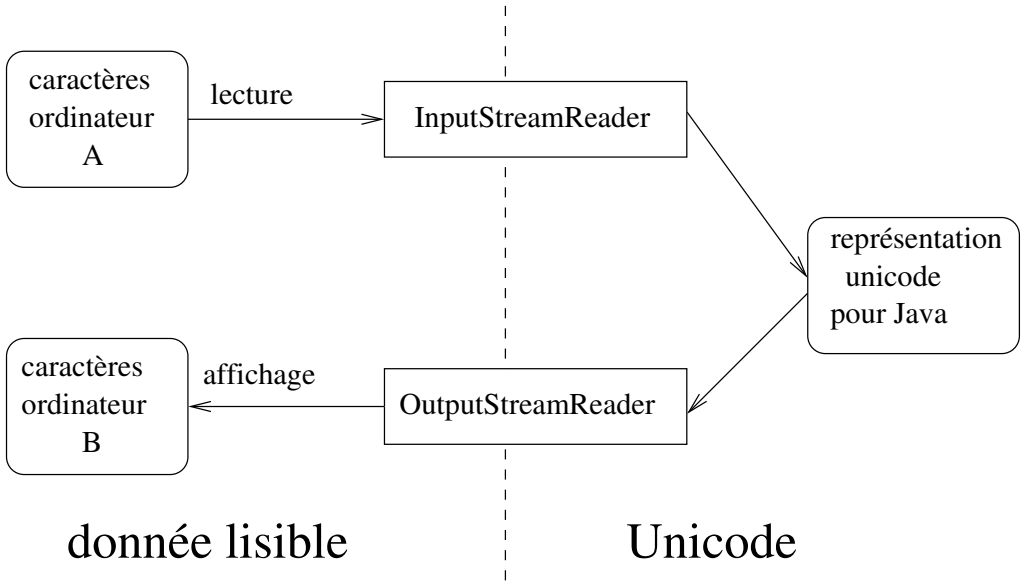

FIG. 4.1 – Java permet d'avoir le même affichage sur deux ordinateurs différents en passant par le codage Unicode.

On peut représenter l'action des objets de ces classes avec le diagramme suivant :

4.3 Quelques classes utiles

4.3.1 Reader et Writer

classes abstraites
close() : pour fermer le flux
read() ou read(Char[]) et write(String|Char[]) : pour les caractères !

4.3.2 InputStreamReader et OutputStreamWriter

Il existe deux très qui font la traduction des emphstreams vers les Reader/Writer :

```
InputStreamReader(InputStream) ;
OutputStreamReader(OutputStream) ;
```

4.3.3 BufferedReader et BufferedWriter

pareil que pour BufferedInputStream et BufferedOutputStream
avec en plus, pour BufferedReader, une methode readLine() (lit une ligne).

4.3.4 StringReader

permet de lire des données dans une chaîne de caractères comme si c'était un flux.
La seule manière pour lire des entrées est d'utiliser BufferReader avec readLine(), pour lire une ligne. On obtient une chaîne de caractères qui représente la ligne.On peut alors la lire en utilisant StringTokenizer (voir plus haut)

4.3.5 PrintWriter

methode en plus : print(x) ou println(x)
avec x = ce qu'on veut (a peu pres)

Les méthodes les plus courament utilisées avec ces classes sont $print$ et $println$, pour les classes dérivées de Writer, et qui envoient leurs arguments terminés par un retour à la ligne.

pour lire les données, c'est plus compliqué : le mieux est d'utilisé la classe BufferedReader, qui donne la méthode $readLine$:

```
BufferedReader in =  new BufferedReader (
        new InputStreamReader (
            new FileInputStream("toto.txt")));
```

C'est un peu compliqué, mais on peut remplacer :

```
InputStreamReader (  new FileInputStream(  "toto.txt"  ) )
```

par :

```
FileReader("toto.txt")
```

Par exemple, le code suivant permet de lire la première ligne du fichier `toto.txt` :

```
...
  String s ;
  BufferedReader in =  new BufferedReader (
        new FileReader("toto.txt")) ;
  s = in.readLine() ;
...
```

4.4 Entrées et sorties standard

Il s'agit du moyen le plus simple dont dispose un programme pour dialoguer avec un utilisateur. Pour les modéliser, on dispose de deux flux du type InputStream et $OutputStream$ qui s'appellent $System.in$ et $System.out$. Ils peuvent être utilisés exactement de la même manière que des flux provenant ou en direction d'un fichier. Pour reprendre l'exemple précédent, si on veut lire la première ligne (suite de caractères terminée par `Entrée`) tapée par l'utilisateur, on peut écrire :

```
...
   String s ;
   BufferedReader in =  new BufferedReader( new InputStreamReader(System.in)) ;
   s = in.readLine() ;
...
```

4.5 Conclusion

Le mécanisme des flux en Java est complexe, et ce cours n'est qu'une introduction très incomplète. L'exemple sur les entrées/sorties standard est assez symptomatique de ce qui attend lme programmeur Java : on veut résoudre un problème simple et très courant (lire une ligne), et il faut connaître au minimum 3 classes de $stream$ pour le résoudre correctement. Pour résoudre un problème d'entrées/sorties un peu plus complexe, il ne faudra pas hésiter à utiliser un manuel exhaustif afin de chercher une solution nette au problème posé.