Le langage JAVA

Les entrées / sorties

LES ENTRÉES / SORTIES

Dans la plupart des langages de programmation les notions d'entrées / sorties sont considérées comme une technique de base, car les manipulations de fichiers, notamment, sont très fréquentes.

En Java, et pour des raisons de sécurité, on distingue deux cas :

- le cas des applications Java autonomes, où, comme dans n'importe quel autre langage, il est généralement fait un usage important de fichiers,
- le cas des applets Java qui, ne peuvent pas, en principe, accéder, tant en écriture qu'en lecture, aux fichiers de la machine sur laquelle s'exécute le navigateur (machine cliente).

LA GESTION DES FICHIERS (1)

La gestion de fichiers proprement dite se fait par l'intermédiaire de la classe File.

Cette classe possède des méthodes qui permettent d'interroger ou d'agir sur le système de gestion de fichiers du système d'exploitation.

Un objet de la classe File peut représenter un fichier ou un répertoire.

LA GESTION DES FICHIERS (2)

Voici un aperçu de quelques constructeurs et méthodes de la classe File :

- File (String name)
- File (String path, String name)
- File (File dir, String name)
- boolean isFile( ) / boolean isDirectory( )
- boolean **mkdir**( )
- boolean **exists**( )
- boolean **delete**( )
- boolean **canWrite**( ) / boolean **canRead**( )
- File **getParentFile**( )
- long **lastModified**( )

LA GESTION DES FICHIERS (3)

```
import java.io.*;
public class ListRep
{
  public static void main(String[] args) {
    listRep(new File("."));
  }

  public static void listRep(File rep) {
    if (rep.isDirectory()) {
      //liste les fichier du répertoire
      String t[]=rep.list();
      for (int i=0;i<t.length;i++)
        System.out.println(t[i]);
   }
 }
}
```

Les objets et classes relatifs à la gestion des fichiers se trouvent dans le package java.io

A partir du chemin d'un dossier ou d'un fichier, on peut créer un objet File : ici on va lister le répertoire courant (« . »)

Les méthodes isFile() et isDirectory() permettent de déterminer si mon objet File est une fichier ou un répertoire

LA GESTION DES FICHIERS (4)

```
import java.io.*;
public class ListRep
{
  public static void main(String[] args) {
    listRep(new File("."));
  }

  public static void listRep(File rep) {
    File r2;
    if (rep.isDirectory()) {
      String t[]=rep.list();
      for (int i=0;i<t.length;i++) {
        r2=new File(rep.getAbsolutePath()+"\\"+t[i]);
        if (r2.isDirectory())
              listRep(r2);
        else
              System.out.println(r2.getAbsolutePath());
      }
    }
  }
}
```

Le nom complet du fichier est rep\fichier

Pour chaque fichier, on regarde s'il est un répertoire.

Si le fichier est un répertoire listRep s'appelle récursivement elle-même

NOTION DE FLUX (1)

Les E / S sont gérées de façon portable (selon les OS) grâce à la notion de flux (*stream* en anglais).

Un flux est en quelque sorte un canal dans lequel de l'information transite. L'ordre dans lequel l'information y est transmise est respecté.

Un flux peut être :

- Soit une source d'octets à partir de laquelle il est possible de lire de l'information. On parle de flux d'entrée.
- Soit une destination d'octets dans laquelle il est possible d'écrire de l'information. On parle de flux de sortie.

Certains flux de données peuvent être associés à des ressources qui fournissent ou reçoivent des données comme :

- les fichiers,
- les tableaux de données en mémoire,
- les lignes de communication (connexion réseau)

NOTION DE FLUX (2)

L'intérêt de la notion de flux est qu'elle permet une gestion homogène :

- quelle que soit la ressource associée au flux de données,
- quel que soit le flux (entrée ou sortie).

Certains flux peuvent être associés à des filtres

- Combinés à des flux d'entrée ou de sortie, ils permettent de traduire les données.

Les flux sont regroupés dans le paquetage **java.io**

Il existe de nombreuses classes représentant les flux

- il n'est pas toujours aisé de se repérer.

Certains types de flux agissent sur la façon dont sont traitées les données qui transitent par leur intermédiaire :

- E / S bufferisées, traduction de données, ...

Il va donc s'agir de combiner ces différents types de flux pour réaliser la gestion souhaitée pour les E / S.

FLUX D'OCTETS ET FLUX DE CARACTÈRES

Il existe des flux de bas niveau et des flux de plus haut niveau (travaillant sur des données plus évoluées que les simples octets). Citons :

- Les flux de caractères
  - classes abstraites **Reader** et **Writer** et leurs sous-classes concrètes respectives.
- Les flux d'octets
  - classes abstraites **InputStream** et **OutputStream** et leurs sous-classes concrètes respectives,

LECTURE DE FICHIER

```
import java.io.*;
public class LireLigne
{
  public static void main(String[] args)
    {
      try
       {
         FileReader fr=new FileReader("c:\\windows\\system.ini");
         BufferedReader br= new BufferedReader(fr);
         while (br.ready())
            System.out.println(br.readLine());
         br.close();
        }
     catch (Exception e)
        {System.out.println("Erreur "+e);}
    }
}
```

A partir du chemin d'un dossier ou d'un fichier, on peut créer un objet FileReader puis à partir ce celui-ci, on crée un BufferedReader

Dans l'objet BufferedReader on dispose d'une méthode readLine()

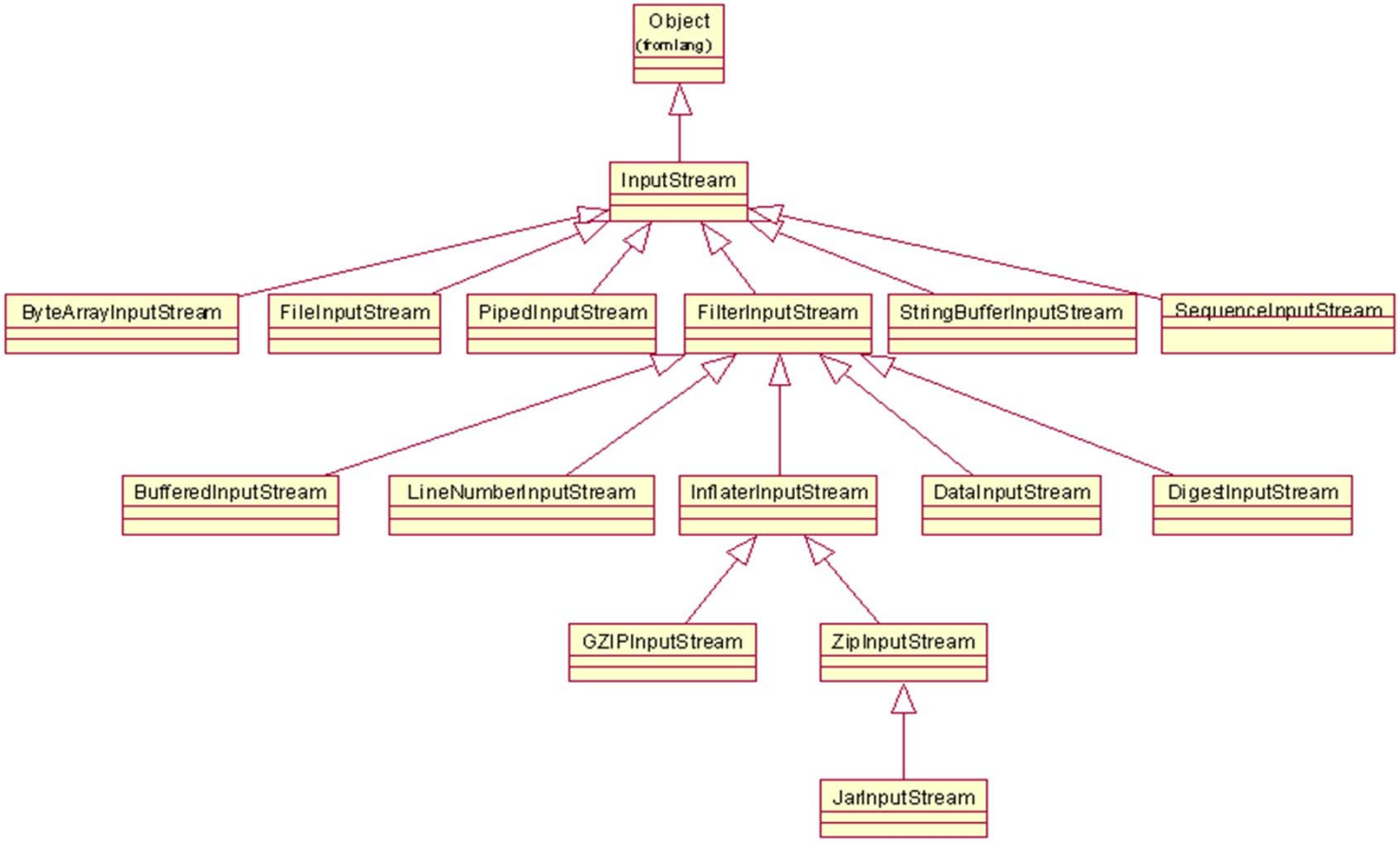
ECRITURE DANS UN FICHIER

```
import java.io.*;
public class Ecrire
{
  public static void main(String[] args)
    {
      try
       {
         FileWriter fw=new FileWriter("c:\\temp\\essai.txt");
         BufferedWriter bw= new BufferedWriter(fw);
         bw.write("Ceci est mon fichier");
         bw.newLine();
         bw.write("Il est à moi...");
         bw.close();
       }
      catch (Exception e)
       { System.out.println("Erreur "+e);}
  }
}
```

A partir du chemin d'un dossier ou d'un fichier, on peut créer un objet FileWriter puis à partir ce celui-ci, on crée un BufferedWriter

Attention, lorsque l'on a écrit, il ne faut pas oublier de fermer le fichier

LA HIÉRARCHIE DES FLUX D'OCTETS EN ENTRÉE



LES FLUX D'OCTETS : LA CLASSE INPUTSTREAM (1)

Un **InputStream** est un flux de lecture d'octets.

InputStream est une classe abstraite.

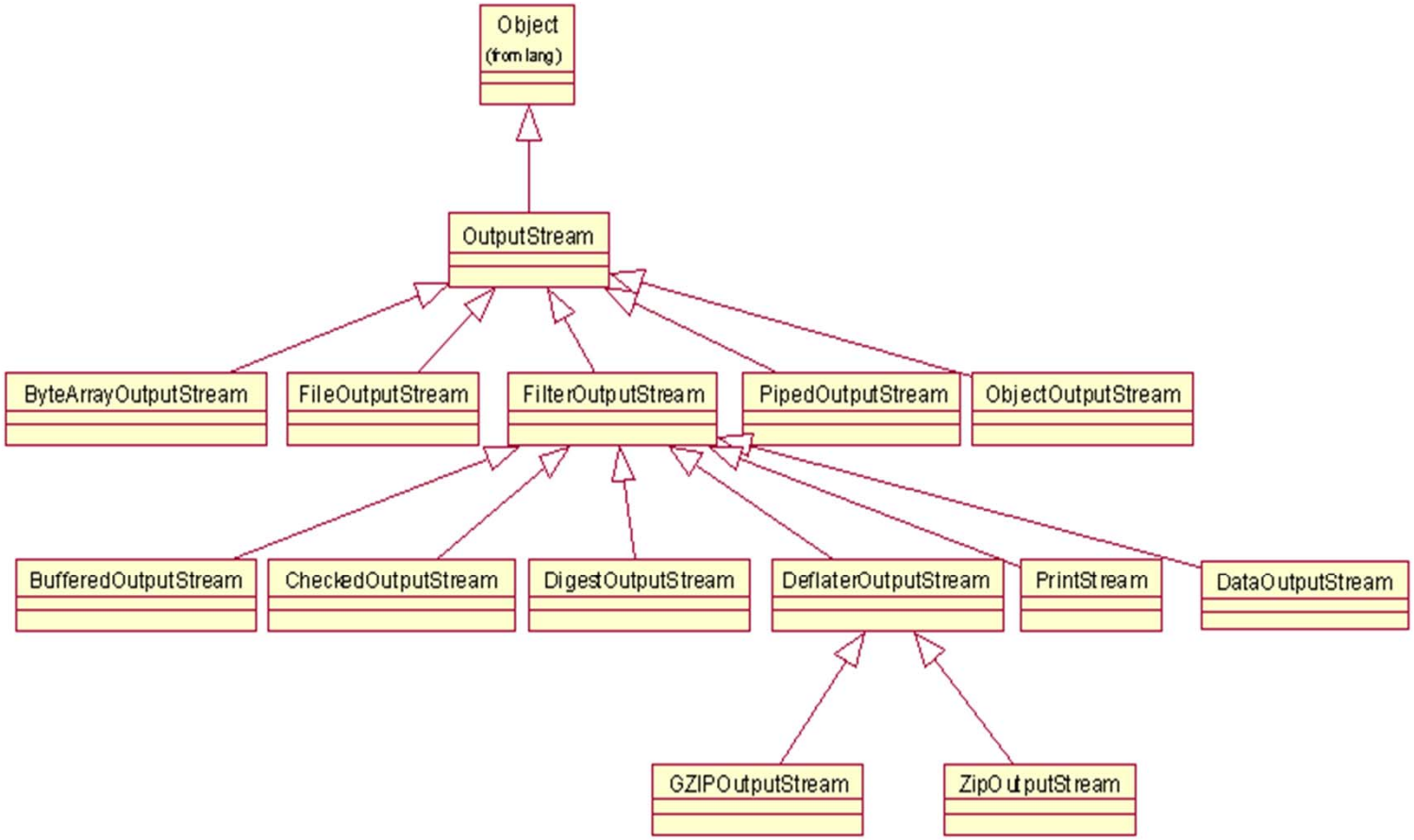
- Ses sous-classes concrètes permettent une mise en œuvre pratique.
- Par exemple, **FileInputStream** permet la lecture d'octets dans un fichier.

LA CLASSE INPUTSTREAM (2)

Les méthodes principales qui peuvent être utilisées sur un **InputStream** sont :

- **public abstract int read () throws IOException** qui retourne l'octet lu ou -1 si la fin de la source de données est atteinte. C'est cette méthode qui doit être définie dans les sous-classes concrètes et qui est utilisée par les autres méthodes définies dans la classe **InputStream**.
- **int read (byte[ ] b)** qui emplit un tableau d'octets et retourne le nombre d'octets lus
- **int read (byte [] b, int off, int len)** qui emplit un tableau d 'octets à partir d 'une position donnée et sur une longueur donnée
- **void close ()** qui permet de fermer un flux,
  - Il faut fermer les flux dès qu'on a fini de les utiliser. En effet, un flux ouvert consomme des ressources du système d'exploitation qui sont en nombre limité.
- **int available ()** qui retourne le nombre d'octets prêts à être lus dans le flux,
  - Attention : Cette fonction permet d'être sûr qu'on ne fait pas une tentative de lecture bloquante. Au moment de la lecture effective, il se peut qu'il y ait plus d'octets de disponibles.
- **long skip (long n)** qui permet d'ignorer un certain nombre d'octets en provenance du flot. Cette fonction renvoie le nombre d'octets effectivement ignorés.

LA HIÉRARCHIE DES FLUX D'OCTETS EN SORTIE



LA CLASSE OUTPUTSTREAM

Un **OutputStream** est un flot d'écriture d'octets.

La classe **OutputStream** est abstraite.

Les méthodes principales qui peuvent être utilisées sur un **OutputStream** sont :

- **public abstract void write (int) throws IOException** qui écrit l'octet passé en paramètre,
- **void write (byte[] b)** qui écrit les octets lus depuis un tableau d'octets,
- **void write (byte [] b, int off, int len)** qui écrit les octets lus depuis un tableau d 'octets à partir d 'une position donnée et sur une longueur donnée,
- **void close ()** qui permet de fermer le flux après avoir éventuellement vidé le tampon de sortie,
- **flush ()** qui permet de purger le tampon en cas d'écritures bufferisées.

LES FLUX D'OCTETS

Classe **DataInputStream**

- sous classes de InputStream permet de lire tous les types de base de Java.

Classe **DataOutputStream**

- sous classes de OutputStream permet d'écrire tous les types de base de Java.

Classes **ZipOutputStream et ZipInputStream**

- permettent de lire et d'écrire des fichiers dans le format de compression zip.

EMPILEMENT DE FLUX FILTRÉS (1)

En Java, chaque type de flux est destiné à réaliser une tâche.

Lorsque le programmeur souhaite un flux qui ait un comportement plus complexe

- Il l'"empile", à la façon des poupées russes, plusieurs flux ayant des comportements plus élémentaires.
  - On parle de flux filtrés.
- Concrètement, il s'agit de passer, dans le constructeur d'un flux, un autre flux déjà existant pour combiner leurs caractéristiques.

FileInputStream

- permet de lire depuis un fichier mais ne sait lire que des octets.

DataInputStream

- permet de combiner les octets pour fournir des méthodes de lecture de plus haut niveau (pour lire un double par exemple), mais ne sait pas lire depuis un fichier.

Une combinaison des deux permet de combiner leurs caractéristiques :

```
FileInputStream fic = new FileInputStream ("fichier");
DataInputStream din = new DataInputStream (fic);
double d = din.readDouble ();
```

EMPILEMENT DE FLUX FILTRÉS (2)

Lecture bufferisée de nombres depuis un fichier :

```
DataInputStream din = new DataInputStream(new BufferedInputStream(
                                    new FileInputStream ("monfichier")));
```

Lecture de nombre dans un fichier au format zip :

```
ZipInputStream zin = new ZipInputStream (
                                    new FileInputStream ("monfichier.zip"));
DataInputStream din = new DataInputStream (zin);
```

FLUX DE FICHIERS À ACCÈS DIRECT

La classe **RandomAccessFile**

- permet de lire ou d'écrire dans un fichier à n'importe quel emplacement (par opposition aux fichiers à accès séquentiels).

Elle implémente les interfaces **DataInput** et **DataOutput**

- permettent de lire ou d'écrire tous les types Java de base, les lignes, les chaînes de caractères ascii ou unicode, etc ...

Un fichier à accès direct peut être :

- ouvert en lecture seule (option `"r"`) ou
- en lecture / écriture (option `"rw"`).

Ces fichiers possèdent un pointeur de fichier qui indique constamment la donnée suivante.

- La position de ce pointeur est donnée par **long getFilePointer ()** et celui-ci peut être déplacé à une position donnée grâce à **seek (long off)**.

LES FLUX DE CARACTÈRES (1)

Ce sont des sous-classes de **Reader** et **Writer**.

Ces flux utilisent le codage de caractères Unicode.

Exemples

- conversion des caractères saisis au clavier en caractères dans le codage par défaut

```
InputStreamReader in = new InputStreamReader (System.in);
```

- Conversion des caractères d'un fichier avec un codage explicitement indiqué :

```
InputStreamReader in = new InputStreamReader (
    new FileInputStream ("chinois.txt"), "ISO2022CN");
```

LES FLUX DE CARACTÈRES (2)

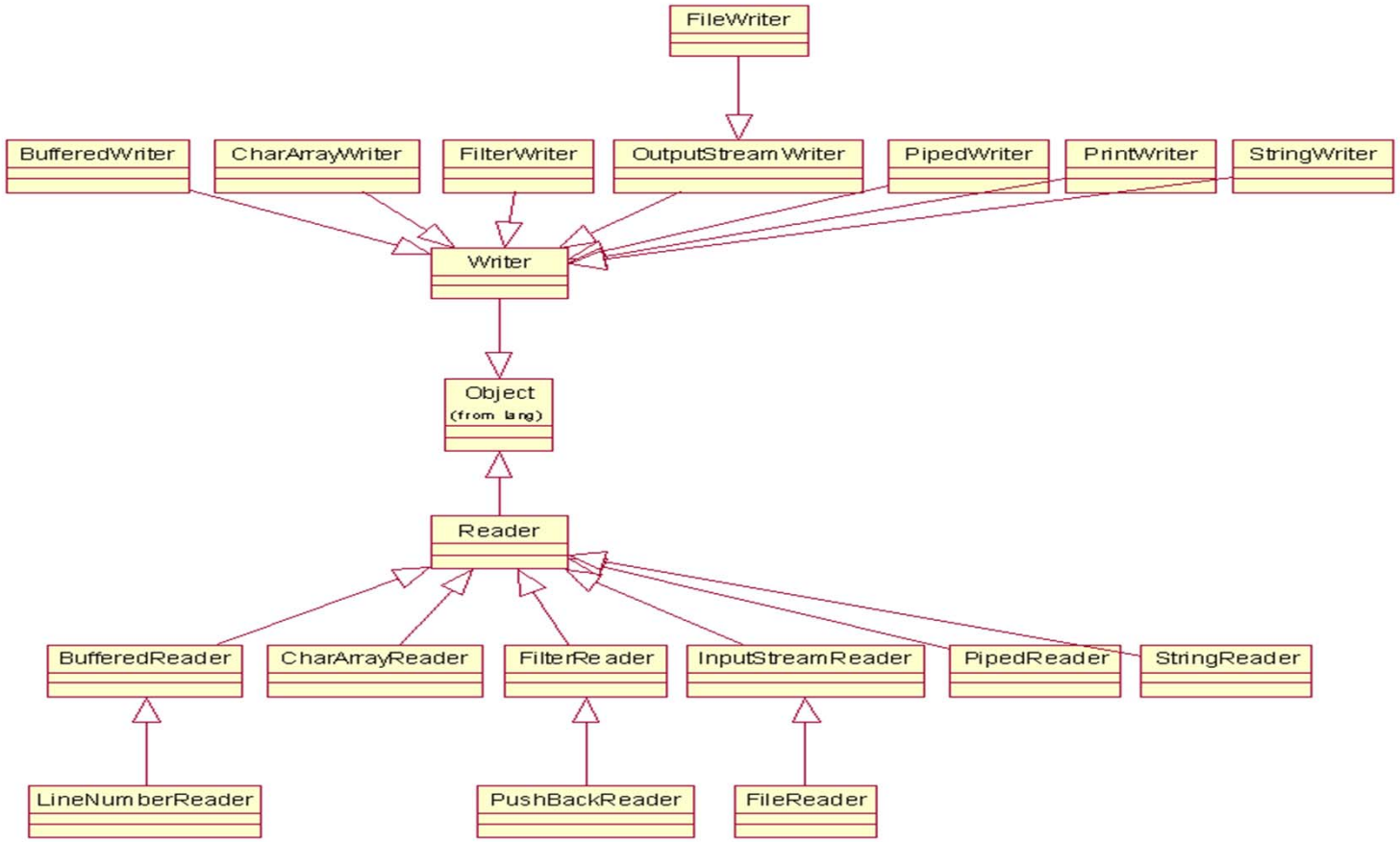
Pour écrire des chaînes de caractères et des nombres sous forme de texte

- on utilise la classe **PrintWriter** qui possède un certain nombre de méthodes **print (...)** et **println (...)**.

Pour lire des chaînes de caractères sous forme de texte, il faut utiliser, par exemple :

- **BufferedReader** qui possède une méthode **readLine()** .
  - Pour la lecture de nombres sous forme de texte, il n'existe pas de solution toute faite : il faut par exemple passer par des chaînes de caractères et les convertir en nombres.

LA HIÉRARCHIE DES FLUX DE CARACTÈRES



LES FLUX DE DONNÉES PRÉDÉFINIS (1)

Il existe 3 flux prédéfinis :

- L'entrée standard **System.in** (instance de InputStream)
- La sortie standard **System.out** (instance de PrintStream)
- La sortie standard d'erreurs **System.err**(instance de PrintStream)

```
try {
    int c;
    while((c = System.in.read()) != -1) {
        System.out.print(c);
    }
} catch (IOException e) {
    System.out.print(e);
}
```

LES FLUX DE DONNÉES PRÉDÉFINIS (2)

La classe **InputStream** ne propose que des méthodes élémentaires. Préférez la classe **BufferedReader**.qui permet de récupérer des chaînes de caractères.

```
try {
        Reader reader = new InputStreamReader(System.in);
        BufferedReader keyboard = new BufferedReader(reader);

        System.out.print("Entrez une ligne de texte : ");
        String line = keyboard.readLine();
        System.out.println("Vous avez saisi : " + line);
   }
catch(IOException e) {
        System.out.print(e);
}
```

LES FLUX DE DONNÉES PRÉDÉFINIS (3)

L'utilisation de flux "bufferisés" permet d'améliorer considérablement les performances :

```
import java.io.*;

public class TestVitesseFlux {
  public static void main(String[] args) {
    FileInputStream fis; BufferedInputStream bis;
    try {
      fis = new FileInputStream(new File("test.txt"));
      bis = new BufferedInputStream(new FileInputStream(new File("test.txt")));
      byte[] buf = new byte[8];
      long startTime = System.currentTimeMillis();
      while(fis.read(buf) != -1);
      System.out.println("Temps de lecture avec FileInputStream : " +
                         (System.currentTimeMillis() - startTime));
      startTime = System.currentTimeMillis();
      while(bis.read(buf) != -1);
      System.out.println("Temps de lecture avec BufferedInputStream : " +
                         (System.currentTimeMillis() - startTime));
      fis.close();
      bis.close();
    }
    catch (FileNotFoundException e) { e.printStackTrace(); }
    catch (IOException e) { e.printStackTrace(); }
  }
}
```

LA SÉRIALISATION

- La sérialisation consiste à prendre un objet en mémoire et à en sauvegarder l'état sur un flux de données (vers un fichier, par exemple).
- Ce concept permet aussi de reconstruire, ultérieurement, l'objet en mémoire à l'identique de ce qu'il pouvait être initialement.
- La sérialisation peut donc être considérée comme une forme de persistance des données.
- 2 classes ObjectInputStream et ObjectOutputStream proposent, respectivement, les méthodes readObject et writeObject
- Attention :
    - Par défaut, les classes ne permettent pas de sauvegarder l'état d'un objet sur un flux de données. Il faut implémenter l'interface java.io.Serializable.
    - Il faut que la classe n'ait pas supprimé le constructeur par défaut.

EXEMPLE DE SÉRIALISATION

```
import java.io.*;
public class TestSerialisation implements Serializable {

  private static final long serialVersionUID = 1L;

  public void sauvegarde(String s) {
    try  {
      FileOutputStream f = new FileOutputStream(new File(s));
      ObjectOutputStream oos = new ObjectOutputStream(f);
      oos.writeObject(this);
      oos.close();
    }
    catch (Exception e) { System.out.println("Erreur "+e); }
  }

  static Object relecture(String s) {
    try  {
      FileInputStream f = new FileInputStream(new File(s));
      ObjectInputStream oos = new ObjectInputStream(f);
      Object o=oos.readObject();
      oos.close();
      return o;
    }
    catch (Exception e) {
      System.out.println("Erreur "+e);
      return null;
    }
  }
}
```

CLASS SCANNER (1)

- Une classe très utile du JDK est Scanner (depuis la version 5 de Java) avec des fonctionnalités très intéressantes pour parser des chaînes de caractères, et en extraire et convertir les composants.
- Un Scanner peut se brancher sur à peu près n'importe quelle source : InputStream, Readable (et donc Reader), File... et bien sûr une simple String.
- Ensuite on utilise les méthodes de type hasNext...(), ou next...(), ou alors les méthodes de type find...() , match() et group().

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

- Méthode hasNext() / next() :

1. Découper la chaîne de caractères en *tokens* grâce à un délimiteur ; il s'agit par défaut d'un caractère "blanc" (espace, tabulation, retour à la ligne...), mais il est évidemment possible de fournir sa propre expression via la méthode useDelimiter(expression).
2. Utiliser ces méthodes pour parcourir, récupérer et convertir ces tokens.

- Les méthodes de type hasNext...() (hasNextInt(), hasNextFloat()...) fonctionnent sur le même principe qu'un Iterator.

CLASS SCANNER (2)

- A l'aide de ces méthodes, il est très facile de parser une chaîne dont vous maîtrisez parfaitement le format, par exemple un fichier .csv :

```
String s = "Dalton;Joe;1.4\n" +
           "Dalton;Jack;1.6\n" +
           "Dalton;William;1.8\n" +
           "Dalton;Averell;2.0";
Scanner scan = new Scanner(s);
scan.useDelimiter(";|\n");
scan.useLocale(Locale.US); // Pour les floats
 while(scan.hasNextLine()) {
    System.out.printf("%2$s %1$s : %3$.1f m %n", scan.next(), scan.next(),
    scan.nextFloat());
}
```