Entrées / Sorties séquentielles

Introduction

Entrées/Sorties en Java
- ➔ package **java.io**
- ➔ vision unifiée des entrées/sorties sous forme de **flots** (ou flux).

Flots : Canaux de transmission de données à partir d'une source ou vers une destination.

Deux sortes des flots :
- ➔D**'entrée** : lecture séquentielle
- ➔De **sortie** : écriture séquentielle

Intérêt : uniformité
- ➔on utilise les même opérations pour lire ou écrire
  - ➔dans un fichier,
  - ➔dans entrees/sorties standard,
  - ➔via le réseau
  - ➔etc…

Inconvénient : au départ ça paraît un peu compliqué…

Introduction

Deux sortes de flots d'entrées/sorties selon l'unité d'information qu'ils permettent de transporter :

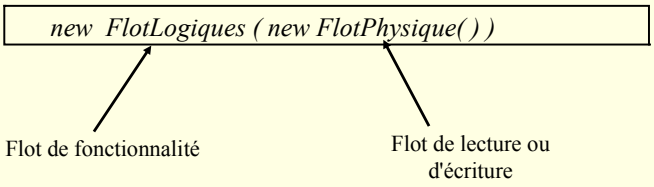
• les flots de binaires (JDK1.0)
qui transportent des octets (binaire).

• les flots de caractères (JDK1.1)
qui transportent des caractères en tenant compte éventuellement d'un codage externe.

Deux types de fonctionnalités de flots :
• **flots physiques** (fichier, mémoire, tube, socket), et
• **flots logiques**, filtres, qui permettent de rajouter les fonctionnalités au-dessus d'un flot physique.

Introduction

Assemblage : un flot d'entrée/sortie sera construit à partir d'un flot physique sur lequel on aura éventuellement *"empilé"* un ou plusieurs flots logiques.



Quatre catégories de flots

Quatre catégories de flots :

	flot d'entrée	flot de sortie
flot d'octets, binaires	**InputStream**	**OutputStream**
flot de caractères	**Reader**	**Writer**

Les flots d'octets (binaires)

Les flots physiques : **InputStream** / **OutputStream**

L'information de base qui est manipulée est l'octet (binaire).

Si la source ou destination est un fichier :
FileInputStream / **FileOutputStream**

Les principaux constructeurs sont :

- FileInputStream(String name)
- FileOutputStream(String name)

Les flots d'octets

Les flots logiques ou filtres : **FilterInputStream** / **FilterOutputStream**

On trouve entre autres filtres (sous-classes des filtres) :

- **BufferedInputStream** / **BufferedOutputStream**

Permet d'ajouter un tampon afin d'optimiser les entrées/sorties.

Les principaux constructeurs sont :
- BufferedInputStream(InputStream in)
- BufferedOutputStream(OutputStream out)

Les flots d'octets

• **DataInputStream** / **DataOutputStream**

Pour faire des entrées/sorties en format binaire des données. Les principaux constructeurs sont :

- DataInputStream(InputStream in)
- DataOutputStream(OutputStream out)

• **PrintStream**

Pour faire des sorties en format texte en utilisant le codage local des caractères. Les principaux constructeurs sont :

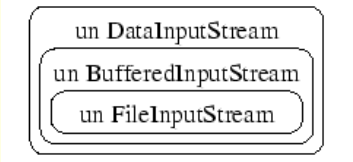
- PrintStream(OutputStream out)
- PrintStream(OutputStream out, boolean autoFlush)

Les flots d'octets

Assemblage de flots

On peut obtenir le flot (la fonctionnalité) des données souhaité en *"empilant"* un ou plusieurs filtres sur un flot physique.

```
new DataInputStream (
    new BufferedInputStream (
          new FileInputStream ("nomFichier.binaire") ) )
```



Les flots d'octets

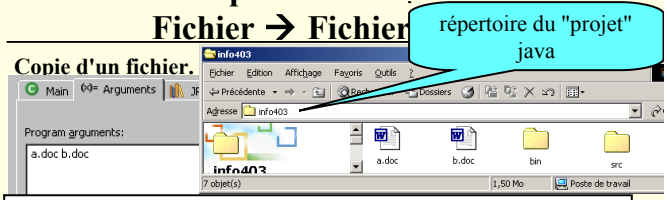
La classe **java.lang.System** définit trois variables statiques qui correspondent aux **flots d'entrée/sorties prédéfinis** :
• Entrée standard : System.in (*static final InputStream in*)
• Sortie standard : System.out (*static final PrintStream out*)
• Sortie d'erreur standard : System.err (*static final PrintStream err*)

Notez bien qu'il s'agit de flots physiques d'octets et non pas de caractères.

Méthodes permettant le changement des flot standards :
- **System.setIn (InputStream in)**
- **System.setOut (PrintStream out)**
- **System.setErr (PrintStream err)**

Exemple flots d'octets : Fichier → Fichier

Copie d'un fichier.




```
package testDiversEntreesSorties;
import java.io.*;

public class Copie {
 public static void main(String[] args) {
    // Test sur le nombre de paramètres passés
    if (args.length != 2) {
       System.out.println("deux arguments !!");
       System.exit(0);
       }
    try {
```


Exemple flots d'octets : Fichier → Fichier (2/3)

Suite copie d'un fichier

public FileInputStream(String name) throws FileNotFoundException

```
// Préparation du flux d'entrée
FileInputStream fis = new FileInputStream(args[0]);
BufferedInputStream bis=new BufferedInputStream(fis);

// Préparation du flux de sortie
FileOutputStream fos = new FileOutputStream(args[1]);
BufferedOutputStream bos = new
                           BufferedOutputStream(fos);
```

Exemple flots d'octets : Fichier → Fichier (3/3)

Suite copie d'un fichier

public int read() throws IOException
Returns: the next byte of data, or -1 if the end of the stream is reached.

```
    // Copie des octets
    int octet;
    while ((octet=bis.read()) > -1 ) {
        bos.write(octet);
        }
    // Fermeture des flux de données
     bos.flush();
     bos.close();
     bis.close();
     } catch (Exception e) {
      System.err.println("Erreur d'acces au fichier");
      e.printStackTrace();
      }
     System.out.println("Copie terminée");
    } // fin main
} // fin classe Copie
```


Exemple flots d'octets: Données → Fichier

Exemple de sauvegarde de données (variables du programme) :

public final void writeInt(int v) throws IOException
Writes an int to the underlying output stream as four bytes.

```
public class DonnéesVersFichier {
public static void main(String[] args) {
   File f = new File("donnees.dat");
   FileOutputStream fos;
   try {
    fos = new FileOutputStream(f);
    BufferedOutputStream bos =
                    new BufferedOutputStream(fos);
    DataOutputStream dos = new DataOutputStream(bos);
    int a = 10;        dos.writeInt(a);
    short s = 3;       dos.writeShort(s);
    boolean b = true; dos.writeBoolean(b);
    dos.close();
    } catch (IOException e) {
     System.err.println("Malaise :" + e.getMessage());
}  }  }
```


Exemple flots d'octets : Fichier → Données

Exemple de récuperation de données :

```
public class FichierVersDonnées {
   public static void main(String[] args) {
   File f = new File("donnees.dat");
   FileInputStream fis;
   try {
    fis = new FileInputStream(f);
    BufferedInputStream bis =
                    new BufferedInputStream(fis);
    DataInputStream dis = new DataInputStream(bis);
    int a = dis.readInt();
    short s = dis.readShort();
    boolean b = dis.readBoolean();
    System.out.println("a = "+a+" s = "+s+" b = "+b);
    dis.close();
    } catch (IOException e) {
     System.err.println("Malaise :" + e.getMessage());
}  }  }
```

```
Console [<terminated> C:\Progra
a = 10 s = 3 b = true
```


Les flots de caractères

Les flots de caractères ont l'intérêt de pouvoir s'adapter à un codage.

En interne, les char de Java sont codés sur deux octets en utilisant Unicode.

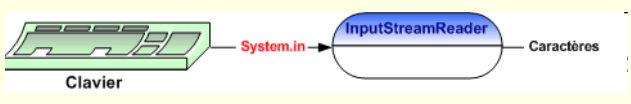
Les classes **Reader** et **Writer** permettent de gérer des flots de caractères.

InputStreamReader, sous-classe de **Reader**, permet de voir un **InputStream** en tant que **Reader** : les octets sont lus et transformés en caractères en accord avec le codage utilisé (Unicode par exemple).

Les flots de caractères

Par exemple, pour lire les caractères de l'entrée standard on fera :

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



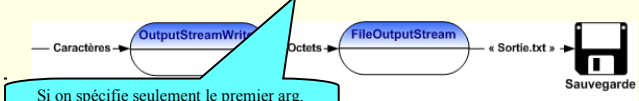
Les flots de caractères

La classe **OutputStreamWriter,** sous-classe de Writer, à un rôle symétrique à **InputStreamReader**.

Elle permet de voir un OutputStream en tant que Writer : les caractères sont transformés (en accord avec le codage utilisé) en octets.

Par exemple pour écrire dans le fichier "iso.txt" un texte encodé en ISO8859_1 on fera :

```
Writer f = new OutputStreamWriter(
                    new FileOutputStream("sortie.txt"),
                    "ISO8859_1") ;
```



Si on spécifie seulement le premier arg, alors l'encodage utilisé est celui par défaut

Quatre grandes catégories de flots

	flot d'entrée	flot de sortie
flot d'octets	**InputStream** File... Data... Buffered... Object...	**OutputStream** File... Data... Buffered... Object... PrintStream
flot de caractères	**Reader** File... Buffered... InputStreamReader	**Writer** File... Buffered... PrintWriter OutputStreamWriter

InputStreamReader

OutputStreamWriter

Exemple flot caractères : Clavier → Fichier (1/2)

Exemple clavier ---> Fichier texte (caractères)

```
public class ClavierVersFichier {
public static void main(String[] args) {
   BufferedReader br;
   BufferedWriter bw;
   try {
     System.out.println("Commencer : " );
     br = new BufferedReader(
                new InputStreamReader(System.in));
     bw = new BufferedWriter(
                new FileWriter("sortie.txt"));
     String ligne;
     while (! (ligne = br.readLine()).equals("")) {

         bw.wr    (ligne); bw.newLine();
       }
```

Autre possibilité avec : **PrintWriter**

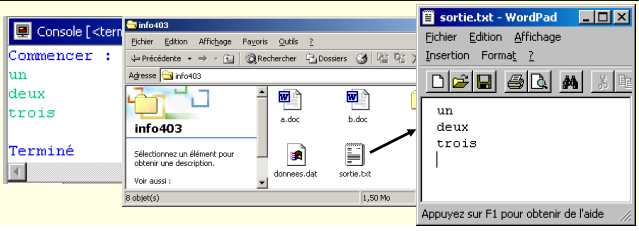
Avec **PrintWriter**, *println(ligne)*

Choix arbitraire : chaîne vide signale la fin

Exemple flot caractères : Clavier → Fichier (2/2)

suite clavier ---> Fichier texte

```
      bw.close();
      System.out.println("Terminé" );
      } catch (IOException e) {
       System.err.println("Malaise :" + e.getMessage());
       }
   }
}
```



Exemple flot caractères : Fichier → Console (1/2)

Exemple Fichier texte (caractères) ---> Console

```
public class FichierVersConsole {
 public static void main(String[] args) {
   BufferedReader br;
   BufferedWriter bw;
   File f = new File("sortie.txt");
   if (f.exists()) {
    try {
    System.out.println("Voici le fichier : " );
    br = new BufferedReader(new FileReader(f)) ;
    bw = new BufferedWriter(
                new OutputStreamWriter(System.out));
    String ligne;
    while ((ligne = br.readLine()) != null) {
      bw.write(ligne); bw.newLine();
      bw.flush(); // necessaire pour afficher
      }
```

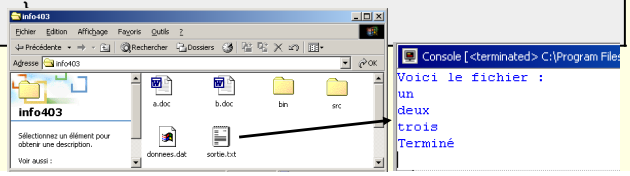
Autre possibilité avec : **PrintWriter**

bw.println(line); avec **PrintWriter**

Exemple flot caractères : Fichier → Console (2/2)

suite exemple Fichier texte ---> Console

```
      br.close();
      System.out.println("Terminé" );
      }
    catch (IOException e) {
      System.err.println("Malaise :" + e.getMessage());
      }
    }
   else
      System.out.println("Le fichier n'existe pas" );
  }
}
```



Sérialisation

La ***sérialisation*** consiste à écrire des données présentes en mémoire vers un flot de données binaires.

Ce procédé permet de rendre les objets persistants.

Interface nécessaire à la sérialisation :
- → interface Serializable

Classes nécessaires à la sérialisation :
- → classe ObjectOutputStream
- → classe ObjectInputStream

Sérialisation

Écriture (*sérialisation*) d'un objet : par défaut le système sauvegarde
- ➔la classe de l'objet et tous les attributs sauf
  - ➔les attributs *static* (car ils sont sur la classe) et
  - ➔les attributs *transient*
- ➔si un attribut contient un objet, il est aussi sauvegardé !

interdiction de sérialisation

Lecture (*désérialisation*) d'un objet : similaire à l'écriture dans le sens inverse

Contrainte :
- ➔l'objet doit être *sérialisable* : i.e. implanter **Serializable**
- ➔tous les attributs à sauver doivent être
  - ➔de type primitif (boolean, int, float, double, …)
  - ➔ou sérialisables (et implanter aussi Serializable)

Sérialisation

Le *serialVersionUID* est un "numéro de version", associé à toute classe implémentant l'interface Serializable.

Il permet de s'assurer, lors de la désérialisation, que les versions des classes Java soient cohérente. Si les versions ne sont pas cohérentes, alors une InvalidClassException est levée.

Une classe sérialisable peut déclarer explicitement son *serialVersionUID* en déclarant un attribut nommé "serialVersionUID".

Exemple :

```
private static final long serialVersionUID = 42L;
```

Calculé par défaut s'il est omis.
Depuis Java 5 un warning invite à le définir explicitement

Sérialisation

ObjectOutputStream et **ObjectInputStream**

Constructeurs (respectivement):
- **ObjectOutputStream** (**OutputStream** in) et
- **ObjectInputStream** (**InputStream** in)

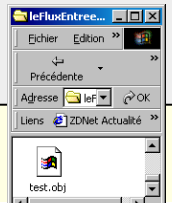
Méthodes (respectivement):
- **void writeObject(Object obj)** écriture d'un objet sérialisable
- **Object readObject( )** lecture d'un objet sérialisable

En plus, des méthodes existent pour écrire ou lire tous les types primitifs : **writeChar**, **readChar**, **writeInt**, **readInt**, etc

Exemple Sérialisation : Objet → Fichier (1/2)

Exemple sérialisation Tableau de Personne ---> Fichier

```
public class ObjetVersFichier {
 public static void main(String[] args)
                                  throws IOException {
   OutputStream out = new FileOutputStream("test.obj");
   ObjectOutputStream oos = new ObjectOutputStream(out);;
   Personne p1= new Personne("Jean","22 rue la poste");
   Personne p2= new Personne("Sylvie", "41 av Foret");
   Personne[] tab = {p1,p2} ;
   oos.writeObject(tab);
   oos.close();
  }
}
```



Exemple Sérialisation : Objet → Fichier (2/2)

Exemple sérialisation Tableau de Personne ---> Fichier

```
public class Personne implements Serializable {
   private String nom;
   private String adresse;
   static private final long serialVersionUID = 6L;
   public Personne(String nom, String adresse) {
     this.nom = nom;
     this.adresse = adresse;
   }
   public String toString(){
     return nom + " - " + adresse;
   }
}
```

S'il est omis il y aura un warning

Le mot clé *transient* permet d'interdire la sérialisation d'un attribut.
Exemple :

```
   transient private String motPasse;
```


Exemple Sérialisation : Fichier → Objet

Exemple sérialisation Fichier ---> Tableau de Personne

```
public class FichierVersObjet {
  public static void main(String[] args)
                                  throws IOException {
   InputStream in = new FileInputStream("test.obj");
   ObjectInputStream ois = new ObjectInputStream(in);
   Personne[] tab = null;
   try { tab = (Personne[])ois.readObject();
         ois.close();
       }
   catch (ClassNotFoundException e) {
     System.err.println("Type d'objet inconnu");
     }
   for (int i=0; i<tab.length; i++)
          System.out.println(tab[i].toString());
   }
}
```

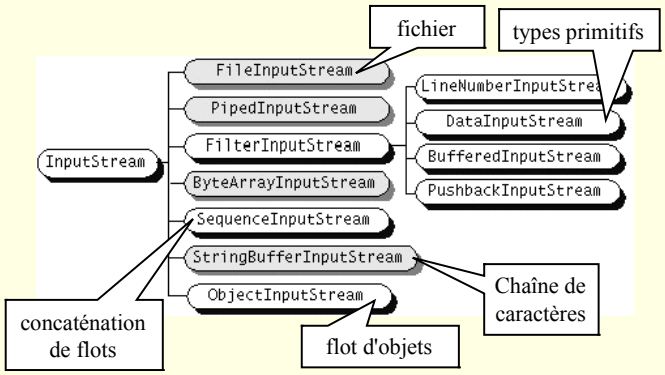
```
Console [<terminated> C:\Program Files\Java\jre
Jean - 22 rue la poste
Sylvie - 41 av Foret
```

Système de fichiers : java.io.File

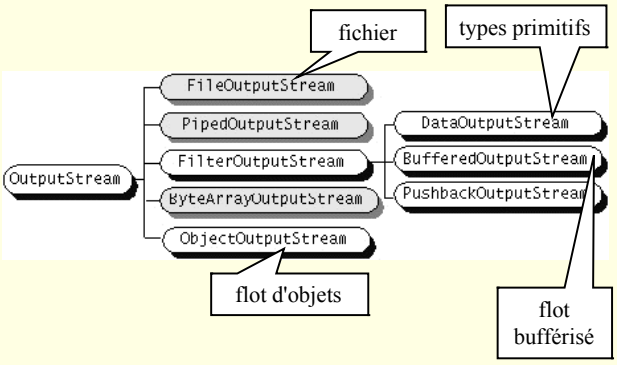
Quelques constructeurs
- **File**(String pathname)
- **File**(File parent, String child)
- **File**(URI uri)

Quelques méthodes
- int **length**( )
- boolean **exists**( )
- boolean **canRead**( )
- boolean **canWrite**( )
- boolean **isDirectory**( )
- boolean **isFile**( )
- boolean **createNewFile**( )
- boolean **delete**( )
- boolean **mkdir**( )
- boolean **renameTo**(File dest)
- String **getAbsolutePath**( )
- URI **toURI**( )
- String **getName**( )
- String **getParent**( )
- String[ ] **list**( )
- File[ ] **listFiles**( )

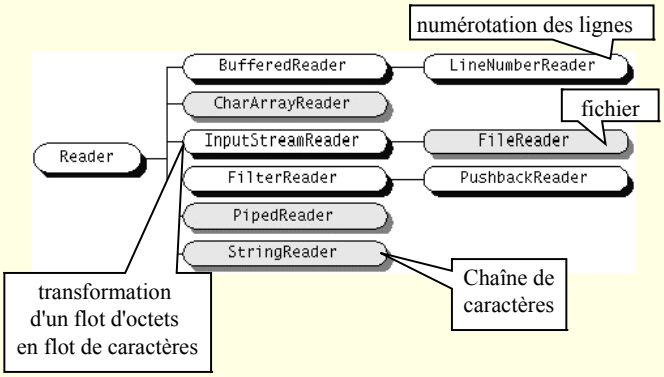
Les sous-classes de **InputStream**



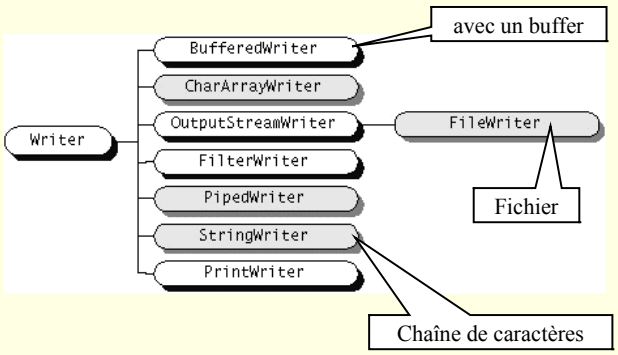
Les sous-classes de **OutputStream**



Les sous-classes de **Reader**



Les sous-classes de **Writer**



Méthodes importantes

Méthodes pour java.io.InputStream :

- void **close**() fermeture du flot.
- int **available**() nombre d'octets disponibles sur le flot.

Lecture
- abstract int **read**() lecture d'un octet.
- int **read**(byte[] b) utilisation d'un tableau d'octets comme buffer.
- int **read**(byte[] b, int off, int len) idem.

Déplacement
- long **skip**(long n) on saute n octets.

Repositionnement
- boolean **markSupported**() si les méthodes mark et reset sont utilisables.
- void **mark**(int readlimit) mémorisation de la position actuelle.
- void **reset**() on revient à la position mémorisée avec mark.

Méthodes importantes

Méthodes pour java.io.Reader :

- void **close**() fermeture du flot.
- boolean **ready**() indique si le flot est prêt pour une lecture.

Lecture
- abstract int **read**() lecture d'un caractère.
- int **read**(char[] cbuf) utilisation d'un tableau de caractères comme buffer.
- int **read**(char[] cbuf, int off, int len) idem.

Déplacement
- long **skip**(long n) on saute n caractères.

Repositionnement
- boolean **markSupported**() si les méthodes mark et reset sont utilisables.
- void **mark**(int readlimit) mémorisation de la position actuelle.
- void **reset**() on revient à la position mémorisée avec mark.

Méthodes importantes

Méthodes pour java.io.OutputStream :

- void **close**() fermeture du flot.
- boolean **flush**() force l'écriture des octets (utile s'il y a un buffer).

Écriture
- abstract void **write**(int b) écriture d'un octet.
- void **write**(byte[] b) utilisation d'un tableau d'octets.
- void **write**(byte[] b, int off, int len) idem.

Méthodes pour java.io.Writer :

- void **close**() fermeture du flot.
- boolean **flush**() force l'écriture des caractères (utile si il y a un buffer).

Écriture
- abstract void **write**(int b) écriture d'un caractère.
- void **write**(char[] cbuf) utilisation d'un tableau de caractères.
- void **write**(char[] cbuf, int off, int len) idem.
- void **write**(String str) utilisation d'une chaîne de caractères.
- void **write**(String str, int off, int len) idem.

Méthodes importantes

Méthodes pour java.io.PrintStream et java.io.PrintWriter :

- **PrintStream(OutputStream** out)
- **PrintStream(OutputStream** out, boolean autoFlush, String encoding)
- **PrintWriter**(OutputStream out)
- **PrintWriter**(OutputStream out, boolean autoFlush)
- **PrintWriter**(Writer out)
- **PrintWriter**(Writer out, boolean autoFlush)

Écriture
- void **print**(boolean b)
- void **println**(boolean b)
- void **print**(char c)
- void **println** (char c)
- …
- void **print**(String s)
- void **println**(String s)
- void **print**(Object obj)
- void **println** (Object obj)

Méthodes importantes

Méthodes pour BufferedReader et BufferedWriter :

java.io.BufferedReader
- **BufferedReader**(Reader in)
- **BufferedReader**(Reader in, int sz)
- String **readLine**() Lire une ligne

java.io.BufferedWriter
- **BufferedWriter**(Writer out)
- **BufferedWriter**(Writer out, int sz)
- void **newLine**() Ecrire une ligne.