Prog. Concurrentielle et Interfaces Interactives

Entrées/Sorties
I/O, Serialization

frederic.vernier@u-psud.fr

Les transparents qui suivent sont inspirés du cours de

- Anastasia Bezerianos (Univ. Paris-Sud)
- Rémi Forax (Univ. Marne la Vallée)

⊥

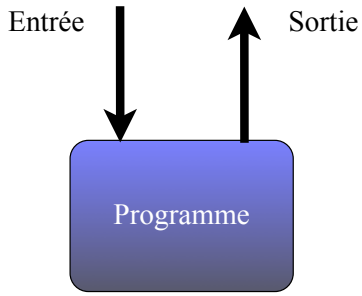
Admin

- **TD** les 3/2, 10/2, 17/2
- **TP** de 7/2, 14/2, 21/2 seront **notés**
- TP de cette semaine (7/2) decalé à 15:00 (JPO)
- Changement des groupes (vite!)

Entée/Sortie

- Entrées / Sorties et flux
- Les différents types de flux (s.18)
- La classe File (s.30)
- Lire et Ecrire ses objets (sérialisation) (s.33)

Entée/Sortie



Entrée et Sortie standard

Sortie standard (principalement la console):

– Normal
`System.out.println(…)`

– Erreur
`System.err.println(…)`

`System.out` et `System.err` sont des attributs de la classe **System** de type *PrintStream* (attend flux ***Character*** qu'il converti en octet/**Byte**)

Entrée standard (principalement le clavier):

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

`System.in` est un attribut de type *InputStream* (= flux ***Byte***) de la classe **System**. InputStreamReader permet de l'utiliser comme un flux ***Character***.

Exemple : lire une ligne sur l'entrée standard

```
import java.io.*;

class InOut {
    public static void main(String args[]) throws
  IOException {
        BufferedReader entreeClavier = new BufferedReader(
                    new InputStreamReader(System.in));
        System.out.println ("Saisissez une phrase :");
        String saisie = entreeClavier.readLine();
        System.out.println ("Merci !");
        System.out.println ("Votre phrase est : ");
        System.out.println (saisie);
    }
}
```


Formatter la sortie

Personnaliser le **format** de sortie :

– une alternative aux méthodes print et println.
– L'equivalent du *printf.*
– Pour plus de détail voir la javadoc Formatter

```
System.out.println("Pi vaut " + Math.PI);
> Pi vaut 3.141592653589793

System.out.format("Pi vaut approximativement %.2f%n",Math.PI);
> Pi vaut approximativement 3,14
```

On peut être même plus spécifique DecimalFormat de **java.text.***

Objectifs des I/O

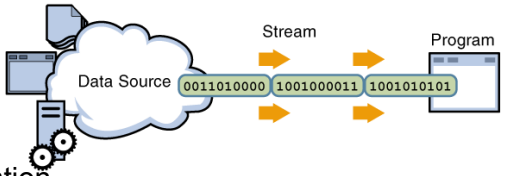
- Communiquer avec le monde extérieur :
  - **qui/quoi** ?
    - fichier sur le disque dur local, console, connexion réseau, ...
  - **comment** ?
    - en binaire, en octet (byte), en caractère,
    - ligne par ligne, par mot, par caractère,
    - séquentiellement, random-acces,
    - buffer.

Objectifs des I/O

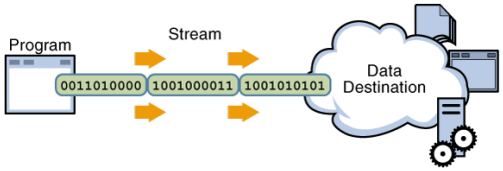
- Communiquer avec le monde extérieur :
  - qui/quoi ?
  - comment ?
- Le concept de base est celui de flux : ***stream***
  - La machine virtuelle se connecte où crée un flux de données.
  - Suivant le type de communication et les besoins, on emboîte les adaptateurs (des objets) et cache les détailes du périphérique E/S.
- A l'origine, l'unité de base d'un flux était l'octet (byte). (Depuis Java 1.4, il existe *java.nio* orienté caractère)

Flux d'entrée / Flux de sortie

Entrée
Ex : lire un fichier
Nous sommes la destination



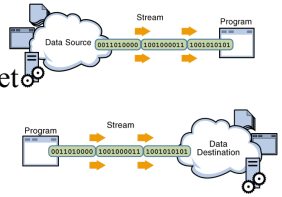
Sortie
Ex : écrire un fichier
Nous sommes la source



Le package **java.io**

- Le package *java.io* regroupe les classes pour les E/S
- Il se divise en 2 catégories
  - entrées et sorties
  - via 2 classes <u>abstraites</u> par catégorie (octet & char)

I/O : unité de base octet
mais on peut le traiter
comme char



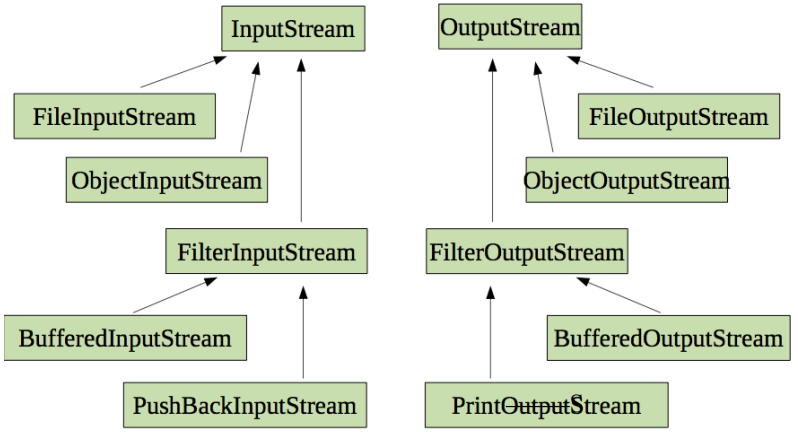
Le package **java.io** (2)

- Les entrées (classes avec méthode ***read()*** *ou* ***read (X)***) :
  - *InputStream* avec la méthode
    - *public int read(**byte**[] b)* throws IOException
  - *Reader* avec la méthode
    - *public int read(**char**[] cbuf)* throws IOException
- N'oublions pas la méthode ***close()***.

Le package **java.io** (3)

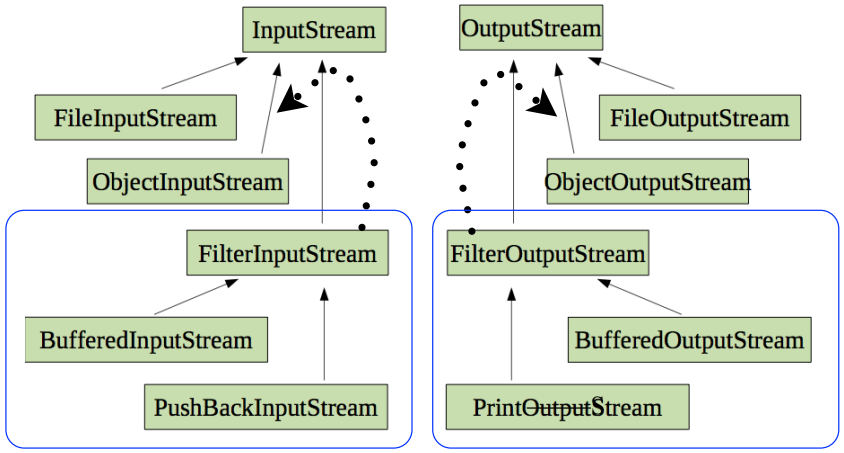
- Les sorties (classes avec méthode ***write()***) :
  - *OutputStream* avec la méthode
    - *public void write(**byte**[] b)* throws IOException
  - *Writer* avec la méthode
    - *public void write(**char**[] cbuf)* throws IOException
    - avec la méthode ***append()*** en plus
- N'oublions pas la méthode ***close()***.

E/S par Octet (byte - 8bit)



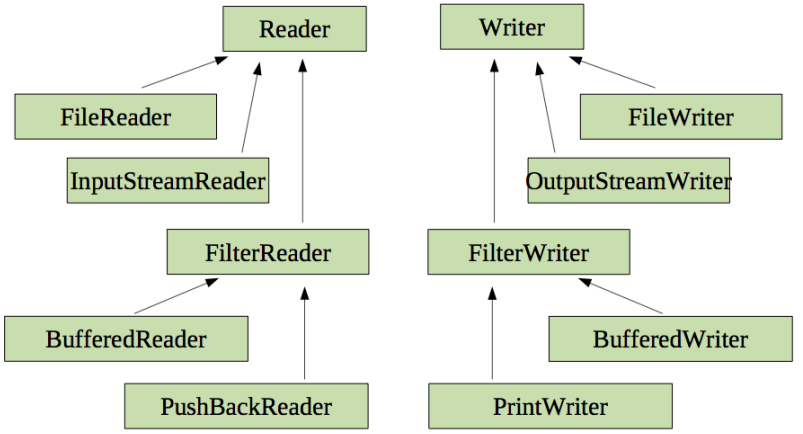
source: cours Remi Forax

E/S par Octet (byte - 8bit)



source: cours Remi Forax

E/S par Char (16 bit)



source: cours Remi Forax

Schéma de programmation

- Choisir son gestionnaire de flux :
  - Basé sur des caractères : *XxxReader/Writer*
  - Basé sur des octets : *XxxInputStream/OutputStream*
    - » Ex : lire un fichier
      - c'est du texte : FileReader
      - c'est des octets : FileInputStream

- **Note:** *System.out* et *System.err* <= PrintStream
  System.in <= InputStream

Schéma de programmation (2)

- Choisir son gestionnaire de flux : caractères ou octets

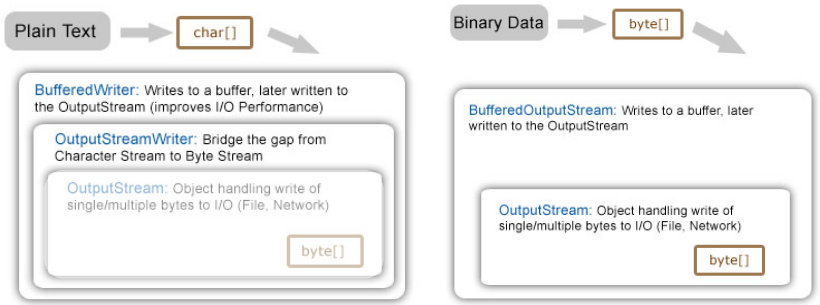
1. **Ouvrir** un flux F (connexion)
   Passe par la construction d'un ou plusieurs objets, afin de s'adapter aux besoins. (via 2 classes abstraites):

2. **Lire** ou **écrire** sur F
   appel de *read()* ou *write()*

3. **Fermer** F ⇐ A ne pas oublier.
   close ()

I/O byte vs Char

- Si on traite des characteurs, on utilise Reader/Writer, sinon InputStream/OutputStream
- À la base, le flux est en byte (mais Reader/Writer le cache)



Types de Flux

Les types de *Stream*

Un objet dérivant d'*InputStream (OutputStream)*
peut lire (écrire) à partir de différentes sources :

- un tableau d'octet, accès bufferisé à la mémoire : *ByteArrayInputStream*
- un objet *string* : *StringBufferInputStream*
- un fichier : *FileInputStream*
- un pipe (tube) : *PipedInputStream* (multithreading)
- un ou plusieur flux : *SequenceInputStream* (voir la javadoc)

Ces classes se généralisent pour l'écriture et pour les *Reader/Writer*.

Ex : copier un fichier textuel

```
import java.io.*;

public class CopyCharacters {

    public static void main(String[] args) throws IOException {
       FileReader input  = null; // flux d'entrée en char
       FileWriter output = null; // flux de sortie en char

       input  = new FileReader("original.txt"); // ouverture
       output = new FileWriter("copie.txt");    // ouverture

       int c;
       while ((c = input.read()) != -1){     //lit caractère par
                                             //caractère jusqu'à la fin
                output.write(c);             // c en int
        }

        if (input != null) {
            input.close();    // ferme le flux d'entrée
        }
        if (output != null) {
            output.close();   // ferme le flux de sortie
        }
    }
}
```

Entrée
Sortie

Ex : copier un fichier textuel

```
import java.io.*;

public class CopyCharacters {

    public static void main(String[] args) throws IOException {
       FileReader input  = null; // flux d'entrée en char
       FileWriter output = null; // flux de sortie en char

       input  = new FileReader("original.txt"); // ouverture
       output = new FileWriter("copie.txt");    // ouverture

       int c;
       while ((c = input.read()) != -1){     //lit caractère par
                                             //caractère jusqu'à la fin
                output.write(c);             // c en int
        }

        if (input != null) {
            input.close();    // ferme le flux d'entrée
        }
```

Entrée
Sortie

à vous : Copier un fichier texte vers deux fichiers (a) et (b).
(a) => texte en minuscules ; (b) => texte en majuscule

```
AIDE: char Character.toLowerCase(char c); char Character.toUpperCase(char c);
```

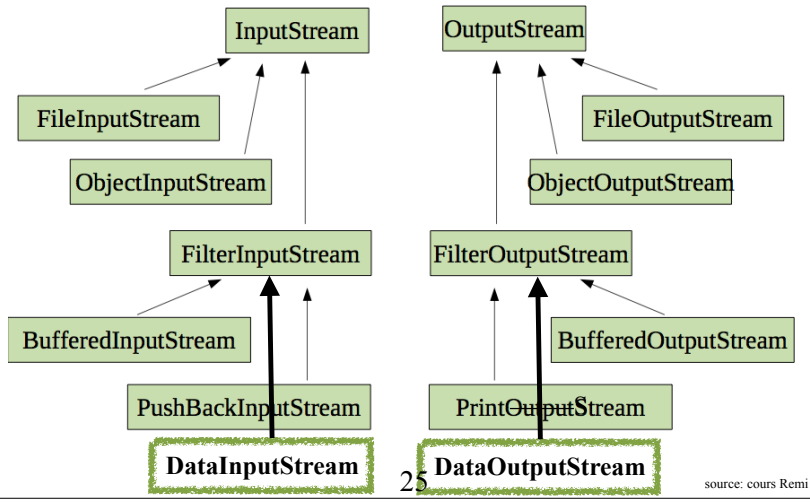
I/O de types primitifs

- Pour faire mieux que des caractères ou des octet, utilisons : *DataInputStream* et *DataOutputStream*
- Ce sont des adaptateurs, ils se branchent sur les *InputStream* et *OutputStream* (**que** des classes Stream)
- Regardons le constructeur pour l'un des deux :

```
DataInputStream(InputStream in) : Creates a
   DataInputStream that uses the specified
         underlying InputStream.
```

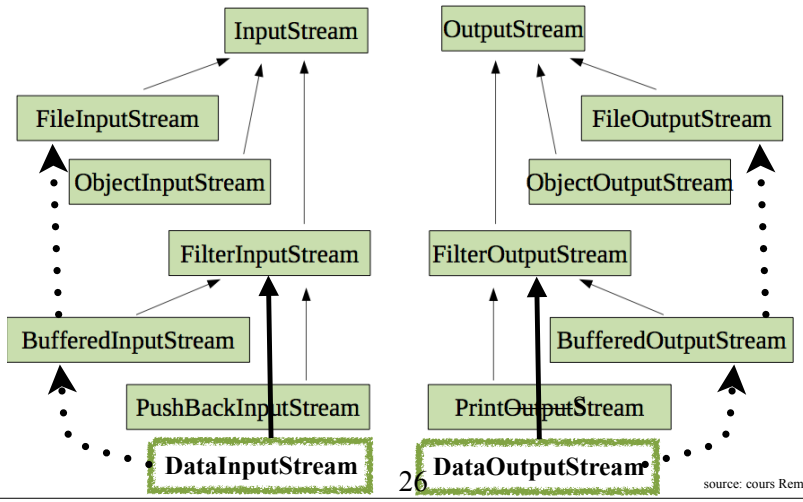
- La liste des méthodes : *readBoolean, readFloat, readDouble, ...*
- ensi que la *available()* pour voir les octets restantes
- Le symétrique existe aussi pour l'écriture

E/S par Octet (byte)



source: cours Remi Forax

E/S par Octet (byte)



source: cours Remi Forax

Exemple de flux de données

```
Ex écrire :
int[] entiers = { 12, 8, 13, 29, 50 };
DataOutputStream out = new DataOutputStream
            (new BufferedOutputStream(new FileOutputStream("nomFichier")));
for (int i = 0; i < entiers.length; i++) {
   out.writeInt(entiers [i]);
}

Ex lire :
DataInputStream in = new DataInputStream(new
               BufferedInputStream(new FileInputStream("nomFichier")));
int somme = 0;
try {
        while (true) {
            somme += in.readInt();
        }
    } catch (EOFException e) {
 }
```


Exemple de flux de données

```
Ex écrire :
int[] entiers = { 12, 8, 13, 29, 50 };
DataOutputStream out = new DataOutputStream
            (new BufferedOutputStream(new FileOutputStream("nomFichier")));
for (int i = 0; i < entiers.length; i++) {
   out.writeInt(entiers [i]);
}

Ex lire :
DataInputStream in = new DataInputStream(new
               BufferedInputStream(new FileInputStream("nomFichier")));
int somme = 0;
try {
        while (true) {
            somme += in.readInt();
        }
    } catch (EOFException e) {
 }
```

ALTERNATIVE:

```
while (in.available() > 0)
```

Flux *Buffered*

- L'usage de flux *Byte* ou *Character* peuvent être coûteux
  - ex : accès au disque dur, connexion réseau, ...
- Les flux bufferisés lisent les données par blocs
  - buffer = tampon
  - une quantité de données est chargée en une fois dans une zone de mémoire plus rapide d'acceder
- Les buffers se branchent comme des convertisseurs sur les flux standards :

```
inputStream = new BufferedReader(
      new FileReader("original.txt"));
outputStream = new BufferedWriter(
      new FileWriter("copie.txt"));
```

```
Flux Character
Flux bufferisés
```


Entrées/Sorties et Exceptions

- Les objets et méthodes d'entrée/sortie produisent de nombreuses exceptions, les plus fréquentes :
  - fichier non trouvé (*FileNotFoundException*)
  - fin de fichier (*EOFException*)
  - accès non-autorisé (*SecurityException*)
- Les Exceptions peuvent survenir à la construction d'un flux, s'il y a un problème de connexion.
- Conseil : Pour une application stable, ces *Exceptions* sont à traiter avec un *try/catch.*

Classe File

La classe File

La classe **java.io.File** rend plus facile l'écriture de programmes qui manipule des fichiers, indépendamment de la plateforme d'exécution.

Un meilleur nom serait "FileName" ou "FilePath", car elle représente soit le *nom* d'un fichier particulier ou les *noms* d'un ensemble de fichiers dans un dossier.

Les fonctionnalités sont nombreuses :

- getAbsolutePath()
- exists()
- isDirectory()
- listFiles()
- list (FilenameFilter ff)
- … Voir la javadoc pour plus de détail.

Exemple de File

```
import java.util.regex.*;
import java.io.*;
import java.util.*;

public class DirList {
  public static void main(String[] args) {
    File path = new File(".");
    String[] list;
    if(args.length == 0)
      list = path.list();
    else
      list = path.list(new DirFilter(args[0]));

    Arrays.sort(list, String. CASE_INSENSITIVE_ORDER);
    for(String dirItem : list)
      System.out.println(dirItem);
  }
}
```

```
class DirFilter implements FilenameFilter{
  private Pattern pattern;
  public DirFilter(String regex) {
    pattern = Pattern.compile(regex); }
  public boolean accept(File dir, String name){
    return pattern.matcher(name).matches(); }
}
```


Sérialisation

Utilisation

Sérialisation: Lire/Écrire plus que les types primitifs (c.a.d. Objets).

- Fichier
  - Sauvegarde (reprise sur panne, persistance)
  - Fichier propriétaire, binaire
- Réseau
  - Partage d'instances entre deux clients
  - Création d'instances sur serveur et distribution
    - sérialisation applicable à l'ensemble des systèmes d'exploitation
- Base de données
  - Stockage d'instances dans une BD
  - Recherche

Les flux d'objets

Lire/Écrire plus que les types primitifs (c.à.d. Objets).

Ex écrire un objet date :

```
FileOutputStream ostream = new FileOutputStream("t.tmp");
ObjectOutputStream p     = new ObjectOutputStream (ostream);
p.writeObject(new Date());
ostream.close();
```

Ex lire un objet date :

```
FileInputStream istream = new FileInputStream("t.tmp");
ObjectInputStream p     = new ObjectInputStream(istream);
Date date = (Date)p.readObject(); //changer type
istream.close();
```

Lire et Écrire mes objets

C'est la sérialisation.

• Pour pouvoir lire ou écrire un objet, la classe de cet objet doit implémenter l'interface *Serializable* ou *Externalizable*.

• L'interface *Serializable* sert à marquer le type de l'objet comme candidat à la serialisation. Elle ne contient rien !

• L'interface *Externalizable* hérite de *Serializable* et permet d'expliciter la manière de lire et d'écrire les objets
(ex compression, meta-données).

```
public void writeExternal(ObjectOutput out) throws
IOException
public void readExternal(ObjectInput in) throws
IOException, ClassNotFoundException
```


Lire et Écrire mes objets

C'est la sérialisation.

• Pour pouvoir lire ou écrire un objet, la classe de cet objet doit implémenter l'interface *Serializable* ou *Externalizable*.

• L'interface *Serializable* sert à marquer le type de l'objet comme candidat à la serialisation. Elle ne contient rien !

• L'interface *Externalizable* hérite de *Serializable* et permet d'expliciter la manière de lire et d'écrire les objets
(ex compression, meta-données).

• Avec *Externalizable* plus de contrôle sur la sérialization.

• Si un attribut est précédé du mot-clé **transient**, il n'est pas inclus dans le processus de sérialisation (à re-visiter).

La base de la sérialisation

1. Pour sérialiser un objet, instancier un type de *OutputStream*,
2. l'associer à un objet de type *ObjectOutputStream*.
3. Il ne reste plus qu'à appeller la méthode **writeObject( )**, et l'objet est sérialisé puis envoyé à l'*OutputStream*.

Ex écrire un objet date :

```
FileOutputStream ostream = new FileOutputStream("t.tmp");
ObjectOutputStream p    = new ObjectOutputStream (ostream);
p.writeObject(new Date());
ostream.close();
```


La base de la sérialisation (2)

L'opération est bien sûr reversible :

1. Instancier un *InputStream*
2. L'associer à un *ObjectInputStream*
3. appeller la méthode **readObject( )**.

4. Vous récupérer un *Object, il faut alors le sous-classer.*

Ex lire un objet date :

```
FileInputStream istream = new FileInputStream("t.tmp");
ObjectInputStream p     = new ObjectInputStream(istream);
Date date = (Date)p.readObject();
istream.close();
```

Créer une classe sérialisable :
Il suffit simplement que la classe implémente l'interface java.io.Serializable".

```
public class Writeable implements java.io.Serializable
```

Note : toutes les variables de la classe doivent également être des objets sérialisables (ou des types primitifs).

Sérialisation d'un object.

```
try {
        ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("monFichier.sav"));
        out.writeObject(monObject1);
        out.writeObject(monObject2);
        out.close();
} catch( IOException e ) {

}
```

Désérialisation d'un object.

```
try {
        ObjectInputStream in = new ObjectInputStream(new FileInputStream("monFichier.sav"));
        MonObject1 monObject1 = (MonObject1)in.readObject();
        MonObject2 monObject2 = (MonObject2)in.readObject();
        in.close();
} catch( ClassNotFoundException e1 ) {

} catch( IOException e2 ) {

}
```


source: http://java.developpez.com/faq/java

Créer une classe sérialisable :
Il suffit simplement que la classe implémente l'interface java.io.Serializable".

```
public class Writeable implements java.io.Serializable
```

Note : toutes les variables de la classe doivent également être des objets sérialisables (ou des types primitifs).

Sérialisation d'un object.

```
try {
        ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("monFichier.sav"));
        out.writeObject(monObject1);
        out.writeObject(monObject2);
        out.close();
} catch( IOException e ) {

}
```

Désérialisation d'un object.

```
try {
        ObjectInputStream in = new ObjectInputStream(new FileInputStream("monFichier.sav"));
        MonObject1 monObject1 = (MonObject1)in.readObject();
        MonObject2 monObject2 = (MonObject2)in.readObject();
        in.close();
} catch( ClassNotFoundException e1 ) {

} catch( IOException e2 ) {

}
```


source: http://java.developpez.com/faq/java

La sérialisation est récursive

- Lors de la serialisation d'un objet *Serializable*, tous les objets qu'il contient sont serialisés et ainsi de suite récursivement.
  - Aussi avec les tableaux d'objets (Array, ArrayList, etc).
- Il y a la construction d'un graphe d'objet à serialiser ("web of objects" ou "object graph") qui est parcouru lors de l'étape de sérialisation.
- La construction et le parcours de ce graphe peut être coûteux.
  - Implémenter alors l'interface Externalizable si tout le contenu d'un objet n'est pas à serialiser par exemple.

Example de Externalizable

(repris du livre de Bruce Eckel)

```
import java.io.*;

public class Blip3 implements Externalizable {
  private int i; private String s;

  public Blip3() {
    print("Blip3 Constructor"); // i,s not initialized
  }
  public Blip3(String x, int a) {
    print("Blip3(String x, int a)");
    i = a; s = x;  // i,s initialized only in non-default constructor.
  }
  public void writeExternal(ObjectOutput out) throws IOException {
    print("Blip3.writeExternal");
    out.writeObject(s); // You must do this
    out.writeInt(i);    // You must do this
  }
  public void readExternal(ObjectInput in) throws IOException,
                                                  ClassNotFoundException {
    print("Blip3.readExternal");
    s = (String)in.readObject(); // You must do this
    i = in.readInt();            // You must do this
  }
}
```

L'alternative à Externalizable

- Implementer l'interface *Serializable* et **ajouter** les méthodes writeObject( ) and readObject( ).
- Attention, il s'agit bien d'ajouter, pas de redéfinir, ni d'implémenter.
- Si ces 2 méthodes sont ajoutées, elles seront appellées lors de la sérialisation.
- Les signatures exactes de ces méthodes :

```
private void writeObject(ObjectOutputStream stream)
   throws IOException;

private void readObject(ObjectInputStream stream)
   throws IOException, ClassNotFoundException
```

- Comment ça marche ?.... C'est la magie de la sérialisation

Exemple de serialisation

On ne souhaite serialiser cet objet :

```
public class Matrix {
   int[][] valeurs;
   String info;
}
```

EXERCICE:
1. On souhaite pouvoir sérialiser les objets de cette classe
2. On souhaite limiter la sérialisation uniquement au champs info et aux valeurs dans la diagonale de la matrice

Exemple de serialisation

On ne souhaite serialiser que les valeurs de la diagonale :

```
public class Matrix implements Serializable {
   transient int[][] valeurs;
   String info;
   private void writeObject(ObjectOutputStream s) throws IOException {
        s.defaultWriteObject();// serialise uniquement info
        // Sauvegarde explicite des valeurs de la diagonale
        for (int i = 0; i < nbValeurs; i++)
               s.writeInt(valeurs[i][i]);
    }
   private void readObject(ObjectInputStream s)
                     throws IOException, ClassNotFoundException  {
        s.defaultReadObject(); // récupération de info
        // Récupération des valeurs de la diagonale
        for (int i = 0; i < nbValeurs; i++)
               valeurs[i][i] = s.readInt();
   }
}
```


Précaution sur la persistance

- La sérialisation est une solution pour effectuer une sauvegarde de l'état d'une application. Cet état peut alors être restaurer ultérieurement.
- Mais attention !
- Que se passe-t-il si vous sérialiser 2 objets contenant chacun une référence sur un même troisième ?
  - Les deux objets sérialisés donnent-ils lieux à deux fichiers avec 2 occurances de la troisième ?
  - Lors de la lecture de ces 2 objets, combien récupère-t-on d'occurrences du troisième ?

Exemple de précaution

Prenons un exemple simple (repris du livre de Bruce Eckel)

Voici deux classes utilisées par notre application :

```
class House implements Serializable {}

class Animal implements Serializable {
  private String name;
  private House preferredHouse;
  Animal(String nm, House h) {
    name = nm;
    preferredHouse = h;
  }
}
```


Exemple de précaution - suite

```
public class MyWorld {
 public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    House house = new House();
    Vector<Animal> animals = new Vector<Animal>();
    animals.add(new Animal("Bosco the dog", house));
    animals.add(new Animal("Ralph the hamster", house));
    animals.add(new Animal("Fronk the cat", house));

    ByteArrayOutputStream buf1 = new ByteArrayOutputStream();
    ObjectOutputStream o1 = new ObjectOutputStream(buf1);
    o1.writeObject(animals);
    o1.writeObject(animals); // on écrit une 2ème fois

    ByteArrayOutputStream buf2 = new ByteArrayOutputStream();
    ObjectOutputStream o2 = new ObjectOutputStream(buf2);
    o2.writeObject(animals); // dans un autre flux
```


Exemple de précaution - fin

```
ObjectInputStream in1 = new ObjectInputStream(
                    new ByteArrayInputStream(buf1.toByteArray()));
ObjectInputStream in2 = new ObjectInputStream(
                    new ByteArrayInputStream(buf2.toByteArray()));
Vector<Animal>
   animals1 = (Vector<Animal>)in1.readObject(),
   animals2 = (Vector<Animal>)in1.readObject(),
   animals3 = (Vector<Animal>)in2.readObject();
```

animals: [Bosco the dog[Animal@1cde100], House@16f0472
, Ralph the hamster[Animal@18d107f], House@16f0472
, Fronk the cat[Animal@360be0], House@16f0472]

"web of objects"
flux de sérialisation

animals1: [Bosco the dog[Animal@e86da0], House@1754ad2
, Ralph the hamster[Animal@1833955], House@1754ad2
, Fronk the cat[Animal@291aff], House@1754ad2]

animals2: [Bosco the dog[Animal@e86da0], House@1754ad2
, Ralph the hamster[Animal@1833955], House@1754ad2
, Fronk the cat[Animal@291aff], House@1754ad2]

"web of objects" preservé
dans un flux de
désérialisation

animals3: [Bosco the dog[Animal@ab95e6], House@fe64b9
, Ralph the hamster[Animal@186db54], House@fe64b9
, Fronk the cat[Animal@a97b0b], House@fe64b9]

"web of objects" non-preservé
entre deux flux de
désérialisation

sérialisation dehors java (XML)

- Une limitation importante de la sérialisation d'objets :
  - c'est une solution uniquement pour Java
  - que les programmes Java peuvent désérialiser des objets
- Une solution plus globale est de convertir les données au format XML, pour être accessible à d'autres plateformes et langages.

sérialisation dehors java (XML)

- Extensible Markup Language (XML) est un langage de balisage (markup language) qui définit un ensemble de règles pour les documents d'encodage dans un format qui est à la fois lisible par nous et par l'ordinateur.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE recipe PUBLIC "-//Happy-Monkey//DTD RecipeBook//EN"
"http://www.happy-monkey.net/recipebook/recipebook.dtd">

<recipe>

    <title>Peanut-butter On A Spoon</title>

    <ingredientlist>
        <ingredient>Peanut-butter</ingredient>
    </ingredientlist>

    <preparation>
        Stick a spoon in a jar of peanut-butter,
        scoop and pull out a big glob of peanut-butter.
    </preparation>

</recipe>
```


sérialisation dehors java (XML)

- Vous même
  - créez des classes et fonctions pour lire et écrire des fichiers XML liés à vos classes et objets
  - utilisez java.xml.stream.**XMLStreamReader** / **XMLStreamWriter**
- Sérialiser les objets (entiers) Java en XML
  - Plusieurs bibliothèques (ex XStream, JAXB)
  - Utiliser XMLDecoder et XMLEncoder de java.beans.

un objet en XML

```
import java.xml.stream.*;

class Book {
    private String mytitle;
    private String myauthor;
    ...
    public void writeBook(XmlStreamWriter w) {
          w.writeStartElement(null, 'book', null);
          w.writeAttribute(null, null, 'author', myauthor);
          w.writeAttribute(null, null, 'title', mytitle);
          w.writeEndElement(); //end book
     }

    public Book readBook(XmlStreamReader r) {...}
  }
<book>
    <author> Mike </author>
    <title> Mike Jazz </title>
</book>
```


fin du cours sur les E/S