Chapitre 4

Les entrées sorties

Lecture du clavier et écriture à l'écran.
Gestion des fichiers.
Lecture et écriture des fichiers binaires séquentiels
Lecture et écriture des fichiers textes
Lecture et écriture des fichiers en accès direct
Lecture et écriture des objets : principe de sérialisation

1. Le package java.io

Le package java.io (io = Input Output) contient toutes les classes de gestion des entrées sorties en java.

Les entrées et sorties du langage Java sont gérées par les classes du package *java.io*.

Ils existent plus de 20 classes de gestion des entrées/sorties!! Nous ne verrons que celles qui sont essentielles à la réalisation de nos programmes.

2. Les instructions standards d'entrée / sortie

Pour pouvoir faire nos premiers programmes Java, il est indispensable de connaître comment on écrit à l'écran et comment il est possible de saisir des valeurs au clavier.

Nous verrons plus tard comment il est possible de gérer les fichiers.

Comme cela est le cas dans la plupart des langages, Java définit les deux flots d'entrée et de sortie standards (l'équivalent en C de *stdin* et *stdout*).

La classe java.lang.**System** contient deux variables de classe:

public static InputStream in

public static PrintStream out

2.1. Gestion du flot de sortie standard : PrintStream

La variable de classe **System.out** est une instance de la classe **PrintStream**.

La classe ***PrintStream*** définit des méthodes permettant d'écrire tous les types Java:

```
System.out.println("Hello world!");
System.out.print("x = ");
System.out.println(x);
System.out.println("y = " + y);
System.out.flush();
```

2.2. Gestion du flot d'entrée standard : BufferedReader

La variable de classe **System.in** est une instance de la classe **InputStream**.

On utilise les méthodes de la classe **BufferedReader** qui sont moins élémentaires et nous permet de lire une chaîne de caractère au clavier.

Pour cela, un constructeur de la classe InputStreamReader, accepte en paramètre un objet de la classe *InputStream.*

Puis on utilise la classe BufferedReader.

```
BufferedReader in2 = new BufferedReader(new InputStreamReader(System.in))
```

On peut ensuite lire une chaine:

```
String s = in2.readLine();
```

2.3. La classe Terminal

```
import java.io.*;
```

```
public class Terminal{
    static BufferedReader in =
        new BufferedReader(new InputStreamReader(System.in));

    public static String lireString() // Lire un String
    {
        String tmp="";
        try {
            tmp = in.readLine();
        }
        catch (IOException e)
            {
                exceptionHandler(e);
            }
        return tmp;
    }

    public static int lireInt()  // Lire un entier
    {
        int x=0;
        try {
            x=Integer.parseInt(lireString());
        }
        catch (NumberFormatException e) {
            exceptionHandler(e);
        }
        return x ;
    }

    protected static void exceptionHandler(Exception ex){
        TerminalException err = new TerminalException(ex);
        throw err;
    }
}

class TerminalException extends RuntimeException{
    Exception ex;
    TerminalException(Exception e){
        ex = e;
    }
}
```

3. Le path (ou File)

La classe **File** permet de gérer les **répertoires** et les **fichiers** du système d'exploitation sur lequel la JVM est exécutée.

La classe File est la classe de représentation d'un répertoire ou d'un nom de fichier dépendant du système d'exploitation. Elle permet de créer le nom du fichier sur lequel on réalisera les opérations d'entrées/sorties. De nombreuses méthodes permettent de : créer le nom de fichier, de tester l'existence du fichier, de connaître le répertoire parent, de lister un répertoire avec un filtre si besoins, …

Exemples:

```
File f = new File("/etc/passwd");
System.out.println(f.exists());     // --> true
```

```
System.out.println(f.canRead());    // --> true
System.out.println(f.canWrite());   // --> false


System.out.println(f.length());   // --> 11345

File d = new File("/etc");
System.out.println(d.isDirectory());  // --> true

String[] files = d.list();
for(int i=0; i < files.length; i++)
    System.out.println(files[i]);
```

Le caractère séparateur de nom de fichier et répertoire : '/ 'ou ' \'

```
String File.separator
char File.separatorChar
```

A ne pas confondre avec le caractère séparateur de path ; ' ; ' ou ' : '

```
String File.pathSeparator
char File.pathSeparatorChar
```

Depuis la version 5, java accepte l'usage du '/' dans l'écriture d'un path d'accès à un fichier sur le système Windows

Exemples : Exemple20_EntreesSorties/ExempleFile.java

```
//
// Importation du package d'entrée sorties (Input/Output)
//  qui contient toutes les classes de gestion des fichiers
//
import java.io.*;

public class ExempleFile
{
    public static void main(String[] args) throws IOException
    {
        // Ne pas confondre le caractère séparateur de répertoire
        //   et de fichier dans une chaîne qui décrit le chemin d'accès
        //   à un fichier ou à un répertoire : / ou \
        // Avec  le caractère séparateur des chemin d'accès se trouvant
        //   par exemple dans les variables d'environnement
        //   PATH ou CLASSPATH.
        //
        System.out.println("----------------------");
        System.out.println("separator : "+File.separator);
        System.out.println("pathSeparator : "+File.pathSeparator);

        // On se propose de créer le répertoire "exemple"
        //  qui contient deux sous-répertoires "rep1" et "rep2.
        // Chacun de ces répertoires contiennent les fichiers:
        //    f1.txt et f2.txt
        //  et rep2 contient le fichier toto.class
        //
        System.out.println("----------------------");
        File rep1 = new File("exemple"+File.separator +"rep1");
        rep1.mkdirs();
        File rep2 = new File("exemple"+File.separator +"rep2");
```

```
        rep2.mkdirs();

        File f;

        f = new File("exemple"+File.separator
                     +"rep1"+File.separator+
                     "f1.txt");
        f.createNewFile();

        f = new File("exemple"+File.separator
                     +"rep1"+File.separator+
                     "f2.txt");
        f.createNewFile();

        f = new File("exemple"+File.separator
                     +"rep2"+File.separator+
                     "f1.txt");
        f.createNewFile();

        f = new File("exemple"+File.separator
                     +"rep2"+File.separator+
                     "f2.txt");
       f.createNewFile();

        f = new File("exemple"+File.separator
                     +"rep2"+File.separator+
                     "toto.class");
        f.createNewFile();

        // Affichage de l'arborescence créé en utilisant
        //  une méthode récursive
        //
        // On obtient :
        //    exemple
        //      rep1
        //        f1.txt
        //        f2.txt
        //      rep2
        //        f1.txt
        //        f2.txt
        //        toto.class
        //
        afficherRep(new File("exemple"),"");

        // On affiche encore toute l'arborescence mais on filtre
        //  certains fichiers (ici les .class)
        //
        System.out.println("----------------------");
        afficherRepFiltre(new File("exemple"),"");

        // Test de la méthode qui crée un fichier temporaire.
        // Il est créé dans un répertoire temporaire dépendant
        //  de l'OS
        //
        System.out.println("----------------------");
        f = File.createTempFile("bidon",".txt");
        System.out.println(f.getAbsolutePath());
        // C:\Temp\bidon1971705398436420305.txt

        // Un File en utilisant les "/" en dur
        // Attention à votre versiond de Java (>=1.5)
```

```
        //
        System.out.println("----------------------");
        f = new File("exemple/rep1/f1.txt");
        System.out.println(f.getAbsolutePath());
        if (f.exists()) System.out.println("f existe");

        // Une autre façon de faire
        //
        System.out.println("----------------------");
        f = new File(new File("exemple","rep1"),"f1.txt");
        System.out.println(f.getAbsolutePath());
        if (f.exists()) System.out.println("f existe");


        // Affichage de la variable d'environnement PATH
        //
        System.out.println("----------------------");
        String path = System.getenv("PATH");

        for(String s:System.getenv("PATH").split(File.pathSeparator))
            System.out.println(s);
    }

    // Méthode récursive d'affichage d'un répertoire
    //
    static void afficherRep(File file,String marge)
    {
        System.out.println(marge+file.getName());
        if (file.isDirectory())
            {
                for(File f:file.listFiles())
                    afficherRep(f,marge+"  ");
            }
    }

    // Méthode récursive d'affichage d'un répertoire
    //   et qui filtre les fichier .class (par exemple)
    //
    static void afficherRepFiltre(File file,String marge)
    {
        System.out.println(marge+file.getName());
        if (file.isDirectory())
            {
                for(File f:file.listFiles(new Filtre()))
                    afficherRepFiltre(f,marge+"  ");
            }
    }
}

// La classe qui implémente l'interface FileFilter
// et qui est utilisée à l'appel de la méthode list
//
class Filtre implements FileFilter
{
    public boolean accept(File f)
    {
        String nom = f.getName();
        if (nom.endsWith(".class")) return false;
        return true;
    }
}
```

Exécution :

```
  ----------------------
  separator : \
  pathSeparator : ;
  ----------------------
  exemple
    rep1
      f1.txt
      f2.txt
    rep2
      f1.txt
      f2.txt
      toto.class
  ----------------------
  exemple
    rep1
      f1.txt
      f2.txt
    rep2
      f1.txt
      f2.txt
  ----------------------
  C:\DOCUME~1\jlaforgu\LOCALS~1\Temp\bidon5609068846625110065.txt
  ----------------------
  F:\Jacques\CNAM\SITE\SITE_NFA001-
002\Exemples\bin\Exemple20\exemple\rep1\f1.txt
  f existe
  ----------------------
  F:\Jacques\CNAM\SITE\SITE_NFA001-
002\Exemples\bin\Exemple20\exemple\rep1\f1.txt
  f existe
  ----------------------
  C:\PROGRA~1\XEmacs\XEmacs-21.1.9\i386-pc-win32
  C:\Program Files\Windows Resource Kits\Tools\
  C:\Program Files\PHP\
  C:\WINDOWS\system32
  C:\WINDOWS
  C:\WINDOWS\System32\Wbem
  C:\Program Files\Hummingbird\Connectivity\11.00\Accessories\

  C:\Program Files\Fichiers communs\Lenovo
  C:\Program Files\Windows Imaging\
  D:\ProgramFiles\go\bin
  C:\Program Files\Java\jdk1.6.0_18\bin
  D:\apache-maven-2.2.1\bin
  C:\Program Files\Bouml
```

4. Les fichiers de données

4.1. Présentation

FileInputStream est la classe permettant de créer un flot de lecture. Attention, cette classe ne gère que les informations du niveau binaire. Pour lire des informations de type int, double, String il est nécessaire d'utiliser les classes **Data**.

FileOutputStream est la classe permettant de créer un flot d'écriture.

DataInputStream est la classe permettant de lire un flot de lecture de données élémentaires. Les méthodes de lecture sont par exemple *readInt, readDouble, readBoolean, readUTF (pour lire des chaînes de caractères), …*

DataOutputStream est la classe permettant d'écrire un flot d'écriture.

4.2. Exemple

Pour écrire des données binaires et typées d'un tableau de double :

```
  // Création du nom du fichier
  //     (à ne pas confondre avec le fichier lui-même)
  File fichier = new File(repertoire,nom);

  // Création du flot d'écriture
  FileOutputStream fos = new FileOutputStream(fichier);
  DataOutputStream dos = new DataOutputStream(fos);
  // Le fichier sera écrasé s'il existe déjà

  // Ecriture des données (un tableau de double)
  dos.writeUTF("Exemple d'écriture du tableau");
  dos.writeInt(tab.length);
  for(int i=0;i< tab.length;i++) dos.writeDouble( tab[i]);

  // Fermeture du fichier
  dos.close();
```

Pour lire les mêmes données qui ont été écrites :

```
  // Création du nom du fichier (à ne pas confondre avec le fichier lui-
même)
  File fichier = new File(repertoire,nom);

  // Création du flot de lecture
  FileInputStream fis = new FileInputStream(fichier);
  DataInputStream dis = new DataInputStream(fis);

  // Lecture des données
  String chaine = dis.readUTF();
  int n = dis.readInt();
  double[] tab = new double[n];
  for(int i=0;i< n;i++) tab[i] = dis.readDouble();

  // Fermeture du fichier
  dis.close();
```

Il faut lire les données comme on les a écrites.

Pour pouvoir respecter facilement ce principe, la meilleure conception est de créer des méthodes d'écriture et de lecture associées à la classe dont on veut écrire les objets.

4.3. Principes de conception

Avec ces primitives, il est donc possible d'écrire et lire toutes les données classiques d'un programme informatique.

Les principes de la programmation classique sont donc applicables et remplissent tous les besoins.

Java étant un langage objet, il est conseillé d'écrire des méthodes d'écriture et de lecture des attributs de l'objet. Par exemple, on peut créer les méthodes *read* et *write* associées à l'objet.

Exemple :

```
public class Livre
{
 ……
 ……

public void write(DataOutputStream dos) throws IOException
  {
    dos.writeUTF(titre);
    dos.writeInt(auteurs.length);
    int i;
    for(i=0;i<auteurs.length;i++)
      {
         dos.writeUTF(auteurs[i]);
      }
    dos.writeInt(genre);
  }

  public void read(DataInputStream dis) throws IOException
  {
    titre=dis.readUTF();
    int n = dis.readInt();
    int i;
    for (i=0;i<n;i++) addAuteur(dis.readUTF());
    genre=dis.readInt();
  }
}
```

A l'appel:

```
  Cet exemple est un extrait de la classe Biblio de gestion d'une
bibliothèque dans laquelle on définit les méthodes des chargement et
sauvegarde de la bibliothèque.

  public void charger() throws IOException
```

```
  {
    File fic = new File(path,nom);
    FileInputStream fout = new FileInputStream(fic);
    DataInputStream fich = new DataInputStream(fout);

    nom = fich.readUTF();
    int n = fich.readInt();
    int i;
    for(i=0;i<n;i++)
      {
          Livre l = new Livre();
          l.read(fich);
          addLivre(l);
      }

    fich.close();
  }

public void sauver() throws IOException
  {
    File fic = new File(path,nom);
    FileOutputStream fin = new FileOutputStream(fic);
    DataOutputStream fich = new DataOutputStream(fin);

    fich.writeUTF(nom);
    int i;
    fich.writeInt(livres.size());
    for(i=0;i<livres.size();i++)
      {
           Livre l=(Livre)livres.elementAt(i);
           l.write(fich);
      }

    fich.close();
  }
}
```

5. La sérialisation

5.1. Présentation

On peut vouloir écrire et lire dans un fichier toutes les informations d'un objet.

En Java, un objet est composé d'attributs de types primitifs mais également de types classes qui sont autant de références (pointeurs). Un objet Java est donc une "arborescence" d'objets. Pour pouvoir écrire toute l'arborescence d'un objet Java puis le lire, il faut que le langage soit capable d'écrire tous les objets puis de les reconstituer lors de l'opération de lecture.

On appelle cette étape d'écriture la "**sérialisation**" d'un objet.

La propriété importante est que tous les attributs doivent être sérialisables et ceci dans toute l'arborescence de l'objet.

Les classes prédéfinies *String, Vector* , *tableau…* sont par défaut sérialisables.

Une classe non prédéfinie est sérialisable si elle implémente l'interface *Serializable*.

Il est inutile d'implémenter les méthodes de cette interface. Par défaut, la sérialisation sera appliquée à tous les attributs de l'objet.

5.2. Exemple

```
public class Livre implements Serializable
{
 …….<inchangée>
}
A l'appel :

  public void charger() throws IOException, ClassNotFoundException
  {
    File fic = new File(path,nom);
    FileInputStream fout = new FileInputStream(fic);
    ObjectInputStream fich = new ObjectInputStream(fout);

    nom = (String)fich.readObject();
    livres = (Vector)fich.readObject();

    fich.close();
  }

  public void sauver() throws IOException
  {
    File fic = new File(path,nom);
    FileOutputStream fin = new FileOutputStream(fic);
    ObjectOutputStream fich = new ObjectOutputStream(fin);

    fich.writeObject((Object)nom);
    fich.writeObject((Object)livres);

    fich.close();
  }
```

ATTENTION : La sérialisation a un prix: la compatibilité binaire des informations écrites qui dépendent des versions JAVA et surtout de la stabilité des classes dont les objets sont écrits sur ce principe.

Il faut donc utiliser ce principe pour écrire et lire les données dans un même cycle d'exécution du programme : écriture/lecture dans un socket, dans des fichiers temporaires ou partagés, … etc…
Ne pas utiliser pour faire de la sauvegarde.

6. Gestion des fichiers textes

6.1. Présentation

Les classes précédentes gèrent les informations au niveau "binaire". C'est un moyen de stockage efficace mais ont les désavantages suivants :

- le niveau **Data** n'est pas lisible dans les fichiers
- le niveau **Object** est dépendant de la définition des classes.

On peut donc vouloir gérer plutôt des fichiers **textes**. Pour cela, les classes suivantes sont utilisées:

- **PrintStream** pour écrire des lignes dans un fichier texte
- **BufferedReader** pour lire les lignes d'un fichier texte.

6.2. Exemple

Pour écrire les lignes d'un fichier texte :

```
File file = new File(rep+File.separator+nom);
FileOutputStream fich = new FileOutputStream(file.getAbsolutePath());
PrintStream flot = new PrintStream(fich);
…….
boucle sur l'instruction:  flot.println(infos);
où infos est une String.
```

Pour lire des lignes dans un fichier texte :

```
File file = new File(rep+File.separator+nom);
FileInputStream fich = FileInputStream(file.getAbsolutePath());
BufferedReader flot = new BufferedReader(new InputStreamReader(fich));

 String ligne=flot.readline();
 while(ligne != null)
 {
   traitement de la chaine lue
   ligne=flot.readline();
 }
```

Au lieu de lire ou écrire ligne à ligne, on peut tout lire et convertir en chaine :

Extrait de la classe Terminal :

```
    public static StringBuffer lireFichier(String nomFichier)
    {
        try{
            File fichier = new File(nomFichier);
            FileInputStream fis = new FileInputStream (fichier);

            byte[] buffer = new byte[(int)fichier.length()];
            fis.read(buffer);
            fis.close();
            return(new StringBuffer(new String(buffer)));
        }
        catch(Exception ex)
            {
                return(null);
            }
    }

    public static void ecrireFichier(String nomFichier,
                                     StringBuffer strbuf)
```

```
    {
        try{
            File fichier = new File(nomFichier);
            FileOutputStream fos = new FileOutputStream (fichier);

            byte[] buffer = strbuf.toString().getBytes();
            fos.write(buffer);
            fos.close();
        }
        catch(Exception ex)
            {
                exceptionHandler(ex);
            }
    }
```

7. Les sockets

7.1. Principe et définition

Dans le contexte logiciel, un socket (mot anglais qui signifie prise) est un tuyau de communication permettant de faire communiquer deux processus entre eux qu'ils soient situés sur la même machine ou sur deux machines différentes

Une JVM étant un processus, le socket est un moyen de faire communiquer deux JVM entre elles.

Le socket est caractérisé par :

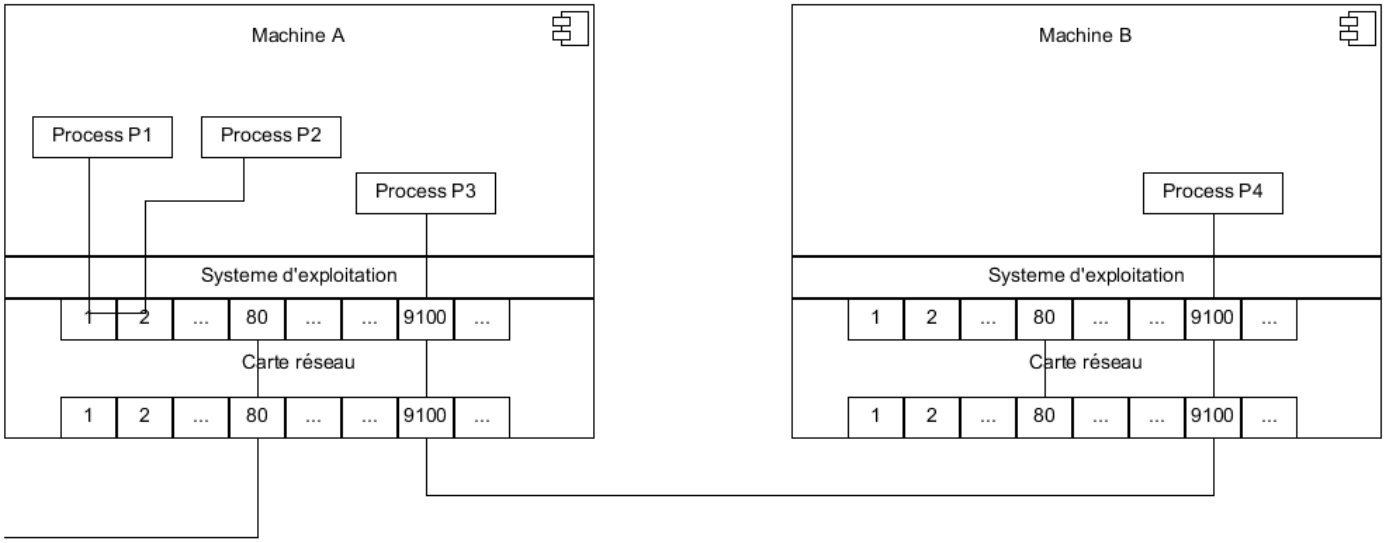
- l'addresse IP ou le hostname de la machine qui crée le socket (machine A)
- le port de la machine (Machine A) qui crée le socket
- l'addresse IP ou le hostname de la machine qui accepte la communication socket (Machine B)
- le port de la machine qui accepte la communication socket (Machine B)

Le port est un **numéro interne** à chaque machine.

Pour créer un socket, il faut qu'il existe au préalable un Serveur de Socket qui gère un port.

Le port utilisé par celui qui crée un socket (Machine A) est par défaut choisi automatiquement sur un port libre. Nous verrons en TP qu'il est possible d'imposer ce port.

Au sens de la carte réseau, la notion de port est bien une notion logique, géré par le système d'exploitation. Un port est une porte ouverte par l'OS au monde extérieur.



Des plages de port sont « réservés » (c'est une convention et non une obligation matérielle) de 0 à 1024. Exemples :

2, pour l'accès à un shell sécurisé Secure SHell, également utilisé pour l'échange de fichiers sécurisés SFTP

23, pour le port telnet

25, pour l'envoi d'un courrier électronique via un serveur dédié SMTP

53, pour la résolution de noms de domaine en adresses IP : DNS

67/68, pour DHCP et bootpc

80, pour la consultation d'un serveur HTTP par le biais d'un Navigateur web

110, pour la récupération de son courrier électronique via POP

123 pour la synchronisation de l'horloge : Network Time Protocol (NTP)

143, pour la récupération de son courrier électronique via IMAP

389, pour la connexion à un LDAP

443, pour les connexions HTTP utilisant une surcouche de sécurité de type SSL : HTTPS

Le principe du Firewall est d'autoriser et filtrer l'accès à ces ports.

7.2. Les classes prédéfinies Java

Les classes utilisées sont **Socket** et **ServerSocket** du package **java.net**.

La classe *ServerSocket* permet côté de serveur de se mettre en attente de la création d'un socket sur un port donné.

La classe *Socket* permet de définir un objet **InputStream** et **OutputStream** afin de lire ou d'écrire des informations. Un socket se manipule donc comme un fichier.

7.3. Exemple: cas 1 (Exemple 33 Cas 1sur le site)

Le serveur attend une seule requête du client. Le client envoie une seule requête.
~~Le client est en attente de lecture de la réponse.~~

Le serveur :

```
import java.io.*;
import java.awt.*;
import java.net.*;

public class Serveur1
{
   public static void main(String args[]) throws Exception
   {
        String str;
        ServerSocket ssoc;
        ssoc = new ServerSocket(9999);

        System.out.println("En attente...");
        Socket soc = ssoc.accept();
        System.out.println("Socket accepte");

        InputStream is = soc.getInputStream();
        DataInputStream dis = new DataInputStream(is);

        System.out.println("Lecture du socket");
        str = dis.readUTF();
        System.out.println("RECU: "+str);

        soc.close();

   }
}
```

Le client :

```
import java.awt.*;
import java.io.*;
import java.net.*;

public class Client1
{
   static public void main(String args[]) throws Exception
   {
        System.out.println("Creation du socket");
        Socket soc = new Socket("localhost",9999);

        OutputStream os=soc.getOutputStream();
        DataOutputStream dos=new DataOutputStream(os);

        DataInputStream in = new DataInputStream(System.in);
        System.out.print("> ");
        System.out.flush();
        String valeur= in.readLine();

        dos.writeUTF(valeur);

        soc.close();
   }
}
```

Le serveur est en attente de la part d'un client de la création d'un socket:
ssoc = new ServerSocket(9100); Le serveur attend sur le port 9100 à son adresse IP
Socket soc = ssoc.accept();
Le client crée le socket :
Socket soc = new Socket(InetAddress.getLocalHost(),9100);
Le premier paramètre est l'adresse IP du serveur. On peut utiliser :
"localhost", "192.168.0.1", InetAddress.getLocalHost() pour désigner l'IP du serveur du poste local, ou
l'adresse IP du serveur si le serveur n'est pas sur la même machine que le client.
L'instruction de lecture readUTF sur le socket est en attente d'écriture du client :
str = dis.readUTF();

7.4. Exemple : cas 2 (Exemple 33 Cas2 sur le site)

Le serveur attend de multiples requêtes du client. Le client envoie de multiples requêtes au serveur. Le client ne reçoit pas de retour du serveur.

Le serveur :

```
public class Serveur2
{
   public static void main(String args[]) throws Exception
   {
        String str;
        ServerSocket ssoc;
        ssoc = new ServerSocket(9999);

        System.out.println("En attente...");
        Socket soc = ssoc.accept();
        System.out.println("Socket accepte");

        InputStream is = soc.getInputStream();
        DataInputStream dis = new DataInputStream(is);
```

```
         while(true)
              {
                   System.out.println("Lecture du socket");
                   str = dis.readUTF();
                   System.out.println("RECU: "+str);
              }
   }
}
```

Le client :

```
public class Client2
{
   static public void main(String args[]) throws Exception
   {
        Socket soc = new Socket(InetAddress.getLocalHost(),9999);

        OutputStream os=soc.getOutputStream();
        DataOutputStream dos=new DataOutputStream(os);

       while(true)
              {
                   DataInputStream in = new DataInputStream(System.in);
                   System.out.print("> ");
                   System.out.flush();
                   String valeur= in.readLine();

                   dos.writeUTF(valeur);
              }
   }
}
```

Le serveur boucle sur la lecture du socket.
Le client boucle sur l'écriture du socket.

On peut s'apercevoir que le socket est géré comme un buffer : si le client écrit plus vite que le serveur lit, alors les informations s'accumulent dans le socket.

7.5. Exemple : avec une IHM (Exemple 35 sur le site)

Le même cas de fonctionnement que le cas précédent mais avec une Ihm côté client et côté serveur afin d'afficher les informations saisies et reçues.

7.6. Exemple : cas 3 (Exemple 33 sur le site)

Identique à l'exemple précédent (cas 2) mais le serveur répond au client comme quoi il a traité la requête du client.

On voit ici que la communication sur un socket se fait bien dans les deux sens.

Le serveur :

```
public class Serveur3
{
   public static void main(String args[]) throws Exception
   {
        String str;
        ServerSocket ssoc;
        ssoc = new ServerSocket(9999);

        System.out.println("En attente...");
        Socket soc = ssoc.accept();
        System.out.println("Socket accepte");

        InputStream is = soc.getInputStream();
        OutputStream os = soc.getOutputStream();
        DataInputStream dis = new DataInputStream(is);
        DataOutputStream dos = new DataOutputStream(os);

        while(true)
             {
                  System.out.println("Lecture du socket");
                  str = dis.readUTF();
                  System.out.println("RECU: "+str);
                  dos.writeUTF("RECU");
                  dos.writeUTF("RECU2"); // Pour faire un test
             }
   }
}
```

Le client :

```
public class Client3
{
   static public void main(String args[]) throws Exception
   {
        Socket soc = new Socket(InetAddress.getLocalHost(),9999);

        OutputStream os=soc.getOutputStream();
        InputStream is = soc.getInputStream();
        DataOutputStream dos=new DataOutputStream(os);
        DataInputStream dis = new DataInputStream(is);

       while(true)
             {
                  DataInputStream in = new DataInputStream(System.in);
                  System.out.print("> ");
                  System.out.flush();
                  String valeur= in.readLine();

                  dos.writeUTF(valeur);

                  String rep = dis.readUTF();
                  System.out.println("REPONSE: "+ rep);
             }
   }
}
```

On voit bien dans ce dernier exemple que la communication se fait bien dans les 2 sens.

7.7. Exemple : cas 3bis (Exemple 33 sur le site)

Dans le serveur, la ligne dos.writeUTF("RECU2"); permet de montrer que le socket est géré sous la forme d'un buffer (flot de données). Les données envoyées par le serveur s'accumulent dans le socket.

Si on veut que le client lise toutes les réponses du serveur :

```
public class Client3bis
{
   static public void main(String args[]) throws Exception
   {
        Socket soc = new Socket(InetAddress.getLocalHost(),9999);

        OutputStream os=soc.getOutputStream();
        InputStream is = soc.getInputStream();
        DataOutputStream dos=new DataOutputStream(os);
        DataInputStream dis = new DataInputStream(is);

       while(true)
            {
                DataInputStream in = new DataInputStream(System.in);
                System.out.print("> ");
                System.out.flush();
                String valeur= in.readLine();

                dos.writeUTF(valeur);

                String rep;
                boolean lecture=true;
                while(lecture)
                    {
                        rep = dis.readUTF();
                        System.out.println("REPONSE: "+ rep);
                        if(dis.available()==0)lecture=false;
                    }
            }
   }
}
```

La boucle while(lecture) permet de lire toutes les informations écrites sur le socket par le serveur.

Ceci met en évidence un problème de fond: le "protocole logique" que le client et le serveur doivent adopter en fonction des informations échangées.

Cette problématique peut être réduite en typant les informations échangées. Nous verrons que dans une architecture distribuée cette problématique tombe d'elle même.

7.8. Exemple : cas 4 (Exemple 33 sur le site)

Dans les deux exemples précédents on peut s'apercevoir que le serveur ne peut pas gérer plus de un client. Si on lance un deuxième client, il est en attente de l'instruction "accept" du serveur de socket qui n'arrivera jamais puisqu'en dehors de la boucle.

La solution est, sur le serveur, de mettre le "accept" dans la boucle. Le client demande alors à chaque requête un nouveau socket.

Le Serveur :

```
public class Serveur4
```

```
{
   public static void main(String args[]) throws Exception
   {
        String str;
        ServerSocket ssoc;
        ssoc = new ServerSocket(9999);

        while(true)
             {
                 System.out.println("En attente...");
                 Socket soc = ssoc.accept();
                 System.out.println("Socket accepte");

                 InputStream is = soc.getInputStream();
                 DataInputStream dis = new DataInputStream(is);

                  System.out.println("Lecture du socket");
                  str = dis.readUTF();
                  System.out.println("RECU: "+str);

                 soc.close();
             }
   }
}
```

Le Client :

```
public class Client4
{
   static public void main(String args[]) throws Exception
   {

       while(true)
             {

                DataInputStream in = new DataInputStream(System.in);
                System.out.print("> ");
                System.out.flush();
                String valeur= in.readLine();

                Socket soc = new Socket(InetAddress.getLocalHost(),9999);

                 OutputStream os=soc.getOutputStream();
                 DataOutputStream dos=new DataOutputStream(os);

                dos.writeUTF(valeur);

                 soc.close();
             }
   }
}
```

7.9. La communication des objets sur un socket : Exemple 34 du site

Typer une information consiste à échanger des données qui correspondent à des classes d'objet. On réalise ce mode de transfert avec le **principe de sérialisation**.

Cet exemple montre des exemples d'utilisation de socket dans le cadre d'une communication d'objet en utilisant le principe de sérialisation.

Les cas 1 ~~et 2~~ introduit le sujet. Ce cas n'utilise pas le principe de sérialisation.

Les autres cas utilisent le principe de sérialisation.

Tous les cas de cet exemple se compilent et s'exécutent de la manière suivante :
Compilations :
javac Client.java
javac Serveur.java
Exécutions:
java Serveur
java Client (dans une autre fenêtre)

7.9.1. Cas 1

On veut envoyer un objet java de classe Individu (nom, prénom, age).
Le client écrit chaque champ de l'individu et le serveur les lit champ à champ.

7.9.2. Cas 2

Utilisation du principe de sérialisation.
La classe Individu implements l'interface Serializable.
Le flot de lecture et d'écriture n'est plus un DataOutputStream (resp DataInputStream) mais un ObjectOutputSream (resp. ObjectInputStream).
L'encodage de l'objet dans le socket devient transparent mais l'objet écrit et lu est de la classe Object.

7.9.3. Cas 3

Le serveur peut recevoir des objets de classes différentes. Exemple: Individu ou Complexe.
Si il est nécessaire de créer le véritable objet, il est indispensable d'utiliser l'opérateur **instanceof** afin de tester la classe d'appartenance.
Dans le sous-cas **Serveur2**, il est inutile d'utiliser instanceof car la méthode appelée (toString) est générique pour les deux classes.

==========

HORS PROGRAMME :

7.9.4. Cas 4

Le client et le serveur s'échangent les objets Java en utilisant le XML.
Dans ce cas de sérialisation, il faut ouvrir un nouveau socket entre chaque requête.

La sérialisation utilisée est celle native de Java : java.beans

Le package java.beans permet le développement de "beans".
Un beans est ....

Une propriété essentielle d'un beans est de pouvoir être échanger entre un serveur et un client. Le serveur contient des beans devant être déployés sur des postes clients. Ce serveur est la plupart du temps un serveur HTTP et les clients des environnements d'exécution Java (Applet, JavaWebStart, ...). Pour cela, le beans doit transiter par un flux de données (socket), le beans doit donc être sérialisé puis dessérialisé. De plus, les environnements d'exécution du client et du serveur peuvent être de version java différente. Il faut donc rendre robuste la sérialisation en n'utilisant pas le principe binaire de sérialisation des objets Java mais un principe de sérialisation dont le format soit textuel (universel) et dont la syntaxe soit la plus ouverte possible. Le format tout trouvé est le XML.

C'est ainsi que le package java.beans possède les classes XMLDecoder et XMLEncoder permettant de sérialiser et dessérialiser n'importe quel beans.

En Java, un beans est une classe Java :

- qui possède au moins un constructeur sans paramètre
- dont certains attributs privés possèdent les méthodes d'accès et de modification (get et set)

Seuls les attributs possédant ces accesseurs et ces assesseurs sont sérialisés au format XML.

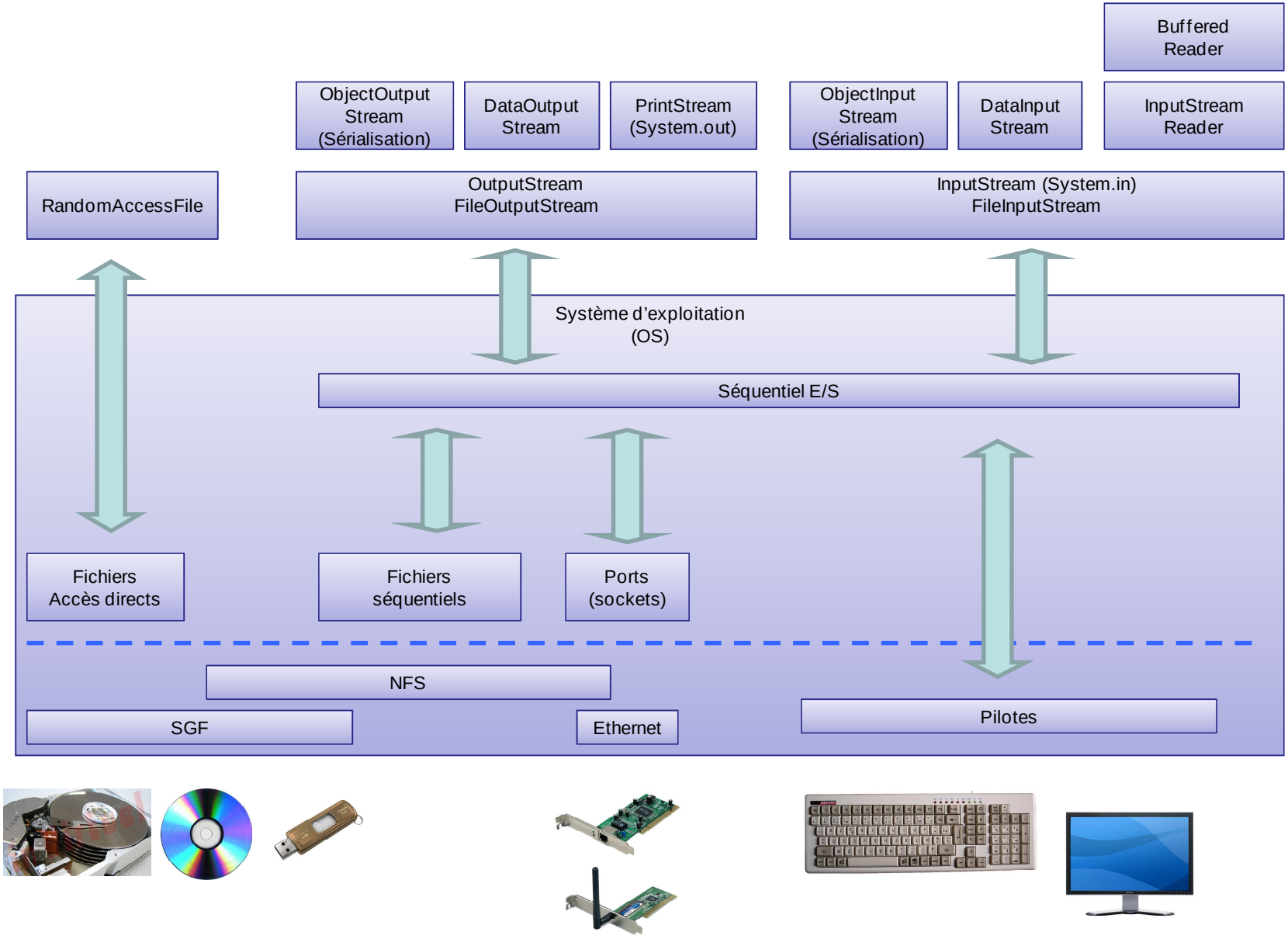
7.9.5. Cas5

Un cas qui fonctionne en encodant toutes les données en chaîne et en utilisant les encodeurs et les l la technologie Beans.

7.9.6. Cas6

Idem cas précédent mais en utilisant un streamer DOM.

8. Schéma de synthèse


Buffered Reader
ObjectOutput Stream (Sérialisation)
DataOutput Stream
PrintStream (System.out)
ObjectInput Stream (Sérialisation)
DataInput Stream
InputStream Reader
RandomAccessFile
OutputStream FileOutputStream
InputStream (System.in) FileInputStream
Système d'exploitation (OS)
Séquentiel E/S
Fichiers Accès directs
Fichiers séquentiels
Ports (sockets)
NFS
SGF
Ethernet
Pilotes

9. Exemple complet sur la bibliothèque

9.1. Exemple 17

Exemple de l'utilisation de DataInputStream et DataOutputStream pour charger et sauvegarder les données de la bibliothèque dans un fichier biinaire.

Voir sur le site http://jacques.laforgue.free.fr
l'exemple **Exemple17_Bibliotheque**

9.1. Exemple 18

Exemple de l'utilisation de la Serialisation avec ObjectInputStream et ObjectOutputStream pour charger et sauvegarder les données de la bibliothèque dans un fichier biinaire.

Voir sur le site http://jacques.laforgue.free.fr
l'exemple **Exemple18_Bibliotheque**

Les classes Livre, Film et Jeu doivent implémenter l'interface Serializable.
Soit elles le font directement ou soit la classe MultiMedia implémente l'interface Serializable. Cela est suffisant car les classes Livre, Film et Jeu héritent de MultiMedia.