Langage Java Chapitre 8

Les flux, les fichiers.
La sérialisation des objets.

La plupart des exemples donnés sont essentiellement extraits de la documentation Oracle.

1. Les flux d'entrée sortie

1.1 Notion de flux d'entrée/sortie – I/O stream-

Un flux d'entrée/sortie – I/O stream – est une source/destination de données. Un flux représente toutes sortes de sources et de destinations comme les fichiers, la mémoire, les sockets réseaux, certains périphériques…
Java implémente un grand nombre de classes pour gérer les flux d'entrée/sortie.

1.2 Les byte Stream

Les **byte Stream** sont utilisés pour transférer des **données 8 bits** (byte). Toutes les classes manipulant les **byte** descendent des classes **InputSream** et **OutputStream**.

Il y a beaucoup de classes qui manipulent les données 8 bits.

Les exemples suivants de Oracle utilisent les classes **FileInputStream** et **FileOutputStream**. L'exemple suivant montre la copie d'un fichier dans un autre. Le fichier à copier "xanadu.txt" doit être préalablement créé avec un éditeur de texte.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes {
   public static void main(String[] args) throws IOException {
      FileInputStream in = null;
      FileOutputStream out = null;
      try {
         in = new FileInputStream("xanadu.txt");
         out = new FileOutputStream("outagain.txt");
         int c;

         while ((c = in.read()) != -1) {
            out.write(c);
         }
      } finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
```

```
            out.close();
        }
    }
}
```

La méthode **read()** retourne une valeur de type **int**: elle retourne soit **-1** quand la fin de fichier est atteinte, soit le **caractère lu placé dans l'octet de poids faible de l'entier retourné**.

Il est impératif de fermer les flux, c'est pour cela qu'un bloc **finally** est utilisé.

Cet exemple montre une manipulation de flux à bas niveau, la documentation Java conseille d'utiliser les **characters streams** plutôt que des **byte stream**.

1.3 Les Character Stream pour les flux "texte"

Java stocke les caractères en utilisant le standard Unicode. A la lecture comme à l'écriture, les **Character Stream I/O** traduisent automatiquement ce format dans le format local. Un des intérêts des flux de caractères est la préparation des données pour l'internationalisation des programmes (fonctionnement du programme quelque soit le langage, la région…).
Les flux de caractères spécialisés utilisés ici sont **FileReader** et **FileWriter**.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CopyCharacters {
    public static void main(String[] args) throws IOException {
        FileReader inputStream = null;
        FileWriter outputStream = null;

        try {
            inputStream = new FileReader("xanadu.txt");
            outputStream = new FileWriter("characteroutput.txt");

            int c;
            while ((c = inputStream.read()) != -1) {
                outputStream.write(c);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

La méthode **read()** return une valeur de type **int**: elle retourne soit **-1** quand la fin de fichier est atteinte, soit **le caractère lu placé dans les 16 bits de poids faible de l'entier retourné**.

1.4 Les flux "texte" organisé en lignes

Un flux texte est souvent constitué d'une suite de lignes séparées par la séquence « Retour à la ligne-Saut de ligne ("\r\n") », un simple « Retour à la ligne ("\r") » ou un simple « Saut de ligne ("\n") ».
Les classes **BufferedReader** et **PrintWriter** disposent respectivement d'une méthode de lecture et d'écriture d'une ligne de texte.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;

public class CopyLines {
   public static void main(String[] args) throws IOException {
      BufferedReader inputStream = null;
      PrintWriter outputStream = null;

      try {
         inputStream =
            new BufferedReader(new FileReader("xanadu.txt"));
         outputStream =
            new PrintWriter(new FileWriter("characteroutput.txt"));

         String l;
         while ((l = inputStream.readLine()) != null) {
            outputStream.println(l);
         }
      } finally {
         if (inputStream != null) {
            inputStream.close();
         }
         if (outputStream != null) {
            outputStream.close();
         }
      }
   }
}
```

La méthode **readLine()** retourne une ligne de texte, la méthode **println()** envoie une ligne de texte en ajoutant le caractère de terminaison de la ligne qui peut être différent.

L'API **Scanning** permet d'analyser un flux texte, de choisir des séparateurs et de convertir ce flux texte en segments de texte.
La plupart des flux objet implémentent l'API Formatting afin de fournir des données humainement accessibles et lisibles.

1.5 Les flux tamponnés Buffered Stream

Les classes vues aux paragraphes 1.2 et 1.3 utilisent des Entrée/Sortie non tamponnées, chaque transfert est alors transmis directement au système d'exploitation, ce qui peut entrainer un ralentissement du fonctionnement du programme.
Java fournit des classes de flux tamponnées, les données sont stockées dans un tampon (buffer), ce qui réduit considérablement les appels à l'OS.
Java fournit 4 classes de flux tamponnées :

- **BufferedInputStream** et **BufferedOutputStream** pour créer des flux tamponnés d'octets.
- **BufferdReader** et **BufferedWriter** pour créer des flux tamponnés de caractères.

Exemple:

```
BufferedInputReader    inputbyte    ;
BufferedOuputWriter    outputbyte  ;

inputbyte     =  BufferedInputStream(new FileOutputStream("fichierdatain"));
outputbyte   =  BufferedOutputStream(new FileOutputStream("fichierdataout"));

BufferedReader    inputStream   ;
BufferedWriter    outputStream  ;

inputStream =  new BufferedReader(new FileReader("xanadu.txt"));
outputStream = new BufferedWriter(new FileWriter("characteroutput.txt"));
```

Attention : vidage du tampon et écriture dans le flux correspondant

Les écritures des données dans des flux tamponnés nécessitent parfois l'obligation de forcer le vidage du tampon qui consiste à écrire les données qu'il contient dans le flux destination. Cette opération est réalisée par la méthode **flush()** du flux de sortie tamponné.

Exemple

```
        PrintWriter outputStream = null;

        // On s'assure d'écrire toutes les données avant de fermer le flux
        outputstream.flush();
        outputStream.close();
```

L'interface Flushable ne contient que la méthode **flush()**, la documentation Java sur l'interface **Flushable** donne la liste de tous les flux de sortie qui implémentent cette interface.

1.6 Exercices

- Etudier et tester l'exemple 1.2.
- Etudier et tester l'exemple 1.3.
- Etudier et tester l'exemple 1.4.
- Etudier la classe FileWriter. Peut-on utiliser cette classe pour écrire à la fin d'un fichier ? Préciser alors le constructeur à utiliser.

- Ecrire un programme permettant de concaténer 2 fichiers en 1.
- Ecrire une classe CreerFichier qui sauvegarde tout ce qui est tapé au clavier dans un fichier jusqu'à ce qu'on tape la chaine "fin" (sur une seule ligne).
- Ecrire une classe CompareFichiers qui compare 2 fichiers et affiche si les 2 fichiers sont identiques.
- Ecrire une classe CompareDiffFichiers qui compare 2 fichiers et affiche toutes les différences de la manière suivante :
    - Ligne 5 caractère a e
    - Ligne 7 caractère d c
    - Ligne 7 caractère u i

  On suppose que le 1er caractère indiqué appartient au 1er fichier et le 2ème caractère au 2ème fichier.

2. Les flux d'entrée/sortie standards : le clavier et l'écran

Le flux d'entrée clavier **System.in** est un flux d'octets **InputStream**.
Habituellement, on encapsule **System.in** dans un **InputStreamReader** par :

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

Transformant ainsi ce flux d'octets en flux de caractères.

Puis, afin de faciliter la lecture du clavier, on encapsule l'objet **InputStreamReader** dans un **BufferedReader** par :

```
BufferedReader in   = new BufferedReader(cin) ;
```

Ou, plus directement:

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

On dispose ainsi de la méthode **readLine()** pour lire le clavier ligne par ligne.

La classe Console

Java Edition 6 propose une classe **Console** pour accéder au clavier-écran standard.

```
import java.io.Console;

public class TestConsole {
  public static void main (String argv[]) {
     Console c = System.console() ;
     String s = c.readLine() ;
     c.printf("lu : " + s + "\n") ;
  }
}
```

Autre exemple:

```
import java.io.Console;
```

```
public class TestConsole {
  public static void main (String argv[]) {
     Console c = System.console() ;
      String login = c.readLine("Taper votre login: ");
      c.printf("login: "+login +"\n") ;
  }
}
```

L'objet Console fournit une méthode **readPassword()** sécurisée de lecture de mot de passe. Cette méthode affiche la demande du mot de passe, saisit le mot de passe, le retourne dans un tableau de caractères et supprime l'écho à l'écran.

```
        char [] newPassword1 =   c.readPassword("Taper votre mot de passe : ");
```

Exercice : Etudier le programme suivant extrait de la documentation Oracle.

```
import java.io.Console;
import java.util.Arrays;
import java.io.IOException;

public class Password {

   public static void main (String args[]) throws IOException {

      Console c = System.console();
      if (c == null) {
         System.err.println("No console.");
         System.exit(1);
      }

      String login = c.readLine("Enter your login: ");
      char [] oldPassword = c.readPassword("Enter your old password: ");

      if (verify(login, oldPassword)) {
         boolean noMatch;
         do {
            char [] newPassword1 =
               c.readPassword("Enter your new password: ");
            char [] newPassword2 =
               c.readPassword("Enter new password again: ");
            noMatch = ! Arrays.equals(newPassword1, newPassword2);
            if (noMatch) {
               c.format("Passwords don't match. Try again.%n");
            } else {
               change(login, newPassword1);
               c.format("Password for %s changed.%n", login);
            }
            Arrays.fill(newPassword1, ' ');
            Arrays.fill(newPassword2, ' ');
         } while (noMatch);
      }
```

```
        Arrays.fill(oldPassword, ' ');
    }

    //Dummy verify method.
    static boolean verify(String login, char[] password) {
        return true;
    }

    //Dummy change method.
    static void change(String login, char[] password) {}
}
```

Le flux standard de sortie est un objet **System.out** instance de la classe **PrintStream**. Cette classe propose les méthodes **print()** et **println()** mais également une méthode **format()** qui s'utilise comme le montre l'exemple suivant:

```
public class Root2 {
    public static void main(String[] args) {
        int i = 2;
        double r = Math.sqrt(i);
        System.out.format("The square root of %d is %f.%n", i, r);
    }
}
```

Voici la sortie : The square root of 2 is 1.414214.
Il est conseillé d'utiliser %n pour effectuer un retour à la ligne suivante à la place de "\n".

Voir le lien **http://java.Oracle.com/javase/6/docs/api/java/util/Formatter.html#syntax** pour la syntaxe.

3. Les classes DataInputStream et DataOutputStream

Les classes **DataInputStream** et **DataOutputStream** sont très pratiques pour **lire/écrire des données de type primitifs (short, int, long, float, double, char, byte et boolean) ainsi que des String.**

Elles possèdent les méthodes appropriées pour lire/écrire des données dont le type est précisé ci-dessus.
Par exemple pour la classe **DataOuputStream** :

writeDouble(double v) ;
writeInt(int v) ;
writeUTF(String v) ;

Exercice : Etudier et tester l'exemple suivant de Oracle.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.IOException;
import java.io.EOFException;
```

```
public class DataStreams {
    static final String dataFile = "invoicedata";

    static final double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
    static final int[] units = { 12, 8, 13, 29, 50 };
    static final String[] descs = { "Java T-shirt",
            "Java Mug",
            "Duke Juggling Dolls",
            "Java Pin",
            "Java Key Chain" };

    public static void main(String[] args) throws IOException {
        DataOutputStream out = null;

        try {
            out = new DataOutputStream(new
                    BufferedOutputStream(new FileOutputStream(dataFile)));

            for (int i = 0; i < prices.length; i ++) {
                out.writeDouble(prices[i]);
                out.writeInt(units[i]);
                out.writeUTF(descs[i]);
            }
        } finally {
            out.close();
        }

        DataInputStream in = null;
        double total = 0.0;
        try {
            in = new DataInputStream(new
                    BufferedInputStream(new FileInputStream(dataFile)));
            double price;
            int unit;
            String desc;

            try {
                while (true) {
                    price = in.readDouble();
                    unit = in.readInt();
                    desc = in.readUTF();
                    System.out.format("You ordered %d units of %s at $%.2f%n",unit, desc, price);
                    total += unit * price;
                }
            } catch (EOFException e) { }
            System.out.format("For a TOTAL of: $%.2f%n", total);
        }
        finally {
            in.close();
        }
```

```
    }
}
```

4. La classe File

La classe **File** permet certaines opérations notamment sur les propriétés des fichiers.
On accède à un fichier par son nom, on peut obtenir certaines informations sur ce fichier et notamment :

Son accessibilité en lecture par la méthode **canRead()**.
Son accessibilité en écriture par la méthode **canWrite().**
Son chemin absolu par la méthode **getAbsoluthPath().**
Sa taille par la méthode **length().**
Son nom par la méthode **getName().**
Son répertoire par la méthode **getparent().**
Si c'est un fichier par la méthode **isFile().**
Si c'est un répertoire par la méthode **isDirectory().**
Si il existe par la méthode **exists()**.
…

Il est possible de connaître le contenu d'un répertoire grâce à la méthode

```
public  String[] list()
```

qui retourne dans un tableau les noms des fichiers/répertoires qu'il contient.

La méthode **mkdir()** permet de créer un répertoire.

Exercice :

41 Afficher à l'écran le contenu d'un répertoire dont le nom sera saisi au clavier. Préciser si l'élément affiché est un fichier ou un répertoire.

42 Etudier la méthode **public File[] listFiles().** Afficher à l'écran le contenu d'un répertoire, de ses sous répertoires et ainsi de suite. Il faut utiliser le principe de la récursivité (la méthode d'analyse d'un répertoire s'appelle elle-même si l'élément trouvé est un répertoire).

5. La classe RandomAccessFile

Cette classe permet des accès non séquentiels ou aléatoires au contenu d'un fichier. Elle dispose d'un grand nombre de méthodes de lecture et d'écriture avec en plus quelques méthodes pour se déplacer à l'intérieur du fichier :
void seek(long) — Déplace le pointeur de fichier du nombre d'octets précisé.
long getFilePointer() — Retourne la position du pointeur de fichier.

6. La sérialisation

La sérialisation est utilisée pour rendre les objets persistants afin de les stocker dans un fichier, dans une base de données, de les envoyer par le réseau ou en utilisant tout autre moyen de stockage.

Le processus inverse de la restauration est connu sous le nom de dé-sérialisation. Dans ce chapitre, nous allons voir comment utiliser la sérialisation et la dé-sérialisation.

Un objet sérialisé est dit persistant. La plupart des objets en mémoire sont transitoires, ce qui veut dire qu'ils disparaissent quand leur référence est hors de portée, que l'application se termine ou que l'ordinateur est éteint. Les objets persistants existent tant qu'un exemplaire d'eux reste stocké quelque part sur un disque, une cartouche ou un CD-ROM.

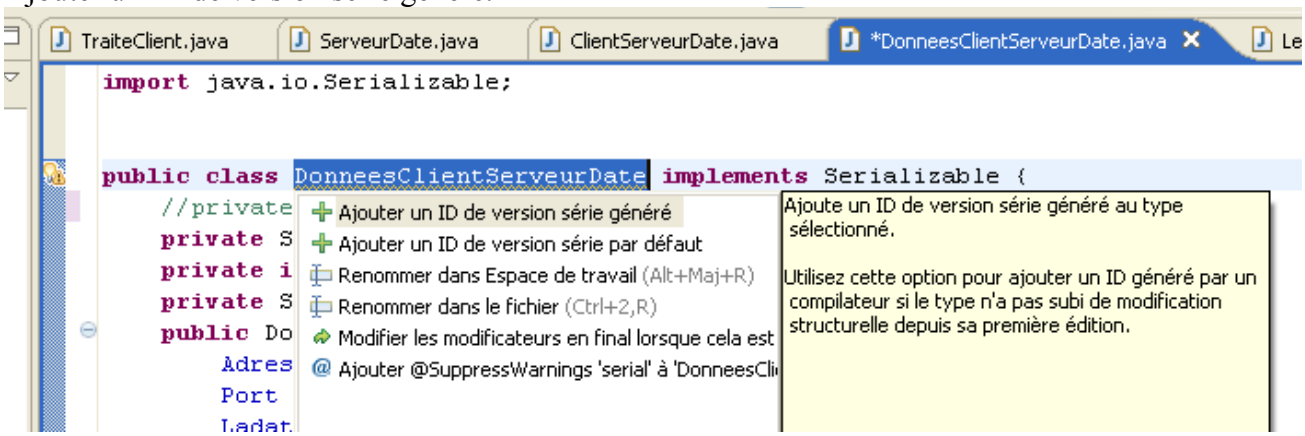
La sérialisation est une fonctionnalité apparue la première fois dans le JDK 1.1. La prise en charge par Java de la sérialisation se compose essentiellement de **l'interface Serializable**, des **classes ObjectOutputStream** et **ObjectInputStream**. Nous allons illustrer ces trois éléments par une application qui enregistre sur disque un objet Abonne et le relit.

Afin de pouvoir sérialiser un objet d'une classe donnée, celle-ci doit implémenter l'interface **Serializable** ou hériter d'une classe elle-même sérialisable.
L'interface **Serializable** ne possède aucun attribut et aucune méthode, elle sert uniquement à identifier une classe sérialisable.
Un objet sérialisé entraîne la sérialisation de ses attributs. Tous les attributs de l'objet sont sérialisés mais à certaines conditions.
Pour être sérialisé, l'attribut doit :
· être lui-même sérialisable ou être un type primitif (qui sont tous sérialisables),
· ne pas être déclaré à l'aide du mot clé static,
· ne pas être déclaré à l'aide du mot clé transient,
· ne pas être hérité d'une classe mère sauf si celle-ci est elle-même sérialisable.

Le processus de sérialisation associe un numéro de version à chaque classe sérialisable appelé **serialVersionUID**. Il faut normalement déclarer cet attribut de type long dans la classe à sérialiser de la manière suivante :

```
private static final long serialVersionUID = 4871459982298687085L;
```

L'ajout de cet attribut ainsi que sa valeur peut être fait automatiquement avec Eclipse. On écrit l'implémentation de l'interface Serializable, puis bouton droit de la souris sur l'avertissement en jaune sur la ligne public class NomClasse…. puis Correctif rapide puis Ajouter un ID de version série généré.



Exemple d'utilisation de l'interface Serializable

Tout objet prévu pour être sérialisé doit implémenter l'interface **Serializable**. Si, par exemple, vous essayez de sérialiser un objet qui n'implémente pas cette interface, une **NotSerializableException** sera déclenchée.
La classe Abonne implémente l'interface **Serializable,** les objets pourront donc être sérialisés.

```
import java.io.*;

public class Abonne implements Serializable{
  private static final long serialVersionUID = 1L;
  String NomAbonne;
  String  NumeroDappel;

  public Abonne(String NomAbo,String NumeroDap) {
    NomAbonne = NomAbo;
    NumeroDappel = NumeroDap;
  }

  public void afficherNom()
  {
     System.out.println(NomAbonne);
  }

   public void afficherNumero()
  {
     System.out.println(NumeroDappel);
  }
  public void afficherAbonne()
  {
     System.out.println("Nom : "+NomAbonne+ " Numero : "+NumeroDappel);
  }
}
```

Objets flux nécessaires à la sérialisation :

- Un objet **`FileOutputStream`** `fos, qui` permet de donner un nom de fichier sur disque:
  file1 = new FileOutputStream(NomDuFichier);
- Un objet **`ObjectOutputStream`** `out,` qui est un flux de sortie avec des méthodes de traitement des objets en sortie vers le fichier.
  out = new ObjectOutputStream(fos);

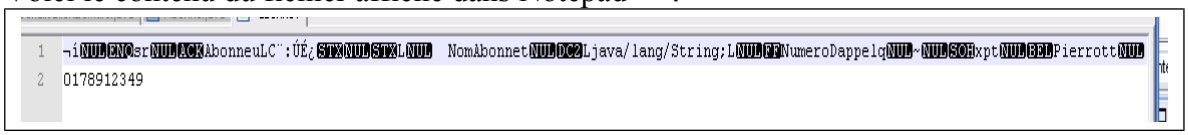
Puis, pour écrire l'abonné dans le fichier :
out.writeObject(abo);

Puis, pour fermer le flux :
out.close();

```
import java.io.ObjectOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;
```

```
public class SerialisationEcriture
 {
public static void main(String [] args)
 {
  String filename = "abonne1";
  if(args.length > 0)    {
      filename = args[0];
   }
   Abonne abo = new Abonne("Pierrot","0078912349");
   FileOutputStream fos = null;
   ObjectOutputStream out = null;
   try   {
            try{
                     fos = new FileOutputStream(filename);
                     out = new ObjectOutputStream(fos);
                     out.writeObject(abo);
            }
            finally {
                     out.close();
            }
    }
    catch(IOException ex)    {
            ex.printStackTrace();
    }
   }
}
```

Voici le contenu du fichier affiché dans Notepad++ :



Objets nécessaires à la dé-sérialisation :

- Un objet **`FileInputStream`** `fis, qui` permet de donner le nom du fichier présent sur le disque:
  fis = new FileInputStream(NomDuFichier);
- Un objet **`ObjectInputStream`**, qui est un flux d'entrée avec des méthodes de traitement des données en entrée depuis le fichier.
  in = new ObjectInputStream(fis);

La lecture de l'objet par :

Abonne abo = (Abonne)in.readObject());

La fermeture du fichier par :

in.close();

```
import java.io.ObjectInputStream;
import java.io.FileInputStream;
import java.io.IOException;
```

```
public class SerialisationLecture
 {
public static void main(String [] args)  {
 String filename = "abonne1";
 if(args.length > 0)  {
      filename = args[0];
    }
    Abonne abo = null;
    FileInputStream fis = null;
    ObjectInputStream in = null;
    try {
      try   {
          fis = new FileInputStream(filename);
          in = new ObjectInputStream(fis);
           abo = (Abonne)in.readObject();
           abo.afficherAbonne() ;
       }
       finally {
         in.close();
       }
    }
    catch(IOException eio)  {
     eio.printStackTrace();
    }
    catch(ClassNotFoundException ex) {
                        ex.printStackTrace();
    }
  }
}
```

La méthode **readObject()** peut ne pas retourner un objet du type attendu et alors déclencher l'exception **ClassNotFoundException** qu'il est nécessaire d'attraper dans un bloc **catch**.

Exercices :

6.1 Tester les exemples donnés dans le paragraphe 6 : classes **Abonne** et **SerialisationEcriture**.

6.2 Vérifier le fonctionnement avec l'exemple suivant **SerialisationLecture**.

6.3 Reprendre la classe Annuaire du Cours Java Chapitre 5.

Enrichir la classe **Annuaire** avec 2 méthodes **sauvegarder()** et **relire()** pour respectivement stocker tous les abonnés du vecteur dans un fichier et relire ce fichier.
Proposer un programme de test qui crée des abonnés, sauvegarde ces abonnés dans un fichier puis relit ce fichier et affiche tous les abonnés.

Attention
Les fichiers d'objets ne fonctionnent pas en mode **append** (ajout). Lorsqu'on veut ajouter des objets à un tel fichier il faut le maintenir ouvert et le fermer quand la sauvegarde des objets est terminée.