Flux de *Sérialisation* d'objets

Introduction

Nous avons vu dans le chapitre qui concernait les bases des flux en Java comment écrire ou lire des caractères (du texte) à partir d'un fichier stocké sur un disque (local, périphérique ou distant, le principe est le même). Nous allons voir dans ce chapitre un principe de flux plus avancé : comment écrire et lire des *objets* dans un fichier, c'est la *sérialisation*. Les fichiers que vous dé*sérialisez* seront lus et chargés dans votre programme en conservant l'état dans lequel ils étaient quand ils ont été *sérialisés*. Par exemple, imaginez que vous créez un objet de type **Integer** (pas **int**, attention !), que vous l'initialisez avec la valeur **0**, que vous le modifiez ensuite pour avoir la valeur **1**, et que, pour finir, vous le sérialisez. Quand vous désérialiserez cet objet, il aura la valeur **1**. Nous allons voir comment créer des objets *sérialisables*, comment les sérialiser et les désérialiser, et comment empêcher que certaines variables des ces objets ne soient sérialisées.

L'interface *Sérializable*

Pour qu'une instance d'une classe (un objet) soit *sérialisable*, la classe doit implémenter l'interface **Serializable** (avec un **Z**). Cette interface est un peu particulière : une classe qui l'implémente n'a pas besoin de redéfinir ses méthodes vides. Il vous suffit donc d'écrire :

```
class maClasse implements Serializable {
// …
}
```

ObjectOutputStream

La classe **ObjectOutputStream** permet de sérialiser des objets et, accessoirement, des données issues des types de base (int, byte, char, double, float…).
Son constructeur admet comme argument un **FileOutputStream**. Examinez le code suivant :

```
class MaClasse implements Serializable {
    //…
}

class AutreClasse {
    MaClasse mc = new MaClasse() ; // créer une nouvelle instance de la classe MaClasse
    public AutreClasse() { // constructeur de la classe autreClasse
   try {
    FileOutputStream fos = new FileOutputStream("monFichier.dat"); // créer un nouveau fichier
    ObjectOutputStream oos = new ObjectOutputStream(fos); // ce fichier stockera des objets
    oos.writeObject(mc); // on écrit l'objet mc dans le flux oos
    oos.close() ; //fermer le flux
   } catch(IOException err) { ;} // ObjectOutputStream susceptible de générer des exceptions
    }
}
```

Vous pouvez trouver d'autres méthodes que **writeObject()** dans ObjectOutputStream :

- write(int)
- write(byte[])
- write(byte[], int, int) (une matrice, le premier élément à écrire et le nombre d'éléments suivants à écrire)
- writeBoolean(boolean)
- writeByte(int)
- writeBytes(String)

- writeChar(int)
- writeChars(String)
- writeDouble(double)
- writeFloat(float)
- writeInt(int)
- writeLong(long)
- writeShort(short)

ObjectInputStream

ObjectInputStream sert à récupérer des objets sérialisés. Elle fonctionne de façon similaire à **ObjectOutputStream**. Voici un exemple :

```
class EncoreAutreClasse {
  public EncoreAutreClasse() { // constructeur de la classe EncoreAutreClasse
   try {
   FileInputStream fis = new FileInputStream("monFichier.dat"); // lire ce fichier
   ObjectInputStream ois = new ObjectInputStream(fis); // ce fichier stocke des objets
   MaClasse mc2 = (MaClasse)ois.readObject() ; // lire un objet et le convertir en MaClasse
   ois.close() ; //fermer le flux
   } catch(IOException err) { ;} // ObjectInputStream susceptible de générer des exceptions
  }
}
```

Nous allons examiner la ligne 6 :
MaClasse mc2 = (MaClasse)ois.readObject() ;
Normalement mc^2 est égal à E ou –E, mais là je crois que j'ai fais une erreur de calcul ;-)
On crée un objet **mc2** de type **MaClasse**, on l'initialise ensuite grâce à l'instruction à droite du signe égale : on lit on objet (type de donnée Object, inexploitable à l'état brut) à parir du flux **ois** et on le convertit du type **Object** vers le type **MaClasse**.
De même, si un objet **String** avait été stocké dans notre fichier, nous aurions écrit :
String monString = (String)ois.readObject() ;
Ce procédé de conversion peut remplacer la méthode **toString()**.

Tout comme **ObjectOutputStream**, **ObjectInputStream** possède plein de méthodes qui servent à lire des types de base :

- read() (lire des octets, retourne des **int**)
- read(byte[], int, int) (la matrice, la position du premier élément à lire, le nombre d'éléments suivants à lire)
- readBoolean()
- readByte()
- readChar()
- readDouble()
- readFloat()
- readInt()
- readLine() (lire des String, mais String est un faux type de base, c'est en fait une classe, un objet, on utilisera donc de préférence readObject())
- readLong()
- readShort()
- readUnsignedByte() (retourne **int**, un vestige du C/C++…)
- readUnsignedShort() (de même)

Note : vous pouvez sérialiser plusieurs objets dans un même fichier en ayant recours plusieurs fois à la méthode writeObject() ou en utilisant les autres méthodes de ObjectOutputStream. Dans ce cas, vous devez appeler plusieurs fois les méthodes de ObjectInputStream. La règle est la suivant : le premier objet sérialisé est le premier fichier désérialisé, le second écrit est le second lu, etc…

Mot réservé *transcient*

La sérialisation ne vous servira le plus souvent pas à écrire et lire des données appartenant aux types de base mais plutôt des objets, donc des instances de classes. Pour économiser de la place et du temps d'écriture, on peut rendre certaines variables de classe non-sérialisables à l'aide du mot réservé **transcient**.
Par exemple, les variables **manger** et **dormir** ne seront pas sérialisée si on sérialise la classe **Activites** :

```
class Activites implements Serializable {
//…
public transcient int manger = 0 ;
transcient int dormir = 1;
private int travailler = 2 ;
protected int programmer = 3;
//…
}
```

Techniquement

Nous allons voir comment se fait la sérialisation d'un objet, théoriquement. Ceci est facultatif, et réservé aux utilisateurs aguerris.
Pour stocker un objet, on doit créer une *description* de la classe de laquelle il est issu :

- le nom de la classe
- une *empreinte* constituant une n° d'identification unique
- la description de la méthode de sérialisation (qui a changé entre Java 1.1 et Java 2)
- la description des champs de données

L'*empreinte* est calculée de la manière suivante :

- Description de la classe, de ses super-classes, des interfaces qu'elle implémente, des types des champs.
- Application de l'algorithme SHA à ces données.

En pratique un identificateur de classe est stocké ainsi:

- **72** (marqueur de début)
- longueur de la classe, sur **2 octets**
- nom de la classe
- empreint sur **8 octets**
- indicateur sur **1 octet**
- nombre de descripteurs de champs sur **2 octets**
- descripteur de champs de données
- **78** (marqueur de fin)
- type de superclasse (**70** s'il n'y en a pas)

L'indicateur de type de sérialisation est obtenu par 3 masques binaires :

```
java.io.ObjectStreamConstants :
static final byte SC_WRITE_METHOD = 1 ;
// données traditionelles
static final byte SC_SERIALIZABLE = 2;
// classe implémentant l'interface Serializable
static final byte SC_EXTERNALIZABLE = 3;
// classe qui implémente l'interface Externalizable
```

Descripteur de champs de format:

- B = byte
- C = char
- D = double
- F = float
- I = int
- J = long
- L = object
- S = short

- Z = boolean
- [= array (matrice)

Exemple concret:

Nous allons créer un programme qui contient une zone de texte (JTextField), une barre de progression (JProgressBar) et un bouton (JButton). Au clic sur le bouton, l'enchaînement suivant se produit :

- 1) Une méthode prend la valeur contenue dans le JTextField
  2) Elle transforme cette valeur du type String en Integer et la transmet à une seconde méthode.
- Une fois cette valeur obtenue, la seconde méthode la sérialise dans un fichier.
- Une troisième méthode est appelée : elle définie la valeur de la barre de progression avec la valeur de retour d'une quatrième méthode.
- La quatrième méthode désérialise la valeur sérialisée par la seconde méthode, le convertie en int pour la rendre utilisable comme *chiffre entier* et la retourne.

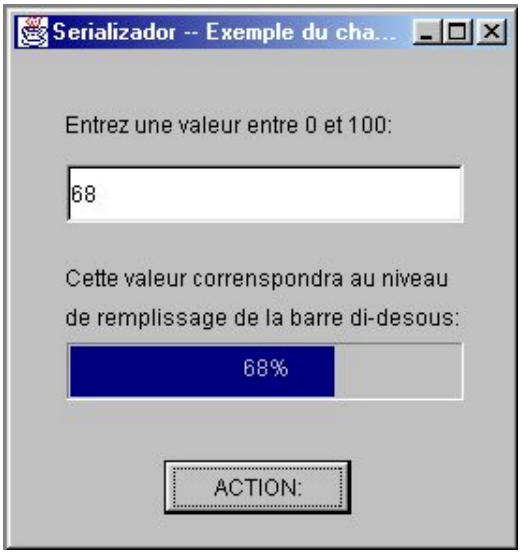
Tout ce programme pourrait être écrit avec seulement ceci : (mais notre but est de travailler sur la sérialisation)
jProgressBar1.setValue(new Integer(jTextField1.getText()).intValue());

Examinez aussi avec attention le code correspondant à la création de l'interface graphique: il manipule un composant que vous ne connaissez pas encore: la JProgressBar. Et puis, on apprend à programmer grâce à la pratique. (C'est pourquoi le chapitre sur la théorie pure de Swing est léger : il est préférable de voir Swing *en action* au travers des exemples des différents chapitres du cours.)

```
/**
* Cette classe contient la méthode main(), elle sert à charger la classe
* qui définit l'interface graphique. Elle est automatiquement générée par
* JBuilder et n'a aucun intêret propre.
*/


package serializador;

import javax.swing.UIManager;

public class RUN {
  boolean packFrame = false;

  //Construire l'application
  public RUN() {
    INTERFACE frame = new INTERFACE();
    //Valider les cadres ayant des tailles prédéfinies
    //Compacter les cadres ayant des infos de taille préférées - ex. depuis leur
disposition
    if (packFrame) {
      frame.pack();
    }
    else {
      frame.validate();
    }
    frame.setVisible(true);
  }

  //Méthode principale
  public static void main(String[] args) {
    try {
      UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
    }
    catch(Exception e) {
      e.printStackTrace();
    }
    new RUN();
  }
}
```

```java
package serializador;

import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import com.borland.jbcl.layout.*;

public class INTERFACE extends JFrame {
  JPanel contentPane;
  BorderLayout borderLayout1 = new BorderLayout();
  JPanel jPanel1 = new JPanel();
  XYLayout xYLayout1 = new XYLayout();
  JTextField jTextField1 = new JTextField();
  JLabel jLabel1 = new JLabel();
  JProgressBar jProgressBar1 = new JProgressBar();
  JLabel jLabel2 = new JLabel();
  JLabel jLabel3 = new JLabel();
  JButton jButton1 = new JButton();

  //Construire le cadre
  // constructeur de la classe
  public INTERFACE() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }


  /**
   * Cette méthode permet de sérialiser, d'enregistrer, dans le fichier
   * Serializador.dat l'Integer qui lui est transmis.
   */
  public void serialize(Integer i) {
    try {
     FileOutputStream fos = new FileOutputStream("Serializador.dat");
     ObjectOutputStream oos = new ObjectOutputStream(fos);
     oos.writeObject(i);
     oos.close();
    }catch(IOException ioe) {jTextField1.setText("Erreur" + ioe);}
  }

  /**
   * Cette méthode permet de lire dans le fichier Serializador.dat un Integer,
   * en l'occurence celui écrit par serialize(Integer i) et de le renvoyer grâce
   * à return sous forme int (type de base).
   */
  public int deserialize() {
    int i = 0;
    try {
     FileInputStream fis = new FileInputStream("Serializador.dat");
     ObjectInputStream ois = new ObjectInputStream(fis);
     Integer integ = (Integer)ois.readObject();
     i = integ.intValue();
     ois.close();
    }catch(Exception err) {jTextField1.setText("Erreur" + err);}
   return i;
  }

  /**
   * Cette méthode permet de prendre une chaîne de texte dans notre zone de texte,
   * de la convertir en Integer et, si elle est comprise entre 0 et 100, de la
   * transmettre comme argument à la méthode serialize(Integer i)
   */
  public void prendreValeur() {
    Integer integ = new Integer(jTextField1.getText());
    if((integ.intValue() < 100) && (integ.intValue() > 0))
    serialize(integ);
  }
```

```
  /**
   * Cette méthode permet de définir la valeur i de type int renvoyée par
   * deserialize() comme valeur de notre barre de progression
   */
  public void mettreValeur() {
    jProgressBar1.setValue(deserialize());
  }




  //Initialiser le composant
  private void jbInit() throws Exception  {
    // le premier panneau
    contentPane = (JPanel) this.getContentPane();
    // son layout
    contentPane.setLayout(borderLayout1);
    // la taille de la fenêtre(this)
    this.setSize(new Dimension(273, 290));
    // son titre
    this.setTitle("Serializador -- Exemple du chapitre sur le Sérialisation de
JGFLsoft");
    /* le layout du panneua sur lequel on va travailler.
     * ATTENTION: ce layout n'est pas disponible avec Java2, il vient du
     * package con.borland.jbcl.layout.XYLayout
     * si vous avez installé JBuilder Foundation (gratuit et le meilleur IDE)
     * vous l'avez par défaut !
    */
    jPanel1.setLayout(xYLayout1);
    jLabel1.setText("Entrez une valeur entre 0 et 100:");
    jLabel2.setText("Cette valeur correnspondra au niveau");
    jLabel3.setText("de remplissage de la barre di-desous:");
    jButton1.setText("ACTION:");

    /*
    * Gestionnaire d'évènement du bouton jButton1 qui exécute les méthodes
    * prendreValeur() et mettreValeur() lors d'une action sur ce bouton
    */
    jButton1.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(ActionEvent e) {
     prendreValeur();
     mettreValeur();
      }
    });

    // l'indicateur n% est affiché
    jProgressBar1.setStringPainted(true);
    contentPane.add(jPanel1, BorderLayout.CENTER);
    // ajout des conposants aux coordonnées:
    // X coin supérieur gauche
    // Y coin supérieur gauche
    // largeur
    // hauteur
    jPanel1.add(jTextField1, new XYConstraints(28, 61, 212, 31));
    jPanel1.add(jLabel1, new XYConstraints(27, 30, 214, 19));
    jPanel1.add(jLabel2, new XYConstraints(27, 112, 214, 19));
    jPanel1.add(jProgressBar1, new XYConstraints(28, 156, 212, 31));
    jPanel1.add(jLabel3, new XYConstraints(27, 133, 214, 19));
    jPanel1.add(jButton1, new XYConstraints(80, 219, 100, 29));
  }

  //Remplacé, ainsi nous pouvons sortir quand la fenêtre est fermée
  protected void processWindowEvent(WindowEvent e) {
    super.processWindowEvent(e);
    if (e.getID() == WindowEvent.WINDOW_CLOSING) {
      System.exit(0);
    }
  }

}
```


Serializador -- Exemple du cha...
Entrez une valeur entre 0 et 100:
68
Cette valeur correnspondra au niveau
de remplissage de la barre di-desous:
68%
ACTION: