La sérialisation binaire en Java

par Yann D'ISANTO (Autres articles)

Date de publication : 06/09/2006

Dernière mise à jour : 06/09/2006

Ce tutoriel a pour but de présenter la sérialisation binaire en Java. Après une initiation aux bases de ce concept, il poursuit sur les fonctionnalités plus complexes qui vous permettront une maîtrise totale de la sérialisation.

I - Avant-Propos

Tout d'abord, qu'est ce que la sérialisation ?

La sérialisation consiste à écrire des données présentes en mémoire vers un flux de données binaires, ce procédé va donc nous permettre de rendre nos objets persistants.

Java a introduit la sérialisation dans le JDK 1.1 et fournit des outils nous permettant de sérialiser nos objets de manière transparente et indépendante du système d'exploitation.

II - Les bases de la sérialisation

La sérialisation en Java s'appuie sur les flux (voir **ce tutoriel sur le package java.io**), c'est pourquoi une certaine connaissance de ceux-ci est souhaitable pour aborder ce tutoriel en toute sérénité.

L'API Java nous fournit les outils nécessaires à la sérialisation suivants :

- l'interface *Serializable*
- la classe *ObjectOutputStream*
- la classe *ObjectInputStream*

L'interface *Serializable* permet d'identifier les classes sérialisables, les classes *ObjectOutputStream* et *ObjectInputStream* implémentent, quand à elle, les mécanismes de sérialisation et de désérialisation afin de nous les abstraire.

Nous avons aussi à notre disposition l'interface *Externalizable* qui nous permet d'implémenter notre propre mécanisme de sérialisation.

II-A - L'interface Serializable

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. Tous les attributs de l'objet sont sérialisés mais à certaines conditions.

Pour être sérialisé, un 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 (nous y reviendrons plus tard)
- ne pas être hérité d'une classe mère sauf si celle-ci est elle-même sérialisable

II-B - Le serialVersionUID

Le serialVersionUID est un "numéro de version", associé à toute classe implémentant l'interface *Serializable*, qui permet de s'assurer, lors de la désérialisation, que les versions des classes Java soient concordantes. Si le test échoue, une *InvalidClassException* est levée.

Une classe sérialisable peut déclarer explicitement son serialVersionUID en déclarant un attribut nommé "serialVersionUID" qui doit être **static**, **final** et de type **long** :

déclaration du serialVersionUID

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

Si une classe sérialisable ne déclare pas explicitement un serialVersionUID, alors le mécanisme de sérialisation Java en calcule un par défaut en se basant sur divers aspects de la classe (ce procédé est régi par la spécification Java(TM) Object serialization Specification). Il est cependant fortement recommandé de déclarer explicitement le serialVersionUID. En effet, le calcul de la valeur par défaut repose sur des paramètres qui peuvent varier selon l'implémentation du compilateur ce qui peut provoquer des *InvalidClassException* inattendues lors de la désérialisation.

Notons que depuis Java 5 un **warning** signale le fait qu'une classe implémentant l'interface *Serializable* ne définit pas explicitement le serialVersionUID (**plus de détails dans la FAQ**). Il est également recommandé de déclarer le serialVersionUID comme **private**.

II-C - L'interface ObjectOutput

L'interface *ObjectOutput* étend l'interface *DataOutput* pour y ajouter l'écriture des objets. En effet, l'interface *DataOutput* implémente les méthodes pour l'écriture des types primitifs, *ObjectOutput* implémente en plus la possibilité d'écrire les objets et les tableaux.

Cette interface est implémentée par la classe *ObjectOutputStream* que nous allons voir maintenant.

II-D - La classe ObjectOutputStream

La classe *ObjectOutputStream* représente "un flux objet" qui permet de sérialiser un objet grâce à la méthode writeObject().

Un petit exemple valant mieux qu' un grands discours, passons directement à la pratique.

Tout d'abord, définissons une classe *Personne* sérialisable.

Personne.java

```
import java.io.Serializable;

public class Personne implements Serializable {
 static private final long serialVersionUID = 6L;
 private String nom;
 private String prenom;
 private Integer age;

 public Personne(String nom, String prenom, Integer age) {
  this.nom = nom;
  this.prenom = prenom;
  this.age = age;
 }

 public String toString() {
  return nom + " " + nom + " " + age + " ans";
 }
}
```

Procédons maintenant à la sérialisation d'un objet *Personne*, pour cela créons la classe *SerializationMain* suivante :

SerializationMain.java

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

public class SerializationMain {

 static public void main(String ...args) {
  try {
   // création d'une personne
   Personne p = new Personne("Dupont", "Jean", 36);
   System.out.println("creation de : " + p);

   // ouverture d'un flux de sortie vers le fichier "personne.serial"
   FileOutputStream fos = new FileOutputStream("personne.serial");

   // création d'un "flux objet" avec le flux fichier
   ObjectOutputStream oos= new ObjectOutputStream(fos);
   try {
    // sérialisation : écriture de l'objet dans le flux de sortie
    oos.writeObject(p);
    // on vide le tampon
    oos.flush();
    System.out.println(p + " a ete serialise");
   } finally {
    //fermeture des flux
    try {
     oos.close();
    } finally {
     fos.close();
    }
   }
  } catch(IOException ioe) {
   ioe.printStackTrace();
  }
 }
}
```

L'exécution de cette classe donne la sortie suivante :

```
creation de : Jean Dupont 36 ans
Jean Dupont 36 ans a ete serialise
```

Comme vous pouvez le voir, la sérialisation d'un objet est vraiment très simple, et la désérialisation l'est tout autant.

II-E - L'interface ObjectInput

L'interface *ObjectInput* étend l'interface *DataInput* pour y ajouter la lecture des objets. En effet, l'interface *DataInput* implémente les méthodes pour la lecture des types primitifs, *ObjectInput* implémente en plus la possibilité de lire les objets et les tableaux.

Cette interface est implémentée par la classe *ObjectInputStream* que nous allons voir maintenant.

II-F - La classe ObjectInputStream

Maintenant que nous avons vu comment sérialiser un objet nous allons voir comment le désérialiser. Il s'agit évidemment de l'opération inverse, tout aussi simple à mettre en oeuvre. Pour cela nous allons utiliser la méthode readObject() de la classe *ObjectInputStream*.

Désérialisons maintenant la personne précédemment sérialisée.

DeserializationMain.java

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

public class DeserializationMain {

 static public void main(String ...args) {
  Personne p = null;
  try {
   // ouverture d'un flux d'entrée depuis le fichier "personne.serial"
   FileInputStream fis = new FileInputStream("personne.serial");
   // création d'un "flux objet" avec le flux fichier
   ObjectInputStream ois= new ObjectInputStream(fis);
   try {
    // désérialisation : lecture de l'objet depuis le flux d'entrée
    p = (Personne) ois.readObject();
   } finally {
    // on ferme les flux
    try {
     ois.close();
    } finally {
     fis.close();
    }
   }
  } catch(IOException ioe) {
   ioe.printStackTrace();
  } catch(ClassNotFoundException cnfe) {
   cnfe.printStackTrace();
  }
  if(p != null) {
   System.out.println(p + " a ete deserialise");
  }
 }
}
```

L'exécution de la classe *DeserializationMain* donne la sortie suivante :

```
Jean Dupont 36 ans a ete deserialise
```

Nous récupérons donc bien notre objet *Personne* précédement sérialisé.

II-G - Le mot clé transient

Le mot clé **transient** permet d'interdire la sérialisation d'un attribut d'une classe. Il est en général utilisé pour les données "sensibles" telles que les mots de passe ou tout simplement pour les attributs n'ayant pas besoin d'être sérialisé.

Prenons le cas de notre classe *Personne* et rajoutons lui un attribut password en prenant la précaution de le déclarer avec le mot clé **transient** :

Personne.java

```
import java.io.Serializable;

public class Personne implements Serializable {
 static private final long serialVersionUID = 51L
 private String nom;
 private String prenom;
 private Integer age;

 transient private String password;
 ...
}
```

Vous pouvez reprendre les codes de sérialisation/désérialisation précédents pour vérifier que l'attribut password n'est pas sérialisé.

II-H - La sérialisation et l'héritage

Voyons comment se comporte la sérialisation face à l'héritage.

Premier cas de figure, très simple, lorsqu'une classe hérite d'une classe sérialisable, elle se comporte comme si elle avait implémenté elle-même l'interface *Serializable*.

Les attributs de la classe mère sont sérialisés selon l'implémentation de celle-ci.

Deuxième cas de figure, une classe implémente l'interface *Serializable* et hérite d'une classe non sérialisable.

Il y a ici deux points fondamentaux à savoir :

- les attributs hérités ne sont pas sérialisés.
- il est nécessaire que la classe étendue possède un constructeur par défaut accessible ; dans le cas contraire, une *InvalidClassException* est levée à l'exécution.

Illustration :

soit une classe *A* non sérialisable et une classe *B* héritant de *A* et implémentant l'interface *Serializable*.

A.java

```
public class A {
```

A.java

```
protected String string;

// le constructeur par défaut est nécessaire
public A() {
 this("<default string>");
}
public A(String string) {
 this.string = string;
}

public String toString() {
 return this.string;
}
}
```

B.java

```
import java.io.Serializable;

public class B extends A implements Serializable {
 static private final long serialVersionUID = 7L;

 private int integer;

 public B(String string, int integer) {
  super(string);
  this.integer = integer;
 }
 public String toString() {
  return super.toString() + " : " + this.integer;
 }
}
```

Exécutons maintenant le code suivant :

Main.java

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

public class Main {

 static public void main(String ...args) {
  try {
   B b = new B("B1", 1);
   System.out.println(b);
   FileOutputStream fos = new FileOutputStream("b.serial");
   ObjectOutputStream oos = new ObjectOutputStream(fos);
   try {
    oos.writeObject(b);
    oos.flush();
   } finally {
    oos.close();
    fos.close();
   }
   b = new B("B2", 2);
   System.out.println(b);
   FileInputStream fis = new FileInputStream("b.serial");
```

Main.java
```
      ObjectInputStream ois = new ObjectInputStream(fis);
      try {
       b = (B) ois.readObject();
      } finally {
       try {
        ois.close();
       } finally {
        fis.close();
       }
      }
      System.out.println(b);
     } catch(ClassNotFoundException cnfe) {
      cnfe.printStackTrace();
     } catch(IOException ioe) {
      ioe.printStackTrace();
     }
    }
   }
```

On obtient alors la sortie suivante :

```
B1 : 1
B2 : 2
<default string> : 1
```

On voit bien que l'attribut string hérité de la classe *A* n'est pas restauré. Réessayez maintenant en commentant le constructeur par défaut de la classe *A*, et voyez l'*InvalidClassException* qui est levée.

III - Personnalisation de la sérialisation

Dans la partie précédente nous avons vu comment procéder à la sérialisation/désérialisation d'un objet en utilisant le mécanisme par défaut fourni par Java.

Cependant, celui-ci peut se révéler limité ou inadéquat dans certain cas. Heureusement, pour palier à cet inconvénient, Java nous offre la possibilté de "personnaliser" la sérialisation et même d'implémenter notre propre mécanisme.

III-A - Les méthodes writeObject() et readObject()

Il se peut que vous ayez besoin de faire un traitement particulier lors de la sérialisation et/ou de la désérialisation comme l'écriture de données supplémentaires (par exemple un attribut hérité d'une classe non sérialisable).

Pour cela, une classe implémentant l'interface *Serializable* peut implémenter les méthodes writeObject() et readObject() où nous pouvons :

- définir notre propre mécanisme de sérialisation mais ceci uniquement pour les attributs propres à la classe (les attributs sérialisables hérités d'une classe sérialisable reste gérés par le mécanisme Java).
- sérialiser des attributs que le mécanisme par défaut ne sérialiserait pas (un attribut static par exemple).

Voici la signature desdites méthodes :

```
private void writeObject(java.io.ObjectOutputStream out) throws IOException;
private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException;
```

Notons que l'écriture suivante applique le mécanisme de sérialisation par défaut :

```
private void writeObject(java.io.ObjectOutputStream out) throws IOException {
 // appel des mécanismes de sérialisation par défaut
 out.defaultWriteObject();
}
private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException {
 // appel des mécanismes de désérialisation par défaut
 in.defaultReadObject();
}
```

Exemple : soit une classe *C* implémentant *Serializable* et une classe *D* qui en hérite. La classe *D* possède les attributs *serialisationCount* et *deserialisationCount* que nous voulons incrémenter lors des opérations de sérialisation/désérialisation.

C.java

```
import java.io.Serializable;

public class C implements Serializable {

 static private final long serialVersionUID = 8L;
 protected String string;
```

C.java

```
    public C() {
     this("");
    }
    public C(String string) {
     this.string = string;
    }
    public String toString() {
     return this.string;
    }
   }
```

D.java

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

   public class D extends C {

    static private final long serialVersionUID = 9L;
    private int integer;

    // on déclare les attributs serialisationCount et deserialisationCount transient
    // afin qu'ils ne soient pas gérés par le mécanisme de sérialisation par défaut.
    private transient int serialisationCount = 0;
    private transient int deserialisationCount = 0;

    public D(String string, int integer) {
     super(string);
     this.integer = integer;
    }
    public String toString() {
     return super.toString() + " a ete serialise " + serialisationCount +
       " fois et deserialise " + deserialisationCount + " fois";
    }

    private void writeObject(ObjectOutputStream out) throws IOException {
     // appel des mécanismes de sérialisation par défaut
     out.defaultWriteObject();

     // on incrémente notre compteur de sérialisation
     serialisationCount ++;

     // on sérialise les attributs normalement non sérialisés
     out.writeInt(serialisationCount);
     out.writeInt(deserialisationCount);
    }

    private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
     // appel des mécanismes de désérialisation par défaut
     in.defaultReadObject();

     // on désérialise les attributs normalement non désérialisés
     serialisationCount = in.readInt();
     deserialisationCount = in.readInt();

     // on incrémente notre compteur de désérialisation
     deserialisationCount ++;
    }
   }
```

Ainsi à chaque sérialisation/désérialisation le compteur associé est incrémenté. Vérifions avec le code suivant :

Main.java

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

public class Main {

 static public void main(String ...args) {
  try {
   D d = new D("D1", 1);
   System.out.println(d);
   FileOutputStream fos = new FileOutputStream("d.serial");
   ObjectOutputStream oos = new ObjectOutputStream(fos);
   try {
    oos.writeObject(d);
    oos.flush();
   } finally {
    try {
     oos.close();
    } finally {
     fos.close();
    }
   }

   System.out.println(d);
   FileInputStream fis = new FileInputStream("d.serial");
   ObjectInputStream ois = new ObjectInputStream(fis);
   try {
    d = (D) ois.readObject();
   } finally {
    try {
     ois.close();
    } finally {
     fis.close();
    }
   }
   System.out.println(d);
  } catch(ClassNotFoundException cnfe) {
   cnfe.printStackTrace();
  } catch(IOException ioe) {
   ioe.printStackTrace();
  }
 }
}
```

L'exécution de code donne la sortie suivante :

```
D1 a ete serialise 0 fois et deserialise 0 fois
D1 a ete serialise 1 fois et deserialise 0 fois
D1 a ete serialise 1 fois et deserialise 1 fois
```

III-B - Les méthodes writeReplace() et readResolve()

Il se peut que lors de la sérialisation et/ou la désérialisation d'un objet, vous vouliez utiliser une autre instance que celle de l'objet en question (pour assurer l'unicité d'un singleton par exemple).

Pour cela la classe sérialisable doit implémenter les méthodes writeReplace() et readResolve() aux signatures suivantes :

```
Object writeReplace() throws ObjectStreamException;
Object readResolve() throws ObjectStreamException;
```

Prenons l'exemple d'une classe *Singleton* implémentant *Serializable*:

Singleton.java

```
import java.io.Serializable;
import java.io.ObjectStreamException;

public class Singleton implements Serializable {

 static private final long serialVersionUID = 33L;
 static private Singleton singleton = null;
 private int data = 0;

 private Singleton() {
 }

 public int getData() {
  return this.data;
 }
 public void setData(int data) {
  this.data = data;
 }

 static public synchronized Singleton getSingleton() {
  if(singleton == null) {
   singleton = new Singleton();
  }
  return singleton;
 }
}
```

A première vue il ne semble y avoir aucun problème. Cependant il y a un risque de perdre l'unicité de notre *Singleton*.

Voyons cela avec le code suivant qui ne fait que sérialiser puis désérialiser notre *Singleton* :

Main.java

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


public class Main {

 static public void main(String ...args) {
  try {
   Singleton s = Singleton.getSingleton();
   s.setData(1);
   System.out.println(s + " : " + s.getData());
```

Main.java
```
      FileOutputStream fos = new FileOutputStream("singleton.serial");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      try {
       oos.writeObject(s);
       oos.flush();
      } finally {
       try {
        oos.close();
       } finally {
        fos.close();
       }
      }
      s.getSingleton();
      s.setData(2);
      System.out.println(s + " : " + s.getData());
      FileInputStream fis = new FileInputStream("singleton.serial");
      ObjectInputStream ois = new ObjectInputStream(fis);
      try {
       s = (Singleton) ois.readObject();
      System.out.println(s + " : " + s.getData());
      } finally {
       ois.close();
       fis.close();
      }
     } catch(ClassNotFoundException cnfe) {
      cnfe.printStackTrace();
     } catch(IOException ioe) {
      ioe.printStackTrace();
     }
    }
   }
```

L'exécution de ce code donne la sortie suivante :

```
Singleton@10b62c9 : 1
Singleton@10b62c9 : 2
Singleton@9cab16 : 1
```

Chaque ligne affiche la référence de l'instance du *Singleton* suivie de sa donnée. Nous voyons donc que même si la donnée est bien restaurée, l'instance de notre singleton n'est plus la même. En effet, le mécanisme de désérialisation renvoie une nouvelle instance de la classe désérialisée, et ceci nous pose donc un gros problème au niveau de notre *Singleton*, qui finalement n'en est plus un.

Pour résoudre ce problème il nous suffit d'implémenter la méthode readResolve() (nous n'avons pas besoin d'implémenter la méthode writeReplace() car notre problème se situe uniquement au niveau de la désérialisation mais son implémentation est identique). Voici la méthode readResolve() à implémenter dans la classe *Singleton* :

```
protected Object readResolve() throws ObjectStreamException {
 // récupération des données désérialisées
 int d = getData();

 // récuperation de l'instance static de Singleton
 Singleton s = getSingleton();

 // affectation des données désérialisées à l'instance static.
 s.setData(d) ;
```

```
    // on renvoie l'instance static
    return s;
  }
```

Réexécutez maintenant le code précédent, vous obtenez alors la sortie suivante :

```
Singleton@10b62c9 : 1
Singleton@10b62c9 : 2
Singleton@10b62c9 : 1
```

Nous pouvons voir que la troisième ligne est identique à la première et donc que l'instance du Singleton reste bien unique.

III-C - L'interface Externalizable

Comme nous l'avons vu, Java nous abstrait le mécanisme de sérialisation afin de nous simplifier son utilisation. Nous avons aussi vu qu'il était possible de personnaliser un peu ce mécanisme grâce aux méthodes writeObject(), readObject(), writeReplace() et readResolve().

Cependant, Java nous offre la possibilité d'implémenter notre propre mécanisme de sérialisation grâce à l'interface *Externalizable* (qui étend *Serializable*). Aux travers de ces méthodes writeExternal() et readExternal(), cette interface nous permet un contrôle total du processus de sérialisation.

La première chose à savoir est qu'un objet implémentant l'interface *Externalizeable* doit posséder un constructeur par défaut **public**. En effet, lors de la désérialisation d'un objet implémentant l'interface *Externalizable*, tous les comportements de construction par défaut sont appliqués (constructeur par défaut, initialisation lors de la déclaration des attributs). Il faut aussi savoir que si une classe implémentant *Externalizable* ne possède pas de constructeur par défaut, le compilateur ne signalera aucune erreur, de même la sérialisation de l'objet se passera sans problème, cependant, une *InvalidClassException* sera levée lors de la désérialisation.

Illustration de l'utilisation de l'interface *Externalizable*. Soit :

- une classe *E* implémentant *Serializable* et ayant un attribut de type *String*.
- une classe *F* héritant de *E*, implémentant l'interface *Externalizable* et possédant un attribut de type int.

Nous allons personnaliser la sérialisation de la classe *F* en stockant le carré de l'entier et en stockant la chaîne de caractère sous la forme d'un tableau de byte auquel nous aurons appliqué un cryptage de césar fonction de l'entier.

E.java

```
import java.io.Serializable;

public class E implements Serializable {
  static private final long serialVersionUID = 12L;
  protected String string;
```

E.java

```
public E() {
 this("");
}
public E(String string) {
 this.string = string;
}

public String toString() {
 return this.string;
}
}
```

F.java

```
import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

public class F extends E implements Externalizable {

 static private final long serialVersionUID = 13L;
 private int integer;

 public F() {
  this("", 0);
 }
 public F(String string, int integer) {
  super(string);
  this. integer = integer;
 }

 /**
 *   Sérialisation des données.
 */
 public void writeExternal(ObjectOutput out) throws IOException {
  // récupération du tableau de byte représentant string
  byte[] b = this.string.getBytes("UTF-8");

  // cryptage de césar avec integer
  for(int i = 0; i < b.length ; i ++) {
   b[i] += this. integer;
  }

  // écriture du carré de integer
  out.writeLong(this.integer * this.integer);

  // écriture du tableau de byte
  out.writeObject(b);
 }

 /**
 *   Désérialisation des données.
 */
 public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
  // lecture du carré de integer
  this.integer = (int) Math.sqrt(in.readLong());

  // lecture du tableau de byte
  byte[] b = (byte[]) in.readObject();
  for(int i = 0; i < b.length; i ++) {
   b[i] -= this. integer;
  }
  this.string = new String(b, "UTF-8");
 }
```

F.java

```
    public String toString() {
     return super.toString() + " : " + this.integer;
    }
   }
```

Code à exécuter

Main.java

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

   public class Main {

    static public void main(String ...args) {
     F f = new F("Chaine de caractere", 3);
     System.out.println(f);
     try {
      FileOutputStream fos = new FileOutputStream("f.external");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      try {
       oos.writeObject(f);
       oos.flush();
      } finally {
       try {
        oos.close();
       } finally {
        fos.close();
       }
      }
      f = null;
      System.out.println(f);

      FileInputStream fis = new FileInputStream("f.external");
      ObjectInputStream ois = new ObjectInputStream(fis);
      try {
       f = (F) ois.readObject();
      } finally {
       try {
        ois.close();
       } finally {
        fis.close();
       }
      }
      System.out.println(f);
     } catch(IOException ioe) {
      ioe.printStackTrace();
     } catch(ClassNotFoundException cnfe) {
      cnfe.printStackTrace();
     }
    }
   }
```

La sortie générée est :

```
   Chaine de caractere : 3
```

```
null
Chaine de caractere : 3
```

L'interface *Externalizable* nous laisse donc une liberté totale quand à l'implémentation de notre propre mécanisme de sérialisation.

III-D - La sérialisation et les servlets (J2EE)

Cette partie s'apparente plus à une remarque, car il s'agit en fait de porter à votre connaissance un comportement spécifique de Java face à la sérialisation. En effet, si vous utilisez J2EE (Java EE) alors vous devez savoir que la **Java Servlet Specification 2.4** (page 61) nous prévient que lors de la sauvegarde ou la migration de sessions (objet *HttpSession*), le Container est libre de sérialiser en utilisant son propre mécanisme.

De ce fait, l'appel aux méthodes readObject() et writeObject() n'est pas garanti.

IV - Liens

- La Javadoc
  - Le Package java.io
  - L'interface Serializable
  - L'interface ObjectOutput
  - La classe ObjectOutputStream
  - L'interface ObjectInput
  - La classe ObjectInputStream
  - L'interface Externalizable
- Object Serialization (guide de la sérialisation sur java.sun.com)
- Tutoriel sur le package java.io
- Les FAQs Java
- Les cours et tutoriels Java
- Les forums Java
- Java Servlet Specification 2.4

V - Remerciements

Je tiens à remercier tous ceux qui ont contribué à la rédaction de ce tutoriel, et tout particulièrement **adiGuba** et **vbrabant** pour leur aide et leurs conseils avisés.

VI - Téléchargements

Article au format pdf : **FTP (lien principal)**, **HTTP (lien de secours)**

Article au format html : **FTP (lien principal)**, **HTTP (lien de secours)**

Code source des exemples de l'article : **FTP (lien principal)**, **HTTP (lien de secours)**