

La sérialisation XML en Java



par Yann D'ISANTO ([Autres articles](#))

Date de publication : 21/12/2007

Dernière mise à jour : 21/12/2007

Ce tutoriel a pour but de présenter la sérialisation XML en Java. Il commence par les bases et continue sur les fonctionnalités plus complexes qui vous permettront une maîtrise totale de ce mécanisme.

- I - Avant-propos
- II - Concepts de base
 - II-A - XMLEncoder
 - II-B - XMLDecoder
 - II-C - Limitations
- III - Concepts avancés
 - III-A - Empêcher la sérialisation d'un attribut
 - III-B - Sérialiser des attributs sans accesseur/modifieur
 - III-C - Sérialiser une classe sans constructeur par défaut
 - III-C-1 - Les arguments du constructeur sont des attributs de la classe
 - III-C-2 - Les arguments du constructeur ne sont pas des attributs de la classe.
 - III-C-3 - Pas de constructeur mais une méthode d'instanciation
 - III-C-4 - Remarque
 - III-D - Traiter les exceptions
- IV - Conclusion
- V - Liens
- VI - Remerciements
- VII - Téléchargements

I - Avant-propos

La "sérialisation XML" est un mécanisme de persistance que nous offre Java depuis sa version 1.4 via les classes *XMLEncoder* et *XMLDecoder*. Cependant, cette appellation est un abus de langage car ce mécanisme n'est pas une vraie sérialisation même si leur principe et leur utilisation possèdent plusieurs similitudes.

II - Concepts de base

La sérialisation XML possède plusieurs différences avec la **sérialisation binaire**. La plus évidente est que les données générées le sont dans un format textuel (XML) et non binaire. Ensuite la sérialisation XML a été spécialement conçue pour être utilisée avec les JavaBeans ce qui apporte certains avantages mais aussi certaines limitations.

Avantages :

- La portabilité : il n'y a aucune dépendance avec l'implémentation propre des classes qui peuvent donc être échangées entre différentes VMs de différents fournisseurs.
- Les documents XML générés sont compacts grâce à l'utilisation d'un algorithme d'élimination des redondances (par exemple les attributs ayant leur valeur par défaut ne sont pas écrits).
- La tolérance aux fautes : si une erreur non structurelle est détectée, la partie affectée n'est pas prise en compte.
- Etant un document XML, le résultat est lisible par l'humain et exploitable par d'autres programmes.

Inconvénients :

- La classe à sérialiser doit posséder un constructeur par défaut (sans paramètre).
- Les attributs de la classe à sérialiser doivent être accessibles via des accesseurs/modifieurs.
- Le résultat de la sérialisation XML est plus gros que celui de la sérialisation binaire.

Nous verrons qu'il est cependant possible de contourner la plupart de ces limitations.

La sérialisation XML repose principalement sur les classes *XMLEncoder* et *XMLDecoder* qui permettent de procéder à la sérialisation et la désérialisation d'objets.

II-A - XMLEncoder

XMLEncoder est la classe qui permet de sérialiser un objet vers un document XML. Son utilisation est très simple et similaire à celle de l'*ObjectOutputStream* de la sérialisation binaire.

Nous allons créer une classe *XMLTools* contenant une méthode statique permettant de sérialiser un objet dans un fichier.

```
XMLTools.java
import java.beans.PersistenceDelegate;
import java.beans.XMLEncoder;
import java.io.FileOutputStream;
```

XMLTools.java

```
import java.io.FileNotFoundException;
import java.io.IOException;

public final class XMLTools {

    private XMLTools() {}

    /**
     * Serialisation d'un objet dans un fichier
     * @param object objet a serialiser
     * @param filename chemin du fichier
     */
    public static void encodeToFile(Object object, String fileName) throws FileNotFoundException,
    IOException {
        // ouverture de l'encodeur vers le fichier
        XMLEncoder encoder = new XMLEncoder(new FileOutputStream(fileName));
        try {
            // serialisation de l'objet
            encoder.writeObject(object);
            encoder.flush();
        } finally {
            // fermeture de l'encodeur
            encoder.close();
        }
    }
}
```

Testons notre code en sérialisant un objet *User*

User.java

```
public class User {

    private String login;
    private String password;

    public User() {
        this("anonymous", "");
    }

    public User(String login, String password) {
        this.login = login;
        this.password = password;
    }

    public String getLogin() {
        return login;
    }

    public void setLogin(String login) {
        this.login = login;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String toString() {
        return login;
    }
}
```

User.java

```
public static void main(String[] args) {
    try {
        User user = new User("admin", "azerty");
        XMLTools.encodeToFile(user, "user.xml");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Nous obtenons alors le fichier "user.xml" suivant :

user.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_03" class="java.beans.XMLDecoder">
  <object class="User">
    <void property="login">
      <string>admin</string>
    </void>
    <void property="password">
      <string>azerty</string>
    </void>
  </object>
</java>
```

II-B - XMLDecoder

L'utilisation de la classe *XMLEncoder* est très simple, et celle de la classe *XMLDecoder* l'est tout autant. Comme vous pouvez vous en douter, la classe *XMLDecoder* sert à désérialiser un objet sérialisé avec *XMLEncoder*. Rajoutons une méthode à notre classe utilitaire *XMLTools* afin de pouvoir charger un objet depuis un fichier.

XMLTools.java

```
/**
 * Désérialisation d'un objet depuis un fichier
 * @param filename chemin du fichier
 */
public static Object decodeFromFile(String fileName) throws FileNotFoundException, IOException {
    Object object = null;
    // ouverture de decodeur
    XMLDecoder decoder = new XMLDecoder(new FileInputStream(fileName));
    try {
        // désérialisation de l'objet
        object = decoder.readObject();
    } finally {
        // fermeture du decodeur
        decoder.close();
    }
    return object;
}
```

Rajoutons Maintenant la désérialisation de l'objet sérialisé dans l'exemple précédent :

```
public static void main(String[] args) {
    try {
        User user = new User("admin", "azerty");
        XMLTools.encodeToFile(user, "user.xml");
        System.out.println(user);
    }
}
```

```
user = new User("newAdmin", "123456");
System.out.println(user);
user = (User) XMLTools.decodeFromFile("user.xml");
System.out.println(user);
} catch(Exception e) {
    e.printStackTrace();
}
}
```

La sortie standard affiche alors :

```
admin
newAdmin
admin
```

Nous retrouvons donc bien l'objet sérialisé.

II-C - Limitations

Comme nous l'avons vu, la sérialisation XML est vraiment très simple. Cependant son implémentation nécessite que les classes à traiter soient des JavaBeans. Ceci induit plusieurs limitations.

La première est l'obligation de posséder un constructeur par défaut. Essayons tout de même de sérialiser une classe n'en ayant pas.

Réessayez le code précédent mais en commentant le constructeur par défaut de la classe User.

La sortie standard affiche alors :

```
java.lang.InstantiationException: User
Continuing ...
java.lang.Exception: XMLEncoder: discarding statement XMLEncoder.writeObject(User);
Continuing ...
admin
newAdmin
java.lang.ArrayIndexOutOfBoundsException: 0
    at com.sun.beans.ObjectHandler.dequeueResult(ObjectHandler.java:139)
    at java.beans.XMLDecoder.readObject(XMLDecoder.java:201)
    at XMLTools.decodeFromFile(XMLTools.java:42)
    at User.main(User.java:43)
```

Les quatre premières lignes indiquent qu'il y a une erreur lors de la sérialisation, cependant vous pouvez remarquer que l'application continue à s'exécuter jusqu'à ce que la tentative de désérialisation lève une exception.

En effet, les exceptions dues à l'absence de constructeur par défaut ne sont pas propagées mais traitées directement dans la méthode *writeObject()* de la classe *XMLEncoder* (nous verrons plus loin comment faire pour traiter soi-même ces exceptions). De même, le fichier XML est malgré tout généré bien qu'il ne contienne pas la description de l'objet (d'où l'erreur lors de la désérialisation).

user.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_03" class="java.beans.XMLDecoder">
</java>
```

La deuxième limitation est que chaque attribut à sérialiser doit posséder un accesseur et un modifieur (getter/setter).

Si un attribut ne possède pas d'accesseur il ne sera pas sérialisé. Par contre s'il possède un accesseur mais pas de modifieur, alors une erreur (*NoSuchMethodException*) est générée lors de la désérialisation. Tout comme pour l'absence de constructeur par défaut, l'exception n'est pas propagée mais traitée directement (dans la méthode *readObject()* cette fois-ci).

Ce système de tolérance aux erreurs assure la sérialisation des objets valides même si une erreur est détectée sur un autre objet.

III - Concepts avancés

III-A - Empêcher la sérialisation d'un attribut

Il y a deux façons d'empêcher la sérialisation d'un attribut. La première est d'utiliser les limitations inhérentes au mécanisme de la sérialisation XML et de ne pas coder d'accessueur/modifieur pour l'attribut considéré.

Dans le cas où l'on veuille conserver les accessueur/modifieur, il faut passer par la classe *PropertyDescriptor* pour rendre l'attribut non sérialisable.

En effet, le mécanisme de sérialisation XML utilise l'introspection pour sérialiser/désérialiser les objets.

Comme son nom l'indique, la classe *PropertyDescriptor* donne la description d'une propriété d'un JavaBean. Elle fait parti de l'api d'introspection Java et s'obtient via le *BeanInfo* de la classe considérée.

Prenons l'exemple de notre classe *User* où, par soucis de sécurité, nous désirons supprimer la sérialisation de l'attribut "password".

Il faut rajouter le code suivant **avant** la sérialisation :

```
// On récupère le BeanInfo de la classe User
BeanInfo info = Introspector.getBeanInfo(User.class);

// On récupère les PropertyDescriptors de la classe User via le BeanInfo
PropertyDescriptor[] propertyDescriptors = info.getPropertyDescriptors();

for (PropertyDescriptor descriptor : propertyDescriptors) {

    // On met la propriété "transient" à vrai pour le PropertyDescriptor de l'attribut "password"
    if (descriptor.getName().equals("password")) {
        descriptor.setValue("transient", Boolean.TRUE);
    }
}
```

Ce qui donne le fichier XML suivant :

```
user.xml
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_03" class="java.beans.XMLDecoder">
  <object class="User">
    <void property="login">
      <string>admin</string>
    </void>
  </object>
</java>
```

Notez que, contrairement à la sérialisation binaire, déclarer l'attribut avec le mot clé **transient** n'empêche pas sa sérialisation XML, il est nécessaire de passer par un *PropertyDescriptor*.

III-B - Sérialiser des attributs sans accessueur/modifieur

Comme nous l'avons vu, les attributs ne sont sérialisés que s'ils possèdent des accesseur/modifieur. Cependant il est possible de passer outre cette limitation à condition d'avoir des méthodes remplaçant les accesseur/modifieur.

Continuons avec notre classe *User* à laquelle nous rajoutons un attribut privé *address* de type *String*. Pour accéder et modifier cet attribut nous décidons de créer les méthodes *locate()* et *move()* au lieu des méthodes *getAddress()* et *setAddress()*.

```
private String address;

public String locate() {
    return address;
}

public void move(String address) {
    this.address = address;
}
```

Du fait que notre classe ne possède pas les méthodes *getAddress()* et *setAddress()*, l'attribut *address* ne sera donc pas sauvegardé lors de la sérialisation XML.

Pour forcer la sérialisation de l'attribut *address*, il est de nouveau nécessaire d'utiliser l'introspection mais cette fois-ci en créant soit même le *BeanInfo*.

Un *BeanInfo* est une classe donnant des informations sur la structure d'un *JavaBean*. Cette classe doit implémenter l'interface *BeanInfo* ou hériter de la classe *SimpleBeanInfo*.

Le *BeanInfo* créé doit avoir le même nom que la classe qu'il décrit suivi de "BeanInfo". Ainsi pour notre classe *User* nous devons créer une classe *UserBeanInfo*.

UserBeanInfo.java

```
import java.beans.IntrospectionException;
import java.beans.PropertyDescriptor;
import java.beans.SimpleBeanInfo;

public class UserBeanInfo extends SimpleBeanInfo {

    private PropertyDescriptor[] pDescriptors;

    public UserBeanInfo() {
        try {
            Class<User> c = User.class;

            // PropertyDescriptor de l'attribut "login"
            PropertyDescriptor loginDescriptor = new PropertyDescriptor("login", User.class);

            // PropertyDescriptor de l'attribut password
            PropertyDescriptor passwordDescriptor = new PropertyDescriptor("password", User.class);

            // PropertyDescriptor de l'attribut "address". On spécifie le nom des méthodes d'accès
            // (accesseur puis modifieur).
            PropertyDescriptor addressDescriptor = new PropertyDescriptor("address", User.class,
            "locate", "move");

            // On rend l'attribut password non sérialisable.
            passwordDescriptor.setValue("transient", Boolean.TRUE);

            pDescriptors = new PropertyDescriptor[] {
                loginDescriptor, passwordDescriptor, addressDescriptor
            }
        }
    }
}
```

UserBeanInfo.java

```
};
} catch(IntrospectionException ex) {
    ex.printStackTrace();
}
}

// On redéfinit la méthode getPropertyDescriptors()
public PropertyDescriptor[] getPropertyDescriptors() {
    return pDescriptors;
}
}
```

En procédant à la sérialisation d'un *User* :

```
public static void main(String ...args) {
    try {
        User user = new User("admin", "azerty");
        user.move("some place");
        System.out.println(user.locate());
        XMLTools.encodeToFile(user, "user.xml");
        user.move("an other place");
        System.out.println(user.locate());
        user = (User) XMLTools.decodeFromFile("user.xml");
        System.out.println(user.locate());
    } catch(Exception e) {
        e.printStackTrace();
    }
}
```

Nous obtenons le fichier XML suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_03" class="java.beans.XMLDecoder">
  <object class="User">
    <void method="move">
      <string>some place</string>
    </void>
    <void property="login">
      <string>admin</string>
    </void>
  </object>
</java>
```

III-C - Sérialiser une classe sans constructeur par défaut

Java nous offre la possibilité de modifier la façon dont le *XMLEncoder* va sérialiser un objet d'une classe donnée. En effet, la classe *XMLEncoder* contient un *Set* de *PersistenceDelegates* organisés en fonction de la classe de l'objet à encoder. Cette table interne peut être interrogée ou modifiée (par la méthode *setPersistenceDelegate()*) pour changer le processus d'encodage d'une classe particulière. La classe abstraite *PersistenceDelegate* contient une seule méthode publique *writeObject()* qui définit l'implémentation de la sérialisation.

Lors de la sérialisation XML, la classe *XMLEncoder* se contente en fait de chercher le *PersistenceDelegate* associé à la classe de l'objet à sérialiser et appelle ensuite sa méthode *writeObject()*. Si aucun *PersistenceDelegate* n'est trouvé, le *XMLEncoder* utilise alors une instance de la classe *DefaultPersistenceDelegate* qui représente le mécanisme par défaut de la sérialisation XML.

Nous allons donc modifier notre classe *XMLTools* afin d'y ajouter une méthode *encodeToFile()* permettant de sérialiser un objet en spécifiant un *PersistenceDelegate*. Voici le code de la méthode

```
public static void encodeToFile(Object object, String fileName, PersistenceDelegate
persistenceDelegate) throws IOException {
    XMLEncoder encoder = new XMLEncoder(new BufferedOutputStream(new FileOutputStream(fileName)));

    // association du PersistenceDelegate à la classe de l'objet.
    encoder.setPersistenceDelegate(object.getClass(), persistenceDelegate);

    try {
        encoder.writeObject(object);
        encoder.flush();
    } finally {
        encoder.close();
    }
}
```

Comme nous l'avons vu, il n'est pas possible de sérialiser un objet si sa classe ne possède pas de constructeur par défaut. Grâce aux *PersistenceDelegates*, nous allons pouvoir contourner cette limitation.

III-C-1 - Les arguments du constructeur sont des attributs de la classe

Pour sérialiser une telle classe, il suffit d'utiliser un *DefaultPersistenceDelegate* auquel ont aura précisé le nom des attributs considérés en utilisant un tableau de *Strings*.

Pour illustrer le propos, reprenons notre classe *User* auquel nous ne laissons pour seul constructeur que celui-ci :

```
public User(String login, String password) {
    this.login = login;
    this.password = password;
}
```

Voici comment utiliser le *DefaultPersistenceDelegate* :

```
public static void main(String ...args) {
    try {
        // on définit les attributs de la classe utilisés par le constructeur
        String[] properties = { "login", "password" };

        // construction du PersistenceDelegate
        PersistenceDelegate persistenceDelegate = new DefaultPersistenceDelegate(properties);

        User user = new User("admin", "azerty");
        XMLTools.encodeToFile(user, "user.xml", persistenceDelegate);
        System.out.println(user);
        user = new User("newAdmin", "123456");
        System.out.println(user);
        user = (User) XMLTools.decodeFromFile("user.xml");
        System.out.println(user);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Et voici le fichier *user.xml* généré :

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<java version="1.6.0_03" class="java.beans.XMLDecoder">
  <object class="User">
    <string>admin</string>
    <string>azerty</string>
  </object>
</java>
```

III-C-2 - Les arguments du constructeur ne sont pas des attributs de la classe.

Dans le cas où les arguments du constructeur ne correspondent pas à des attributs de la classe, il faut utiliser un *PersistenceDelegate* dont on doit réécrire la méthode *instanciate()*. Cette méthode renvoie un objet de type *Expression*. Une *Expression* représente une méthode ou un constructeur (dans notre cas un constructeur).

Modifions le constructeur de notre classe *User* comme suit :

```
public User(char[] login, char[] password) {
    this.login = new String(login);
    this.password = new String(password);
}
```

Et voyons maintenant l'utilisation du *PersistenceDelegate* :

```
public static void main(String ...args) {
    try {
        // construction du PersistenceDelegate
        PersistenceDelegate persistenceDelegate = new PersistenceDelegate() {

            // On redéfinie la méthode instanciate
            protected Expression instanciate(Object object, Encoder out) {
                User user = (User) object;

                //// Mise en tableau des paramètres du constructeur.
                // Dans notre cas, le constructeur attend deux paramètres de type char[].
                // Nous récupérons donc les attribut de type String de notre objet User
                // pour en extraire les char[] associés.
                Object[] params = {
                    user.getLogin().toCharArray(),
                    user.getPassword().toCharArray()
                };

                //// On retourne l'expression représentant le constructeur de la classe
                // Le premier paramètre est l'objet.
                // Le second paramètre est la classe de l'objet.
                // Le troisième paramètre est la chaîne de caractère "new" pour spécifier qu'il
                // s'agit d'un constructeur.
                // Le quatrième paramètre est un tableau d'objet représentant les paramètres à
                // passer au constructeur.
                return new Expression(object, object.getClass(), "new", params);
            }
        };

        User user = new User("admin".toCharArray(), "azerty".toCharArray());
        XMLTools.encodeToFile(user, "user.xml", persistenceDelegate);
        System.out.println(user);
        user = new User("newAdmin".toCharArray(), "123456".toCharArray());
        System.out.println(user);
        user = (User) XMLTools.decodeFromFile("user.xml");
        System.out.println(user);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

```
}
```

Voici le fichier user.xml généré :

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_03" class="java.beans.XMLDecoder">
  <object class="User">
    <array class="char" length="5">
      <void index="0">
        <char>a</char>
      </void>
      <void index="1">
        <char>d</char>
      </void>
      <void index="2">
        <char>m</char>
      </void>
      <void index="3">
        <char>i</char>
      </void>
      <void index="4">
        <char>n</char>
      </void>
    </array>
    <array class="char" length="6">
      <void index="0">
        <char>a</char>
      </void>
      <void index="1">
        <char>z</char>
      </void>
      <void index="2">
        <char>e</char>
      </void>
      <void index="3">
        <char>r</char>
      </void>
      <void index="4">
        <char>t</char>
      </void>
      <void index="5">
        <char>y</char>
      </void>
    </array>
  </object>
</java>
```

III-C-3 - Pas de constructeur mais une méthode d'instanciation

Un dernier cas peu se présenter à vous, il s'agit d'une classe qui ne possède pas de constructeur public et dont une instance ne peut être obtenue que par l'appel à une méthode d'instanciation.

La solution ressemble beaucoup à celle du cas précédent à la différence que l'on appelle ladite méthode en lieu et place du constructeur. Modifions la classe User en rendant son constructeur privé et en ajoutant une méthode d'instanciation statique `getUser()` :

```
private User(String login, String password) {
    this.login = login;
    this.password = password;
}
```

```
public static User getUser(String login, String password) {  
    return new User(login, password);  
}
```

Voici le code de sérialisation :

```
public static void main(String ...args) {  
    try {  
        // construction du PersistenceDelegate  
        PersistenceDelegate persistenceDelegate = new PersistenceDelegate() {  
  
            // On redéfinie la méthode instantiate  
            protected Expression instantiate(Object object, Encoder out) {  
                User user = (User) object;  
  
                //// Mise en tableau des paramètres du constructeur.  
                // Dans notre cas, le constructeur attend deux paramètres de type char[].  
                // Nous récupérons donc les attribut de type String de notre objet User  
                // pour en extraire les char[] associés.  
                Object[] params = {  
                    user.getLogin(),  
                    user.getPassword()  
                };  
  
                //// On retourne l'expression représentant le constructeur de la classe  
                // Le premier paramètre est l'objet.  
                // Le second paramètre est la classe de l'objet.  
                // Le troisième paramètre est le nom de la méthode.  
                // Le quatrième paramètre est un tableau d'objet représentant les paramètres à  
                // passer au constructeur.  
                return new Expression(object, object.getClass(), "getUser", params);  
            }  
        };  
  
        User user = User.getUser("admin", "azerty");  
        XMLTools.encodeToFile(user, "user.xml", persistenceDelegate);  
        System.out.println(user);  
        user = User.getUser("newAdmin", "123456");  
        System.out.println(user);  
        user = (User) XMLTools.decodeFromFile("user.xml");  
        System.out.println(user);  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

Voici le fichier user.xml généré :

```
<?xml version="1.0" encoding="UTF-8"?>  
<java version="1.6.0_03" class="java.beans.XMLDecoder">  
    <object class="User" method="getUser">  
        <string>admin</string>  
        <string>azerty</string>  
    </object>  
</java>
```

III-C-4 - Remarque

Un des avantages de la sérialisation XML, est que dans le cas où vous implémenteriez votre propre mécanisme de sérialisation via un *PersistenceDelegate*, il n'y a nul besoin d'implémenter le mécanisme de désérialisation associé.

III-D - Traiter les exceptions

Les classes `XMLEncoder` et `XMLDecoder` gèrent les exceptions levées lors des opérations de sérialisation/désérialisation afin de continuer le traitement des parties non affectées par l'erreur. Cependant, il est légitime de vouloir traiter soi-même ces exceptions. Pour cela il est possible d'affecter un `ExceptionListener` aux classes `XMLEncoder` et `XMLDecoder`.

L'interface `ExceptionListener` fournit une méthode `exceptionThrown()` qui est appelée lorsqu'une exception est levée :

```
XMLEncoder encoder = new XMLEncoder(System.out);
encoder.setExceptionListener(new ExceptionListener() {
    public void exceptionThrown(Exception ex) {
        ex.printStackTrace();
    }
});
e.writeObject( ... );
```



L'affectation d'un `ExceptionListener` à un objet `XMLDecoder` doit se faire par son constructeur et non par la méthode `setExceptionListener()`. En effet, la mise en place du mécanisme de désérialisation s'effectue à l'instanciation du `XMLDecoder`, et l'affectation d'un `ExceptionListener` par la méthode `setExceptionListener()` reste donc sans effet.

```
XMLDecoder decoder = new XMLDecoder(in, null, new ExceptionListener() {
    public void exceptionThrown(Exception exception) {
        exception.printStackTrace();
    }
});
```

IV - Conclusion

La sérialisation XML de l'API standard présente de nombreux avantages et est très simple à utiliser dans le cas où les objets à sérialiser répondent aux spécifications des JavaBeans. Dans le cas contraire les solutions existent mais elles se révèlent assez laborieuses à mettre en place et il est sûrement préférable d'utiliser une API externe.

L'API XMLStream est, par exemple, une alternative très intéressante.

V - Liens

-  [XMLEncoder](#)
-  [XMLDecoder](#)
-  [PersistenceDelegate](#)
-  [Using XMLEncoder](#)
-  [La sérialisation XML facile avec l'API XStream](#)
-  [BeanInfo](#)
-  [PropertyDescriptor](#)

VI - Remerciements

Je remercie Baptiste Wicht, RideKick et rem02 pour leurs corrections.

VII - Téléchargements

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

