
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES CENTRE VAL DE LOIRE

# Java

benjamin.nguyen@insa-cvl.fr

d’après J.-M. Doudoux

# Sérialisation

# But du cours (compétences)

- Savoir coder une application complexe en Java (avec des tests unitaires)
- Savoir débugger une application complexe en Java
- Savoir gérer les fichiers en Java
- Savoir gérer la persistence en Java avec la sérialisation
- Savoir utiliser les threads en Java
- Savoir créer une interface graphique simple en Java (approche modèle MVC)

*Si le temps le permet …*

- Connaître les bibliothèques cryptographiques en Java
- Comprendre et avoir utilisé le mécanisme d'introspection en Java
- Savoir gérer la persistence en Java avec une base de données

# L'interface `java.io.Serializable`

- Avantage : si on est dans un cas "simple" : il n'y a (pratiquement) rien à faire !
- Utilisation d'un `ObjectOutputStream` et d'un `ObjectInputStream`, ouverts sur un `FileOutputStream` / `FileInputStream`
- La classe qu'on veut sérialise doit implémenter l'interface `Serializable`, mais on n'a besoin d'implémenter aucune méthode !
- Voir exemples :
  - ExempleSimpleSer et ExempleSimpleDeser

# Utilisation de `serialVersionUID`

- Si on ne met rien : valeur calculée par défaut et mise à jour si on modifie la classe
- Sinon : on doit gérer soi-même les incompatibilités

- Voir ExempleSimpleSer avec des modifications

# Reconstruction d’un objet

- Une reserialisation n’appelle pas le constructeur ! (un peu comme ce qu’on a vu avec les fichiers)
- Peut poser des problèmes !

- Voir ExempleSerialize et ExempleLecture
- Il faut dans ce cas redéfinir la méthode :

```
private void readObject(java.io.ObjectInputStream in)
                        throws IOException, ClassNotFoundException
```

- Voir EmployeAmeliore, ExempleSerializeAmeliore, ExempleLectureSansErreurCompte

# Validation au chargement de classe

- Utilisation de la méthode

```
public void validateObject() throws InvalidObjectException
```

Voir : MaClasseValidation et SerDeserMaClasseValidation

# Utilisation de `transient`

- Si on n’a pas envie de sérialiser un champ !

- Voir ExempleTransient

# Attention à l’héritage lors de la construction de classes sérialisables

Si une classe mère implémente l'interface Serializable alors toutes les classes filles en héritent : il est donc inutile d'implémenter explicitement l'interface Serializable pour une classe fille.

De fait, il n'est pas possible de facilement savoir si une classe est sérialisable ou non uniquement en regardant son code source : il est nécessaire de savoir si une classe mère implémente l'interface Serializable.

# Classe mère non sérialisable

L'héritage peut aussi avoir un impact sur la sérialisation notamment lorsqu'une classe fille est sérialisable mais qu'aucune de ses classes mères l'est.

Dans ce cas, l'état des propriétés de la classe mère est ignoré lors de la sérialisation : la valeur par défaut de ces propriétés est alors celle par défaut selon leur type.

Si la classe mère n'est pas sérialisable, son état n'est pas pris en compte lors de la sérialisation/désérialisation même si ses champs sont hérités et accessibles dans la classe fille.

Voir : MaClasseMere, MaClasseFille, SerDeserMaClasseFille

# Sérialisation personnalisée

- Il est possible de personnalisé la sérialisation et désérialisation
- Dans certains cas, il est nécessaire d'utiliser cette personnalisation, par exemple :
  - pour contrôler les instances utilisées
  - pour tenir compte des données sensibles
  - pour gérer la sérialisation de différentes versions de la classe
- Plusieurs mécanismes sont proposés en standard pour personnaliser la sérialisation/désérialisation :
  - définir explicitement la liste des champs à inclure dans la sérialisation
  - définir les méthodes writeObject() et readObject()
  - définir les méthodes writeReplace() et readResolve()
  - implémenter l'interface Externalizable

# Sérialisation personnalisée : choix des champs

- Il est possible de préciser explicitement la liste des champs qui devront être pris en compte lors de la sérialisation/désérialisation en définissant un champ qui est un tableau de type java.io. ObjectStreamField.
- Ce champ doit obligatoirement se nommer `serialPersistentFields` et doit être déclaré avec les modificateurs `private`, `static` et `final`.
- **Attention :** l'utilisation d'un champ serialPersistentFields remplace purement et simple le mécanisme de recherche par défaut des champs à sérialiser. Ainsi un champ marqué transient mais défini dans le champ serialPersistentFields sera sérialisé.

- Voir ExemplePerso1, ExemplePerso2

# Sérialisation personnalisée : avec writeObject et readObject

- Les mécanismes de sérialisation/désérialisation par défaut ne sont pas toujours adaptés à certains besoins spécifiques qui requièrent une personnalisation des actions réalisées.
- Pour cela, il est possible de définir les méthodes writeObject() et readObject() dans la classe à sérialiser.
- Comme défini dans les spécifications de l'API Serialization, la signature de ces deux méthodes doit obligatoirement être :
  - private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException
  - private void writeObject(ObjectOutptStream oos) throws IOException
- Lors de la sérialisation/désérialisation d'un objet de la classe, ces deux méthodes seront invoquées en remplacement du mécanisme standard.
- Cette section propose plusieurs cas d'utilisation de la personnalisation en utilisant les méthodes writeObject() et readObject().
- Le premier cas permet de ne pas sérialiser un champ qui peut par exemple contenir des données sensibles comme un mot de passe.

# Gestion de multiples instances writeReplace() et readResolve()

- Lors de l'utilisation du mécanisme de sérialisation, il est parfois nécessaire d'avoir un contrôle sur l'instance obtenue ou à sérialiser. C'est notamment le cas si la classe est un singleton.
- Ce cas d'utilisation met en oeuvre les méthodes writeReplace et readResolve().
- La méthode readResolve() permet d'avoir un contrôle direct sur le type et l'instance retournés lors de la désérialisation. Elle doit avoir la signature suivante :

  Object readResolve() throws ObjectStreamException;

- La classe ObjectInputStream vérifie si la classe possède une méthode readResolve() avec cette signature et si c'est le cas elle l'invoque à la place du mécanisme standard. L'objet retourné par la méthode readResolve() doit cependant être compatible avec la classe de l'objet sérialisé sinon une exception de type ClassCastException est levée.
- La méthode writeReplace() permet de remplacer l'instance de l'objet qui sera sérialisé. Elle doit avoir la signature suivante :

  Object writeReplace() throws ObjectStreamException;

- La classe ObjectInputStream vérifie si la classe possède une méthode writeReplace() avec cette signature et si c'est le cas elle l'invoque pour obtenir l'instance à sérialiser. Le type de cette instance doit être compatible avec la classe de l'objet sérialisé sinon une exception de type ClassCastException est levée.
- Ces deux méthodes sont utilisables pour des classes qui implémentent Serializable ou Externalizable.
- Les méthodes readResolve() et WriteReplace() peuvent avoir n'importe quel modificateur d'accès.

# Interface `Externizable`

- L'implémentation de l'interface Externalizable permet d'avoir un contrôle très fin sur les opérations de sérialisation et désérialisation lorsque les mécanismes de sérialisation par défaut ne répondent pas au besoin.
- L'interface Externalizable hérite de l'interface Serializable. Elle définit deux méthodes :

| Méthode | Rôle |
|---|---|
| void readExternal(ObjectInput in) | Désérialiser de manière personnalisée l'objet à partir du flux passé en paramètre. Les méthodes de l'objet de type DataInput permettent de lire des valeurs primitives et la méthode readObject() de la classe ObjectInput permet de lire et créer des objets |
| void writeExternal(ObjectOutput out) | Sérialiser de manière personnalisée l'état de l'objet dans le flux passé en paramètre. Les méthodes de l'objet de type DataOutput permettent d'écrire des valeurs primitives et la méthode writeObject() de la classe ObjectOutput permet d'écrire des objets |

# Note sur la sécurité

- La sérialisation n'est pas sécurisée : même si le format par défaut est un format binaire, ce format est connu et documenté. Le contenu des données binaires peut assez facilement permettre de définir une classe qui permettra de lire le contenu du résultat de la sérialisation.
- Un simple éditeur hexadécimal permet d'obtenir les valeurs des différents champs sérialisés même ceux qui sont déclarés privés.
- Il ne faut pas sérialiser de données sensibles par le processus de sérialisation standard car cela rend public ces données. Une fois un objet sérialisé, les mécanismes d'encapsulation de Java ne sont plus mis en oeuvre : il est possible d'accéder aux champs private par exemple. Il faut soit :
  - exclure ces champs de la sérialisation
  - encrypter/décrypter ces données avec une personnalisation de la sérialisation
  - encrypter tous les résultats de la sérialisation en utilisant la classe javax.crypto.SealedObject ou la classe java.security.SignedObject qui implémente l'interface Serializable. Elles permettent d'encrypter et de signer un objet
  - sérialiser une instance d'un proxy qui ne contient pas ces données
- Le mécanisme de désérialisation permet de créer de nouvelles instances : tous les contrôles qui sont faits dans le constructeur doivent aussi être faits lors de la désérialisation.

(voir exemples Personne, Personne2 et Personne3)

# Exceptions liées à la sérialisation

| Exception | Rôle |
|---|---|
| ObjectStreamException | Classe mère des exceptions liées à la sérialisation |
| InvalidClassException | Levée lorsque qu'un objet ne peut pas être désérialisé à cause de sa classe :<br>•le numéro de version de la classe ne correspond pas à celui des données sérialisées<br>•le type d'un champ primitif ne correspond pas à celui d'une donnée sérialisée<br>•la classe implémente l'interface Externalizable mais ne possède pas de constructeur par défaut accessible<br>•La classe implémente l'interface Serializable mais une classe mère n'est pas sérialisable et ne possède pas de constructeur par défaut accessible |
| NotSerializableException | La classe n'est pas sérialisable |
| StreamCorruptedException | Le flux qui contient les données sérialisées est corrompu ou invalide (lecture de données sérialisées au format v2 avec un JDK inférieur à 1.1.5) |
| NotActiveException | La sérialisation n'est pas active |
| InvalidObjectException | La validation d'un objet désérialisé a échoué |
| OptionalDataException | La lecture de données primitives a échoué |
| WriteAbortedException | Une erreur est survenue durant l'écriture du flux |