

# Persistence

F. Mallet

*miage.m1@gmail.com*

*<http://deptinfo.unice.fr/~fmallet/>*

# La persistance

- Sauvegarder l'état d'une application
  - L'état des objets est perdu si l'application termine
  - On veut réaliser une fonction **sauve/restaure**
  - Voir le patron « Memento »
  - Il faut stocker l'état des objets dans
    - Un fichier sur le disque
    - Une base de données
- Plusieurs mécanismes possibles
  - Interfaces Serializable/Externalizable
  - Java Persistence API (JPA), Java Data Object (JDO),  
...

# Mémento (1/2)

## □ Intention

- Mécanisme de sauvegarde et de restauration d'objets

## □ Structure

- Le **créateur/originator** : dont on veut sauver l'état
- Le **Memento** : celui qui conserve l'état
  - L'introspection ou la Serialization peuvent être une solution
- Le **gardien/caretaker** : qui stocke les mémento



# Memento (2/2)

## □ Exemple

```
class Créateur {
    Object state;
    Object save() {
        return new Memento(state);
    }
    void restore(Object o) {
        Memento m = (Memento)o;
        this.state = m.state;
    }
    static class Memento {
        Object state;
        Memento(Object s) { state = s; }
    }
}

class Gardien {
    ArrayList<Object> states = new ArrayList<Object>();
    void addMemento(Object o) { states.add(o); }
}
```

### □ Interface sans méthode

- Une annotation aurait fait l'affaire !

### □ Exemple

```
class Foo implements Serializable {  
    Random generateur = new Random();  
    int state;  
    Foo() { this.next(); }  
    void next() { state = generateur.nextInt(20); }  
    public String toString() {  
        return "Foo("+state+")";  
    }  
}
```

# java.io.ObjectOutputStream

## □ Sérialisation / *Marshalling*

- Pour sauvegarder l'état il faut encoder l'objet

## □ Usage

```
FileOutputStream fos = new FileOutputStream("Foo.bin");  
ObjectOutputStream oos = new ObjectOutputStream(fos);  
Foo foo = new Foo();  
System.out.println(foo);  
oos.writeObject(foo);  
foo.next();  
System.out.println(foo);  
oos.close(); fos.close();
```

# java.io.ObjectInputStream

## □ Désérialisation / *Unmarshalling*

- Pour récupérer l'état il faut décoder l'objet

## □ Usage

```
FileInputStream fis = new FileInputStream("Foo.bin");  
ObjectInputStream ois = new ObjectInputStream(fis);  
Foo foo = (Foo)ois.readObject();  
System.out.println(foo);  
foo.next();  
System.out.println(foo);  
ois.close();  
fis.close();
```

cf. Demo

# Sauvegarde en profondeur

- Il ne s'agit pas seulement de sauvegarder un int
  - Tous les champs Serializable sont sauvegardés
  - Et tous les champs des champs
  - ...



# Le mot clé *transient*

- Identifie les champs qui ne doivent pas être sérialisés
  - Mais ils ne sont pas restaurés, non plus !

## □ Exemple

```
class Foo implements Serializable {  
    transient Random generateur = new Random();  
    int state;  
    Foo() { this.next(); }  
    void next() { state = generateur.nextInt(20); }  
    public String toString() {  
        return "Foo("+state+")";  
    }  
}
```

# Sérialisation et dé-sérialisation

## □ Sérialisation

- L'état complet de l'objet est transformé en chaîne de caractères : publics, privés, hérités
- Une numéro de version est généré automatiquement à partir d'une analyse structurelle de la classe:
  - **static final int serialVersionUID**

## □ Dé-sérialisation

- L'objet est reconstruit à partir de la chaîne
- Numéro de version incorrect => **InvalidClassException**
- Le constructeur n'est pas invoqué et les initialisations par défaut ne s'appliquent pas !
  - Le champ `generateur` n'est donc pas reconstruit si il est déclaré *transient*

# serialVersionUID

- ❑ La moindre modification modifie le numéro de version
  - La dé-sérialisation devient alors impossible
  - Le champ `state` devient `value`

## ❑ Exemple

```
class Foo implements Serializable {  
    Random generateur = new Random();  
    int state;  
    Foo() { this.next(); }  
    void next() { state = generateur.nextInt(20); }  
    public String toString() {  
        return "Foo("+state+")";  
    }  
}
```

# serialVersionUID

- ❑ La moindre modification modifie le numéro de version
  - La dé-sérialisation devient alors impossible
  - Le champ `state` devient `value`

## ❑ Exemple

```
class Foo implements Serializable {
    Random generateur = new Random();
    int value;
    Foo() { this.next(); }
    void next() { value = generateur.nextInt(20); }
    public String toString() {
        return "Foo("+value+")";
    }
}
```

# serialVersionUID

- Définir son propre numéro de version
  - Définir un champ final `static long serialVersionUID` pour chaque classe qui implémente `serializable`

## □ Exemple

```
class Foo implements Serializable {
    static final long serialVersionUID = 42L;
    Random generateur = new Random();
    int value;
    Foo() { this.next(); }
    void next() { value = generateur.nextInt(20); }
    public String toString() {
        return "Foo("+value+")";
    }
}
```

# writeObject et readObject

□ Pour contrôler le processus, on peut définir les méthodes **privées** suivantes:

- Sauvegarder l'état de l'objet dans `out`

```
private void writeObject(ObjectOutputStream out)  
                                throws IOException;
```

- Seulement les champs **non-statique** et **non-transient** de la classe courante, pas les champs de la classe mère
- Peut invoquer la méthode **defaultWriteObject** qui implémente le comportement par défaut

# writeObject et readObject

- Pour contrôler le processus, on peut définir les méthodes **privées** suivantes:

- Restaure l'état depuis `in`

```
private void readObject(ObjectInputStream in)  
    throws IOException, ClassNotFoundException;
```

- Peut invoquer la méthode **defaultReadObject** qui implémente le comportement par défaut
- Si la classe hérite d'une classe qui n'est pas **Serializable**, la classe mère doit avoir un constructeur par défaut qui sera invoqué automatiquement.

# writeObject et readObject

- ❑ Permet de sérialiser manuellement les objets non Serializable (Thread, JComponent, ...)
- ❑ Example:

```
class Foo implements Serializable {
    static final long serialVersionUID = 42L;
    Random generateur = new Random();
    int value;
    ...
    private void writeObject(ObjectOutputStream out)
                                throws IOException {
        out.writeInt(value);
        out.writeObject(generateur);
    }
}
```



# writeObject et readObject

## □ Example:

```
class Foo implements Serializable {
    static final long serialVersionUID = 42L;
    Random generateur = new Random();
    int value;
    ...
    private void readObject(ObjectInputStream in)
        throws IOException, ClassNotFoundException {
        value = in.readInt();
        generateur = in.readObject();
    }
}
```

# readObjectNoData

## □ Méthode de secours

- Si le flot ne contient pas d'information sur l'objet
- Par exemple, lorsqu'un champ a été ajouté depuis la précédente sérialisation

## □ Example:

```
class Foo implements Serializable {
    static final long serialVersionUID = 42L;
    Random generateur = new Random();
    int value;
    ...
    private void readObjectNoData()
                                throws ObjectStreamException {
        generateur = new Random();
        this.next();
    }
}
```

# writeReplace

- Donne un alternative pour la sérialisation
  - Si la méthode `writeReplace` est définie et est accessible
    - l'objet qu'elle renvoie est sérialisé à la place de `this`

## □ Example:

```
class Foo implements Serializable {
    static final long serialVersionUID = 42L;
    Random generateur = new Random();
    int value;
    ...
    Object writeReplace() throws ObjectOutputStreamException {
        return generateur;
    }
}
```

# readResolve

- Donne un alternative pour la désérialisation
  - Si la méthode `readResolve` est définie et est accessible
    - l'objet qu'elle renvoie est désérialisé à la place de `this`

## □ Example:

```
class Foo implements Serializable {
    static final long serialVersionUID = 42L;
    Random generateur = new Random();
    int value;
    ...
    private Object readResolve() throws ObjectStreamException {
        return generateur;
    }
}
```

## java.io.Externalizable

- Donne un contrôle complet sur la sérialisation et désérialisation
  - Seul le numéro de version est stocké
  - Rien n'est automatique (même pas super)
  - Doit avoir un constructeur par défaut !

```
interface Externalizable extends Serializable {  
    public void writeExternal(ObjectOutput out)  
        throws IOException;  
    public void readExternal(ObjectInput in)  
        throws IOException, ClassNotFoundException;  
}
```

# Conclusion

## □ Si un objet implémente **Externalizable**

- Séri­a­li­sa­tion: `writeExternal` sauf si `writeReplace`
- Déséri­a­li­sa­tion : `readExternal` sauf si `readResolve`
  - Le constructeur par défaut est appelé

## □ Sinon, si l'objet implémente **Serializable**

- Séri­a­li­sa­tion: `writeObject`
- Déséri­a­li­sa­tion: `readObject` ou `readObjectNoData`
  - Le constructeur n'est pas invoqué et n'existe pas nécessairement