INFO-H-200
Programmation orientée objet

Séance d'exercices 7
Persistence

Université libre de Bruxelles
École polytechnique de Bruxelles

Professeur : Hugues Bersini

2014-2015

Persistance

Sur fichiers texte

Persistance

La **persistance** des données réfère au mécanisme responsable de la sauvegarde et de la restauration de données. Ces mécanismes font en sorte qu'un programme puisse se terminer sans que ses données et son état d'exécution ne soient perdus.

- La sauvegarde sous forme de fichiers texte
- La sérialisation
- La sauvegarde dans une base de données

Exemple

```
String data;
BufferedWriter file;
file = new BufferedWriter(new FileWriter(filename));
file.write(data);
file.close();
```

Persistance

Sérialisation

Persistance

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.

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

Persistance

Pour sérialiser un objet d'une classe, celle-ci doit implémenter l'interface **Serializable** ou hériter d'une classe sérialisable

L'interface **Serializable** ne possède ni attribut et ni méthode, elle sert uniquement à identifier une classe sérialisable. Tous les attributs de l'objet sont sérialisés mais à certaines conditions :

- être 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
- être hérité d'une classe mère sérialisable

Persistance

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.

Le **serialVersionUID** est un "numéro de version", associé à toute classe sérialisable, 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.

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

Exemple

```
public class TextSaved implements Serializable {
private static final long serialVersionUID = 43L;
public Integer value;
public String text;
public TextSaved(String text) {
value = 0;
this.text = text;
}
}
```

Exemple

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

```
TextSaved t = new TextSaved ("Hello World !");
ObjectOutputStream oos= new ObjectOutputStream(new
      FileOutputStream("personne.serial"));
oos.writeObject(t);
oos.flush();
oos.close();
```

Exemple

Pour l'opération inverse, nous allons utiliser la méthode readObject() de la classe **ObjectInputStream**.

```
TextSaved t = null;
ObjectInputStream ois= new ObjectInputStream(new
      FileInputStream("personne.serial"));
t = (TextSaved ) ois.readObject();
ois.close();
```

Sérialisation et héritage

1. Premier cas de figure, lorsqu'une classe hérite d'une classe sérialisable, elle se comporte comme si elle avait implémenté l'interface Serializable. Les attributs de la classe mère sont sérialisés selon son implémentation.
2. Deuxième cas de figure, une classe implémente l'interface Serializable et hérite d'une classe non sérialisable.
   - les attributs hérités ne sont pas sérialisés.
   - la classe étendue doit possèder un constructeur par défaut accessible ; dans le cas contraire, une InvalidClassException est levée à l'exécution.

Personnalisation de la sérialisation

1. 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).
2. sérialiser des attributs que le mécanisme par défaut ne sérialiserait pas (un attribut static par exemple).

Exemple

```
public class TextSaved {
private transient int serialisationCount = 0;

private void writeObject(ObjectOutputStream out) throws IOException {
out.defaultWriteObject();
out.writeInt(serialisationCount);
}
private void readObject(ObjectInputStream in) throws IOException,
      ClassNotFoundException {
in.defaultReadObject();
serialisationCount = in.readInt();
}
}
```

Persistance

Base de données

Persistance

Avec la plupart des bases de données, les interactions avec une application se font via l'utilisation de requêtes **SQL** (Structured Query Language).

Pour la persistance des données, on va se limiter à trois éléments :

- la création d'une base de données ;
- l'ajout d'un élément ;
- la récupération des éléments.

Exemple

```
SQLiteConnection db;
File f = new File(filename);
db = new SQLiteConnection(f);
db.open(true);
db.exec("CREATE TABLE products(id INTEGER PRIMARY KEY, name TEXT, size INTEGER,
     price INTEGER);");
```

Exemple

```
db.exec("INSERT INTO products(name, size, price)" + "VALUES('car', 3, 19)");

string t = "car";
int a = 3;
int b = 18;
db.exec("INSERT INTO products(name, size, price) " + "VALUES('" + t + "',"+a+
     ","+b+")");
```

Exemple

```
SQLiteStatement st = db.prepare("SELECT id, name, size, price FROM products;");
int colnum = st.columnCount();
string cn = st.getColumnName(i);
string cv = st.columnValue(i).toString();
```