Introduction à la programmation en JAVA

Table des matières

La sérialisation

Aperçu du chapitre

I . Les flux I/O

- La **sérialisation** est un procédé qui permet de rendre un objet de la JVM **persistant** pour **stockage** ou **échange** et vice versa.

- Cet objet est mis sous une forme sous laquelle il pourra **être reconstitué** à l'identique.

➔ Ainsi, il pourra être stocké sur un disque dur ou transmis au travers d'un réseau pour le créer dans une autre JVM.

- Un objet dans lequel on peut écrire une séquence d'octets (bytes) est appelé **output stream** (flux de sortie).

- Un objet duquel on peut lire une séquence d'octets (bytes) est appelé **input stream** (flux d'entrée).

I . Les flux I/O

- Les classes abstraites **OutputStream** et **InputStream** sont à la base de la hiérarchie des classes I/O et donc, de la sérialisation.

- Les **sources** et **destinations** des flux sont généralement des **fichiers**, mais il pourrait également s'agir de connexions réseau, de blocs mémoire, …

- Toute opération impliquant la sérialisation doit prévoir un comportement en cas de **IOException**.

Aperçu du chapitre

I. Les flux I/O

II. Flux de sortie

III. Flux d'entrée

IV. Exercices

II . Flux de sortie

Pour lier un flux de sortie à un fichier, on utilise la classe **FileOutputStream** qui comprend les 4 constructeurs suivants :

- FileOutputStream(File)
- FileOutputStream(String)
- FileOutputStream(File, boolean)
- FileOutputStream(String, boolean)

Deux méthodes sont donc possibles :

1. On passe un objet File en paramètre. Cet objet est obtenu de la manière suivante :
   File f = new File (« chemin_acces_fichier »);
2. On passe une chaîne de caractères correspondant au chemin d'accès du fichier.

II . Flux de sortie

Facultatif, l'attribut booléen permet de spécifier si l'ancien contenu du fichier doit être écrasé ou non (si le fichier existe préalablement) :

- false (par défaut) : le nouveau contenu remplace l'ancien
- true : le nouveau contenu est placé juste en dessous de l'ancien

Toute opération impliquant l'utilisation de fichiers doit prévoir un comportement en cas de **FileNotFoundException**.

II . Flux de sortie – *Sauvegarde des types primitifs*

Pour écrire des types primitifs dans un flux, on utilise la classe **DataOutputStream**.

```
DataOutputStream out = new DataOutputStream(new FileOutputStream
(« chemin_acces_fichier »));
```

Les méthodes suivantes sont disponibles :

- writeBoolean(boolean)
- writeChar(char)
- writeDouble(double)
- writeFloat(float)
- writeInt(int)
- writeLong(long)
- writeShort(short)

Attention, pour éviter toute fuite mémoire, un flux doit être fermé après utilisation. Ceci se fait à l'aide de la méthode **close**.

```
out.close();
```

II . Flux de sortie – *Sauvegarde des types objets*

Pour écrire des types objets dans un flux, on utilise la classe **ObjectOutputStream**.

```
ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(« … »));
```

Les méthodes suivantes sont disponibles :

- writeObject(Object);

Pour **écrire un objet** dans un flux, sa classe doit impérativement implémenter l'**interface Serializable** !

Attention, pour éviter toute fuite mémoire, un flux doit être fermé après utilisation. Ceci se fait à l'aide de la méthode **close**.

```
out.close();
```

II . Flux de sortie – *Sauvegarde des chaînes de caractères*

Pour écrire des String dans un flux, on utilise les classes **PrintWriter** et **FileWriter**.

```
PrintWriter out = new PrintWriter (new FileWriter(« chemin_acces_fichier »));
```

Les méthodes suivantes sont disponibles :

- print(String)
- println(String)

Attention, pour éviter toute fuite mémoire, un flux doit être fermé après utilisation. Ceci se fait à l'aide de la méthode **close**.

```
out.close();
```

Aperçu du chapitre

I. Les flux I/O

II. Flux de sortie

III. Flux d'entrée

IV. Exercices

III . Flux d'entrée

Pour lier un flux d'entrée à un fichier, on utilise la classe **FileInputStream** qui comprend les 2 constructeurs suivants :

- FileInputStream(File)
- FileInputStream(String)

Deux méthodes sont donc possibles :

1. On passe un objet File en paramètre. Cet objet est obtenu de la manière suivante :
   File f = new File (« chemin_acces_fichier »);
2. On passe directement une chaîne de caractères correspondant au chemin d'accès du fichier.

Toute opération impliquant l'utilisation de fichiers doit prévoir un comportement en cas de **FileNotFoundException**.
De plus, la lecture de fichiers implique de prévoir également un comportement en cas de **EOFException**.

III . Flux d'entrée – *Lecture des types primitifs*

Pour lire des types primitifs dans un flux, on utilise la classe **DataInputStream**.

```
DataInputStream in = new DataInputStream(new FileInputStream
(« chemin_acces_fichier »));
```

Les méthodes suivantes sont disponibles :

- readBoolean()
- readChar()
- readDouble()
- readFloat()
- readInt()
- readLong()
- readShort()

Attention, pour éviter toute fuite mémoire, un flux doit être fermé après utilisation. Ceci se fait à l'aide de la méthode **close**.

```
in.close();
```

III . Flux d'entrée – *Lecture des types objets*

Pour lire des types objets dans un flux, on utilise la classe **ObjectInputStream**.

```
ObjectInputStream in = new ObjectInputStream(new FileInputStream(« … »));
```

Les méthodes suivantes sont disponibles :

- readObject();

Pour **lire un objet** dans un flux, sa classe doit impérativement implémenter l'**interface Serializable** !

De plus, pour la lecture des objets, il faut prévoir un comportement en cas de **ClassNotFoundException**.

Attention, pour éviter toute fuite mémoire, un flux doit être fermé après utilisation. Ceci se fait à l'aide de la méthode **close**.

```
in.close();
```

III . Flux d'entrée – *Lecture des chaînes de caractères*

Pour lire des String dans un flux, on peut utiliser la classe **Scanner** ou la classe **BufferedReader** avec la classe **FileReader**.

Scanner :

```
Scanner scan = new Scanner(new FileReader(« chemin_acces_fichier »));
while (scan.hasNext()) {
        String ligne = scan.nextLine()
        …
}
scan.close();
```

BufferedReader :

```
BufferedReader br = new BufferedReader(new FileReader(« chemin_acces_fichier »));
while (true) {
        String ligne = br.readLine()
        if (ligne == null) { break; }
        …
}
```

Attention, pour les String, pas de EOFException!

Aperçu du chapitre

I. Les flux I/O

II. Flux de sortie

III. Flux d'entrée

IV. Exercices

IV . Exercices

1. Créez et exécutez un programme qui demande à l'utilisateur d'encoder un nom de fichier.
Créez ce fichier et demandez ensuite à l'utilisateur d'encoder des nombres entiers.
Ecrivez chaque nombre entier encodé dans le fichier précédemment créé.
Si l'utilisateur encode le chiffre 0, l'application se termine en affichant dans la console le contenu du fichier.
Pensez à gérer toutes les exceptions !

2. Faites le même exercice que le 1e en remplaçant les nombres entiers par des String (« stop » servira à arrêter le programme). Pour la lecture du fichier, utilisez les deux classes Scanner et BufferedReader.

3. Créez une classe Personne avec comme attributs : nom, prénom, date de naissance et métier.
Créez un objet Personne et écrivez son état dans la console.
Ecrivez-le ensuite dans un fichier .dat.
Enfin, lisez le fichier que vous venez de créer et affichez l'état de l'objet Personne dans la console.

IV . Exercices

4. Créer une classe Personne.

Créer ensuite deux classes Main :

- La première demande à l'utilisateur d'encoder un nom de fichier pour le créer. Elle lui propose ensuite de créer autant d'objets Personne voulus (via un menu). Chaque objet Personne sera stocké dans un document XML (utiliser la sérialisation).
- La deuxième demande à l'utilisateur d'encoder un nom de fichier pour le lire. Utiliser une API au choix pour lire le fichier XML et stocker toutes les Personnes dans une List. Afficher ensuite la List dans la console.