Sérialisation

Table des matières

Sérialisation 1
Introduction 1
Utiliser un fichier 2
Avant Java 7 : File 2
Path 2
Files 3
Lecture et Ecriture dans un fichier 3
try-with-resources 5
Sérialiser les objets 8
La sérialisation personnalisée 9
Fichier de `Properties` 9
Lire dans `Properties` 9
Ecrire dans `Properties` 10
`Properties` en XML 10

Introduction

Les objets que nous manipulons jusqu'à présent naissent et vivent dans la JVM jusqu'à ce que le garbage collector s'occupe d'eux. Il arrive qu'il soit nécessaire de conserver l'état de certains objets entre deux exécutions, par exemple la modification du solde d'un compte en banque suite à une opération ou encore la modification d'une adresse.

Il est parfois indispensable donc de mémoriser l'information avant d'interrompre un programme. Par mémoriser, on entend ici mémoriser sur le disque dur. Plusieurs possibilités s'offrent à nous pour faire cela :

- Utiliser un fichier
- Sérialiser les objets
- Manipuler des bases de données

Cette dernière manière de faire ne sera pas abordée dans ce cours-ci.

Utiliser un fichier

Avant Java 7 : File

Avant Java7, l'accès au système de fichiers se faisait via la classe `File` qui permet de manipuler les fichiers et répertoires. L'utilisation de cette classe présente de nombreux inconvénients :

- Plusieurs méthodes ne lèvent pas une exception en cas de problème mais renvoient un booléen. Ceci ne respecte pas ce qu'il est possible de faire pour gérer les erreurs avec Java, rend l'API inconsistante et ne permet pas de connaître l'origine du problème mais seulement de savoir que la fonctionnalité a échoué.
- Certaines méthodes (comme `rename()`) n'ont pas le même comportement sur toutes les plateformes.
- Les métadonnées (attributs de permissions, propriétaire, sécurité, ...) sont peu ou mal supportées.
- Certaines fonctionnalités de base sont absentes de l'API comme la copie ou le déplacement d'un fichier.
- Certaines fonctionnalités sont peu performantes comme par exemple la méthode `listFiles()` avec un répertoire contenant de nombreux fichiers.

Depuis Java 7, la manipulation des fichiers repose sur plusieurs classes et interfaces définies dans le package `java.nio.file`. Les principales classes et interfaces de ce package sont :

- `Path` : encapsule un chemin dans le système de fichiers
- `Files` : contient des méthodes statiques pour manipuler les éléments du système de fichiers
- `FileSystemProvider` : service provider qui interagit avec le système de fichiers sous-jacent
- `FileSystem` : encapsule un système de fichiers

Path

L'interface `Path` encapsule un chemin permettant de localiser un fichier. Ce chemin localisera indifféremment un fichier ou un répertoire, avec ou sans chemin d'accès, relatif ou absolu. Un objet de type `Path` encapsule le chemin d'un élément du système de fichiers composés d'un ensemble d'éléments organisés de façon hiérarchique grâce à un séparateur spécifique au système d'exploitation. Path permet d'obtenir des informations sur le chemin, accéder aux éléments du chemin, convertir le chemin ou extraire des sous-chemins, ...

Pour créer un `Path`, on invoque la méthode `getPath`() d'une instance de type `FileSystem` ou en appelant la méthode statique `get`() de la classe `Paths` qui invoque elle-même la méthode `FileSystems.getDefault`().`getPath`().

Ex :

```
Path chemin = FileSystems.getDefault().getPath("c:/java/test/monfichier.txt");
```

Files

La classe `Files` contient une cinquantaine de méthodes statiques permettant de réaliser des opérations de base sur des fichiers ou des répertoires dont le chemin est encapsulé dans un objet de type `Path` (création, ouverture, suppression, test d'existence, changement des permissions, ...).

Lecture et Ecriture dans un fichier

Pour écrire et lire dans un fichier, on doit créer des flux de sortie `Writer` ou `OutputStream` et d'entrée `Reader` ou `InputStream`.

	Flux d'octets	Flux de caractères
Flux d'entrée	`InputStream`	`Reader`
Flux de sortie	`OutputStream`	`Writer`

La classe Files propose plusieurs méthodes pour faciliter la lecture ou l'écriture de fichiers et de flux selon les besoins allant des plus simples aux plus complexes. Par exemple, les méthodes `newInputStream`() et `newOutputStream`() permettent de créer des flux d'entrée/sortie.

Quand un programme lit ou écrit des octets dans un fichier sur un support matériel (disque dur, clé USB, etc.), chaque instruction de lecture ou d'écriture provoque la mise en marche du matériel. Il faut éviter cela sinon le programme sera lent, inefficace, etc.

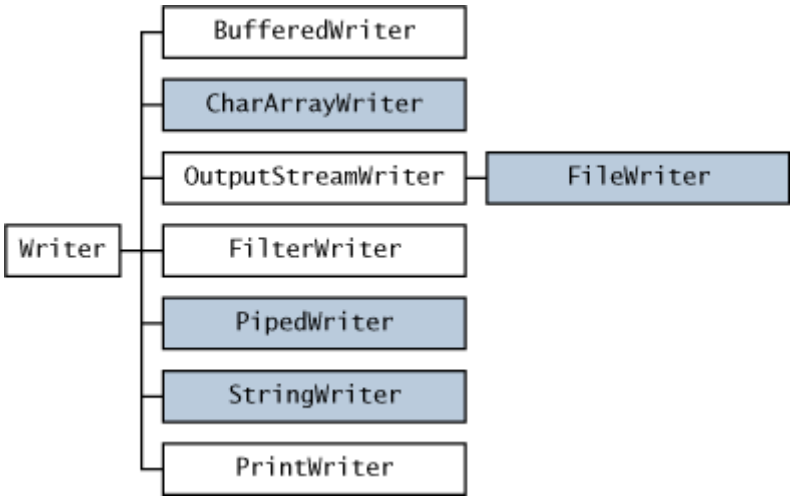
Pour bufferiser un flux d'entrée d'octets en entrée, on encapsule ce flux dans un `BufferedInputSream`.

```
Path path = FileSystems.getDefault().getPath("c:/java/test/monfichier.txt");
InputStream fis = Files.newInputStream(path) ;
BufferedInputStream bis = new BufferedInputStream(fis) ;
```

Le `BufferedInputSream` s'utilise comme le `FileInputStream` mais il réalise l'opération de bufferisation.

Writer

La classe abstraite `Writer` expose les méthodes qui permettent d'écrire des caractères dans un fichier, une `String`, la console Système, un socket, …




```
public static void main(String[] args) {
        Path path = FileSystems.getDefault().getPath("Z:/zouc/coucou.text");
        try {
            // Création du flux de sortie créé par le fichier
            Writer writer = Files.newBufferedWriter(path, StandardCharsets.UTF_8) ;

            // Ecriture dans le fichier
            writer.write("Coucou l'IPL !") ;

            // Fermeture du flux (cette méthode appelle flush())
            writer.close() ;
        } catch (IOException e) {
            System.out.println("Erreur " + e.getMessage()) ;
            e.printStackTrace() ;
        }

}
```

Ce code ci-dessus fonctionne : il écrit "Coucou l'IPL !" dans un fichier qui s'appelle coucou.text dans le répertoire Z:/zouc. Attention, il faut créer le répertoire zouc !

MAIS … ce code n'est absolument pas une bonne manière de faire !

La méthode `close()` (appelle `flush()` qui permet de vider les buffers d'écriture vers la sortie) n'est appelée que si aucune erreur n'est rencontrée lors de l'exécution du code. Mais, si l'écriture ne se déroule pas correctement et jette une exception, alors l'exécution du code passe dans le bloc `catch`. Dans ce cas, le fichier qui a été ouvert n'est pas fermé. Dans ce petit bout de code, ce n'est pas un problème car la JVM s'interrompt à la fin du `main` mais dans des applications plus élaborées, ceci est un problème récurrent. Il est donc indispensable d'ouvrir et de fermer correctement les fichiers.

```
public static void main(String[] args) {
	Path path = FileSystems.getDefault().getPath("Z:/zouc/coucou.text");
	Writer writer = null ;
	try {
		writer = Files.newBufferedWriter(path, StandardCharsets.UTF_8) ;
		writer.write("Coucou l'IPL !") ;
	}  catch (IOException e) {
		System.out.println("Erreur " + e.getMessage()) ;
		e.printStackTrace() ;
	}  finally {
		// si le writer n'avait pas été initialisé
		if (writer != null) {
			try {
				writer.close() ;
			   }  catch (IOException e) {
				System.out.println("Erreur " + e.getMessage()) ;
				e.printStackTrace() ;
			   }
		}
	}
}
```

try-with-resources

L'inconvénient de cette solution est que l'exception qui peut être levée par la méthode `close` n'est pas propagée, elle est attrapée et écrite dans le stack trace.

Avec Java 7, le mot clé `try` peut être utilisé pour déclarer une ou plusieurs ressources en paramètres de celui-ci : on utilise alors le mot clé `try` avec une ou plusieurs ressources définies dans sa portée, chacune séparée par un point-virgule.

Une ressource est un objet qui doit être fermé lorsque l'on a plus besoin de lui : généralement cette ressource encapsule ou utilise des ressources du système : fichiers, flux, connexions vers des serveurs, ...

L'utilisation d'un bloc `try-with-resources` garantit que chaque ressource sera fermée lorsqu'elle n'est plus utilisée. Une ressource est un objet qui implémente l'interface `AutoCloseable`.

L'interface `java.lang.AutoCloseable` a été définie pour indiquer qu'une ressource peut être fermée automatiquement. Tout objet qui implémente l'interface `AutoCloseable` peut être utilisé dans une instruction de type `try-with-resources`. Cette instruction `try` avec des ressources garantit que chaque ressource déclarée sera fermée à la fin de l'exécution de son bloc de traitement.

L'interface `Autocloseable` possède une unique méthode `close` qui sera invoquée pour fermer automatiquement la ressource encapsulée par l'implémentation de l'interface.

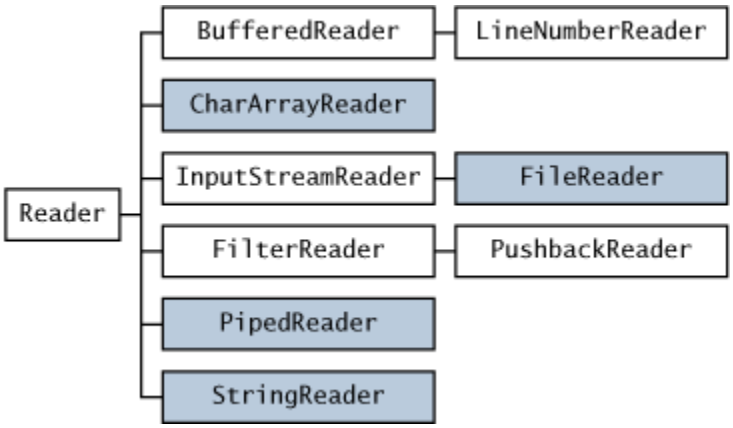
Plusieurs classes du JDK implémentent l'interface `AutoClosable` : `java.io.InputStream`, `OutputStream`, `Reader`, `Writer`, `java.sql.Connection`, `Statement`, et `ResultSet`.

```
public static void main(String[] args) {
	Path path = FileSystems.getDefault().getPath("Z:/zouc/coucou.text");
	try (Writer writer = Files.newBufferedWriter(path,
				StandardCharsets.UTF_8)) {
		writer.write("Coucou l'IPL !");

	} catch (IOException e) {
		System.out.println("Erreur " + e.getMessage());
		e.printStackTrace();
	}
}
```

Reader

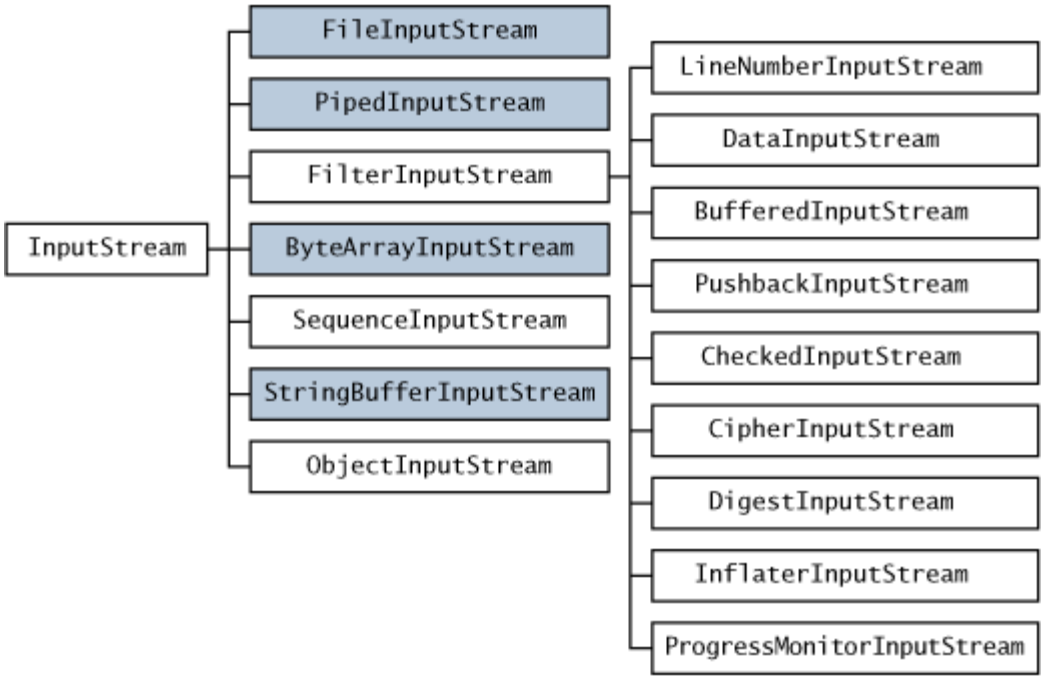
La classe `Reader` gère les flux de caractères en lecture.



Pour afficher à l'écran le contenu du fichier créé avec les `Writer`, il suffit de ces quelques lignes de codes :

```
Path path = FileSystems.getDefault().getPath("Z:/zouc/coucou.text");
BufferedReader br= Files.newBufferedReader(path);
Scanner sc = new Scanner(br);
String monTexte = sc.nextLine();
System.out.println(monTexte);
```

InputStream



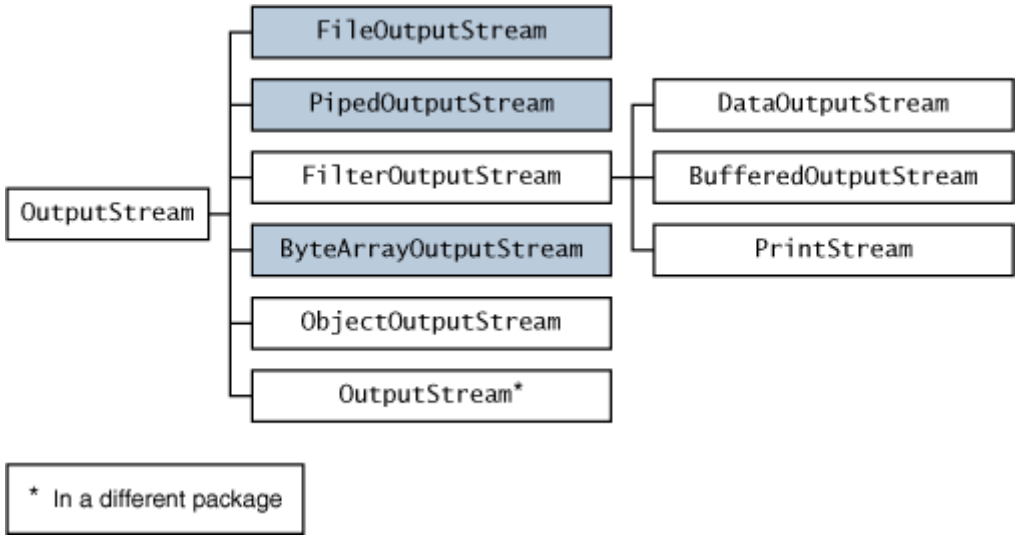
Pour les flux d'octets, le principe est le même.

On le manipule via les méthodes : `in.readDouble(), in.readInt(), ... in.readUTF()`

On le ferme : `in.close()`.

La fin de fichier est signalée par une `EOFException`.

OutputStream



On le manipule via les méthodes : `writeDouble(d), writeInt(n), ... writeUTF(str); // pour les strings`

On le ferme : `out.close();`

Lire au clavier

Pour lire à la console, il suffit d'employer la classe `Scanner`.

```
Scanner clavier = new Scanner(System.in)
```

Les méthodes `nextInt(), nextDouble(), ... nextLine()` permettent la lecture.

Sérialiser les objets

La sauvegarde de l'information telle que nous l'avons abordée jusque maintenant n'est pas vraiment adaptée à la programmation orientée Objet. La structuration des objets est totalement perdue ! Les associations entre les objets du système le sont aussi … la galère !

Heureusement, il existe une classe `ObjectOutputStream` qui supporte l'écriture directe d'objets sur des flux. Ce mécanisme s'appelle **sérialisation**. La sérialisation permet de garantir, entre autres, qu'une instance d'une classe écrite dans un fichier est bien recréée dans la bonne classe, identique à la première (entendez qui possède le même numéro de série).

Toute classe qui souhaite que ses instances puissent être sérialisées doit implémenter l'interface `Serializable`. L'implémentation de cette interface n'impose aucune méthode à la classe qui l'implémente ; elle signale simplement que la classe peut être sérialisée.

Les attributs `static` ne sont jamais sérialisés. Tous les autres attributs le sont sauf ceux qui sont déclarés explicitement `transient`.

Pour sérialiser :

```
HashMap<String,Article> map = /*…*/;
Path p = FileSystems.getDefault().getPath("fichier.ser");
// tester si le fichier est ouvrable en écriture
try (OutputStream out = Files.newOutputStream(p); ObjectOutputStream o =
                                                    new ObjectOutputStream(out);){
    o.writeObject(map);
    o.writeInt(125);
}
```

Pour désérialiser :

```
Path p = FileSystems.getDefault().getPath("fichier.ser");
// tester l'existence du fichier et s'il est ouvrable en lecture
try (InputStream in = Files.newInputStream(p); ObjectInputStream o =
                                                     new ObjectInputStream(in);){
    HashMap<String,Article> map = (HashMap<String,Article>)o.readObject();
    int i = o.readInt();
}
```

La sérialisation personnalisée

Pour personnaliser la manière dont la sérialisation va se dérouler. Il faut implémenter les méthodes `private void writeObject(ObjectOutputStream oos)throws IOException` et `private void readObject(ObjectInputStream ois)throws IOException,ClassNotFoundException.`

Lorsque la machine Java constate qu'une classe `Serializable` comporte ces deux méthodes, alors elle les appelle plutôt que d'utiliser ses mécanismes internes de sérialisation.

La méthode `writeObject()` a la responsabilité d'écrire les champs de l'objet sur le flux sérialisé passé en paramètre. On peut choisir de n'écrire qu'une partie des champs, cela ne pose pas de problème.

La méthode `readObject()` a la responsabilité de restaurer les valeurs des champs de l'objet. Son processus de lecture doit correspondre au processus d'écriture utilisé par la méthode `writeObject()`. Si une partie des champs n'a pas été écrite par la méthode d'écriture, alors la méthode `readObject()` peut restaurer ces valeurs à partir d'informations externes.

Il est important de noter que ces deux méthodes fonctionnent en tandem, et doivent donc être compatibles l'une avec l'autre.

```
private void writeObject(ObjectOutputStream oos) throws IOException {

	temp = motDePasse.clone();
	motDePasse = crypt(motDePasse);
	oos.defaultWriteObject();
	motDePasse = temp;
}

private void readObject(ObjectInputStream ois) throws IOException,
ClassNotFoundException {

	ois.defaultReadObject();
	motDePasse = deCrypt(motDePasse);
}
```

Fichier de `Properties`

Un fichier `Properties` est un fichier qui se compose de lignes ; chaque ligne est constituée d'une clef associée à une valeur.

Lire dans `Properties`

Pour récupérer des valeurs d'un fichier `Properties` :

```
	String configPath = "properties.prop";
	Properties properties = new Properties();

	Path p = FileSystems.getDefault().getPath(configPath);
	// tester l'existence du fichier et s'il est ouvrable en lecture
```

```
try (InputStream in = Files.newInputStream(p)) {
      properties.load(in);
} catch (IOException e) {
      e.printStackTrace();
}
// Pour afficher la valeur correspondant à la clef "uneCle" existante
// et une "uneSansCle" non existante
System.out.println(properties.getProperty("test", "default"));
System.out.println(properties.getProperty("uneSansCle", "defaultSansCle"));
```

Ecrire dans `Properties`

Pour écrire dans un fichier `Properties` :

```
String configPath = "properties.prop";
Properties properties = new Properties();
properties.setProperty("uneCle", "uneValeur");

try (OutputStream out = Files.newOutputStream(p)) {
      properties.store(out, "---config comment---");

 } catch (IOException e) {
      System.err.println("Unable to write config file.");
 }
```

`Properties` en XML

Il est possible de faire en sorte que le fichier de `Properties` soit du XML. Il suffit de remplacer les lignes de codes :

```
properties.load(in) →                 properties.loadToXML(in)

properties.store(out, "---bla---") →  properties.storeToXML(out, "---bla---")
```

Le fichier properties.xml contient :

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>---config comment---</comment>
<entry key="uneCle">uneValeur</entry>
</properties>
```

Le fichier properties.dtd contient les règles suivantes :

```
<!ELEMENT properties ( comment?, entry* ) >
<!ATTLIST properties version CDATA #FIXED "1.0">
<!ELEMENT comment (#PCDATA) >
<!ELEMENT entry (#PCDATA) >
<!ATTLIST entry key CDATA #REQUIRED>
```