Input / Output

Rémi Forax

Historique

Java 1.0 java.io

- InputStream/OutputStream, IOException

Java 1.1

- Reader/Writer

Java 1.4 java.nio.[buffer, charset]

- ByteBuffer, Charset, *Channel

Java 1.7 java.nio.file

- Path, Files
- try-with-resources

A retenir

Binaire: InputStream/OutputStream
Texte: Reader/Writer avec un Charset

Un fichier ouvert est une ressource système qu'il faut fermer

IOException est une exception qu'il faut propager (throws) pas traiter sur place

java.io.File ne marche pas correctement, utiliser java.nio.file.Path à la place

java.nio.file.Path

Représente un chemin dans l'arborescence

Création par static factory

- Paths.get(filename)
- Paths.get(directory, filename)

En interne, liste chainée de nom jusqu'à la racine

/ ← home ← forax ← ens ← io.odp

La gestion des '/' et '\' est fait en interne par la classe Path

Chemins

Il existe plusieurs sortes de chemins :

- Relatif ../toto/titi.txt
  dépend du répertoire courant où on lance le programme
- Absolue /tmp/../titi.txt
  démarre par une racine
- Canonique /tmp/
  pas de .. (ils sont résolus)
- Real c:\Program Files\Java\
  le fichier existe, les majuscules/minuscules sont résolus, c'est le chemin pour l'OS

Gestion des chemins

Nom textuel
- String **toString**()

Dernier élement comme un chemin
- Path **getFileName**()

Répertoire père
- Path **getParent**()

Vers un chemin absolue
- Path **toAbsolutePath**()

Vers un chemin réel
- Path **toRealPath**(LinkOption)

    linkOption permet de résoudre les liens symboliques

Résolution de chemin

Un Path par rapport à un autre Path/String

- Enlève les ..
  - Path **normalize**()
- Concatène à un chemin
  - Path **resolve**(Path) / **resolve**(String)
- Substitue par un frère
  - Path **resolveSibling**(Path) / **resolveSibling**(String)
- Créé un chemin relatif (inver de resolve)
  - **relativize**(Path path)

java.nio.file.Files

Permet de voir un Path comme un fichier

Garbage class contenant toutes les méthodes permettant de créer/détruire/lire/écrire sur des fichiers

Lève IOException si cela se passe mal

Attention, l'accès au système de fichier est concurrent, ce n'est pas parce que l'on test l'existence d'un fichier qu'il existe toujours à l'instruction suivante !

Méthodes statiques

Création

- createDirectory(), createFile(), createLink(), createSymbolicLink(), createTempDirectory(), createTempFile()

  prend en paramètre un Path et un FileAttribute

  (FileAttribute est créé à partir d PosixFilePermissions)

Existence

- isDirectory(), isRegularFile(), isHidden()

Droit

- getOwner(), getPosixFilePermission(), getFileAttributeView()

Parcourir les fichiers d’un répertoire

- newDirectoryStream()
  - attention pas un java.util.Stream, marche comme un itérateur
- Stream<Path> list()
  - Sous fichiers d’un répertoire sous forme de java.util.Stream
- walkFileTree()
  - Parcours une arborescence avec un Visiteur
- Stream<Path> walk()
  - Sous fichier de tous les répertoire d’une arborescence sous forme de java.util.Stream

Flux binaire /
Flux de caractères

Flux: binaire vs caractère

Deux versions suivant que l'on a besoin d'un Charset ou pas

InputStream/OutputStream

Flux d'**octets** (byte)

Reader/Writer

Flux de **caractères** (char)

nécessite un encodage (Charset)

Exemple

- Lire un fichier de propriété → Reader
- Ecrire dans un fichier binaire → OutputStream

Création

Sur java.nio.file.Files

- Flux binaire en lecture
  - InputStream **newInputStream**(Path)
- Flux binaire en écriture
  - OutputStream **newOutputStream**(Path)
- Flux textuel en lecture
  - BufferedReader **newBufferedReader**(Path, Charset)
  - BufferedReader **newBufferedReader**(Path)
    - Encodage UTF-8 par défaut
- Flux textuel en écriture
  - BufferedWriter **newBufferedWriter**(Path)
  - BufferedWriter **newBufferedWriter**(Path, Charset)
    - Encodage UTF-8 par défaut

InputStream

Flux de byte en entrée

Lit un byte et renvoie ce byte ou -1 si c'est la fin du flux

- int **read**()

  A ne jamais utiliser !

Lit un tableau de byte (plus efficace)

- int **read**(byte[] b)
- int **read**(byte[] b, int off, int len)

Saute un nombre de bytes

- long **skip**(long n)

Ferme le flux

- void **close**()

InputStream et appel bloquant

Les méthodes read() sur un flux sont bloquantes s'il n'y a pas au moins un byte à lire

Il existe une méthode available() dans InputStream qui est sensée renvoyer le nombre de byte lisible sans que la lecture sur le flux soit bloquée mais mauvais support au niveau des OS (à ne pas utiliser)

Lecture avec buffer

Lecture dans un tableau de bytes

- int **read**(byte[] b)
  renvoie le nombre de bytes lus
- int **read**(byte[] b, int off, int len)
  renvoie le nombre de bytes lus

Attention, la lecture est une demande pour remplir le buffer, le système essaye de remplir le buffer au maximum mais **peut décider** ne pas le remplir complètement

InputStream et problème d’efficacité

- Contrairement au C (stdio) en Java, les entrées sortie ne sont pas bufferisés par défaut
  - Il faut utiliser read qui prend un tableau de bytes
- Une librairie qui existe déjà lit byte par byte
  - On utilise un BufferedInputStream qui n’évite pas un appel de méthode par octet mais au moins évite un appel système par octet

Bufferisation automatique

BufferedInputStream/BufferedOuputStream agissent comme des proxies en installant un buffer intermédiaire entre le flux et le système

Constructeurs :

- BufferedInputStream(InputStream input,int bufferSize)
- BufferedOutputStream(OutputStream input,int bufferSize)

Attention à éviter le créer des buffered de buffered de buffered ...

OutputStream

Flux de byte en sortie

Ecrit un byte, en fait un int pour qu'il marche avec le read

void **write**(int b)
Ne jamais utiliser

Ecrit un tableau de byte (plus efficace)

- void **write**(byte[] b)
- void **write**(byte[] b, int off, int len)

Demande d'écrire ce qu'il y a dans le buffer

si c’est un BufferedOutputStream

- void **flush**()

Ferme le flux

- void **close**()

Exemple – Copie de flux

Copie en utilisant un buffer

```
public static void copy(InputStream in, OutputStream out)
  throws IOException {

  byte[] buffer = new byte[8192];
  int size;
  while((size = in.read(buffer)) != -1) {
    out.write(buffer, 0, size);
  }
}
```

La méthode existe déjà

```
inputStream.transferTo(outputStream);
```

Ressource Système

Ressource Système

Chaque processus possède une table des descripteurs ouverts

- A chaque fois que l'on ouvre un fichier en lecture/écriture, on utilise une case de la table
- La table possède une taille
  (on peut la changer avec ulimit sous Unix)
- Si la table est pleine, on reçoit une IOException

Si on supprime pas le descripteur de fichier une fois que l'on a fini d'utiliser le fichier, la table va finir pas être pleine :(

Resources Systèmes en GC

Le GC sait lorsqu'un objet n'est plus accessible

- Idée: utiliser le GC pour libérer

Pas une bonne idée

- On ne contrôle pas quand le GC se déclenche
  - On peut quand même avoir la table des descritpeurs pleines avant que le GC n'intervienne

On doit gérer la libération des descripteurs de fichier à la main :(

try/finally

Java possède une syntaxe exprès pour cela

- Le try/finally

```
Path path = ...
var input = Files.newInputStream(path);
try {
  input.transferTo(System.out);
} finally {
  input.close();
}
```

Le code de finally est appelé et si le code du try fini normalement et si une exception est lévée

L'appel à **close**() permet de libérer la case dans la table des descripteurs de fichier

Le bug du try/finally

En fait, on utilise pas un try/finally car il y a un bug génant

```
try {
  throw new IOException("1");
} finally {
  throw new IOException("2");
}
```

renvoie IOException("2") et on perd les informations liées à la première exception

Pas un problème théorique car close() renvoie une IOException :(

Le try-with-ressources

Aussi appelée le try parenthèse

```
Path path = ...
try(var input = Files.newInputStream(path)) {
  input.transferTo,(System.out);
} // calls close() implicitly here
```

- La ressource doit être java.lang.AutoCloseable
  - super interface de java.io.Closeable
- appel la méthode close() implicitement à la fin du bloc
- si close lève une exception, est est ajouté en tant que “suppress exception” de l’exception initiale

Avec plusieurs ressources

Si on a plusieurs ressources, il faut appeler les closes dans l'ordre inverse des assignations

```
Path inputPath = …
Path outputPath = ...
try(var input = Files.newInputStream(inputPath);
    var output = Files.newOutputStream(OutputPath)) {
  input.transferTo(output);
}
// appel output.close() puis input.close()
```

Les instructions d'initialisation des ressources à l'intérieur du try sont séparés des '**;**'

Le try() et les Buffered*

Attention, si on utilise des Buffered*, il faut utiliser plusieurs lignes du try

Code faux:

```
Path path = ...
try(var buffered = new BufferedInputStream(
                              Files.newInputStream(path)) {
  ...
}
```

Si le constructeur de Buffered* lève une exception, le descripteur stocké dans l'InputStream du fichier est pas libéré !!

Code bon:

```
Path path = ...
try(var input = Files.newInputStream(path);
    var buffered = new BufferedInputStream(input)) {
  ...
}
```

java.util.Stream

Les Stream correspondant à des ressources système doivent aussi être fermé !

```
Path path = ...
try(Stream<Path> stream = Files.list(path)) {
  stream.forEach(System.out::println);
}
```

Attention, le Stream n’est plus valide après le try, donc pas de return du Stream dans le try

En terme de programmation

Si on stocke un champ Closeable, alors la classe doit aussi être Closeable

- La méthode close() doit déléguer l'appel à close() sur le champ

On prèfère les méthodes (statiques) qui alloue et ferme la resource dans la même méthode

- on peut allouer l'InputStream au début du try-with-ressources
- L'appel à close() est fait implicitement à la fin

Encodage et Charset

Encodage

En Java, les caractères sont manipulés en unicode (char est sur 16 bits)

Les fichiers peuvent être dans un autre format

- En ASCII (8bits sans accent)
- En ISOLatin 1 (8 bits avec accent)
- En Windows Latin 1 (microsoft pas à la norme ISO)

  etc.

java.nio.Charset

La classe java.nio.Charset définie la table de conversions de byte vers char et vice versa.

Possède deux méthodes principales :

- CharsetDecoder newDecoder()

  byte → char
- CharsetEncoder newEncoder()

  char → byte

Charset.forName(String name) permet d'obtenir un charset à partir de son nom

java.nio.charset.StandartCharsets

L’enum java.nio.charset.StandartCharsets définie l’ensemble des Charsets par défaut

US-ASCII

- seven-bit ASCII, a.k.a. ISO646-US

ISO-8859-1

- ISO Latin Alphabet No. 1, a.k.a. ISO-LATIN-1

UTF-8 (8/16/24 bits)

- eight-bit UCS Transformation Format

UTF-16BE Sixteen-bit UCS Transformation Format,

- big-endian byte order

UTF-16LE Sixteen-bit UCS Transformation Format,

- little-endian byte order

UTF-16

- sixteen-bit UCS Transformation Format, byte order identified by an optional byte-order mark

String

Convertir **en mémoire** un tableau de byte[] en String suivant un certain codage

En utilisant un décodeur

- String(byte[] bytes, Charset charset)
- String(byte[] bytes, int offset, int length, Charset charset)

En utilisant un encodeur

- string.getBytes(Charset charset)

Reader

Flux de caractères en entrée

Lit un char et renvoie celui-ci ou -1 si fin de flux

- int **read**()
  A ne pas utiliser, trop lent

Lit un tableau de char

- int **read**(char[] buffer)
- int **read**(char[] buffer, int offset, int length)

Saute un nombre de caractères

- long **skip**(long n);

Ferme de flux

- void **close**();

BufferedReader

Hérite de Reader

Ajoute

- Lire une ligne sans le séparateur de ligne (ou renvoie null)
  - String **readLine**()
- Renvoie un Stream de toutes les lignes
  - Stream<String> **lines**()

Exemple

Pour compter le nombre d'occurence de chaque mot, on peut décomposer par ligne puis par mot

```
public static Map<String, Long> wordCount(BufferedReader reader) {
  return reader.lines()
    .flatMap(line -> Arrays.stream(line.split(" ")))
    .collect(Collectors.groupingBy(w -> w, Collectors.counting()));
}

public static void main(String[] args) throws IOException {
  var path = Paths.get(args[0]);
  try(var reader = Files.newBufferedReader(path)) {
    System.out.println(wordCount(reader));
  }
}
```

LineNumberReader

Hérite de BufferedReader

- donc on peut utiliser readLine()

maintient un numéro de ligne ce qui permet par exemple de reporter un nunéro de ligne en cas d’erreur

Writer

Flux de caractère en sortie

Ecrire un caractère, un int pour qu'il marche avec le read

- void **write**(int c)
  - Pas efficace !

Ecrire un tableau de caractère

- void **write**(char[] b)
- void **write**(char[] b, int off, int len)

Ecrire une chaine de caractères

- void **write**(String s)

Demande d'écrire ce qu'il y a dans le buffer

- void **flush**()

Ferme le flux

- void **close**()

Pont binaire → textuelle

Créer un Writer/Reader à partir d'un InputStream/OutputStream

- new InputStreamReader(
  InputStream inputstream, Charset charset)
- new OutpuStreamWriter(
  OutputStream outputstream, Charset charset)

Console

Entrée clavier/sortie console

Les constantes :

Entrée standard

System.in est un InputStream (pas bufferisé)

Sortie standard

System.out est un PrintStream

un OutputStream avec print()/println() en plus

Sortie d'erreur standard

System.err est un PrintStream

java.io.Console

System.console() permet d’accèder à la console

- Attention peut renvoyer null si le processus n’est pas attaché à une console (un tty)

  Lecture :

  - Reader **reader**()
  - String **readLine**(String fmt,Object... args)
  - char[] **readPassword**(String fmt,Object... args)

  Ecriture :

  - Writer **writer**()
  - Console **printf**(String format, Object... args)
  - Console **flush**()

Exemple

Copier l'entrée standard ou un fichier sur la sortie sandard (comme cat)

```
public static void main(String[] args)
  throws IOException {

  var console = System.console();
  if (console == null) {
    throw new IOException("no console available");
  }
  var input = (args.length == 0) ?
    console.reader():
    Files.newReader(Paths.get(args[0]));
  try(var reader = input;
      var writer = console.writer()) {
    reader.transferTo(writer);
  }
}
```

Entrée standard et console

Si on lit sur la console, la console ne transmet les bytes au programme que lorsque l'on appuis sur return

Un programme ne peut pas lire sur la console, et réagir à chaque caractère tapé

Il faut passer la console en mode canonique

Accès directe aux données d’un fichier

java.nio.channel.FileChannel

Création

FileChannel.open(Path, OpenOption…)

- CREATE (crée e fichier si nécessaire)
- CREATE_NEW (marche pas si un fichier existe déjà)
- APPEND (écrit à la fin)
- TRUCATE_EXISTING (on commence à zéro)
- DELETE_ON_CLOSE (le fichier disparait après close)
- SPARSE (ne doit pas allouer la place si possible)
- SYNC (synchro à chaque read/write)
- DSYNC (synchro avec metadata à chaque read/write)

Utilisation d’un FileChannel

Lecture bloquante
- **read**(ByteBuffer) ou **read**(ByteBuffer, position)

Ecriture bloquante
- **write**(ByteBuffer) ou **write**(ByteBuffer, position)

Demander la synchro sur le disque
- **force**(boolean metadata)

Un verrou sur le fichier
- **lock**(), **lock**(position, size, shared), **tryLock**()

Fermer le fichier
- **close**()

Fichier mappé en mémoire

Permet de voir le contenu d’un fichier comme de la mémoire

- MappedByteBuffer map(mode, position, size)
  - Mode: READ_ONLY, READ_WRITE, PRIVATE

Le système se débrouille pour prendre les parties de fichier qui vont bien et les montés en mémoire quand il faut

AsynchronousFileChannel

Même opération qu'un FileChannel

- open(), close(), force(), lock()

Les read et write sont asynchrones

- Future<Integer> read/write(ByteBuffer, position)
  - Le future permet d'obtenir le résultat ultérieurement
- void read/write(ByteBuffer, position, attachment, completionHandler)
  - Le CompletionHandler est appelé lorsque le système a réèllement effectué l'opération
  - L'attachement est un objet qui sera passé au completionHandler

CompletionHandler

CompletionHandler<V, A>

- V: Le type de la valeur de retour (ici, un Integer)
- A: Le type de l'attachement

Deux méthodes

- Appelée si l'appel à réussi
  - void completed(V result, A attachment);
- Appelé si l'appel à raté
  - void failed(Throwable t, A attachment);

Serialization

Serialization

Permet de sauver/charger un objet Java dans un flux binaire

- Object ObjectInputStream.**readObject**()
- ObjectOutputStream.**writeObject**(Object o)

Pour sérializer une objet, il faut que sa classe soit d'accord.

- Elle doit implanter java.io.**Serializable** (qui est vide)

Serialization par défaut

Lorsque l’on sauve un object, les champs de celui-ci sont sauvergardés récursivement (avec détection des cycles)

- Donc on sauvegarde un graphe d’objets

Seuls les champs des classes/superclasses Serializable sont sauvegardés

Si un champs est déclaré **transient**, il n’est pas sauvegardé

Deserialization par défaut

Le constructeur sans paramètre de la première superclasse non sérializable est appelé

Puis pour les classes Serializable

- Les champs non transient sont initialisés avec les valeurs dans l'ObjectInputStream
- Les champs transients sont initialisés avec leur valeur par défaut (0, false, null, etc)

comme c'est la VM qui fait les initialisations, les champs final ne posent pas de problème

Exemple

```
public class A {
  private final int a;
  public A() { a = 3; }
}

public class B extends A implements Serializable {
  private transient int b = 8;
  private final int c;
  public B(int c) { this.c = c; }
}

B b = new B(5);                  // a = 3, b = 8, c = 5
out.writeObject(b);              // sauvegarde c = 5

B b = (B) in.readObject();  // a = 3, b = 0, c = 5
```

Prendre la main sur la Serialization

On peut définir précisément le format de serialization en écrivant les méthodes

- private void **writeObject**(ObjectOutputStream out)
  - Dans la méthode, out.**defaultWriteObject** demande la serialization par défaut
- private void **readObject**(ObjectInputStream in)
  - Dans la méthode, in.**defaultReadObject**() demande la deserialization par défaut

Ce n’est pas une interface car on veut que les méthodes soient privée (appel par réflection)

Lire/Ecrire des valeurs primitives

ObjectInputStream

- Lire un type primitif
  - **readByte**(), **readInt**(), **readLong**(), etc
- Lire une String en UTF8 (modified)
  - **readUTF**()

ObjectOutputStream

- Ecrire un type primitif
  - **writeByte**(byte), **writeInt**(int), **writeLong**(long), etc
- Ecrire une String en UTF8 (modified)
  - **writeUTF**(String)

Version de classe

Par défault, le mécanisme de serialization assigne une valeur de version en calculant un MD5 sur la classe (champs, héritage, etc)

Il est possible de définir sa propre valeur en indiquant le serialVersionUID

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

Cela permet dans readObject() de faire la différence entre les versions

Serialization Proxy

Il est possible d'indiquer que l'on veut serializer une classe comme une autre

- On renvoie le Serialization proxy

  private Object **writeReplace**()
  throws ObjectStreamException;

- Sur la classe proxy, on implante

  private Object **readResolve**()
  throws ObjectStreamException;

  pour indiquer comment à partir du serilization proxy on retrouve la classe d'origine

Legacy

java.io.File

Ancienne version des java.nio.file.Path, mélange la notion de fichier et de chemin

A **ne pas utiliser** dans du nouveau code

- Le nom des fichiers est décoder en essayant de deviner le Charset
  - Dès fois, ça rate
- Problème de performance lorsque l'on demande l'accès aux droits
- Certaine méthodes (par ex, File.delete()) renvoie false au lieu de lever une IOException

Path.toFile() permet de discuter avec du code legacy mais l'appel peut ne pas marcher !

Les classes File*Stream

Ces classes ont été remplacés par des implantations dans java.nio.file.Files

- FileInputStream, FileOutputStream
  - getChannel() permet d'obtenir le channel correspondant
  - Remplacer par: Files.newInputStream()/newOutputStream()
- FileReader, FileWriter
  - Remplacer par: Files.newBufferedReader()/newBufferedWriter()

RandomAccesFile

Ancienne version, replacé par FileChannel

new RandomAccessFile(String mode)

- "r" lecture seule ;
- "rw" lecture et écriture, avec création du fichier ;
- "rws" comme "rw", mais chaque écriture est sync ;
- "rwd" comme "rws", mais chaque écriture ou changement de metadata est sync

getChannel() permet de récupérer le FileChannel correspondant