

L3. Programmation orientée objet. Cours 6

Marie-Pierre Béal (Cours de Rémi Forax)

Entrées - sorties



Entrées / Sorties

Entrées / Sorties

Chemin et fichier

Chemin et fichier

java.nio.file.Path

La classe `Path` représente un chemin dans l'arborescence

- Abstrait le séparateur de fichier (WORA Write Once Run Anywhere)
- Marche pour les fichiers sur le disque ou dans des zip/jar

`Path.of(repertoire, fichier)`

ou

`Path.of(URI)`

permet de créer un chemin.

Ne pas utiliser des Strings pour représenter des chemins !

Différentes sortes de chemins

Il y a trois sortes de chemins

- Relatif au répertoire courant : `./foo`, `../foo`
- Absolu (commence par un `'/'`) : `/foo`, `/home/foo`
- Canonique (unique) : `/home/foo`

Un chemin canonique demande au FileSystem d'avoir un chemin unique

- Attention, c'est une opération super lente

`resolve()` `/relativize()` `getFileName()`

`path.resolve(filename)` permet d'ajouter un nom de fichier à un chemin (utilise `'/'` ou `'\'`)

```
Path.of("foo").resolve("bar")    // Path foo/bar
```

`path.relative(path2)` extrait la partie spécifique de `path2` dans `path`

```
Path.of("foo").relative(Path.of("foo", "bar")) // Path bar
```

`path.getFileName()` renvoie un `Path` contenant le nom de fichier après le dernier slash

```
Path.of("foo", "bar").getFileName() // Path bar
```

java.nio.file.Files

Files contient plein de méthodes static pour gérer des fichiers (utilise UTF8 par défaut)

- lecteur/écrivain

- `newBufferedReader(path)`, `newBufferedWriter(path)`,
`newInputStream(path)`, `newOutputStream(path)`

- méthodes pratiques

- `readAllLines(path)`, `readAllBytes(path)`,
`readString(path)`, `write(path, lines)`,
`writeString(path, text)`

- créer, copier ou détruire

- `createDirectory(path)`, `createDirectories(path)`,
`createSymbolicLink(path, toPath)`,
`createTempFile(prefix, suffix)`, `copy(path, toPath)`,
`delete(path)`, `deleteIfExists(path)`

java.io.File (ne plus utiliser)

Ancienne version de java.nio.file.Path

Bugs d'encodage (comme Python 2)

- `file.listFiles()` essaye de deviner l'encodage des noms des fichiers et renvoie null si n'a pas les droits
- `FileInputStream/FileOutputStream/FileReader/FileWriter` essayent de deviner l'encodage du contenu du fichier

Date d'avant que UTF8 ne gagne !

- à ne plus utiliser

Checked Exception

Checked Exception

IOException

Les méthodes de `java.nio.file.Files` lèvent l'exception `IOException` s'il y a un problème

- disque plein, pas le droit en écriture/lecture, etc

En Java, il y a deux sortes d'exception

- Les erreurs de programmation (`NullPointerException`, `IllegalArgumentException`, `IllegalStateException` et `AssertionError`)
 - Appelées runtime exceptions : il faut les laisser planter le programme
- Les erreurs dues à des conditions extérieures, comme `IOException`. Le compilateur vous oblige à faire quelque chose.
 - Appelées checked exceptions : on les propage jusqu'au `main()` et on met un message à l'utilisateur

Exemple

Lire un fichier de configuration

```
public record Conf() {  
    public static Conf load(Path path) {  
        var reader = Files.newBufferedReader(path);  
        // ne compile pas  
        ...  
    }  
}  
  
void main() {  
    var userDir = System.getProperty("user.dir");  
    var confPath = Path.of(userDir).resolve("conf");  
    var conf = Conf.load(confPath);  
    ...  
}
```

Utiliser throws

Throws demande à l'appelant de gérer l'exception checked

```
public record Conf() {  
    public static Conf load(Path path) throws IOException {  
        // OK  
        var reader = Files.newBufferedReader(path);  
        ...  
    }  
}  
  
void main() {  
    var userDir = System.getProperty("user.dir");  
    var confPath = Path.of(userDir).resolve("conf");  
    var conf = Conf.load(confPath); // ne compile pas  
    ...  
}
```

Permet de faire descendre/propager toutes les exceptions à un même endroit.

Reprendre sur l'erreur avec try/catch

On utilise try/catch pour reprendre sur l'erreur

```
public record Conf() {  
    public static Conf load(Path path) throws IOException {  
        ...  
    }  
}  
  
void main() {  
    var userDir = System.getProperty("user.dir");  
    var confPath = Path.of(userDir).resolve("conf");  
    Conf conf;  
    try {  
        conf = Conf.load(confPath);  
    } catch(IOException e) {  
        System.err.println(e.getMessage());  
        // on reporte l'erreur à l'utilisateur (le vrai message)  
        System.exit(1); // on indique au shell que cela c'est mal passé (pas 0)  
        return; // on arrête le main() sinon le compilateur va dire  
    } // que la variable conf n'est pas initialisée par  
        // tous les chemins  
    ...  
}
```

Exceptions et IDEs

Le problème principal des exceptions en Java est que les IDEs ne savent pas gérer correctement les exceptions.

Voilà le code généré par un IDE

```
public static void main(String[] args) {  
    var userDir = System.getProperty("user.dir");  
    var confPath = Path.of(userDir).resolve("conf");  
    Conf conf; // conf n'est pas initialisée car il manque  
               // le return dans le catch  
    try {  
        conf = Conf.load(confPath);  
    } catch(IOException e) {  
        e.printStackTrace();  
        // printStackTrace() affiche le stacktrace  
        // à l'utilisateur, comme si cela l'intéressait ??  
    }  
    ...  
}
```



Autre bug fréquent

Ne jamais faire un `catch(Exception e)`

Car cela attrape **toutes** les exceptions

- les checked comme `IOException`
- et les runtime comme `NullPointerException`,
`IllegalArgumentException`, `IllegalStateException`

Les **checked** doivent être attrapées dans le `main()` mais les **runtime** doivent faire planter le programme !

Ressources système

Ressources système

Ressources systèmes

Ouvrir un fichier en lecture ou écriture consomme une ressource système, un descripteur de fichier

- Après utilisation, il faut libérer la ressource sinon la table va être pleine et on ne pourra plus ouvrir de nouveau fichier
- Comme en C, il faut appeler la méthode `close()` pour libérer la ressource

Ressources systèmes et exceptions

Il y a deux façons de sortir d'une méthode

- Soit par un `return`, et donc il faut mettre le `close()` avant !
- Soit par une exception, et là il faut être sûr que le `close()` se fait quand même

La structure de contrôle `try-with-resources` résout ce problème

Try-with-resources

Le `try()` (try parenthèses) appelle `close()` sur toutes les variables initialisées dans les parenthèses

```
try(var reader = .. ;    // <- ';' si plusieurs ressources
    var writer = ...) {
    ...
} // appelle writer.close() puis reader.close()
```

Les appels à `close()` sont fait même en cas d'exceptions (youpi!)

Try-with-resources vs try/catch

Les deux syntaxes commencent par try mais elles ont des rôles complètement différents. Ne les confondez pas.

- Libérer des ressources système automatiquement

```
try(var resource = ...) {  
    ...  
} // appelle resource.close() automatiquement
```

- Reprendre sur une exception particulière

```
try {  
    ...  
} catch(IOException e) {  
    // reprend sur l'erreur  
}
```

Binaire ou texte

Binaire ou texte

BufferedReader

Permet de lire un fichier texte

- Buffered veut dire que l'implantation utilise un buffer intermédiaire et ne lit pas caractère par caractère
- À la création, il faut spécifier un encodage (Charset)

```
Files.newBufferedReader(path,  
    /* encoding = */ StandardCharsets.ISO_LATIN1);
```

sinon UTF8 est utilisé

```
Files.newBufferedReader(path);
```

BufferedReader.readLine()

reader.readLine() lit ligne à ligne

- renvoie null si il n'y a plus de ligne
- chaque ligne ne contient pas \n ou \r\n à la fin (WORA)

```
public static void printAllLines(Path path) throws IOException {  
    try(var reader = Files.newBufferedReader(path)) {  
        String line;  
        while ((line = reader.readLine()) != null) {  
            IO.println(line);  
        }  
    } // appelle reader.close()  
}
```

BufferedWriter

Permet d'écrire dans un fichier texte

- Buffered veut dire que l'implantation utilise un buffer intermédiaire et n'écrit pas caractère par caractère
- A la création, il faut spécifier un encodage (Charset)

```
Files.newBufferedWriter(path,  
    /* encoding = */ StandardCharsets.ISO_LATIN1);
```

- sinon UTF8 est utilisé

```
Files.newBufferedWriter(path);
```

BufferedWriter.write(String)

Écrit une chaîne de caractères dans le fichier

- Elle est encodée avec l'encodage spécifié à la création
- Elle est écrite dans un buffer qui est écrit dans le fichier quand le buffer est plein ou lors de l'appel à `close()`

```
public static void writeAllLines(Path path, List<String> lines)
    throws IOException {
    try(var writer = Files.newBufferedWriter(path)) {
        for(var line: lines) {
            writer.write(line);
            writer.newLine(); // \n ou \r\n (WORA)
        }
    } // appelle writer.close()
}
```

InputStream / OutputStream

InputStream /
OutputStream

InputStream / OutputStream

InputStream permet de lire un tableau de bytes

- `read(byte[] buffer)`
 - Bloquant jusqu'à ce qu'au moins un octet puisse être lu
 - Remplit le buffer et renvoie le nombre d'octets lus (pas forcément autant que la taille du buffer)
 - renvoie -1 si on est à la fin du fichier

OutputStream permet d'écrire un tableau de bytes

- `write(byte[] buffer, offset, length)`
 - Bloque tant que les "length" octets ne sont pas écrits

Copier un fichier

Attention, le kernel peut lire moins d'octets que la taille du buffer, donc il faut récupérer la valeur de retour de `InputStream.read()`

```
public static void copy(Path from, Path to) throws IOException {  
    try(var input = Files.newInputStream(from);  
        var output = Files.newOutputStream(to)) {  
        var buffer = new byte[8192];  
        int read;  
        while ((read = input.read(buffer)) != -1) {  
            output.write(buffer, 0, read);  
        }  
    } // appelle output.close() puis input.close()  
}
```

Copier un fichier avec readAllBytes()

On peut lire tous les octets d'un coup avec `input.readAllBytes()` mais les tableaux sont limités à 2G en Java, donc on risque le `OutOfMemoryError` pour les gros fichiers.

```
public static void copy(Path from, Path to) throws IOException {  
    try(var input = Files.newInputStream(from);  
        var output = Files.newOutputStream(to)) {  
        var buffer = input.readAllBytes(); // lit tout d'un coup  
        output.write(buffer, 0, buffer.length);  
    } // appelle output.close() puis input.close()  
}
```

Copier un fichier avec transferTo()

En fait, la copie est déjà implantée en utilisant
`input.transferTo(output)`

```
public static void copy(Path from, Path to) throws IOException {  
    try(var input = Files.newInputStream(from);  
        var output = Files.newOutputStream(to)) {  
        input.transferTo(output);  
    } // appelle output.close() puis input.close()  
}
```

Console

Console

PrintStream

Un `PrintStream` est un `OutputStream` avec des méthodes `println`

- `println = write(String) + newLine()`

`System.out` et `System.err` sont des `PrintStream`

`System.in` est `InputStream`

java.io.Console

Permet d'accéder à la console si elle existe

- `System.console()` renvoie un objet `Console` ou `null`

Console

- `charset()` : l'encodage de la console (pas forcément UTF8)
- `reader()/writer()` : pour lire/écrire sur la console
- `readLine()` : marche comme `BufferedReader.readLine()`
- `readPassword()` : lit un mot de passe sans qu'il soit visible

En résumé

En résumé

Les entrées / sorties

`java.nio.file.Path` représente un chemin dans l'arborescence

`java.nio.file.Files` possède plein de méthodes static pour manipuler les fichiers

`BufferedReader/BufferedWriter` est un lecteur/écrivain bufferisé de `String`

Si on ouvre un fichier, il faut fermer l'objet en appelant `close()`

- Utiliser pour cela un `try()` (try-with-resources)