Programmation orientée objet. Cours 7

Marie-Pierre Béal et Carine Pivoteau
BUT 1

Entrées/sorties.

Fichiers avec `java.nio`

Dans `java.nio`

- `java.nio.file.Path` : interface représentant une abstraction d'un chemin vers un élément du système de fichiers (fichier, répertoire, lien symbolique, etc).
Et depuis Java 11 : méthodes *factory* pour construire un `Path`.
- `java.nio.file.Files` : contient des méthodes statiques pour manipuler des éléments du système de fichiers.

L'objet Path

- Path est un chemin dans l'arborescence du système de fichiers manipulé par le programme.
- Pour créer un chemin :
 - `Path.of(fichier)`
 - `Path.of(repertoire, fichier)`
 - `Path.of(URI)`
- Ne pas utiliser des `String` pour représenter des chemins.

L'objet Path

- Il existe 3 types de chemins :
 - Relatif au répertoire courant : `../foo`
 - Absolu (commence par '/') : `/home/foo`,
`/home/../home/foo`
 - Canonique (unique) : `/home/foo`
- Les séparateurs sont gérés automatiquement, par contre, il n'y a rien pour gérer les extensions.

Création et utilisation d'un Path

```
import java.io.IOException;
import java.nio.file.Path;

public class PathTest {
    public static void main(String[] args) throws IOException {
        var path1 = Path.of("/Users/beal/Java/BUT2025/workspaceBUT2025/cours7");
        System.out.println(path1);
        System.out.println(path1.getFileName());
        System.out.println(path1.getParent());
        // Attention, ne compile pas, car aucun des deux n'est une String
        // System.out.println(path1.getParent() + path1.getFileName());
        var path2 = Path.of("../.");
        System.out.println("toAbsolutePath() : " + path2.toAbsolutePath());
        System.out.println("toRealPath() : " + path2.toRealPath()); //canonique
    }
}
```

/Users/beal/Java/BUT2025/workspaceBUT2025/cours7

cours7

Users/beal/Java/BUT2025/workspaceBUT2025

toAbsolutePath() : /Users/beal/Java/BUT2025/workspaceBUT2025/cours7/../../

toRealPath() : /Users/beal/Java/BUT2025/workspaceBUT2025

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)`, `createSymbolicLink(path, toPath)`,
`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

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

Rappel : 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  
        ...  
    }  
}  
  
public static void main(String[] args) {  
    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);  
        ...  
    }  
}  
  
public static void main(String[] args) {  
    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 {  
        ...  
    }  
}  
  
public static void main(String[] args) {  
    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  
  
    ...  
}
```

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 !

Ressource système

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

Ressource système 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 !

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) {  
            System.out.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()
}
```

Lire sur l'entrée standard avec Scanner

Attention, on doit **toujours tester l'existence** d'un prochain élément à lire. Par exemple si on lit un entier, on doit appeler la méthode `hasNextInt()` avant de pouvoir appeler `nextInt()`.

Exemple : lire un entier puis un mot

```
public static void main(String[] args) {
    try (var scanner = new Scanner(System.in)) {
        if (scanner.hasNextInt()) {
            var i = scanner.nextInt();
            System.out.println(i);
        }
        if (scanner.hasNext()) {
            var s = scanner.next();
            System.out.println(s);
        }
    }
}
```

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 petit fichier avec readAllBytes()

On peut lire tous les octets d'un coup avec `input.readAll()` 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()
}
```

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`

Bonus : flots d'objets

Flots d'objets ou sérialisation

- Un **flot d'objets** permet d'écrire ou de lire des objets Java dans un flot. Ce service est appelé **sérialisation**.
- On utilise `ObjectInputStream` et `ObjectOutputStream` (créés à partir d'`InputStream` et d'`OutputStream`).
- Les applications qui échangent des objets via le réseau utilisent la sérialisation.
- Pour sérialiser un objet, on utilise la méthode `writeObject` d'un flot implémentant l'interface `ObjectOutput`.
- Pour désérialiser un objet, on utilise la méthode `readObject` d'un flot implémentant l'interface `ObjectInput`.
- Pour qu'un objet puisse être inséré dans un flot, sa classe doit implémenter l'interface `Serializable`. Cette interface ne contient pas de méthode.

Example

```
import java.io.Serializable;
public record Pixel(int x, int y) implements Serializable {
    @Override
    public String toString(){
        return "(" + x + "," + y + ")";
    }
}
```

Exemple

```
public static void main(String[] args)
    throws IOException, ClassNotFoundException {
    var p1 = new Pixel(1, 2);
    var path = Path.of("backup");
    try (var back = Files.newOutputStream(path);
        var out = new ObjectOutputStream(back)){
        out.writeObject(p1); // sauvegarde
    }
    Pixel p2;
    try (var back = Files.newInputStream(path);
        var in = new ObjectInputStream(back)){
        p2 = (Pixel) in.readObject(); // récupération
    }
    System.out.println(p1); // (1,2)
    System.out.println(p2); // (1,2)
    System.out.println(p1 == p2); // false
    System.out.println(p1.equals(p2)); // true
}
```

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`)