IN201 : 8 – Exceptions

Author : Christophe Garion <garion@isae.fr>
Public : SUPAERO 2A
Date :

Résumé

Le but de ce TP est de manipuler des exceptions prédéfinies et de créer ses propres exceptions. On utilisera des fichiers au format XML.

1 Objectifs

Les objectifs de ce TP sont les suivants :
- utiliser des méthodes pouvant lever des exceptions ;
- créer ses propres exceptions ;
- spécifier des exceptions ;
- utiliser l'API JAXP permettant de manipuler des fichiers XML.

2 Problématique

On souhaite récupérer des données provenant d'expériences d'aérodynamique dans lesquelles on relève des pressions. Ces données sont donc des réels qui doivent normalement être positifs. Les fichiers seront des fichiers au format XML [3]. XML est un langage de marquage qui permet d'ajouter du contenu sémantique à un fichier contenant du texte. Il est facilement lisible par un humain et par une machine, ce qui en fait son intérêt.
Un fichier XML représente un *arbre* contenant des *éléments* délimités par des *balises* auxquels on peut ajouter des *attributs*. Les fichiers de stockage des expériences auront le format suivant :

```
<experience resp-name="Allan Bonnet">
  <pression>25.754</pression>
  <pression>10.432</pression>
  <pression>30.6754</pression>
  <pression>42.543</pression>
  <pression>100.678</pression>
</experience>
```

Dans ces fichiers :
- l'élément représentant l'ensemble des relevés est délimité par la balise `experience`. On remarquera que le début d'un élément `balise` est représenté par une balise ouvrante `<balise>` et la fin de ce même élément par une balise fermante `</balise>`.
- la balise `experience` possède un attribut dont la clé est `resp-name` et dont la valeur représente le nom du responsable de l'expérience. Les valeurs des attributs ne peuvent être que des chaînes de caractères délimitées par `""` (ou `''`). Cet attribut peut ne pas être présent, on suppose dans ce cas que le responsable de l'expérience est inconnu.
- chaque relevé de pression est délimité par la balise `pression`.

Les données contenues dans les balises ne pourront être récupérées que comme des chaînes de caractères depuis Java.

On va donc créer une classe qui devra offrir les services suivants :
- lecture du fichier de données et vérification de la cohérence des données ;
- stockage des données ;
- accès au nom du responsable de l'expérience ;
- renvoi d'un itérateur sur l'ensemble de données.

Lorsqu'une erreur dans les champs représentant une valeur numérique du fichier est présente, on souhaite lever une exception et permettre à l'utilisateur de récupérer les chaînes de caractères des champs posant problème.

3 Conception de la classe `Acquisition`

La classe `Acquisition` est la classe qui va nous permettre de récupérer, puis de stocker les valeurs contenues dans le fichier.

1. quels est(sont) le(s) attribut(s) de la classe `Acquisition` ?
2. quelles sont les méthodes de la classe `Acquisition` ?
4. construire le diagramme UML de la classe.

4 Implantation de la classe `Acquisition`

Pour réaliser le TP, vous allez devoir créer un projet Java sous Eclipse en utilisant votre dépôt Subversion. Si vous avez configuré votre dépôt pour qu'il soit disponible dans la vue *SVN Repository Exploring* d'Eclipse, vous créerez votre projet en faisant un *checkout* **à partir du répertoire TP8** de votre dépôt en cliquant droit dessus. N'oubliez pas de configurer votre *build path* pour pouvoir utiliser les éventuelles archives JAR placées dans le répertoire `lib`.

Il faut maintenant implanter la classe `Acquisition`.

1. choisir un attribut permettant de stocker les valeurs ;
2. pour lire les données contenues dans le fichier, utiliser la documentation de l'API JAXP présentée dans la section 5. Utiliser également le programme présenté durant le cours pour comprendre comment transformer une chaîne de caractères en nombre. On pourra également regarder la section 6 qui présente de façon générale la gestion des entrées/sorties en Java, même si on ne s'en sert pas directement dans le TP ;
3. construire les exceptions nécessaires ;
4. les exceptions levées dans les méthodes doivent-elle être rattrapées et traiter dans la classe `Acquisition` ou propagées ?
   Voici un bon exemple de traitement local des exception : je traite localement les exceptions de type `NumberFormatException` pour pouvoir remplir la liste d'erreurs.
5. utiliser les fichiers proposés dans le dépôt pour vérifier que la classe est opérationnelle (ajouter le répertoire `data` contenant les fichiers au CLASSPATH). Réfléchir également à l'écriture d'une classe de test JUnit.
6. (facultatif) comment vérifier que le fichier XML a le bon format ?

5 L'API JAXP (*Java API for XML Processing*)

Java fournit une API permettant de travailler avec des fichiers XML de façon unifiée[1], JAXP. Si vous en avez besoin, vous trouverez un bon tutoriel sur JAXP sur le site d'Oracle [2]. Pour utiliser les classes présentées dans ce qui suit, il va falloir lire leur documentation javadoc qui est disponible ici [1].
Nous allons ici reconstruire l'arbre représenté par le fichier XML entièrement en mémoire en utilisant ce que l'on appelle un *parser* DOM (*Document Object Model*, i.e. un parser qui reconstruit l'arbre XML entièrement en mémoire). Pour cela, deux classes appartenant au paquetage `javax.xml.parsers` interviennent : `DocumentBuilder` et `DocumentBuilderFactory`. Ces deux classes vont nous permettre de construire un objet de type `org.w3c,dom.Document` qui représentera notre document XML. Un diagramme de séquence montrant l'utilisant de ces classes est présenté sur la figure 2.
Attention, dans ce diagramme on ne précise pas quelles sont les exceptions qui peuvent être levées par les appels des méthodes. On remarquera également que la méthode `newInstance` est statique. C'est une *factory method*, i.e. une méthode permettant de construire des objets (cf. transparents du premier cours sur classes et objets).
L'objet de type `org.w3c.dom.Document` ainsi récupéré représente le document XML contenu dans le fichier. Dans la suite de cette section, toutes les classes appartiennent au paquetage `org.w3c.dom`.

1. Unifiée s'entend en particulier ici pour ce qui concerne les *parsers*, i.e. les outils permettant de reconstruire l'arbre décrit par le fichier XML.

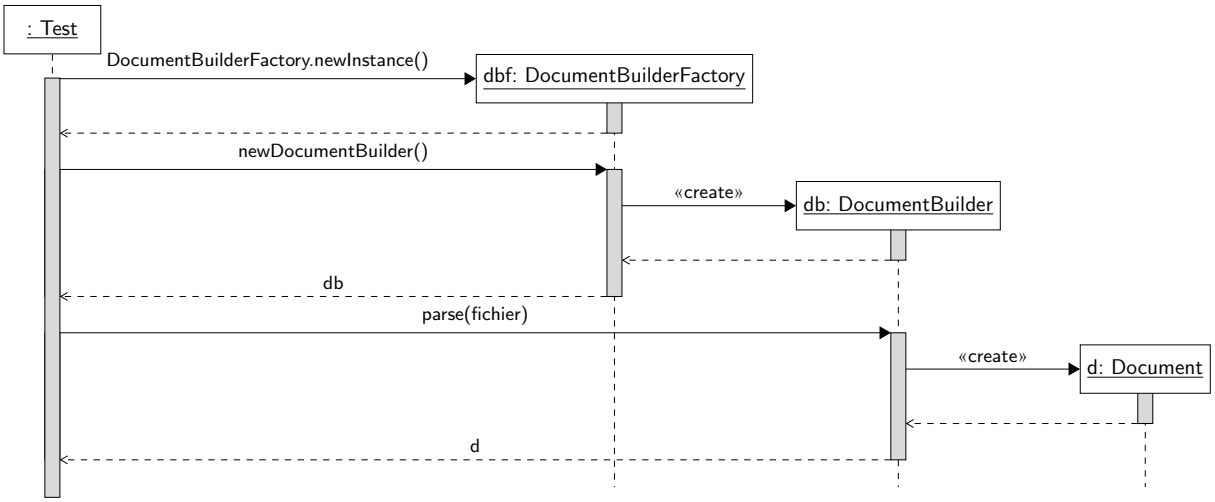

FIGURE 2 – Diagramme de séquence illustrant la création d'un objet modélisant un document XML

Les noeuds du document XML sont représentés par l'interface `Node`. Attention, il ne s'agit pas forcément d'éléments, il y a d'autres types de nœuds (dont nous n'avons pas besoin ici). Pour connaître le type d'un nœud, on peut utiliser la méthode `getNodeType` de `Node` et comparer son retour avec les constantes `Node.ELEMENT_NODE` etc. contenues dans l'interface `Node`. La méthode `getDocumentElement()` appelée sur un document permet de récupérer l'élément de plus haut niveau du document.
La méthode `getTextContent` de `Node` permet de récupérer le texte contenu dans un nœud et les nœuds qu'il contient (attention !). La méthode `getChildNodes` permet de récupérer tous les nœuds fils du nœud courant sous la forme d'une instance de `NodeList`.
L'interface `Element` représente les éléments XML et possède par exemple une méthode `getTagName` pour récupérer le nom d'une balise etc. Elle possède également une méthode `getAttribute(String name)` qui permet de récupérer la valeur de l'attribut `name` de l'élément.
Pour notre problème particulier, l'algorithme d'acquisition des données pourra être celui représenté sur l'algorithme 5.1.

Algorithme 5.1 : acquisition des données de l'expérience contenues dans le fichier XML

```
1 construire le document correspondant au fichier ;
2 récupérer l'élément de plus haut niveau du document ;
3 récupérer le nom du responsable de l'expérience via getAttribute ;
4 construire la liste l des nœuds fils de l'élément de plus haut niveau ;
5 pour tous les nœuds e dans l faire
6 |   si e est un élément XML alors
7 |   |   convertir le texte contenu dans e en réel et l'ajouter à la liste des données ;
8 |   fin
9 fin
```

6 L'API d'entrée/sortie en Java

L'API d'entrée/sortie de Java est définie dans le paquetage `java.io`. Cette API fournit une interface standard pour gérer les flux d'entrée/sortie. Un flux est une séquence ordonnée de données qui peut avoir une *source* (*input streams*) ou une *destination* (*output streams*). Les classes fournies par le paquetage libèrent le programmeur de tous les détails d'implantation spécifiques au système d'exploitation par exemple.

6.1 Classes de base

Il y a deux types de flux qui sont gérés en Java :
- les flux de caractères (codés sur 16 bits en Unicode), donc lisibles par un être humain ;
- les flux d'octets (codés sur 8 bits), par exemple une image.

À ces deux types de flux correspondent quatres classes *abstraites* :

	Caractères	Bytes
Entrée	`Reader`	`InputStream`
Sortie	`Writer`	`OutputStream`

On va maintenant décrire ces quatre classes. La documentation javadoc de toutes les classes est disponible sur [1] en sélectionnant le paquetage `java.io`.

6.1.1 La classe `InputStream`

La classe `InputStream` représente un flux d'octets en entrée. Elle propose plusieurs méthodes intéressantes :
- **public abstract int** `read()` **throws** `IOException` qui permet de lire un octet de la source et le renvoie sous forme d'un entier compris entre 0 et 255. Si le résultat est -1, c'est que l'on a atteint la fin du flux ;
- **public int** `read(`**byte**`[] buf, `**int**` offset, `**int**` count)` **throws** `IOException` qui permet de lire les valeurs de `buf` depuis la position `offset` jusqu'à `offset+count` ;
- **public long** `skip(`**long**` count)` **throws** `IOException` permet de « sauter » `count` octets depuis le début du flux (le nombre effectif d'octets « sautés » est renvoyé) ;
- **public void** `close()` **throws** `IOException` permet de fermer le flux.

6.1.2 La classe `OutputStream`

La classe `OutputStream` permet de modéliser des flux d'octets vers une destination. Elle possède en particulier les méthodes suivantes :
- **public abstract void** `write(`**int** `b)` **throws** `IOException` : écrit l'entier `b` comme un octet ;
- **public void** `write(`**byte**`[] buf, `**int**` offset, `**int**` count)` **throws** `IOException` écrit `count` octets du tableau d'octets `buf`, depuis `buf[offset]` ;
- **public void** `flush()` permet de vider le flux si celui-ci a « bufferisé » un certain nombre de flux. Si la destination est un autre flux, celui-ci est également flushé ;
- **public void** `close()` **throws** `IOException` qui permet de fermer le flux de sortie.

6.1.3 La classe `Reader`

La classe `Reader` propose plusieurs méthodes intéressantes qui sont très proches de celles de `InputStream`, mais qui travaillent avec des caractères :
- **public int** `read()` **throws** `IOException` qui permet de lire un caractère de la source et le renvoie sous forme d'un entier compris entre 0 et 255. Si le résultat est -1, c'est que l'on a atteint la fin du flux ;
- **public** abtract **int** `read(`**char**`[] buf, `**int**` offset, `**int**` count)` **throws** `IOException` ;
- **public long** `skip(`**long**` count)` **throws** `IOException` ;
- **public void** `close()` **throws** `IOException`.

6.1.4 La classe `Writer`

La classe `Writer` propose le pendant des méthodes de `OutputStream` pour les caractères :
- **public void** `write(`**int** `ch)` **throws** `IOException` : écrit l'entier `ch` comme un caractère ;
- **public abstract void** `write(`**char**`[] buf, `**int**` offset, `**int**` count)` **throws** `IOException` ;
- **public void** `flush()` ;
- **public void** `close()` **throws** `IOException`.

6.2 Les classes `InputStreamReader` et `OutputStreamWriter`

Les classes `InputStreamReader` et `OutputStreamWriter` permettent de transformer un flux d'octets en un flux correspondant de caractères. Elles possédent les constructeurs suivants :
- **public** `InputStreamReader(InputStream in)`
- **public** `InputStreamReader(InputStream in, String encoding)`
- **public** `OutputStreamWriter(OutputStream out)`
- **public** `OutputStreamWriter(OutputStream out, String encoding)`

Ces classes peuvent être très utiles, en particulier lorsque l'on utilise les entrées et sorties standards, `System.in` et `System.out`, qui sont des flux d'octets et non de caractères.

6.3 Quelques classes concrètes du paquetage `java.io`

6.3.1 Les flux « bufferisés »

Les flux bufferisés sont représentés par les classes (`BufferedInputStream`, `BufferedOutputStream`, `BufferedReader` et `BufferedWriter`). Ces classes permettent d'éviter de lire ou d'écrire chaque octet ou caractère dans les flux (cf. section 6.4 pour un exemple). Cela sert particulièrement pour les fichiers, car il serait très peu performant d'écrire octet par octet dans un fichier par exemple.
Lors d'une lecture sur un flux bufferisé, le flux est rempli au maximum par son flux d'entrée. Lors d'une écriture sur un flux bufferisé, on remplit le buffer et on peut le vider avec `flush`.

6.3.2 Les flux `Print`

Les flux de type `Print`, i.e. `PrintStream` et `PrintWriter` permettent d'écrire d'écrire facilement des valeurs de types primitifs (**`int`**, **`double`** etc.). Ils disposent de deux méthodes `print` et `println`.

6.3.3 Les flux vers et depuis les fichiers

On peut également utiliser des flux vers et depuis des fichiers. Ils sont représentés par les classes `FileInputStream`, `FileOutputStream`, `FileReader` et `FileWriter`. Les constructeurs de ces classes peuvent prendre en paramètre :
- une chaine de caractère qui est le nom du fichier (le plus utilisé...) ;
- un objet de type `File` ;
- un objet de type `FileDescriptor`.

On utilisera principalement le premier constructeur. Attention, ces constructeurs peuvent lever les exceptions suivantes :
- `FileNotFoundException` si le fichier n'existe pas ;
- `SecurityException` si on n'a pas le droit de lire le fichier.

Dans le cas des deux premiers constructeurs et d'un flux d'écriture, si le fichier n'existe pas, il est créé.

6.3.4 Les autres classes

Il existe d'autres classes qui représentent des flux particuliers : `Piped` streams qui ont une entrée et une sortie qui communiquent, `Scanner` qui permet de « parser » un flux en utilisant des tokens qui servent de délimiteurs etc. Vous trouverez tous les renseignements possibles dans la documentation de l'API.

6.4 Exemple d'utilisation

Voici un exemple d'utilisation : il s'agit d'une classe qui possède une méthode statique permettant de récupérer un entier entré au clavier et de l'afficher à l'écran.

Listing 1– `EntierClavier.java`

```
import java.io.*;  // attention, il faut l'importer !

public class EntierClavier {
    public static void lireEntier() {
        try {
            System.out.print("Entrez un entier : ");

            // on prend l'entree standard et on la place dans un InputStreamReader
            // pour pouvoir travailler avec des caracteres
            InputStreamReader aux = new InputStreamReader(System.in);

            // on met tout ca dans un buffer pour faciliter la lecture
            BufferedReader in = new BufferedReader(aux);

            // on lit l'entree du buffer. La methode trim() permet d'enlever les
            // eventuels espaces a la fin de la chaine
            String s = in.readLine().trim();

            // on essaye de transformer la chaine en entier grace a la classe
            // Integer. Attention aux exceptions !
```

```
            int n = Integer.parseInt(s);

            System.out.println("Le nombre est : " + n);
        } catch (NumberFormatException  e) {
            System.out.println("Ce n'est pas un entier !"); }
        catch (IOException e) {
            System.out.println("Erreur d'entree/sortie !"); }
    }

    public static void main(String[] args) {
        lireEntier();
    }
}
```

Références

[1] Oracle. *Java API specifications*. 2013. URL : http://docs.oracle.com/javase/7/docs/api/index.html.

[2] Oracle. *The Java Tutorials – Trail : Java API for XML Processing (JAXP)*. 2013. URL : http://docs.oracle.com/javase/tutorial/jaxp/index.html.

[3] Wikipedia contributors. *XML*. 2103. URL : http://en.wikipedia.org/wiki/XML.

License CC BY-NC-SA 3.0