IN201 : 13 – 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.

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 d'entrée/sortie de Java.

2 Problématique

On souhaite récupérer des données provenant d'un fichier. Ces données sont des réels qui doivent normalement être positifs (par exemple, ils représentent une pression). 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 ;
– renvoi d'un itérateur sur l'ensemble de données.

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` ? Faudra-t-il utiliser des exceptions ? Si oui, faudra-t-il en définir vous-même ?
3. construire le diagramme UML de la classe.

4 Implantation de la classe `Acquisition`

Il faut maintenant implanter la classe `Acquisition`.

1. choisir un attribut permettant de stocker les valeurs ;
2. utiliser la documentation de l'API d'entrée/sortie de Java présentée dans la section 5 pour lire le fichier. Utiliser également le programme présenté dans la section 5.4 pour comprendre comment récupérer des entiers ;
3. construire les exceptions dont vous avez besoin ;
4. les exceptions levées dans les méthodes doivent-elle être rattrapées et traiter dans la classe ou propagées ?
5. utiliser les fichiers proposés sur le site pour vérifier que la classe est opérationnelle. En particulier, réfléchir à l'écriture d'une classe de test JUnit.

5 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.

5.1 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 [**javadoc_api**] en sélectionnant le paquetage `java.io`.

5.1.1 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.

5.1.2 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.

5.1.3 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`.

5.1.4 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`.

5.2 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.

5.3 Quelques classes concrètes du paquetage `java.io`

5.3.1 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 5.4 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`.

5.3.2 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`.

5.3.3 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.

5.3.4 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.

5.4 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();
    }
}
```