UFR ST - Besançon - L2 Info - POA

TP 3 - Exceptions, Gestion de fichiers

1 Exceptions : écriture et gestion

Exercice 1. Exception personnalisée

On imagine que les notes d'une promo sont stockées dans un tableau de notes. On va écrire une méthode qui inspecte ce tableau de notes et qui lance une exception aussitôt qu'elle repère une note négative, ou qu'elle repère une note strictement supérieure à 20.

Question 1. Créez une classe d'exception GradeException par héritage de la classe Exception. N.B. Vous pouvez pour cela simplement adapter, i.e. en changeant son nom, le code de la classe BadIndexesException qui vous est donné dans le PDF « CM 3 part. 2 - Les exceptions en java » sur Moodle.

Question 2. Rédigez, dans une classe Main (celle qui contiendra votre programme principal) une méthode statique qui inspecte un tableau de notes et lance une exception de type GradeException si elle détecte soit une note négative, soit une note strictement supérieure à 20. Le message de l'exception devra indiquer lequel de ces deux cas a été rencontré, ainsi que la note fautive.

On va maintenant rédiger un programme principal qui crée un tableau de notes, puis demande son inspection par la méthode précédente.

Question 3. Rédigez un tel programme principal qui ignore l'exception grâce à une instruction throws.

Question 4. Modifiez votre programme principal pour qu'il invoque la méthode d'inspection dans un bloc try/catch/finally. Le bloc catch affichera la note incorrecte et la raison (< 0 ou > 20). Le bloc finally affichera selon les cas le message "Toutes les notes sont correctes" ou "Des notes incorrectes ont été trouvées".

2 Découverte des entrées/sorties en java

En java, comme en programmation d'une manière générale, la gestion des entrées/sorties est un vaste sujet. Ce TP a pour modeste objectif de vous faire découvrir quelques uns des mécanismes de base en java. Les classes permettant la gestion des entrées/sorties se trouvent rassemblées dans le package java.io.

Attention. La plupart des méthodes de gestion des entrées/sorties lèvent des exceptions (fichier non trouvé, fin de fichier rencontrée, etc.). Vous devrez donc englober leurs appels dans des blocs try/catch/finally pour les invoquer proprement.

2.1 La classe File

La classe java File n'offre pas de méthodes permettant l'écriture et la lecture des fichiers, mais des méthodes permettant de gérer leur nommage, leur position dans l'arborescence du système, les droits en lecture/écriture, etc. Vous pouvez consulter son API pour en découvrir le contenu. On retient par exemple les méthodes (ou constructeurs) suivants :

- `File (String pathname)` : crée une instance de File désignée par le chemin pathname.
- `File(String parent, String child)` : crée une instance de File sous le nom child dans le répertoire parent.
- `canWrite(), canRead(), canExecute()` : teste respectivement si le fichier est accessible en lecture, en écriture, ou est exécutable.

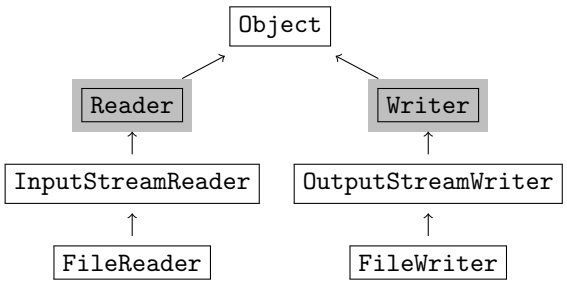
— `exists()` : teste si le fichier existe.
— `delete()` : efface le fichier.
— etc.

Les constructeurs des classes permettant la lecture ou l'écriture de fichiers peuvent prendre en paramètre une instance de la classe File. On note qu'ils peuvent alternativement prendre en paramètre une chaîne de caractères indiquant directement le nom du fichier. Le fichier est alors créé par défaut dans l'espace de travail courant, et on n'a pas la souplesse des méthodes décrites précédemment.

2.2 Les fichiers texte

Les fichiers texte enregistrent des séquences de caractères, qui peuvent être lues par n'importe quel éditeur de texte.

2.2.1 Hiérarchie simplifiée



Les classes permettant la lecture du contenu de fichiers texte dérivent de la classe abstraite Reader, et celles permettant l'écriture dans un fichier texte dérivent de la classe abstraite Writer. Les deux sous-classes concrètes correspondantes les plus basiques sont respectivement InputStreamReader et OutputStreamWriter : elles assurent la conversion de flux d'octets en flux de caractères.

En pratique, on utilisera plutôt leurs sous-classes respectives FileReader et FileWriter qui fournissent des constructeurs plus élaborés (voir l'API de ces classes).

2.2.2 Ecriture de fichiers texte

Le mécanisme d'écriture dans un fichier est résumé ainsi :
— créer une instance de FileWriter (ou utiliser une instance existante : le contenu en sera écrasé),
— écrire la succession des caractères au moyen d'appels répétés à la méthode write(),
— refermer le fichier au moyen de la méthode close().

Gestion des exceptions. N'oubliez pas que les accès aux fichiers (ici écriture et fermeture) sont susceptibles de lever des exceptions, et devront être englobés dans des blocs try/catch/finally. Comme un fichier devra être refermé dans tous les cas (qu'une exception ait été rencontrée ou nom), sa fermeture sera réalisée dans le bloc finally, qui est toujours exécuté. Mais comme la fermeture soulève elle-même une éventuelle exception, le bloc finally devra englober un bloc try/catch !

Le schéma recommandé est donc le suivant :

```
// instanciation, ou utilisation d'une instance existante
try {
   // écritures successives avec write(...)
}
catch (IOException e) {
  // traitement de l'exception, par exemple System.out.println("Erreur : "+e.getMessage());
}
```

```
finally {
   try {
      // fermeture du fichier
   }
   catch (IOException e) {
      // traitement
   }
}
```

Exercice 2. Ecriture d'un fichier texte aléatoire, caractère par caractère

1. Dans un programme principal, définissez deux constantes entières `NB_LINES` (nombre de lignes) et `LINE_SIZE` (nombre de caractères par ligne), initialisées par exemple respectivement à 10 et 30.
2. Ouvrez un fichier (en écriture) en créant une instance de FileWriter (et de File).
3. Remplissez le de `NB_LINES` lignes de `LINE_SIZE` caractères chacune. Les caractères seront choisis au hasard entre `'A'` et `'Z'`. On rappelle qu'une fin de ligne est obtenue en écrivant le caractère `'\n'`.
4. N'oubliez pas de refermer le fichier une fois celui-ci rempli.
5. Visualisez le fichier dans un éditeur de texte pour en vérifier le contenu.

2.2.3 Lecture de fichiers texte

L'ouverture d'un fichier texte en lecture s'obtient en instanciant un objet de la classe FileReader. **Attention.** Cette instanciation peut déclencher une FileNotFoundException si le fichier n'est pas trouvé.

La lecture des caractères successifs est obtenue par des appels répétés à la méthode read(). Une méthode ready() renvoie un booléen disant s'il reste des caractères à lire. **Attention.** La méthode read() ne renvoie pas un `char` mais un `int` correspondant au code ASCII du caractère lu (ou -1 s'il n'y avait plus de caractère à lire).

N'oubliez pas de gérer les exceptions et de refermer le fichier en fin de lecture par un appel à la méthode close() dans un bloc finally.

Exercice 3. Lecture d'un fichier texte

Relisez le fichier texte créé à l'exercice précédent, en envoyant un à un sur la console les caractères lus.

2.3 Fichiers texte bufferisés

Si vous regardez l'API de la classe FileReader (ou plutôt de sa super-classe InputStreamReader), vous verrez que la méthode de lecture read() est surchargée pour pouvoir lire aussi des caractères en série plutôt que un par un. La série lue est stockée dans un tableau de `char`. Un tel tableau s'appelle un *buffer* (en français : un « tampon »). Cette technique, désignée par l'anglicisme *bufferisation*, augmente l'efficacité des opérations de lecture et d'écriture.

On va la tester en générant puis en lisant un gros fichier, d'abord sans bufferisation puis avec bufferisation. On mesurera les divers temps d'exécution pour pouvoir les comparer.

Mesures du temps d'exécution. L'appel java `System.currentTimeMillis()` retourne un entier `long` indiquant le nombre de millisecondes écoulées depuis le début du 1er janvier 1970. Le schéma suivant

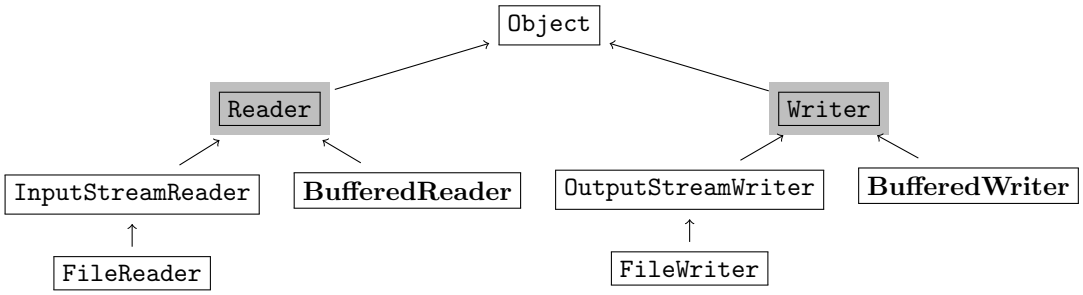
```
long startTime, endTime;
startTime = System.currentTimeMillis();
   // Portion de code à exécuter
endTime = System.currentTimeMillis();
System.out.println("Temps d'exécution : "+(endTime-startTime));
```

permet donc de chronométrer le temps d'exécution d'une portion de code.

Exercice 4. Mesures des temps de génération et de lecture d'un gros fichier, en version non bufferisée

1. Modifiez les constantes de votre programme pour générer un gros fichier : vous pouvez par exemple fixer `NB_LINES` à 2 000 000 et `LINE_SIZE` à 80. Comme la génération de nombres aléatoires est lente, il est conseillé de remplacer la génération de caractères aléatoires par l'écriture d'un caractère fixe (par exemple 'A').
2. Mesurez et affichez le temps d'exécution correspondant à la génération de ce fichier.
3. Même chose pour la relecture.

2.3.1 Les classes BufferedReader et BufferedWriter



Au delà de la bufferisation de la méthode read() évoquée précédemment, l'API java propose deux classes nommées BufferedReader et BufferedWriter. Elles dérivent respectivement directement des classes abstraites Reader et Writer, et réalisent automatiquement des opérations respectivement de lecture et d'écritures bufferisées.

Les classes BufferedReader et BufferedWriter permettent d'envelopper (en anglais *to wrap*) des « readers » et « writers » tels que FileReader et FileWriter. En clair dans notre exemple, cela signifie que l'on va passer des instances de FileReader et FileWriter en paramètres aux constructeurs respectifs de BufferedReader et BufferedWriter.

Exemple.

```
FileReader myReader = new FileReader("MyFile.txt");
BufferedReader myBufReader = new BufferedReader(myReader);
```

ou directement

```
BufferedReader myBufReader = new BufferedReader(new FileReader("MyFile.txt"));
```

On peut maintenant confier les opérations de lecture et d'écriture aux méthodes (plus efficaces) de BufferedReader et BufferedWriter.

Ecriture avec BufferedWriter. Consultez l'API de la classe BufferedWriter.

- Méthode write() : elle prend un code ASCII en paramètre et écrit le caractère correspondant dans le fichier. Mais ce n'est pas fait en direct : la classe dispose d'un buffer permettant de stocker plusieurs caractères (8192 octets par défaut), et l'écriture n'a lieu qu'une fois le buffer plein.
- Méthode flush() : permet de vider le buffer pour provoquer l'écriture (sauf cas particulier, on ne l'appellera pas nous-mêmes, java s'en charge).
- Méthode close() : le buffer est vidé, et donc écrit, avant la fermeture du fichier.
- ...

Lecture avec BufferedReader. Consultez l'API de la classe BufferedReader.

- Méthode read() : elle permet de lire le fichier caractère par caractère, mais de manière bufferisée.
- Méthode readLine() : elle permet de lire directement toute une ligne de texte (et toujours de manière bufferisée).
- ...

Exercice 5. Mesures des temps de génération et de lecture d'un gros fichier en version *bufferisée*

1. Complétez votre programme pour générer un deuxième gros fichier en ayant au préalable enveloppé un FileWriter dans un BufferedWriter.
2. Complétez votre programme pour relire ce fichier **caractère par caractère** (avec la méthode read() donc) en ayant au préalable enveloppé un FileReader dans un BufferedReader.
3. Mesurez et affichez les temps d'exécution correspondants, et comparez avec les versions non bufferisées.
4. Complétez votre programme pour relire cette fois le fichier ligne par ligne (avec la méthode readLine()).
5. Mesurez et affichez les temps d'exécution de cette opération : impressionnant non ?

2.4 Les fichiers de données

En plus de lire et écrire des fichiers texte, java permet également de de lire et d'écrire des données de n'importe quel type primitif. Mais à la différence des fichiers texte, ils ne pourront être relus que par des programmes java.

Le concept général des entrées/sorties est que les données transitent par des *flux* (en anglais *stream*) : en lecture, les données proviennent d'un flux, et en écriture, les données sont dirigées vers un flux. Ces flux peuvent correspondre à des fichiers, mais aussi à des dispositifs tel que des claviers et des écrans, ou encore à des sites distants... A la base les flux transportent des octets.

Nous avons déjà étudié le concept de fichiers textes, qui sont des flux où les octets sont interprétés comme codant des caractères. On va généraliser à des flux binaires où les octets seront interprétés comme codant les différents types primitifs du langage.

Les flux binaires entrants (pour les opérations de lecture) dérivent de la classe abstraite InputStream et les flux binaires sortants (pour les opérations d'écriture) dérivent de la classe abstraite OutputStream. Lorsqu'il s'agit de fichiers, les sous-classes concrètes correspondantes sont respectivement FileInputStream et FileOutputStream.

L'API java pour les manipulations de flux est très riche. On n'entrera pas dans les détails ici, l'objectif étant de se limiter aux opérations les plus courantes. On retient juste qu'on ajoute des fonctionnalités à un objet flux existant grâce à des classes particulières appelées « filtres ». L'idée est d'envelopper (*wrap*) un flux binaire dans une classe offrant plus de fonctionnalités (comme on l'avait fait par exemple pour les fichiers texte en enveloppant un FileReader dans un BufferedReader).

OK, et en pratique alors ? Pour lire un flux binaire depuis un fichier (FileInputStream), on peut d'abord l'envelopper dans sa version bufferisée (BufferedInputStream), puis l'envelopper encore dans une version capable de décoder les types java primitifs (DataInputStream).

Ainsi, de manière usuelle, on construit un objet `myReader` pour lire un fichier de données (désigné par `fileName`) au moyen de :

```
DataInputStream myReader = new DataInputStream(new BufferedInputStream(new FileInputStream(fileName));
```

Et dans l'autre sens, c'est à dire pour écrire, c'est la même chose en remplaçant Input par Output :

```
DataOutputStream myWriter = new DataOutputStream(new BufferedOutputStream(new FileOutputStream(fileName));
```

Exercice 6. Création et relecture d'un fichier de données

1. Dans un programme principal, créez un fichier de données que vous remplirez de quelques entiers lus au clavier : saisissez les par exemple avec la méthode `Clavier.saisirInt()` fournie sur Moodle dans la classe `Clavier` ; puis écrivez les dans le fichier avec la méthode `writeInt()` (voir l'API de la classe `DataOutputStream`).
2. Ouvrez le fichier ainsi créé dans Eclipse : que voyez-vous ? Savez-vous pourquoi ? (Si la réponse est non, attendez la question suivante.)
3. Relancez votre programme en saisissant maintenant successivement les entiers 65, 83, 67, 73 puis 73.
4. Ré-ouvrez le fichier dans Eclipse : que voyez-vous ? Vous avez maintenant l'explication ?
5. Complétez votre programme pour relire le fichier grâce à un objet de type DataInputStream, afin de récupérer les entiers qui y ont été enregistrés. **Attention.** Assez curieusement, la fin de fichier en java doit être gérée comme une exception de type `EOFException` : voir l'API de la méthode `readInt()`. On lit donc les entiers jusqu'à ce qu'une exception `EOFException` soit levée, que l'on doit rattraper pour mettre fin à la lecture.

---

3 Pour terminer

Le clavier est d'ordinaire considéré comme « l'entrée standard ». Cela correspond en java au flux d'entrée standard intitulé `System.in` (de même que le flux de sortie standard, en général la console, est désigné par `System.out`).

La technique usuelle pour récupérer des données saisies au clavier consiste à envelopper l'entrée standard dans un objet de type `InputStreamReader`, à son tour enveloppé dans un objet de type `BufferedReader` :

```
BufferedReader lecteurClavier = new BufferedReader(new InputStreamReader(System.in));
```

Par des appels à la méthode `readLine()`, on peut alors récupérer les lignes saisies au clavier :

```
String lineRead = lecteurClavier.readLine();
```

Il faut encore interpréter une telle ligne au bon format : si c'est un entier (`int`) qui est supposé être lu, alors on essaie de décoder la ligne au moyen de `Integer.parseInt(lineRead)` (dans le cas où la saisie ne correspondrait pas à un entier, cela déclencherait une exception). De même si on attend un réel (`double`), on essaie `Double.parseDouble(lineRead)`, etc.

Vous comprenez maintenant pourquoi une classe telle que `Clavier` vous est fournie pour faciliter les opérations de lecture ! La classe `Clavier` a été développée à l'UFR ST et ne fait pas partie de l'API java standard. Pour une solution de saisie issue de l'API java vous pouvez consulter l'API de la classe `Scanner`.

Exercice 7. Réaliser une saisie d'entiers au clavier sans passer par une classe utilitaire

Vous avez tous les ingrédients pour pouvoir saisir des données au clavier sans passer par la classe Clavier ou la classe Scanner. Réalisez la saisie d'entiers au clavier depuis votre programme principal.
