Leçon 2 : Gestions des flux d'entrées, de sortie et des exceptions

Université Virtuelle de Côte d'Ivoire

Table des matières

I - **1- La gestion des exceptions** 3

II - **Exercice** 7

III - **2- La gestion Les flux d'entrée et de sortie** 8

IV - **Exercice** 12

1- La gestion des exceptions

I

Définition : 1.1- Définition

Les exceptions représentent le mécanisme de gestion des erreurs intégré au langage Java. Il se compose d'objets représentant les erreurs et d'un ensemble de trois mots clés qui permettent de détecter et de traiter ces erreurs (*try, catch et finally*) et de les lever ou les propager (*throw et throws*).

Lors de la détection d'une erreur, *un objet qui hérite de la classe Exception est créé* (on dit qu'une exception est levée) et *propagé à travers la pile d'exécution jusqu'à ce qu'il soit traité.*

Ces mécanismes permettent de renforcer la sécurité du code Java.

1.2- Le Bloc try, catch et finally

Le *bloc try* rassemble les appels de méthodes susceptibles de produire des erreurs ou des exceptions. L'instruction *try* est suivie d'instructions entre des accolades.

La gestion d'une exception se fait selon le schéma suivant :

- *Cas 1 :*

try{

appel de la fonction susceptible de générer l'exception

} catch (Exception e){

traiter l'exception e

}

instruction suivante

Remarque :

Si la fonction ne génère pas d'exception, on passe alors à instruction suivante, sinon on passe dans le corps de la clause *catch* puis à instruction suivante. *e* est un objet dérivé du type Exception. On peut être plus précis en utilisant des types tels que IOException, SecurityException,ArithmeticException, etc .

Il existe une vingtaine de types d'exceptions. En écrivant catch (Exception e), on indique qu'on veut gérer toutes les types d'exceptions.

Si le code de la clause try est susceptible de générer plusieurs types d'exceptions, on peut vouloir être plus précis en gérant l'exception avec plusieurs clauses catch :

- *Cas 2* :

```
try{
appel de la fonction susceptible de générer l'exception
} catch (IOException e){
traiter l'exception e
}
} catch (ArithmeticException e){
traiter l'exception e
}
} catch (RunTimeException e){
traiter l'exception e
}
instruction suivante
```

- *Cas 3 : On peut ajouter aux clauses try/catch, une clause finally :*

```
try{
appel de la fonction susceptible de générer l'exception
} catch (Exception e){
traiter l'exception e
}
finally{
code exécuté après try ou catch
}
instruction suivante
```

Remarque :

- Qu'il y ait exception ou pas, le code de la clause *finally* sera toujours exécuté.
- La classe Exception a une méthode *getMessage()* qui rend un message détaillant l'erreur qui s'est produite. Ainsi si on veut afficher celui-ci, on écrira :

```
catch (Exception ex){
System.out.println("L'erreur suivante s'est produite : "+e.getMessage());
...
}
```

- La classe Exception a une méthode toString() qui rend une chaîne de caractères indiquant le type de l'exception ainsi que la valeur de la propriété Message. On pourra ainsi écrire :

```
catch (Exception e){
System.out.println ("L'erreur suivante s'est produite : "+e.toString());
...
```

}

- On peut écrire aussi :

catch (Exception ex){

System.out.println ("L'erreur suivante s'est produite : "+e);

...

}

Nous avons ici une opération string + Exception qui va être automatiquement transformée en string + Exception. toString() par le compilateur afin de faire la concaténation de deux chaînes de caractères.

```
1 public class TestException {
2 public static void main(String[] args) {
3   int j = 20, i = 0;
4   try {
5     System.out.println(j/i);
6   } catch (Exception e) { // Récupère l'exception a
7     System.out.println("L'erreur suivante s'est produite : "+e.getMessage());
8   }
9   }
10 }
```

1.3- Les exceptions personnalisées

Pour créer une exception personnalisée il faut :

- *créer une classe héritant de la classe Exception, par convention, le nom de la classe exception se termine par Exception »*
- *renvoyer l'exception levée à notre classe créer précédemment*
- *gérer maintenant l'exception levée dans notre classe*

Remarque :

- *throws* : ce mot clé permet de signaler à la JVM(Java virtual machine) qu'un morceau de code, une méthode, une classe... est potentiellement dangereux et qu'il faut utiliser un bloc try{...}catch{...}. Il est suivi du nom de la classe qui va gérer l'exception.
- *throw* : celui-ci permet tout simplement de lever une exception manuellement en instanciant un objet de type Exception (ou un objet hérité).

Syntaxe classe execption :

public class ApplicationException extends Exception {

public ApplicationException () {

//opération à réaliser

}

}

Exemple :

```
1 //**************Création de la classe d'exception *****
2 public class ControlSaisieException extends Exception { //Crétion de la classe
  ControlSaisieException  qui hérite la //classe EXCEPTION
```

```
3     public ControlSaisieException() { // construction de la classe
  ControlSaisieException
4      System.out.println("Le champ ne doit pas être vide");
5     }
6   }
7 //**************Création de la classe Textsaisie utilisant l'exeception *****
8 public class Textsaisie{      // création de la classe Textsaisie
9     public static void controle(String chaine) // Création de la méthode controle
10    throws ControlSaisieException { //signifie qu'il y a un code à vérifier
11    if (chaine.equals("") == true) // vérifie si le paramètre chaine est vide
12    throw new ControlSaisieException(); // fait appel au constructeur de la
  classe ControlSaisieException
13    }
14
15
16   public static void main(String[] args) { // Point d'exécution de la classe
  Textsaisie
17    // TODO Auto-generated method stub
18    String chaine1 = "A"; // variable chaine1 de type String
19    try { // permet de capturer une erreur lors de l'exéction de la méthode
  contrôle
20       controle(chaine1); // appel de la méthode contrôle en passant chaine1 en
  paramètre
21    }
22    catch (ControlSaisieException e) {}; // capture l'erreur si chaine est vide
  et affiche le message de la classe //ControlSaisieException
23
24
25   }
26
27 }
28
```

Exercice

II

Énoncé :

Créer un classe exception pour empêcher les nombres nuls

Solution :

ControlValeurException {

{

("Le nombre ne doit pas être nul");

}

}

2- La gestion Les flux d'entrée et de sortie

2.1- La présentation des flux

Un programme a souvent besoin d'échanger des informations pour recevoir des données d'une source ou pour envoyer des données vers un destinataire.

La source et la destination de ces échanges peuvent être de nature multiple : un fichier, une socket réseau, un autre programme, etc ...

De la même façon, la nature des données échangées peut être diverse : du texte, des images, du son, etc ...

Les flux (*stream* en anglais) permettent d'encapsuler ces processus d'envoi et de réception de données. Les flux traitent toujours les données de façon séquentielle.

En java, les flux peuvent être divisés en plusieurs catégories :

- *les flux d'entrée (inputstream) et les flux de sortie (outputstream)*
- *les flux de traitement de caractères et les flux de traitement d'octets*

Java définit des flux pour lire ou écrire des données mais aussi des classes qui permettent de faire des traitements sur les données du flux. Ces classes doivent être associées à un flux de lecture ou d'écriture et sont considérées comme des filtres.

Par exemple, il existe des filtres qui permettent de mettre les données traitées dans un tampon (buffer) pour les traiter par lots.

Toutes ces classes sont regroupées dans le *package java.io*

Définition : 2.2- Les flux d'entrée/sortie

Une entrée/sortie en Java consiste en un échange de données entre le programme et une autre source, par exemple la mémoire, un fichier, le programme lui-même... .

Pour réaliser ce échange, Java emploie ce qu'on appelle *un stream (qui signifie « flux »).* Celui-ci joue le rôle de médiateur entre la source des données et sa destination.

Toute opération sur les entrées/sorties doit suivre le schéma suivant : *ouverture, lecture, fermeture du flux.*

2.3 Les classes de gestion de flux

Il est difficile de choisir la classe de gestion de flux qui convient le mieux en fonction des besoins. Pour faciliter ce choix, il faut comprendre la dénomination des classes : cela permet de sélectionner la ou les classes adaptées aux traitements à réaliser.

Le nom des classes se compose d'un préfixe et d'un suffixe. Il y a quatre suffixes possibles en fonction du type de flux (flux d'octets ou de caractères) et du sens du flux (entrée ou sortie).

Il existe donc quatre hiérarchies de classes qui encapsulent des types de flux particuliers. Ces classes peuvent être séparées en deux séries de deux catégories différentes :

les classes de lecture et d'écriture et les classes permettant la lecture de caractères ou d'octets.

	Flux d'octets	**Flux de caractères**
Flux d'entrée	InputStream	Reader
Flux de sortie	OutputStream	Writer

- les sous-classes de *Reader* sont des types de flux en lecture sur des ensembles de caractères
- les sous-classes de *Writer* sont des types de flux en écriture sur des ensembles de caractères
- les sous-classes de *InputStream* sont des types de flux en lecture sur des ensembles d'octets (images, sons etc)
- les sous-classes de *OutputStream* sont des types de flux en écriture sur des ensembles d'octets (images, son etc).

Pour le préfixe, il faut distinguer les flux et les filtres. Pour les flux, le préfixe contient la source ou la destination selon le sens du flux.

Préfixe du flux	**Source ou destination du flux**
ByteArray	tableau d'octets en mémoire
CharArray	tableau de caractères en mémoire
File	fichier
Object	objet
Pipe	pipeline entre deux threads
String	chaîne de caractères

Pour les filtres, le préfixe contient le type de traitement qu'il effectue. Les filtres n'existent pas obligatoirement pour des flux en entrée et en sortie.

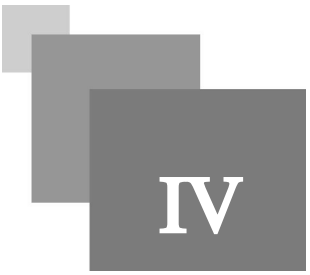
Type de traitement	Préfixe de la classe	En entrée	En sortie
Mise en tampon	Buffered	Oui	oui
Concaténation de flux	Sequence	Oui pour flux d'octets	non
Conversion de données	Data	Oui pour flux d'octets	Oui pour flux d'octets
Numérotation des lignes	LineNumber	Oui pour les flux de caractères	non
Lecture avec remise dans le flux des données	PushBack	Oui	non
Impression	Print	Non	oui
Sérialisation	Object	Oui pour flux d'octets	Oui pour flux d'octets
Conversion octets /caractères	nputStream / OutputStream	Oui pour flux d'octets	Oui pour flux d'octets

- *Buffered* : ce type de filtre permet de mettre les données du flux dans un tampon. Il peut être utilisé en entrée et en sortie
- *Sequence* : ce filtre permet de fusionner plusieurs flux.
- *Data* : ce type de flux permet de traiter les octets sous forme de type de données
- *LineNumber* : ce filtre permet de numéroter les lignes contenues dans le flux
- *PushBack* : ce filtre permet de remettre des données lues dans le flux
- *Print* : ce filtre permet de réaliser des impressions formatées
- *Object* : ce filtre est utilisé par la sérialisation
- *InputStream / OuputStream* : ce filtre permet de convertir des octets en caractères

La *package java.io* définit ainsi plusieurs classes :

	Flux en lecture	Flux en sortie
Flux de caractères	BufferedReader CharArrayReader FileReader InputStreamReader LineNumberReader PipedReader PushbackReader StringReader	BufferedWriter CharArrayWriter FileWriter OutputStreamWriter PipedWriter StringWriter
Flux d'octets	BufferedInputStream ByteArrayInputStream DataInputStream FileInputStream ObjectInputStream PipedInputStream PushbackInputStream SequenceInputStream	BufferedOutputStream ByteArrayOutputStream DataOuputStream FileOutputStream ObjetOutputStream PipedOutputStream PrintStream

Exercice



1. La package regroupant les flux est
2. Le flux de lecture sur des ensembles d'octets est
3. Pour réaliser un échange de données entre le programme et une autre source java utilise un
4. Le flux en écriture sur des ensembles d'octets est
5. Le filtre qui permet de numéroter les lignes contenues dans le flux est