Programmation Orientée Objet
M2 CCI 2008–2009
hoenen@icps.u-strasbg.fr

Université de Strasbourg
Département d'informatique
28 janvier 2009

TP5 : i/o et exceptions

1 Les exceptions

Pourquoi des exceptions ? Lorsqu'il y a une erreur durant l'exécution d'une méthode, un objet d'exception est créé puis *lancé* par cette méthode. La JVM cherche alors un morceau de code présent dans la pile d'appels de cette méthode et qui est capable *d'attraper* cette exception (il faut que le type de l'exception lancée et attrapée soit le même). Si l'exception n'a pas été traitée, alors le programme se termine immédiatement.

Différents types d'exception Il y existe différents types d'exceptions ne nécessitant pas le même traitement. Les exceptions de type *Error*, *RuntimeException* et de toutes les classes qui en héritent ne sont pas à traiter systématiquement, il est même préferrable que le programme s'arrête en affichant la pile des appels. En effet, il s'agit souvent d'erreurs de logique ou de bugs dans le programme. Les autres exceptions sont des *checked exceptions*, qu'il est obligatoire de traiter par la règle *catch or specify* :

- **cacth :** englober l'appel de la méthode dans un bloc *try-catch* pour la traiter directement,
- **specify :** lors de sa définition, ajouter après les arguments de la méthode le mot-clé `throws` suivie du type de l'exception (ou plusieurs types séparés par une virgule). Cela permet de faire suivre l'exception à la méthode appelante qui devra alors suivre la règle du *catch or specify*.

Attraper et traiter une exception Un grand intérêt de la méthode de bloc *try-catch* pour gérer les exceptions sous Java et de différencier clairement les parties de code algorithmiques et de gestion des erreurs. Le bout de code suivant montre comment englober un bloc de code ou d'appels de méthodes dans un `try`, et comment capter une possible exception dans un `catch`. Il faut noter que plusieurs types d'exceptions peuvent être lancées depuis un même bloc `try`, il convient alors de mettre plusieurs blocs `catch` à la suite, chacun traitant un type d'exception différent.

```
try {
  // code susceptible de lancer des exceptions de type TypeException
}
catch (TypeException e) {
  // code pour traiter une exception de type TypeException
}
```

1. Écrire un programme simple pour calculer la moyenne de nombre flottants rentrés en paramètre.
2. Capter l'exception qui est lancée lorsqu'on parse une chaîne de caractères qui ne correspond pas à un nombre flottant. Afficher un message d'erreur mais ne pas stopper le calcul.

Bloc `finally` Le mot-clé `finally` sert à définir un bloc de code qui sera toujours exécuté après un bloc `try`, même si aucune exception n'a été lancée. Ce bloc permet d'effectuer des opérations de nettoyage qu'il aurait éventuellement été nécessaire de dupliquer dans les parties `try` et `catch` : c'est par exemple un bon endroit pour fermer des flux qui ont été ouvert dans le `try` afin d'éviter toute fuite de ressources. Le bout de code suivant est alors à rajouter à la suite du code *try* précédent :

```
finally {
  // code toujours exécuté à la suite d'un try
}
```

Si l'action effectuée dans le `finally` peut lancer une exception, on peut englober ce *try-finally* dans un *try-catch*, ou simplement propager l'exception à la méthode appelante par `throws`.

2 i/o : entrées-sorties

Les flux Pour obtenir des données (des valeurs entières, flottantes, des caractères, des chaînes de caractères, etc...), un programme utilise un *flux de données* (un *stream*) sur une source (un fichier, le disque, le clavier, etc...). Les flux de données sont gérés en Java à l'aide de classes du package `java.io` qui permettent de manipuler les entrées-sorties (i/o = input-output) dans un format caractère ou dans un format binaire. Ces classes peuvent être regroupées en 4 catégories :

- les flux binaires pour lire et écrire octet par octet :
  - la classe `InputStream` et ses sous-classes (`FileInputStream`) pour la lecture
  - le classe `OuputStream` et ses sous-classes (`FileOutputStream`) pour l'écriture
- les flux de caractères pour lire et écrire caractère par caractère :
  - la classe `Reader` et ses sous-classes (`FileReader`, `BufferedReader`) pour la lecture
  - la classe `Writer` et ses sous-classes (`FileWriter`, `BufferedWriter`) pour l'écriture

L'écriture d'une chaîne de caractères dans un fichier peut par exemple se faire à l'aide de `FileWriter` de la manière suivante :

```
public static void main(String[] args) {
  FileWriter fileOut = new FileWriter("fichier.txt");
  fileOut.write("texte à entrer dans le fichier");
  fileOut.close();
}
```

1. Placé dans une classe (par exemple `TestWrite`), ce bout de code provoque une erreur à la compilation. Pourquoi ? Faire les corrections nécessaires.
2. Modifier le code pour qu'un appel à ce programme n'écrase pas les données déjà présentent dans le fichier.

Toujours fermer un flux Il est très important de toujours penser à fermer un flux qui a été ouvert dans un programme. Dans le cas contraire, cela pourrait provoquer des fuites mémoires et d'autres désagréments. Une bonne habitude de programmation consiste à écrire l'appel à `close()` juste après avoir ouvert un flux, histoire de ne pas l'oublier par la suite.

3. Imaginons que l'ouverture du fichier se déroule correctement, mais que pour une raison ou une autre l'écriture de la chaîne de caractère échoue. Le ligne de code qui ferme le flux ne sera alors pas exécutée. Modifier le code pour que le flux soit fermé dans tous les cas de figure.
4. Ajouter une méthode qui affiche à l'écran le contenu d'un fichier parcouru caractère par caractère : la méthode `read()` renvoie un entier qui correspond à la valeur Unicode [1] du caractère lu.

Flux standards Quand le programme est exécuté depuis une console, l'intéraction avec l'utilisateur se fait souvent via l'enviremment de ligne de commande. Pour y accéder, la plateforme Java supporte 3 flux standards : l'entrée standard `System.in`, la sortie standard `System.out`, et la sortie d'erreur `System.err`. Ces flux sont des flux binaires et non des flux de caractères, mais `System.out` et `System.err` sont définis comme des objets `PrintStream`, qui utilise des objets internes de flux de caractères et possèdent par conséquent certaines méthodes de flux de caractères. Par contre, `System.in` ne possède aucune caractéristique de flux de caractères, c'est pourquoi il faut envelopper (ou *wrapper*) l'entrée standard dans un `InputStreamReader` pour récupérer des caractères. Si on veut de plus que la lecture s'effectue de manière bufferisée (par séquence de caractères et non pas caractère par caractère) on peut wrapper le flux dans un `BufferedReader`. Par exemple, la lecture d'une ligne de caractère (jusqu'à un retour chariot) par l'entrée standard se demande par :

[1] Rappel : les caractère en Java sont codés en Unicode et ont une valeur comprise entre `'\0000'` (0) et `'\uffff'` (65535)

```
BufferedReader cin = new BufferedReader(new InputStreamReader(System.in));
String s = cin.readLine();
```

5. Écrire un programme qui prend en argument un nom de fichier et un nombre de lignes n. On écrira alors n lignes de texte rentrées par l'utilisateur dans le fichier spécifié.

3 Persistance des objets

Sérialisation L'interface `java.io.Serializable` permet de sauver des objets sous forme de séquence d'octets, puis de les reconstruire par la suite même si la machine virtuelle qui a été arrêté entre temps. Les deux règles à suivre pour rendre un objet sérialisable sont les suivantes :

- l'objet doit implémenter l'interface `Serializable` ou hériter d'une classe qui l'implémente
- les champs non-sérialisables (Thread, OutputStream, ...) ou que l'on ne souhaite pas retrouvé dans l'objet sérialisé doivent être marqué `transient`

Pour réaliser cette persistance, par exemple en sauvant l'objet dans un fichier, on utilise peut utiliser la classe `java.io.ObjetOutputStream`. Par exemple pour sauver un objet de la classe `MaClasse` qui implémente `Serializable` :

```
MaClasse obj = new MaClasse();
FileOutputStream fos = new FileOutputStream("monObjet.ser");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(obj);
oos.close();
```

Il s'agit d'un code concis qui ne prend pas en compte les exceptions qu'il faudra bien sur traiter par la suite. Puis pour reconstruire cet objet :

```
MaClasse obj = null;
FileInputStream fis = new FileInputStream("monObjet.ser");
ObjectInputStream ois = new ObjectInputStream(fis);
obj = (MaClasse)ois.readObject();
ois.close();
```

1. Reprendre la classe `Adresse` du TP2 et lui faire implémenter l'interface `Serializable`.
2. Faire 2 petits programmes pour tester la persistance d'un objet `Adresse` : le premier sauvegarde un objet `Adresse` dans un fichier, le second reconstruit l'objet `Adresse` à partir du même fichier.
3. On veut maintenant ajouter un attribut `digicode` à un objet `Adresse`. Effectuer les modifications nécessaires, sachant qu'on ne souhaite pas que la valeur de ce champ soit transmise lors d'un sérialisation.