Programmation Orientée Objet
IG2I, LE2 – 2018-2019

TP6 – Exceptions

Diego Cattaruzza, Philippe Kubiak, Maxime Ogier

Contexte et objectif du TP

Ah le nord ! Partie deux ! On disait qu'au nord ils boivent beaucoup de bière. Bon, toutes les caisses sont encore au dépôt. Elles attendent vos indications pour être livrées.

Dans ce TP, nous allons d'abord étudier le concept d'exception. Nous allons après mettre en œuvre des méthodes qui nous permettrons de lire les fichiers de données proposés par Monsieur Houblon : chaque jour il vous donnera un fichier texte avec les cordonnées des clients à livrer ainsi que leur demande. Nous aimons bien bosser, mais nous ne voulons pas rentrer à la main, chaque matin, la liste des clients à livrer : on préfère tout de même le faire d'une manière automatisée.

Ce TP permet d'illustrer les notions suivantes :

- la notion d'exception ;
- les blocs **try** et **catch** pour la gestion des exceptions ;
- la lecture d'un fichier de texte et l'écriture dans un fichier de texte à l'aide des classes **PrintWriter** et **FileReader**.

On s'attachera en particulier dans ce TP à commenter proprement le code et à garantir que le code fonctionne et respecte les spécifications données.

Bonnes pratiques à adopter

Pour réaliser ce TP, il vous est très fortement recommandé d'utiliser la Javadoc pour comprendre le fonctionnement des méthodes que vous appelez. Par ailleurs, il est aussi important que votre propre code soit commenté au format Javadoc (commentaire commençant par "`/**`" au-dessus des définitions des attributs et méthodes). Ceci vous permet (1) de spécifier rigoureusement le comportement d'une méthode au moment où vous l'implémentez, (2) de savoir ce que fait une méthode sans avoir besoin de regarder son code (qui d'ailleurs peut ne pas être accessible), et donc (3) de faciliter le partage et la réutilisabilité du code.

Créez vos classes en mettant toujours les attributs en premier, suivis par les constructeurs (avec le constructeur par défaut en premier). Après les constructeurs, insérez les accesseurs et les mutateurs de la classe. Faites suivre les accesseurs et les mutateurs par toutes vos méthodes.

1 Exceptions

Tout le monde sait que les programmes codés par les étudiants d'IG2I sont parfaits et sans la moindre erreur ! Pourtant, même lorsqu'un programme est au point, certaines circonstances *exceptionnelles* peuvent compromettre la poursuite de son exécution ; il peut s'agir par exemple de données incorrectes rentrées par l'utilisateur, ou encore de la rencontre d'une fin de fichier prématurée (alors qu'on a besoin d'informations supplémentaires pour continuer le traitement).

On pourrait toujours essayer d'examiner toutes les situations possibles au sein du programme (avec plein de conditions **if**, **else**, **switch**, **case**, ...) et donc prendre les décisions qui s'imposent chaque fois qu'une circonstance exceptionnelle survient.

Cependant, cette approche pose deux problèmes : (1) le programmeur risque d'omettre certaines situations ; et (2) les codes risquent d'être très difficiles à lire (même ceux des étudiants d'IG2I qui sont toujours très bien commentés sous format Javadoc !).

Java dispose d'un mécanisme très souple nommé *gestion d'exceptions*, qui permet à la fois : (1) de dissocier la détection d'une anomalie de son traitement ; (2) de séparer la gestion des anomalies du reste du code, donc de contribuer à la lisibilité des programmes.

En général, (1) les exceptions sont déclenchées par l'instruction **throw**, et (2) les instructions qui peuvent les déclencher sont mises dans un bloc appelé **try** et elles sont récupérées/traitées dans un bloc appelé **catch**.

Nous allons à présent nous familiariser avec ces instructions dans un premier exemple.

1.1 Comment déclencher une exception avec throw

Dans le projet **Livraison** commencé au TP précédent, considérons la classe **Client**. Nous savons que nous ne pouvons pas livrer à un client un nombre de caisses négatif. En effet, cela n'a pas de sens, et si nous appelons notre algorithme de construction de tournées avec des demandes négatives, les résultats aussi n'auront aucun sens ! De plus, pour livrer un client il faut lui apporter au moins une caisse de bière. On pourrait envisager de faire un traitement par défaut (comme nous l'avons déjà fait précédemment dans d'autres TPs). Mais le désavantage de ces traitements par défaut est que l'utilisateur ne se rend pas compte qu'il y a un problème.

Nous pouvons, au sein du constructeur, vérifier la validité des paramètres fournis et lorsque la quantité à livrer au client est incorrecte nous déclenchons une exception à l'aide de l'instruction **throw**. Nous créons donc (un peu artificiellement) une classe que nous nommerons **ErrQuantite**. Java impose que cette classe hérite de la classe **Exception**.

Question 1. Après avoir ouvert Netbeans, reprenez le code développé lors de la dernière séance. Créez un nouveau paquetage **mesExceptions** (qui contiendra nos exceptions), et écrivez une classe **ErrQuantite** qui représente l'exception que l'on vient de décrire. Cette classe héritera la classe **Exception**. Pour le moment elle est vide.

Pour lancer une exception de ce type au sein d'une méthode (ou d'un constructeur) de la classe **Client**, nous devons :

- indiquer que la méthode peut déclencher une exception : pour ce faire, on ajoute **throws ErrQuantite** dans l'en-tête de la méthode (avant l'accolade ouvrant le bloc contenant le corps de la méthode) ;
- déclencher l'exception de la façon suivante : **throw new ErrQuantite()**.

Question 2. Modifiez le constructeur de la classe **Client** de façon à préparer la méthode au déclenchement d'une exception de type **ErrQuantite**. Puis, déclenchez une exception lorsque vous détectez une quantité à livrer strictement inférieure à une caisse de bière.

Vous allez probablement voir des erreurs apparaître dans votre code. Ne vous inquiétez pas, c'est normal ! On réglera cela dans quelques instants.

1.2 Comment utiliser les exceptions

Voyons maintenant comment procéder pour gérer convenablement les éventuelles exceptions de type **ErrQuantite** qui peuvent être déclenchées. Pour ce faire nous devons :

- inclure dans un bloc particulier, dit bloc **try**, les instructions dans lesquelles on risque de voir déclenchée une telle exception (dans notre cas, tous les appels au constructeur de la classe **Client**) ;
- faire suivre ce bloc de la définition des différents gestionnaires d'exception ; chaque définition de gestionnaire est précédée d'un en-tête introduit par le mot-clé **catch** (comme si catch était le nom d'une méthode gestionnaire).

Question 3. Ajoutez une méthode principale (si vous ne l'avez pas déjà) dans la classe **Client**. Écrivez-y puis exécutez le code suivant :

```
1  try {
2     Client c1 = new Client(5, 5, 10);
3     System.out.println("Creation ok");
4     Client c2 = new Client(5, −5, 0);
5     System.out.println("Creation ok");
6  } catch (ErrQuantite ex) {
7     System.out.println("Erreur: quantité negative");
8     System.exit(−1);
9  }
```

Que se passe-t-il ? L'appel à la méthode **exit()** termine l'exécution. Par convention, une valeur non nulle est passée à la méthode **exit()** si une interruption anormale du code s'est produite.

Question 4. Corrigez votre code (dans toutes les classes où de nouveaux clients sont créés) en gérant les exceptions déclenchées par votre nouveau constructeur (attention : effacer les appels au constructeur élimine les erreurs, mais ce n'est pas la bonne réponse).

On peut remarquer que le bloc **catch** doit être contigu au bloc **try**, et il ne doit donc pas y avoir d'instructions entre les deux.

Par ailleurs, on peut noter que le gestionnaire d'exception est défini indépendamment des méthodes susceptibles de déclencher l'exception. Ainsi, à partir du moment où la définition d'une classe est séparée de son utilisation (ce qui est souvent le cas en pratique), il est tout à fait possible de prévoir un gestionnaire différent d'une utilisation à une autre de la même classe. Ainsi dans l'exemple proposé, on peut vouloir à certains endroits afficher un message avant d'arrêter l'exécution du programme, ou bien ne rien afficher ou encore tenter de trouver une solution par défaut (mettre une quantité positive). Ceci permet donc d'apporter beaucoup de souplesse à la gestion des exceptions dans votre code.

1.3 Propriétés de la gestion d'exception

L'exemple que vous venez de faire était illustratif, mais restrictif pour certaines raisons :

- le gestionnaire d'exception (le **catch**) ne reçoit aucune information ; il reçoit juste un objet qu'il n'utilise pas ; nous allons voir comment utiliser cet objet pour communiquer des informations au gestionnaire ;
- le gestionnaire d'exception se contentait d'interrompre le programme (appel à **exit()**) ; nous verrons qu'il est possible de poursuivre l'exécution.

L'objet fourni à l'instruction **throw** est récupéré par le gestionnaire d'exception sous la forme d'un argument. Le gestionnaire d'exception peut donc l'utiliser pour transmettre une information. Il suffit pour cela de prévoir les attributs appropriés dans la classe correspondante (on peut aussi y trouver des méthodes).

Question 5. Ajoutez dans la classe **ErrQuantite** un attribut de type entier nommé **quantite** (avec une visibilité privée, bien entendu). Ajoutez un constructeur de la classe **ErrQuantite** par donnée de la quantité. Ajoutez les accesseurs et les mutateurs nécessaires et une méthode **toString()**. Mettez également à jour les appels au constructeur de **ErrQuantite** dans le reste du code.

Question 6. Dans la méthode principale de la classe **Client**, modifiez le gestionnaire des exceptions pour imprimer des informations supplémentaires sur la raison de l'exception, notamment la valeur exacte de la quantité qui a provoqué le déclenchement de l'exception. Testez votre code. Que se passe-t-il si vous utilisez **System.err.println()** à la place d'utiliser **System.out.println()** ?

La classe **Exception** (dont hérite votre classe) dispose d'un constructeur à un argument de type **String**, et elle dispose également d'une méthode **getMessage()** qui renvoie la chaîne de caractères passée en paramètre du constructeur. La classe **ErrQuantite** possède donc cette méthode **getMessage()** (car **ErrQuantite** hérite de **Exception**). Pour en tirer pleinement parti, il suffit de prévoir dans le constructeur de votre classe exception **ErrQuantite** de faire un appel au constructeur de la classe mère **Exception** auquel on passe en argument une chaîne de caractères qui décrit l'erreur.

Question 7. Modifiez le constructeur de la classe **ErrQuantite** en ajoutant un argument de type **String** qui représente le message d'erreur à afficher, et en faisant appel au constructeur par donnée d'une **String** de la classe mère **Exception**. Testez le bon fonctionnement de cette modification. Pour cela, dans le gestionnaire d'exception **catch**, imprimez le message à l'aide de **System.out.println()** et de la méthode **getMessage()** de la classe **Exception**.

1.4 Poursuite de l'exécution

Dans tous les exemples précédents, le gestionnaire d'exception mettait fin à l'exécution du programme en appelant la méthode **System.exit(-1)**. Cela n'est pas une obligation : en fait, après l'exécution des instructions du gestionnaire, l'exécution se poursuit simplement avec les instructions suivant le bloc **catch** associé au bloc **try** dans lequel a été déclenchée l'exception.

Question 8. Pour bien comprendre que nous pouvons poursuivre l'exécution de notre programme même après le déclenchement d'une exception ; dans la méthode principale de la classe **Client**, effacez l'appel à **System.exit (-1)**. Après le bloc **catch** ajoutez un appel à la méthode **System.out.println()** avec un message. Testez votre code. Retrouvez-vous ce dernier message dans la console ?

Dans notre exemple de livraison, on pourrait imaginer d'imprimer un message d'erreur lorsqu'un client ne demande pas au moins une caisse de bière. Par contre on peut imaginer de continuer l'exécution si le nombre de caisses demandées est zéro (auquel cas on peut tout de même afficher un message pour informer l'utilisateur).

Question 9. Modifiez le gestionnaire d'exception dans la méthode principale de la classe **Client** de telle sorte qu'un message soit affiché lorsqu'une exception de type **ErrQuantite** est déclenchée. L'exécution du programme continue si le client demande zéro caisse, et elle se termine si le nombre de caisses est négatif. Testez le bon fonctionnement de votre code.

Si l'appel au constructeur de **Client** lève une exception et que l'exécution du code se poursuit, le client a-t-il été créé ?

Une des exceptions standards du langage Java et que vous avez certainement déjà rencontrée est l'exception **NullPointerException** lorsque par exemple vous tentez d'accéder à un attribut ou de faire appel à une méthode d'un objet non initialisé (qui vaut donc **null**).

Une autre exception standard du langage Java est l'exception **IOException** utilisée par les méthodes d'entrées/sorties. Elle est utilisée pour signaler des interruptions ou des échecs des opérations d'*input* ou d'*output* (I/O). On fera sa connaissance dans la section qui suit !

2 Flux et fichiers de texte

Au cours des précédentes séances de TP, nous avons affiché des informations dans la fenêtre console en utilisant la méthode **System.out.println()**. En fait, **out** est ce que l'on nomme un *flux de sortie*. En Java, la notion de flux est très générale puisqu'un flux de sortie désigne n'importe quel *canal* susceptible de recevoir de l'information sous forme d'une suite d'octets (regroupement de 8 bits codant une information) : console, périphérique d'affichage, fichier, ou une connexion à un site distant.

De façon comparable, il existe des flux d'entrée, c'est-à-dire des canaux délivrant de l'information sous forme d'une suite d'octets. Là encore, il peut s'agir d'un périphérique de saisie (votre clavier), d'un fichier, d'une connexion ou d'un emplacement en mémoire centrale.

Monsieur Houblon s'attend à avoir son planning de livraison bien imprimé dans un fichier : autrement il devrait se balader avec votre ordinateur pour examiner les informations imprimées dans votre console. De même, Monsieur

Houblon vous donnera un fichier texte avec tous les clients à visiter. Il ne serait pas très efficace de devoir rentrer à la main tous les clients à livrer avec leurs coordonnées et leur demande. Surtout que dans le Nord on peut imaginer devoir livrer des bières à pas mal de clients !

Java nous offre la possibilité d'automatiser l'écriture et la lecture des données sur des fichiers avec les classes **PrintWriter** et **FileReader**.

2.1 Écriture de texte dans un fichier

Nous allons voir comment nous pouvons écrire notre planning dans un fichier texte. Pour cela nous allons utiliser un objet de la classe **PrintWriter**. Ce nouvel objet sera chargé d'écrire les informations sur le fichier choisi.

La classe **PrintWriter** possède un constructeur par donnée du nom du fichier dans lequel on souhaite écrire. Par exemple, si votre fichier s'appelle `fichierSortie.txt`, l'initialisation d'un tel objet sera :

```
1  PrintWriter p = new PrintWriter("fichierSortie.txt");
```

Si le fichier n'existe pas, il est créé automatiquement. Par contre, s'il existe, son contenu est effacé. La création d'un objet **PrintWriter** peut lever une exception de type **IOException**. Parmi les méthodes de la classe **PrintWriter**, nous avons la possibilité d'utiliser les méthodes **print()** et **println()**. Leur comportement est similaire à celui des méthodes **print()** et **println()** que vous utilisez pour afficher des informations sur la console. Par ailleurs, la classe **PrintWriter** dispose d'une méthode **close()** qui ferme le flux de sortie et libère les ressources système associées. Il ne faudra donc pas oublier de faire appel à cette méthode lorsque l'écriture est terminée.

Le texte que vous souhaitez écrire dans le fichier n'est pas directement écrit dans le fichier, mais il est d'abord stocké dans un tampon. Le tampon est vidé lors de l'appel à la méthode **flush()** ou lors de la fermeture du flux d'écriture avec l'appel à la méthode **close()**. Seulement à ce moment là, vous verrez apparaître le texte dans le fichier.

Le format texte du planning souhaité par Monsieur Houblon est le suivant. Dans la première ligne, on peut y lire les informations générales du planning : nombre de tournées, distance totale parcourue et nombre total de caisses livrées. Après il souhaite avoir des informations sur chaque tournée. Pour chaque tournée, il souhaite avoir une première ligne explicative avec le nombre de clients visités, la longueur et le nombre de caisses livrées, ainsi que (sur les lignes suivantes) la liste des clients visités avec leurs coordonnées et la quantité demandée.

Question 10. Ajoutez dans la classe **Routage** une méthode **ecriturePlanning(String fichierSortie)**. Cette méthode initialise un objet de type **PrintWriter** avec le constructeur par donnée du nom du fichier et puis elle écrit le planning complet dans le fichier texte. Si vous avez bien codé vos classes, pour ce faire, vous utiliserez *seulement* les méthodes **toString** de la classe **Routage** et **print** (ou **println**) de la classe **PrintWriter**. La méthode **toString** de la classe **Planning** fait appel à la méthode **toString** de la classe **Tournee**, qui fait appel à la méthode **toString** de la classe **Client** (**Depot**), ...

Vous allez donc modifier les méthodes **toString** des classes **Planning, Tournee**, **Client** et **Point** pour donner à Monsieur Houblon le résultat souhaité. Vous devez aussi gérer l'exception déclenchée par le constructeur de la classe **PrintWriter**.

2.2 Lecture d'un fichier

Pour lire des informations depuis un fichier nous pouvons utiliser la classe **FileReader**. Un objet de cette classe est initialisé en lui passant en paramètre le nom du fichier à lire :

```
1  FileReader f = new FileReader("fichierEntree.txt");
```

La classe **FileReader** est basique, et avec elle, nous ne pouvons lire que des caractères et aussi savoir si la fin du fichier est atteinte ou non. En couplant l'utilisation de la classe **FileReader** avec la classe **BufferedReader** nous pouvons utiliser les méthodes propres à cette dernière pour nous faciliter la vie : **BufferedReader** dispose d'une méthode **readLine**, pour lire chacune des lignes de notre fichier. Nous devons donc seulement nous occuper de traiter les informations de chaque ligne. Nous créons un objet de type **BufferedReader** de la façon suivante:

```
1  FileReader f = new FileReader("fichierEntree.txt");
2  BufferedReader br = new BufferedReader(f);
```

Nous pouvons lire les lignes du fichier avec le code suivant :

```
1  String ligne = br.readLine();
2  while(ligne != null) {
3     traitementLigne(ligne);
4     ligne = br.readLine();
5  }
```

Lors de la création d'un objet de type **FileReader**, une exception de type **IOException** peut être levée. De plus, la méthode **readLine()** de la classe

BufferedReader déclenche une exception de type **FileNotFoundException**. Cette exception est déclenchée lorsque le fichier que vous souhaitez ouvrir n'est pas trouvé.

Comme pour l'objet **PrintWriter**, les objets de type **FileReader** et **BufferedReader** possèdent des méthodes **close()** qui permettent de fermer les flux d'entrée et de libérer toutes les ressources système associées. Il ne faut donc pas oublier de faire appel à ces méthodes lorsque la lecture du fichier est terminée.

Question 11. Créez une méthode **traitementLigne(String ligne)** dans la classe **Routage**. Pour le moment, cette méthode affiche dans la console la ligne en paramètre de la méthode.

Question 12. Créez une méthode **lectureClient(String fichierEntree)** dans la classe **Routage**. Cette méthode initialise un objet de type **BufferedReader** et prépare la lecture du fichier **fichierEntree**. Ainsi, la méthode parcourt toutes les lignes du fichier et fait appel à la méthode **traitementLigne()**.

Dans cette méthode vous devez traiter deux exceptions : **FileNotFoundException** et **IOException**. Vous avez le droit de faire suivre un bloc **catch** par un autre bloc du même type.

Mettez dans le dossier de votre projet un fichier de type texte (`test.txt` par exemple) dans lequel vous avez écrit quelques lignes de texte. Réussissez-vous à lire ce fichier ?

Monsieur Houblon vous dit qu'il vous donnera chaque jour un fichier texte dont la première ligne contient la capacité de ses véhicules (nombre de type **int**). Puis, il y aura une ligne pour chaque point du réseau routier utilisé pour la livraison. La première de ces lignes (donc la deuxième du fichier) contient deux nombres de type **double**. Ces nombres sont les coordonnées du dépôt. Les informations dans les autres lignes seront pour les clients (un client sur chaque ligne). Les deux premiers nombres (de type **double**) sont respectivement l'abscisse et l'ordonnée du client, le troisième (de type **int**) est la demande. Il y a une tabulation entre deux valeurs de chaque ligne.

Question 13. Modifiez la méthode **traitementLigne** pour qu'elle puisse bien lire les données du fichier : capacité du véhicule, coordonnées et la demande de chaque client. Lors de la lecture de la première ligne, vous allez initialiser la valeur de la capacité du véhicule. Pensez alors bien à modifier en conséquence l'attribut de type **Planning** afin qu'il ait la valeur correcte de la capacité du véhicule. À la lecture des lignes suivantes, vous allez créer un objet du type **Depot** ou **Client**. Dans ce dernier cas, vous ajouterez le client à l'ensemble de clients **mesClients**.

Pour lire les nombres présents dans une ligne, vous pouvez utiliser la méthode **split(String val)** de la classe **String**. Cette méthode divise la chaîne de caractères par rapport à la chaîne de caractères **val** passée en paramètre. Elle renvoie les différents morceaux de la chaîne dans un tableau de **String**. Testez le bon fonctionnement de vos méthodes de lecture sur le fichier `testLecture0.txt` que vous trouvez sur Moodle.

Question 14. Lisez le fichier `testLecture1.txt` disponible sur Moodle. Trouvez-vous des erreurs de frappe de la part de Monsieur Houblon?

Question 15. Ajoutez dans la classe **Routage** une méthode statique **public static void testLecture(String nomFichier)**. Cette méthode crée un objet de type **Routage**, et lit le fichier `nomFichier` pour créer les clients à visiter et puis elle initialise les routes du réseau. Puis, vous pouvez faire appel à la méthode **planificationBasique()** afin de réaliser la planification. Enfin, écrivez la solution obtenue dans un fichier nommé `sol_` suivi de `nomFichier`.

Testez cette méthode avec les fichiers `test1.txt`, `test2.txt`, `test3.txt`, `test4.txt` et `test5.txt` disponibles sur Moodle.

References

[1] Delannoy, C., *Programmer en Java, Eyrolles, 2014*