Programmation Orientée Objet
IG2I, LE2 – 2023-2024

TP6 – Exceptions

Diego Cattaruzza, Maxime Ogier, Pablo Torrealba

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. Dans un second temps, nous compléterons le modèle objet commencé à la séance précédente en y intégrant la modélisation des tournées de véhicules.

Ce TP permet ainsi de revoir les notions suivantes :

- l'interface **List** et ses implémentations (**ArrayList** et **LinkedList**).

Ce TP permet d'illustrer les notions suivantes :

- la notion d'exception ;
- les blocs **try** et **catch** pour la gestion des exceptions.

Bonnes pratiques à adopter

- Testez votre code au fur et à mesure de votre avancement.
- Respectez les conventions de nommage Java.
- Faites des indentations de manière correcte.
- Donnez des noms compréhensibles et pertinents à vos variables, attributs, classes, méthodes.
- Respectez autant que possible le principe d'encapsulation.
- Utilisez des structures de données adaptées.
- Faites des méthodes courtes avec peu de niveaux d'indentation.
- Ne faites pas de duplication de code, mais privilégiez le code réutilisable.
- Utilisez les outils de votre IDE pour faire du *refactoring* (clic-droit puis '*Refactor*' sur IntelliJ IDEA).
- Utilisez la Javadoc pour comprendre le fonctionnement des méthodes que vous appelez.
- Quand c'est nécessaire, commentez votre propre code de façon pertinente au format Javadoc (commentaire commençant par `"/**"` au-dessus des définitions des attributs et 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.

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, et le programme continue son exécution comme si tout s'était passé normalement. Il n'est pas toujours souhaitable d'avoir ce type de comportement.

Nous proposons ici, au sein du constructeur de la classe **Client**, de vérifier la validité des paramètres fournis et lorsque la quantité à livrer au client est incorrecte nous déclenchons alors une exception.

Le langage Java dispose d'une classe **Exception** qui est la classe mère de tous les types d'exception qui peuvent survenir dans un programme. Cette classe permet de garder en mémoire des informations comme la pile des appels qui a mené au déclenchement de l'exception, ou bien des détails sur l'exception déclenchée. Remarquez que certains types d'exceptions existent déjà dans le langage Java (vous avez déjà dû voir **NullPointerException** par exemple). Mais il est également possible de créer notre propre type d'exception. Pour cela, il suffit de créer une classe qui hérite de **Exception**.

Question 1. Après avoir ouvert IntelliJ IDEA, reprenez le code développé lors de la dernière séance. Créez un nouveau paquetage **exceptions** (qui contiendra vos types d'exceptions), et ajoutez une classe **ExceptionQuantite** qui représente l'exception que l'on vient de décrire. Cette classe héritera de la classe **Exception**. Pour le moment elle est vide, nous reviendrons la compléter plus tard.

Dans le corps d'une méthode (ou d'un constructeur), si l'on remarque une anomalie, au lieu de chercher un traitement par défaut, il est possible de déclencher une exception en utilisant l'instruction **throw** suivie d'un objet de type **Exception** (ou d'une classe qui hérite de **Exception**). L'appel à cette instruction arrête alors le déroulement de la méthode et renvoie directement l'objet de l'instruction **throw** au lieu de renvoyer un objet du type de retour de la méthode. Par ailleurs, dans ce cas, il est nécessaire d'indiquer dans la signature de la méthode qu'elle peut potentiellement renvoyer un certain type d'exception. On utilise pour cela le mot clé **throws** (faites bien attention au 's' final).

Voici un exemple de code dans lequel la méthode **methodeAvecException** peut lever une exception de type **ExceptionType** :

```
1  public int methodeAvecException(double val) throws ExceptionType {
2      if(val < 0) { // test sur la valeur de val
3          ExceptionType ex = new ExceptionType();
4          throw ex;
5      } else {
6          // deroulement normal de la methode
7          // ...
8          return ...; // retourne une valeur entiere
9      }
10 }
```

Question 2. Modifiez le constructeur de la classe **Client** de façon à déclencher une exception de type **ExceptionQuantite** si la quantité à livrer passée en paramètre est 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 gérer le déclenchement d'une exception

Voyons maintenant comment procéder pour gérer convenablement les éventuelles exceptions qui peuvent être déclenchées lors de l'appel d'une méthode. Tout d'abord, le langage Java ne permet pas d'appeler une méthode qui déclare dans sa signature qu'elle peut lever une exception (avec le mot clé **throws**) sans faire un "traitement particulier". Pour que le code compile, il faut choisir parmi deux possibilités, comme présenté par la suite.

1. La méthode appelante déclare elle-même qu'elle peut potentiellement déclencher une exception. Par exemple, on pourrait avoir le code suivant :

```
1  public void methodeAppelante() throws ExceptionType {
2      // ... debut de la methode
3
4      double val = ...;
5      int res = methodeAvecException(val);
6
7      // ... fin de la methode
8  }
```

2. La méthode appelante fait l'appel à une méthode qui peut déclencher une exception au sein d'un bloc **try**. Ce bloc **try** est immédiatement suivi d'un bloc **catch**, appelé *gestionnaire d'exception* dans lequel on définit ce qu'il se passe si une exception a été déclenchée. Ce gestionnaire d'exception prend en paramètre un objet du type d'exception qui peut être déclenchée dans le bloc **try**. Par exemple, on pourrait avoir le code suivant :

```
1  public void autreMethodeAppelante() {
2      // ... debut de la methode
3
4      try {
```

```
5         // ...
6         double val = ...;
7         int res = methodeAvecException(val);
8         // ...
9      } catch (ExceptionType ex) {
10        // si une exception est declenchee a la ligne 7
11        // on arrive directement dans ce bloc catch
12        // on indique ce qu'on fait dans ce cas la
13     }
14
15     // ... fin de la methode
16  }
```

Par ailleurs, notez bien que si on choisit la première option, il faudra ensuite se poser la même question pour le traitement de l'exception déclenchée part la méthode appelante. Donc à un certain moment, il faudra bien gérer l'exception dans un gestionnaire d'exceptions. De plus, tout l'intérêt de ce mécanisme de gestion des exceptions est sa souplesse. La méthode **methodeAvecException** peut être appelée à plusieurs endroits dans le code, et à chaque fois on pourrait utiliser un gestionnaire d'exception différent, dans lequel on ferait un traitement différent, même si c'est la même exception qui a été déclenchée. Ainsi, 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. Ceci permet donc d'apporter beaucoup de souplesse à la gestion des exceptions dans votre code. Imaginez à quoi ressemblerait la méthode **methodeAvecException** si elle ne déclenchait pas l'exception, et qu'elle devait la gérer avec différents cas.

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 (ExceptionQuantite 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).

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 **ExceptionQuantite** un attribut de type entier nommé **quantite**. Ajoutez un constructeur de la classe **ExceptionQuantite** 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 **ExceptionQuantite** dans le reste du code.

Question 6. Dans la méthode principale de la classe **Client**, modifiez le gestionnaire d'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.

La classe **Exception** (dont hérite votre classe) dispose d'un constructeur qui prend en argument un objet de type **String** qui représente un message lié à l'exception. La classe **Exception** dispose également d'une méthode **getMessage()** qui renvoie ce message.

Question 7. Modifiez le constructeur de la classe **ExceptionQuantite** en ajoutant un argument de type **String** qui représente le message lié à l'exception.

Testez le bon fonctionnement de cette modification. Pour cela, dans le gestionnaire d'exception **catch**, vous pouvez imprimez le message de l'exception sur la console.

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 **ExceptionQuantite** 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 le TP suivant !

2 Modélisation du planning de livraisons

À présent, nous reprenons la modélisation du problème de tournées. Nous avons mis en place tout ce qui est nécessaire à la gestion du dépôt et des clients. Nous allons à présent nous intéresser aux tournées qui contiennent les clients.

2.1 Création de la classe Tournee

Nous avons maintenant tous les éléments pour modéliser le déplacement des véhicules. Nous allons donc créer des tournées. Une tournée est une suite de livraisons chez les clients qui part du dépôt et termine au dépôt. Elle est caractérisée par la distance parcourue (depuis le départ du dépôt jusqu'au retour au dépôt) et le nombre de caisses de bière livrées. De plus, une tournée est effectuée par un camion qui a une capacité limitée (en nombre de caisses). Une tournée est donc également caractérisée par une capacité. Le nombre de caisses livrées dans la tournée ne doit pas dépasser la capacité.

Question 10. Ajoutez un paquetage **planning**. Ajoutez-y une nouvelle classe **Tournee**. Ajoutez ses attributs, parmi lesquels vous aurez un attribut de type **Depot** et un attribut qui représente les clients à livrer lors de cette tournée. Justifiez le type de collection utilisé pour ce dernier attribut. Ajoutez un constructeur par données du dépôt et de la capacité. Ajoutez les accesseurs et mutateurs nécessaires.

Question 11. Ajoutez une méthode **public boolean ajouterClient(Client client)** qui ajoute le client dans la collection des clients livrés (on suppose que ce client est ajouté à la fin de la tournée, avant de retourner au dépôt). Cette méthode renvoie **true** si l'ajout a bien eu lieu, et **false** sinon. Notez bien qu'il n'est pas possible d'ajouter un client si cela engendre un dépassement de la capacité. Cette méthode doit mettre à jour la distance de la tournée ainsi que le nombre de caisses livrées. Faites attention au fait qu'une tournée commence et termine toujours au dépôt. Il faut donc considérer les routes qui vont du dépôt au premier client et du dernier client au dépôt. N'hésitez pas à faire un dessin.

Question 12. Ajoutez une méthode **public boolean ajouterClient(List<Client> clients)** qui ajoute tous les clients de la liste **clients** à la tournée (en mettant à jour la distance parcourue et la quantité livrée). Cette méthode renvoie **true** si tous les ajouts ont bien eu lieu, et **false** sinon (certains clients n'ont pas pu être ajoutés).

Question 13. Ajoutez une méthode **toString()** qui renvoie une chaîne de caractères avec les informations sur la tournée. Ajoutez également une méthode principale dans la classe. Testez votre classe avec le code suivant :

```
1  Depot d1 = new Depot(0, 0);
2  Client c1 = new Client(5, 5, 10);
3  Client c2 = new Client(−5, 5, 10);
4  Client c3 = new Client(−5, −5, 10);
5  Client c4 = new Client(5, −5, 10);
6  Set<Point> ensPoint = new HashSet<>();
7  ensPoint.add(c1);
8  ensPoint.add(c2);
9  ensPoint.add(c3);
```

```
ensPoint.add(c4);
d1.ajouterRoutes(ensPoint);
ensPoint.add(d1);
c1.ajouterRoutes(ensPoint);
c2.ajouterRoutes(ensPoint);
c3.ajouterRoutes(ensPoint);
c4.ajouterRoutes(ensPoint);
List<Client> clients = new ArrayList<>();
clients.add(c1);
clients.add(c2);
clients.add(c3);
clients.add(c4);
Tournee t = new Tournee(d1, 50);
t.ajouterClient(clients);
System.out.println(t);
```

2.2 Planning

La brasserie que nous avons le plaisir d'aider a tellement de clients qu'un seul véhicule n'est pas suffisant pour livrer toutes les caisses de bière (car les véhicules ont une capacité limitée). On a donc besoin de créer plusieurs tournées. Pour ce faire on crée une nouvelle classe **Planning**. Ses attributs sont un ensemble de tournées, la distance totale parcourue, le nombre total de caisses de bières livrées, ainsi que la capacité des camions.

Question 14. Créez dans le paquetage **planning** une nouvelle classe **Planning** avec ses attributs et un constructeur par donnée de la capacité. Justifiez le choix de la collection que vous utilisez pour stocker les tournées. Ajoutez les accesseurs et les mutateurs nécessaires. Ajoutez une méthode **private boolean ajouterTournee(Tournee t)** qui ajoute une tournée à l'ensemble des tournées. Cette méthode mettra à jour les attributs pour la distance totale et le nombre total de caisses livrées. Ajoutez une méthode **toString()** qui renvoie une chaîne de caractères avec les informations sur le planning et une méthode principale pour tester la classe.

Question 15. Ajoutez dans votre classe **Planning** une méthode **public void planificationBasique(Depot depot, Set<Client> clients)**. Cette méthode commence par créer une tournée vide, que nous nommerons *tournée courante*. Puis, elle itère sur l'ensemble des clients et les ajoute à la *tournée courante*. Lorsque l'ajout d'un client à la *tournée courante* n'est pas possible, alors la *tournée courante* est ajoutée au planning (avec la méthode **ajouterTournee**). Puis une nouvelle *tournée courante* vide est créée et le client qui n'avait pas pu être ajouté précédemment lui est ajouté. Il suffit ensuite de réitérer ce processus. On s'assure bien à la fin que la dernière *tournée courante* est ajoutée au planning.

References

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