# Guilde AOC

Objectif

L'objectif de la guilde AoC est l'amélioration continue de chacun via la résolution d'exercices disponibles sur le site <https://adventofcode.com/>.

Dans l'idée, à chaque séance les participants sont censés arriver et avoir déjà résolu l'exercice du jour. Ensuite, les différentes solutions sont lues et expliquées, et on essaie d'en tirer le meilleur afin d'avoir une solution optimisée et respectant les normes de développement (nommage, découpage, commentaires, etc.).

Déroulé de la séance

Pendant une séance, trois parties sont prévues.

* Correction de l'exercice du jour : Durant 1h environ, les membres de la guilde vont comparer leurs solutions (si elles sont différentes), et les points d'améliorations seront indiqués. Il est toujours bien de présenter sa solution si elle diffère des autres, afin que les différentes implémentations puissent être présentées.
* Résumé de la séance : Un temps est pris pour faire un résumé de ce qui a été appris durant la séance. Ceci a pour but de garder une trace des séances et d'obtenir une vue des grands axes d'amélioration possibles au sein de l'équipe.
* Explication de l'exercice suivant : Le reste du temps est dédié à la lecture et à l'explication de l'exercice suivant, afin de pouvoir donner à chacun les clés qui lui manqueraient pour le résoudre. Durant ce temps, les questions visant à mieux intégrer les données de l'exercices sont les bienvenues.

Rejoindre l'équipe

Pour rejoindre le leaderboard des participants, il suffit de vous authentifier sur le site de l'Advent of Code, d'aller dans la section private leaderboard, et de renseigner le code suivant :

1718617-89c0b478

L'invitation à la réunion récurrente de la guilde se trouve dans les fichiers, onglet en haut à gauche.

Echanger ensemble

Vous pouvez paramétrer votre compte AoC pour que votre GitHub soit accessible (en lecture bien entendu, pas de panique !) depuis votre profil, ce qui permet de simplement accéder aux solutions réalisées par chacun.

Comptes Git perso / pro

Afin d'éviter de faire des commit sur AOC avec votre email pro et des commits sur le git Geodis avec votre email perso, nous vous conseillons de bien configurer vos adresses mail et logins. Ainsi, nous vous conseillons de mettre cette config globale :

git config --global user.name [VOTRE USERNAME GEODIS]

git config --global user.email [VOTRE EMAIL GEODIS]

Et **dans votre projet AoC** :

git config --local user.name [VOTRE USERNAME AOC]

git config --local user.email [VOTRE EMAIL AOC]

# Exercices par séance

La plupart des exemples de solution fonctionnelles se trouvent ici sur les repos suivants :

<https://github.com/Manerial/AdventOfCode/blob/develop/src/exercises/>

<https://github.com/Jueverhard/AdventOfCode/tree/develop/src/main/java/calendar/year>

|  |  |  |
| --- | --- | --- |
| Date de la séance | Enoncé de l'exercice | Exemple de solution fonctionnelle |
| 25/04/2023 | [Day 3 - Advent of Code 2018](https://adventofcode.com/2018/day/3) |  |
| 18/04/2023 | [Day 3 - Advent of Code 2017](https://adventofcode.com/2017/day/3) | [Exemple pour 2017 jour 3](https://github.com/Manerial/AdventOfCode/tree/develop/src/exercises/aoc2017/day03) |
| 11/04/2023 | [Day 3 - Advent of Code 2016](https://adventofcode.com/2016/day/3) | [Exemple pour 2016 jour 3](https://github.com/Manerial/AdventOfCode/tree/develop/src/exercises/aoc2016/day03) |
| 04/04/2023 | [Day 3 - Advent of Code 2015](https://adventofcode.com/2015/day/3) | [Exemple pour 2015 jour 3](https://github.com/Manerial/AdventOfCode/tree/develop/src/exercises/aoc2015/day03) |
| 28/03/2023 | [Day 2 - Advent of Code 2021](https://adventofcode.com/2021/day/2) | [Exemple pour 2021 jour 2](https://github.com/Manerial/AdventOfCode/tree/develop/src/exercises/aoc2021/day02) |
| 21/03/2023 | [Day 2 - Advent of Code 2020](https://adventofcode.com/2020/day/2) | [Exemple pour 2020 jour 2](https://github.com/Manerial/AdventOfCode/tree/develop/src/exercises/aoc2020/day02) |
| 14/03/2023 | [Day 2 - Advent of Code 2019](https://adventofcode.com/2019/day/2) | [Exemple pour 2019 jour 2](https://github.com/Manerial/AdventOfCode/tree/develop/src/exercises/aoc2019/day02) (JHE)  [Exemple pour 2019 jour 2](https://github.com/Jueverhard/AdventOfCode/tree/develop/src/main/java/calendar/year/_2019/day02) (JEV) |
| 07/03/2023 | [Day 2 - Advent of Code 2018](https://adventofcode.com/2018/day/2) | [Exemple pour 2018 jour 2](https://github.com/Manerial/AdventOfCode/tree/develop/src/exercises/aoc2018/day02) |
| 28/02/2023 | [Day 2 - Advent of Code 2017](https://adventofcode.com/2017/day/2) | [Exemple pour 2017 jour 2](https://github.com/Manerial/AdventOfCode/tree/develop/src/exercises/aoc2017/day02) |
| 21/02/2023 | [Day 2 - Advent of Code 2016](https://adventofcode.com/2016/day/2) | [Exemple pour 2016 jour 2](https://github.com/Manerial/AdventOfCode/tree/develop/src/exercises/aoc2016/day02) |
| 14/02/2023 | [Day 2 - Advent of Code 2015](https://adventofcode.com/2015/day/2) | [Exemple pour 2015 jour 2](https://github.com/Manerial/AdventOfCode/tree/develop/src/exercises/aoc2015/day02) |
| 08/02/2023 | [Day 1 - Advent of Code 2020](https://adventofcode.com/2020/day/1) | [Exemple pour 2020 jour 1](https://github.com/Manerial/AdventOfCode/blob/develop/src/exercises/aoc2020/day01) |
| 31/01/2023 | [Day 1 - Advent of Code 2019](https://adventofcode.com/2019/day/1) | [Exemple pour 2019 jour 1](https://github.com/Manerial/AdventOfCode/blob/develop/src/exercises/aoc20/day01) |
| 24/01/2023 | [Day 1 - Advent of Code 2018](https://adventofcode.com/2019/day/1) | [Exemple pour 2018 jour 1](https://github.com/Manerial/AdventOfCode/blob/develop/src/exercises/aoc20/day01) |
| 17/01/2023 | [Day 1 - Advent of Code 2017](https://adventofcode.com/201/day/1) | [Exemple pour 2017 jour 1](https://github.com/Manerial/AdventOfCode/blob/develop/src/exercises/aoc201/day01) |
| 10/01/2023 | [Day 1 - Advent of Code 2016](https://adventofcode.com/201/day/1) | [Exemple pour 2016 jour 1](https://github.com/Manerial/AdventOfCode/blob/develop/src/exercises/aoc2016/day01) |
| 03/01/2023 | [Day 1 - Advent of Code 2015](https://adventofcode.com/201/day/1) | [Exemple pour 2015 jour 1](https://github.com/Manerial/AdventOfCode/blob/develop/src/exercises/aoc2015/day01) |

## Session du 25/04/2023

### Résumé de l'exercice partie 1 :

* Pour cet exercice, il faudra travailler sur une grille carrée de 1000 unités de côté.
* L’input contient une liste de rectangles sous la forme #ID @ X,Y: AxB.
* Voici le détail d’une ligne d’input :
  + Une ligne correspond à un rectangle.
  + #ID correspond à l’identifiant du rectangle.
  + X et Y correspondent aux coordonnées du rectangle.
  + AxB (A fois B) correspond à la largeur et à la hauteur de ce rectangle.
* Chaque rectangle couvre un certain nombre de cases de la grille, et il peut y avoir des chevauchements entre deux rectangles.
* Les rectangles sont pleins, leur intérieur compte dans le chevauchement.
* Le but de l’exercice est de trouver combien de cases de la grille sont couvertes par au moins deux rectangles.

### Résumé de l'exercice partie 2 :

* Dans cette deuxième partie, il faut trouver quel est le seul #ID de rectangle qui ne se chevauche avec aucun autre.

### Ce qui a été vu en séance :

* Pour faire un collectors.toMap, on peut utiliser Object.identity() pour récupérer le type de l’objet.
* Les variables uniquement utilisées à l’intérieur d’une boucle devraient être définies dans cette boucle.
* Lors de l’utilisation d’un stream, chaque appel de méthode sur le stream lui-même devrait retourner à la ligne afin de gagner en lisibilité.
* Si un stream appelle un lambda sur plusieurs lignes, il est peut-être nécessaire d’extraire ce lambda dans une fonction.
* Pour faire un map() sur une condition négative, on peut utiliser Predicate.not()
* @Builder permet d’obtenir un constructeur vide par défaut, utilisable avec les méthodes fournies dans la fonction builder() générée dans l’objet. Exemple :
  + object.builder().varA(4).varB("toto").build();

## Session du 18/04/2023

### Résumé de l'exercice partie 1 :

* Le principe de cet exercice est de trouver la distance entre deux positions dans une grille théoriquement infinie.
* Chaque cellule de la grille possède un index auto-incrémenté défini par un motif en spirale. La position de départ possède l'index 1.
* Voici à quoi ressemble la grille :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 17 | 16 | 15 | 14 | 13 |
| 18 | 5 | 4 | 3 | 12 |
| 19 | 6 | 1 | 2 | 11 |
| 20 | 7 | 8 | 9 | 10 |
| 21 | 22 | 23 | 24 | ... |

* L'input de l'exercice correspond à un des index du tableau.
* Il faut trouver la [distance de Manhattan](https://fr.wikipedia.org/wiki/Distance_de_Manhattan) de cet index par rapport à la position initiale (index 1).

### Résumé de l'exercice partie 2 :

* Dans la seconde partie, chaque cellule est calculée par rapport à ses voisines (diagonales comprises), selon la même spirale que dans l'exercice précédent.
* Ainsi, chaque valeur de cellule est égale à la somme de ses voisines.
* La première position a comme valeur 1.
* Voici à quoi ressemble la grille :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 147 | 142 | 133 | 122 | 59 |
| 304 | 5 | 4 | 2 | 57 |
| 330 | 10 | 1 | 1 | 54 |
| 351 | 11 | 23 | 25 | 26 |
| 362 | 747 | 806 | 854 | ... |

* Cette fois, la réponse à l'exercice est la première valeur calculée qui dépasse la valeur de l'input. Par exemple, si l'input est 100, la solution sera 122.

### Ce qui a été vu en séance :

* Penser à écrire de la javadoc quand on fait une fonction complexe, afin qu'un lecteur puisse comprendre rapidement ce qu'elle fait.
  + C'est d'autant plus important quand la fonction n'a pas de nom assez explicite.
  + Javadoc :
    - Pour rédiger facilement de la javadoc, au-dessus d'une déclaration de fonction, taper /\*\* + entrer.
    - @Param whateverParam commentaire permet de définir un paramètre.
    - @Return commentaire permet d'expliciter ce à quoi correspond la valeur retournée.
    - @Throw whateverException1, whateverException2 commentaire permet d'expliquer dans quel cas on throw quelle exception.
* Dans un Enum, les valeurs sont ordonnées et accessible via MonEnum.values()
* L'annotation @Data de Lombok redéfinit les fonctions hashcode() et equals(), il n'est pas nécessaire de les surcharger.
* Quand on fait une fonction qui "fait des maths", il est toujours nécessaire d'expliquer ce qu'elle fait et de bien la découper.
  + Les informaticiens ne sont pas des mathématiciens, et tout le monde ne réfléchit pas de la même manière.
* Quand on a une idée spécifique de ce que représente une variable, mettre un commentaire pour expliciter afin que tout le monde puisse avoir les mêmes bases.
* FAITES DES FONCTIONS !!!
* Solutions "Mathématiques" : 4/6.

NB : En guilde AOC, il est préférable d'avoir juste une partie 1 qui reprenne ce qui a été vu en guilde AOC plutôt que les deux parties qui ne prennent pas en compte les recommandations.  
C'est une guilde d'amélioration continue dont l'objectif est en partie de voir les différentes solutions de chacun, et en partie de s'améliorer côté programmation pure.

## Session du 11/04/2023

### Résumé de l'exercice partie 1 :

* Le but de l'exercice est de prendre des valeurs de longueurs et d'indiquer si elles peuvent se retrouver dans un triangle.
* Par exemple, un triangle de 1 x 2 x 6 est impossible, quand un triangle de 3 x 4 x 6 est possible.
* Dans l'input, chaque ligne contient trois valeurs représentant ces longueurs.
* Il faut retourner le nombre de triangles valides à la fin de l'exercice.

### Résumé de l'exercice partie 2 :

* Dans cette seconde partie, l'input ne doit plus être lu en ligne mais en colonne. Ainsi, pour le premier triangle, les valeurs à lire seront les premières de la ligne 1, 2 et 3.

### Ce qui a été vu en séance :

* ParallelStream est identique à Stream, mais plus performant au-dessus de 1000 lignes.
  + <https://www.axopen.com/blog/2014/05/java-8-stream-parallelstream-performance-string/>
* L'utilisation du trim() avant un split() permet de nettoyer une chaine de caractères de ses caractères blancs superflu.
* Séparer les appels de fonction dans des variables au fur et à mesure afin de simplifier la lecture de code.
* Utilisation de constantes quand une variable (nombre ou chaîne de caractères) est répétée plusieurs fois.
  + Attention à ne pas mettre de constante pour deux valeurs qui ne sont pas liées !
* Attention à bien attribuer à chaque constructeur la responsabilité qui lui est due.
  + Si on veut faire un triangle aux valeurs triées par grandeur, le faire dans le constructeur permet de simplifier d'autres fonctions.

## Session du 04/04/2023

### Résumé de l'exercice partie 1 :

* Pour cet exercice, il faudra travailler sur une grille théoriquement infinie.
* Il y a un pointeur dont la position de départ est 0-0.
* Une chaîne de caractère, contenant les caractères suivants : ^v<>, indiquera les directions à suivre.
* Chaque fois que le pointeur passe sur une cellule, la valeur de la cellule augmente de 1.
* La position initiale est incrémentée également.
* Il faut, pour terminer, compter le nombre de cellules dont la valeur est supérieure ou égale à 1.

### Résumé de l'exercice partie 2 :

* Cette fois, il y a deux pointeurs.
* Chaque instruction fait bouger alternativement les deux pointeurs.
* Idem que pour l'exercice partie 1, il faudra donner le nombre de cellules dont la valeur est >= 1.

### Ce qui a été vu en séance :

* Exception : lancer une exception spécialisée.
* On peut même créer notre propre exception.
* Les exceptions sont utiles, notamment pour le cas default d'un switch.
* Utiliser l'interface List plutôt que la Classe Spécifique ArrayList (pareil pour Map et Hashmap). Il est important de renvoyer et d'utiliser les bons niveaux d'interface. Un type trop spécialisé peut-être trop restrictif tandis qu'un type trop général peut créer des erreurs et nous empêcher d'utiliser des implémentations efficaces.
* Attention, la fonction contains() des listes utilisent le equals() qui est à disposition et donc faire des comparaisons par référence.
* Attention lors de la comparaison d'objet. Lorsqu'on n’implémente pas soi-même la fonction equals(), la comparaison se fera par référence au lieu de se faire par valeur. Cela peut provoquer des faux positifs au niveau de la vérification d'égalité.

## Session du 28/03/2023

### Résumé de l'exercice partie 1 :

* L'objectif est de calculer la position finale d'un sous-marin en fonction d'une série d'instructions :
  + up X : diminue la profondeur de X,
  + down X : augmente la profondeur de X,
  + forward X : augmente la position horizontale de X.
* Au départ, profondeur et position horizontale sont de 0
* Le résultat de l'exercice sera la profondeur finale multipliée par la position finale.

### Résumé de l'exercice partie 2 :

* Cette fois, une variable supplémentaire vient s'ajouter : la cible (qui représente un objectif, une direction, un point visé).
* A partir de maintenant, les directives up et down influencent la cible et non plus directement la profondeur.
* La directive forward, en plus de modifier la position horizontale, va maintenant modifier la profondeur selon le calcul suivant :
  + profondeur = profondeur + cible \* X.
* Le résultat de l'exercice sera la profondeur finale multipliée par la position finale.

### Ce qui a été vu en séance :

* Le mot clé "super" permet d'appeler les fonctions de la classe parent.
* Pour créer un objet qui hérite d'un parent, il faut appeler this.super() pour faire appel au constructeur parent.
* Attention à ne pas sur complexifier le code.
* Lors d'un héritage, les variables peuvent se noter "protected" pour être accessibles par les héritiers.
* Attention à ne pas conserver du "code mort".

## Session du 21/03/2023

### Résumé de l'exercice partie 1 :

* L'input de cet exercice représente un ensemble de couples (mot de passe, règle)
* L'objectif de l'exercice est de compter combien des mots de passe présentés correspondent à la règle qui leur est associée
* La règle pour cette partie consiste en un nombre d'occurrences d'un caractère dans le mot de passe

### Résumé de l'exercice partie 2 :

* Cette fois, la règle consiste en la présence d'un caractère à certains index du mot de passe.
* **Attention**: le caractère devra apparaître à **une et une seule** des positions.

### Ce qui a été vu en séance :

* Attention à la clarté des noms de variables :
  + Appellation des variables count et count2 pas top.
* Pour renommer une variable, soit faire :
  + droit > refactor > rename,
  + maj + f6 (sur intelliJ),
  + maj + ctrl + R (sur Eclipse).
* Attention à votre indentation : CTRL + ALT + L !
* Penser à utiliser un opérateur ternaire :
  + Si on a un if (C) {X += Y}, on peut le transformer en X += C ? Y : 0
* Si du code est dupliqué, penser à l'extraire dans une classe utilitaire si l'objet s'y prête
* Penser à avoir une structure de packages cohérente !
* Lors d'un split, on peut utiliser une regex pour splitter sur plusieurs caractères différents.
* Si on fait un split, il peut être intéressant de rappeler en commentaires un exemple de contenu de la chaîne à splitter, et un exemple de ce qu'on attend. Ça simplifie la relecture de code.
* Quand plusieurs informations sont liées entre elles, étudier la faisabilité d'un objet peut être intéressant.
* Il existe un opérateur, comme le AND && et le OR ||, pour exprimer un XOR. Il s'agit du caractère ^ (il va tout seul, ce n'est pas "^^").

## Session du 14/03/2023

### Résumé de l'exercice partie 1 :

* L'input de cet exercice représente la mémoire d'un ordinateur. Elle est segmentée en blocs d'instructions de 4 entiers :
  + Un [OPCODE], pour code opération : sa valeur détermine l'action à faire sur la suite.
  + Deux adresses mémoire d'entrée : c'est sur ces dernières que vont se dérouler l'exercice.
    - La première s'appelle le [NOM],
    - La seconde s'appelle le [VERBE].
  + Une adresse mémoire de sortie [OUTPUT] : c'est à cette adresse que doit être enregistré le résultat de l'opération.
* Pour chaque bloc d'instructions, vous devrez effectuer l'opération [OPCODE] sur les valeurs situées aux adresses mémoire [NOM] et [VERBE], et enregistrer le résultat à l'adresse mémoire [OUTPUT].
* Enfin, vous devrez afficher le résultat obtenu à l'adresse mémoire 0 (résultat de votre programme).
* Voici les opérations à effectuer pour chaque [OPCODE] :
  + 1 : addition des valeurs aux adresses [NOM] et [VERBE] à enregistrer à l'adresse [OUTPUT],
  + 2 : multiplication des valeurs aux adresses [NOM] et [VERBE] à enregistrer à l'adresse [OUTPUT],
  + 99 : fin du programme.
* **Attention**: pour cette partie, vous devrez initialiser les valeurs du couple [NOM] / [VERBE] de votre **premier**bloc :
  + [NOM] = 12,
  + [VERBE] = 2.

### Résumé de l'exercice partie 2 :

* Pour la partie 2, vous devrez chercher les valeurs du couple [NOM] / [VERBE] de votre **premier**bloc pour que le résultat de à l'adresse 0 soit 19690720.
* Le résultat de l'exercice sera le résultat du calcul suivant, avec les valeurs que vous aurez trouvées :
  + [NOM] \* 100 + [VERBE].

### Ce qui a été vu en séance :

* Prendre le temps de bien s'approprier l'énoncé/les spécifications avant de se lancer dans la conception.
* Être attentif au passage par référence/par valeur.
* Discussions sur ce qu'est un hash et les fonctions de hash.

## Session du 07/03/2023

### Résumé de l'exercice partie 1 :

* Vous devrez compter séparément le nombre de caractères qui apparaissent 2 et 3 fois dans une chaine.
* Vous devrez ensuite multiplier le nombre de caractères qui apparaissent 2 fois avec le nombre de caractères qui apparaissent 3 fois.
* Le résultat de cette multiplication est votre résultat d'exercice.

### Résumé de l'exercice partie 2 :

* Cette fois, vous devrez trouver quelle chaine dans votre input est identique à une autre, à un caractère près.
* Vous devrez renvoyer cette chaine, amputée du caractère qui différait.
* **Attention**: L'input de test de la partie 2 n'est pas le même que dans la partie 1.

### Ce qui a été vu en séance :

* Il vaut mieux éviter les négations dans les conditions losque c'est possible.
* Quand une condition doit être couplée avec une autre, il vaut mieux utiliser elseIf plutôt qu'enchainer les if. Ainsi on est certain de ne pas vérifier la seconde condition et de ne pas rentrer dans le bloc.
* Faites des fonctions pour découper votre code, c'est vital pour la compréhension.
* Le break ne casse qu'une seule boucle, pas les boucles parent. Il vaut mieux extraire la boucle la plus "haute" et en sortir avec un return.
* Les Maps ont une fonction "containsValue" pour retrouver une valeur précise.
* Il peut valoir le coup de calculer une liste en amont plutôt que de refaire un calcul plusieurs fois.

## Session du 28/02/2023

### Résumé de l'exercice partie 1 :

* Vous aurez à traiter un tableau à deux entrées contenant des nombres séparés par des tabulations.
* Pour chaque ligne, vous devrez trouver le plus petit nombre et le plus grand nombre et retourner la somme de leur écart.

### Résumé de l'exercice partie 2 :

* Dans cette partie, il faudra trouver quel nombre peut en diviser un autre et avoir un résultat entier.
* Il faudra retourner la somme des résultats de ces divisions (ligne par ligne).
* Dans chaque ligne, il n'y a qu'un nombre qui peut en diviser un autre sans laisser de reste.
* **Attention**: L'input de test de la partie 2 n'est pas le même que dans la partie 1.

### Ce qui a été vu en séance :

* Attention à vos données en input.
* Attention à ne pas faire de redondances de code :
  + Array.stream(maVariable).toList().stream() => pas terrible.
* Quand on fait une double boucle de vérification sur une liste (for i { for j {...}}) on peut faire démarrer j à i + 1 pour ne pas ré-analyser ce qui l'a déjà été.
* Dans l'utilisation d'un Stream, il peut être intéressant de s'interroger sur l'ordre des filtres. Ça peut permettre de gagner du temps d'exécution si le filtre le plus restrictif est appliqué en premier.
* Les corrections auto de IntelliJ ne renomment pas forcément bien les variables. Lors d'une correction auto, c'est bien de relire le code et de vérifier le nommage.
* La regex \\s+ est plus intéressante pour parser un fichier que \\s, dans le cas où on se trouve avec plusieurs espaces au lieu d'une tabulation.
* Différence entre FindFirst et FindAny :
  + FindFirst : Regarde le stream dans l'ordre et récupère le premier trouvé.
  + FindAny : Regarde le premier qu'il trouve, mais pas forcément le premier. Plus performant, mais non ordonné.
* Il existe plusieurs manières de récupérer le min / max d'une liste :
  + list.stream().min(),
  + list.stream().reduce(Integer::min) => pas la meilleure technique du monde...
  + Collections.min().
* Idem pour la somme d'une list :
  + list.stream().reduce(Integer::sum).orElse(0)
  + list.stream().mapToInt(Integer::intValue).sum()
* A l'utilisation d'un Set / List, il faut faire attention au choix à faire ([comment choisir ?](https://www.javaguides.net/2018/06/java-collection-framework-best-practices.html)) :
  + La liste est moins gourmande à l'enregistrement / suppression de valeurs
  + Le set est moins gourmand en accession de valeurs
    - Comme le set ne contient que des valeurs uniques, à l'utilisation d'un addAll, il est bien moins économique qu'une liste.
    - De manière générale, le set est plus gourmand que la liste. Certains cas spécifiques le rendent plus économiques.

## Session du 21/02/2023

### Résumé de l'exercice partie 1 :

* Vous devrez naviguer sur un clavier.
* Il faudra suivre quatre lignes d'instructions du type ULDR (respectivement Up, Left, Down, Right).
* En cas de dépassement du clavier, vous restez sur la touche sur laquelle vous êtes (exemple : L pour la touche 1 => touche 1).
* Chaque ligne d'instruction vous donnera, à la fin de son exécution, l'un des numéros du code final.
* La touche de départ est celle du milieu, soit la touche 5.
* Pour chaque nouvelle ligne d'instruction, il faudra repartir de la dernière touche.
* Voici la tête du clavier à manipuler :

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

### Résumé de l'exercice partie 2 :

* Même chose, mais cette fois il y a un clavier en losange.
* Voici la tête du clavier à manipuler :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | 1 |  |  |
|  | 2 | 3 | 4 |  |
| 5 | 6 | 7 | 8 | 9 |
|  | A | B | C |  |
|  |  | D |  |  |

### Ce qui a été vu en séance :

* Parfois, c'est mieux de faire simple que de faire compliqué... Il peut être plus simple de recommencer sur une autre piste que de continuer dans le mur.
* Pour simplifier le code et faire des fonctions, il existe sur la plupart des IDE une fonction d'extraction qui permet de factoriser du code.
* Extraction de méthode sur IntelliJ :
  + Clic droit > refactor > extract code
  + Ctrl + Alt + M
* En parlant fonctions, il faut faire attention à bien les nommer. La méthode de l'oracle (on commence par écrire les fonctions puis on les spécifie) peut être d'une grande aide pour faire ça.
* Eviter les variables en dur et préférer des constantes.
* Attention à la convention de nommage :
  + private static final [TYPE] XXX\_YYY
* On a toujours besoin d'un default dans un switch case : parfois on veut un comportement spécifique, souvent on a besoin d'un throw car on est en dehors de notre cas d'utilisation.
* Quand on fait une comparaison, on préfèrera utiliser la Yoda condition : on met la valeur attendue avant la valeur testée.
* Faites attention à ce que vous dit votre IDE : Il est généralement plus malin que vous, il va vous aider à améliorer votre code.