🚀 Java Hero Tour 👑

Prerequis

- Avoir déjà coder un programme dans n'importe quel language

Compétences clés

- Principes objet (constructeur, héritage, interface, classe)
- Types de bases:
  - int, float, char, double, boolean
  - String,
  - Char,
  - Booléens,
  - Entiers,
  - Flottants,
  - BigDecimal
- Structure de contrôle (boucle, condition, switch)
- Tableaux
- Listes, Sets et Maps
- Fonctions et paramètres (copie vs référence) *
- Streams
- Ligne de commande (arguments, entrée clavier) *
- Gestion des exceptions (try/catch/finally/try with resources)
- Tests unitaires avec JUnit
- Génériques (utilisation)
- Records, sealed classes
- Patrons de conception (Design pattern) : Singleton, Factory, Decorator...

Exercice 0 - Hello World !

Coder une application Hello World. Utiliser un IDE pour vous simplifier la vie.

Exercice 1 - Classe

Dans cet exercice, vous allez écrire du code pour vous aider à cuisiner de délicieuses lasagnes à partir de votre livre de cuisine préféré.

Vous avez quatre tâches, toutes liées au temps passé à cuire les lasagnes.

1. Définissez le temps de cuisson au four prévu en minutes

Définissez la méthode `expectedMinutesInOven()` qui ne prend aucun paramètre et renvoie combien de minutes les lasagnes doit être au four. Selon le livre de cuisine, le temps de cuisson prévu en minutes est de 40:

```
Lasagna lasagna = new Lasagna();
lasagna.expectedMinutesInOven();
// => 40
```

2. Calculer le temps de four restant en minutes

Définissez la méthode `remainingMinutesInOven()` qui prend les minutes réelles pendant lesquelles la lasagne a été dans le four comme paramètre et renvoie le nombre de minutes pendant lesquelles la lasagne doit encore rester dans le four, en fonction du temps de four prévu en minutes de la tâche précédente.

```
Lasagna lasagna = new Lasagna();
lasagna.remainingMinutesInOven(30);
// => 10
```

3.Calculer le temps de préparation en minutes

Définissez la méthode `preparationTimeInMinutes()` qui prend le nombre de couches que vous avez ajoutées à la lasagne comme paramètre et renvoie le nombre de minutes que vous avez passées à préparer la lasagne, en supposant que chaque couche vous prend 2 minutes à préparer.

```
Lasagna lasagna = new Lasagna();
lasagna.preparationTimeInMinutes(2);
// => 4
```

4. Calculer le temps de travail total en minutes

Définissez la méthode `totalTimeInMinutes()` qui prend deux paramètres : le premier paramètre est le nombre de couches que vous avez ajoutées à la lasagne, et le deuxième paramètre est le nombre de minutes pendant lesquelles la lasagne a été au four. La fonction doit renvoyer le nombre de minutes au total que vous avez passé sur la cuisson de la lasagne, qui est la somme du temps de préparation en minutes et du temps en minutes que la lasagne a passé dans le four en ce moment.

```
Lasagna lasagna = new Lasagna();
lasagna.totalTimeInMinutes(3,20);
// => 26
```

Exercice 2 - les Classes

Dans cet exercice, vous jouerez avec une voiture télécommandée, pour laquelle vous avez finalement économisé suffisamment d'argent pour l'acheter.

Les voitures démarrent avec des batteries pleines (100%). Chaque fois que vous utilisez la voiture à l'aide de la télécommande, celle-ci parcourt 20 mètres et décharge 1 % de la batterie.

La voiture télécommandée est dotée d'un écran LED sophistiqué qui affiche deux informations:

- La distance totale qu'elle a parcourue, affichée sous la forme : « mètres parcourus ».
- La charge restante de la batterie, affichée sous la forme : "Batterie à %".

Si la batterie est à 0%, vous ne pouvez plus utiliser la voiture et l'écran de la batterie affichera "Batterie vide".

Vous avez donc six tâches à réaliser, dont chacune fonctionnera avec des instances de voiture télécommandée.

1. Acheter une nouvelle voiture télécommandée

Implémentez la méthode (statique) `ElonsToyCar.buy()` pour renvoyer une nouvelle instance de voiture télécommandée:

```
ElonsToyCar car = ElonsToyCar.buy();
```

2. Afficher la distance parcourue

Implémentez la méthode `distanceDisplay()` de la classe `ElonsToyCar` pour renvoyer la distance telle qu'elle est affichée sur l'écran LED:

```
ElonsToyCar car = ElonsToyCar.buy();
car.distanceDisplay();
// => "0 mètres parcourus"
```

3. Afficher le pourcentage de charge restante de la batterie

Implémentez la méthode `batteryDisplay()` de la classe `ElonsToyCar` pour renvoyer le pourcentage de batterie tel qu'affiché sur l' écran LED:

```
ElonsToyCar car = ElonsToyCar.buy();
car.batteryDisplay();
// => "Batterie chargée à 100%"
```

4. Mettre à jour le nombre de mètres parcourus en roulant

Implémentez la méthode `drive()` de la classe `ElonsToyCar` qui met à jour le nombre de mètres parcourus:

```
ElonsToyCar car = ElonsToyCar.buy();
car.drive();
car.drive();
car.distanceDisplay();
// => "40 mètres parcourus"
```

5. Mettre à jour le pourcentage de batterie lors de la conduite

Mettez à jour la méthode `drive()` de la classe `ElonsToyCar` pour mettre à jour le pourcentage de charge de la batterie:

```
ElonsToyCar car = ElonsToyCar.buy();
car.drive();
car.drive();
car.batteryDisplay();
// => "Batterie chargée à 98%"
```

6. Empêcher de conduire lorsque la batterie est déchargée

Mettre à jour la méthode `drive()` de la classe `ElonsToyCar` pour ne pas augmenter la distance parcourue ni diminuer le pourcentage de batterie lorsque la batterie est déchargée (à 0%):

```
ElonsToyCar car = ElonsToyCar.buy();

// Décharger la batterie
// ...

car.distanceDisplay();
// => "2000 mètres parcourus"

car.batteryDisplay();
// => "Batterie vide"
```

Exercice 3 - Arrays et Streams

Vous êtes un ornithologue passionné qui garde une trace du nombre d'oiseaux qui ont visité votre jardin au cours des sept derniers jours.

Vous avez six tâches à réaliser, toutes traitant du nombre d'oiseaux qui sont venus dans votre jardin.

1. Vérifiez quels étaient les comptes la semaine dernière

À des fins de comparaison, vous gardez toujours une copie des décomptes de la semaine dernière à proximité, qui étaient : 0, 2, 5, 3, 7, 8 et 4. Implémentez la méthode `BirdWatcher.getLastWeek()` qui renvoie les décomptes de la semaine dernière:

```
BirdWatcher.getLastWeek();
// => [0, 2, 5, 3, 7, 8, 4]
```

2. Vérifiez combien d'oiseaux sont venus aujourd'hui

Implémentez la méthode `getToday()` dans la classe `BirdWatcher` pour renvoyer le nombre d'oiseaux qui sont venus dans votre jardin aujourd'hui. Les comptages d'oiseaux sont classés par jour, le premier élément étant le comptage du jour le plus ancien et le dernier élément étant le comptage d'aujourd'hui.

```
int[] birdsPerDay = {2,5,0,7,4,1};
BirdWatcher birdCount = new BirdWatcher(birdsPerDay);
birdCount.getToday();
// => 1
```

3. Incrémenter le décompte d'aujourd'hui

Implémentez la méthode `incrementTodaysCount()` dans la classe `BirdWatcher` pour incrémenter le décompte d'aujourd'hui:

```
int[] birdsPerDay = {2,5,0,7,4,1};
BirdWatcher birdCount = new BirdWatcher(birdsPerDay);
birdCount.incrementTodaysCount();
birdCount.getToday();
// => 2
```

4. Vérifiez s'il y a eu une journée sans visite d'oiseaux

Implémentez la méthode `hasDayWithoutBirds()` dans la classe `BirdWatcher` qui retourne `true` s'il y a eu un jour où aucun oiseau n'est venu dans le jardin sinon, elle retourne `false` :

```
int[] birdsPerDay = {2,5,0,7,4,1};
BirdWatcher birdCount = new BirdWatcher(birdsPerDay);
birdCount.hasDayWithoutBirds();
// => true
```

5. Calculer le nombre d'oiseaux visiteurs pour les n premiers jours de la semaine

Implémentez la méthode `getCountForFirstDays(n)` dans la classe `BirdWatcher` qui renvoie le nombre d'oiseaux qui ont visité votre jardin depuis le début de la semaine, mais limitez le nombre au nombre `n` de jours spécifié depuis le début de la semaine.

```
int[] birdsPerDay = {2,5,0,7,4,1};
BirdWatcher birdCount = new BirdWatcher(birdsPerDay);
birdCount.getCountForFirstDays(4);
// => 14
```

6. Calculer le nombre de jours chargés

Certaines journées sont plus chargées que d'autres. Une journée bien remplie est une journée où 5 oiseaux ou plus ont visité votre jardin. Implémentez la méthode `getBusyDays()` dans la classe `BirdWatcher` pour renvoyer le nombre de jours chargés:

```
int[] birdsPerDay = {2,5,0,7,4,1};
BirdWatcher birdCount = new BirdWatcher(birdsPerDay);
birdCount.getBusyDays();
// => 2
```

Exercice 4 - Héritage et Interface

Dans cet exercice, vous jouez à un jeu de rôle nommé "Wizards and Warriors", qui vous permet d'incarner un sorcier ou un guerrier.

Il existe différentes règles pour les guerriers et les sorciers afin de déterminer le nombre de points de dégâts qu'ils infligent.

Pour un Warrior, voici les règles :

- Infligez 6 points de dégâts si le combattant attaqué n'est pas vulnérable
- Infligez 10 points de dégâts si le combattant attaqué est vulnérable

Pour un Wizard, voici les règles :

- Infligez 12 points de dégâts si le Wizard a préparé un sort à l'avance
- Infligez 3 points de dégâts si le Wizard n'a pas préparé de sort à l'avance

En général, les combattants ne sont jamais vulnérables. Cependant, les sorciers sont vulnérables s'ils n'ont pas préparé de sort.

Vous avez six tâches qui fonctionnent avec les guerriers et les sorciers (et bonus une septième avec les voleurs).

1. Décrire un combattant

Surcharger la méthode `toString()` sur la classe `Fighter` pour renvoyer une description du combattant, formatée comme "Le combattant est un <TYPE_COMBATTANT>".

```
Fighter warrior = new Warrior();
warrior.toString();
// => "Le combattant est un Warrior"
```

2. Rendre les combattants non vulnérables par défaut

Assurez-vous que la méthode `isVulnerable()` de la classe `Fighter` renvoie toujours `false` .

```
Fighter warrior = new Warrior();
warrior.isVulnerable();
// => false
```

3. Autoriser les Sorciers à préparer un sort

Implémentez la méthode `prepareSpell()` de la classe `Wizard` pour permettre à un Wizard de préparer un sort à l'avance.

```
Wizard wizard = new Wizard();
wizard.prepareSpell();
```

4. Rendre les Sorciers vulnérables lorsqu'ils n'ont pas préparé de sort

Assurez-vous que la méthode `isVulnerable()` renvoie `true` si le Sorcier n'a pas préparé de sort ; sinon, renvoie `false` .

```
Fighter wizard = new Wizard();
wizard.isVulnerable();
// => true
```

5. Calculer les points de dégâts d'un Sorcier

Implémentez la méthode `damagePoints()` de la classe `Wizard` pour retourner les points de dégâts infligés: 12 points de dégâts lorsqu'un sort a été préparé, 3 points de dégâts sinon. Pensez à vérifier également qu'un sorcier ne puisse pas faire des dommages à un autre Sorcier.

```
Wizard wizard = new Wizard();
Warrior warrior = new Warrior();

wizard.prepareSpell();
wizard.damagePoints(warrior);
// => 12
```

6. Calculer les points de dégâts pour un Guerrier

Implémentez la méthode `damagePoints()` de la classe `Warrior` pour renvoyer les points de dégâts infligés: 10 points de dégâts lorsque la cible est vulnérable, 6 points de dégâts dans le cas contraire.

```
Warrior warrior = new Warrior();
Wizard wizard = new Wizard();

warrior.damagePoints(wizard);
// => 10
```

7. Ajouter un Voleur qui ne provoque aucun point de dégâts et peut voler

Implémentez la classe `Thief` pour qu'un voleur ne provoque aucun point de dégâts à aucun combattant. Implémentez l'interface `Theft` avec une méthode `canSteal()` qui retourne par défaut `true` (mot clé `default` dans les interfaces).

Exercice 5 - Les collections

Tu dois concevoir un petit système qui permet de :

- Ajouter des livres à la bibliothèque.
- Suivre les utilisateurs inscrits.
- Gérer les emprunts de livres en respectant certaines règles (un livre ne peut être emprunté qu'une fois à la fois).
- Afficher les informations avec tri et suppression des doublons.

Tu vas utiliser les différentes collections Java :

- List : Gérer une liste de livres.
- Set : Gérer des utilisateurs uniques.
- Map : Suivre les emprunts entre utilisateurs et livres.
- Queue : Gérer une liste d'attente si un livre est déjà emprunté.

1. Définir les classes de base

Nous avons besoin de deux classes simples : `Book` avec un titre et un auteur et `User` avec un nom.

2. Gestion de la bibliothèque

La classe `Library` utilise plusieurs types de collections pour gérer les livres, les utilisateurs et les emprunts. Dans cette classe, implémentez :

- une méthode `addBook` pour ajouter un livre à la bibliothèque. Il peut y avoir plusieurs exemplaires d'un même livre (doublon permis).
- une méthode `addUser` pour inscrire un utilisateur (un utilisateur est unique dans la bibliothèque).
- une méthode `loanBook` pour emprunter un livre. Un emprunt est une association `Map` entre un `Book` et un `User`. On gèrera une file d'attente (Queue) d'utilisateurs pour chaque livre. Pensez à initialiser cette file lors de l'ajout d'un nouveau livre. On gèrera dans cette méthode les cas d'un livre inconnu et d'un livre déjà emprunté.
- une méthode `returnBook` pour gérer le retour d'un livre emprunté. Pensez à la gestion de la file d'attente d'utilisateurs.
- une méthode `showBooks` pour lister les livres de manière unique (attention aux multiples exemplaires d'un même livre).
- une méthode `showLoans` pour lister les emprunts de la bibliothèque.

3. Tester la bibliothèque

Préparer la bibliothèque :

```
- Ajouter 2 livres dont un en double exemplaire.
- Ajouter 3 utilisateurs
```

Exemple de scénario d'emprunts :

```
// Emprunter des livres
library.loanBook(alice, book1);  // OK
library.loanBook(bob, book1);  // Ajout à la file d'attente
library.loanBook(charlie, book2);  // OK
library.loanBook(charlie, book3);  // OK

// Afficher les emprunts actuels
library.showLoans();

// Retourner un livre
library.returnBook(book1);  // Alice retourne, Bob l'emprunte

// Afficher les emprunts mis à jour
library.showLoans();
```

Exercice 6 - Les exceptions

Tu développes un gestionnaire de réservation de vol. Le système doit :

- Vérifier que le vol existe.
- Vérifier que le nombre de sièges demandés est disponible.
- Traiter les erreurs de manière appropriée en lançant et capturant des exceptions personnalisées.

1. Exceptions personnalisées

Créer deux exceptions personnalisées :

- `FlightNotFoundException` : Si le vol demandé n'existe pas.
- `InsufficientSeatsException` : Si le nombre de sièges demandés dépasse les sièges disponibles.

2. Gestion des vols

Nous allons créer une classe `FlightManager` qui stocke les vols avec le nombre de sièges disponibles (méthode `addFlight`) et gère les réservations (méthode `bookFlight`). Cette classe doit gérer les cas de vol inconnu et de sièges non disponibles.

2. Mettre tout ensemble

Les exceptions doivent être catcher dans la classe `Main`. On peut ajouter un bloc `finally` pour comprendre son comportement.

- Ajouter des vols
- Essayer de réserver un des vols précédemment ajoutés jusqu'à ne plus avoir de siège disponible (Exception `InsufficientSeatsException`)
- Essayer de réserver un vol inconnu (Exception `FlightNotFoundException`)

PS: Pensez au multi-catch !

Exercice 7 - Les tests

1. La calculatrice

Créer la classe `Calculator` qui fournira les fonctionnalités suivantes :

- Addition : `add(int a, int b)`
- Soustraction : `subtract(int a, int b)`
- Multiplication : `multiply(int a, int b)`

- Division : `divide(int a, int b)` (doit gérer la division par zéro avec une `IllegalArgumentException`)

2. Tester c'est douter !

Tester chaque opération de la classe `Calculator` , tester la division par zéro avec `assertThrows` et tester les tests paramétrés avec `@ParameterizedTest` et `@CsvSource`

3. En plus !

Vous pouvez maintenant tester également les résultats des exercices 1 (Lasagna), 3 (BirdWatcher) ou 4 (Warrior/Wizard).

Exercice 8 - Boîte à objets générique

1. Créer une classe générique Box

Créer une classe `Box` générique qui peut contenir n'importe quel type d'objet, ici un type T.

2. Créer une méthode générique qui manipule les boîtes

Dans une classe utilitaire, `BoxUtil` , créer une méthode générique qui prend une `Box` de n'importe quel type et affiche son contenu.

3. Tester la classe avec différents types d'objets

Créer plusieurs boîtes avec différents types d'objets (Integer, String, Double, etc.), et utilise la méthode générique pour afficher leur contenu.

4. Améliorations

Améliorer la classe `Box` en ajoutant les fonctionnalités suivantes :

- Créer une méthode qui compare le contenu de deux boîtes (en utilisant Comparable).
- Ajouter la capacité de stocker une collection d'objets (par exemple, List).
- Ajouter des restrictions de types sur les génériques (par exemple, T extends Number).

Exercice 9 - Design Pattern

Travailler sur les patterns de création, de structure et de comportement.

Système de commande de restauration rapide

Tu dois concevoir un système pour un restaurant rapide où un client peut passer des commandes, les cuisiniers préparent les plats, et le système de caisse gère les paiements. Le système doit utiliser plusieurs design patterns pour être modulable et extensible.

1. Utilisation du pattern Singleton pour le gestionnaire de caisse

Crée une classe `Cashier` qui suit le pattern `Singleton` . Ce gestionnaire sera responsable d'encaisser les paiements des clients via une méthode `processPayment` par exemple.

2. Utilisation du pattern Factory pour créer des commandes

Le restaurant propose plusieurs types de repas.

Utilise le pattern `Factory` pour créer les différents types de commandes avec une description et un coût pour chaque type (par exemple PIZZA, BURGER, DRINK). L' `OrderFactory` lèvera une exception pour un type de commande inconnu.

3. Utilisation du pattern Decorator pour ajouter des options à la commande

Le client peut ajouter des options à son repas (par exemple du fromage supplémentaire pour 1.5€ de plus et des frites pour 2€ en plus).

Utilise le pattern `Decorator` pour étendre les fonctionnalités des commandes et créer un `CheeseDecorator` et un `FriesDecorator`.

4. Utilisation du pattern Observer pour notifier les cuisiniers d'une nouvelle commande

Utilise le pattern `Observer` pour que chaque cuisinier (`PizzaCook`, `BurgerCook` et `DrinkCook`) soit informé lorsqu'une nouvelle commande est passée.

5. Utilisation du pattern Command pour gérer la préparation des commandes

Utilise le pattern `Command` pour encapsuler les actions de préparation des repas (classe `PrepareOrderCommand`) par les cuisiniers dans une cuisine (méthode `takeOrder` de la classe `Kitchen`).

6. Mettre tout ensemble

Crée une classe `Main` pour orchestrer tous les patterns et simuler le passage d'une commande dans le restaurant en respectant l'ordre suivant :

- Créer les commandes via l'OrderFactory (Factory : Crée des commandes de repas)
- Ajouter des options via les décorateurs (Decorator : Ajoute des options aux commandes)
- Notifier les cuisiniers (Observer : Notifie les cuisiniers quand une nouvelle commande est passée)
- Gérer les préparations dans la cuisine via le PrepareOrderCommand (Command : Encapsule les actions des cuisiniers lors de la préparation des repas)
- Payer les commandes (Singleton : Gère la caisse avec une seule instance)
- Préparer les commandes de la cuisine

Exercice 10 - Records et classes scellées

L'objectif est de pratiquer les records et les sealed classes en modélisant un système de gestion d'un zoo. Chaque animal aura des propriétés uniques, et nous utiliserons des sealed classes pour restreindre les types d'animaux possibles.

- Utiliser des records pour représenter les animaux et leurs caractéristiques immuables.
- Créer des sealed classes pour définir une hiérarchie fermée entre différents types d'animaux.

1. Créer une `sealed class` pour les animaux

Les animaux du zoo sont divisés en mammifères, oiseaux, et reptiles. Ils auront un nom et un age. Utilisez une `sealed class` pour restreindre cette hiérarchie.

2. Créer les sous classes scellées

Chaque sous-classe représente un type d'animal spécifique et utilise un record pour être immuable. Créer 3 records pour les mamifères, les oiseaux et les reptiles par exemple. Les mamifères auront une propriété supplémentaire pour savoir si ils sont en danger ou non. Les oiseaux auront une propriété supplémentaire pour connaitre leur envergure. Les reptiles auront une propriété supplémentaire pour savoir si ils sont venimeux ou non.

3. Créer une classe Zoo

La classe stockera une liste d'animaux et qui permettra d'ajouter un animal, de lister les animaux de la liste et de trouver un animal présent dans la liste par son nom.

4. Tester le tout

- Ajouter au zoo un lion, un aigle et un cobra.

- Lister les animaux présents dans le zoo.
- Rechercher l'aigle dans le zoo.

Exercice 11 - Les chiffres romains

Convertir n'importe quel nombre entier compris entre 1 et 3000 inclus en écriture romaine sachant que Les chiffres romains modernes sont écrits en exprimant chaque chiffre séparément en commençant par le chiffre le plus à gauche et en sautant tout chiffre ayant une valeur de zéro. Dans la pratique, avec l'exemple de 1990. En chiffres romains 1990 s'écrit MCMXC : 1000=M 900=CM 90=XC 2008 s'écrit MMVIII : 2000=MM 8=VIII

Exercice 12 - Compter les nombres premiers positifs inférieur à n

Bonus performance

Contrainte de temps d'exécution inférieur à 10 secondes pour n=1 000 000