TD4 : Généricité et exceptions

I. Objectifs du TD

1. Comprendre la généricité et son intérêt
2. Renforcer votre compréhension de l'héritage
3. Savoir utiliser quelques méthodes de la classe ArrayList
4. Utiliser des exceptions (bases).
5. Manipuler des codes encore plus complexes mais organisés et testés.

Dans ce dernier TD introductif à la POO, vous devez encore essayer de gagner encore en autonomie. Ce TD est potentiellement très long. Si vous n'y parvenez pas à le terminer en séance, vous devez le terminer pendant les vacances, Une solution vous sera donnée à l'issue du TD dans cet objectif. Dans les TDs suivants, tous les éléments présentés dans ces 4 premiers TDs sont considérés comme acquis. Nous allons peu à peu nous intéressez davantage à l'algorithmique maintenant que 1) nous parlons le même langage, 2) vous savez structurer vos ~~textes~~ codes.

Dans ce TD, nous allons manipuler des flottes de véhicules différemment selon qu'il s'agit de drone ou de vélos par exemple et pour finir, nous protégerons nos codes de certaines erreurs.

II. Point sur le précédent TD

Il est grandement préférable que vous utilisiez vos propres codes, quitte à les modifier ou les compléter. Néanmoins vous trouverez ICI des codes qui correspondent au TD3.

III. Pourquoi la Généricité ?

III.1 Des implémentations qui utilisent la généricité : listes

Vous travaillez pour une entreprise de transport qui a récemment investi dans une flotte de véhicules électriques. Votre tâche consiste à développer une classe ***FleetOfVehicles*** pour gérer cette flotte. La classe **FleetOfVehicles** est paramétrée pour prendre en charge différents types de véhicules électriques.

Q.1. Vous définissez la classe *FleetOfVehicules* qui hérite de la classe ArrayList dans le package *fr.epu.fleets*

```
public class FleetOfVehicles<T extends ElectricVehicle>  extends ArrayList<T>
```

Ainsi définie une *FleetOfVehicles* a toutes les propriétés d'une ArrayList, c'est-à-dire, ajout d'élément (add), retrait d'élément (remove), etc.

Q.2. Vous créez une classe de test *FleetOfVehiclesTest* et vous ajoutez le test suivant que vous devez comprendre.

a. Que peut contenir la variable `fleetOfVehicles` ?
b. Quelles méthodes de ArrayList sont utilisées ?

c. Avez-vous compris les deux tests qui utilisent *contains* ? (Distinction entre == et equals)
d. Avez-vous compris les deux tests qui utilisent *get* ?

```
@Test
void testSimpleFleetOfVehicles() {
    FleetOfVehicles<ElectricVehicle>fleetOfVehicles = new FleetOfVehicles();
    assertEquals(0, fleetOfVehicles.size());
    ElectricVehicle electricVehicle = new ElectricCar(30, "AB-123-CD");
    fleetOfVehicles.add(electricVehicle);
    assertEquals(1, fleetOfVehicles.size());
    assertTrue(fleetOfVehicles.contains(electricVehicle));
    //Without the equals method, we cannot check if the vehicle is already in the
fleet
    assertFalse(fleetOfVehicles.contains(new ElectricCar(30, "AB-123-CD")));

    fleetOfVehicles.add(new ElectricCar(40, "AB-125-XS"));
    assertEquals(2, fleetOfVehicles.size());


      ElectricVehicle electricVehicle2 = fleetOfVehicles.get(1);
      assertEquals(40,electricVehicle2.getBatteryCapacity());
      assertEquals(30, fleetOfVehicles.get(0).getBatteryCapacity());
//assertEquals("AB-125-XS", electricVehicle2.getRegistrationNumber());
}
```

Q.3. Pourquoi la dernière ligne du test précédent est-elle commentée ? Expliquez. Le fait qu'elle ne compile pas, n'est pas la réponse 😊 Pourquoi elle ne compile pas ?

III.2 Utiliser la généricité : listes de vélos

Q.4. Ajoutez le test suivant à votre *FleetOfVehiclesTest*
a. Que peut contenir la variable `fleetOfVehicles` ?

```
@Test
void testGenericOnASubclassBike(){
    FleetOfVehicles<ElectricBike>fleetOfVehicles = new FleetOfVehicles();
    assertEquals(0, fleetOfVehicles.size());
```

Q.5. Nous avons écrit le code suivant à ajouter au test, est-il valide ? Sinon quel est le problème ?

```
@Test
void testGenericOnASubclassBike(){
FleetOfVehicles<ElectricBike>fleetOfVehicles = new FleetOfVehicles();
assertEquals(0, fleetOfVehicles.size());
ElectricVehicle electricVehicle = new ElectricBike(30, new double[]{0.2, 0.5, 0.8});
fleetOfVehicles.add(electricVehicle);
```

Q.6. Nous avons écrit le code suivant à ajouter au test, est-il valide ?

```
FleetOfVehicles<ElectricBike>fleetOfVehicles = new FleetOfVehicles();
assertEquals(0, fleetOfVehicles.size());
ElectricBike electricVehicle = new ElectricBike(30, new double[]{0.2, 0.5, 0.8});
fleetOfVehicles.add(electricVehicle);
```

Q.7. Nous avons écrit le code suivant à ajouter au test, est-il valide ?

```
ElectricBike bike = fleetOfVehicles.get(0);
assertEquals(0, bike.getPedalAssistLevel());
```

III.3 Utiliser la généricité : listes de voitures électriques

Q.8. Ajoutez un nouveau test pour gérer des flottes de voitures.

```
@Test
void testGenericOnASubclassECar(){
```

A VOUS

Q.9. Le test qui a la question Q.3 doit à présent être valide dans votre nouveau test.

A cette étape du TD, vous devez avoir compris les notions de classes, sous-classes et leurs relations avec la généricité et le sous-typage.

IV. Héritage et listes

Nous allons vous donner dans cet exercice quelques bases de l'utilisation des listes en java.

IV.1 Manipulation de listes : **FleetOfVehicles**

Les différents codes demandés doivent être testés.

Q.10. Complétez la classe **FleetOfVehicles** en implémentant la méthode suivante **calculateTotalMaxRange()**.

calculateTotalMaxRange(): Calcule et renvoie l'autonomie totale maximale de la flotte sur ***leur configuration courante***. L'autonomie est la distance maximale qu'un véhicule peut parcourir avec sa charge de batterie actuelle.

Voici les codes pour ceux qui en ont besoin.

```
public double calculateTotalMaxRange() {
    double totalMaxRange = 0;
    for (T vehicle : this) {
        totalMaxRange += vehicle.calculateMaxRange();
    }
    return totalMaxRange;
}
```

a. Comprenez et expliquez ce code. Identifiez bien
   i. la définition de la variable *vehicle* de type **T** dans la boucle
   ii. l'exploitation du type T comme un *extends de ElectricVehicle* qui nous permet de faire appel à *calculateMaxRange*.
   iii. l'appel à **this** : une *FleetOfVehicles* étendant *Arraylist*, est elle-même une Liste.

Q.11. Complétez la classe **FleetOfVehicles** en implémentant la méthode suivante **calculateAverageMaxRange ()**

calculateAverageMaxRange(): Calcule et renvoie l'autonomie moyenne des véhicules de la flotte. Assurez-vous de gérer le cas où la flotte est vide.

Vous devriez cette fois-ci l'écrire seul, si besoin le voici à nouveau.

```
public double calculateAverageMaxRange() {
    if (size() == 0) {
        return 0;
    }
    double totalMaxRange = calculateTotalMaxRange();
```

```
    return totalMaxRange / size();
}
```

a. Comprenez et expliquez ce code. A nouveau l'appel à size() est un appel à this.size() qui utilise le fait qu'une *FleetOfVehicles est une liste.*

Q.12. Complétez la classe **FleetOfVehicles** en implémentant la méthode suivante **chargeFullAllVehicles ()**

chargeFullAllVehicles(): Recharge tous les véhicules de la flotte à pleine capacité.

Vous faîtes seuls : simple utilisation d'une boucle et appel à la méthode *chargeToFull*

Q.13. Complétez la classe **FleetOfVehicles** en implémentant la méthode suivante **calculateMinDistance ()**

calculateMinDistance(): Calcule et renvoie la distance la plus courte qu'un véhicule de la flotte peut parcourir à pleine charge.

Si besoin

```
public double calculateMinDistance() {
    double minDistance = Double.MAX_VALUE;
    for (T vehicle : this) {
        double maxDistance = vehicle.calculateMaxRange();
        if (maxDistance < minDistance) {
            minDistance = maxDistance;
        }
    }
    return minDistance;
}
```

Q.14. Complétez la classe **FleetOfVehicles** en implémentant la méthode suivante **calculateMaxDistance ()**

calculateMaxDistance(): Calcule et renvoie la distance la plus longue qu'un véhicule de la flotte peut parcourir à pleine charge.

Q.15. Complétez la classe **FleetOfVehicles** en implémentant la méthode suivante **chargeVehiclesBelowPercentage ()**

chargeVehiclesBelowPercentage(double percentage): Recharge les véhicules dont la batterie est chargée à moins d'un pourcentage donné (fourni en argument).

Q.16. Complétez la classe **FleetOfVehicles** en implémentant la méthode suivante **driveVehicles(double distance)**

driveVehicles(double distance): Conduit certains véhicules de la flotte sur une certaine distance et retourne le nombre total de kilomètres parcourus. Assurez-vous que la charge de la batterie des véhicules a été mise à jour après le trajet.

Q.17. Complétez la classe **FleetOfVehicles** en implémentant la méthode suivante **findVehicleWithMaxRange ()**

findVehicleWithMaxRange(): Trouve et renvoie le véhicule de la flotte ayant la plus grande autonomie.

Q.18. Testez votre nouvelle classe en ajoutant des drones, des voitures et des vélos. Assurez-vous que les méthodes calculent correctement les statistiques de la flotte et que les véhicules sont rechargés et conduits correctement.

Vous trouverez un code de test sous GitHub. Mais attention, il a été écrit à l'arrache !

V. Utilisation de la généricité et héritage par l'exemple

V.1 Une flotte de drone (Cet exercice peut être réalisé pendant les vacances.)

Votre entreprise de transport a décidé d'investir dans une flotte de drones pour effectuer des livraisons rapides.

Vous êtes chargé de développer une classe ***DroneFleet*** pour gérer cette flotte de drones. Cette classe est une sous-classe de la classe ***FleetOfVehicles*** que vous avez précédemment développée. Elle gère spécifiquement les drones.

A chaque ajout d'un drone à la flotte vous l'enregistrez dans un système de traçage. Vous devez pouvoir faire revenir les drones qui n'ont plus assez de batterie pour parcourir une distance donnée. On suppose que la distance donnée permettra à ces drones de rentrer en toute sécurité.

Q.19. Définir la classe ***DroneFleet*** qui étend votre classe ***FleetOfVehicles*** en précisant le type et ajoute, conformément à la spécification précédente, les fonctionnalités ci-après.
Cette classe n'a pas besoin de beaucoup de lignes de codes, réutilisez bien les méthodes définies dans *FleetOfVehicles*.

```
public class DroneFleet extends FleetOfVehicles<Drone>
```

- **addDrone(Drone drone)**: Ajoute un drone à la flotte de drones. Assurez-vous également d'ajouter le drone au système de suivi (**trackingSystem**) pour garder une trace de sa position.
- **getAllDronePositions()**: Utilise le système de suivi (**trackingSystem**) pour obtenir la position actuelle de tous les drones de la flotte et renvoyer une liste de positions.

- **findPositionOfDroneWithLongestDistance()**: Trouve le drone de la flotte ayant la plus grande autonomie et renvoie sa position. Si aucun drone n'est disponible, renvoyez **Optional.empty()**.
- **takeOffDronesWithRange(double range)**: Fait décoller tous les drones de la flotte qui ont une autonomie supérieure à la distance donnée en argument. Cette méthode renvoie le nombre de drones qui ont décollé.
- **returnDronesWithLowBattery(double returnDistance)**: Fait revenir les drones qui sont loin et n'ont plus assez de batterie pour parcourir la distance donnée. On suppose que la distance donnée permettra à ces drones de rentrer en toute sécurité.

Q.20. Effectuez des tests pour vérifier le bon fonctionnement de la classe *DroneFleet*. Assurez-vous que les méthodes gèrent correctement les opérations de décollage et de retour des drones, ainsi que la récupération de leur position.

Vous constatez que grâce à la généricité le type *Drone* étant connu à la compilation, c'est simple.

V.2 Une flotte de voitures électrique (facultatif)

Vous créez une flotte de voitures électriques et vous renvoyez les voitures avec le maximum d'autonomie lorsque la climatisation est activée.

Q.21. Créez cette classe et les tests associés.

VI. Interfaces et Généricité

Nous souhaitons améliorer notre système de location.

En effet, en fonction des objets à la location, nous aimerions pouvoir ajouter des méthodes telles que : rechercher des objets par d'autres propriétés telle que leur capacité maximale, par exemple par le nombre de niveau d'assistance, …

Voici notre vision : La classe *SystemRental* devient générique et nous définissons des sous classes qui implémentent cette généricité et ajoutent des comportements.

VI.1 Rendre une classe générique : RentalSystem

Q.22. Modifier la classe en ***RentalSystem<T extends RentableItem >***

a. Vous remplacez toutes les occurrences du **type** *RentableItem* par *T*

b. Vous re-testez votre nouvelle classe. Vous avez très peu de corrections à faire.

VI.2 Implémenter une classe : BikeRentalSystem

Q.23. La classe **BikeRentalSystem**

a. Est un système de réservation de vélos électriques

b. Elle permet d'obtenir la liste des vélos « louables » (au sens *findAvailableMatches*) mais qui ont, en plus, un nombre de niveaux d'assistance supérieur à un chiffre donné.

VII. Exceptions

VII.1 Levée d'exception de type Error : *IllegalArgumentException*

Q.24. Dans la classe *ElectricVehicle* modifiez votre méthode *charge* pour qu'elle lève l'exception *IllegalArgumentException* lorsque la charge passée en paramètre est négative ou nulle.

Voici les codes pour aider ceux qui ont du mal.

```
public boolean charge(double chargeAmount) {
        boolean success = false;
        if (chargeAmount <= 0 ) {
            throw new IllegalArgumentException("Charge amount must be positive");
            //return success;
        }
        if (currentCharge + chargeAmount <= batteryCapacity) {
            currentCharge += chargeAmount;
            success = true;
        } else {
            success = false;
        }
        return success;
}
```

VII.2 Tester la levée des exceptions

Q.25. Vous lancez les tests. Certains sont en erreur (sinon c'est que vous n'aviez pas suffisamment testé votre code sur ses limites !)

Voici un exemple de "test" d'exception dans un test.

```
@Test
void testChargeWith0() {
    //assertFalse((electricVehicle.charge(0)));
    assertThrows(IllegalArgumentException.class, () -> electricVehicle.charge(0));
    assertEquals(0, electricVehicle.getCurrentCharge());
}
```

Q.26. Vous modifiez vos tests, pour que tous passent.

VII.3 Levée d'exception dédiée : *NoSuchItemException*

Nous souhaitons gérer les cas où l'on cherche à accéder à des items qui n'ont pas été enregistrés dans le système de location.

Nous créons une exception dédiée.

VII.3.1 Créer une nouvelle exception : NoSuchItemException

Q.27. Créez l'exception suivante dans le package *fr.epu.rentals*

```
public class NoSuchItemException extends Exception {
    public  NoSuchItemException(RentableItem item) {
        super(item.getName() + " is not available for rental");
    }
}
```

VII.3.2 Lever une exception dédiée : *isRentable*

Q.28. Dans la classe *RentalSystem*, redéfinir la méthode *isRentable* pour qu'elle lève l'exception *NoSuchItemException* lorsque l'item passé en paramètre n'est pas connu du système de location.

```
public boolean isRentable(T item, LocalDate beginDate, LocalDate endDate)
throws NoSuchItemException {
…
    if (! (items.containsKey(item.getName()) )) {
        throw new NoSuchItemException(item);
    }
    }
```

Beaucoup de parties des codes sont touchées. Nous allons à présent « attraper » les exceptions.

VII.3.3 Laisser passer une exception : *RentItem*

Nous considérons que si l'exception est levée lors de la location d'un item alors elle doit être propagée à l'appelant pour qu'il la gère à sa convenance.

Q.29. Modifiez la méthode *rentItem* en laissant passer l'exception (même correction pour les 2 méthodes). Vous ajoutez pour cela simplement l'exception dans l'entête de la méthode.

```
public boolean rentItem(T item, LocalDate beginDate, LocalDate endDate, double cost)
throws NoSuchItemException {
```

VII.3.4 Attraper une exception dédiée et l'encapsuler dans une RuntimeException: *findAvailableMatches*

Q.30. Modifiez *findAvailableMatches*

Dans findAvailableMatches nous ne devrions pas avoir une telle erreur puisque nous travaillons sur les éléments contenus dans le système de location. Nous rattrapons donc l'erreur éventuelle pour la relancer comme une exception inattendue.

```
try {
        for (T item : matchingItems) {
            if (isRentable(item, beginDate, endDate)) {
        ….
    } catch (NoSuchItemException e) {
        throw new RuntimeException("This should not happen",e);
    }
}
```

VII.3.5 Tests et exceptions

Q.1. Corrigez les tests. Le plus souvent il s'agit de seulement modifier la signature de la méthode de test comme par exemple :

```
void testIsRentable() throws NoSuchItemException
```

Q.2. Ajoutez un test, si vous ne l'avez pas déjà, pour vérifier que l'exception est bien levée quand l'objet est inconnu.

VIII. Conclusion

À l'issue de ce TD vous maitrisez les bases de java et nous l'espérons plus largement celles de la POO.