2019

Compte rendu de TP : Rappel de notions de programmation objet

Java avancé – TP n°1

Jonathan CRÉTÉ – Filière Informatique

Exercice 1 - Location de voitures

1) Écrire une classe Car dans le package fr.umlv.rental, correspondant à un véhicule qui pourra être loué. Un vehicule est décrit par un modèle (une chaine de caractères) ainsi qu'une année de fabrication.

On écrit la classe demandée (**Voir Car.java**).

On vérifie bien que l'on ne crée pas un objet Car avec un champ model ***null*** dans le constructeur :

```
/**
 * @param model
 * @param yearBuild
 */
 public Car(String model, int year) {
      super(year);
      this.model = Objects.requireNonNull(model);
 }
```

2) Modifier la classe Car pour que le code suivant affiche le texte "ford mustang 2014"

On redéfini la méthode publique toString de *java.lang.String*

```
@Override
public String toString() {
      return model + " " + this.getYear();
}
```

3) Créer une classe CarRental (toujours dans le package fr.umlv.rental) qui stocke l'ensemble des véhicules qui peuvent être loués dans une liste. La classe CarRental doit posséder une méthode add qui permet d'ajouter un véhicule dans la liste. Faire en sorte que la liste ne puisse pas contenir null en empêchant d'ajouter des voitures null.

On créer la classe demandée dans le bon package. On utilise un objet ArrayList<> nommé *rental*, que l'on initialise dans notre constructeur. (**Voir CarRental.java**).

On développe la méthode publique add(..) en vérifiant à l'appel de la méthode add() de *java.util.ArrayList* que l'on n'ajoute pas un objet **null**.

```
/**
 * add a vehicle to rental arrayList
 * @param vh
 */
 public void add(Vehicle vh) {
      rental.add(Objects.requireNonNull(vh));
 }
```

4) Pour visualiser une instance de la classe CarRental, on devra afficher l'ensemble des véhicules de la liste, séparés par des retours à la ligne (mais sans retour à la ligne final !). Écrire le code correspondant en utilisant la classe StringBuilder.

Pour développer la méthode en utilisant un objet StringBuilder et sans retour à la ligne final, on utilise la méthode substring :

```
public String toString() {
	StringBuilder st  = new StringBuilder();

	for(Car car: this.cars) {
			st.append(car);
			st.append("\n");
	}
	if(!cars.isEmpty()) {

		st.substring(0, st.length()-1);
		return st.toString();
	}
	return "";
}
```

5) Écrire une méthode remove qui permet de retirer un véhicule de la liste. Que faire si le véhicule n'a pas été préalablement ajouté ? Vérifier que le test shouldRemoveCarOfRental est valide. Sinon, expliquez quel est le problème et corrigez-le.

On développe la méthode *remove(...)* demandée (**Voir CarRental.java**)

Dans le cas où un véhicule n'a pas été préalablement ajouté, on lève une IllegalStateException(..). On vérifie également que l'on ne supprime pas un objet avec comme valeur **null.**

6) Rappeler à quoi sert l'interface Stream en Java, comment obtenir un stream à partir d'une liste, comment marchent les méthodes filter, map et collect et enfin comment peut-on utiliser le collecteur Collectors.joining() pour simplifier l'implantation de la méthode d'affichage que vous venez d'écrire.

L'interface Stream est une séquence d'éléments qui nous permet d'effectuer un groupe d'opérations de manière séquentielle ou parallèle.

Pour obtenir un stream à partir d'une liste, on appelle la méthode *stream()* sur l'objet de type List.

- ***filter(Predicate<? super String> predicate)*** --> Cette méthode nous permet de filtrer un stream, en retournant un stream composé des éléments de ce stream qui correspondent au prédicat donné.
- ***map(Function<? super T,? extends R> mapper)*** --> map(..) nous permet de modifier directement ce que nous venons de récupérer. Ici, la méthode map(...) va prendre en argument un lambda appelant sur un x la méthode donnée.
- ***collect()*** --> Cette méthode effectue une opération de réduction mutable sur des éléments de ce stream en utilisant un objet de type Collector. Un Collector permet de

réaliser une opération de réduction qui accumule les éléments d'un Stream dans un conteneur mutable.

On simplifie la méthode *toString(..)* avec les explications données plus haut.

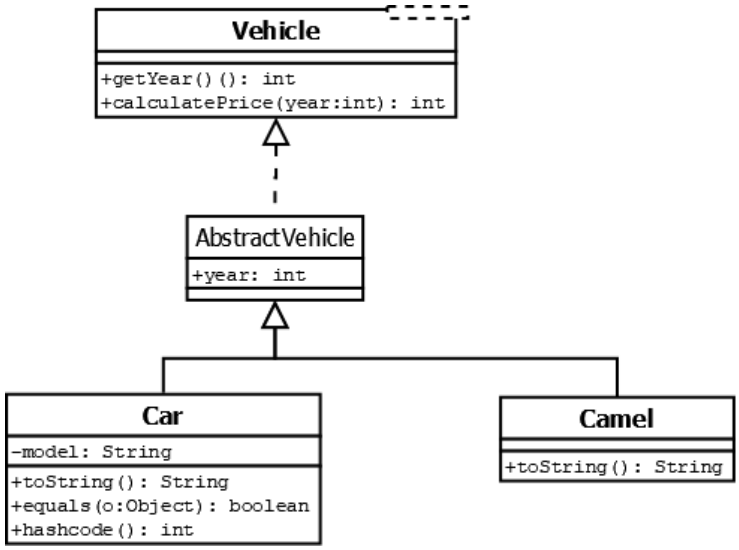
7) On cherche à connaitre toutes les voitures enregistrées dans le CarRental ayant la même année de fabrication. Écrire une méthode findAllByYear(int year) qui prend en paramètre une année et renvoie une liste des voitures ayant l'année de fabrication demandée. Que doit-on faire s'il n'y a pas de voiture correspondant à l'année demandée.

On développe la méthode *findAllByYear(...)* en utilisant l'API stream : la méthode filter(...) nous permet d'utiliser un lambda de manière à filtrer si l'année passée en paramètre correspond à une année de fabrication d'une voiture (Car) dans l'ArrayList.

Dans le cas ou il n'y a pas de voiture correspondant à l'année demandée, on retourne une ArrayList vide.

8) L'application que vous développez doit aussi être vendue en Egypte où malheureusement, il n'est pas rare de manquer d'essence. Pour éviter de mettre la clé sous la porte, les loueurs de voitures ont trouvé une solution de secours en louant aussi des chameaux. Modifier le code de votre application pour permettre de louer non plus uniquement des véhicules mais aussi des chameaux, sachant qu'un chameau possède juste une date de naissance et que son affichage est "camel" suivi d'un espace et de sa date de naissance.

On développe la classe Camel (**Voir Camel.java**)



On Développe une interface publique *Vehicle*. Aussi, on écrit une classe abstraite *AbstractVehicle* de manière à factoriser le code commun. Le champ yeat est « partagé » entre les classes Car et Camel.

On rédéfini la méthode toString pour avoir l'affichage demandé :

```
@Override
public String toString() {
      return "camel " + getYear() ;
}
```

9) Comment faire pour que la date de fabrication d'un véhicule et de naissance d'un chameau correspondent à un seul et même champ partagé par les classes Car et Camel?

Pour que la date de fabrication d'un véhicule et de naissance d'un chameau correspondent à un seul et même champ il suffit de déclarer le champ *year* dans une classe abstraite nommée *AbstractVechicle* dont les classes Car et Camel vont hériter.

10) Finalement, est-il vraiment nécessaire d'utiliser une interface?

Oui il est vraiment nécessaire d'utiliser une interface de manière à donner le type commun *Vehicle* à nos objets, qui sont utilisés dans nos méthodes de classes.

11) Écrire dans la classe CarRental, une méthode insuranceCostAt qui permet de calculer le coût total pour assurer tous les véhicules pour une année donnée (passée en paramètre). Note: pensez à gérer le cas où la date est plus ancienne l'année ce création du véhicule ou de naissance des chameaux.

Afin de calculer le prix de l'assurance pour une voiture ou un chameau on définit la signature de notre méthode *calculatePrice(...)* dans l'interface publique Vehicle. Ensuite on redefinie cette méthode dans les classes Car et Camel et on effectue le calcul, en fonction des critères demandés (200 euros l'assurance si moins la voiture à moins de 10 ans...)

Voir **Car.java/Camel.java/CarRental.java**

```
/**
 * return price of insurance
 * @param year
 * @return
 */
public int insuranceCostAt(int year) {
      return rental.stream()
      .mapToInt(x -> x.calculatePrice(year))
      .sum();
}
```

Puis on développe notre méthode *insuranceCostAt(...),* en utilisant la méthode stream sur notre ArrayList rental.On appelle la méthode mapToInt(...) qui nous permet de modifier directement ce que nous venons de récupérer, pour chaque élément en retournant un objet de type IntStream. La méthode mapToInt(...) va prendre en argument une lambda appelant sur un x la méthode *calculatePrice(...).*

Enfin la méthode sum(...) va effectuer la somme totale correspondant au coût total pour assurer tous les véhicules pour une année donnée.

12) Enfin, écrire dans la classe CarRental, une méthode findACarByModel qui permet de trouver une voiture à partir de son modèle passé en paramètre. Expliquer de plus pourquoi cette méthode doit retourner un objet de type Optional.

```
/**
 * return a car with his model
 * @param model
 * @return
 */
public Optional<Car> findACarByModel(String model) {
      Objects.requireNonNull(model);
      List<Vehicle> car = rental.stream()
      .filter(x -> x instanceof Car && ((Car)x)
            .getModel().equals(model))
      .collect(Collectors.toList());
      if(!car.isEmpty()) {
            return Optional.of((Car)car.get(0));
      }
      else {
            return Optional.empty();
      }

}
```

On développe la méthode demandée (voir ci-dessus) qui doit retourner un Optional<Car>. On commence par vérifier que l'objet passé en paramètre n'est pas **null**. Ensuite on déclare et initialise une *List<Vehicle>* une Stream. Avec la méthode *stream()*, on appelle la méthode *filter()* pour appliquer le filtre suivant :

- Avec un lambda, on s'assure que l'objet cherché est du type Car avec la méthode *instanceof*
- On concatène le prédicat et on teste si le modèle de la voiture passé en paramètre est égal à l'objet Car en cours (dans la List).

Ensuite on teste si la List est vide ou non : Si ce n'est pas le cas, on retourne un Optional de Car à l'aide d'un cast. Sinon, on retourne un objet de type Optional vide.

Cette méthode doit retourner un Optional de manière à éviter de retourner null et lever une NullPointerException.

Conclusion

Durant ce premier TP de JAVA avancé, j'ai pu revoir des notions importantes de la programmation orienté objet : encapsulation, redéfinition et appel de méthode, exceptions, implémentation d'une interface...

J'ai surtout pu remettre en place les bonnes pratiques (redéfinition *toString, equals & hashcode,* utilisation d'une classe abstraite pour factoriser le code...). Les tests unitaires m'ont permis de me rendre compte si ce que j'avais codé était cohérent avec le résultat attendu.

J'ai pu comprendre l'intérêt de l'API Stream avec des lambda, qui nous demander de penser de façon fonctionnelle et non impérative. En effet, une lambda nous fournit une implémentation pour une interface fonctionnel donnée. Toutefois seul souci, j'ai remarqué qu'il est impossible de lever une exception dans une lambda (à moins de la déclarer dans l'interface fonctionnelle) ce qui m'a un peu embêté au début du TP. J'ai rencontré une difficulté à comprendre comment fonctionne la méthode *map(...)* de l'API stream.

Je dois bien revoir les Exceptions et j'aimerai travailler sur l'implémentation de lambda à l'aide de tests unitaires (avec JUnit 5).