Aix*Marseille université
Initiative d'excellence
Année 2018-19

Domaine Sciences et Technologies
Licence informatique

Programmation 2 : TP 8 (informatique)
Code UE : SIN3U02

TP noté : agence de location

1 Consignes à suivre

1.1 TP noté

Ce TP est à faire en 8h de séances de TP comprise entre le 18 novembre et le premier décembre. Vous avez le droit de faire le projet seul ou en binôme. Vous avez aussi le droit de travailler en dehors des heures de TP. Il faudra que le dépôt *git* soit à jour pour le premier décembre à 23h. Toute mis à jour après cette date sera impossible. Le TP sera évalué et comptera pour 20% de la note finale.

1.2 Consignes pour démarrer le TP

Comme pour les TP précédents, on va utiliser git pour la gestion de versions. Il vous faut donc vous reporter aux consignes du premier TP.

Lien vers la classroom du TP : lien

1.2.1 Respect de la propriété intellectuelle

Comme pour tout TP noté, nous vous demandons de ne pas partager votre programme, complet ou partiel, avec des membres d'autres équipes que la votre. Le non-respect de cette consigne vous expose à recevoir une note nulle. Tout emprunt que vous effectuez doit être proprement documenté en indiquant quelle partie de votre programme est concerné et de quelle source elle provient (nom d'un autre étudiant, site internet, *etc.*).

1.2.2 Critères d'évaluation

Ce TP sera évalué et comptera pour 20% de la note finale de l'unité d'enseignement de programmation 2. Vous serez évalué sur :

- **La propreté du code :** comme indiqué dans le cours, il est important de programmer proprement. Des répétitions de code trop visibles, des noms mal choisis ou des fonctions ayant beaucoup de lignes de code (plus de dix) vous pénaliseront. Le sujet vous donne les méthodes que vous devez absolument écrire mais il est tout à fait autoriser d'écrire des méthodes supplémentaires, de créer des constantes, ... pour augmenter la lisibilité du code. On rappelle que vous devez écrire le code en anglais.
- **La correction du code :** on s'attend à ce que votre code soit correct, c'est-à-dire respecte les spécifications dans le sujet. Vous avez tout intérêt à tester votre code pour vérifier son comportement.
- **Modificateurs d'accès et attributs `final` :** dans ce sujet, on ne vous donnera pas les modificateurs d'accès pour les différents éléments du code. Ce sera donc à vous de choisir l'accessibilité de tous ces éléments entre : `private`, `public`, `protected` et *default* (pas de mot-clé). Vous aurez aussi à choisir si vous souhaitez utiliser le mot-clé `final` sur les attributs des classes. Vous serez évalué sur ces choix.
- **Les commit/push effectués :** il vous faudra travailler en continu avec `git` et faire des *push/commit* le plus régulièrement possible. Un projet ayant très peu de *push/commit* effectués juste avant la date limite sera considéré comme suspicieux et noté en conséquence. Vous devez faire un commit par méthode que vous codez et un push après chaque tâche. Si vous êtes en binôme, il faudra qu'il y ait des *push/commit* avec les comptes *github* des deux membres.

1.3 Première modification de votre dépôt

Ajouter un fichier `README.md` à la racine du projet. Ce fichier devra contenir votre nom ainsi que le nom de votre éventuel co-équipier. Ce fichier aura le format suivant :

```
# Agence de voyage

## Description du projet

Ce projet consistera à coder des classes permettant à une agence de location (*rental agency* en ang

## Membres du projet

- NOM prénom du premier participant
- NOM prénom du deuxième participant
```

2 L'agence de location

2.1 Introduction

Une agence de location (*rental agency* en anglais) de vehicle (*vehicles*) offre à ses clients la possibilité de choisir la voiture qui souhaite louer en fonction de différents critères. L'agence propose la location de deux types de vehicles : des motos (*motorbikes*) et des voitures (*cars*). Vous allez donc créer deux classes `Car` et `Motorbike` qui implémenteront une interface `Vehicle`.

2.2 Tâche 1 : interface `Vehicle`

Vous avez deux choses à faire pour cette tâche :

— *Créer un package* ***`agency`*** *dans* ***`src`*** *->* ***`main`*** *->* ***`java`***.

— *Créer une interface* ***`Vehicle`*** *dans le package* ***`agency`*** *qui contient les méthodes suivantes* :

- `String getBrand()` : renvoie la marque du véhicule sous la forme d'une chaîne de caractères.
- `String getModel()` : renvoie le modèle du véhicule sous la forme d'une chaîne de caractères.
- `int getProductionYear()` : renvoie l'année de fabrication du véhicule.
- `double dailyRentalPrice()` : renvoie le prix de la location pour un jour du véhicule.
- `boolean equals(Object o)` : teste l'égalité entre le véhicule et l'objet `o`. Cette méthode devra renvoyer vrai si `o` correspond au même véhicule que `this`. On considérera que deux véhicules sont égaux s'ils sont des instances de la même classe et qu'ils ont la même marque, le même modèle et la même année de production. Pour tester si deux objets ont la même classe ont pourra récupérer leur classe avec la méthode `getClass()` d'`Object` et l'opérateur `==` pour tester si deux classes sont égales.
- `String toString()` : renvoie une chaîne de caractères qui comprend dans l'ordre et séparé par des espaces :
  - le type de véhicule : *Motorbike* ou *Car*
  - la marque (*brand*) du véhicule,
  - le modèle (*model*) du véhicule,
  - l'année de fabrication (*production year*) du véhicule,
  - des détails entre parenthèses sur le véhicule (spécifique à chaque type de véhicule),
  - deux point :
  - le prix par jour du véhicule en euros suivi du caractère €.

2.3 Tâche 2 : classe `Car`

Pour cette tâche vous aurez besoin de déterminer l'année courante. Pour récupérer la valeur de l'année en cours vous pouvez utiliser la méthode `public static int currentYearValue()` de la classe `TimeProvider` du package `util` qui vous est fourni dans le projet.

Pour cette tâche, vous devez créer une classe ***`Car`*** *à l'intérieur du package* ***`agency`*** *qui implémente l'interface* ***`Vehicle`***. *Cette classe possédera les éléments suivants* :

— un attribut `brand` de type `String`,
— un attribut `model` de type `String`,
— un attribut `productionYear` de type `int`,
— un attribut `numberOfSeats` de type `int` et qui correspond au nombre de sièges de la voiture,
— un constructeur `Car(String brand, String model, int productionYear, int numberOfSeats)` qui devra lever une exception `IllegalArgumentException` (exception qui existe déjà en Java et donc pas besoin de la définir) dans les deux cas suivants :
  — si l'année donnée en argument est inférieure strictement à `1900` ou supérieur strictement à l'année en cours. Dans ce cas le message de l'exception devra contenir l'année de production donnée en argument du constructeur.
  — si le nombre de sièges est inférieur strictement à 1. Dans ce cas le message de l'exception devra contenir le nombre de siège donné en argument du constructeur.

  Dans les deux cas, le message de type `String` donné au constructeur de l'exception devra indiquer pourquoi l'exception s'est produite.
— une méthode `String toString()` qui devra respecter le contrat défini dans l'interface `Vehicle`. Les détails d'une voiture à afficher entre parenthèse correspondront à son nombre de sièges. Par exemple, pour une voiture ayant 3 sièges il faudra que la chaîne contiennent `(3 seats)` alors qu'elle devra contenir `(1 seat)` si la voiture n'a qu'un siège.
— une méthode `boolean isNew()` qui devra retourner `true` si la voiture a 5 ans ou moins (modèle récent) et faux sinon. Attention votre code devra fonctionner en utilisant l'année en cours : en 2019 les vieux modèles seront ceux datant d'au moins 2013 alors qu'en 2020, les voitures de 2014 seront considérées comme des vieux modèles.
— une méthode `double dailyRentalPrice()` qui devra retourner le prix suivant la formule suivante : une voiture (vieux modèle) ayant plus de 5 ans coûtera 20 euros par siège alors qu'une voiture ayant 5 ans ou moins (modèle récent) coûtera 40 euros par siège.

Pour tester votre classe `Car`, vous pouvez décommenter le code contenu dans la classe de test nommée `TestCar` qui se trouve dans `src` -> `test` -> `java` puis lancer les tests en cliquant deux fois sur `agency-***` -> `Tasks` -> `verification` -> `test`.

2.4 Tâche 3 : classes `Motorbike` et `AbstractVehicle`

Pour cette tâche, vous devez créer une classe ***`Motorbike`*** *à l'intérieur du package* ***`agency`*** *qui implémente l'interface* ***`Vehicle`****. Cette classe possédera les éléments suivants* :

— un attribut `brand` de type `String`,
— un attribut `model` de type `String`,
— un attribut `productionYear` de type `int`,
— un attribut `cylinderCapacity` de type `int` et qui correspond à la cylindrée en centimètre cube de la moto,
— un constructeur `Motorbike(String brand, String model, int productionYear, int cylinderCapacity)` qui devra lever une exception `IllegalArgumentException` (exception qui existe déjà en Java et donc pas besoin de la définir) dans les deux cas suivants :
  — si l'année donnée en argument est inférieure strictement à `1900` ou supérieur strictement à l'année en cours.
  — si la cylindrée est inférieure à 50.

  Dans les deux cas, le message de type `String` donné au constructeur de l'exception devra indiquer pourquoi l'exception s'est produite.
— la méthode `String toString()` qui devra respecter le contrat défini dans l'interface `Vehicle`. Les détails d'une moto à afficher entre parenthèse avant le prix correspondront à sa cylindrée. Par exemple, pour une moto ayant un moteur de 500cm^3 il faudra que la chaîne de caractères contienne `(500cm³)`.
— la méthode `double dailyRentalPrice()` devra retourner le prix suivant la formule suivante : une moto

coûtera 0,25€ par centimètre cube de cylindrée.

Vous pouvez remarquer que la classe `Motorbike` a beaucoup de points communs avec la classe `Car`. Afin d'éviter la duplication de code, *nous vous conseillons l'une de faire l'une des deux choses suivantes (et seulement une, pas les deux) :*

- créer une classe abstraite `AbstractVehicle` qui contiendra le code en commun des deux classes et qui sera étendue par les deux classes `Car` et `Motorbike`.
- créer une interface `VehiclesSpecifics` implémentée par deux classes : `CarSpecifics` et `MotorbikeSpecifics` et utiliser la composition avec les classes `Motorbike` et `Car`.

2.5 Tâche 4 : `RentalAgency` et `UnknownVehicleException`

Pour cette tâche, vous devez créer une classe ***`RentalAgency`*** *à l'intérieur du package* ***`agency`*** *qui contiendra les éléments suivants (pour le moment)* :

- un attribut `List<Vehicle> vehicles` qui est la liste des véhicules de l'agence,
- un constructeur `RentalAgency()` qui construit une agence sans véhicules,
- un constructeur `RentalAgency(List<Vehicle> vehicles)` qui construit une agence avec les véhicules contenus dans la liste spécifiée en argument,
- une méthode `boolean add(Vehicle vehicle)` qui ajoute un véhicule à l'agence si celui-ci n'est pas déjà un véhicule de l'agence et qui ne fait rien sinon, cette méthode renvoie `true` si elle réussi à ajouter un véhicule et `false` sinon,
- une méthode `void remove(Vehicle vehicle)` qui enlève un véhicule à l'agence. Si le véhicule n'est pas un véhicule disponible à l'agence, une exception de type `UnknownVehicleException` devra être levée (exception définie ci-dessous),
- une méthode `boolean contains(Vehicle vehicle)` qui teste si un véhicule est dans l'agence ou pas, c'est-à-dire qui renvoie `true` si le véhicule est dans l'agence et `false` sinon.
- une méthode `List<Vehicle> getVehicles()` qui renvoie la liste des véhicules de l'agence.

Vous devez aussi créer une classe ***`UnknownVehicleException`*** *qui étend* ***`RuntimeException`*** *et qui correspond à l'exception levée lorsqu'on essaie de supprimer un véhicule n'étant pas dans l'agence.* Cette classe contiendra :

- un attribut `Vehicle vehicle` correspondant au véhicule qu'on a essayé d'enlever.
- un constructeur `UnknownVehicleException(Vehicle vehicle)` évident,
- la redéfinition de la méthode `String getMessage()` : cette méthode qui renverra une chaîne de caractère indiquant que le véhicule n'existe pas dans l'agence. Cette chaîne de caractères devra contenir la représentation sous forme de chaîne de caractères du véhicle (obtenu via un appel à `toString()`).

Pour tester votre classe `RentalAgency`, vous pouvez décommenter le code correspondant dans la classe de test nommée `TestRentalAgency`.

2.6 Tâche 5 : critères et filtres

On souhaite pouvoir sélectionner parmi les véhicules à louer toutes les véhicules satisfaisant un critère (`criterion`) donné. Un critère peut être satisfait ou pas par un véhicule, cela signifie que le critère permet de filtrer (ou sélectionner) des véhicules ayant certaines propriétés. Ce concept existe déjà en java (depuis Java 1.8) sous la forme d'une interface générique `Predicate<T>` qui correspond à l'interface suivante :

```
/**
 * Represents a predicate (boolean-valued function) of one argument.
 *
 * @param <T> the type of the input to the predicate
 *
 */
public interface Predicate<T> {
```

```
    /**
     * Evaluates this predicate on the given argument.
     *
     * @param t the input argument
     * @return {@code true} if the input argument matches the predicate,
     * otherwise {@code false}
     */
    boolean test(T t);
}
```

Cette interface permet donc de définir un critère (on dit aussi un prédicat) sur des objets d'un type `T` quelconque. On peut tester le critère avec l'unique méthode abstraite `test` de l'interface.

Le premier critère qu'on souhaite définir va nous permettre de filtrer les véhicules ayant une marque spécifique.

Écrire une classe ***`BrandCriterion`*** *implémentant l'interface* ***`Predicate<Vehicle>`***.

Cette classe contiendra :

— un attribut `String brand` correspondant à la marque des véhicules qu'on souhaite sélectionner,

— un constructeur `BrandCriterion(String brand)` évident,

— une méthode `boolean test(Vehicle vehicle)` qui renvoie `true` si le véhicule a une marque égale à l'attribut `brand` et `false` sinon.

Écrivez une classe ***`MaxPriceCriterion`*** *implémentant l'interface* ***`Predicate<Vehicle>`*** *qui permet de sélectionner les véhicules ayant un prix de location inférieur ou égal à un montant que l'on passera en paramètre du constructeur de la classe.*

Ajoutez deux méthodes ***`select`*** *et* ***`printSelectedVehicles`*** *dans la classe* ***`RentalAgency`*** *qui remplissent les contrats suivants :*

```
/**
 * Returns the list of vehicles of this agency that satisfy the specified criterion
 * The returned vehicles are then << filtered >> by the criterion.
 *
 * @param criterion the criterion that the selected cars must satisfy
 * @return the list of cars of this agency that satisfy the given criterion
 */
public List<Vehicle> select(Predicate<Vehicle> criterion)

/**
 * Prints the vehicles (one by line) of this agency that satisfy the specified criterion
 *
 * @param criterion the criterion that the selected cars must satisfy
 */
public void printSelectedVehicles(Predicate<Vehicle> criterion)
```

On peut naturellement souhaiter faire des intersections de critères, ce qui revient à appliquer le *et* logique entre les critères. On obtient alors un nouveau critère qui est satisfait si et seulement si tous les critères qui le composent sont satisfaits.

Écrivez une classe ***`IntersectionCriterion`*** *qui permet de définir des critères par intersection de plusieurs critères.*

Cette classe devra être générique avec un paramètre de type `T` et devra implémenter l'interface `Predicate<T>`. Elle aura en constructeur prenant en paramètre une liste de critères. Sa méthode `test` retournera vrai si et seulement tous les critères passe le test.

Pour tester la méthode `select`, vous pouvez décommenter le code correspondant dans la classe de test nommée `TestRentalAgency`.

2.7 Tâche 6 : gestion des locations

On souhaite ajouter à la classe `RentalAgency` la gestion de locations de véhicules par des clients. Un client ne peut louer qu'un véhicule à la fois.

2.7.1 Classe `Client`

Écrivez une classe `Client` qui permet de définir un client.

Un client aura une année de naissance, un nom de famille et un prénom. Il vous faudra générer les *getters* de cette classe ainsi que les méthodes `equals` et `hashCode` avec le menu *generate* d'IntelliJ.

2.7.2 Gestion de location dans la classe `RentalAgency`

Pour gérer des locations, vous allez utiliser une table (interface `Map<K,V>` du package `java.util`). Cette interface, qui est implémentée (en autre) par la classe `HashMap<K,V>`, permet d'associer des clés (de type `K`) à des valeurs (de type `V`). Elle contient (entre autre) les méthodes suivantes :

- `boolean containsKey(Object key)` : Returns `true` if this map contains a mapping for the specified `key`.
- `V get(Object key)` : Returns the value to which the specified `key` is mapped, or `null` if this map contains no mapping for the `key`.
- `V put(K key, V value)` : Associates (maps) the specified `value` with the specified `key` in this map.
- `Collection<V> values()` : Returns a Collection view of the values contained in this map.

Dans notre cas, les clients seront les clés et les véhicules qu'ils ont loué seront les valeurs. Une association clé/valeur correspondra donc à un client ayant loué un véhicule. Un client n'est donc présent dans cette table que si il est en train de louer un véhicule. Il en donc supprimé dès qu'il rend un véhicule.

Complétez la classe RentalAgency avec les éléments suivants :

- un attribut `Map<Client, Vehicle> rentedVehicles` qui contiendra les associations entre clients et véhicules loués.
- une méthode `double rentVehicle(Client client, Vehicle vehicle) throws UnknownVehicleException, IllegalStateException` : permet au client `client` de louer le véhicule `vehicle`. Le résultat est le prix de location. L'exception `UnknownVehicleException` est levée si le véhicule n'existe pas dans l'agence et `IllegalStateException` est levée s'il est déjà loué ou que le client loue déjà un autre véhicule.
- une méthode `boolean aVehicleIsRentedBy(Client client)` : renvoie `true` si et seulement si `client` est un client qui loue actuellement un véhicule et donc `false` sinon.
- une méthode `boolean vehicleIsRented(Vehicle v)` : renvoie `true` si et seulement si le véhicule est actuellement loué, `false` sinon.
- une méthode `void returnVehicle(Client client)` : le client `client` rend le véhicule qu'il a loué. Il ne se passe rien s'il n'avait pas loué de véhicule.
- une méthode `Collection<Vehicle> allRentedVehicles()` : renvoie la collection des véhicules de l'agence qui sont actuellement loués.

Pour tester la classe `Client`, vous pouvez décommenter le code correspondant dans la classe de test nommée `TestClient`.

Pour tester les nouvelles méthodes de `RentalAgency`, vous pouvez décommenter le code correspondant dans la classe de test nommée `TestRentalAgency`.

2.8 Tâches optionnelles

- **Ajout de tests :** Ajouter les tests manquants comme par exemple des tests pour la classe `Motorbike`.
- **Ajout de la javadoc :** Ajouter une documentation java (de préférence en anglais) aux élément du codes pour lesquels cela vous semble utile.

— **Nouveaux critères :** rajoutez des classes supplémentaires pour les critères de sélection comme :
  — `MinPriceCriterion` qui permet de sélectionner des véhicules coûtant au moins un prix.
  — `ModelCriterion` qui permet de sélectionner des véhicules ayant correspondant à un modèle.
  — ...
— **Sauvegarde/chargement depuis des fichiers :** Modifiez votre code pour permettre la sauvegarde sous forme de fichier de la liste des véhicules d'une agence ainsi que le chargement depuis un fichier. L'idée serait de stocker les véhicules sous la forme de chaîne caractères avec les représentations des attributs et des types de véhicule séparés par un symbole spécial (par exemple `;`) et de mettre un véhicule par ligne du fichier. Vous pourrez utiliser la classe `Scanner` pour cela.
— **Prise en charge de rabais :** on pourrait imaginer des agences qui gèrent des rabais. Pour cela, vous pouvez faire une classe `Discount` qui contiendrait par exemple des conditions sur le client et le véhicule, et le taux de rabais en pourcentage que l'agence appliquerait lors d'une location.