TD2 : Héritage en action

I. Objectifs du TD

1. Bases de l'héritage
2. Manipulation de tableaux

Les codes développés seront réutilisés au TD suivant.

Vous devez absolument utiliser le temps du TD pour apprendre et effectuer les exercices.
L'objectif n'est surement pas que vous les fassiez en dehors des TDs.
Vous devez toujours terminer les TDs en vous assurant que :

- Vous avez répondu à toutes les questions ;
- Votre code est vraiment bien testé, n'hésitez pas à partager vos tests pour aider vos camarades ;
- La qualité de vos codes est bonne :
  i. utilisez sonarLint et profitez-en pour apprendre de nouvelles choses,
  ii. demandez à votre professeur si vous avez un doute.

Si vous êtes des étudiants avancés et que vous avez bien rempli toutes les tâches précédentes :

- Vous introduisez de nouvelles fonctionnalités et de nouveaux tests que vous partagez avec votre professeur puis, éventuellement, vos camarades ;
- Vous aidez les camarades en difficulté ; c'est étonnant tout ce que l'on peut apprendre en expliquant aux autres 😉.

Dans ce TD, nous allons définir différents types de véhicules électriques, des vélos, des voitures et des drones. Les stations de charge auront la capacité de charger tous ces véhicules sans que vous ayez à écrire beaucoup de codes. Enfin vous gérerez une flotte de véhicules électriques dont certains seront disponibles et d'autres en réparation.

Avertissement :
Sur ces premiers exercices assez simples, il est possible que les outils d'IA génératives vous donnent de bonnes solutions. Par la suite, les solutions données ne seront peut-être pas aussi adéquates et encore plus tard, il ne pourra plus vous répondre, en tous cas pour l'instant [Article].
Vous devez absolument être capable de produire seuls les codes pour apprendre en vous trompant notamment. Ensuite, vous aurez la capacité d'évaluer les solutions générées et même de diriger l'outil. Mais si vous ne pouvez pas évaluer ses réponses, vous n'en serez jamais le pilote.

II. Point sur le précédent TD

Nous allons repartir des résultats obtenus la semaine dernière.
Sous https://github.com/MireilleBF/StudentCodeBase/tree/main/POOToASD/TD1/CODES, vous trouvez des solutions aux exercices de la semaine dernière.

Attention la maitrise de l'environnement doit, au moins dans les tâches essentielles, être acquise.

III. Héritage : partage de codes

III.1 La classe Drone : une simple extension d'un véhicule électrique

Un drone est un véhicule électrique.
Dans ce premier exercice nous vous aidons pas à pas.

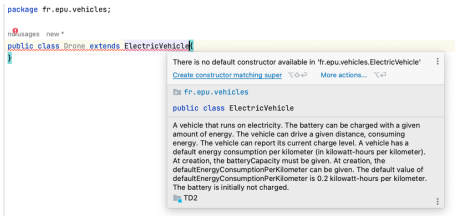
III.1.1 Déclarer une classe comme une extension d'une autre classe

Q.1. Créez la classe `Drone` comme une extension de *`ElectricVehicle`*

```
public class Drone extends ElectricVehicle{
```

III.1.2 Héritage et Constructeurs

*Dans le constructeur d'une sous-classe, le premier appel doit être soit à un constructeur explicite de la superclasse (**super(...)**), soit à un autre constructeur de la même classe (**this(...)**). Cet appel doit être la première instruction du constructeur.*



Q.2. Vous définissez donc un simple constructeur qui fait appel au constructeur défini dans la classe *ElectricVehicle*. Par exemple,

```
    public Drone(double batteryCapacity, double energyConsumptionPerKilometer) {
        super(batteryCapacity, energyConsumptionPerKilometer);
    }
}
```

III.1.3 Utiliser les méthodes définies par la super-classe

Q.3. Vous testez votre nouvelle classe en utilisant les méthodes de la super-classe

Par exemple

```
class DroneTest {

    @Test
    void testInitialiseDrone() {
        Drone drone = new Drone(30, 0.2);
        assertEquals(30, drone.getBatteryCapacity());
        assertEquals(0, drone.getCurrentCharge());
        assertEquals(0.2, drone.getEnergyConsumptionPerKilometer());
    }
}
```

Vous avez à présent une classe Drone, que nous avons faiblement testé. Vous pouvez reprendre des tests réalisés sur la classe *ElectricVehicle* pour les appliquer sur un drone.

III.2 La classe ElectricBike : bases de l'héritage avec de l'aide

III.2.1 Hériter des codes de la super classe

Q.4. Vous devez implémenter une classe *ElectricBike* qui représente un vélo électrique.

La classe *ElectricBike* étend la classe *ElectricVehicle*.

III.2.2 Constructeurs hérités et ajout de variables d'instances

Q.5. Vous devez ajouter à la classe *ElectricBike* les codes correspondants aux spécifications suivantes.

- Le ***niveau d'assistance*** au pédalage du vélo électrique détermine la force du soutien du moteur lorsque le cycliste pédale. Un niveau plus élevé implique un soutien plus puissant du moteur, entraînant une consommation d'énergie accrue pour la même distance.
  - Par défaut, le niveau est à 0.
- Le ***niveau peut être ajusté en temps réel.***
  - Si une valeur de niveau supérieure au niveau maximal est fournie, le niveau est réglé sur le niveau maximal autorisé. Par exemple, si un vélo a trois niveaux d'assistance, si vous demandez à passer au niveau 5, le vélo passera au niveau 3.
  - Si une valeur de niveau inférieur à 0 est fournie, le niveau est réglé sur le niveau 0.
- Il existe une ***relation entre les niveaux d'assistance au pédalage à la consommation d'énergie (en kilowatt-heures par kilomètre).***
  - Vous pouvez représenter cette relation par un tableau
    - Par exemple double[] energyConsumptionLevels = {0.1, 0.2, 0.3, 0.4}; signifie que
      - Il y a 4 niveaux d'assistances.
      - Le niveau 0 est quand même consommateur.
      - Le niveau maximum est le niveau 2.
      - Pour le niveau d'assistance 0, la consommation d'énergie est de 0.1 KW par km ; pour le niveau d'assistance 1, la consommation d'énergie est de 0.2 KW par km, … pour le niveau d'assistance 3, la consommation d'énergie est de 0.4 KW par km.

- **A la construction du vélo**, on précise le niveau de batterie et la relation entre les niveaux d'assistance et l'énergie consommée au km.

```
public ElectricBike(double batteryCapacity, double[] energyConsumptionLevels)
{
    super(batteryCapacity);
    this.energyConsumptionLevels = energyConsumptionLevels;
}
```

- La méthode *getEnergyConsumptionForAssistLevel* retourne l'énergie la consommation d'énergie en fonction du niveau d'assistance passé en paramètre. Elle correspond à une simple lecture dans le tableau.

Q.6. Vous testez la classe *ElectricBike*

a. Les codes suivants vous montrent seulement comment utiliser les tableaux à l'initialisation. Vous les complétez pour tester votre classe.

Par exemple

```
class ElectricBikeTest {

    ElectricBike bike;
    final int  batteryCapacity = 30;
    @BeforeEach
    void setUp() {
        bike = new ElectricBike(batteryCapacity, new double[]{0.2, 0.5, 0.8});
    }

    @org.junit.jupiter.api.Test
    void testInitialise() {
        assertEquals(0, bike.getCurrentCharge());
        assertEquals(0, bike.getPedalAssistLevel());
    }

    @org.junit.jupiter.api.Test
    void testMaxRangeOfBike() {
        bike = new ElectricBike(batteryCapacity, new double[]{0.2, 0.5, 0.8});
        assertEquals(0, bike.getCurrentCharge());
        assertEquals(0, bike.getPedalAssistLevel());
        assertEquals(3,bike.getNumberOfAvailableLevels());

        assertEquals(0.2, bike.getEnergyConsumptionPerKilometer());

        assertEquals(0.5, bike.getEnergyConsumptionForAssistLevel(1));
        assertEquals(0.8, bike.getEnergyConsumptionForAssistLevel(2));

        assertEquals(150, bike.calculateMaxRange());
        bike.setPedalAssistLevel(1);
        assertEquals(1, bike.getPedalAssistLevel());
…
```

III.2.3 Surcharger (overriding) des méthodes de la super-classe.

Nous souhaitons que le niveau de consommation d'énergie dépende du niveau d'assistance courant. Pour cela nous surchargeons la méthode *getEnergyConsumptionPerKilometer,* c'est-à-dire que nous masquons la méthode définie dans la classe *ElectricVehicule* par une méthode de même nom qui, en fonction du niveau d'assistance courant, récupère le niveau de consommation d'énergie.
Ainsi un appel à la méthode *getEnergyConsumptionPerKilometer* sur un vélo électrique n'aura pas le même comportement que si vous appelez exactement la même méthode sur un drone.

```
@Override
public double getEnergyConsumptionPerKilometer() {
    return getEnergyConsumptionForAssistLevel(pedalAssistLevel);
}
```

Q.7. Surchargez la méthode *getEnergyConsumptionPerKilometer* comme définie ci-dessus.

Q.8. Testez vos codes. En particulier, on s'attend à présent à ce que les calculs de distance varient en fonction du niveau d'assistance.

Par exemples

```
ElectricBike bike =
   new ElectricBike(batteryCapacity, new double[]{energyConsumptionPerKilometer, 0.5, 0.8});
assertEquals(0, bike.getPedalAssistLevel());
assertEquals(0.2, bike.getEnergyConsumptionPerKilometer());
assertEquals(150, bike.calculateMaxRange());

bike.setPedalAssistLevel(1);
assertEquals(1, bike.getPedalAssistLevel());
assertEquals(0.5, bike.getEnergyConsumptionPerKilometer());
assertEquals(batteryCapacity / 0.5, bike.calculateMaxRange());
```

Q.9. Prenez quelques minutes pour analyser vos codes :

a. Quelle est la super-classe ?
b. Quelles sont ses sous-classes ?
c. Quelles méthodes avez-vous surchargé ? ajouté ?
d. A quels messages peut répondre un drone ? un vélo électrique ?

III.3 Héritage de tests (facultatif)

Q.10. Reprenez vos tests sur un véhicule électrique pour les appliquer à un vélo électrique

a. Pour cela, vous déclarez la classe de tests comme héritant de la classe de tests sur les véhicules électriques
b. Vous surchargez la méthode *setup* et vous mettez à jour la variable *ev*.
c. Quand vous lancez vos tests, les tests de la super-classe s'exécutent sur le vélo.

```
class ElectricBikeTest extends fr.epu.vehicles.ElectricVehicleTest {

    ElectricBike bike;
    final int  batteryCapacity = 30;

    @BeforeEach
    void setUp() {
        bike = new ElectricBike(batteryCapacity, new double[]{0.2, 0.5, 0.8});
        ev = bike;
    }
```

Q.11. Complétez les tests avec des tests spécifiques aux méthodes de `ElectricBike`

III.4 La classe *ElectricCar* : Héritage en action (à faire en autonomie)

La classe Java appelée *ElectricCar* représente une voiture électrique.

a) Cette classe est une sous-classe de la classe *ElectricVehicle*.
b) Elle définit un attribut *coolingSystemActive* de type ***boolean*** qui représente l'état du système de refroidissement (actif ou inactif).
c) Elle définit le facteur d'impact du système de refroidissement sur la consommation d'énergie. Sa valeur est fixée à 1.2.
d) Elle est identifiable par sa plaque d'immatriculation : private String licensePlate;

e) Elle est caractérisée par un modèle que l'on définit comme une simple chaine de caractère.
f) Le modèle est initialisé à la construction de la voiture sinon il est mis en « undefined ».
g) Elle expose un constructeur public qui prend en paramètre *batteryCapacity* de type double et la plaque d'immatriculation qui est obligatoire également. Ce constructeur initialise la capacité de la batterie à l'aide du constructeur de la superclasse *ElectricVehicle*. Il initialise également l'attribut *coolingSystemActive* à false pour indiquer que le système de refroidissement est inactif par défaut.
h) Elle définit les méthodes :
    a. *turnOnCoolingSystem()* : Une méthode publique qui active le système de refroidissement de la voiture en affectant *coolingSystemActive* à true.
    b. *turnOffCoolingSystem()* : Une méthode publique qui désactive le système de refroidissement de la voiture en affectant *coolingSystemActive* à *false*.
    c. *getEnergyConsumptionPerKilometer()* : Une méthode qui override (surcharge en masquant) la méthode de la superclasse pour calculer la consommation d'énergie par kilomètre. Si le système de refroidissement est actif, la consommation d'énergie est augmentée de 20% en utilisant le facteur (constante) *COOLING_SYSTEM_FACTOR*.
    d. *isOnCoolingSystem()* : Une méthode publique qui retourne true si le système de refroidissement est actif, sinon false.

Q.12. Utilisez l'héritage pour hériter des fonctionnalités de la classe ElectricVehicle.
Q.13. Pensez à encapsuler les attributs appropriés et utilisez des méthodes publiques pour accéder à ces attributs lorsque c'est nécessaire.
Q.14. Testez soigneusement chaque méthode pour vous assurer qu'elle fonctionne comme prévu.

IV. Héritage : sous-typage

IV.1 Utilisation du sous-typage : des stations de charge multi-véhicules sans rien faire !

Un des grands avantages de l'héritage est de pouvoir manipuler les objets de la hiérarchie de la même façon.

Nous ne modifions pas du tout le code de *ChargingStation* écrit au précédent TD et qui a été défini sur des *ElectricVehicle*.

Q.15. Voici une série de tests additionnels sur *ChargingStation*
    a. Pourquoi fonctionnent-ils ?
    b. Comprenez-vous les deux versions qui font également la même chose ? Pourquoi préférons-nous la deuxième ?

```
class ChargingStationTest {

    @Test
    void testInitialiseChargingStation() {
```

```
    … VOIR CODES DONNES  au TD1  }

@Test
void testConnect() {
    … VOIR CODES DONNES  au TD1  }
```

```
    @Test
    void testInheritance(){
        ChargingStation chargingStation = new ChargingStation("Charging Station 1", 10);
        ElectricVehicle ev = new ElectricVehicle(30);
        double charge = chargingStation.connectToChargingPoint(ev);
        assertEquals(ev.getBatteryCapacity(), ev.getCurrentCharge());

        ElectricCar ec = new ElectricCar(30, "ABC123");
        charge = chargingStation.connectToChargingPoint(ec);
        assertEquals(ec.getBatteryCapacity(), ec.getCurrentCharge());

        ElectricBike eb = new ElectricBike(30, new double[]{0.1,0.2,0.5});
        charge = chargingStation.connectToChargingPoint(ec);
        assertEquals(ec.getBatteryCapacity(), ec.getCurrentCharge());

        ev = ec;
        //Forbidden : ec = ev;
        //Forbidden : ec = eb;

    }

    @Test
    void testInheritanceSecondVersion(){
        ChargingStation chargingStation = new ChargingStation("Charging Station 1", 10);
        ElectricVehicle ev = new ElectricVehicle(30);
        testIsFullCharged(chargingStation, ev);
        ElectricCar ec = new ElectricCar(30, "ABC123");
        testIsFullCharged(chargingStation, ec);
        ElectricBike eb = new ElectricBike(30, new double[]{0.1,0.2,0.5});
        testIsFullCharged(chargingStation, eb);
    }

    private static void testIsFullCharged(ChargingStation chargingStation,
                                          ElectricVehicle ev) {
        double charge = chargingStation.connectToChargingPoint(ev);
        assertEquals(ev.getBatteryCapacity(), ev.getCurrentCharge());
    }

}
```

IV.2 Un service de véhicules électriques dans le package *fr.epu.fleets*

C'est le seul exercice qui peut ne pas être terminé en TD, si vous n'avez pas le temps. Cependant, il permet de renforcer vos connaissances sur les tableaux, et de vous entrainer à développer des codes qui mettent en jeux de multiples classes.

Votre mission consiste à développer un système de gestion de véhicules électriques pour une entreprise. Vous devez créer la classe ***VehicleService*** qui gérera les véhicules disponibles et les véhicules en réparation.

Spécifications de la classe *VehicleService* :

1. La classe *VehicleService* doit avoir deux tableaux pour stocker les véhicules :
   - *availableVehicles* : Un tableau de véhicules électriques disponibles.
   - *vehiclesInRepair* : Un tableau de véhicules électriques en cours de réparation.

2. La classe doit inclure un constructeur qui initialise les deux tableaux de véhicules avec une taille maximale spécifiée.

3. Elle implémente les méthodes suivantes :

- *addAvailableVehicle* : Cette méthode prend un véhicule électrique comme argument et l'ajoute au tableau *availableVehicles* s'il y a de la place.

- *addVehicleInRepair* : Cette méthode prend un véhicule électrique comme argument et l'ajoute au tableau *vehiclesInRepair* s'il y a de la place.

- *moveVehicleToRepair* : Cette méthode permet de déplacer un véhicule électrique du tableau *availableVehicles* au tableau *vehiclesInRepair* pour indiquer qu'il est en réparation.

- *moveVehicleToAvailable* : Cette méthode permet de déplacer un véhicule électrique du tableau *vehiclesInRepair* au tableau *availableVehicles* pour le remettre à disposition.

- *findAvailableVehicleWithMaxDistance* : Cette méthode doit rechercher le véhicule disponible avec la plus grande autonomie possible, en fonction de sa charge de batterie actuelle, et le retourner.

- *findVehicleWithMaxRangeInAvailableAndInRepair* : Cette méthode doit rechercher le véhicule (disponible ou en réparation) avec la plus grande autonomie possible en fonction de la capacité de sa batterie, puis le retourner.

- *getNbOfVehiclesInRepair* : Cette méthode renvoie le nombre de véhicules électriques en réparation.

- getNbOfAvailableVehiclesInCharge : Cette méthode renvoie le nombre de véhicules disponibles qui sont *connectés*.

- getNbOfAvailableVehicles : Cette méthode renvoie le nombre total de véhicules électriques disponibles.

Q.16. Implémentez la spécification ci-dessus dans un nouveau package *fr.epu.fleets*

Q.17. Mettez en place des tests

a. Avez-vous pensé dans le cas d'un déplacement à tester le cas où le véhicule n'est pas présent ?
b. N'oubliez pas de tester vos tableaux après avoir déplacé des objets…

V. Conclusion

À l'issue de ce TD vous devez :

- Avoir acquis les rudiments de la programmation de boucles et la manipulation de tableaux en java ;
- Maitriser les bases de l'héritage et savoir l'utiliser en java ;
- Savoir écrire des tests simples ;
- Organiser votre code : (voir figure ci-dessous)
  - En package
  - En séparant code source et tests.
- Comprendre la structure de votre code (vous pouvez générer le même diagramme que celui présenté ci-dessous et vous devez avoir la capacité de l'expliquer).

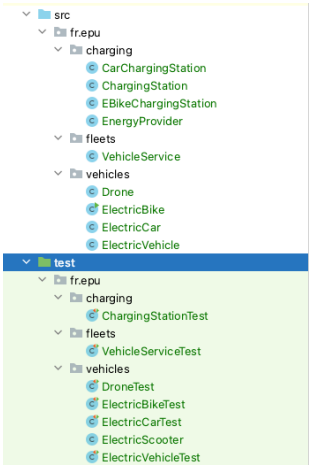


Figure 1 : Ne tenez pas compte de CarChargingStation, EBikeChargingStation et ElectricScooter : elles ne vous sont pas demandées.