TD1 : Prise en main de l'environnement IntelliJ

I. Objectifs du TD

Une maîtrise solide de votre environnement de programmation est essentielle. Lorsque vous êtes à l'aise avec vos outils, tels que les fonctionnalités de votre environnement, vous pouvez vous concentrer pleinement sur la création et le développement, sans vous soucier de problèmes techniques. Cette maîtrise libère votre esprit pour l'essentiel : la résolution de problèmes et la création de logiciels de qualité.

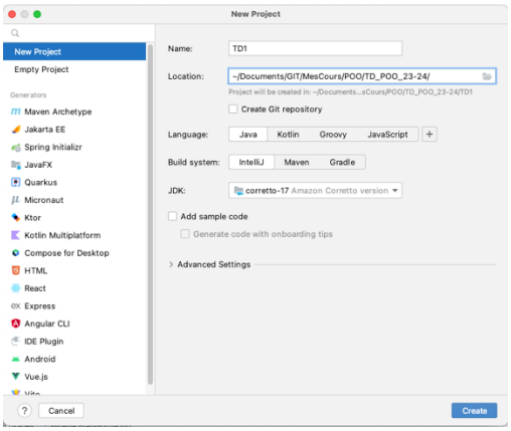
1. La prise en main et l'installation des outils pour le développement en Java.
2. Les petits codes développés seront réutilisés au TD suivant.

II. Environnement de programmation

II.1 Installer

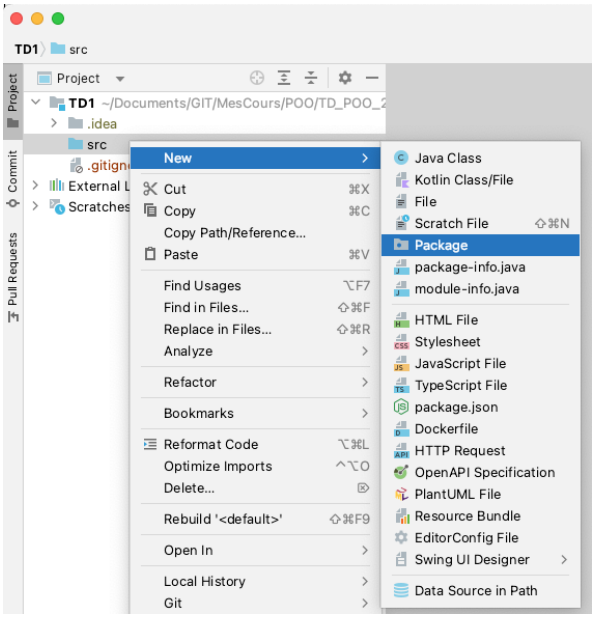
Q.1. Commencer si cela n'est pas déjà fait par installer votre environnement de travail

a. Installation de l'Environnement et familiarisation
b. Attention la version de java utilisée est la 17.0.4

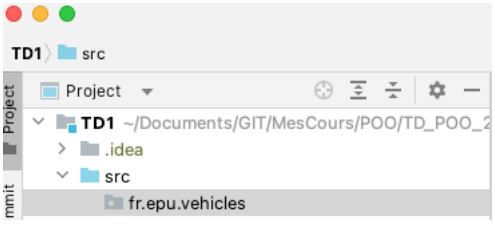


II.2 Créer un package : *fr.epu.vehicles*

Q.2. Clic droit sur le projet et créer un package.



Voici la structure de votre projet



II.3 Créer une classe *ElectricVehicle*

Q.3. Clic droit sur le package et créer une java class

Voici la structure initiale de la classe

```
public class ElectricVehicle {
        private double batteryCapacity;  // Capacity of the battery in kilowatt-hours
        private double currentCharge;    // Current charge level of the battery
```

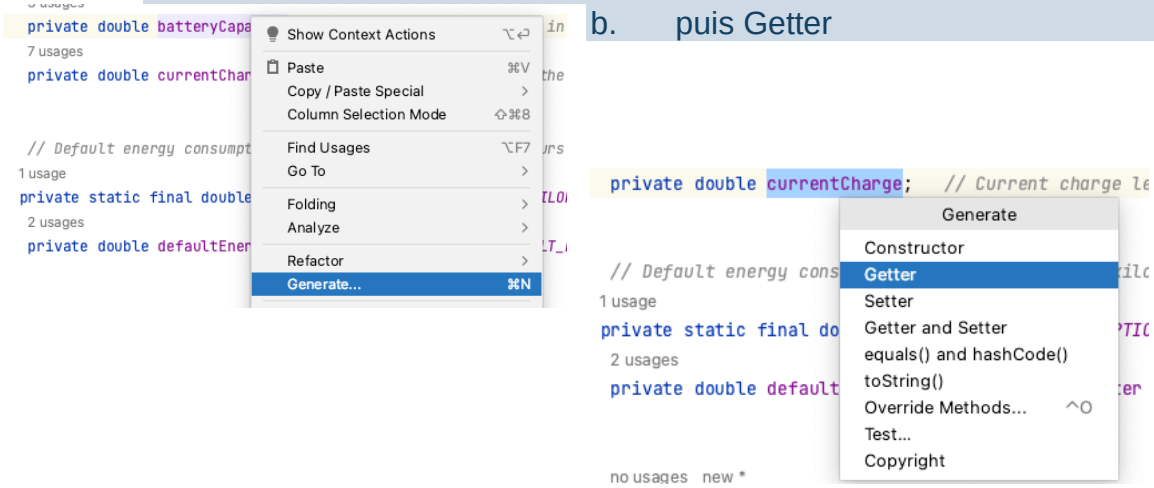
II.4 Générer les accesseurs

Nous souhaitons à présent accéder à la charge actuelle de la batterie. Nous ne souhaitons pas que l'on puisse modifier la charge. Il s'agit donc de générer un getter très simple.

Q.4. Clic droit sur la variable d'instance

a. Generate

b. puis Getter



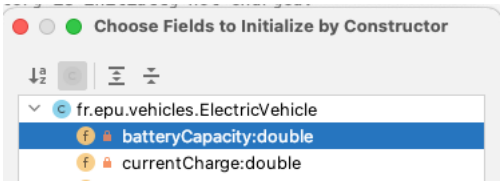
Q.5. De la même manière nous souhaitons avoir accès en lecture à la capacité de la batterie.

II.5 Générer un constructeur

Ajoutez un constructeur prenant en paramètre la capacité de la batterie pour l'initialiser. Le niveau de charge initial doit être défini à 0.

Q.6. Clic droit dans la classe, *generate*, puis *constructor ;*

a. Ne sélectionnez ensuite que la variable batteryCapacity.
b. Complétez le constructeur.



II.6 Compléter votre code

La méthode *charge* prend en paramètre *chargeAmount*, c'est-à-dire, la quantité d'énergie à charger en kilowattheures. Si la somme du niveau de charge actuel et de la quantité à charger est inférieure ou égale à la capacité de la batterie, la charge est effectuée avec succès et la méthode renvoie true. Dans le cas contraire, la capacité de la batterie serait dépassée par cet ajout, la méthode renvoie false et la charge de la batterie n'est pas modifiée.

Q.7. Implémenter la méthode *charge* qui respecte la spécification donnée ci-dessus.

⚠ Pour les étudiants qui ne parviendraient pas à écrire cette méthode *(à moins que vous ne soyez en harmonisation, attention, cet exercice est considéré comme extrêmement facile !) voici le code de cette méthode.*

```
/**
 * Charge the battery with the given amount of energy.
 * @param chargeAmount
 * @return true if the battery was charged, false otherwise
 */
public boolean charge(double chargeAmount) {
        boolean success = false;
        if (currentCharge + chargeAmount <= batteryCapacity) {
            currentCharge += chargeAmount;
            success = true;
        } else {
            success = false;
        }
        return success;
}
```

Q.8. Ajouter une méthode `chargeToFull()` qui permet de charger le véhicule à sa capacité maximum et retourne la quantité d'énergie qui a été ajoutée.

⚠ Pour les étudiants qui ne parviendraient pas à écrire cette méthode …

```
/**
 * Charge the battery to full capacity.
 * @return the amount of energy that was added to the battery
 */
public double chargeToFull() {
```

```
        double chargeAmount = batteryCapacity - currentCharge;
        charge(chargeAmount);
        return chargeAmount;
    }
```

Q.9. Ajouter dans votre classe toutes les informations nécessaires pour respecter la spécification suivante.

Nous souhaitons introduire le concept d'Energy Consumption Per Kilometer (Consommation d'Énergie par Kilomètre). (ii) Nous souhaitons donc pouvoir fournir à l'initialisation d'un véhicule électrique sa consommation moyenne d'énergie quand nous la connaissons. Dans le cas contraire, nous affectons (iii) une valeur de 0,2 soit 20 kWh pour parcourir 100 kilomètres. (i) Cette valeur doit évoluer en fonction de la conduite aussi n'est-elle accessible que par (iv) une méthode dédiée *getEnergyConsumptionPerKilometer.*

Pour les débutants :

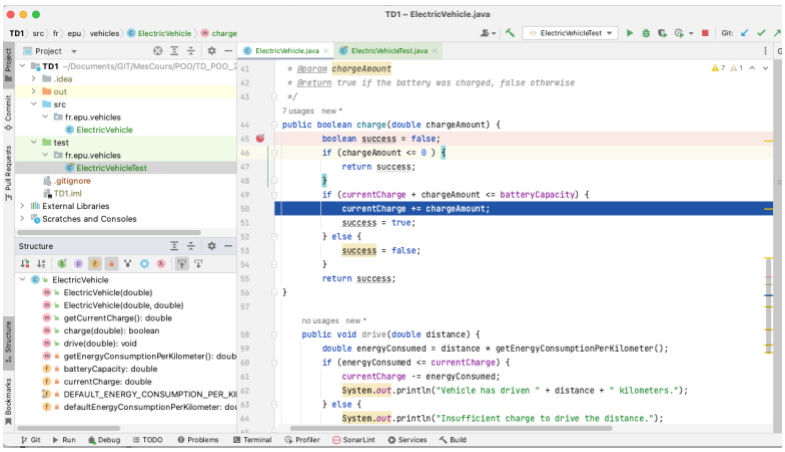
(i) Ajouter une variable d'instance de type double
(ii) Créer un constructeur qui prend en paramètre ECpK
(iii) Affecter la valeur par défaut de 0,2 lorsque ECPK n'est pas donnée.
(iv) Créer une méthode *getEnergyConsumptionPerKilometer* qui retourne le contenu de ECpK

II.7 Naviguer

Vous êtes dans un environnement de développement qui est très puissant et, comme vous l'avez déjà vu, vous offre plein de facilités pour rendre votre développement plus aisé. Nous n'avons pas le temps dans ce TD de tout explorer, mais vous pouvez explorer directement les tutoriels de IntelliJ.

Q.10. Remarquez cependant

a. La partie *Projet* qui vous montre les classes et packages qui composent votre projet
b. La partie Structure qui vous permet d'accéder directement à une méthode par exemple.

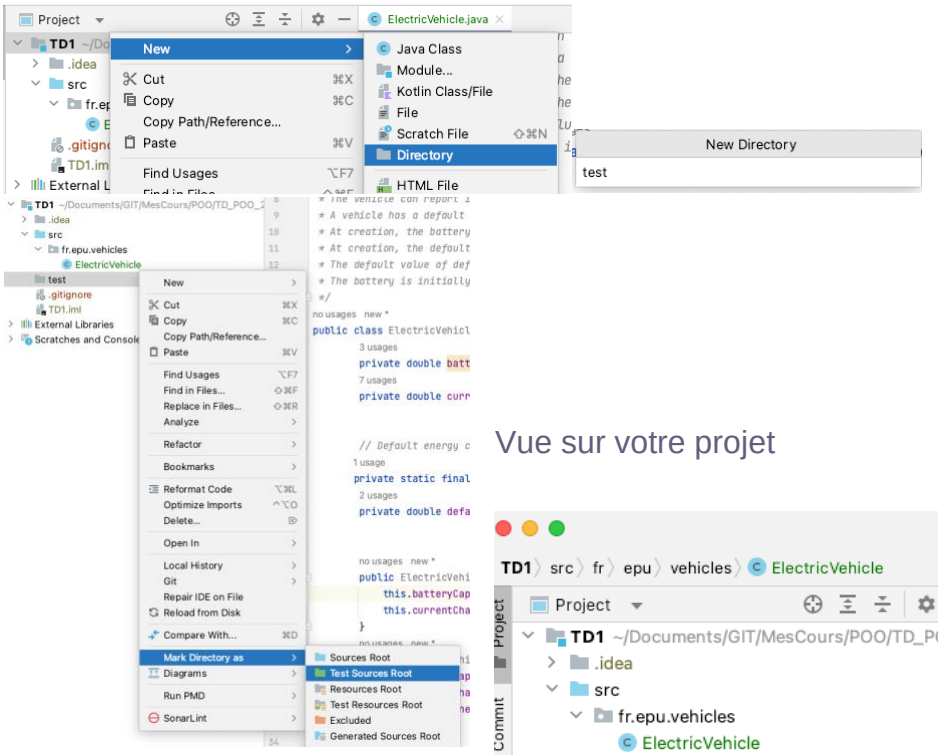


III. Tester

III.1 Créer un répertoire de tests

Q.11. Clic droit sur le projet,

a. new, puis directory ;
b. Nommer le répertoire **test**
c. Lui donner la nature 'test source root'

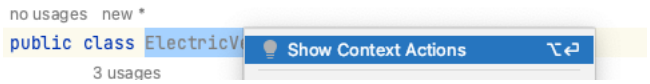


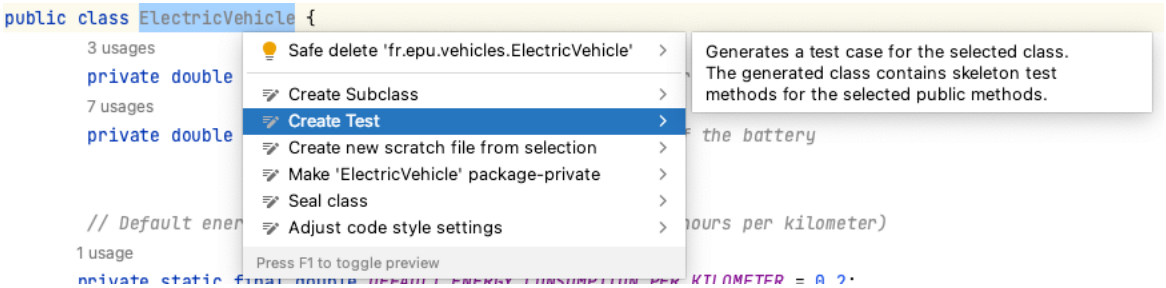
Vue sur votre projet

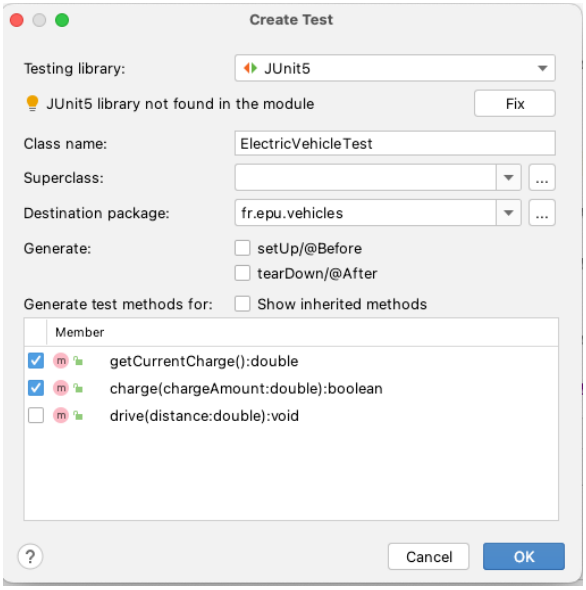
III.2 Générer un test

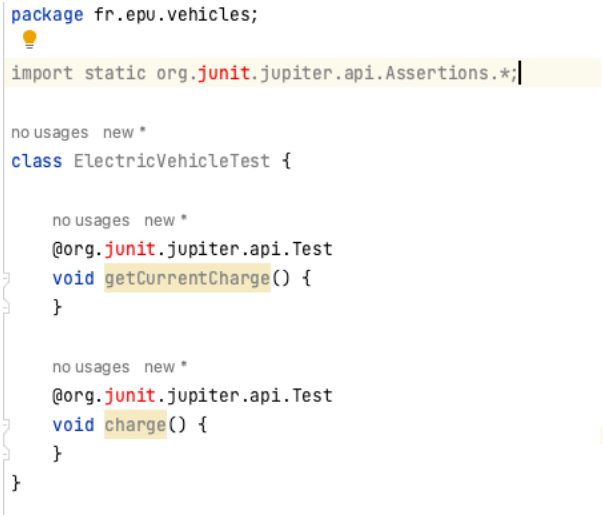
Q.12. Clic droit dans la classe

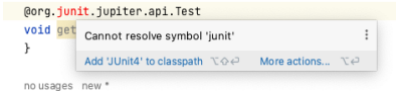
a. Show context actions
b. Create Test
c. Sélectionner la méthode à tester et **choisissez bien junit5**
d. **Le code généré signale des erreurs**
    i. Positionnez votre curseur sur le texte en rouge
    ii. Sélectionner le texte en rouge, puis show context actions et sélectionner bien Junit 5.8.1 à ajouter au classpath.
    iii. Le code n'est plus en erreur.

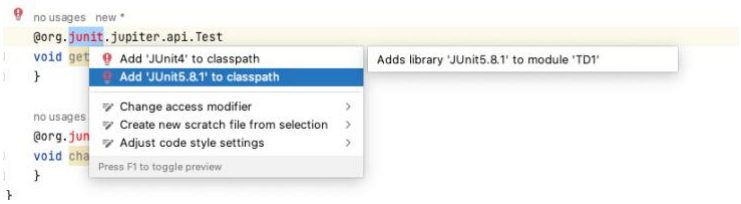

no usages new *
public class ElectricV
Show Context Actions
3 usages

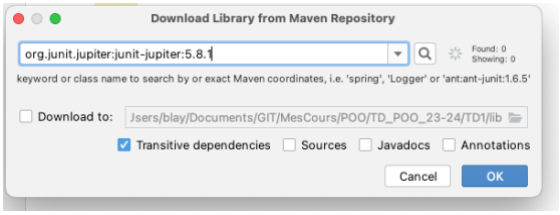

public class ElectricVehicle {
3 usages
private double
7 usages
private double
// Default ener
1 usage
Safe delete 'fr.epu.vehicles.ElectricVehicle'
Create Subclass
Create Test
Create new scratch file from selection
Make 'ElectricVehicle' package-private
Seal class
Adjust code style settings
Press F1 to toggle preview
Generates a test case for the selected class.
The generated class contains skeleton test
methods for the selected public methods.


Create Test
Testing library:
JUnit5
JUnit5 library not found in the module
Fix
Class name:
ElectricVehicleTest
Superclass:
Destination package:
fr.epu.vehicles
Generate:
setUp/@Before
tearDown/@After
Generate test methods for:
Show inherited methods
Member
getCurrentCharge():double
charge(chargeAmount:double):boolean
drive(distance:double):void
Cancel
OK


package fr.epu.vehicles;
import static org.junit.jupiter.api.Assertions.*;
no usages new *
class ElectricVehicleTest {
no usages new *
@org.junit.jupiter.api.Test
void getCurrentCharge() {
}
no usages new *
@org.junit.jupiter.api.Test
void charge() {
}
}


@org.junit.jupiter.api.Test
Cannot resolve symbol 'junit'
Add 'JUnit4' to classpath
More actions...
no usages new *


no usages new *
@org.junit.jupiter.api.Test
Add 'JUnit4' to classpath
Add 'JUnit5.8.1' to classpath
Adds library 'JUnit5.8.1' to module 'TD1'
Change access modifier
Create new scratch file from selection
Adjust code style settings
Press F1 to toggle preview



Q.13. Implémenter les tests.

a. Pour cela vous pouvez recopier les codes ci-dessous, MAIS essayez de comprendre ce qu'ils font.

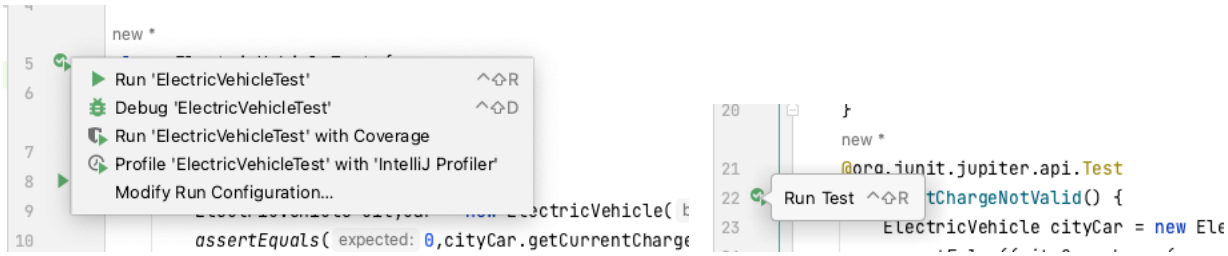
```
package fr.epu.vehicles;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class ElectricVehicleTest {
    ElectricVehicle ev;

    @BeforeEach
    void setUp() {
        ev = new ElectricVehicle(30);
    }

    @org.junit.jupiter.api.Test
    void testInitialiseVE() {
        assertEquals(30, ev.getBatteryCapacity());
        assertEquals(0, ev.getCurrentCharge());
        //System.out.println(ev);
    }

    @org.junit.jupiter.api.Test
    void testChargeValid() {
        assertTrue((ev.charge(     10)));
        assertEquals(10,ev.getCurrentCharge());
        assertTrue((ev.charge(     20)));
        assertEquals(30,ev.getCurrentCharge());
    }
    @org.junit.jupiter.api.Test
    void testChargeNotValid() {
        assertFalse((ev.charge(     100)));
        assertEquals(0,ev.getCurrentCharge());
        assertTrue((ev.charge(     10)));
        assertEquals(10,ev.getCurrentCharge());
        assertFalse((ev.charge(     21)));
        assertEquals(10,ev.getCurrentCharge());
    }

    @org.junit.jupiter.api.Test
    void testCheckChargeParameter() {
        assertFalse((ev.charge(-10)));
        assertEquals(0, ev.getCurrentCharge());
    }
    @org.junit.jupiter.api.Test
    void testChargeWith0() {
        assertFalse((ev.charge(0)));
        assertEquals(0, ev.getCurrentCharge());
    }

    @Test
    void testChargeToFull() {
        double charge = ev.chargeToFull();
        assertEquals(30, ev.getCurrentCharge());
        assertEquals(30, charge);
```

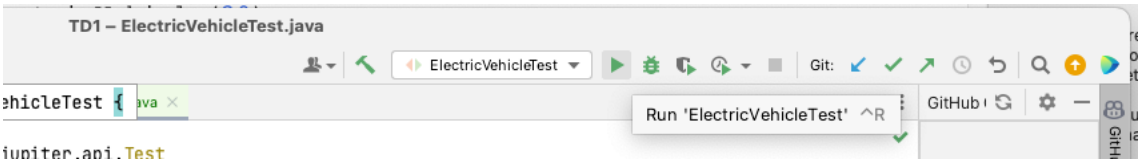
```
        ev = new ElectricVehicle(30);
        ev.charge(10);
        charge= ev.chargeToFull();
        assertEquals(30, ev.getCurrentCharge());
        assertEquals(20, charge);
    }

}
```

III.3 Lancer les tests

Q.14. Pour lancer les tests vous pouvez (voir captures d'écran)

a. Soit cliquer sur la flèche verte à côté du nom de la classe, vous lancerez alors tous les tests définis dans cette classe
b. Soit cliquer sur la flèche à côté d'une méthode
c. Soit en haut, toujours sur la flèche verte.





Q.15. Visualisez les résultats de vos tests s'affichent en bas de votre écran.

III.4 Visualiser les tests en erreur

Q.16. Vous devez comprendre les messages d'erreurs

a. Ajoutez les tests suivants
b. Lancez les tests
c. Pourquoi sont-ils en erreur ? Quel est le message d'erreur ?
d. Remarquez que la ligne en erreur vous est signalée et qu'un simple clic vous amène sur la ligne.

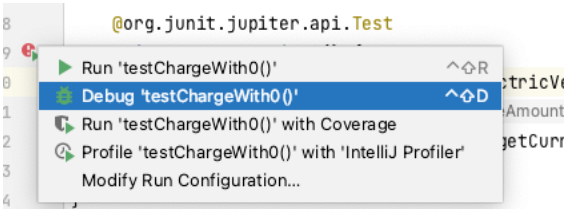
```
@org.junit.jupiter.api.Test
void testCheckChargeParameter() {
    ElectricVehicle cityCar = new ElectricVehicle(30);
    assertFalse((cityCar.charge(-10)));
    assertEquals(0, cityCar.getCurrentCharge());
}
@org.junit.jupiter.api.Test
void testChargeWith0() {
    ElectricVehicle cityCar = new ElectricVehicle(30);
    assertFalse((cityCar.charge(0)));
    assertEquals(0, cityCar.getCurrentCharge());
}
```

Dans cet exemple, nous n'avons pas besoin du débugger pour déterminer quel est l'erreur. Cependant pour vous aider à maitriser votre environnement nous lançons le débugger.

III.5 Debugger

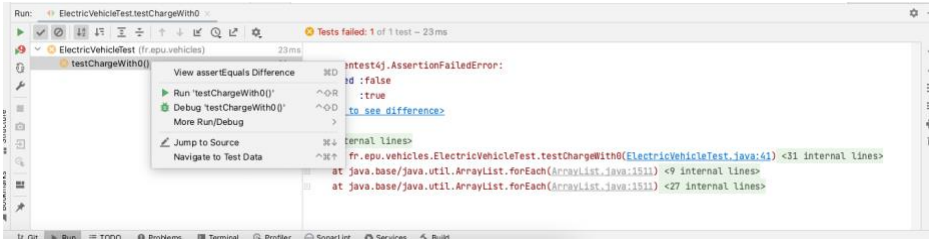
Vous avez un tutoriel ICI. Nous ne faisons que survoler ce point.

Pour lancer le débugger vous pouvez le faire par les mêmes moyens que ceux avec lesquels vous lancer les tests mais en sélectionnant Debug qui est symbolisé par un « Bug ».



Nous faisons le choix de lancer le débugger directement dans la fenêtre de test.

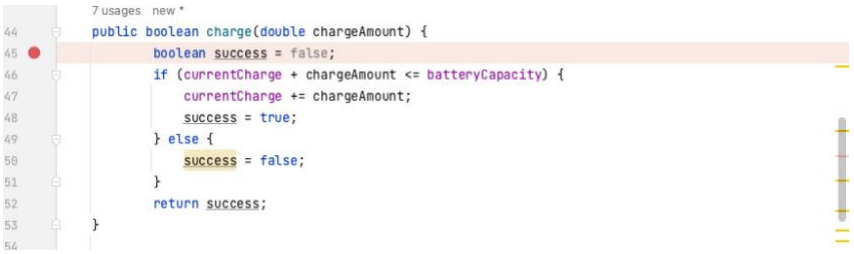
Q.17. A gauche de la fenêtre, cliquez sur le test en erreur, puis sélectionnez debug.

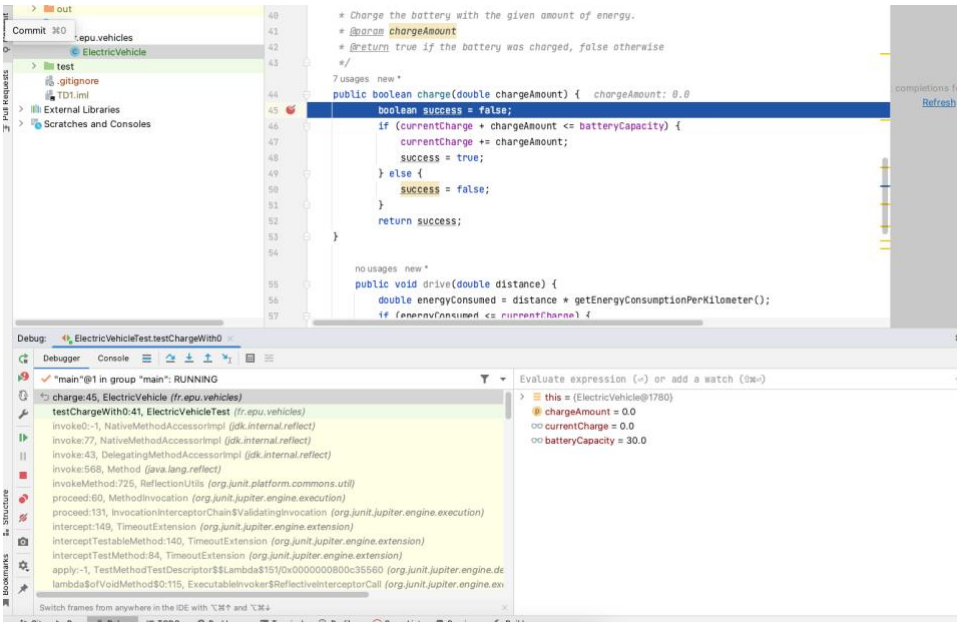


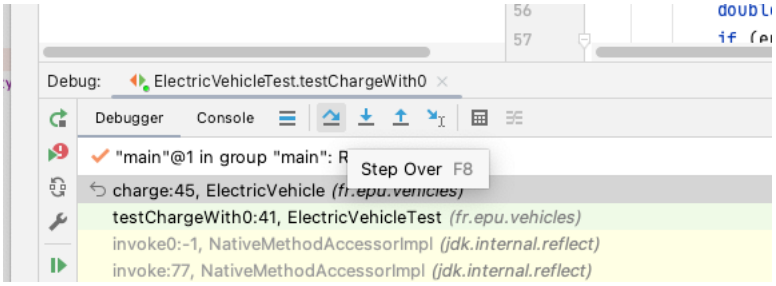
Cela n'a rien changé.

Q.18. Pour débugger vous devez préciser où l'exécution doit s'arrêter. Pour cela, placez un point d'arrêt dans notre programme.

a. Un simple clic ajoute un point d'arrêt à partir de la ligne de code où nous souhaitons commencer à tracer notre programme (cf. copie d'écran)
b. Relancez le test. Il s'arrête sur votre point d'arrêt. Vous avez alors accès au contenu des variables en bas à droite.
c. Pour passer à l'instruction suivante : en bas à gauche, flèche brisée (*step over)*





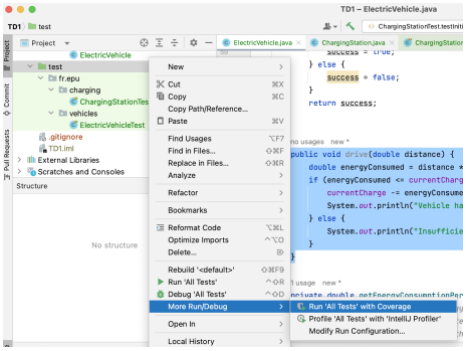


Q.19. Corrigez votre code et faîtes passer les tests.

III.6 Couverture de Tests

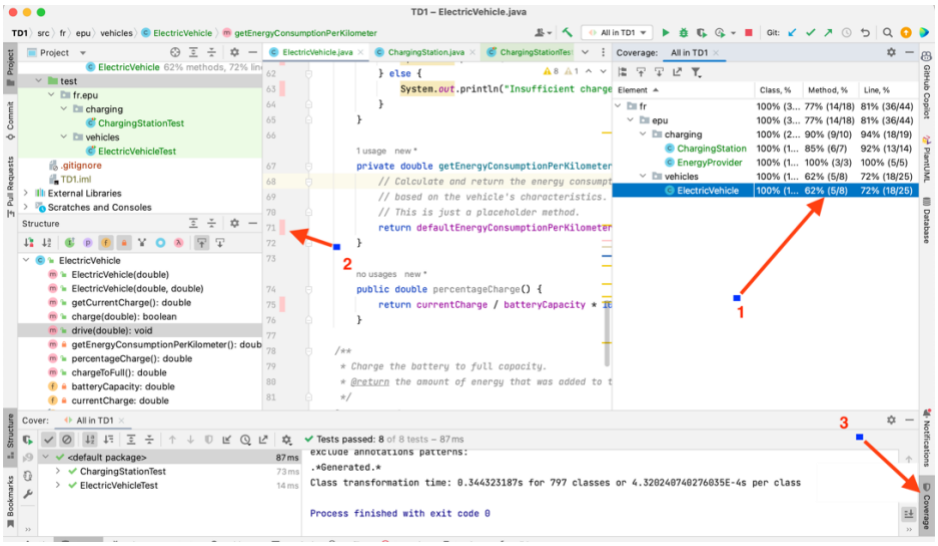
*En génie logiciel, la **couverture de code** est une mesure utilisée pour décrire le taux de code source exécuté d'un programme quand une suite de test est lancée. Un programme avec une haute couverture de code, mesurée en pourcentage, a davantage de code exécuté durant les tests ce qui laisse à penser qu'il a moins de chance de contenir de bugs logiciels non détectés, comparativement à un programme avec une faible couverture de code (Wikipedia)*

Q.20. Lancez les tests avec la couverture de tests comme ci-dessous



Q.21. Analysez les résultats

a. Le point 1 vous montre la métrique associée à la couverture de tests, classe par classe.
b. PAS FINI



IV. Vos codes dans le système de fichiers

Comme vous le savez déjà les classes sont définies dans un fichier qui porte leur nom + « .java ». Nous allons ajouter du code et visualiser ensuite la structure dans le répertoire.

Nous étendons notre étude à la gestion de l'énergie. Nous allons donc définir des producteurs d'énergie, des stations de recharge et un réseau de stations de recharge. En connectant une station à la voiture on peut la charger complétement.

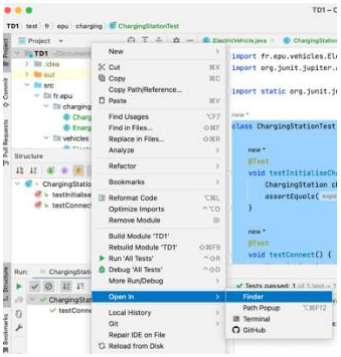
Q.22. Créez un package de nom *fr.epu.charging*

Q.23. Créez la classe *EnergyProvider*. Vous avez le code ci-après mais prenez quand même le temps de le comprendre.

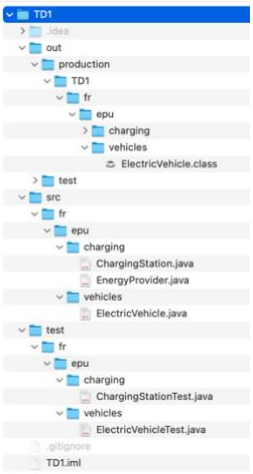
```
package fr.epu.charging;

public class EnergyProvider {
        private String providerName;
        private String energySource; // Ex: Solar, Wind, Grid, etc.

        public EnergyProvider(String providerName, String energySource) {
            this.providerName = providerName;
            this.energySource = energySource;
        }

        public String getProviderName() {
            return providerName;
        }
        public String getEnergySource() {
            return energySource;
        }
    }
```

Q.24. Visualisez la structure fichier de vos codes.

a. Placez-vous sur votre projet
b. Cliquez sur *Open In*



Sous src vous trouvez vos codes sources, sous out les codes compilés et sous tests vos codes de test, toujours dans la hiérarchie.



V. Aides au développement

Attention dans cette partie, nous allons volontairement accélérer un peu le développement pour vous aider à identifier de nouveaux outils.

Q.25. Créez une classe ***ChargingStation*** en respectant la spécification suivante.

Elle définit les attributs **stationName** (nom de la station), **availableChargingPoints** (nombre de points de charge disponibles) et **energyProvider**.

Elle implémente les accesseurs associés.

Q.26. Associer des constructeurs adaptés. En l'absence d'information sur le fournisseur à la création d'une instance de ChargingStation, on lui affecte un nouveau fournisseur de nom « EDF » et de source « Solar ».

V.1 Génération de méthodes

Q.27. Ajouter la méthode suivante à la classe ***ChargingStation***

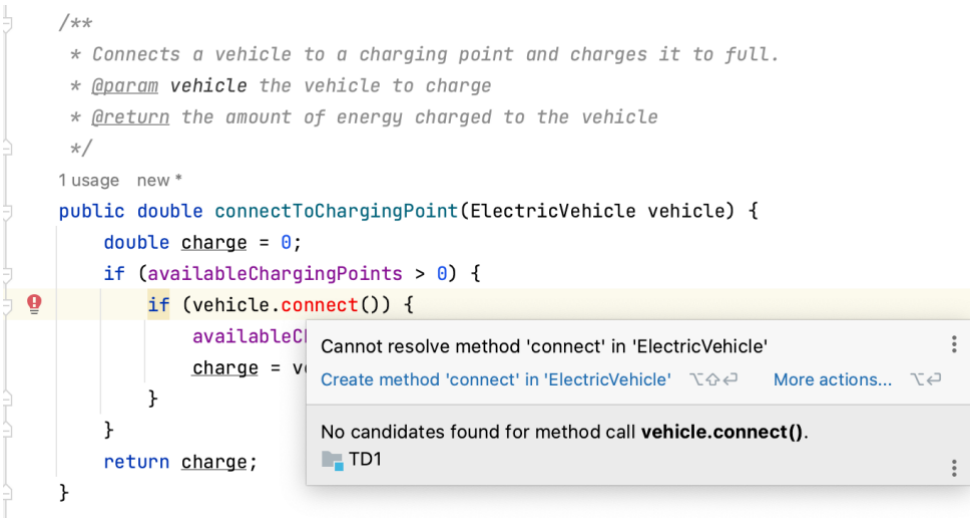
La méthode *connectToChargingPoint* permet de connecter un véhicule à un point de recharge et de le charger entièrement. Elle retourne la quantité d'énergie chargée dans le véhicule. Cette méthode vérifie d'abord s'il y a des points de recharge disponibles. Si c'est le cas, Elle connecte alors le véhicule. Si elle y parvient, elle décrémente le nombre de points de recharge disponibles puis utilise la méthode *chargeToFull()* du véhicule pour le charger entièrement. Finalement, elle retourne la quantité d'énergie chargée dans le véhicule.

Voici le code :

```java
/**
 * Connects a vehicle to a charging point and charges it to full.
 * @param vehicle the vehicle to charge
 * @return the amount of energy charged to the vehicle
 */
public double connectToChargingPoint(ElectricVehicle vehicle) {
    double charge = 0;
    if (availableChargingPoints > 0) {
        if (vehicle.connect()) {
            availableChargingPoints--;
            charge = vehicle.chargeToFull();
        }
    }
    return charge;
}
```

Q.28. Vous constatez que la méthode *connect* apparait en rouge et qu'une ampoule rouge est apparu au début de la ligne. **Pourquoi** ?

a. En passant avec le curseur sur l'erreur vous voyez le message suivant (voir figure)
b. Il est essentiel de comprendre pourquoi un code est en erreur.
c. Ici Vous voyez que sous l'erreur, IJ nous propose de générer pour nous la méthode manquante.
d. **Cliquer sur le lien en bleu** pour que la méthode *connect* soit générée dans la classe ElectricVehicle



Q.29. Dans la classe *ElectricVehicle* nous implémentons la connexion d'un véhicule électrique comme suit :

a. Nous ajoutons un attribut *isConnected* qui vérifie si le véhicule est ou non connecté
b. La méthode *connect* vérifie que la voiture n'est pas connectée puis la connecte. Sinon ielle renvoie faux.

La méthode suivante est créée dans la classe *ElectricVehicle*

```java
1 usage  new *
public boolean connect() {
}
```

Pour les débutants :

```
private boolean isConnected = false;
public boolean connect() {
    if (isConnected) {
        return false;
    }
    isConnected = true;
    return true;
}
```

Q.30. Dans la classe ***ChargingStation*** définit une méthode qui permet de déconnecter un véhicule : *disconnectFromChargingPoint.*

a. Elle déconnecte le véhicule et incrémente les points de charge.

Q.31. Testez vos classes.

a. Vous avez ci-dessous des esquisses de tests mais prenez le temps d'améliorer vos tests.

```
class ChargingStationTest {

    @Test
    void testInitialiseChargingStation() {
        ChargingStation chargingStation = new ChargingStation("Charging Station 1", 10);
        assertEquals(10, chargingStation.getAvailableChargingPoints());
    }

    @Test
    void testConnect() {
        ChargingStation chargingStation = new ChargingStation("Charging Station 1", 10);
        ElectricVehicle cityCar = new ElectricVehicle(30);
        double charge = chargingStation.connectToChargingPoint(cityCar);
        assertEquals(9, chargingStation.getAvailableChargingPoints());
        assertEquals(30, cityCar.getCurrentCharge());
        assertEquals(30, charge);
        chargingStation.disconnectFromChargingPoint();
        assertEquals(10, chargingStation.getAvailableChargingPoints());
    }
```

```
    @Test
    void testInitialiseChargingStation() {
        ChargingStation chargingStation =
      new ChargingStation("Charging Station 1", 10);
        assertEquals(10, chargingStation.getAvailableChargingPoints());
        EnergyProvider provider = chargingStation.getEnergyProvider();
        assertEquals("EDF", provider.getProviderName());
        assertEquals("Solar", provider.getEnergySource());
    }
```

Q.32. Dans la classe ElectricVehiculeTest, si le test vous pose un problème, vous savez quoi faire maintenant 😉

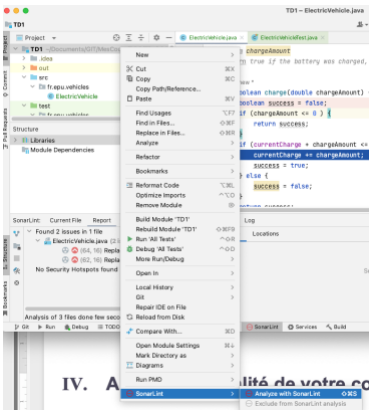
```
@Test
void testConnexions() {
    ElectricVehicle cityCar = new ElectricVehicle(30);
    assertFalse(cityCar.isConnected());
    assertTrue(cityCar.connect());
    assertTrue(cityCar.isConnected());
    assertFalse(cityCar.connect());
```

```
        assertTrue(cityCar.isConnected());
        cityCar.disconnect();
        assertFalse(cityCar.isConnected());
    }
```

VI. Analyser la qualité de votre code avec SonarLint

Q.33. Lancez SonarLint

a. Cliquez sur le projet puis tout en bas vous trouverez SonarLint (si bien sûr vous l'avez installé comme nous vous l'avons demandé).



Q.34. Analysez les retours qui vous sont faits.

Il est possible que vous n'ayez quasiment aucune erreur. Elles apparaissent en bas de votre écran.

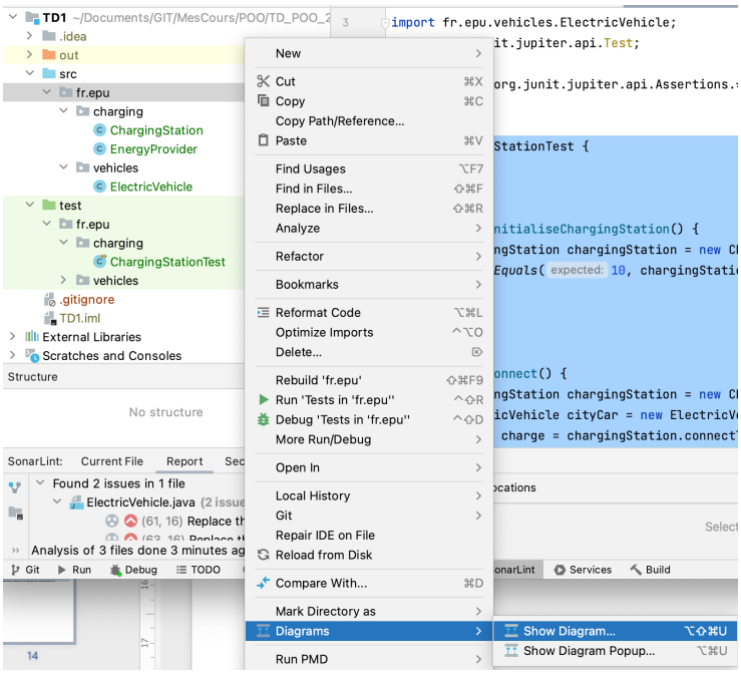
La partie à gauche vous montre les erreurs et quand vous les sélectionner la partie à droite vous donne des explications. Cet outil vous sera très outil au fur et à mesure que vous développerez vos codes.

VII. Visualiser vos codes sous forme d'un diagramme

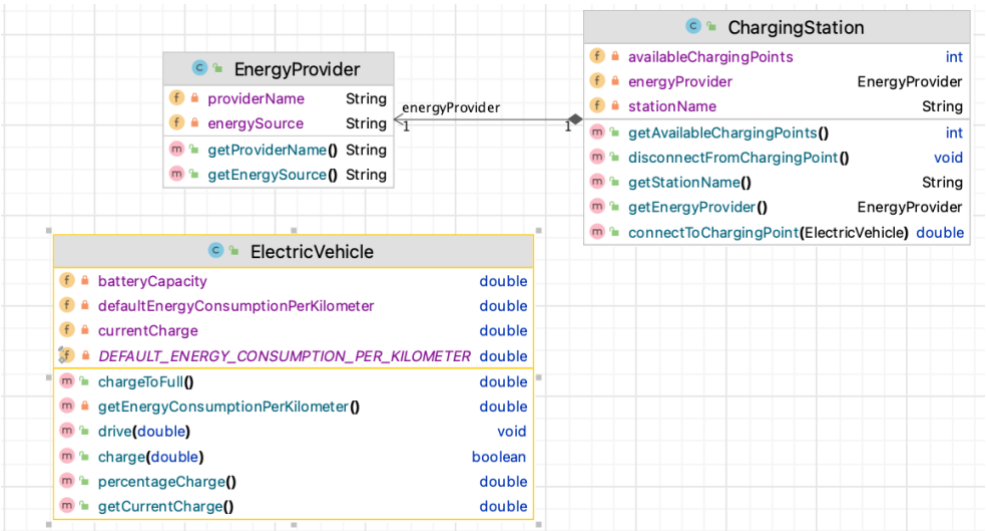
Q.35. Créez un modèle en vous positionnant sur le projet

a. puis clic droit
b. puis Diagrams

c. puis Show Diagram



Q.36. Vous obtenez en jouant avec les visualisations possibles en haut à gauche de la visualisation du diagramme, un diagramme qui ressemble à ce qui suit.
Une version complète du diagramme est présentée à la fin du TD.



VIII. Générer la javadoc avec IntelliJ

Vous devez commenter vos classes pour faciliter leur réutilisation, y compris par vous-même. Pour plus d'information sur la javadoc.

Q.37. Ajoutez un commentaire à votre classe
a. Positionnez-vous sur l'entête de la classe et

b. Commencer à taper /** et return, IJ génère pour vous l'entête de la classe ou bien de la méthode. Il ne vous reste plus qu'à le compléter.

Par exemple :

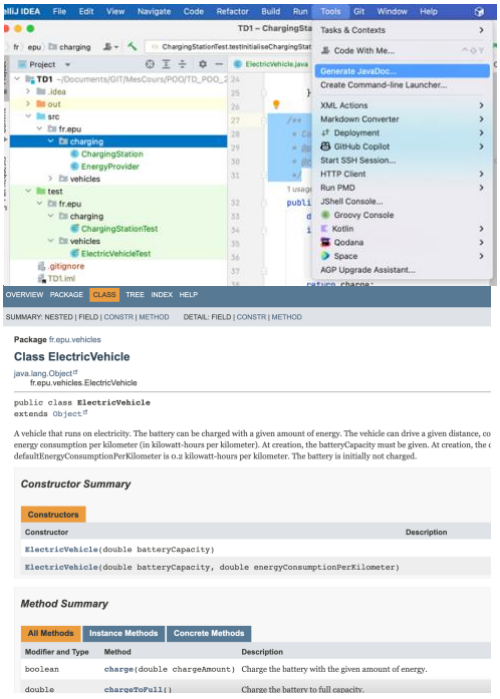
```
/**
 * Connects a vehicle to a charging point and charges it to full.
 * @param vehicle the vehicle to charge
 * @return the amount of energy charged to the vehicle
 */
```

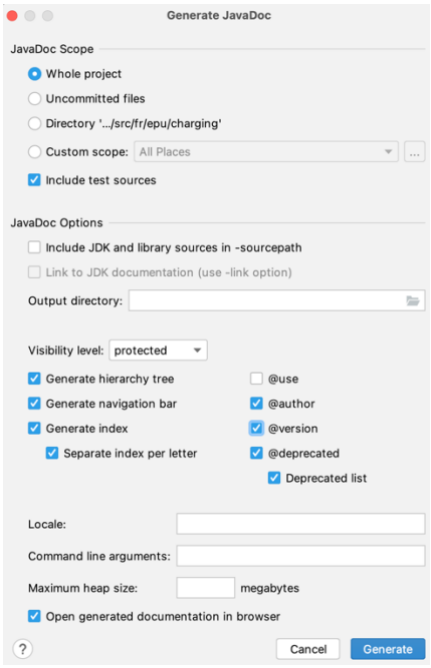
Q.38. Plus d'informations sur la génération des commentaires dans IJ

Q.39. Générer la javadoc et placer les codes html produits sous out/doc (ou build/doc)

a. Sous Tools Generate JavaDoc

b. Vous pouvez alors visualiser votre documentation dans un navigateur





IX. Codes à terminer

Vous devez avoir implémenté à cette étape toutes les fonctionnalités qui vous ont été demandées. Les questions qui suivent sont faciles. Elles vous permettent de vérifiez que vous avez bien compris.

Q.40. Vous complétez votre code en ajoutant une méthode percentageCharge() qui calcule et renvoie le pourcentage de charge actuel par rapport à la capacité de la batterie.

a. Et testez-la.

Q.41. Vous ajoutez une méthode qui calcule la distance maximum que le véhicule peut parcourir avec la charge actuelle.

Q.42. Vous ajoutez une méthode qui calcule la distance maximum que le véhicule peut parcourir en fonction de la capacité de sa batterie

Q.43. Vous complétez à présent la classe *ElectricVehicule* une méthode *drive* spécifiée comme suit puis testez la.

Cette méthode simule la conduite d'un véhicule électrique sur une certaine distance. Elle prend en paramètre la *distance* à parcourir. Elle calcule la quantité d'énergie consommée pour parcourir la distance donnée en multipliant la distance par la consommation d'énergie par kilomètre obtenue à partir de la méthode `getEnergyConsumptionPerKilometer()`. Ce point est important car il nous permettra plus tard de moduler la consommation en fonction du contexte. Elle vérifie si la quantité d'énergie consommée pour parcourir cette distance est inférieure ou égale au niveau de charge actuel de la batterie. Si c'est le cas, elle déduit la quantité d'énergie consommée du niveau de charge actuel et retourne *true* qui signifie qu'elle a bien été en capacité de parcourir cette distance. A l'inverse, lorsque la quantité d'énergie nécessaire est supérieure au niveau de charge actuel, elle retourne *false* qui signifie qu'elle n'est pas en capacité de parcourir cette distance.

Q.44. Vous testez vos codes et vérifiez que la couverture est correcte. Cependant attention si passer par une fonction suffit à la considérer testée, vous savez que ce n'est pas le cas. Cette mesure est donc uniquement une indication.

X. Conclusion

À l'issue de ce TD vous devez maitriser les différents éléments de votre environnement de programmation pour à partir de maintenant vous focalisez davantage sur les concepts et les principes de la POO et algorithmique que sur les outils.