



# Exercices Java

## L'héritage

### - Série 2 -





# Sommaire

## 1) Objectifs :

Découvrir et expérimenter la création de **classes**. Mettre en œuvre les mécanismes de base de la P.O.O.. Assimiler la relation **d'agrégation**. Maîtriser le mécanisme des **exceptions**.

## 2) Estimation du temps de réalisation : 10 heures.

## 3) Vocabulaire utilisé : encapsulation, classe, héritage, classe abstraite, polymorphisme, ...

## 4) Environnement technique : J2SE, JDK, un EDI (type *Eclipse/NetBeans*).

La documentation **Java Oracle** .



## 5) Pré-requis et recommandations :

- a) N'abordez pas les TP dédiés aux **classes** et **héritage** tant que ces notions ne vous sont pas complètement familières.



# TP 1 : *JeuEchecs*

## Objectifs :

Assimiler les notions objet incontournables : la **classe** et l'**héritage**.

## Déroulement :

Vous allez dans ce TP reprendre les premières notions abordées dans le support :  
« **Réutilisez et spécialisez les classes : L'héritage** ».

Pour cela, vous construirez une hiérarchie de classes, implémenterez les concepts d'**héritage**, d'**implémentation** et de **composition**.

Il est demandé de créer un projet **Java SE** nommé *JeuEchecs* avec l'IDE de votre choix, *Eclipse*, *NetBeans*, ...

Vous rangerez les classes métier dans le package *entites* et la classe *Principale* dans le package *application*.

**Remarques** : Revoyez éventuellement comment créer un projet avec *Eclipse* ou *NetBeans* dans le livret de séance d'apprentissage « *S'approprier l'environnement de développement* ».

Au cours de ce TP, vous mettrez en évidence :

- ✓ La notion de **classe usuelle**.
- ✓ La notion de **méthode**.
- ✓ La notion d'**héritage**.
- ✓ La notion de **classe abstraite**.

**Temps alloué** : 5 h.

**Sujet** : Il s'agit de développer la première partie d'une application de jeu d'échecs avec les différentes pièces d'échecs à créer et les méthodes de déplacement et de prise ...



## Jeu d'échecs

8y	T	C	F	D	R	F	C	T
7y	P	P	P	P	P	P	P	P
6y								
5y								
4y								
3y								
2y	P	P	P	P	P	P	P	P
1y	T	C	F	D	R	F	C	T
	1x	2x	3x	4x	5x	6x	7x	8x

Les pièces blanches sont en bas.

Les pièces noires sont en haut.

La dame se situe sur sa couleur.

Les cases paires sont noires.

## Etapes chronologiques à réaliser

### 1. Classe PieceEchecs

Définissez une classe *PieceEchecs* avec :

- 2 données membres *int* privées définissant les coordonnées (de 1 à 8) de la case sur laquelle la pièce se trouve.
- 1 donnée membre *int* privée définissant la couleur de la pièce avec la convention : 1 = blanche; 2 = noire
- 1 constructeur permettant l'initialisation des données membres.
- 1 méthode *getCouleur()* qui retourne un *int* valant 1 ou 2 suivant que la pièce est blanche ou noire.
- 1 méthode *getCouleurCase()* qui retourne un *int* valant 1 ou 2 suivant que la case sur laquelle se trouve la pièce est blanche ou noire.

Précision : Si les coordonnées ou la couleur sont incorrectes, forcez les valeurs au plus près pour créer une pièce valide.

Testez avec une application Java.

### 2. Classes Cavalier et Fou

Définissez 2 classes *Cavalier* et *Fou* héritant de *PieceEchecs*.

Chacune de ces classes implémentera :

- Un constructeur faisant simplement appel au constructeur de la superclasse par le mot-clé *super*.
- Une méthode *peutAllerA(int xD, int yD)* qui renvoie une valeur booléenne indiquant si la pièce en question peut aller à (xD, yD), compte tenu de sa position actuelle.

Testez avec une application java.

Aller plus loin :

Créez des setters dans la classe *PieceEchecs* pour permettre la modification des données membres. (Les setters doivent garantir les contraintes de valeurs pour les coordonnées et la couleur.)

Créez la méthode *estDansLEchiquier(int,int)* qui retourne vrai si la position passée en paramètre fait partie de l'échiquier.

### 3. Tableau de PieceEchecs (facultatif)

Depuis une application Java, créez un tableau de pièces d'échecs. Chaque pièce sera soit un *Fou* soit un *Cavalier*.

Après l'initialisation du tableau, l'application déterminera pour chaque pièce si elle peut ou non aller à la case (5,5).

Remarque : Telles que les classes *Cavalier* et *Fou* ont été définies, chacune dispose d'une méthode *peutAllerA()*, ce qui n'est pas le cas de la classe *PieceEchecs*.

Dans votre application, pour appliquer la méthode *peutAllerA()* à un élément du tableau, vous devrez donc d'abord savoir s'il s'agit d'un cavalier ou d'un fou. Vous pourrez pour cela utiliser la méthode *getClass()* de la classe *Object* et la méthode *getName()* de la classe *Class*.

Par exemple :

```
if ((p[i].getClass().getName()).compareTo("Cavalier")==0) ...
```

Ensuite vous pourrez convertir l'objet p[i] en un objet de la classe *Cavalier* ou *Fou*.

Par exemple :

```
Cavalier c = (Cavalier) p[i];
```

Ensuite, rien ne vous empêchera plus d'appliquer la méthode *peutAllerA()* à l'objet c dont la classe est cette fois, clairement définie.

Testez et essayez d'imaginer quel serait votre programme si votre tableau de *PieceEchecs* pouvait contenir indifféremment des pièces d'un des six types existants (*Roi*, *Dame*, *Fou*, *Cavalier*, *Tour*, *Pion*).

### 4. Méthode abstraite peutAllerA()

Le but de l'exercice est de réaliser la même fonctionnalité que l'exercice précédent, mais en utilisant le mécanisme de la signature dynamique (le polymorphisme).

Vous définirez donc une méthode abstraite *peutAllerA()* dans la classe *PieceEchecs*.

Cela vous permettra d'alléger considérablement le code de votre application

## 5. Classe Roi

Créez une classe *Roi* dérivant aussi de *PieceEchecs*, avec les mêmes membres que *Fou* et *Cavalier*.

Testez avec une application Java

## 6. Classe Pion

Créez une classe *Pion* dérivant aussi de *PieceEchecs*, avec les mêmes membres.

Testez avec une application Java

## 7. Méthode peutManger(PieceEchecs pe)

Maintenant, on veut aussi pouvoir dire si une pièce donnée peut « manger » une autre pièce. Ecrivez des méthodes *peutManger(PieceEchecs pe)* qui renvoient une valeur booléenne indiquant si la pièce en question peut manger la pièce *pe*, compte tenu des positions respectives des pièces et de leurs couleurs.

Testez avec une application Java



## TP 2 : *LocAuto*

### Objectifs :

Assimiler les notions objet incontournables : la **classe** et l'**héritage**.

### Déroulement :

Vous allez dans ce TP reprendre l'ensemble des notions abordées dans le support :  
« **Réutilisez et spécialisez les classes : L'héritage** ».

Pour cela, vous construirez une hiérarchie de classes, implémenterez les concepts d'**héritage**, d'**implémentation** et de **composition**.

Il est demandé de créer un projet **Java SE** nommé *Locauto* avec l'IDE de votre choix, *Eclipse*, *NetBeans*, ...

Vous rangerez les classes métier dans le package *entites* et la classe *Principale* dans le package *application*.

**Remarques** : Revoyez éventuellement comment créer un projet avec *Eclipse* ou *NetBeans* dans le livret de séance d'apprentissage « *S'approprier l'environnement de développement* ».

Au cours de ce TP, vous mettrez en évidence :

- ✓ La notion de **classe usuelle**.
- ✓ La notion de **méthode**.
- ✓ La notion **d'héritage**.
- ✓ La notion de **classe abstraite**.

**Temps alloué** : 5 h.

**Sujet** : Il s'agit de simuler un parc d'automobiles avec des **véhicules motorisés** de type *Voiture* et *Scooter*, chacun de ces véhicules sera doté d'un moteur. Il faudra démarrer les véhicules , les faire rouler, faire le plein , gérer les pannes d'essence...



## Etapes chronologiques à réaliser

8. Construisez une classe **abstraite** *Vehicule* possédant deux variables d'instance : *marque* et *modele* (du véhicule) de type **String**, initialisées par le constructeur.

Cette classe **abstraite** doit contenir trois méthodes abstraites : *demarrer()*, *arreter()* et *faireLePlein()*. La méthode *demarrer()* n'admet aucun paramètre et retourne une valeur booléenne pour indiquer si l'opération a réussi ou non. La méthode *arreter* n'admet aucun paramètre et ne renvoie rien. La méthode *faireLePlein()* admet un paramètre de type *float* (le volume en litre de carburant) et ne renvoie rien.



9. Créez une classe *Moteur* comportant trois variables d'instance : *volume\_reservoir* ( de type *float* ) représentant le nombre actuel de litres dans le réservoir, *volume\_total* ( de type *float* ) représentant le nombre total de litres que le moteur a reçu au fil des pleins effectués et le booléen *démarré* qui précise si le moteur tourne ou non.
10. Ajoutez les accesseurs en lecture pour *volume\_reservoir*, *volume\_total* et *démarré* .
11. Ajoutez pour cette classe *Moteur* les méthodes d'instance suivantes : *demarrer()*, *utiliser()*, *faireLePlein()* et *arreter()*. Dans un premier temps, le traitement de chaque méthode se résumera à afficher, dans la console, l'action concernée ( « Je démarre » , « le moteur utilise .... » , .... ) avec le niveau de carburant restant et/ou la consommation de carburant nécessaire (*utiliser,demarrer*) .
12. Les méthodes *demarrer()* et *utiliser()* de *Moteur* impliquent inévitablement une consommation de carburant. La méthode *demarrer()*, s'il reste de l'essence pour effectuer l'action, réduit le volume de carburant disponible d' 1/10 de litre et retourne un booléen indiquant si l'opération a abouti ou pas.

La méthode *utiliser()* reçoit en paramètre le volume de carburant nécessaire pour le trajet lié à l'utilisation du moteur et **retourne le niveau de carburant** après consommation. Notez que la consommation minimum pour un trajet est soit le nombre de litres nécessaire pour le trajet ( reçu en paramètre), soit le volume restant dans le réservoir si celui-ci est inférieur au volume nécessaire pour effectuer le trajet.

Exemple 1 : la méthode *utiliser()* reçoit 50 litres pour effectuer le trajet correspondant. Il reste 63 litres dans le réservoir : la consommation effective sera de 50 litres ( il restait suffisamment de carburant ) . Il reste 13 litres - 1/10 de litre pour démarrer dans le réservoir après le voyage.

Exemple 2 : la méthode *utiliser()* reçoit 37 litres pour effectuer le trajet correspondant. Il ne reste que 24 litres dans le réservoir : la consommation de carburant sera la totalité du volume restant dans le réservoir, soit 24 litres puisque le trajet en exige 37. Il s'en suivra inévitablement une panne d'essence.

13. Comme il doit être possible d'effectuer le plein de carburant, ajoutez une méthode *faireLePlein()*, avec en argument la quantité de carburant ajoutée. Mettez à jour les variables d'instance *volume\_reservoir* et *volume\_total*. Affichez l'action effectuée comme, par exemple :

```
System.out.println("Plein effectué avec " + carburant + " litres");
```

14. Créez maintenant une sous-classe de **Vehicule**, **abstraite** elle aussi, et nommée **VehiculeAMoteur**. Cette sous-classe a une propriété de type **Moteur** (qu'il faudra instancier ...). Implémentez dans cette classe les méthodes *demarrer()* et *arreter()* grâce à l'attribut *moteur* : *demarrer* et *arreter* de **VehiculeAMoteur** délèguent au moteur chaque opération respective :

```
public boolean demarrer() {  
    return moteur.demarrer();  
}  
...  
public void arreter() {  
    moteur.arreter();  
}
```

... et ajoutez la méthode *faireLePlein()* avec en argument la quantité de carburant ajoutée. Cette méthode *faireLePlein()* dans **VehiculeAMoteur** respectera le cycle suivant : arrêt du moteur, faire le plein (du moteur), démarrer le moteur. Chacune de ces 3 étapes correspond à l'appel de la méthode associée du moteur.

15. Créez deux sous-classes, **concrètes** cette fois, de **VehiculeAMoteur** : **Voiture** et **Scooter**. Ajoutez, dans chaque classe, une méthode *rouler()*. Cette méthode prend en argument (de type *float*) la consommation de carburant nécessaire pour un trajet donné. Pour rouler, il faudra démarrer le moteur, s'il ne l'est pas déjà. Cette méthode *rouler()* va déléguer au moteur cette simulation en appelant sa méthode *utiliser()*. Voici à quoi pourrait ressembler cette méthode *rouler()* (incomplète ci-dessous) :

```
public void rouler(float consommation) throws ..... {  
    if ( !getMoteur().isDemarré() ) {  
        getMoteur().demarrer();  
    }  
    float carburant = moteur.utiliser(consommation);  
    .....  
}
```

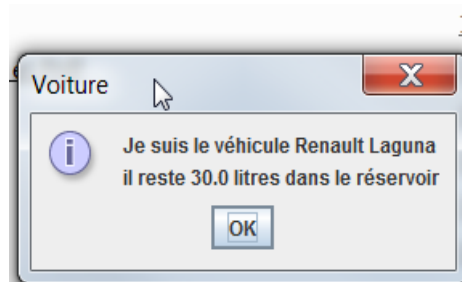
16. Implémentez dans les classes les méthodes de description que vous jugerez nécessaires : *toString()* .... et qui seront utilisées pour produire les affichages relatifs aux jeux d'essais fournis.

17. Dans une classe *Principale*, implémentez la méthode *main()*. Instanciez une Renault Laguna avec 30 litres dans le réservoir. Démarrez la *Laguna*, effectuez un trajet correspondant à une consommation de 25 litres. Affichez les caractéristiques de la Laguna, avant et après avoir effectué ce trajet.
18. Ajoutez toutes les méthodes et fonctions d'affichage nécessaires pour produire les copies d'écran suivantes :

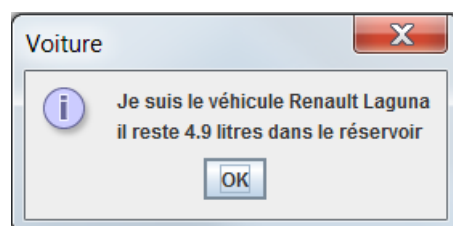
```
Voiture laguna = new Voiture( "Renault", "Laguna",30.0f);  
System.out.println(laguna);  
laguna.demarrer() ;  
laguna.rouler(25);  
System.out.println(laguna);
```

19. Afin de produire des affichages plus séduisants, utilisez les boîtes de dialogues fournies en standard dans le package *javax.swing*.

```
JOptionPane.showMessageDialog(null,laguna,"Voiture",JOptionPane.INFORMATION_MESSAGE);
```



Avant le trajet de 25 litres.



Après avoir démarré et effectué le trajet de 25 litres.

20. Dans l'exemple ci-dessus, vous remarquerez que le volume de carburant initial est **supérieur** à celui nécessaire pour le trajet. Mais que se passerait-il si le besoin en carburant dépasse le nombre de litres disponibles dans le réservoir ? C'est la panne d'essence !! Vous allez gérer cette situation en implémentant la notion très importante d'**exception**.

Dans la méthode `rouler()` de **Voiture** indiquez, grâce à la clause **throws**, qu'elle peut lever une exception de type **PanneEssenceException**. Vous allez donc créer cette classe dans le package `com.votreprenom.exceptions`. Elle héritera de la classe **Java Exception**.

Son **constructeur** se contentera d'appeler celui d'**Exception** avec la chaîne de caractères correspondant au message de l'erreur. Voilà, tout simplement :

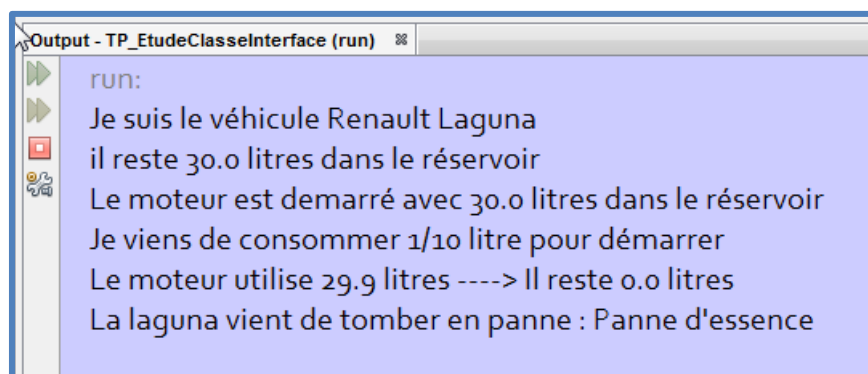
```
public PanneEssenceException(String message) {  
    super(message);  
}
```

Maintenant que la méthode `rouler()` indique qu'elle est susceptible d'émettre une exception :

```
public void rouler(float consommation) throws PanneEssenceException {  
    ....  
    if (carburant == 0)  
        // Levée de l'exception ...  
}
```

... essayez de rouler plus que le volume de carburant dans le réservoir ne vous le permet :

```
public static void main(String[] args) {  
    Voiture laguna = new Voiture("Renault", "Laguna", 30.0f);  
    System.out.println(laguna);  
    try {  
        laguna.rouler(35);  
    } catch (PanneEssenceException ex) {  
        System.out.println("La laguna vient de tomber en panne : " + ex.getMessage());  
    }  
    System.out.println(laguna);  
}
```



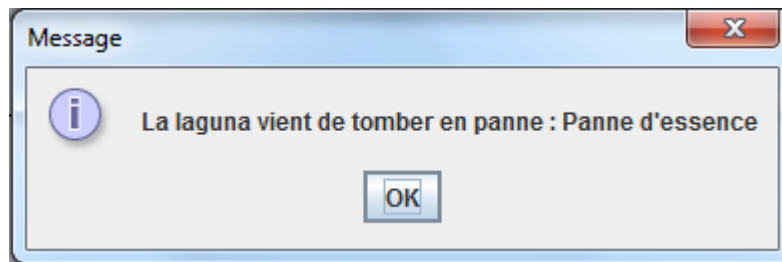
```
Output - TP_EtudeClasseInterface (run)
run:
Je suis le véhicule Renault Laguna
il reste 30.0 litres dans le réservoir
Le moteur est démarré avec 30.0 litres dans le réservoir
Je viens de consommer 1/10 litre pour démarrer
Le moteur utilise 29.9 litres ----> Il reste 0.0 litres
La laguna vient de tomber en panne : Panne d'essence
```

Remplacez

```
System.out.println(" .. ");
```

par :

```
JOptionPane.showMessageDialog(null,"La laguna vient de tomber en panne : " +  
ex.getMessage());
```



21. L'étape suivante va consister à **remédier à la panne d'essence** précédente en proposant de remplir le réservoir avec une valeur arbitraire de 50 litres puis avec un nombre de litres saisi par l'utilisateur grâce à une boîte de dialogue de type « Entrée de donnée » vue précédemment .

```
String resultat = JOptionPane.showInputDialog(null,"Veuillez saisir le nombre de litres SVP");
```

Vous transformerez la variable *resultat* de type *String* en un *Integer* :

```
Integer i = new Integer (resultat);
```

22. Mettez en œuvre un jeu de tests consistant à instancier une **Citroën C5** avec 40 litres. Puis, effectuez, au sein d'une boucle, 6 trajets de 10 litres, déclenchant donc une panne d'essence.
23. Gérer l'exception en produisant un message adéquat et en remplissant le réservoir avec 50 litres. N'oubliez pas de redémarrer la voiture et donc le moteur après avoir fait le plein. Continuez le trajet jusqu'à son terme.
24. Rajoutez la fonctionnalité suivante : la gestion du volume total de litres de carburant consommés au cours d'un trajet, notamment si, à l'issue d'une panne d'essence, il a fallu remplir le réservoir.
25. A la fin du trajet, affichez le nombre total de litres restant et le nombre total de litres versés dans le réservoir. Soit :

```
Output - TP_EtudeClasseInterface (run) #2  Notifications  Analyzer  Terminal  Action

run:
Le moteur est démarré avec 40.0 litres dans le réservoir
Je viens de consommer 1/10 litre pour démarrer
Je suis le véhicule Citroën C5
il reste 39.9 litres dans le réservoir vient de démarrer
Le moteur utilise 10.0 litres ----> Il reste 29.9 litre(s)
Je viens de consommer 10 litres
Le moteur utilise 10.0 litres ----> Il reste 19.9 litre(s)
Je viens de consommer 20 litres
Le moteur utilise 10.0 litres ----> Il reste 9.9 litre(s)
Je viens de consommer 30 litres
Le moteur utilise 9.900002 litres ----> Il reste 0.0 litre(s)
Panne d'essence

-----

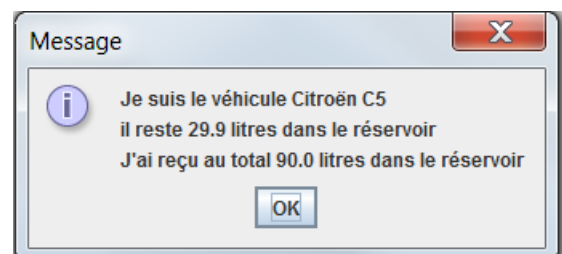
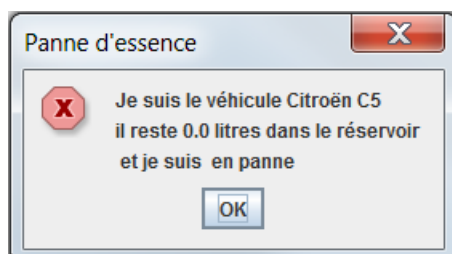
Je suis le véhicule Citroën C5
il reste 0.0 litres dans le réservoir
et je viens de tomber en panne

-----

Le moteur est arrêté
Plein effectué avec 50.0 litres
Le moteur est démarré avec 50.0 litres dans le réservoir
Je viens de consommer 1/10 litre pour démarrer
Le moteur utilise 10.0 litres ----> Il reste 39.9 litre(s)
Je viens de consommer 11 litres
Le moteur utilise 10.0 litres ----> Il reste 29.9 litre(s)
Je viens de consommer 21 litres
Le moteur est arrêté

-----
-----

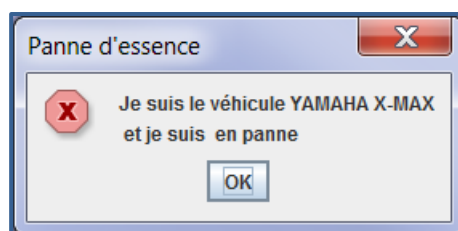
BUILD SUCCESSFUL (total time: 6 seconds)
```





1. Créez maintenant une autre classe concrète : *Scooter* , selon les mêmes principes que la classe *Voiture* .
2. Intanciez un scooter **Yamaha** de modèle **X-MAX** avec 20 litres . Effectuez une petite promenade correspondant à un trajet de 3 fois 10 litres . Traitez l'exception, remettez 15 litres et continuez votre périple.

```
Output - TP_EtudeClasseInterface (run) #2  Notifications  Analyzer  Terminal  Action
run:
Le moteur est démarré avec 20.0 litres dans le réservoir
Je viens de consommer 1/10 litre pour démarrer
X-MAX vient de démarrer
Le moteur utilise 10.0 litres ----> Il reste 9.9 litre(s)
Le moteur utilise 9.9 litres ----> Il reste 0.0 litre(s)
X-MAX est en Panne
Le moteur est arrêté
Plein effectué avec 15.0 litres
Le moteur est démarré avec 15.0 litres dans le réservoir
Je viens de consommer 1/10 litre pour démarrer
Le moteur utilise 10.0 litres ----> Il reste 4.9 litre(s)
Le moteur est arrêté
BUILD SUCCESSFUL (total time: 1 second)
```



## Poursuite du TP :

Une société de location de véhicules propose à ses clients en ligne des **voitures** et **scooters** .

1. Le mécanicien du parc de véhicules de ce loueur contrôle régulièrement le parc et fait tourner les moteurs de ces véhicules et effectue le plein, si nécessaire.
2. Simuler cette activité en construisant une classe *ParcVehicules* composé d'un ensemble de véhicules. Ce parc de véhicules sera, dans un premier temps, implémenté sous forme d'un tableau. Puis, par la suite, existera en tant que collection.
3. Dimensionnez le tableau grâce à une constante *final*. Déterminez le type de chacun des éléments de ce tableau. Mettez en œuvre les accesseurs.



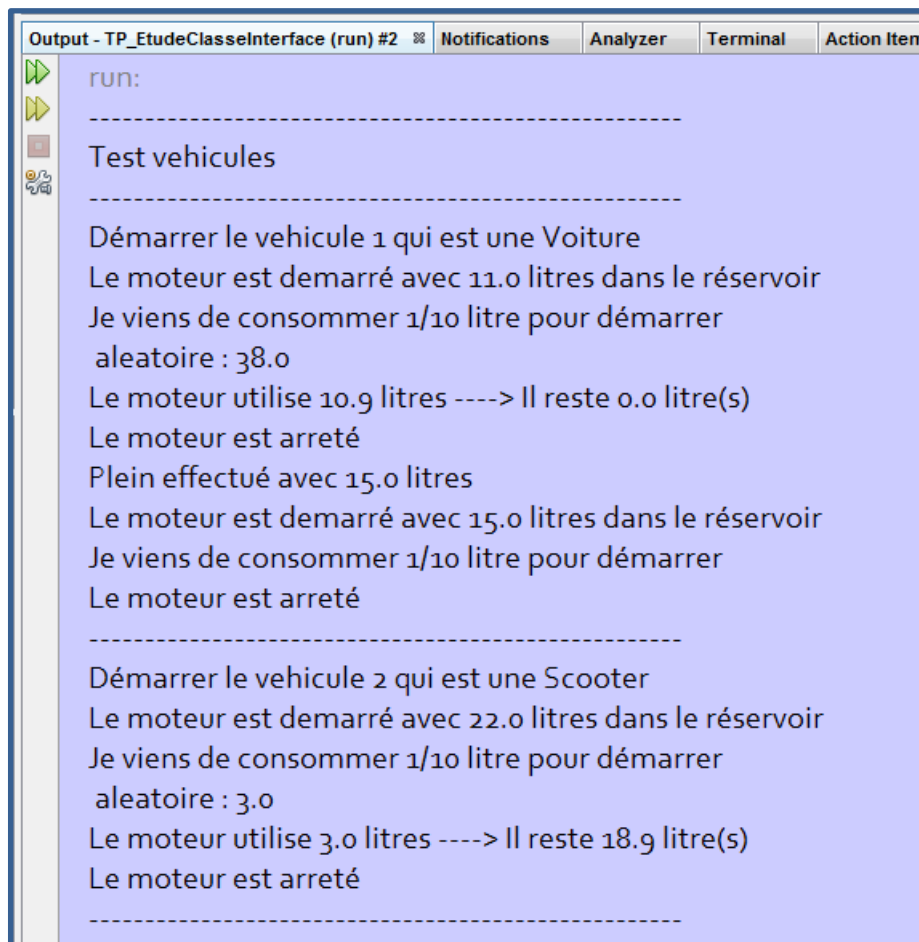
Implémenter une méthode d'instance *contrôlerVehicules()* réalisant les activités chronologiques suivantes :

Pour chaque véhicule, quel qu'il soit :

- le démarrer.
- faire tourner son moteur pour un trajet aléatoire compris entre 1 et 5 kilomètres.
- En cas de panne d'essence, ajouter du carburant pour un volume compris entre 1 et 10 litres.
- Afficher son type dynamiquement (*Voiture* ou *Scooter*).

Le constructeur de *ParcVehicules* va ainsi recevoir un tableau de *Véhicule*. Les véhicules insérés dans ce tableau seront donc instanciés avant le parc qui les contient.

4. Instanciez 4 voitures et 3 scooters , rangez-les dans le parc et testez ces véhicules. Les affichages produits devraient ressembler à l'extraction suivante :



```
run:
-----
Test vehicules
-----
Démarrer le vehicule 1 qui est une Voiture
Le moteur est démarré avec 11.0 litres dans le réservoir
Je viens de consommer 1/10 litre pour démarrer
aleatoire : 38.0
Le moteur utilise 10.9 litres ----> Il reste 0.0 litre(s)
Le moteur est arrêté
Plein effectué avec 15.0 litres
Le moteur est démarré avec 15.0 litres dans le réservoir
Je viens de consommer 1/10 litre pour démarrer
Le moteur est arrêté
-----
Démarrer le vehicule 2 qui est une Scooter
Le moteur est démarré avec 22.0 litres dans le réservoir
Je viens de consommer 1/10 litre pour démarrer
aleatoire : 3.0
Le moteur utilise 3.0 litres ----> Il reste 18.9 litre(s)
Le moteur est arrêté
-----
```

# Copyright

➤ **Chef de projet (responsable du produit de formation)**

PERRACHON Chantal, DIIP Neuilly-sur-Marne

➤ **Ont participé à la conception**

COULARD Michel, CFPA Evry Ris-Orangis

➤ **Réalisation technique**

COULARD Michel, CFPA Evry Ris-Orangis

➤ **Crédit photographique/illustration**

Sans objet

➤ **Reproduction interdite / Edition 2014**

AFPA Février 2014

**Association nationale pour la Formation Professionnelle des Adultes**

13 place du Général de Gaulle – 93108 Montreuil Cedex

[www.afpa.fr](http://www.afpa.fr)