



Exercices Java

L'héritage

- Série 2 -





Sommaire

1) Objectifs :

Découvrir et expérimenter la création de **classes** et **d'interfaces**. 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 : 12 heures.

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

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

La documentation **Java Oracle** .



5) Pré-requis et recommandations :

a) Réalisez **préalablement** le **Q.C.M.** du document **QCM-Exercices-Java-Série-2** avec rigueur et réflexion avant de réaliser les T.P.s. Donnez un sens à chaque réponse que vous jugerez bonne.

b) N'abordez pas les TP's dédiés aux **classes** et **interfaces** tant que ces notions ne vous sont pas complètement familières.



TP 1.1 : *TP_EtudeClasseInterface*

Objectifs :

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

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 et d'interfaces, implémenterez les concepts d'**héritage**, d'**implémentation** et de **composition**.

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

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

Remarques : Revoyez éventuellement comment créer un projet avec *NetBeans* – ou *Eclipse* – 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**.
- ✓ La notion **d'interface**.

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

1. 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.



2. 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.
3. Ajoutez les accesseurs en lecture pour *volume_reservoir*, *volume_total* et *démarré* .
4. 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*) .
5. 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.

6. 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");
```

7. 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.

8. 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 );  
    .....  
}
```

9. 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.

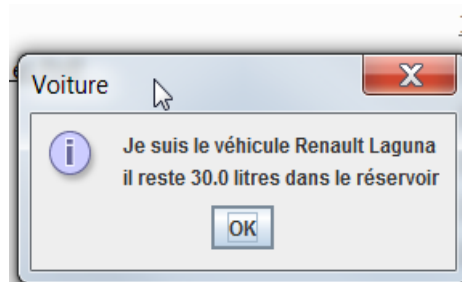
10. Dans une classe *Principale*, implémentez la méthode *main*. Instanciez une Renault Laguna avec 30 litres dans le réservoir. Démarrez la *Laguna*, effectuez un trajet correspondant à une consommation de 25 litres. Affichez les caractéristiques de la *Laguna*, avant et après avoir effectué ce trajet.
11. 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);
```

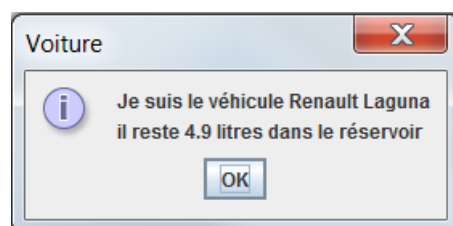
```
run:  
Je suis le véhicule Renault Laguna  
il reste 30.0 litres dans le réservoir  
Le moteur est démarré avec 30.0 litres dans le réservoir  
Je viens de consommer 1/10 litre pour démarrer  
Le moteur utilise 25.0 litres ----> Il reste 4.9 litres  
Je suis le véhicule Renault Laguna  
il reste 4.9 litres dans le réservoir  
BUILD SUCCESSFUL (total time: 5 seconds)
```

12. 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.

13. 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.votreprénom.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 :

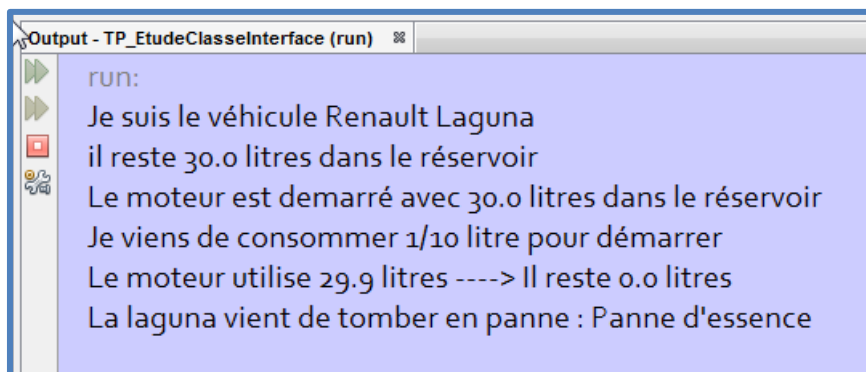
```
public PanneEssenceException( String message ) {  
    super(message);  
}
```

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) x  
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());
```



14. 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 ) ;
```

15. 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.
16. 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.
17. 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.
18. 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

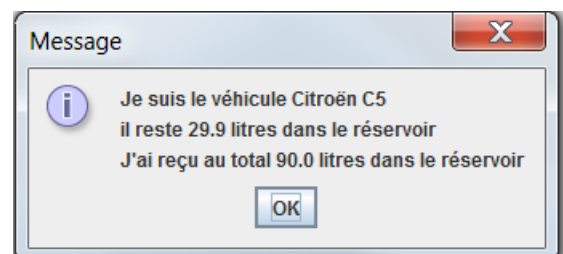
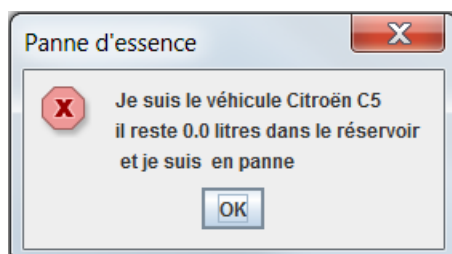
-----

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

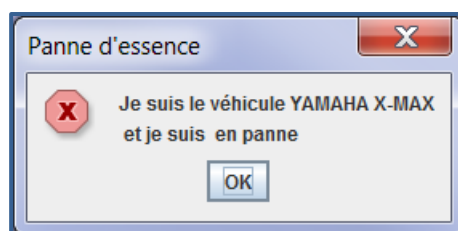
-----
BUILD SUCCESSFUL (total time: 6 seconds)
```





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.

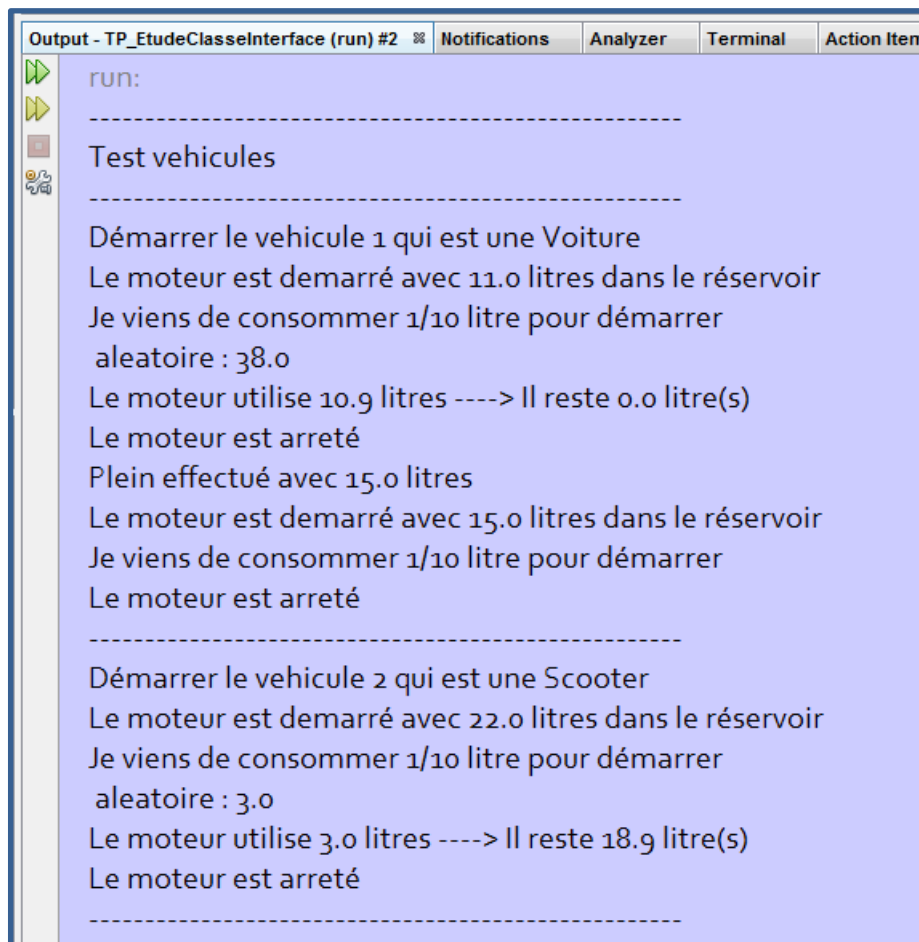
4. Implémenter une méthode d'instance *contrôlerVehicules* réalisant les activités chronologiques suivantes :

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.

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



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

TP 1.2 : ForumNouvelles

Objectifs :

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

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 et d'interfaces, implémenterez les concepts d'**héritage**, d'**implémentation** et de **composition**.

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

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

Remarques : Revoyez éventuellement comment créer un projet avec *NetBeans* – ou *Eclipse* – 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**.
- ✓ La notion d'**interface**.

Temps alloué : 7h.

Sujet :

On veut gérer un forum de simpliste de nouvelles, disponible sur un réseau. Ce forum a un nom et une date de création. Il gère des **nouvelles** et des **abonnés**. Un modérateur veille au bon respect d'une charte d'éthique. Si besoin, ce **modérateur** pourra intervenir sur le forum en :

- ✓ **Supprimant** une nouvelle.
- ✓ **Bannissant** un abonné.
- ✓ **Ajoutant** un abonné qui en fait la demande.
- ✓ **Listant** tous les abonnés.
- ✓ **Listant** toutes les nouvelles.



Un **abonné** doit avoir les prérogatives suivantes :

- ✓ **Ajouter** une nouvelle.
- ✓ **Consulter** une nouvelle.
- ✓ **Répondre** à une nouvelle.

- Un **abonné** a un prénom (*String*), un nom(*String*), un âge(*int*). Idem pour le **modérateur**.
- Un **nouvelle** est caractérisée un sujet et un texte descriptif. Tous deux de type *String*.

Il vous est demandé :

- d'écrire un projet **Java** SE nommé *ForumNouvelles* pour implémenter tous ces besoins.
- d'élaborer un scénario de tests visant à vérifier le bon fonctionnement des fonctionnalités liées au forum.

Pour cela, réalisez les étapes chronologiques suivantes :

1. Créez la classe *Abonne* avec les variables d'instance *prenom*, *nom* et *age* évoqués ci-dessus.
2. Créez la classe *Moderateur* avec les variables d'instance *prenom*, *nom* et *age* évoqués ci-dessus.
3. Créez la classe *Nouvelle* avec les variables d'instance *sujet* et *texte* (de type *String*) et *dateCreation* (de type *Date*).
4. Créez la classe *Forum* avec les variables d'instance *dateCreation* (de type *Date*) et *nom* (de type *String*).
5. Pour ces classes, créez les constructeurs respectifs nécessaires pour valoriser les variables d'instance décrites sauf les v.i. *dateCreation* qui seront automatiquement valorisées, au moment de l'instanciation.

On veut maintenant **collectionner** les abonnés et les nouvelles qu'ils créent. L'endroit le plus adapté pour gérer ces deux ensembles est bien sûr le forum .

6. Ajoutez pour cela deux variables d'instance au sein de la classe *Forum* qui sont d'abord des tableaux puis des collections sous formes d'*ArrayList* de *Nouvelle* pour l'une et d'*ArrayList* d'*Abonne* pour l'autre. Vous utiliserez pour cela la **généricité des collections**.

Se pose maintenant le problème des **activités réalisables** par les deux catégories d'utilisateurs : les abonnés et les modérateurs.

Comment réunir ces **privileges** à répartir selon ces deux types d'utilisateur ?

Vous allez créer **deux interfaces** : une qui va déclarer l'ensemble des actions autorisées pour un **abonné** - *IAbonneForum* - et l'autre qui va déclarer l'ensemble des actions autorisées pour un **modérateur** - *IModerateurForum* -.

7. Ces deux interfaces seront ensuite communément implémentées par la classe *Forum*. Et il suffira par la suite de ne donner aux deux utilisateurs *Abonne* et *Moderateur* que **la vue restreinte à leurs privileges** pour une même et unique instance de *Forum*.

En clair : **une seule instance de forum** est générée . Et cette instance, unique, sera transmise en tant que variable commune et partagée dans les deux classes d'utilisateur. C'est-à-dire en tant que variable de classe.

Seul , le type de cette variable va différer au sein des classes *Abonne* et *Moderateur*.



- Pour le premier, on offrira les services du forum uniquement liés aux abonnés. Le type du forum au sein de la classe *Abonne* sera donc de type *IForumAbonne*.
- Pour le second, on offrira les services du forum uniquement liés aux modérateurs. Le type du forum au sein de la classe *Moderateur* sera donc de type *IForumModerateur*.

Il suffira ensuite de coder au sein des classes *Abonne* et *Moderateur* les méthodes liées aux activités effectuées par l'utilisateur spécifique .

Chacune de ces méthodes va déléguer au forum, dont on ne connaît que **la référence sur l'interface adaptée**, les fonctions d'ajout, de suppression, de consultation, de réponse, d'affichage , ... d'abonnés et de nouvelles. **Tout ceci selon le type de l'utilisateur** .

8. Créez l'interface *IForumAbonne* . Déclarez les méthodes suivantes, relatives aux activités d'un abonné :

```
boolean ajouterNouvelle ( Nouvelle n );  
void consulterNouvelle ( int i );  
void repondreNouvelle ( int i );
```

9. Créez l'interface *IForumModérateur* . Déclarez les méthodes suivantes, relatives aux activités d'un modérateur :

```
boolean supprimerNouvelle ( Nouvelle n );  
void bannirUnAbonne ( Abonne a );  
int ajouterAbonne ( Abonne a );  
void listerAbonnes();
```

10. Créer maintenant la classe *Forum* en y implémentant simultanément **les deux interfaces** présentées ci-dessus.

11. Prévoyez au sein de *Forum* les variables d'instance suivantes :

- a. *dateCreation* (*Date*) (valorisée dans le constructeur)
- b. le nom du forum (*String*).
- c. La collection de messages émis sur le forum (d'abord sous forme de tableau , puis sous forme d'*ArrayList* lorsque vous en aurez les compétences)
- d. La collection d'abonnés inscrits dans le forum par un modérateur (d'abord sous forme de tableau , puis sous forme d'*ArrayList*)

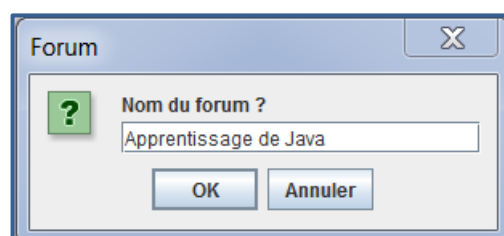
12. Implémentez le **constructeur** qui ne prendra que le nom du forum comme unique argument.

13. Mettez en œuvre tous les **accesseurs** nécessaires.

14. Codez dans *Forum* l'ensemble de toutes les méthodes des deux interfaces *IForumAbonne* et *IForumModérateur* que la classe *Forum* implémente.

Concernant le parcours des collections d'abonnés et de messages, vous utiliserez la notion d'itérateur sur ces collections (Interface *Iterator*)

15. Commencez l'application en soumettant à l'utilisateur le nom qu'il souhaite donner au forum en utilisant la boîte de dialogue d'entrée d'information. Ainsi :



16. Attribuez ce nom ensuite au constructeur de *Forum*.

17. Une fois le forum nommé et instancié, transmettez-le par le biais des accesseurs en écriture associés et en tant que variable de classe aux deux classes *Abonne* et *Moderateur*.



Rappel : le type du forum diffère au sein de ces deux classes pour n'offrir que le strict nécessaire en terme de services aux deux catégories d'utilisateur. Un forum « est un » *IForumAbonne* mais aussi « est un » *IForumModerateur*. On voit ici l'immense intérêt des interfaces

18. Définissez maintenant au sein de la classe *Abonné* les méthodes suivantes :

- ✓ *void creerNouvelle()* : Faire apparaître les boîtes de dialogue pour saisir sujet et texte de la nouvelle. La déposer ensuite dans le forum.
- ✓ *void deposerNouvelle (Nouvelle nelle)* : l'abonné a la responsabilité de déposer la nouvelle reçue dans le forum.
- ✓ *public void lireNouvelle(int i)* : l'abonné lit la nouvelle de rang *i* dans le tableau ou la collection.
- ✓ *public void repondreNouvelle(Nouvelle nelle)* : l'abonné répond à la nouvelle reçue en générant une autre **du même sujet bien sûr**.

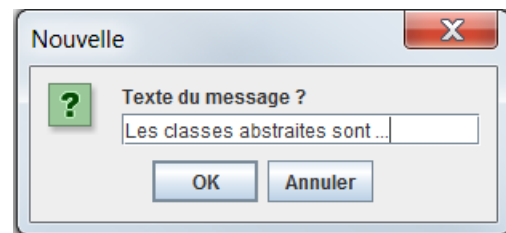
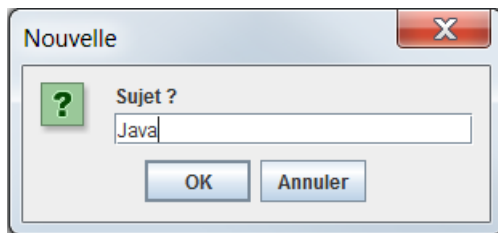
Chacune de ces méthodes de la classe *Abonne* délèguera au forum, dont elle possède une référence, l'activité concernée.

19. Définissez maintenant au sein de la classe *Moderateur* les méthodes suivantes :

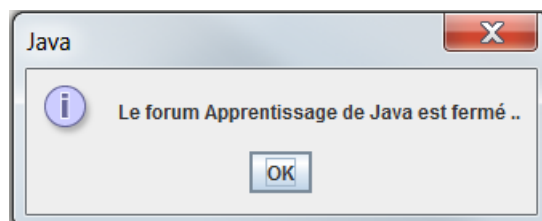
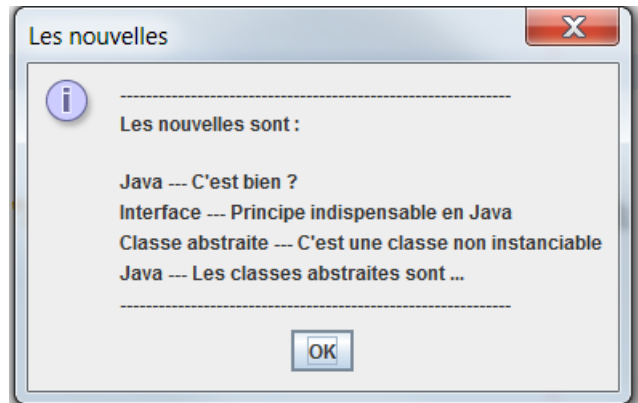
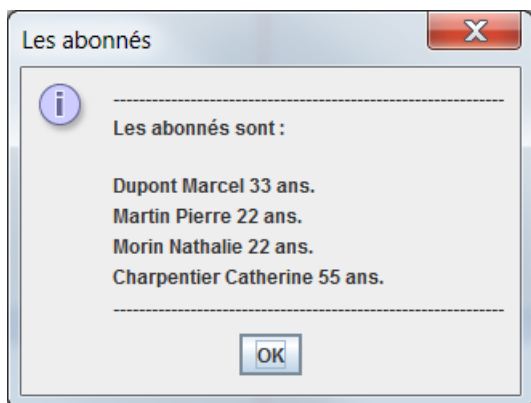
- ✓ *public int ajouterAbonne (Abonne a)* : le modérateur ajoute l'abonné reçu en s'adressant au forum.
- ✓ *public int exclureAbonne(Abonne a)* : le modérateur bannit l'abonné en appelant la méthode du forum adaptée.
- ✓ *public void supprimerNouvelle(Nouvelle n)* : le modérateur délègue au forum cette suppression.
- ✓ *public void afficherLesAbonnes()* : délégation au forum avec *listerAbonnes*.

20. Dans *main*, instanciez des abonnés, un modérateur, appelez la méthode adéquate pour ajouter ces abonnés dans le forum.

21. Instanciez des nouvelles, appelez la méthode adéquate pour les insérer dans le forum via les abonnés.



22. Vérifiez au sein de la classe *Abonne*, que les méthodes réservées à *Moderateur* - nécessitant des privilèges - ne lui sont pas accessibles .
23. Affichez la liste des abonnés, celle des nouvelles et enfin le nom du forum et sa date de création.



Défi



24. Ajoutez la notion d'avertissement au sein de la classe *Abonné* . Ajoutez un compteur d'avertissements . Donnez la possibilité à un modérateur d'avertir n fois un abonné avant de l'exclure définitivement du forum .

25. Rajoutez pour cette fonctionnalité toutes les méthodes que vous jugerez utiles .

26. Avertissez un abonné grâce à un modérateur, puis bannissez cet abonné .



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