TD 4 L'héritage

Exercice I

Vous aves les classes Java suivantes :

```
Animal
Arbre
Humain
Fleur
Loup
Lion
Mammifère
Insecte
Pigeon
Abeille
Arbuste
Plante
```

- Ordonnez ces classes en indiquant quelle(s) classe(s) héritent de quelles classes.

SOLUTION :

Animal
- Mammifere
  - Loup
  - Lion
  - Humain
- Pigeon
- Insecte
  - Abeille

Plante
- Arbre
  - Arbuste
- Fleur

- Et si on rajoute la classe `Organisme` ?

SOLUTION :

La classe Organisme sera la superclasse de toutes les autres classes.

- Les objets dans toutes ces classes peuvent grandir. Par contre, seulement les animaux et les insectes peuvent bouger. Au contraire, seulement les plantes peuvent faire une

photosynthèse pour se nourrir. Dans quelles classes ci-dessus voulez-vous ajouter les méthodes qui représentent ces trois capacités : grandir, bouger, photosynthese ?

SOLUTION :

Le but de l'héritage est de « factoriser » des traits communs à plusieurs sousclasses dans une superclasse. C'est pourquoi les méthodes qui modélisent des traits communs doivent toujours se trouver dans la plus haute superclasse possible. Comme tout objet pourra grandir, nous allons mettre grandir dans la classe Organisme. Selon le même principe nous allons mettre bouger dans la classe Animal et photosynthese dans la classe Plante.

Exercice 2

Nous allons simuler des divers types de véhicules : par exemple des voitures et des camions. Ces véhicules ont quelques caractéristiques communes, mais aussi des caractéristiques spéciales. Ils ont un type de carburant, une capacité du réservoir, un nombre maximal de passagers. Les deux types de véhicules peuvent faire le plein, consommer du carburant et prendre des passagers ; par contre, un camion pourra aussi charger/décharger de la marchandise.

Ceci justifie l'utilisation de l'héritage.

Dans cet exercice, nous allons commencer sur une superclasse `Vehicule`.

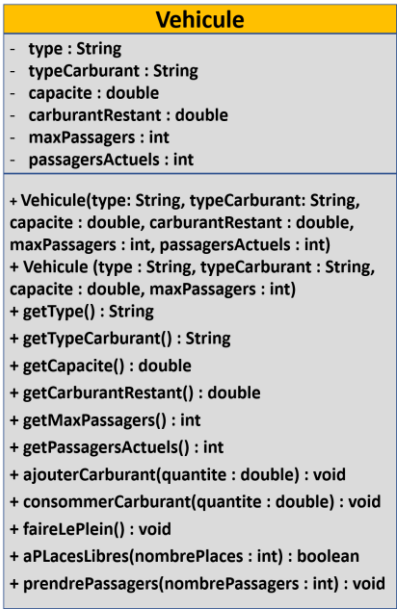
Dans la classe `Vehicule` (voir le diagramme de classe ci-dessous), nous avons les prochains attributs :

- Un attribut `type` de type `String` qui indique si le véhicule est une (”`Voiture`” ou un ”`Camion`”),
- Un attribut de type `String` qui indique type de carburant qu'on utilise
- La capacité du réservoir du véhicule,
- Le volume de carburant dont il dispose maintenant,
- Le nombre maximal de passagers qu'il peut accueillir, y compris le chauffeur,
- Le nombre actuel de passagers.

Les méthodes de cette classe sont :

- Deux constructeurs
- Des getters pour chaque attribut
- Une méthode toString() qui retourne [<nom d'attribut> : <valeur de l'attribut>] séparés par des vergules.
- Des méthodes pour manipuler la quantite de carburant : `void consommerCarburant()`, `void faireLePlein()` et `void ajouterCarburant()`
- Des méthodes pour manipuler le nombre de passagers, notamment `boolean aPlacesLibres()` et `void prendrePassagers(int nombrePassagers).`

Consultez les diagrammes de classe ci-dessous.



- Le premier constructeur mettra les valeurs de chaque attribut de la classe à la valeur correspondante mise en entrée. Pour le deuxième constructeur, le carburant restant sera mis par défaut à la valeur de la capacité du réservoir, tandis que le nombre actuel de passagers sera 1 (le chauffeur). Ecrivez ce constructeur.

SOLUTION :

Pour éviter la déduplication du code, on appelle le premier constructeur dans le deuxième :

```
public Vehicule(String type, String typeCarburant, double capacite, int maxPassagers) {
    this(type, typeCarburant, capacite, capacite, maxPassagers, 1);
}
```

- Les méthodes `ajouterCarburant(double quantite)` et `consommerCarburant(double quantite)` nous permettent de modifier la quantité de carburant actuelle qui existe dans la voiture par la quantité mise en entrée, sans dépasser la capacité et sans tomber sous 0. Ecrivez la méthode `ajouterCarburant`.

SOLUTION :

Cette méthode ne devrait pas vous poser des problèmes. On a plusieurs choix pour la vérification qu'on n'a pas dépassé la capacité maximale de carburant. Celle que j'ai choisi ci-dessous est une technique facile, qui semble bien modéliser ce qui se passe en réalité : si on essaie de mettre trop de carburant dans la voiture, le réservoir se remplit et la quantité en excès tombe à côté de la voiture.

```
	public void ajouterCarburant(double quantite) {
		this.carburantRestant += quantite;
		if (this.carburantRestant > this.capacite) {
			this.carburantRestant = this.capacite;
			System.out.println("STOP ! Vous etes a pleine capacite !");
		}
	}
```

- La méthode `void faireLePlein()` remplira la voiture d'essence. Utilisez la méthode `ajouterCarburant(double quantite)` pour écrire cette méthode.

SOLUTION :

Nous avons plusieurs choix pour écrire la méthode faireLePlein(). So on veut être précis, alors il faudrait utiliser la syntaxe :

```
	public void faireLePlein() {
		this.ajouterCarburant(this.capacite-this.carburantRestant);
	}
```

Mais rappelez-vous qu'on a écrit la méthode `ajouterCarburant` tel que, si on a un excès de carburant, l'excès se perdra sans conséquence pour l'objet en question. Alors on peut s'imaginer une méthode `faireLePlein()` écrite comme ci-dessous :

```
	public void faireLePlein() {
		this.ajouterCarburant(this.capacite);
	}
```

- La méthode `boolean aPlacesLibres()` rend `true` s'il y a encore de la places pour prendre des passagers dans le véhicule, et `false` autrement. La méthode `void prendrePassagers(int nombrePassagers)` vérifie si on a encore de la place pour le nombre indiqué de passagers et, si c'est le cas, on y ajoute le nombre de passagers. Ecrivez la méthode `void prendrePassagers(int nombrePassagers)`.

SOLUTION:

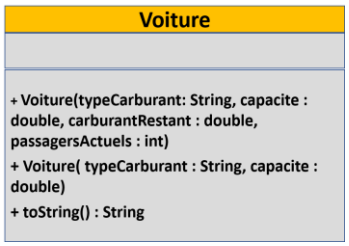
La vérification nécessaire pour vérifier le nombre de places disponibles est déjà programmée dans la méthode `aPlacesLibres`. En utilisant cette méthode on peut écrire :

```
	public void prendrePassagers(int nombrePassagers) {
		if (this.aPlacesLibres(nombrePassagers)) {
			this.passagersActuels += nombrePassagers;
		}
		else {
			System.out.println("Vous n'avez pas assez de place pour prendre "
+nombrePassagers + " passager(s). Il vous en restent " + (this.maxPassagers -
this.passagersActuels + " places disponibles dans le vehicule."));
		}
	}
```

Exercice 3

Nous aurons une classe `Voiture` qui hérite de la classe `Vehicule`. Les objets de la classe `Voiture` auront l'attribut type de la classe `Voiture` mis à "Voiture". Meme si ceci n'est pas toujours le case en réalité nous allons supposer que toutes les voitures ont 5 places (y compris le chauffeur). La classe `Voiture` est une sousclasse de la classe `Vehicule`.

La classe `Voiture` aura le diagramme de classe suivant :



Notamment, cette classe n'a aucun attribut par rapport à sa superclasse. Elle aura, par contre, deux constructeurs et elle modifiera la méthode toString() de sa superclasse.

- Rappel : qu'est-ce qu'une superclasse et qu'est-ce qu'une sousclasse ?

SOLUTION :

Une superclasse est un moule qui modélise des objets de divers types, mais qui ont des caractéristiques en commun. Une sousclasse (d'une superclasse) est un type d'objet de la superclasse. Exemple : superclasse : `Personne` ; sousclasse : `Eleve`.

- Qu'est-ce que la classe `Voiture` hérite de sa superclasse ?

SOLUTION :

La classe Voiture hérite explicitement les attributs et toutes les méthodes de la classe Vehicule à l'exception de la méthode toString(), qu'elle modifie, et à l'exception des constructeurs, qui sont hérités indirectement (utilisés dans les constructeurs de Voiture).

- Les attributs de la classe `Vehicule` sont des valeurs privées. Est-ce qu'on aura accès à ces attributs à partir de la classe `Voiture` ?

SOLUTION :

Comme les attributs sont privés, même si la classe `Voiture` hérite de la classe `Vehicule`, nous ne pourrons jamais faire une référence du type `this.capacite` dans la classe `Voiture`. Cependant, nous pourrons faire une référence du type `this.getCapacite()`. Ceci indique le fait que la méthode `getCapacite()` est utilisable sans être explicitement écrite pour la classe `Voiture`, puisqu'elle existe dans la superclasse. De plus l'attribut capacite est implicitement utilisé dans la classe `Voiture` (car `getCapacite()` rend justement cette valeur).

- Quel est le mot dédié qui indique à Java qu'une classe hérite d'une autre classe ?

SOLUTION :

Il s'agit du mot `extends` .

- Quel est le mot dédié qui fait référence à une superclasse à partir d'une de ses sousclasses ?

SOLUTION :

C'est le mot `super` .

- Est-ce qu'on peut hériter de plusieurs classes ?

SOLUTION :

Non, en Java on hérite d'au maximum une classe.

- Pourquoi la classe `Voiture` ne peut pas utiliser les constructeurs de la classe `Vehicule` directement ?

SOLUTION :

Rappel : un constructeur est une méthode spéciale, permettant d'instancier des instances de la classe en question. Cette méthode DOIT porter le nom de la classe.

Or, le(s) constructeur(s) de la superclasse portent le nom de la superclasse, tandis que la sousclasse attend un constructeur qui porte le nom dela sousclasse.

Donc la réponse est non, on ne peut jamais utiliser les constructeurs de la classe `Vehicule` directement dans la classe `Voiture`.

- Ecrivez le premier constructeur de la classe `Voiture` en utilisant le constructeur de sa superclasse.

SOLUTION :

On utilise le constructeur de la classe Vehicule dans la classe Voiture en utilisant le mot clé super.

```
	public Voiture(String typeCarburant, double capacite, double carburantRestant, int passagersActuels) {
		super("Voiture", typeCarburant, capacite, carburantRestant, 5, passagersActuels);
	}
```

Ce code appelle le premier constructeur de la classe Vehicule.

- Le deuxième constructeur de la classe `Voiture` crée une voiture qui a le réservoir plein et seulement 1 passager (le chauffeur). Indiquez deux façons d'écrire ce deuxième constructeur et utilisez une de ces méthodes pour écrire le deuxième constructeur.

SOLUTION :

On évite la déduplication de code.

On peut soit utiliser le deuxième constructeur de la superclasse (`Vehicule`), ou on peut utiliser le premier constructeur de la classe `Voiture`. J'ai préféré la deuxième option, car cela demande moins de duplication.

- Dans la classe `Voiture` on veut utiliser un autre format pour renvoyer les attributs des objets de type `Voiture`, notamment :
  `Voiture[typeCarburant> ; Carburant : <carburantRestant> / <capacite> ; Passagers : <passagersActuels> / <maxPassagers>]`

  Quelle est la mention qu'il faut ajouter lorsqu'on réécrit une méthode de la superclasse ?
  Ecrivez la méthode `String toString()` de la classe `Voiture`.

SOLUTION :

Si on modifie une méthode qui existe déjà dans la superclasse il faut utiliser la mention @Override .

La partie la plus difficile de cette méthode est comment avoir accès aux attributs de la superclasse au sein de la sousclasse. Comme nous avons vu avant, la sousclasse ne peut PAS accéder directement aux attributs de la superclasse si ceux-ci sont privés. Cependant, on a un accès direct aux méthodes de la superclasse, y compris aux getters.

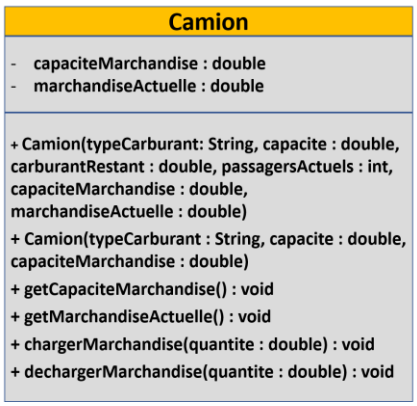
Ce qui donne :

```
	@Override
	public String toString() {
		return("Voiture[" + this.getTypeCarburant() + "; Carburant : " +
this.getCarburantRestant() + "/" + this.getCapacite() + "; Passagers: " +
this.getPassagersActuels() + "/" + this.getMaxPassagers());
	}
```

Exercice 4

Voici le diagramme de classe de la classe `Camion`, qui hérite de la classe `Vehicule` de la même façon que la classe `Voiture`.

La classe `Camion` aura deux attributs propres, au-delà des attributs hérités de la superclasse (`Vehicule`) :



- Un attribut `capaciteMarchandise` de type `double`, qui indique la quantité totale (en m^3) de marchandise que le camion peut porter
- Un attribut `marchandiseActuelle` de type `double` qui indique la quantité totale (en m^3) de marchandise portée actuellement par le camion.

La classe Camion aura également les méthodes suivantes :

- Deux constructeurs (le deuxième met la quantité de marchandise actuelle à 0)
- Deux getters pour les nouveaux attributs
- Deux méthodes pour manipuler la marchandise `void chargerMarchandise(double quantite)` et `void dechargerMarchandise(double quantite).`

On va faire écrire le code suivant :

- Ecrivez le premier constructeur de la classe `Camion`
- Ecrivez la méthode `void dechargerMarchandise(double quantite)`

SOLUTIONS :

La classe Camion a deux types d'attributs : les attributs implicitement hérités de la classe Vehicule et les attributs explicitement déclarés dans la classe Camion. Pour les premiers attributs nous ferons appel au constructeur de la classe Vehicule en utilisant le mot clé super. Pour les attributs spécifiques à la classe Camion, une instanciation directe suffit.

Ce qui donne le code suivant :

```
	public Camion(String typeCarburant, double capacite, double carburantRestant,
int passagersActuels, double capaciteMarchandise, double marchandiseActuelle) {
		super("Camion", typeCarburant, capacite, carburantRestant, 2,
passagersActuels);
		this.capaciteMarchandise = capaciteMarchandise;
		this.marchandiseActuelle = marchandiseActuelle;
	}
```

Pour la méthode `void dechargerMarchandise(double quantite)` nous allons prendre la même approche que pour l'ajout du combustible : on décharge la quantité demandée et, si la quantité demandée dépasse la quantité qui existe actuellement dans le camion, la marchandiseActuelle est mise à 0 et on affiche un message informant l'utilisateur qu'on a vidé le camion :

```
	public void dechargerMarchandise(double quantite) {
		this.marchandiseActuelle -= quantite;
		if (this.marchandiseActuelle < 0) {
			System.out.println("Vous avez vide le camion !");
			this.marchandiseActuelle = 0;
		}
	}
```

Exercice 5

Dans cet exercice le but sera d'utiliser le code réalisé pour les exercices précédents. Sauf indication contraire tout le code ci-dessus sera dans une classe TD4 dans une méthode `public static void main(String[] args)`.

- Dans la méthode principale (main) nous aurons le code suivant :
  `Vehicule[] vehicules = new Vehicule[3]`
  Ce tableau contiendra deux voitures (places 0 et 2) et un camion (position 1), les trois avec ¼ de la capacite maximale de carburant. Vous pouvez choisir les autres paramètres librement.

  Ecrivez du code qui vous permettra justement de remplir le tableau.

SOLUTION :

On a plusieurs solutions pour résoudre cette question. Une façon serait de déclarer et instancier deux voitures et un camion, puis de donner aux éléments du tableau ces valeurs. J'ai préféré une façon plus succincte pour cette instanciation -- notamment en ne donnant qu'une référence mémoire aux objets créés :

```
	Vehicule[] vehicules = new Vehicule[3];
	vehicules[0] = new Voiture("Diesel", 60, 15, 2);
	vehicules[1] = new Camion("Diesel", 160, 40, 2, 100, 15);
	vehicules[2] = new Voiture("Essence", 60, 15, 2);
```

- Ce code est un exemple de quel concept de la Programmation Orientée Objet ?

SOLUTION :

C'est un exemple de polymorphisme. Le tableau vehicules est un exemple de tableau polymorphe.

- Pourquoi est-ce qu'on peut mettre un objet de type `Camion` dans un tableau de type `Vehicule[]` ?

SOLUTION :

La classe `Camion` hérite de la classe `Vehicule`, ce qui veut dire qu'un camion est un type de `Vehicule`. C'est pourquoi on peut spécifier un objet de type `Vehicule` comme étant un camion. L'inverse n'est pas vrai : on ne peut jamais définir un objet déclaré comme étant de type `Camion` en tant qu'un vehicule.

- Utilisez les méthodes `toString()` pour faire afficher les caractéristiques de chaque vehicule. Quelles méthodes sont utilisées dans chaque cas ?
- Faites tous les véhicules faire le plein.
- On suppose qu'il y a deux passagers dans chaque vehicule. Ecrivez du code qui essaie de faire chaque vehicule prendre un nouveau passager. Quel devrait être le résultat ?

SOLUTIONS :

Je vais mettre les trois lignes de code ensemble dans la même boucle for, qui traverse les éléments du tableau. La possibilité de parcourir le tableau et faire exécuter la même instruction (avec des effets différents) est en fait un avantage d'avoir un tableau polymorphe.

```
	for (int i = 0; i<vehicules.length; i++) {
		System.out.println("Pour l'element " + i + " : ");
		System.out.println(vehicules[i]);
		vehicules[i].faireLePlein();
		vehicules[i].prendrePassagers(1);
	}
```

Nous allons premièrement noter que la classe Camion n'a pas une méthode redéfinie toString() : elle garde la méthode toString() de la classe Vehicule (qu'elle hérite automatiquement). Au contraire, la classe Voiture a sa propre méthode toString(). Dans cette boucle, la méthode toString() de la classe Voiture sera appelée pour les objets de type Voiture, notamment pour vehicules[0] et vehicules[2]. Pour vehicules[1] on appelle la méthode hérité toString() de la classe Vehicule.

Nous nous rappelons que la méthode prendrePassagers (de la classe Vehicule) cherche premièrement si on a assez de place dans le vehicule pour prendre plus de passagers. Pour les deux voitures on n'aura aucun problème : une voiture est définie comme ayant 5 places dont on n'occupe actuellement que 2. Pour le camion on ne peut pas prendre plus que deux passagers, donc on déclenche le message d'erreur nous disant qu'on n'a pas assez de place pour prendre des passagers.

- Qu'est-ce que se passe lorsqu'on écrit la ligne de code dans la classe principale :

```
vehicules[1].chargerMarchandise(12.5);?
```

SOLUTION :

La méthode `void chargerMarchandise(double quantite)` n'est pas définie pour la classe `Vehicule`. Elle apparaît seulement dans la classe `Camion`. Comme le tableau `vehicules` est défini en tant que tableau de `Vehicule` (type `Vehicule[]`), nous aurons une erreur de compilation (le compilateur cherche la méthode dans la superclasse, ne la trouve pas et affiche un message).

Pour mieux comprendre la différence entre cette situation et celle de tout-à-l 'heure avec la méthode `toString()`, je vous fais un schéma très primitif et simplifié de comment l'héritage et le polymorphise marche-t-ils :

ACCES :

Attributs privés dans la superclasse :

- Dans la superclasse : accès direct (en utilisant `this.<nom de l'attribut>`)
- Dans la sousclasse : accès indirect via un getter (en utilisant `this.<nom de la méthode getter>`)
- Dans une autre classe (qui n'est pas une sousclasse) : accès via un objet de la superclasse ou de la sousclasse (on définit un objet de la superclasse et on appelle le getter pour cet objet-là).

Méthodes privées dans la superclasse :

- Dans la superclasse : accès direct (en utilisant `this.<nom de la méthode>`)
- Dans une sousclasse : pas d'accès
- Dans une autre classe : pas d'accès

Méthodes publiques dans la superclasse directement héritées (donc toutes méthodes sauf les constructeurs et sauf des méthodes modifiées) :

- Mêmes règles que pour les attributs

Méthodes publiques dans la superclasse qui ne sont pas directement héritées (les constructeurs, une méthode modifiée, etc.)

- Dans la superclasse : accès direct (en utilisant `this.<nom de la méthode>`)
- Dans une sousclasse : accès direct (en utilisant `super.<nom de la méthode>`)
- Dans une autre classe (qui n'est pas une sousclasse) : accès via un objet de la superclasse ou de la sousclasse (on définit un objet de la superclasse et on appelle le getter pour cet objet-là).

Méthodes publiques de la sousclasse :

- Dans la superclasse ou dans une autre classe : accès via un objet de la sousclasse
- Dans la sousclasse : accès direct (`this.<nom de la méthode>`)

UTILISATION :

Un objet déclaré ET instancié en tant qu'objet de la superclasse `<Superclasse> <nom d'objet> = new <Superclasse>(<paramètres du constructeur de la Superclasse>) ;` pourra utiliser seulement les méthodes de la superclasse, telles qu'elle sont définies dans la superclasse.

Un objet déclaré en tant qu'un objet de la superclasse, mais instancié en tant qu'objet de la sousclasse `<Superclasse> <nom d'objet> = new <Sousclasse>(<paramètres du constructeur de la Sousclasse>) ;` pourra utiliser les méthodes déclarées dans la superclasse. De plus si la sousclasse redéfinit une des méthodes déjà existantes dans la superclasse, alors le programme exécute la méthode telle qu'elle est définie dans la sousclasse. Cependant, nous ne pourrons jamais faire exécuter pour cet objet une méthode spécifique à la sousclasse, mais qui n'existe pas dans la superclasse !

Un objet déclaré ET instancié en tant qu'objet de la sousclasse `<Sousclasse> <nom d'objet> = new <Sousclasse> (<paramètres du constructeur de la Sousclasse>) ;` pourra utiliser toutes les méthodes héritées de la superclasse (sauf son constructeur), ainsi que toutes les méthodes spécifiques à la classe. Si une méthode de la superclasse a été modifiée dans la sousclasse, on exécute la méthode modifiée, qui se trouve donc dans la sousclasse.