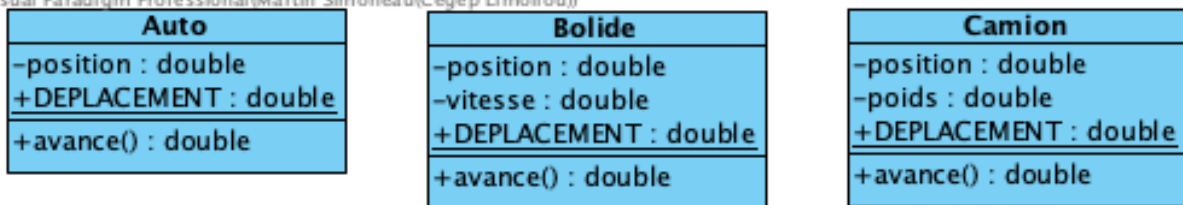


Programmation orientée objet II Hiver 2024 Cégep Limoilou Département d'informatique Professeur : Martin Simoneau	Formatif 2 S2-C2 <i>Compatibilité avec héritage</i>
--	---

S2-C2-p1

Exercice1 – *Auto bolide et camion (extends, Instanceof, @Override)*

Visual Paradigm Professional(Martin Simoneau(Cegep Limoilou))



N.B le camion sera fait à la fin de l'exercice

En regardant les classes *Auto* et *Bolide*, vous constaterez qu'elles ont un attribut *position* et une constante *DEPLACEMENT* en commun. On peut les réutiliser comme nous l'avons vu dans le formatif précédent. Nous observons qu'elles ont également une méthode *avance* avec une signature compatible, mais dont l'implémentation diffère d'une classe à l'autre.

Ouvrez la classe *Application*. Nous allons nous intéresser à la compatibilité des 3 méthodes *avance*. Au début du *main*, il y a 2 boucles qui font avancer chaque type de véhicule. Jusque-là, rien n'a vraiment changé. On place ensuite des *Auto* et des *Bolide* dans un tableau d'objet. Décommentez la boucle suivante. Celle qui contient

```
vehicules[i].avance();
```

Pourquoi cette instruction ne compile pas. *Auto* et *bolide* ont pourtant tous deux une méthode *avance* ! La raison est que le langage *Java* s'assure qu'une méthode peut être appelée au moment de compiler le code. Pour y arriver, elle consulte le type de l'objet. Il s'agit ici de *Object* (le parent de tous les objets Java), cette classe ne possède pas de méthode *avance*, le compilateur refuse alors de compiler le code tout simplement.

- Pour pouvoir appeler ce code, il faut d'abord s'assurer en temps d'exécution que l'objet est bien d'un type connu. On peut le faire avec un test de type.

```
vehicule instanceof Auto
```

- Si le test passe, on peut effectuer le transtypage en toute sécurité.
Auto autoCourante = (Auto) vehicule;
- La variable *autoCourante* est maintenant de type *Auto* et on peut donc appeler la méthode *avance*.
autoCourante.avance();

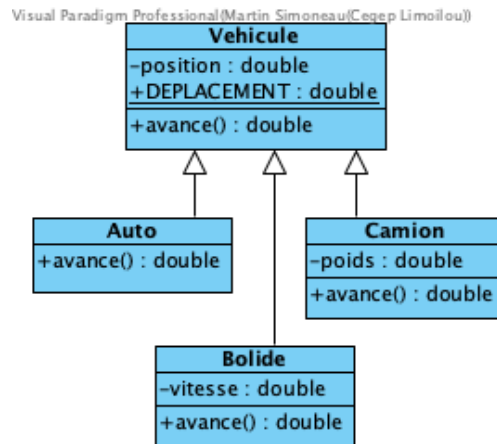
Bien que le transtypage fonctionne, cette façon de faire est considérée comme une TRÈS MAUVAISE approche. Pourquoi ?

- Parce qu'il faut connaître tous les types possédant une méthode *avance* et tester chacun d'eux à chaque endroit où l'on doit appeler cette méthode.
- ■ Le code de l'application doit donc être modifié chaque fois que l'on ajoute un nouveau type qui *avance*. Le code doit être **fermé** pour les modifications (changer le code), mais **ouvert** pour une extension (création de classes enfant).

C'est l'un des 5 principes *SOLID*, très reconnus et importants en conception OO. Avec *SOLID* le *O* signifie *open/close principle*.

S2-C2-p2

Vous allez modifier le code comme suit :



- On ajoute un parent *Vehicule* qui permet de réutiliser *position* et *DEPLACEMENT*.
- Remarquez que *Vehicule* possède aussi une méthode *avance*. Cette méthode ne fait rien. Lorsqu'elle est appelée, elle retourne simplement 0¹. Qu'est-ce que cette méthode peut bien changer ! Tout...
- Dans la méthode main de la classe *Application*, créez un tableau de *Vehicule*.
`Vehicule[] vehiculesPolymorphiques = new Vehicule[6];`
- Placez des objets *Auto* et *Bolide*, dans le tableau. Remarquez que c'est possible parce qu' *Auto* et *Bolide* sont réputés « être des » *Vehicule*. En programmation OO, une classe enfant doit TOUJOURS pouvoir être substituée à toutes ses classes parentes. Un enfant ne fait pas que réutiliser le contenu de ses parents, il devient un type compatible avec ses parents ! C'est le principe de substitution de *Liskov* (le L dans *SOLID*). Vérifiez-le en programmant les lignes suivantes :
`Vehicule vehicule = new Auto();`
`Vehicule vehicule2 = new Bolide(3);`
- Maintenant, essayez d'appeler directement la méthode *avance* dans une boucle sur le tableau de *vehicule* :
- ```
for (int i = 0; i < vehiculesPolymorphiques.length; i++) {
 vehiculesPolymorphiques[i].avance();
 System.out.println("le vehicule par polymorphisme est rendu a " + vehiculesPolymorphiques[i].getPosition());
}
```
- Vous remarquerez que la méthode *avance* qui est appelée n'est pas celle de *Vehicule*, mais celle de *Auto* ou *Camion* selon l'objet qui a véritablement été instancié avec le *new*! Concrètement, lorsqu'une même méthode à la même signature dans une classe enfant que dans la classe parente, on dit qu'elle est **redéfinie** (attention de ne pas confondre *redéfinition* et *surcharge*). Le

<sup>1</sup> On proposera une meilleure solution dans le formatif3

mécanisme qui fait en sorte que c'est toujours la méthode la plus spécialisée (proche de l'enfant) qui est appelée s'appelle le **polymorphisme**.

- Utiliser le polymorphisme doit toujours être priorisé à une solution avec transtypage (évitez les instruction *instanceof* autant que possible).
- Pour vous pratique ajoutez une classe *Camion* enfant de *Vehicule*
  - Elle possède un attribut **poids**
  - La méthode *avance* augmente la position du *camion* de 4 divisé par le poids du *Camion*. Plus il est lourd, moins il avance!
  - Pour faire fonctionner le *Camion* avec la boucle qui utilise le transtypage, il faut ajouter explicitement un cas pour gérer le camion.
  - Remarquez que vous pouvez maintenant ajouter des *Camion* dans le tableau de *véhicules* sans rien avoir à changer dans le code de la boucle qui appelle la méthode *avance*! On respect donc le principe *Open Close* (SOLID) !
- On peut demander au compilateur de nous assurer qu'une méthode qu'on veut redéfinir dans une classe enfant existe bien dans la classe parente (éviter les erreurs de signature). En ajoutant l'annotation **@Override** devant la méthode *avance* des classes enfants, on demande au compilateur de vérifier qu'une méthode avec une signature similaire existe bien dans une classe parent (ici *Vehicule*).

```
@Override
```

```
public double avance()
```

### Exercice2 – Animaux (*extends*, *@Override*)



- Dans le *formatif1* vous avez créé une classification d'animaux (ours, saumon, ver de terre, fourmi). On utilisait alors les méthodes *decritOurs*, *decritSaumon*... Ces méthodes n'étaient malheureusement pas compatibles entre elles. Dans le code fourni, on a changé les *decritNNN* par une méthode ***decritAnimal()***. Cette méthode a la même signature partout, mais les implémentations changent d'un animal à l'autre.
- On vous demande d'utiliser le polymorphisme en redéfinissant cette méthode afin de la rendre compatible partout.
- Considérez ici que chaque mot dans la chaîne de texte est un élément qu'on doit réutiliser au maximum. On vous demande donc d'utiliser le mot clé ***super.decritAnimal()*** afin de réutiliser au maximum des chaînes de texte contenues dans les classes parentes. Remontez les portions de chaîne de caractères qui sont communes à plusieurs enfants.

### Exercice3 – ABC (*super* et chaîne d'appels)

- Le troisième exercice est très simple, il sert à vérifier si vous avez bien compris le fonctionnement du polymorphisme. On vous donne des classes A, B et C qui forment une ligne hiérarchique. A est parent de B et B est parent de C.

- Dans la classe *Application*, on vous demande premièrement de corriger l'erreur d'exécution qui se produit lorsqu'on lance l'exécution.
- Vous devez ensuite suivre l'exécution du programme en mode débogue pour bien comprendre la séquence d'appels des différentes méthodes.
- On vous demande finalement d'expliquer les différentes sorties dans la console en fonction des instructions qui les ont produites.

#### Exercice4 – *Tablettes (heritage et délégation)*

- Héritage 
  - Pour cet exercice, on modifie le code de *Tablette*, *TablettePlus5G* et *TablettePlusLTE*. La première est une simple tablette avec une méthode pour prendre des photos (*capturePhoto*). Les 2 autres (*TablettePlusNN*) sont des tablettes plus évoluées qui peuvent se connecter à un réseau cellulaire et envoyer les photos directement lorsqu'on les prend (*connect*, *envoie* et *capturePhoto* modifié). On vous demande d'utiliser l'héritage pour réutiliser au maximum et pour rendre les méthodes similaires compatibles.
- Héritage et délégation 
  - Pour cet exercice, on refait la même chose que l'exercice précédent, mais cette fois au lieu de n'utiliser que l'héritage on combine héritage et délégation pour produire un code plus flexible (important pour les étudiants intéressés par le développement d'application).
  - Indice : penser à une *antenne*.