La Programmation Orientée Objet (POO)

UMONS – Année académique 2014-2015

Vous avez certainement déjà tous remarqué, lors des travaux pratiques des cours de programmation et algorithmique, que la manière de programmer en Java est différente de celle de programmer en Python. En effet, les langages de programmation (il y en a beaucoup d'autres) suivent un ou plusieurs styles de programmation. Ceux-ci sont appelés *paradigmes de programmation* et permettent de classer les différents langages :

- la programmation fonctionnelle : Scheme, Python, OCaml, Haskell, ... ;
- la programmation impérative/procédurale : Python, OCaml, Fortran, Cobol, C, ... ;
- la programmation orientée objet : Java, Python, OCaml, C++, Ada, ... ;
- la programmation déclarative : Prolog, Datalog, ... ;
- la programmation concurrente : Ada, ...

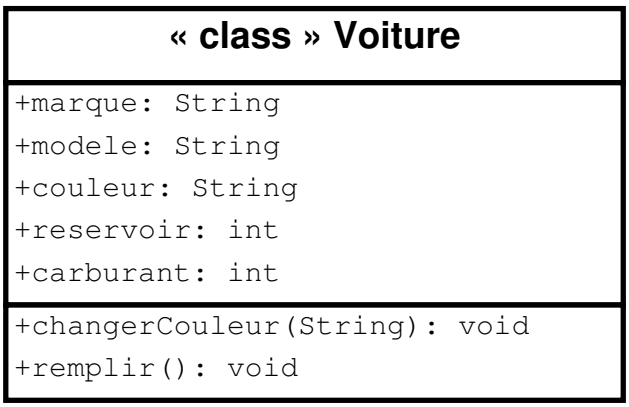
Un langage offre donc des outils et des concepts spécifiques au paradigme de programmation sur lequel il est conçu. Ce document a pour objectif de présenter différents concepts offerts par la programmation orientée objet utiles à la réalisation de votre projet en Java.

1 Classe et objet

Réf. au cours de Prog. et Algo. 2 : Chapitre 2.

Un programme écrit dans un langage orienté-objet est principalement subdivisé en plusieurs classes. Une classe est une structure modélisant une personne, un objet, un lieu, un concept, etc., en lui associant une liste de propriétés et de comportements. Dans une classe, les variables d'instances peuvent être vues comme les propriétés et les méthodes comme les comportements. Un objet (ou une instance) est alors issu d'une classe en attribuant à chacune de ses propriétés une valeur bien précise.

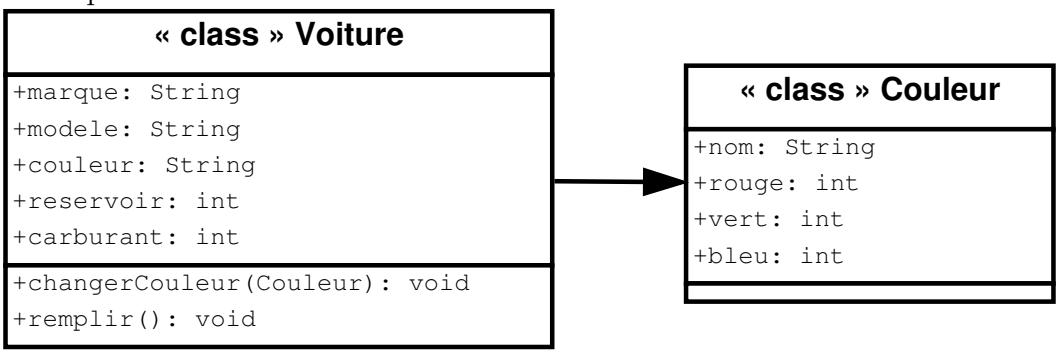
Exemple : Une voiture peut être déterminée avec sa `marque`, son `modèle`, sa `couleur`, le volume de son `réservoir` et la quantité de `carburant` courante dans son réservoir. À partir d'une classe `Voiture`, nous pouvons alors créer les objets "Audi A4 rouge avec 30l de carburant sur 50l", "Fiat Punto blanche avec 13l de carburant sur 45l", ... Des comportements que l'on pourrait associer à une voiture sont de faire le plein (`remplir()`) et de modifier sa couleur (`changerCouleur()`).



Les classes d'un programme peuvent être liées entre elles de plusieurs manières.

— l'héritage : voir Section 3

— la contenance : se dit lorsqu'un objet en contient un autre.
Exemple : Une couleur pourrait être un objet créé à partir d'une classe `Couleur` dont les propriétés seraient le `nom` de la couleur et l'`intensité` de chaque couleur primaire qui la compose. Un objet `Voiture` contiendrait alors un objet `Couleur` pour représenter sa couleur.



2 Polymorphisme

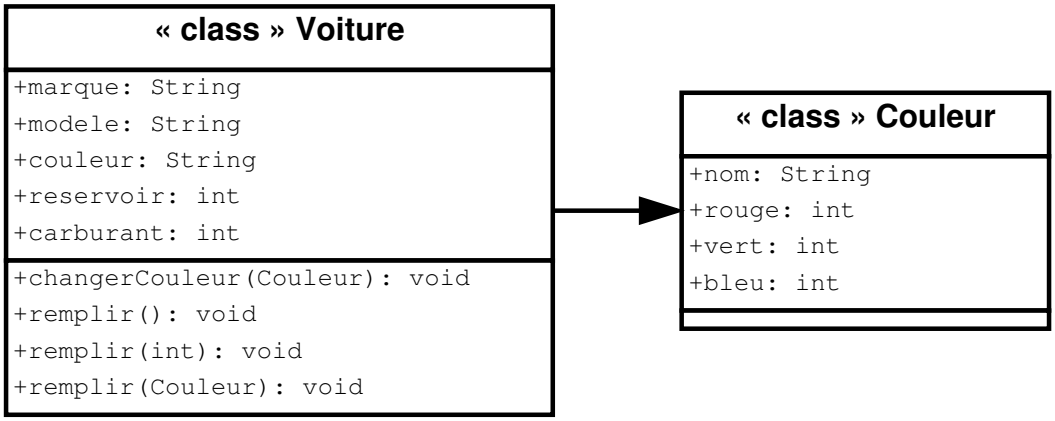
Le mot polymorphisme provient du grec et signifie "qui peut prendre plusieurs formes". En POO, nous parlons de *polymorphisme ad-hoc* lorsque plusieurs méthodes du même nom ont des comportements qui diffèrent en fonction du type et du nombre de paramètres de la méthode. Nous parlerons plus tard du *polymorphisme d'héritage*. Il existe aussi le *polymorphisme paramétrique* (*generics* en Java) qui permet d'écrire du code générique, manipulant des objets d'un type inconnu (mais paramétré) au moment de l'implémentation (voir Chapitre 11 du cours de Prog. et Algo. 2).

Une classe peut définir deux méthodes de même nom ayant des paramètres différents (en type et/ou en nombre).

Exemple : La classe `Voiture` peut définir deux autres méthodes ayant pour nom `remplir` :

— `remplir(int n)` : permettant d'ajouter `n` litres dans le réservoir ;

— `remplir(Couleur c)` : permettant de remplir le réservoir uniquement si la voiture est de couleur `c`.



Si on crée une voiture `voit` et une couleur `coul`, les appels aux différentes méthodes `voit.remplir()`, `voit.remplir(5)` et `voit.remplir(coul)` auront les effets escomptés.

3 Héritage

Réf. au cours de Prog. et Algo. 2 : Chapitre 4.

Le principe d'héritage permet de lier des classes d'une manière particulière et d'éviter la redondance de code.

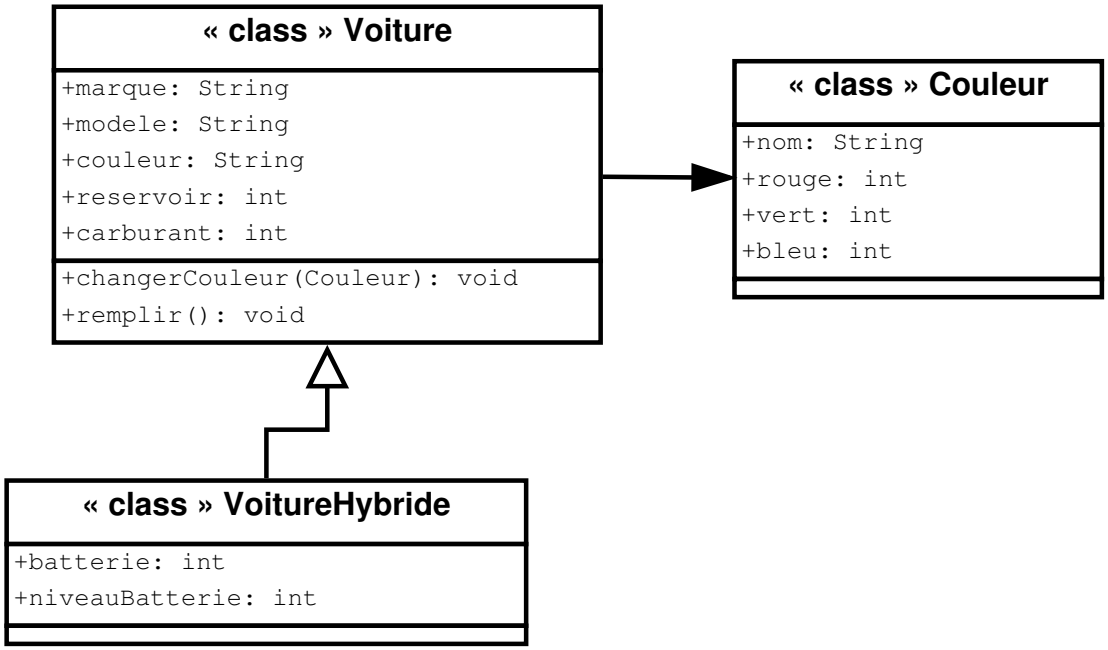
L'héritage peut s'appliquer lorsqu'un objet est plus spécifique (ou plus détaillé) qu'un autre. Certains comportements seront alors identiques aux deux objets alors que d'autres seront adaptés aux spécificités du premier objet.

Exemple : On peut considérer une voiture hybride comme une voiture avec un réservoir électrique supplémentaire. Tout ce qui a été codé dans la modélisation d'une voiture peut donc être réutilisé dans la modélisation d'une voiture hybride. Nous n'allons pas coder à nouveau tout ce qui a déjà été fait dans la classe `Voiture`. Le mot clé `extends`, disponible à Java, permet à la classe `VoitureHybride` d'hériter des méthodes et des variables définies dans `Voiture`. Seuls les éléments d'une voiture hybride non présents dans une simple voiture seront implémentés dans cette classe. Nous dirons alors que la classe `VoitureHybride` est une fille de la classe `Voiture`.

Il est donc possible d'ajouter de nouvelles variables et de nouvelles méthodes spécifiques aux classes filles. Il est également possible d'en redéfinir si elles sont déjà présentes dans la classe parent.

Exemple : Les variables d'instance `batterie` et `niveauBatterie` sont ajoutées à la classe `VoitureHybride` pour indiquer, respectivement, la capacité électrique de la voiture et le niveau courant de la batterie. La méthode `voit.remplir()` peut-être redéfinie dans

`VoitureHybride` pour que son nouveau comportement remplisse également la batterie électrique de la voiture.



La méthode qui sera exécutée dépendra alors de l'objet qui l'invoque, il s'agit de *polymorphisme d'héritage*. Pour plus de précision à ce niveau, voir Chapitre 4 (slides 50-64) du cours de Programmation et Algorithmique 2.

4 Classes abstraites

Réf. au cours de Prog. et Algo. 2 : Chapitre 5.

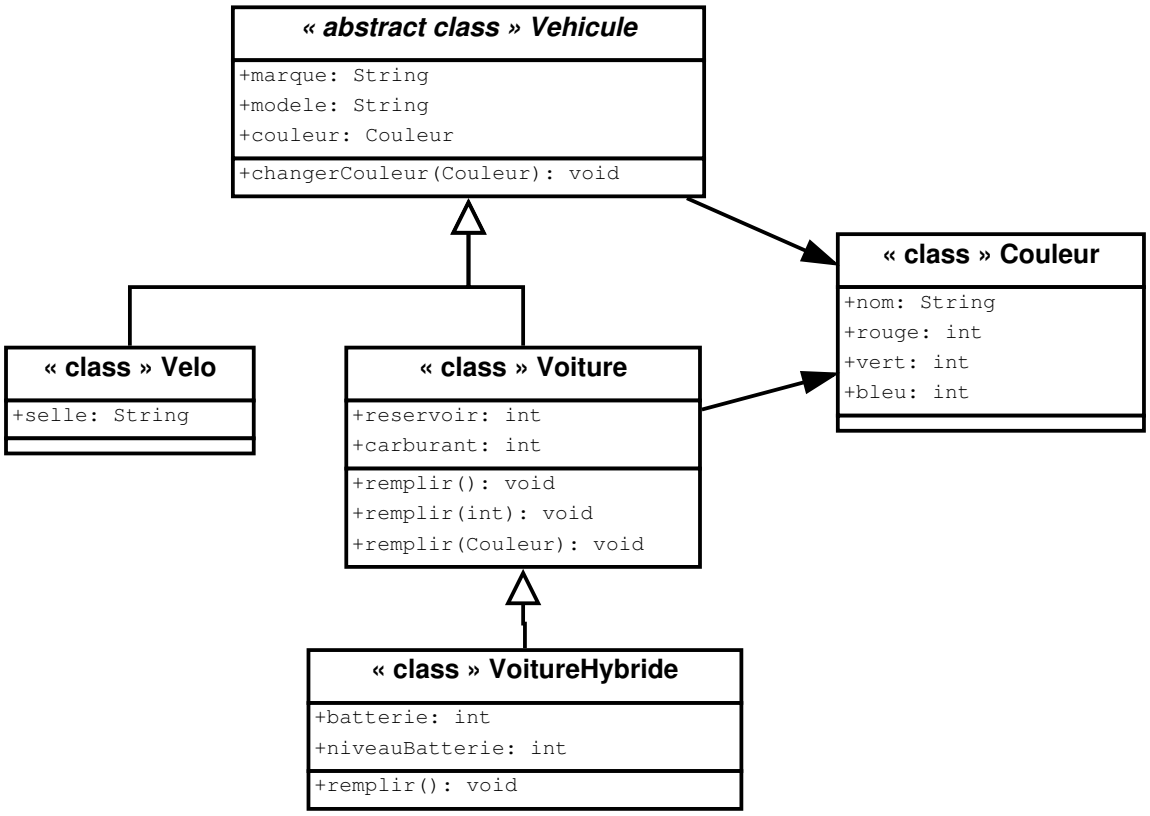
Dans certains cas, deux classes peuvent avoir des points communs, mais aucune d'elle n'est plus spécifique que l'autre. Dans ce cas, l'héritage ne peut être directement appliqué sur ces deux classes.

Pour ne pas coder deux fois les éléments en commun, nous utilisons une troisième classe qui contient ce code et qui sera parent des deux autres. Ces dernières héritent alors de leur mère et ne contiennent que les variables et méthodes qui leurs sont spécifiques.

Exemple : Considérons la classe `Velo` caractérisant un vélo par sa `marque`, son `modèle`, sa `couleur` et son type de `selle`. Clairement, un vélo n'est pas une voiture car il n'a pas de réservoir tout comme une voiture n'est pas un vélo spécifique, car elle n'a pas de selle. La classe `Vehicule` reprendra alors les points communs de tous les véhicules (vélo et voiture dans notre cas). Elle contient donc la marque, le modèle, la couleur du véhicule ainsi que la méthode `changerCouleur()` qui a le même comportement quelque soit le véhicule. Ces éléments ne sont plus repris dans les deux autres classes qui les hériteront directement de `Vehicule`.

La notion de classe abstraite intervient lorsque l'on crée une telle classe reprenant des éléments, mais dont la création d'objet pour cette classe n'a pas de sens. Le mot-clé `abstract` permet de préciser qu'une classe est abstraite.

Exemple : C'est le cas de la classe `Vehicule` qui n'est pas assez spécifique que pour donner la possibilité de créer un objet. Son rôle permet uniquement d'éviter la redondance de code. La déclaration de la classe est de la forme `public abstract Vehicule`.



5 Interface

Réf. au cours de Prog. et Algo. 2 : Chapitre 5.

5.1 Le principe

Une interface est une liste de (signatures de) méthodes liées à une propriété, une caractéristique, un modèle, ...

Exemple : L'interface `Comparable` de Java représente le fait que deux objets d'une même classe peuvent être comparés. Cette interface contient alors la signature de méthode `compareTo` dont le but est d'indiquer si un objet est plus petit, plus grand ou égale qu'un autre. C'est le cas des objets de type `String`, `Integer`, ... Dès lors, si

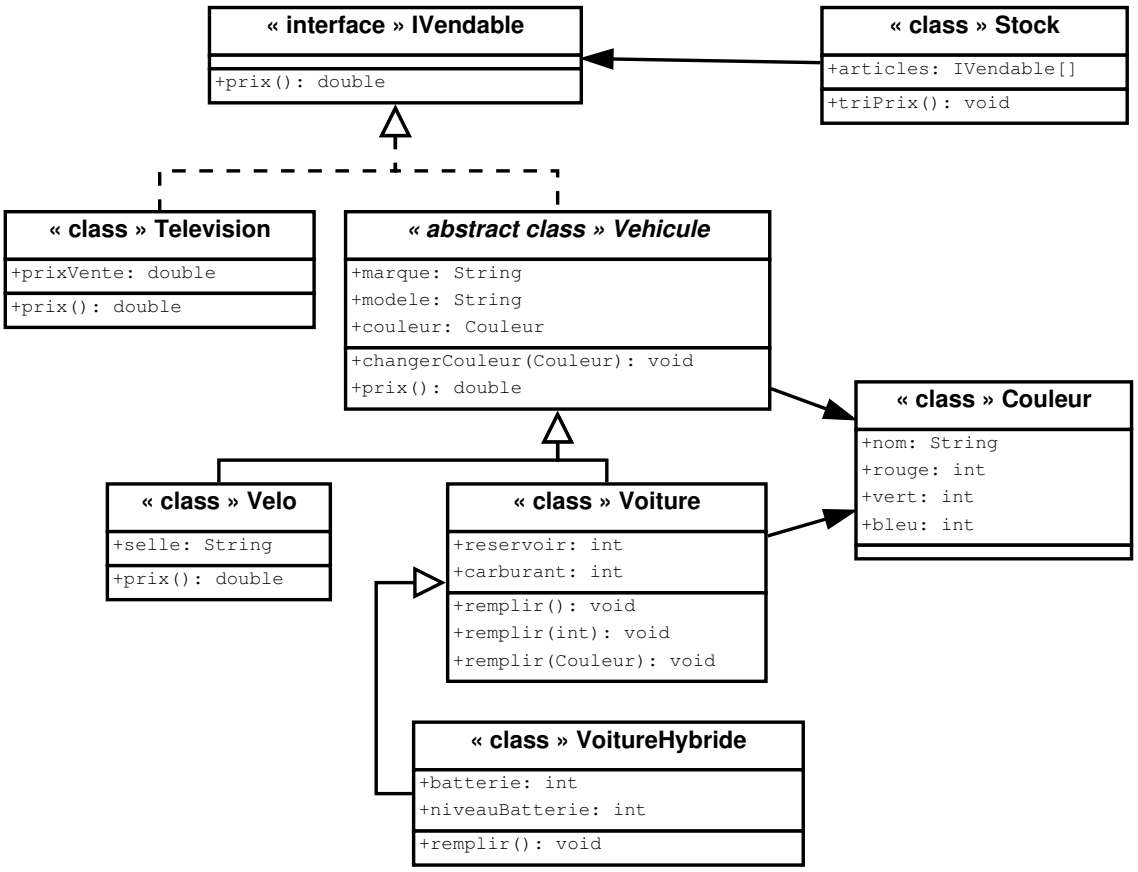
vous créez une classe dont les objets sont comparables, vous pouvez indiquer qu'elle *implémente* l'interface `Comparable` à l'aide du mot clé `implements`. Dans ce cas, il est indispensable d'implémenter toutes les méthodes listées dans l'interface.

Remarque : Seules les signatures de méthodes sont présentes dans une interface, on n'y trouve aucune implémentation. Le mot clé `implements` est donc approprié pour indiquer que la classe "implémente" les méthodes de l'interface.

5.2 L'intérêt

Il se peut qu'une méthode soit applicable non pas à un objet de type particulier, mais à des objets respectant une certaine propriété, i.e., des objets dont la classe implémente une certaine interface. Pour ne pas devoir créer cette méthode pour chacune de ces classes, Java permet de déclarer une variable avec le nom d'une interface. Une telle variable peut alors être initialisée avec un objet dont la classe implémente l'interface en question.

Exemple : Considérons l'interface `IVendable` qui spécifie qu'un objet peut être vendu. Cette interface contient dès lors la signature d'une méthode `prix()` dont le rôle est de retourner le prix de l'objet. Considérons aussi la classe `Television` permettant d'instancier des télévisions. Les télévisions ainsi que les véhicules sont des objets vendables. Nous implémentons alors la méthode `prix()` dans chacune de ces classes. Considérons maintenant une classe `Stock` qui gère les stocks d'objets vendables. Cette classe peut contenir alors une liste d'objets dont la classe implémente l'interface `IVendable` sans se soucier du type de l'objet. La méthode `triPrix()` permet de trier la liste des objets en fonction de leur prix. Cette méthode peut donc appeler la méthode `prix()` quelque soit l'objet de la liste, mais en aucun cas une méthode non présente dans l'interface.



Remarque : Si une classe implémente une interface, toutes ses filles l'implémentent implicitement. En effet, si une classe implémente les méthodes de l'interface, les filles héritent de ces méthodes, elles implémentent donc également l'interface.

5.3 Ne pas confondre avec Interface graphique et Interface publique

L'interface graphique (ou GUI) est l'ensemble des composants graphiques offerts par un programme permettant une interaction plus conviviale avec l'utilisateur.

L'interface publique reprend la liste des constructeurs, variables, méthodes ainsi qu'une description de chacun d'eux, qu'une classe met à disposition de l'utilisateur. L'interface publique peut également fournir les liens d'héritages, la liste des interfaces implémentées ou encore le package dans lequel une classe se trouve.

6 Synthèse

Classe Abstraite	**Interface**
Pas de création d'objet possible	Pas de création d'objet possible
Implémentation de méthodes autorisée, mais pas obligatoire	Pas d'implémentation de méthode, juste une liste de signature
Utile pour rassembler des **implémentations** communes à plusieurs classes	Utile pour spécifier des **propriétés** communes à plusieurs classes, mais dont le comportement peut différer.
Une classe ne peut hériter que d'une classe (abstraite)	Une classe peut implémenter plusieurs interfaces

7 Exercice

Modélisez votre programme en différentes classes. Intégrez et utilisez au mieux les notions décrites dans ce document. Réalisez ensuite le diagramme de classe de votre programme.

Questions à se poser :
— Quels objets vais-je devoir créer ?
— Quel est le rôle précis des mes différentes classes ?
— Quels sont les liens entre mes différentes classes ?
— Quels sont les points communs de mes différentes classes ?
— Est-ce que j'évite au plus la redondance de code ?