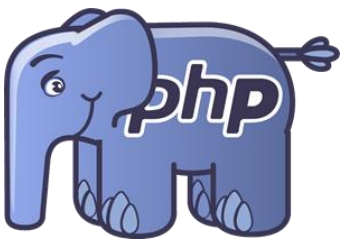
Afpa

Exercices

Réutiliser et spécialiser les classes



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**.

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

3) **Vocabulaire utilisé** : encapsulation, classe, héritage, classe abstraite, polymorphisme, …

4) **Environnement technique** : PHP, un EDI (type Visual Studio Code, *Eclipse, NetBeans*).
La documentation PHP : https://www.php.net/manual/fr/book.classobj.php.

5) **Pré-requis et recommandations** :

a) N'abordez pas les TPs dédiés aux **classes** et **héritage** tant que ces notions ne vous sont pas complétement 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 **PHP** nommé *JeuEchecs* avec l'IDE de votre choix, VSC, *Eclipse*, *NetBeans*, ...

Vous rangerez les classes métier dans le package *classes* et la classe *Principale* à la racine du projet.

Remarques : Revoyez éventuellement comment créer un projet avec *votre IDE.*

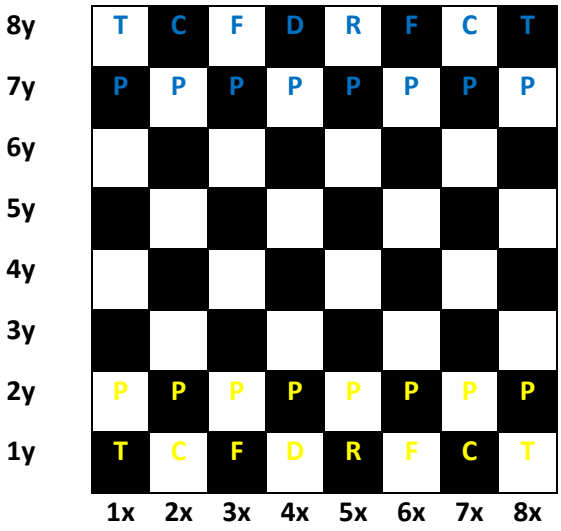
Au cours de ce TP, vous mettrez en évidence :

- ✔ 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



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. (*Le type* int *ne vous est donné qu'à titre d'information, PHP ne permet pas de typer les variables*).
- 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.
- Une méthode *peutAllerA($x, $y)* qui renvoie une valeur booléenne indiquant si la pièce en question peut aller en case ($x, $y), compte tenu de sa position actuelle.

Testez avec une application.

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($x, $y)* qui retourne vrai si la position passée en paramètre fait partie de l'échiquier.

3. Tableau de PieceEchecs

Depuis une application PHP, créez un tableau de pièces d'échecs. Chaque pièce sera soit un *Fou* soit un *Cavalier*.
L'application déterminera pour chaque pièce si elle peut ou non aller à la case (5,5).

4. Méthode peutAllerA()

Pratique la manipulation des pièces dans un tableau, à condition que tous les objets placés dans le tableau possède la méthode peutAllerA() !

Mettez en place une solution pour éviter que l'on oublie cette méthode dans les futurs développements.

5. Classe Roi

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

Testez avec une application

6. Classe Pion

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

Testez avec une application

7. Méthode peutManger(PieceEchecs $piece)

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

Testez avec une application

TP 2 : *ParcAuto*

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 : « ***Réutiliser et spécialiser les classes*** ».

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 **PHP** nommé *ParcAuto* avec l'IDE de votre choix.

Vous rangerez les classes métier dans le répertoire *classes* et la classe *Principale* à la racine du projet.

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 : *volumeReservoir* (de type *float*) représentant le nombre actuel de litres dans le réservoir, *volumeTotal* (de type *float*) représentant le nombre total de litres que le moteur a reçu au fil des pleins effectués et le booléen *demarre* qui précise si le moteur tourne ou non.

10. Ajoutez les accesseurs en lecture pour *volumeReservoir, volumeTotal* et *demarre .*

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 *volumeReservoir* et *volumeTotal.* Affichez l'action effectuée comme, par exemple :

```
echo ("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 function demarrer() : bool
{
        return $this->moteur->demarrer();
}
...
public function arreter() {
        $this->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 function rouler(float $volume) {
        if ( ! $this->isDemarre()) {
            $this->demarrer();
        }
        $carburant = $this->utiliser($volume);
        .....
}
```

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 le script de test, 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.

```
<?php // parcAuto.php
spl_autoload_register(function($classe){
   include "classes/" . $classe . ".class.php";
});
echo "Bonjour\n";
$laguna = new Voiture("Renault", "Laguna", 30);
echo $laguna . "\n";
$laguna->demarrer();
$laguna->rouler(25);
echo $laguna . "\n";
```

18. Complétez si besoin toutes les méthodes et fonctions d'affichage nécessaires pour produire le résultat suivant :

```
> php -f parcAuto.php
Bonjour
Voiture: Renault-Laguna, il reste 30 litre(s) dans le réservoir
Le moteur est démarré avec 30 litre(s) dans le réservoir
Consommation de 1/10 litre pour démarrer, il reste 29.9 litre(s)
Le moteur utilise 25 litre(s), il reste 4.9 litre(s)
Voiture: Renault-Laguna, il reste 4.9 litre(s) dans le réservoir
```

19. 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-t-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*** levez, grâce à la clause *throw*, une exception de type *PanneEssenceException*. Vous allez donc créer cette classe dans le répertoire *classes*. Elle héritera de la classe **PHP** *Exception*.

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

```
<?php // classes/PanneEssenceException.class.php
class PanneEssenceException extends Exception
{
}
```

Pas besoin d'en faire plus.

Maintenant la méthode *rouler()* peut lever cette nouvelle exception :

```
public function rouler (float $volume)
{
   if ($carburant = 0)
      // levée de l'exception
      ...
}
```

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

```
<?php  // parcAuto.php
...
echo "Bonjour\n";
$laguna = new Voiture("Renault", "Laguna", 30);
echo $laguna."\n";
try {
   $laguna->rouler(50);
} catch (PanneEssenceException $e) {
   echo "La laguna vient de tomber en panne : " . $e->getMessage() . "\n";
}
echo $laguna . "\n";
```

```
> php -f parcAuto.php
Bonjour
Voiture: Renault-Laguna, il reste 30 litre(s) dans le réservoir
Le moteur est démarré avec 30 litre(s) dans le réservoir
Consommation de 1/10 litre pour démarrer, il reste 29.9 litre(s)
Le moteur utilise 29.9 litre(s), il reste 0 litre(s)
La laguna vient de tomber en panne : Panne d'essence !
Voiture: Renault-Laguna, il reste 0 litre(s) dans le réservoir
```

20. 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.

21. 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.

22. 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.

23. 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.

24. A la fin du trajet, affichez le nombre total de litres restant et le nombre total de litres versés dans le réservoir. Soit :

```
> php -f parcAuto.php
Bonjour
Voiture: Citroën-C5, il reste 40 litre(s) dans le réservoir

Parcours de 6 fois 10 :
Le moteur est démarré avec 40 litre(s) dans le réservoir
Consommation de 1/10 litre pour démarrer, il reste 39.9 litre(s)
Le moteur utilise 10 litre(s), il reste 29.9 litre(s)
Le moteur utilise 10 litre(s), il reste 19.9 litre(s)
Le moteur utilise 10 litre(s), il reste 9.9 litre(s)
Le moteur utilise 9.9 litre(s), il reste 0 litre(s)
La C5 vient de tomber en panne : Panne d'essence !
Je vais faire le plein ...
Le moteur est arrêté
Je fais le plein avec 50 litre(s)
Le moteur est démarré avec 50 litre(s) dans le réservoir
Consommation de 1/10 litre pour démarrer, il reste 49.9 litre(s)
Le moteur utilise 10 litre(s), il reste 39.9 litre(s)
Le moteur utilise 10 litre(s), il reste 29.9 litre(s)
Le moteur est arrêté
Consommation du trajet : 59.9 litre(s)
Nombre total de litres versés : 90 litre(s), reste 29.9 litre(s)
```

1. Créez maintentant 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.

```
> php -f parcAuto.php
Bonjour
Scooter: Yamaha-X-MAX, il reste 20 litre(s) dans le réservoir

Parcours de 3 fois 10 :
Le moteur est démarré avec 20 litre(s) dans le réservoir
Consommation de 1/10 litre pour démarrer, il reste 19.9 litre(s)
Le moteur utilise 10 litre(s), il reste 9.9 litre(s)
Le moteur utilise 9.9 litre(s), il reste 0 litre(s)
L'X-MAX vient de tomber en panne : Panne d'essence !
Je vais faire le plein ...
Le moteur est arrêté
Je fais le plein avec 15 litre(s)
Le moteur est démarré avec 15 litre(s) dans le réservoir
Consommation de 1/10 litre pour démarrer, il reste 14.9 litre(s)
Le moteur utilise 10 litre(s), il reste 4.9 litre(s)
Le moteur est arrêté
```

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 implémenté sous forme d'un tableau.

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 :

```
> php -f parcAuto.php
Bonjour
Les véhicules du parc :
Voiture: De Lorean-DMC-12, il reste 1 litre(s) dans le réservoir
Voiture: Dodge-Charger, il reste 1 litre(s) dans le réservoir
Voiture: Cadillac-Miller-Meteor, il reste 3 litre(s) dans le réservoir
Voiture: Ford-Gran Torino, il reste 3 litre(s) dans le réservoir
Scooter: Piaggo-Vespa, il reste 5 litre(s) dans le réservoir
Scooter: Honda-Dax, il reste 4 litre(s) dans le réservoir
Scooter: Peugeot-S57C, il reste 2 litre(s) dans le réservoir

Contrôle du parc :
Démarrer le véhicule : 1 (DMC-12) de type : Voiture
Le moteur est démarré avec 1 litre(s) dans le réservoir
Consommation de 1/10 litre pour démarrer, il reste 0.9 litre(s)
Trajet de : 4
Le moteur utilise 0.9 litre(s), il reste 0 litre(s)
DMC-12, vient de tomber en panne : Panne d'essence !
Je vais faire le plein ...
Le moteur est arrêté
Je fais le plein avec 5 litre(s)
Le moteur est démarré avec 5 litre(s) dans le réservoir
Consommation de 1/10 litre pour démarrer, il reste 4.9 litre(s)
Le moteur est arrêté
...
Démarrer le véhicule : 7 (S57C) de type : Scooter
Le moteur est démarré avec 2 litre(s) dans le réservoir
Consommation de 1/10 litre pour démarrer, il reste 1.9 litre(s)
Trajet de : 3
Le moteur utilise 1.9 litre(s), il reste 0 litre(s)
S57C, vient de tomber en panne : Panne d'essence !
Je vais faire le plein ...
Le moteur est arrêté
Je fais le plein avec 9 litre(s)
Le moteur est démarré avec 9 litre(s) dans le réservoir
Consommation de 1/10 litre pour démarrer, il reste 8.9 litre(s)
Le moteur est arrêté
```

TP 3 : Loc*Auto*

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 : « ***Réutiliser et spécialiser les classes*** ».

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

Il est demandé de créer un projet **PHP** nommé *LocAuto* avec l'IDE de votre choix

Vous rangerez les classes métier dans le répertoire *classes* et le script *principal locAuto.php* à la racine du projet.

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 :

- Une société de location « Loc-auto » souhaite informatiser son parc de véhicules selon les spécifications suivantes :

- Loc-auto propose trois types de véhicules à la location : Citadine, Familiale et Utilitaire.

- Chaque citadine est caractérisée par une marque, un modèle, une autonomie et un identifiant (qui sera son immatriculation).

- Chaque familiale est caractérisée par une marque, un modèle, un identifiant, un nombre de passagers maxi.

- Chaque utilitaire est caractérisé par une marque, un modèle, un identifiant et une charge utile.

- Chaque véhicule doit faire l'objet d'une révision périodique à un kilométrage et une fréquence spécifiques selon sa catégorie.

Problème :

- On souhaite construire un modèle de classes représentant chaque type de véhicule. La consigne pour une bonne modélisation étant de trouver les propriétés communes entre classes et celles qui caractérisent, au contraire, spécifiquement chacune d'elles.

- Voyons comment gérer les points communs, mais aussi les différences entre ces 3 types de véhicules.

Quelles sont les propriétés communes ?

- A la lecture du cahier des charges, on distingue 3 propriétés communes à chaque type de véhicule : *marque*, *modèle* et *identifiant*.

- On retrouvera donc ces propriétés en tant que variables d'instances pour chaque objet créé, issu des trois classes.

- La classe *Citadine* possède une propriété supplémentaire : l'*autonomie*.

- La classe *Familiale* possède une propriété supplémentaire : le *nombre de passagers maxi*.

- La classe *Utilitaire* possède une propriété supplémentaire : la *charge utile.*

- Concernant les méthodes maintenant :

  - on va doter chaque classe d'une méthode de description *toString()* qui va varier d'un véhicule à l'autre mais qui s'appuiera sur une partie commune aux 3 classes.

  - chaque classe possédera sa propre méthode *planifierRevision()* : les 3 méthodes porteront le même nom dans chaque classe mais auront une forme et un comportement distincts les unes des autres.

L'idée généraliste de la modélisation est de mettre en facteur et de remonter les informations communes le plus haut possible dans une hiérarchie.

C'est ainsi que l'on va créer la classe *Vehicule* dans laquelle on va retrouver toutes les variables d'instances communes aux 3 types de véhicules à louer. Cette classe sert à factoriser du contenu (les variables d'instances) et la façon de les valoriser : le constructeur. Une méthode de description initiale sera aussi présente dans cette classe.

Mais cette classe **ne peut donner lieu à une instanciation** : on peut louer **concrètement** une citadine, une familiale, ou encore un utilitaire mais pas un véhicule : ce concept de véhicule est *abstrait* et pas assez précis pour qu'il soit éligible à un phénomène d'instanciation.

La classe *Vehicule* est **abstraite**

- Construisons cette classe particulière *Vehicule* :

```php
<?php //classes/Vehicule.class.php
abstract class Vehicule
{
    protected $marque; // marque du véhicule
    protected $modele; // modèle du véhicule
    protected $identifiant;

    /**
     * Constructeur
     *
     * @param string $marque
     * @param string $modele
     */
    public function __construct(string $marque, string $modele, string $identifiant)
    {
        $this->marque = $marque;
        $this->modele = $modele;
        $this->identifiant = $identifiant;
    }

    /**
     * Planifie une révision
     *
     * @return void
     */
    public abstract function planifierRevision();

    /**
     * Retourne les propriétés de l'objet
     *
     * @return string
     */
    public function __toString() : string
    {
        return $this->marque . " " . $this->modele . ", " . $this->identifiant;
    }

}
```

Afin de bien appréhender ce concept et ceux qui vont suivre, construisez ce TP au fur et à mesure des explications, vous allez l'enrichir petit à petit pour aboutir à une solution complète et parfaitement maîtrisée.

Remarques diverses et variées résultant de la lecture du code de la classe abstraite *Vehicule* :

1. La classe est qualifiée d'*abstract* .Conséquence immédiate : elle n'est pas instanciable. C'est-à-dire qu'on ne peut pas exécuter la séquence suivante :

```
$vehicule = new Vehicule(…..) ;
```

   En clair, on ne peut pas instancier un « *Vehicule* » au sens classe du terme.

2. Les 3 variables d'instance *marque*, *modèle* et *identifiant* se retrouvent bien factorisées dans cette classe abstraite.

3. La méthode de description *toString()* permet d'afficher ces 3 informations communes aux 3 types de véhicules.

4. La classe *Vehicule* est dotée d'un constructeur qui prend 3 arguments, précisément ceux dont on a besoin pour valoriser ces ***v.i.***

5. Question taraudante : comment se fait-il qu'une classe qui n'est pas instanciable soit munie d'un constructeur ? Hum ? Bizarre …

Tentez d'y réfléchir et d'y apporter une réponse.

6. Le dernier point remarquable concerne :

```
public abstract function planifierRevision() ;
```

On y voit une méthode *planifierRevision()* qui n'est que déclarée : elle ne possède pas de paire *{ … }* mais uniquement un point-virgule ;

De plus, elle est elle aussi qualifiée d'abstract. Que cela signifie-t-il ?

Réponse : cela implique que toute sous-classe qui va hériter de cette classe abstraite *Vehicule* va devoir de façon contraignante et obligatoire définir (c'est-à-dire fournir une définition avec *{ … }*) pour cette méthode .

Si elle ne le fait pas, elle ne pourra être instanciée à son tour et obligera son concepteur à la qualifier aussi d'abstract pour ne pas s'arrêter au stade de la compilation.

Hériter d'une classe abstraite

Rappels :

- La classe *Vehicule* n'est pas instanciable.
- Elle **déclare** la méthode **abstraite** *planifierRevision()* : toute sous-classe de *Vehicule* devra définir cette méthode pour être instanciable.
- Définissons maintenant les classes concrètes *Citadine*, *Familiale* et *Utilitaire*.

```php
<?php //classes/Citadine.class.php
class Citadine extends Vehicule
{
    private $autonomie;

    public function __construct(string $marque, string $modele,
          string $identifiant, int $autonomie)
    {
        parent::__construct($marque, $modele, $identifiant);
        $this->autonomie = $autonomie;
    }

    public function planifierRevision()
    {
        echo "Planifier révision pour une citadine\n";
    }

    public function __toString() : string
    {
        return parent::__toString() . ", autonomie : " . $this->autonomie . " kms";
    }

}
```

Commentaires sur la classe *Citadine* :

- Son constructeur fait appel au constructeur de la classe abstraite *Vehicule* afin de lui confier la valorisation des ***v.i.*** et de les centraliser en un seul endroit.
  On voit donc qu'un constructeur d'une classe abstraite peut être sollicité comme n'importe quelle méthode même si cette classe n'est pas instanciable !

- L'instruction :

```
parent::__construct(marque, nom,  identifiant) ;
```

- … est la 1ere instruction du constructeur de la classe *Citadine*. Une fois exécuté, on termine en valorisant la ***v.i. autonomie***, spécifique à *Citadine*.

- La méthode de description *toString()* s'appuie sur *toString()* (*parent ::__toString()*) de *Vehicule* puis la complète pour afficher l'autonomie.

- Nous avons eu **l'obligation** de respecter l'imposition exprimée dans la classe abstraite *Vehicule* : celle qui consiste à définir *planifierRevision()* qui est déclarée *abstract* dans cette classe.

Rappel : lorsqu'une référence sur une instance *$ref* est fournie dans un ordre d'affichage *echo($ref)*, PHP déclenche automatiquement la méthode *toString()*, spécifique au type de l'objet si elle a été redéfinie ; ce qui est le cas pour *Citadine* .

- 21 - Précisions sur les classes abstraites

Une classe abstraite est définie grâce au modificateur *abstract*. Elle peut contenir deux types de méthodes :

- des méthodes abstraites : elles sont déclarées avec le modificateur *abstract* mais ne sont pas définies : elles jouent le rôle d'un **outil de spécification,** en forçant toute sous-classe instanciable à implémenter le comportement correspondant.

- des méthodes non abstraites, complètement définies, qui représentent soit des comportements par défaut pour l'héritage, soit un comportement commun hérité.

Classes abstraites et méthodes abstraites

- Une classe est abstraite si elle est définie avec le modificateur *abstract*. Elle peut déclarer zéro ou plusieurs méthodes abstraites.

- Une méthode abstraite est déclarée (et non définie) avec le modificateur *abstract*.

- Une classe ***MaClasse*** qui hérite d'une classe abstraite ***ClasseAbstraite*** doit implémenter toutes les méthodes déclarées *abstract*, sauf si elle est elle-même **abstraite**.

- Si ***MaClasse*** n'est pas définie *abstract*, et si toutes les méthodes abstraites héritées de ***ClasseAbstraite*** ne sont pas définies dans ***MaClasse***, la compilation de ***MaClasse*** provoque une erreur.

- 22 - Enregistrer les instances issues de la hiérarchie *Vehicule*

Poursuite de l'étude de cas :

Nous souhaitons pouvoir enregistrer chaque véhicule proposé à la location. On va créer pour cela une classe *ParcVehicules* au sein de laquelle une méthode nommée *enregistrer* va permettre de déclencher des ordres visant à stocker chaque type de véhicule et ses variables d'instances associées.

La classe *ParcVehicules*

- Cette classe sert à enregistrer les objets concrets représentant des véhicules. Elle définit une méthode *enregistrer* qui prend en argument l'objet nommé logiquement *vehicule* et l'enregistre dans un tableau.

Code de ParcVehicule :

```
private static $parc ;
public static function enregistrer (...  $vehicule) {
  self ::$parc[] = $vehicule;
}
```

- La méthode *enregistrer* reçoit donc une référence nommée *$vehicule* (nous allons débattre sur son type dans quelques instants) et mémorise dans le tableau *$parc* la référence *$vehicule*. La méthode et la propriété sont statiques.

- Réfléchissons sur le type d'appartenance de ce paramètre *$vehicule*.

  La référence *$vehicule*, si elle représente un véhicule, ne va pas recevoir une instance de la classe *Vehicule* (puisque celle-ci est abstraite) mais une instance de *Citadine*, *Familiale* ou *Utilitaire*, mais comment choisir le type à déclarer ?

  Heureusement pour nous, les langages savent faire de la conversion automatique appelée « transtypage » ou « cast » en anglais. C'est à dire qu'un objet peut être transtypé dans le type d'une des classes dont il hérite.

  Nos trois classes héritant de Vehicule, PHP sait transtyper les objets de ces classes en Vehicule.

  Ce qui nous permet de définir le type du paramètre *$vehicule* dans *enregistrer* comme étant *Vehicule*.

```
public static function enregistrer (Vehicule  $vehicule) {
  self ::$parc[] = $vehicule;
}
```

Cette configuration fonctionne parfaitement : *$vehicule* peut recevoir n'importe quelle instance issue d'une classe concrète qui hérite de la classe abstraite ***Vehicule***.

- Nos trois classes doivent également pouvoir fournir certaines informations comme leurs identifiants ou des infos complètes (*getIdentifiant()* et *getInfosCompletes()*).

  Ces méthodes, avec un peu d'imagination, renvoient pour la première l'identifiant du véhicule et pour la seconde, l'ensemble de toutes les ***v.i.*** qui le caractérisent.

On peut envisager de définir ces méthodes spécifiquement dans les classes *Citadine*, *Familiale* et *Utilitaire* en imposant leur présence par :

```
public abstract function getIdentifiant() : string
```

et

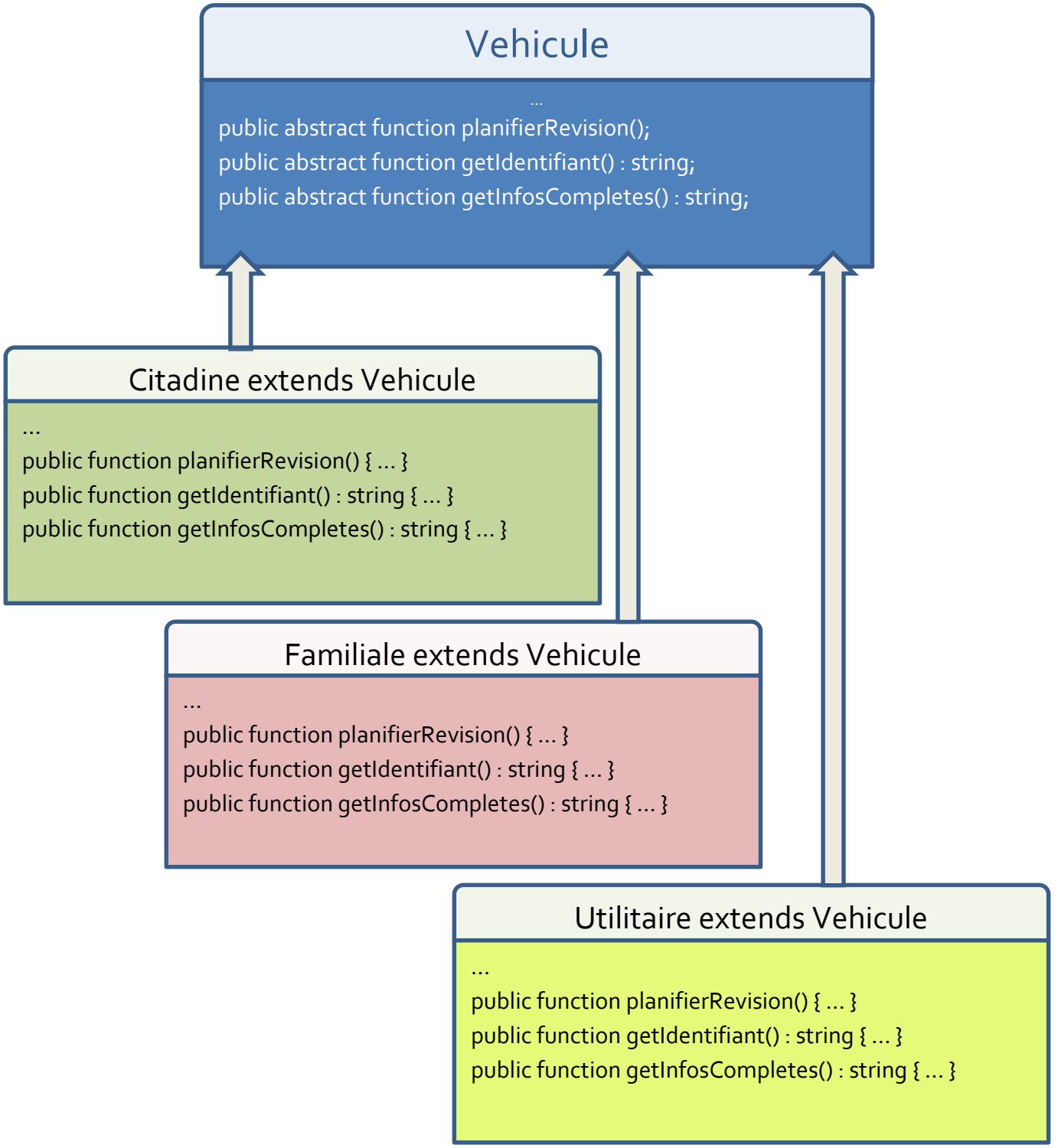
```
public abstract function getInfosCompletes() : string
```

dans *Vehicule* comme nous l'avons fait avec *planifierRevision()*.

```
class Vehicule {
    ....
    ....
    public abstract function planifierRevision();
    // Penser à ajouter ces 2 méthodes abstract pour obliger à les définir ...
    public abstract function getIdentifiant() : string;
    public abstract function getInfosCompletes() : string;
}
```

On est certain ainsi que les instances de *Citadine*, *Familiale* et *Utilitaire* répondront parfaitement aux méthodes *getIdentifiant()* et *getInfosCompletes()* puisque la présence de abstract dans *Vehicule* oblige le développeur à les définir dans les sous-classes.

Hiérarchie de l'étude précédente



Le loueur de véhicules souhaite maintenant que les garages où sont stockés les véhicules soient également répertoriés dans la base de données pour gérer ses locaux professionnels.

Ces garages seront représentés par une classe ***Garage*** dotée des ***v.i. surface*** pour la ***surface du garage***, ***capacite*** pour le ***nombre de véhicules stockables*** et ***niveaux*** pour le ***nombre d'étages***.

Les instances de cette classe ***Garage*** ne peuvent pas être transmises à la méthode *enregistrer(Vehicule vehicule)* car on ne peut pas raisonnablement hériter de la classe ***Vehicule*** pour générer la classe ***Garage*** !!!

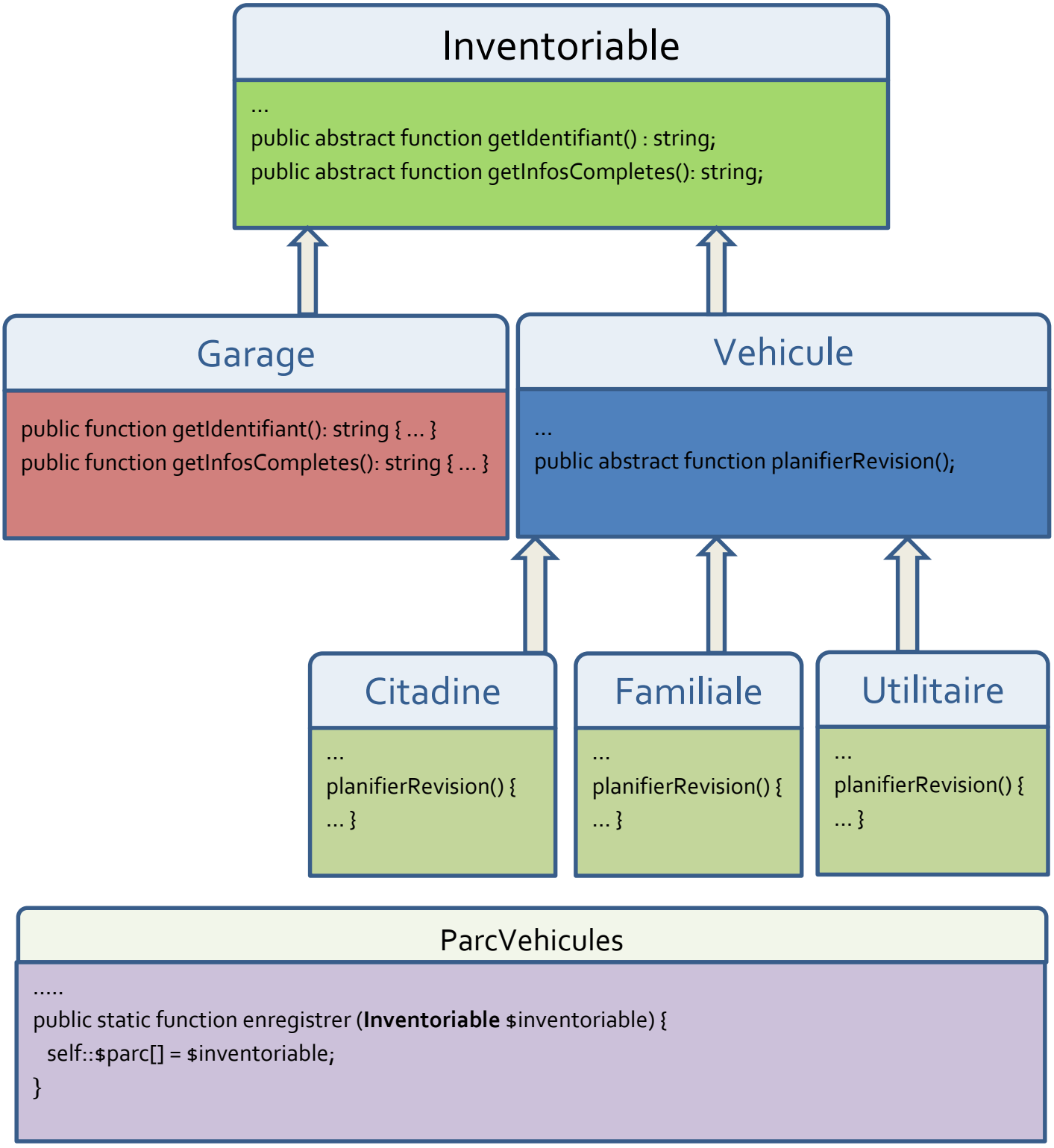
En d'autres termes : un garage n'est pas un véhicule. Or, la relation d'héritage traduit bien la notion « est-un ».

Ce serait un défaut conceptuel de faire hériter la classe ***Garage*** de ***Vehicule***.

On va donc générer une classe d'un niveau supérieur nommée ***Inventoriable*** dont vont hériter les classes ***Vehicule*** et ***Garage***.

Un garage et un véhicule (et ses classes dérivées bien sûr) sont bien, par héritage, des « ***Inventoriable*** » au sens objet.

Ce qui donne :


Inventoriable
...
public abstract function getIdentifiant() : string;
public abstract function getInfosCompletes(): string;
Garage
public function getIdentifiant(): string { ... }
public function getInfosCompletes(): string { ... }
Vehicule
...
public abstract function planifierRevision();
Citadine
...
planifierRevision() {
... }
Familiale
...
planifierRevision() {
... }
Utilitaire
...
planifierRevision() {
... }
ParcVehicules
.....
public static function enregistrer (Inventoriable $inventoriable) {
self::$parc[] = $inventoriable;
}

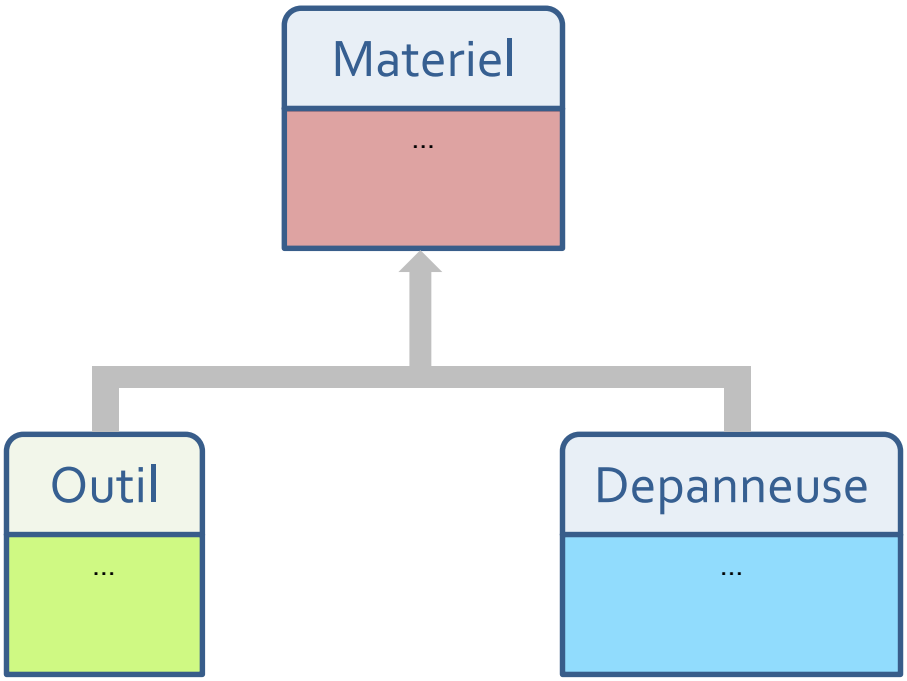
- 23 – Le code de la hiérarchie de classes

```
abstract class Inventoriable {
        public abstract function getIdentifiant() : string;
        public abstract function getInfosCompletes() : string;
}

abstract class Vehicule extends Inventoriable {
        // …
}

class Garage extends Inventoriable  {
        // …
}
```

Les classes ***Vehicule*** et ***Garage*** héritant maintenant de ***Inventoriable***, on peut changer le type (et le nom) du paramètre de la méthode *enregistrer* de ***ParcVehicules***.

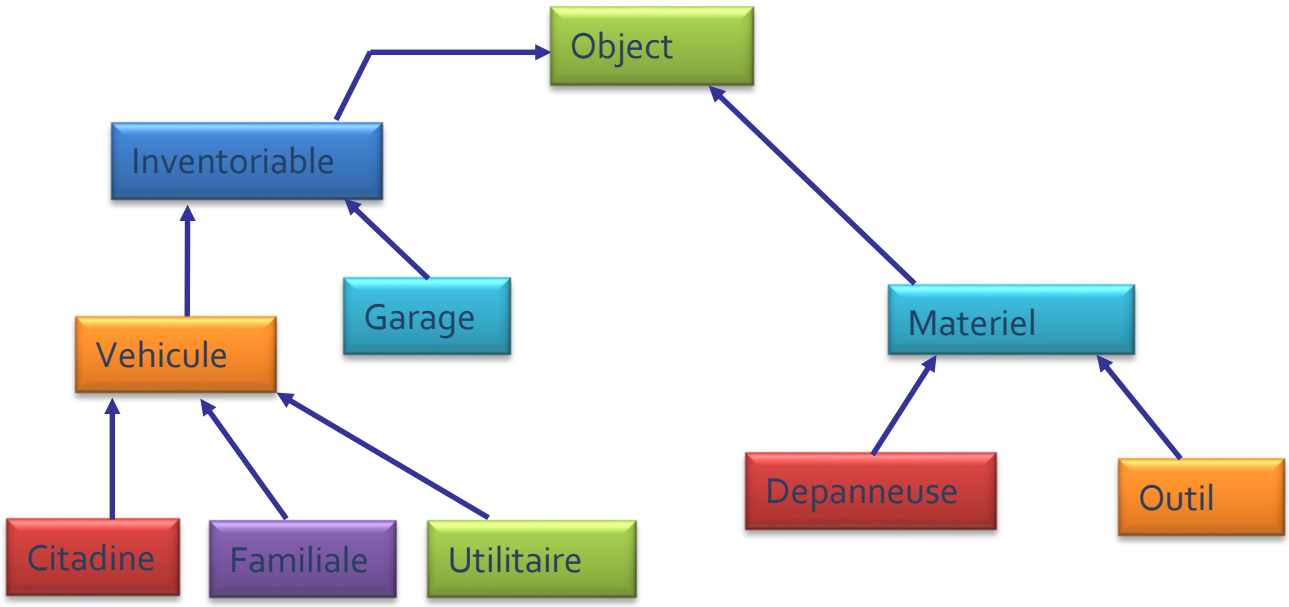
```
public static function enregistrer (Inventoriable  $inventoriable) {
…
}
```

✔ On veut aussi répertorier dans l'inventaire les dépanneuses servant à rapatrier les différents véhicules loués tombés en panne lors de leur utilisation par les clients. Les dépanneuses appartiennent à une hiérarchie déjà constituée selon le schéma suivant :



Materiel est la classe de base de ***Outil*** (ceux liés à la mécanique qui permettent la révision des véhicules) et de ***Depanneuse***.

On a, finalement, la hiérarchie suivante :



La hiérarchie complète de notre étude

Selon la hiérarchie précédente, si l'on veut pouvoir enregistrer les dépanneuses, il suffit de faire hériter la classe ***Depanneuse*** de la classe ***Inventoriable***.

Les instances de ***Depanneuse*** seraient ainsi des « ***Inventoriable*** » et toute instance de ***Depanneuse*** pourraient convenir pour la méthode *enregistrer* qui attend, rappelons-le, un paramètre de type ***Inventoriable***. Mais :

Depanneuse hérite déjà de la classe ***Materiel*** : elle ne peut, de fait, hériter en plus de ***Inventoriable*** : PHP interdit tout héritage multiple comme cela a été indiqué en début de ce cours.

Ceci met en évidence un nouveau concept en PHP : on va faire de ***Inventoriable*** non plus une classe abstract mais une **interface** dont on va implémenter les méthodes.

- 24 - L'interface *Inventoriable*

Définir une interface

On définit les comportements liés à l'enregistrement dans l'interface ***Inventoriable.***

On transforme la classe abstraite ***Inventoriable*** en **interface**.

✔ Une classe ne peut hériter de plusieurs classes mais elle peut implémenter une, voire plusieurs interfaces en complément de l'héritage.

✔ Une interface est **L'outil de spécification** par excellence en POO.

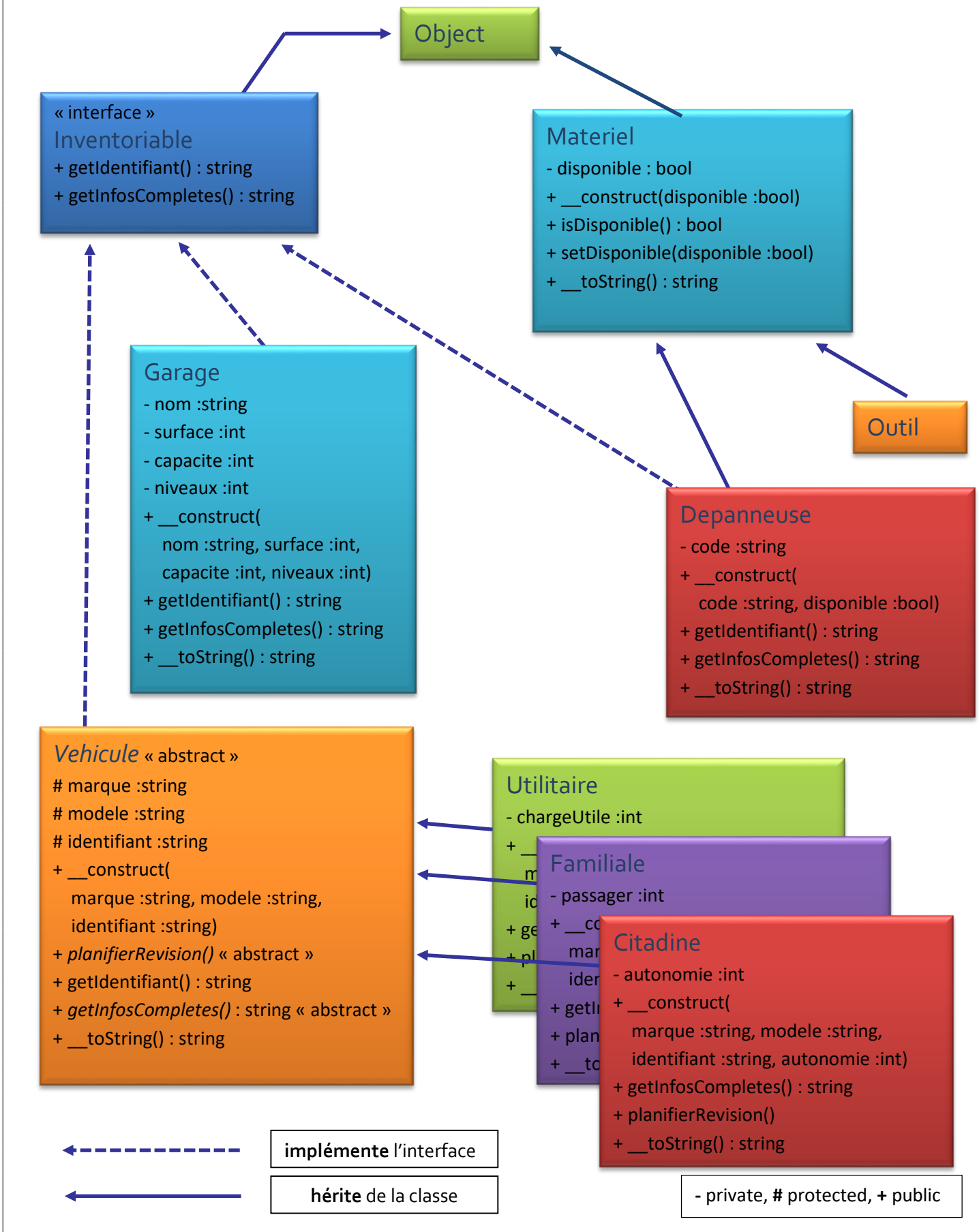
```
interface Inventoriable {
        public function getIdentifiant() : string;
        public function getInfosCompletes() : string;
} // interface Inventoriable

class Depanneuse extends Materiel implements Inventoriable {
         // .... etc
        public function getIdentifiant() : string
        {  return $identifiant ; }
        public function getInfosCompletes() :string
        { ... }
} // class Depanneuse
```

✔ Une **interface** est un outil de spécification dans lequel les méthodes sont implicitement abstract. Toute classe qui se déclare candidate à implémenter

une **interface** s'engage à respecter le contrat qui consiste à donner un sens aux méthodes et donc les définir.

✔Toute classe ne peut hériter que d'une seule classe mais peut implémenter autant d'interfaces qu'elle le souhaite.

Une nouvelle hiérarchie



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

Association nationale pour la Formation Professionnelle des Adultes
13 place du Général de Gaulle – 93108 Montreuil Cedex

www.afpa.fr