

La programmation Orientée Objet : concepts avancés

Table des matières

I. Contexte	3
II. L'héritage	3
III. Exercice : Appliquez la notion	5
IV. Le polymorphisme	6
V. Exercice : Appliquez la notion	11
VI. Les interfaces	12
VII. Exercice : Appliquez la notion	19
VIII. Les classes abstraites	21
IX. Exercice : Appliquez la notion	28
X. Auto-évaluation	29
A. Exercice final	29
B. Exercice : Défi	32
Solutions des exercices	33

I. Contexte

Durée : 1 h

Environnement de travail : Local

Pré-requis : Avoir installé et configuré un serveur de test local

Contexte

Les classes sont des outils très pratiques qui permettent notamment de rassembler dans une même structure des fonctionnalités et les données qui leur sont associées.

Pour autant, il ne s'agit que d'une petite partie de ce qu'il est possible de faire avec la Programmation Orientée Objet.

Dans ce chapitre, nous allons aller plus loin dans les mécanismes de la POO et aborder les fonctionnalités plus poussées de ce paradigme.

II. L'héritage

Objectifs

- Comprendre l'héritage
- Apprendre à manipuler des classes mères et des classes filles

Mise en situation

L'**héritage** est un mécanisme de la POO permettant de définir une hiérarchie entre nos classes.

Fondamental Le principe de l'héritage

Le principe est de créer une **classe mère** qui va contenir des propriétés et des méthodes de base. Vient ensuite la seconde étape : définir des **classes filles** qui étendent cette classe mère et qui, par conséquent, vont hériter de ses propriétés et méthodes.

Depuis une classe fille, il sera donc possible d'utiliser les méthodes d'une classe mère, mais aussi de modifier son comportement.

Le meilleur exemple d'héritage, c'est vous : vous avez hérité de certaines caractéristiques de vos parents, mais d'autres ont été modifiées pour faire de vous un être unique. On peut imaginer que vos yeux ont la même couleur que ceux de votre mère, mais que leur forme est complètement différente.

Exemple Une application sans héritage

Imaginons que nous ayons une classe permettant de gérer des utilisateurs d'un site e-commerce. Chaque utilisateur possède un nom et un prénom, ainsi qu'une méthode permettant de générer le nom à afficher.

```
1 <?php
2
3 class User
4 {
5     public string $name;
6     public string $surname;
7
8     public function __construct(string $name, string $surname)
```

```

9      {
10         $this->name = $name;
11         $this->surname = $surname;
12     }
13
14     public function getDisplayedName(): string
15     {
16         return $this->name . ' ' . $this->surname;
17     }
18 }

```

Certains utilisateurs de notre site peuvent être des vendeurs, c'est-à-dire des utilisateurs avec certaines informations en plus.

Par exemple, chaque vendeur est rattaché à une société, dont le nom est également stocké dans notre utilisateur. Il conviendrait alors de créer une deuxième classe pour représenter cela :

```

1 <?php
2
3 class Seller
4 {
5     public string $name;
6     public string $surname;
7     public string $company;
8
9     public function __construct(string $name, string $surname, string $company)
10    {
11        $this->name = $name;
12        $this->surname = $surname;
13        $this->company = $company;
14    }
15
16    public function getDisplayedName(): string
17    {
18        return $this->name . ' ' . $this->surname;
19    }
20 }

```

Notre classe `Seller` est donc un copié/collé de notre classe `User`, étant donné que notre `Seller` est un `User`, mais avec certaines informations en plus.

Ce code est fonctionnel, mais il n'est pas pratique à maintenir. Imaginons que l'ordre du nom et du prénom ne nous convienne pas et que l'on veuille l'inverser : nous allons devoir modifier le code à deux endroits différents.

Cela ne paraît pas forcément compliqué pour le moment, mais imaginez ce code dans une application possédant plusieurs centaines de classes, dont 10 types d'utilisateurs différents (vendeurs, acheteurs, administrateurs, modérateurs, gestionnaires de stocks...), tous dupliqués depuis la classe `User`.

Chaque évolution apportée aux utilisateurs devrait être répétée 10 fois, avec le risque d'oublier une classe.

L'héritage va nous permettre d'éviter de dupliquer du code en nous donnant la possibilité de dire qu'un `Seller` est un `User` plus élaboré, mais qui partage les mêmes fonctionnalités de base.

Méthode

Pour faire hériter une classe d'une autre, il faut utiliser le mot-clé `extends` suivi du nom de la classe à étendre au moment de la définition de la classe fille. Par défaut, la classe fille aura les mêmes attributs et méthodes que sa classe mère. Pour redéfinir une méthode dans la classe fille, il suffit de déclarer une méthode ayant le même nom.

```
1 <?php
2
3 require_once 'User.php';
4 class Seller extends User
5 {
6     public string $company;
7
8     public function __construct(string $name, string $surname, string $company)
9     {
10         $this->name = $name;
11         $this->surname = $surname;
12         $this->company = $company;
13     }
14 }
```

En faisant cela, notre `Seller` hérite de toutes les propriétés et méthodes de notre `User`. `User` est la **classe mère** et `Seller` est la **classe fille**. Ainsi, même si elle n'est pas explicitement visible dans le code de la classe `Seller`, il est possible d'appeler la méthode `getDisplayedName`.

```
1 <?php
2
3 require_once 'Seller.php';
4
5 $seller = new Seller('John', 'Doe', 'JD Incorporated');
6 echo $seller->getDisplayedName(); // Affiche "John Doe"
```

Attention

En PHP, **chaque classe ne peut hériter que d'une seule classe mère**, contrairement à d'autres langages qui autorisent l'héritage multiple, c'est-à-dire le fait qu'une classe fille puisse avoir plusieurs classes mères.

Syntaxe À retenir

- Pour faire hériter une classe d'une autre, il faut utiliser le mot-clé `extends` dans la **classe fille** en lui donnant le nom de la **classe mère**.
- Lorsqu'une classe hérite d'une autre, elle récupère toutes ses propriétés et ses méthodes.
- Une classe fille ne peut avoir **qu'une seule classe mère**.

Complément

L'héritage (documentation officielle)¹

III. Exercice : Appliquez la notion

Un concessionnaire automobile vend deux types de voiture : les voitures à essence et les voitures électriques.

- Pour les deux types de voitures, nous avons besoin de connaître leur marque et leur prix.
- Pour une voiture électrique, nous avons également besoin de connaître l'autonomie de sa batterie
- Pour une voiture à essence, on doit indiquer le taux de CO₂ rejeté par kilomètre parcouru.

On dispose de la classe `Car`.

¹ <https://www.php.net/manual/fr/language.oop5.inheritance.php>

```

1 <?php
2
3 // Fichier Car.php
4 class Car
5 {
6     public float $price;
7     public string $brand;
8
9     public function __construct(float $price, string $brand)
10    {
11        $this->price = $price;
12        $this->brand = $brand;
13    }
14 }

```

Pour réaliser cet exercice, vous pouvez travailler sur l'environnement de travail :



Question

[solution n°1 p.35]

En utilisant l'héritage, créez deux classes `ElectricCar` et `GasolineCar` permettant de structurer les données manipulées par ce concessionnaire et instanciez une voiture électrique et une voiture à essence avec les valeurs de votre choix.

IV. Le polymorphisme

Objectifs

- Comprendre le concept de polymorphisme
- Apprendre à surcharger des méthodes

Mise en situation

L'héritage est un outil très puissant permettant de centraliser une partie du code et d'éviter la duplication. Mais, plus que le simple abandon du copier/coller, c'est toute la manipulation de nos objets qui est modifiée grâce à ce principe. Voyons ensemble comment l'héritage va nous aider à simplifier notre code et le rendre plus compréhensible.

Méthode

Lorsqu'une classe hérite d'une autre classe, on se retrouve alors dans une situation où deux types de données différents possèdent des méthodes ayant le même nom.

Reprenons nos utilisateurs et nos vendeurs :

```

1 <?php
2
3 // Fichier User.php
4 class User
5 {
6     public string $name;
7     public string $surname;

```

1 <https://repl.it/>

```
8
9  public function __construct(string $name, string $surname)
10 {
11     $this->name = $name;
12     $this->surname = $surname;
13 }
14
15 public function getDisplayedName(): string
16 {
17     return $this->name.' '.$this->surname;
18 }
19 }
20
21 // Fichier Seller.php
22 <?php
23
24 require_once 'User.php';
25
26 class Seller extends User
27 {
28     public string $company;
29
30     public function __construct(string $name, string $surname, string $company)
31     {
32         $this->name = $name;
33         $this->surname = $surname;
34         $this->company = $company;
35     }
36 }
```

La méthode `getDisplayedName()` existe à la fois dans la classe `User`, où elle est déclarée, mais également dans la classe `Seller`, où elle est héritée.

Le fait d'avoir plusieurs méthodes ayant le même nom dans plusieurs classes s'appelle le **polymorphisme**.

Le polymorphisme a l'avantage de permettre un traitement similaire pour les objets d'une même hiérarchie de classe. Ainsi, que l'on instancie un `User` ou un `Seller`, nous savons que nous pouvons utiliser la méthode `getDisplayedName()`.

Complément

Le polymorphisme permet également d'aller plus loin : grâce à lui, il est possible de créer des fonctions ayant **une classe mère en signature**, mais de lui passer **une instance de classe fille** au moment de l'appel.

Cela permet de faire du code générique et de traiter, de la même manière, une instance de la classe mère ou n'importe quelle instance qui l'hérite.

Exemple

Il est d'usage, sur la majorité des sites web, d'afficher, dans un coin de l'écran, le nom de l'utilisateur connecté.

Créons donc une fonction permettant d'afficher la phrase "Connecté en tant que :" suivi du nom de l'utilisateur. Cette fonction va prendre en paramètre un `User` et utiliser la méthode `getDisplayedName()` précédemment définie :

```
1 <?php
2
3 // Fichier "displayFunctions.php"
4 require_once 'User.php';
5
6 function displayUserName(User $user): void
7 {
8     echo 'Connecté en tant que : '.$user->getDisplayedName();
9 }
10
```

Au moment d'appeler cette fonction, nous pouvons ainsi lui donner une instance de `User`, mais également une instance de `Seller`.

```
1 <?php
2
3 require_once 'Seller.php';
4 require_once 'displayFunctions.php';
5
6 $user = new User('John', 'Doe');
7 $seller = new Seller('Laure', 'Dupond', 'JD Incorporated');
8
9 displayUserName($user); // Affiche 'Connecté en tant que : John Doe'
10 displayUserName($seller); // Affiche 'Connecté en tant que : Laure Dupond'
```

Sans l'héritage, nous aurions dû créer une fonction `displayUserName` et une fonction `displaySellerName`, mais, grâce au polymorphisme, nous pouvons centraliser notre code. Ainsi, il est possible de donner à `displayUserName` **n'importe quelle classe fille de `User`**.

Tout ce qui compte pour la fonction `displayUserName` est qu'elle s'attend à recevoir au moins un `User`, et donc qu'elle ne va avoir accès qu'aux méthodes et propriétés d'un `User`.

Ainsi, on ne pourra pas utiliser la propriété `company` dans notre fonction qui s'attend à recevoir un `User`, étant donné que toutes les classes filles de `User` n'ont pas cette propriété.

Remarque Redéfinir les méthodes

Il reste une dernière utilité très importante au polymorphisme et à l'héritage : la **redéfinition des méthodes**. Dans une classe fille, il est possible de redéfinir le comportement d'une méthode mère.

Exemple

Dans notre exemple, imaginons que, lorsqu'on affiche le nom d'un vendeur, nous souhaitons rajouter le nom de son entreprise entre parenthèses.

Pour cela, il nous suffit de redéfinir la méthode `getDisplayedName` dans la classe `Seller` pour effectuer les changements.

```
1 <?php
2
3 require_once 'User.php';
4
5 class Seller extends User
6 {
```



```

7     public string $company;
8
9     public function __construct(string $name, string $surname, string $company)
10    {
11        $this->name = $name;
12        $this->surname = $surname;
13        $this->company = $company;
14    }
15
16    public function getDisplayedName(): string // Nous créons une nouvelle méthode
17    "getDisplayedName" qui va remplacer celle de la classe mère
18    {
19        return $this->name . ' ' . $this->surname . ' (' . $this->company . ')';
20    }

```

Ainsi, selon si notre objet est un `User` ou un `Seller`, c'est l'une ou l'autre des méthodes qui va être appelée.

```

1 <?php
2
3 $user = new User('John', 'Doe');
4 $seller = new Seller('Laure', 'Dupond', 'JD Incorporated');
5
6 echo $user->getDisplayedName(); // Affiche "John Doe" : la méthode appelée est celle de la
   classe User
7 echo $seller->getDisplayedName(); // Affiche "Laure Dupond (JD Incorporated)" : la méthode
   appelée est celle de la classe Seller

```

Fondamental L'intérêt de redéfinir ses méthodes

La redéfinition des méthodes semble annuler les bénéfices apportés par l'héritage. En effet, tout l'intérêt de déclarer une classe fille était de ne pas avoir à réécrire nos méthodes.

Reprenons notre idée d'inverser le nom et le prénom au moment de l'affichage : nous aurions de nouveau la problématique de devoir faire cette modification dans plusieurs fichiers.

Pour éviter cela, il est possible de faire directement appel à une méthode de la classe mère dans la redéfinition de la classe fille. Ainsi, une partie du code reste en commun et ne devra être modifiée qu'une seule fois.

Méthode

Pour appeler une méthode de la classe mère dans la classe fille, il suffit d'utiliser la syntaxe `parent::` suivie du nom de la méthode.

Exemple

Dans la classe `Seller`, il y a deux endroits où du code de la classe `User` est dupliqué : dans la méthode `getDisplayedName` (la concaténation du prénom et du nom) et dans le constructeur (l'affectation du nom et du prénom).

Grâce au mot-clé `parent`, nous pouvons réécrire ces méthodes en utilisant la méthode parente plutôt que réécrire les parties communes.

```

1 <?php
2
3 require_once 'User.php';
4
5 class Seller extends User
6 {
7     public string $company;

```

```

8
9     public function __construct(string $name, string $surname, string $company)
10    {
11        parent::__construct($name, $surname); // Nous appelons le constructeur de la méthode
12        mère. Comme tout appel de fonction, il faut lui passer les paramètres.
13        $this->company = $company;
14    }
15
16    public function getDisplayedName(): string
17    {
18        return parent::getDisplayedName().' ('.$this->company.')'; // Nous appelons la méthode
19        getDisplaydName de la méthode mère, qui retourne la concaténation du nom et du prénom, à
        laquelle nous rajoutons l'entreprise.
20    }
21 }

```

Remarque Le mot-clé "parent"

L'utilisation de `parent` n'est pas limitée à l'appel d'une méthode mère au sein de sa méthode fille.

Ce mot-clé permet plus généralement d'appeler n'importe quelle méthode mère depuis n'importe quelle méthode fille. Il serait par exemple possible d'appeler `parent::getDisplayedName()` dans le constructeur de `Seller`.

Redéfinir une méthode dans une classe fille permet d'utiliser toute la puissance du polymorphisme : notre fonction `displayUserName()`, bien qu'elle prenne un `User` en paramètre, appellerait correctement la méthode de la classe correspondante.

```

1 <?php
2
3 require_once 'Seller.php';
4 require_once 'displayFunctions.php';
5
6 $user = new User('John', 'Doe');
7 $seller = new Seller('Laure', 'Dupond', 'JD Incorporated');
8
9 displayUserName($user); // Affiche "John Doe"
10 displayUserName($seller); // Laure Dupond (JD Incorporated)

```

Ainsi, bien qu'il ne soit pas possible d'utiliser le champ `"company"` dans la fonction `displayUserName` (puisque cette dernière prend en paramètre un `User`), il n'y a aucun problème à l'utiliser dans la méthode de la classe fille.

Fondamental

Lorsque l'on donne un type dans une signature de fonction (que ce soit en paramètre ou en type de retour), il faut plutôt voir cela comme une indication des méthodes et propriétés minimales auxquelles nous avons accès lorsque l'on écrit notre fonction.

Notre fonction `displayUserName` prend en paramètre un `User` et utilise sa méthode `getDisplayedName`. Cela signifie qu'il est possible de lui passer **n'importe quel type héritant de `User`** et la fonction va automatiquement appeler la méthode `getDisplayedName` qui correspond : soit la version redéfinie, si elle existe, soit la version de base.

Le nom "polymorphisme" signifie "plusieurs formes". Cela désigne le fait qu'une même méthode (ici, `getDisplayedName`) peut avoir plusieurs implémentations dans plusieurs classes différentes, et donc de faire des choses différentes (avoir plusieurs "formes").

Syntaxe **À retenir**

- Pour modifier le comportement d'une méthode dans une classe fille, il suffit de définir une méthode ayant le même nom.
- Avoir le même nom de méthode dans plusieurs classes s'appelle le **polymorphisme** et cela permet de simplifier le code qui va utiliser nos classes.
- Si on veut faire appel à une méthode de la classe mère dans la classe fille, il faut utiliser la syntaxe `parent::nomDeLaMethode()`.

Complément

L'héritage (documentation officielle)¹

V. Exercice : Appliquez la notion

Soit le code d'une application de gestion de concession automobile :

```

1 <?php
2 // Fichier Car.php
3 class Car
4 {
5     public float $price;
6     public string $brand;
7
8     public function __construct(float $price, string $brand)
9     {
10         $this->price = $price;
11         $this->brand = $brand;
12     }
13 }
14
1 <?php
2 // Fichier ElectricCar.php
3 require_once 'Car.php';
4
5 class ElectricCar extends Car
6 {
7     public float $batteryAutonomy;
8
9     public function __construct(float $price, string $brand, float $batteryAutonomy)
10    {
11        parent::__construct($price, $brand);
12        $this->batteryAutonomy = $batteryAutonomy;
13    }
14 }
1
2 <?php
3 // Fichier GasolineCar.php
4 require_once 'Car.php';
5
6 class GasolineCar extends Car
7 {
8     public float $co2Emission;
9 }

```

¹ <https://www.php.net/manual/fr/language.oop5.inheritance.php>

```

9     public function __construct(float $price, string $brand, float $co2Emission)
10    {
11        parent::__construct($price, $brand);
12        $this->co2Emission = $co2Emission;
13    }
14 }

1 <?php
2 // Fichier index.php
3 require_once 'ElectricCar.php';
4 require_once 'GasolineCar.php';
5
6 $tesla = new ElectricCar(50000, 'Tesla', 560);
7 $renault = new GasolineCar(20000, 'Renault', 100);

```

Pour réaliser cet exercice, vous pouvez travailler sur l'environnement de travail :



Question 1

[solution n°2 p.35]

Lorsque nous allons afficher la fiche produite de l'une de nos voitures sur notre application, les différentes caractéristiques seront affichées dans une liste à puces avec leurs valeurs associées. Afin de gérer cela simplement du côté *front*, nous allons créer une fonction qui retourne un tableau associatif ayant pour clé le nom de la caractéristique et pour valeur la valeur de caractéristique.

Créez une fonction `getCharacteristics` retournant les caractéristiques de chaque type de voiture. Pour une voiture électrique, nous aurons besoin de son prix, sa marque et l'autonomie de sa batterie ; et pour une voiture à essence, son prix, sa marque et la quantité d'émission de CO₂.

Attention : nous pourrions rajouter des caractéristiques à une voiture dans des évolutions futures, donc évitez la duplication de code qui rendrait l'application difficilement maintenable.

Question 2

[solution n°3 p.37]

Écrivez maintenant une fonction `displayCharacteristics` prenant en paramètre une voiture, quelle qu'elle soit, et qui affiche la liste de ses caractéristiques dans une liste à puces.

- price : 50000
- brand : Tesla
- batteryAutonomy : 560

- price : 20000
- brand : Renault
- co2Emission : 100

VI. Les interfaces

1 <https://repl.it/>

Objectifs

- Créer des interfaces
- Implémenter des interfaces

Mise en situation

Le polymorphisme est un outil très puissant qui permet d'appeler la bonne méthode dans une hiérarchie de classe. En déclarant une fonction qui prend en paramètre un objet, il est ainsi possible de lui donner n'importe quel objet héritant de la classe attendue pour que la méthode correspondante soit automatiquement appelée.

Cependant, la limite à ce polymorphisme est que notre objet doit obligatoirement hériter de la classe que l'on attend en paramètre. Cela peut être contraignant dans certains cas où deux objets n'ont rien en commun, mais que l'on souhaite quand même écrire du code générique qui servirait dans plusieurs cas. C'est pourquoi il existe un autre moyen d'utiliser le polymorphisme : les **interfaces**.

Dans un cas réel

Imaginons que nous devons gérer un site e-commerce où des utilisateurs, qui travaillent pour des entreprises, vendent des produits. D'autres utilisateurs, les acheteurs, peuvent acheter les produits. Créons les classes permettant de gérer ces éléments :

```
1 <?php
2
3 // Fichier Company.php
4 class Company
5 {
6     public string $name;
7     public string $type;
8     public string $domain;
9
10    public function __construct(string $name, string $type, string $domain)
11    {
12        $this->name = $name;
13        $this->type = $type;
14        $this->domain = $domain;
15    }
16 }
17
18 // Fichier Product.php
19 class Product
20 {
21     public string $title;
22     public string $description;
23
24     public function __construct(string $title, string $description)
25     {
26         $this->title = $title;
27         $this->description = $description;
28     }
29 }
30
31 // Fichier User.php
32 class User
33 {
34     public string $firstName;
35     public string $lastName;
36 }
```

```

37     public function __construct(string $firstName, string $lastName)
38     {
39         $this->firstName = $firstName;
40         $this->lastName = $lastName;
41     }
42 }
43
44 // Fichier Seller.php
45 require_once 'User.php';
46
47 class Seller extends User
48 {
49     public Company $company;
50
51     public function __construct(string $name, string $surname, Company $company)
52     {
53         parent::__construct($name, $surname);
54         $this->company = $company;
55     }
56 }
57
58 // Fichier index.php
59 require_once 'Seller.php';
60 require_once 'Company.php';
61 require_once 'Product.php';
62
63 $user = new User('John', 'Doe');
64 $company = new Company('JD', 'SARL', 'composants informatiques');
65 $seller = new Seller('Laure', 'Dupond', $company);
66 $product = new Product('Carte mère', 'Socket LGA1151');

```

S'il est logique que la classe `Seller` hérite de la classe `User`, les autres classes n'ont aucune propriété en commun, donc aucun rapport de hiérarchie entre elles.

Cependant, quelque chose qui serait utile sur notre application serait d'ajouter une infobulle, composée d'un titre et d'un texte court, au survol d'un de ces éléments de la souris.

Par exemple, si on est sur la page d'un article, on pourrait survoler le nom du vendeur ou le nom de l'entreprise pour avoir plus d'informations.

Si l'on voulait implémenter cette fonctionnalité en l'état, il nous faudrait créer trois fonctions :

- `displayUserTooltip(User $user)` pour afficher l'infobulle des `User` et des `Seller`,
- `displayProductTooltip(Product $product)` pour afficher celle des produits,
- `displayCompanyTooltip(Company $company)` pour afficher celle des entreprises.

Chacune de ces fonctions aurait pour but d'extraire de leurs paramètres un titre et une description à afficher dans notre infobulle.

Il serait également possible de créer une classe vide, que les autres classes pourraient hériter, mais l'héritage multiple n'existant pas en PHP, on se retrouverait bloqué pour une autre fonctionnalité.

Pour résoudre ce problème simplement, il est possible d'utiliser les **interfaces**.

Méthode Les interfaces

Une **interface** désigne une liste de méthodes qui devront être présentes dans toutes les classes qui souhaitent implémenter l'interface. Ainsi, une interface permet de forcer un développeur à créer des méthodes ayant un nom précis lorsqu'il crée une classe. L'intérêt des interfaces est qu'elles **peuvent être utilisées dans les signatures de fonctions**, ce qui permet de faire du polymorphisme sans devoir nécessairement utiliser l'héritage.

Une interface se déclare via le mot-clé `interface` suivi du nom de l'interface. La liste des méthodes à implémenter s'écrit de la même manière que les méthodes d'une classe, à la différence qu'elles ne doivent pas avoir de corps : au lieu des accolades, elles doivent se terminer par un point-virgule.

Pour qu'une classe implémente une interface, il faut ajouter le mot-clé `implements` suivi du nom de l'interface lors de la définition de la classe.

Exemple Mettre en place une interface

Afin de pouvoir gérer nos infobulles, créons une interface `Tooltipable` (comprendre "qui peut être tooltipé") possédant deux méthodes : `getTitle()`, qui retournera le titre de l'infobulle, et `getDescription()`, qui retournera la description à afficher dans l'infobulle.

Toutes les classes qui implémenteront cette interface devront implémenter leur propre version de ces deux méthodes, ce qui leur permettra de pouvoir être affichées dans une infobulle.

```
1 <?php
2
3 // Fichier Tooltipable.php
4
5 interface Tooltipable
6 {
7     public function getTitle(): string;
8     public function getDescription(): string;
9 }
```

Cela nous permet de créer une fonction `displayTooltip` qui prend en paramètre un `Tooltipable`.

Nous n'avons aucune information sur l'objet qui va être passé à la fonction, mais nous savons qu'il possède les deux méthodes `getTitle()` et `getDescription()`, ce qui nous suffit pour afficher l'infobulle.

```
1 <?php
2
3 // Fichier tooltip.php
4
5 function displayTooltip(Tooltipable $tooltipable)
6 {
7     echo '<h3>'.$tooltipable->getTitle().'</h3><p>'.$tooltipable->getDescription().'</p>';
8 }
```

Implémentons maintenant cette interface dans notre classe `Company` :

```
1 <?php
2
3 // Fichier Company.php
4
5 require_once 'Tooltipable.php';
6
7 class Company implements Tooltipable
8 {
9     public string $name;
10    public string $type;
11    public string $domain;
12
13    public function __construct(string $name, string $type, string $domain)
14    {
15        $this->name = $name;
16        $this->type = $type;
17        $this->domain = $domain;
18    }
```

```

19
20     public function getTitle(): string
21     {
22         return $this->name;
23     }
24
25     public function getDescription(): string
26     {
27         return $this->name.' est une '.$this->type.' spécialisée dans '.$this->domain;
28     }
29 }

```

Maintenant, nous pouvons passer une `Company` en tant que paramètre de notre fonction `displayTooltip` :

```

1 <?php
2
3 // Fichier index.php
4
5 require_once 'tooltip.php';
6 require_once 'Company.php';
7 $company = new Company('JD', 'SARL', 'les composants informatiques');
8 displayTooltip($company);

```

Faisons la même chose pour nos classes `User` et `Product`.

À noter qu'il n'est pas nécessaire d'implémenter les méthodes dans `Seller` si sa classe mère (`User`) l'implémente déjà. En revanche, il serait possible de redéfinir les méthodes dans `Seller`.

```

1 <?php
2
3 // Fichier Product.php
4
5 require_once 'Tooltipable.php';
6
7 class Product implements Tooltipable
8 {
9     public string $title;
10    public string $description;
11
12    public function __construct(string $title, string $description)
13    {
14        $this->title = $title;
15        $this->description = $description;
16    }
17
18    public function getTitle(): string
19    {
20        return $this->title;
21    }
22
23    public function getDescription(): string
24    {
25        return $this->description;
26    }
27 }
28
29
30 // Fichier User.php
31 require_once 'Tooltipable.php';
32

```



```
33 class User implements Tooltipable
34 {
35     public string $firstName;
36     public string $lastName;
37
38     public function __construct(string $firstName, string $lastName)
39     {
40         $this->firstName = $firstName;
41         $this->lastName = $lastName;
42     }
43
44     public function getTitle(): string
45     {
46         return $this->firstName.' '.$this->lastName;
47     }
48
49     public function getDescription(): string
50     {
51         return $this->getTitle().' est un utilisateur de notre site';
52     }
53 }
54
55 // Fichier index.php
56
57 require_once 'Seller.php';
58 require_once 'Product.php';
59 require_once 'tooltip.php';
60 require_once 'Company.php';
61
62 $company = new Company('JD', 'SARL', 'les composants informatiques');
63 $user = new User('John', 'Doe');
64 $seller = new Seller('Laure', 'Dupond', $company);
65 $product = new Product('Carte mère', 'Socket LGA1151');
66
67 displayTooltip($company);
68 displayTooltip($user);
69 displayTooltip($seller);
70 displayTooltip($product);
```

JD

JD est une SARL spécialisée dans les composants informatiques

John Doe

John Doe est un utilisateur de notre site

Laure Dupond

Laure Dupond est un utilisateur de notre site

Carte mère

Socket LGA1151

Syntaxe À retenir

- Les **interfaces** permettent de définir des "contrats" que les développeurs devront suivre lors de la création de leurs classes.
- Elles définissent un ensemble de méthodes à créer dans les classes qui implémentent l'interface et permettent d'utiliser le polymorphisme avec des classes qui n'ont aucun lien entre elles.

```
new User( username: 'John', email: 'john@doe.fr');
```

Attention, dans la vidéo, ci-dessus, le code affiché dans le logiciel PHPStorm contient des infobulles d'aide (sur fond gris clair) : ce code n'est pas à recopier si vous souhaitez reprendre l'exercice. Dans l'exemple ci-contre, vous ne recopierez ni "username :" ni "email :" dans votre éditeur de code favori.

Attention, dans la vidéo, ci-dessus, le code affiché dans le logiciel PHPStorm contient des infobulles d'aide (sur fond gris clair) : ce code n'est pas à recopier si vous souhaitez reprendre l'exercice. Dans l'exemple ci-contre, vous ne recopierez ni "username :" ni "email :" dans votre éditeur de code favori.

```
new User( username: 'John', email: 'john@doe.fr');
```

Complément

Les interface¹s (documentation officielle)²

¹ <https://www.php.net/manual/fr/language.oop5.interfaces.php>

² <https://www.php.net/manual/fr/language.oop5.inheritance.php>

VII. Exercice : Appliquez la notion

En plus des voitures électriques et à essence, notre concessionnaire vend également des pneus. Un pneu possède certaines caractéristiques, comme sa largeur, sa hauteur et son diamètre.

Sur la fiche produit de nos pneus, ces caractéristiques vont être affichées de la même manière que les caractéristiques de nos voitures, c'est-à-dire dans une liste à puces. Le but de cet exercice est d'adapter le code pour rendre cela possible.

Pour réaliser cet exercice, vous pouvez travailler sur l'environnement de travail :



Question 1

[solution n°4 p.38]

Créez une classe `Tire` permettant de gérer les pneus.

Question 2

[solution n°5 p.38]

Le code de notre application est le suivant :

```
1 <?php
2
3 // Fichier Car.php
4 class Car
5 {
6     public float $price;
7     public string $brand;
8
9     public function __construct(float $price, string $brand)
10    {
11        $this->price = $price;
12        $this->brand = $brand;
13    }
14
15    public function getCharacteristics(): array
16    {
17        return [
18            'price' => $this->price,
19            'brand' => $this->brand,
20        ];
21    }
22 }
23
24 // Fichier ElectricCar.php
25 require_once 'Car.php';
26
27 class ElectricCar extends Car
28 {
29     public float $batteryAutonomy;
30
31     public function __construct(float $price, string $brand, float $batteryAutonomy)
32     {
33         parent::__construct($price, $brand);
34         $this->batteryAutonomy = $batteryAutonomy;
35     }
36 }
```

1 <https://repl.it/>

```

36
37     public function getCharacteristics(): array
38     {
39         $characteristics = parent::getCharacteristics();
40         $characteristics['batteryAutonomy'] = $this->batteryAutonomy;
41
42         return $characteristics;
43     }
44 }
45
46 // Fichier GasolineCar.php
47 require_once 'Car.php';
48
49 class GasolineCar extends Car
50 {
51     public float $co2Emission;
52
53     public function __construct(float $price, string $brand, float $co2Emission)
54     {
55         parent::__construct($price, $brand);
56         $this->co2Emission = $co2Emission;
57     }
58
59     public function getCharacteristics(): array
60     {
61         $characteristics = parent::getCharacteristics();
62         $characteristics['co2Emission'] = $this->co2Emission;
63
64         return $characteristics;
65     }
66 }
67
68 // Fichier carFunctions.php
69 require_once 'Car.php';
70
71 function displayCharacteristics(Car $car): void
72 {
73     echo '<ul>';
74     foreach ($car->getCharacteristics() as $name => $value) {
75         echo '<li>'.$name.' : '.$value.'</li>';
76     }
77     echo '</ul>';
78 }
79
80 // Fichier index.php
81 require_once 'ElectricCar.php';
82 require_once 'GasolineCar.php';
83 require_once 'carFunctions.php';
84
85 $tesla = new ElectricCar(50000, 'Tesla', 560);
86 $renault = new GasolineCar(20000, 'Renault', 100);
87
88 displayCharacteristics($tesla);
89 displayCharacteristics($renault);

```

Créez une interface `CharacteristicsDisplayable` qui définit les méthodes nécessaires à l'affichage des caractéristiques sous forme de listes à puces.

Ainsi, la fonction `displayCharacteristics` ne prendra plus en paramètre un objet de type `Car`, mais un objet implémentant `CharacteristicsDisplayable`.

Effectuez ensuite les modifications nécessaires pour que nos voitures et nos pneus puissent être affichés grâce à la fonction `displayCharacteristics`.

VIII. Les classes abstraites

Objectifs

- Comprendre le concept de classe abstraite
- Savoir quand utiliser les classes abstraites

Mise en situation

Lorsque l'on crée l'architecture de notre application, on peut parfois se retrouver avec des classes qui ne servent qu'à poser les bases du code qui sera utilisé par les classes filles. Ces classes n'ont pas vocation à être instanciées, mais simplement à centraliser des propriétés et des méthodes.

Il est possible de définir un tel comportement grâce aux **classes abstraites**.

Dans un cas réel

Un site e-commerce peut vendre plusieurs produits : des livres, des disques durs... Chaque type possède des caractéristiques qui lui sont propres : les livres ont des auteurs et un nombre de pages, tandis qu'un disque dur aura une capacité et une marque. Pour gérer chaque cas dans notre code, nous allons donc créer une classe par produit.

Cependant, tous ces produits auront également des propriétés communes : un prix, un titre, un texte de description...

Nous pourrions créer une classe `Product` qui contiendra ces informations et dont héritera chaque classe représentant un type de produit.

```
1 <?php
2
3 // Fichier Product.php
4 class Product
5 {
6     public float $price;
7     public string $name;
8
9     public function __construct(float $price, string $name)
10    {
11        $this->price = $price;
12        $this->name = $name;
13    }
14 }
15
16 // Fichier Book.php
17 require_once 'Product.php';
18
19 class Book extends Product
20 {
21     public string $author;
22     public int $pagesNumber;
23
24     public function __construct(float $price, string $title, string $author, int $pagesNumber)
25    {
```

```

26         // Le titre du livre correspond à la propriété "name" de notre produit
27         parent::__construct($price, $title);
28         $this->author = $author;
29         $this->pagesNumber = $pagesNumber;
30     }
31 }
32
33 // Fichier HardDrive.php
34 require_once 'Product.php';
35
36 class HardDrive extends Product
37 {
38     public int $capacity;
39     public string $brand;
40
41     public function __construct(float $price, string $name, int $capacity, string $brand)
42     {
43         parent::__construct($price, $name);
44         $this->capacity = $capacity;
45         $this->brand = $brand;
46     }
47 }

```

Étant donné que, quel que soit le type de produit, nous allons avoir une classe le représentant, la classe `Product` n'est jamais destinée à être instanciée directement.

Le type en lui-même pourra être utilisé dans des signatures de fonction, mais ce sont ces classes filles qui seront instanciées.

C'est ce qu'on appelle une classe **abstraite**.

Méthode Créer une classe abstraite

Pour créer une classe abstraite, il suffit d'ajouter le mot-clé `abstract` devant la définition de la classe. Le reste de la classe peut être écrit comme n'importe quelle autre classe.

```

1 <?php
2
3 abstract class MyClass {
4     // Méthodes et propriétés
5 }

```

Déclarer une classe comme abstraite interdit son instantiation : essayer de créer un objet à partir d'une classe abstraite soulèvera une erreur !

```

1 <?php
2
3 require_once 'MyClass.php';
4
5 $object = new MyClass(); // Souleve une erreur : MyClass est une classe abstraite !

```

Exemple

Transformons notre classe `Product` en classe abstraite :

```

1 <?php
2
3 // Fichier Product.php
4 abstract class Product // Nous déclarons la classe comme abstraite
5 {
6     public float $price;

```

```

7     public string $name;
8
9     public function __construct(float $price, string $name)
10    {
11        $this->price = $price;
12        $this->name = $name;
13    }
14 }

```

Ainsi, nous pourrons instancier des livres et des disques durs, mais pas des "produits".

Fondamental Les méthodes abstraites

Dans une classe abstraite, il est possible de créer des **méthodes abstraites**, c'est-à-dire des méthodes qui n'auront pas de corps et qui devront être implémentées dans les classes filles.

L'intérêt est le même que pour les interfaces : les méthodes déclarées comme abstraites pourront être utilisées dans les fonctions ayant pour signature notre classe abstraite.

On peut ainsi voir les classes abstraites comme des "interfaces évoluées" qui contiennent la base de la logique que devront respecter nos classes filles.

Méthode Créer une méthode abstraite

Pour créer une méthode abstraite, il suffit d'ajouter le mot-clé `abstract` devant la définition de la méthode. Comme pour les interfaces, les méthodes abstraites n'auront pas de corps et devront ainsi se terminer par des points-virgules.

```

1 <?php
2
3 abstract class MyClass {
4     abstract function myMethod(): string;
5 }

```

Exemple

Quels que soient nos produits, nous allons devoir afficher leur prix total. Cependant, la méthode de calcul de chaque article sera différente : les livres ont une TVA à 5,5 %, tandis que les disques durs ont une TVA à 20 %.

Nous pouvons donc déclarer une méthode abstraite `getTotalPrice()` dans notre classe `Product` pour nous assurer que la méthode de calcul sera bien définie dans les classes filles.

```

1 <?php
2
3 // Fichier Product.php
4 abstract class Product
5 {
6     public float $price;
7     public string $name;
8
9     public function __construct(float $price, string $name)
10    {
11        $this->price = $price;
12        $this->name = $name;
13    }
14
15     public abstract function getTotalPrice(): float;
16 }
17

```

```

18 // Fichier Book.php
19 require_once 'Product.php';
20
21 class Book extends Product
22 {
23     private const TAX_ON_BOOKS = 5/100;
24     public string $author;
25     public int $pagesNumber;
26
27     public function __construct(float $price, string $title, string $author, int $pagesNumber)
28     {
29         // Le titre du livre correspond à la propriété "name" de notre produit
30         parent::__construct($price, $title);
31         $this->author = $author;
32         $this->pagesNumber = $pagesNumber;
33     }
34
35     public function getTotalPrice(): float
36     {
37
38         return $this->price * (1 + TAX_ON_BOOKS);
39     }
40 }
41
42 // Fichier HardDrive.php
43 require_once 'Product.php';
44
45 class HardDrive extends Product
46 {
47     private const TAX_ON_HARD_DRIVES = 20/100;
48     public int $capacity;
49     public string $brand;
50
51     public function __construct(float $price, string $name, int $capacity, string $brand)
52     {
53         parent::__construct($price, $name);
54         $this->capacity = $capacity;
55         $this->brand = $brand;
56     }
57
58     public function getTotalPrice(): float
59     {
60         return $this->price * (1 + self::TAX_ON_HARD_DRIVES);
61     }
62 }

```

Fondamental Classes abstraites et interfaces

Tout comme les classes classiques, les classes abstraites peuvent implémenter des interfaces.

Cependant, étant donné que les classes abstraites ne peuvent pas être instanciées, il est possible de laisser l'implémentation des méthodes de l'interface aux classes filles.

D'une manière générale, la structure de notre code sera la suivante :

- les **interfaces** vont **définir les comportements** possibles pour nos objets,
- les **classes abstraites** vont **définir la logique** de base permettant de réaliser ces comportements,
- les **classes filles**, qui vont **implémenter cette logique**.

C'est la raison pour laquelle il n'est pas possible de définir des propriétés dans les interfaces : elles ne décrivent que des comportements, elles n'ont aucune idée de l'implémentation qu'il y a derrière.

Exemple

Reprenons notre exemple d'interface `Tooltipable` permettant de définir les méthodes que doit avoir un objet pour apparaître dans une infobulle.

```
1 <?php
2
3 // Fichier Tooltipable.php
4
5 interface Tooltipable
6 {
7     public function getTitle(): string;
8     public function getDescription(): string;
9 }
```

Cette interface n'a aucune idée de la logique qu'il doit y avoir derrière les méthodes `getTitle` et `getDescription` : elle déclare juste que tous les objets souhaitant apparaître dans une infobulle doivent implémenter ces fonctions.

Ainsi, la fonction d'affichage de l'infobulle sait qu'elle peut les utiliser.

```
1 <?php
2
3 // Fichier tooltip.php
4
5 function displayTooltip(Tooltipable $tooltipable)
6 {
7     echo '<h3>'.$tooltipable->getTitle().'</h3><p>'.$tooltipable->getDescription().'</p>';
8 }
```

Notre classe abstraite `Product` va implémenter `Tooltipable`, car nous voulons que tous nos produits puissent apparaître dans une infobulle.

Elle peut ne pas implémenter ces méthodes, car chaque produit va intégrer les informations qui lui sont propres en titre et en description.

```
1 <?php
2
3 // Fichier Product.php
4 require_once 'Tooltipable.php';
5 abstract class Product implements Tooltipable
6 {
7     // Le reste du fichier est similaire : nous n'ajoutons pas les méthodes dans la classe
    abstraite
8 }
```

Puis, chaque produit va implémenter, à sa manière, les méthodes de l'interface.

```
1 <?php
2
3 // Fichier Book.php
4 require_once 'Product.php';
5
```

```

6 class Book extends Product
7 {
8     public string $author;
9     public int $pagesNumber;
10
11     public function __construct(float $price, string $title, string $author, int $pagesNumber)
12     {
13         // Le titre du livre correspond à la propriété "name" de notre produit
14         parent::__construct($price, $title);
15         $this->author = $author;
16         $this->pagesNumber = $pagesNumber;
17     }
18
19     public function getTotalPrice(): float
20     {
21         return $this->price * 1.2;
22     }
23
24     // Implémentation de l'interface :
25     public function getTitle(): string
26     {
27         return $this->name.', de '.$this->author;
28     }
29
30     public function getDescription(): string
31     {
32         $description = $this->name.' est un livre de '.$this->author.'.';
33         if ($this->pagesNumber < 100) {
34             $description .= 'Avec ses '.$this->pagesNumber.' pages, ce livre est idéal pour
35 les lecteurs occasionnels';
36         } else {
37             $description .= 'Seuls les lecteurs acharnés viendront à bout de ses '.$this->
38 >pagesNumber.' pages.';
39         }
40
41         return $description;
42     }
43 }
44 // Fichier HardDrive.php
45 require_once 'Product.php';
46
47 class HardDrive extends Product
48 {
49     public int $capacity;
50     public string $brand;
51
52     public function __construct(float $price, string $name, int $capacity, string $brand)
53     {
54         parent::__construct($price, $name);
55         $this->capacity = $capacity;
56         $this->brand = $brand;
57     }
58
59     public function getTotalPrice(): float
60     {
61         return $this->price * 1.055;

```

```

62     }
63
64     // Implémentation de l'interface :
65     public function getTitle(): string
66     {
67         return $this->name.' - '.$this->capacity.' Go';
68     }
69
70     public function getDescription(): string
71     {
72         return 'Disque dur '.$this->name.' de la marque '.$this->brand.' ayant une capacité de
73         stockage de '.$this->capacity.' Go';
74     }

```

Ainsi, il est possible d'afficher nos produits dans des infobulles :

```

1 <?php
2
3 // Fichier index.php
4
5 require_once 'tooltip.php';
6 require_once 'Book.php';
7 require_once 'HardDrive.php';
8
9 $mobyDick = new Book(19.99, 'Moby Dick', 'Herman Melville', 752);
10 $hardDrive = new HardDrive(49.90, 'BarraCuda', 1000, 'Seagate');
11 displayTooltip($mobyDick);
12 displayTooltip($hardDrive);

```

Syntaxe À retenir

- Les **classes abstraites** permettent de définir des classes qui ne peuvent pas être instanciées directement.
- Elles peuvent comporter des **méthodes abstraites** qui devront être définies dans les classes filles.
- Une classe abstraite peut implémenter une interface, ce qui permet de structurer son code de manière claire et séparer la partie "comportement" (interfaces) de la partie "logique" (classe abstraite) et de l'implémentation de cette logique (classes filles).

Complément

Les interface¹s (documentation officielle)²

¹ <https://www.php.net/manual/fr/language.oop5.interfaces.php>

² <https://www.php.net/manual/fr/language.oop5.inheritance.php>

IX. Exercice : Appliquez la notion

Utilisons les classes abstraites pour structurer plus clairement notre application de concession automobile.

Pour réaliser cet exercice, vous pouvez travailler sur l'environnement de travail :



Question 1

[solution n°6 p.40]

Reprenons le code de notre application de concession des exercices précédents :

```

1 <?php
2
3 // Fichier CharacteristicsDisplayable.php
4 interface CharacteristicsDisplayable
5 {
6     public function getCharacteristics(): array;
7 }
8
9 // Fichier Car.php
10 require_once 'CharacteristicsDisplayable.php';
11
12 class Car implements CharacteristicsDisplayable
13 {
14     public float $price;
15     public string $brand;
16
17     public function __construct(float $price, string $brand)
18     {
19         $this->price = $price;
20         $this->brand = $brand;
21     }
22
23     public function getCharacteristics(): array
24     {
25         return [
26             'price' => $this->price,
27             'brand' => $this->brand,
28         ];
29     }
30 }
31
32 // Fichier ElectricCar.php
33 require_once 'Car.php';
34
35 class ElectricCar extends Car
36 {
37     public float $batteryAutonomy;
38
39     public function __construct(float $price, string $brand, float $batteryAutonomy)
40     {
41         parent::__construct($price, $brand);
42         $this->batteryAutonomy = $batteryAutonomy;
43     }

```

1 <https://repl.it/>

```

44
45     public function getCharacteristics(): array
46     {
47         $characteristics = parent::getCharacteristics();
48         $characteristics['batteryAutonomy'] = $this->batteryAutonomy;
49
50         return $characteristics;
51     }
52 }
53
54 // Fichier GasolineCar.php
55 require_once 'Car.php';
56
57 class GasolineCar extends Car
58 {
59     public float $co2Emission;
60
61     public function __construct(float $price, string $brand, float $co2Emission)
62     {
63         parent::__construct($price, $brand);
64         $this->co2Emission = $co2Emission;
65     }
66
67     public function getCharacteristics(): array
68     {
69         $characteristics = parent::getCharacteristics();
70         $characteristics['co2Emission'] = $this->co2Emission;
71
72         return $characteristics;
73     }
74 }

```

Dans ce code, identifiez quelle classe n'a pas de sens à être instanciée et rendez-la abstraite.

Question 2

[solution n°7 p.40]

Nous allons maintenant calculer le prix final de nos voitures.

Pour les voitures électriques, une prime de conversion de 2500 € est déduite du prix original afin d'encourager les gens à passer à l'électrique. En revanche, pour les voitures à essence, un malus de 50 € par taux d'émission au-delà de 119 g/km sera ajouté au prix original.

Créez une méthode `getFinalPrice()` permettant de calculer le prix final de toutes nos voitures. Créez également une fonction `displayPrice()` qui prend en paramètre un objet de type `Car` et qui affiche son prix final, suivi de son prix original entre parenthèses.

X. Auto-évaluation

A. Exercice final

Exercice 1

[solution n°8 p.41]

Exercice

Qu'est-ce que l'héritage ?

- ☐ Le fait d'importer des fonctions stockées dans un autre fichier
- ☐ Une manière de transmettre des données d'une classe lorsque son destructeur est appelé
- ☐ Une manière de partager des propriétés et des méthodes d'une classe avec une autre

Exercice

Qu'est-ce que le polymorphisme ?

- ☐ Le fait qu'une classe puisse posséder les propriétés et les méthodes d'une autre classe
- ☐ Le fait que plusieurs méthodes de plusieurs objets différents puissent avoir le même nom, mais un corps différent
- ☐ Le fait que plusieurs variables puissent avoir le même nom dans des fonctions différentes

Exercice

Qu'est-ce qu'une interface ?

- ☐ L'ensemble des éléments visuels avec lesquels un utilisateur peut interagir sur une application
- ☐ Un "contrat" forçant les développeurs à implémenter les méthodes qui sont dans l'interface
- ☐ Une classe qui ne peut pas être instanciée

Exercice

Quelle est la différence entre une classe abstraite et une classe normale ?

- ☐ Une classe abstraite implémente une interface
- ☐ Une classe abstraite ne peut pas être instanciée
- ☐ Une classe abstraite ne possède pas de code, uniquement des définitions de méthodes

Exercice

Soit le code suivant :

```

1 <?php
2
3 class Drink {
4     public function getDescription(): string
5     {
6         return 'Une boisson';
7     }
8 }
9
10 class Soda extends Drink {
11     public function getDescription(): string
12     {
13         return parent::getDescription().' pétillante';
14     }
15 }
```

Que vont afficher les instructions suivantes ?

```

1 <?php
2
3 $soda = new Soda();
4 echo $soda->getDescription();
```

- ☐ Une boisson
- ☐ pétillante
- ☐ Une boisson pétillante

Exercice

Avec le code de la question précédente, que vont afficher les instructions suivantes ?

```
1 <?php
2 $drink = new Drink();
3 echo $drink->getDescription();
```

- ☐ Une boisson
- ☐ pétillante
- ☐ Une boisson pétillante

Exercice

Au code des questions précédentes, nous rajoutons le code suivant :

```
1 <?php
2 class EnergyDrink extends Drink {
3     public function getDescription(): string
4     {
5         return 'énergisante';
6     }
7 }
```

Qu'affiche le code suivant ?

```
1 <?php
2 $energyDrink = new EnergyDrink();
3 echo $energyDrink->getDescription();
```

- ☐ Une boisson
- ☐ énergisante
- ☐ Une boisson énergisante
- ☐ Une boisson pétillante

Exercice

Au code des questions précédentes, nous rajoutons le code suivant :

```
1 <?php
2 class Water extends Drink {}
```

Qu'affiche le code suivant ?

```
1 <?php
2 $water = new Water();
3 echo $water->getDescription();
```

- ☐ Une boisson
- ☐ Une boisson pétillante
- ☐ Cela provoque une erreur

B. Exercice : Défi

Dans ce défi, vous allez réaliser une application permettant de gérer des étudiants et des professeurs pour une université. Le but de cet exercice est d'utiliser tout ce que nous avons vu dans ce chapitre pour minimiser les répétitions de code : soyez donc attentifs aux copiés/collés !

Pour réaliser cet exercice, vous pouvez travailler sur l'environnement de travail :



Question

[solution n°9 p.43]

Une université gère des professeurs, des étudiants et les cours qui leur sont associés.

Un professeur possède un prénom, un nom de famille et enseigne une matière : chaque professeur ne peut enseigner qu'une seule matière. Un élève possède un prénom, un nom de famille et une liste de matières qu'il souhaite suivre. Enfin, une matière est composée d'un titre et d'une description.

Le résumé de chacun de ces éléments (professeur, élève ou matière) doit pouvoir être affiché sur le site Internet de l'université. Ce résumé sera simplement composé d'un titre de niveau 3 et d'une description, contenue dans une balise p.

Le titre d'un étudiant est son nom suivi de son prénom, celui d'un professeur est similaire, mais préfixé de "Professeur" et celui d'une matière est simplement son titre.

La description d'un étudiant est la liste de titres des matières qu'il étudie, celle d'un professeur est le nom de la matière qu'il enseigne, et celle d'une matière est simplement sa description.

Voici l'exemple d'un fichier *index.php* final que l'on souhaite et son résultat :

```
1 <?php
2
3 require_once 'Student.php';
4 require_once 'Course.php';
5 require_once 'Teacher.php';
6 require_once 'functions.php';
7
8 $french = new Course('Français', 'La langue de Molière');
9 $computerScience = new Course('Informatique', 'Apprendre à développer des sites internet');
10 $john = new Student('John', 'Doe', [$french, $computerScience]);
11 $laure = new Teacher('Laure', 'Dupond', $computerScience);
12
13 displayDescription($french);
14 displayDescription($computerScience);
15 displayDescription($john);
16 displayDescription($laure);
```

1 <https://repl.it/>

Français

La langue de Molière

Informatique

Apprendre à développer des sites internet

Doe John

Etudie Français, Informatique

Professeur Dupond Laure

Enseigne Informatique

Créez un script permettant de gérer l'ensemble de ces éléments en minimisant la répétition de code au maximum.

Indice :

Utilisez une classe abstraite permettant de centraliser tout ce qu'il y a en commun entre un étudiant et un professeur.

Pensez au fait qu'il doit être facile d'inverser le nom et le prénom des étudiants et des professeurs, par exemple.

Solutions des exercices

p. 6 Solution n°1

```
1 <?php
2
3 // Fichier ElectricCar.php
4 require_once 'Car.php';
5
6 class ElectricCar extends Car
7 {
8     public float $batteryAutonomy;
9
10    public function __construct(float $price, string $brand, float $batteryAutonomy)
11    {
12        $this->price = $price;
13        $this->brand = $brand;
14        // La caractéristique batteryAutonomy étant absente d'une voiture thermique,
15        // on ajoute ce paramètre uniquement lors de la "construction" de l'objet ElectricCar
16
17        $this->batteryAutonomy = $batteryAutonomy;
18    }
19 }

```

```
1 <?php
2
3 // Fichier GasolineCar.php
4 require_once 'Car.php';
5
6 class GasolineCar extends Car
7 {
8     public float $co2Emission;
9
10    public function __construct(float $price, string $brand, float $co2Emission)
11    {
12        $this->price = $price;
13        $this->brand = $brand;
14        // La caractéristique batteryAutonomy étant absente d'une voiture thermique,
15        // on ajoute ce paramètre uniquement lors de la "construction" de l'objet GasolineCar.
16
17        $this->co2Emission = $co2Emission;
18    }
19 }

```

```
1 <?php
2
3 // Fichier index.php
4 require_once 'ElectricCar.php';
5 require_once 'GasolineCar.php';
6
7 $tesla = new ElectricCar(50000, 'Tesla', 560);
8 $renault = new GasolineCar(20000, 'Renault', 100);

```

p. 12 Solution n°2

```

1 <?php
2
3 // Fichier Car.php
4 class Car
5 {
6     public float $price;
7     public string $brand;
8
9     public function __construct(float $price, string $brand)
10    {
11        $this->price = $price;
12        $this->brand = $brand;
13    }
14
15    public function getCharacteristics(): array
16    {
17        return [
18            'price' => $this->price,
19            'brand' => $this->brand,
20        ];
21    }
22 }
23
24
25
26

```

```

1 // Fichier ElectricCar.php
2 require_once 'Car.php';
3
4 class ElectricCar extends Car
5 {
6     public float $batteryAutonomy;
7
8     public function __construct(float $price, string $brand, float $batteryAutonomy)
9     {
10        parent::__construct($price, $brand);
11        $this->batteryAutonomy = $batteryAutonomy;
12    }
13
14    public function getCharacteristics(): array
15    {
16        // Ici, on fait appel à la fonction qui se situe dans l'objet parent (Car)
17        $characteristics = parent::getCharacteristics();
18        // Lors de l'appel de la fonction, on ajoute un champ à notre tableau qui est,
19        // comme son nom l'indique : batteryAutonomy. Et qui va venir utiliser le troisième paramètre
20        // passé à la fonction __construct et l'ajouter dans le champs "batteryAutonomy" que l'on crée
21        // grâce à :
22        $characteristics['batteryAutonomy'] = $this->batteryAutonomy;
23
24        return $characteristics;
25    }
26 }

```

```

1 <?php
2 // Fichier GasolineCar.php
3 require_once 'Car.php';
4
5 class GasolineCar extends Car

```

```

6 {
7     public float $co2Emission;
8
9     public function __construct(float $price, string $brand, float $co2Emission)
10    {
11        parent::__construct($price, $brand);
12        $this->co2Emission = $co2Emission;
13    }
14
15    public function getCharacteristics(): array
16    {
17        // Ici, on fait appel à la fonction qui se situe dans l'objet parent (Car)
18        $characteristics = parent::getCharacteristics();
19        // Lors de l'appel de la fonction, on ajoute un champ à notre tableau qui est,
20        // comme son nom l'indique : co2Emission. Et qui va venir utiliser le troisième paramètre
21        // passé à la fonction __construct et l'ajouter dans le champs "co2Emission" que l'on crée grâce
22        // à :
23        $characteristics['co2Emission'] = $this->co2Emission;
24        return $characteristics;
25    }
26 }

```

```

1 <?php
2
3 // Fichier index.php
4
5 require_once 'ElectricCar.php';
6 require_once 'GasolineCar.php';
7
8 $tesla = new ElectricCar(50000, 'Tesla', 560);
9 $renault = new GasolineCar(20000, 'Renault', 100);
10
11 var_dump($tesla->getCharacteristics());
12 var_dump($renault->getCharacteristics());

```

p. 12 Solution n°3

```

1 <?php
2
3 // Fichier carFunctions.php
4 require_once 'Car.php';
5
6 function displayCharacteristics(Car $car): void
7 {
8     echo '<ul>';
9
10    // Cette boucle va servir à prendre chaque champ du tableau que retourne la fonction
11    // getCharacteristics() et à les intégrer dans une balise <li> sous le format que l'on souhaite
12    // leur donner. Ici, "$name : $value"
13
14    foreach ($car->getCharacteristics() as $name => $value) {
15        echo '<li>'.$name.' : '.$value.'</li>';
16    }
17    echo '</ul>';
18 }

```

```

1
2 // Fichier index.php
3 require_once 'ElectricCar.php';
4 require_once 'GasolineCar.php';
5 require_once 'carFunctions.php';
6
7 $tesla = new ElectricCar(50000, 'Tesla', 560);
8 $renault = new GasolineCar(20000, 'Renault', 100);
9
10 displayCharacteristics($tesla);
11 displayCharacteristics($renault);

```

p. 19 Solution n°4

```

1 <?php
2
3 // Fichier Tire.php
4 class Tire
5 {
6     public float $width;
7     public float $height;
8     public float $diameter;
9
10    public function __construct(float $width, float $height, float $diameter)
11    {
12        $this->width = $width;
13        $this->height = $height;
14        $this->diameter = $diameter;
15    }
16 }

```

p. 19 Solution n°5

```

1 <?php
2
3 // Fichier CharacteristicsDisplayable.php
4 interface CharacteristicsDisplayable
5 {
6     // On vient définir notre fonction getCharacteristics() qui devra être présente dans les
6     classes Car et Tire.
7     public function getCharacteristics(): array;
8 }
9
10 // Fichier Car.php
11 // On vient préciser avec le require_once que nous allons avoir besoin de l'interface qui se
11 trouve dans le fichier CharacteristicsDisplayable
12 require_once 'CharacteristicsDisplayable.php';
13
14 class Car implements CharacteristicsDisplayable
15 {
16     // Le reste de la classe est identique, étant donné que la méthode getCharacteristics
16 existe déjà
17 }
18
19 // Fichier Tire.php
20 require_once 'CharacteristicsDisplayable.php';

```

```

21
22 class Tire implements CharacteristicsDisplayable
23 {
24     public float $width;
25     public float $height;
26     public float $diameter;
27
28     public function __construct(float $width, float $height, float $diameter)
29     {
30         $this->width = $width;
31         $this->height = $height;
32         $this->diameter = $diameter;
33     }
34 // On vient définir, après avoir utilisé notre require_once, la fonction getCharacteristics()
    qui se trouve au sein du fichier CharacteristicsDisplayable
35
36     public function getCharacteristics(): array
37     {
38         return [
39             'width' => $this->width,
40             'height' => $this->height,
41             'diameter' => $this->diameter,
42         ];
43     }
44 }
45
46 // Fichier carFunctions.php
47 require_once 'CharacteristicsDisplayable.php';
48
49 function displayCharacteristics(CharacteristicsDisplayable $characteristicsDisplayable): void
50 {
51     echo '<ul>';
52     // Cette boucle va venir chercher TOUS les champs trouvable avec la fonction
    getCharacteristics() qui se situe dans le fichier characteristicsDisplayable (pour rappel,
    nous avons utiliser la fonction aussi bien dans le fichier Car que dans le fichier Tire) et
    les afficher sous le forme que nous souhaitons lui donner, ici, une liste à puce qui regroupe
    donc le nom du champ ainsi que la valeur de celui-ci.
53     foreach ($characteristicsDisplayable->getCharacteristics() as $name => $value) {
54         echo '<li>'. $name. ' : ' . $value. '</li>';
55     }
56     echo '</ul>';
57 }
58
59 // Fichier index.php
60 require_once 'ElectricCar.php';
61 require_once 'GasolineCar.php';
62 require_once 'Tire.php';
63 require_once 'carFunctions.php';
64
65 $tesla = new ElectricCar(50000, 'Tesla', 560);
66 $renault = new GasolineCar(20000, 'Renault', 100);
67 $tire = new Tire(205, 55, 32);
68
69 displayCharacteristics($tesla);
70 displayCharacteristics($renault);
71 displayCharacteristics($tire);

```

p. 28 Solution n°6

La classe Car ne sert que de base aux autres voitures, donc peut être abstraite.

```
1 <?php
2
3 abstract class Car implements CharacteristicsDisplayable
4 {
5     // Le reste de la classe est identique
6 }
```

p. 29 Solution n°7

```
1 <?php
2
3 // Fichier Car.php
4 require_once 'CharacteristicsDisplayable.php';
5
6 abstract class Car implements CharacteristicsDisplayable
7 {
8     // Le reste de la classe est identique
9     public abstract function getFinalPrice(): float;
10 }
11
12 // Fichier ElectricCar.php
13 require_once 'Car.php';
14
15 class ElectricCar extends Car
16 {
17     // Le reste de la classe est identique
18     public function getFinalPrice(): float
19     {
20         return $this->price - 2500;
21     }
22 }
23
24 // Fichier GasolineCar.php
25 require_once 'Car.php';
26
27 class GasolineCar extends Car
28 {
29     // Le reste de la classe est identique
30     public function getFinalPrice(): float
31     {
32         $excessiveEmissions = $this->co2Emission - 119;
33         if ($excessiveEmissions <= 0) {
34             return $this->price;
35         }
36
37         return $this->price + 50 * $excessiveEmissions;
38     }
39 }
40
41 // Fichier carFunctions.php
42
43 require_once 'Car.php';
44
```



```


45 function displayPrice(Car $car): void
46 {
47     echo $car->getFinalPrice(). ' ('. $car->price. ')';
48 }
49
50 // Fichier index.php
51 require_once 'ElectricCar.php';
52 require_once 'GasolineCar.php';
53 require_once 'carFunctions.php';
54
55 $tesla = new ElectricCar(50000, 'Tesla', 560);
56 $renault = new GasolineCar(20000, 'Renault', 130);
57
58 displayPrice($tesla);
59 displayPrice($renault);

```

Exercice p. 29 Solution n°8


Exercice

Qu'est-ce que l'héritage ?

- ☐ Le fait d'importer des fonctions stockées dans un autre fichier
- ☐ Une manière de transmettre des données d'une classe lorsque son destructeur est appelé
- ☒ Une manière de partager des propriétés et des méthodes d'une classe avec une autre
-  L'héritage permet à une classe mère de partager ses propriétés et ses méthodes avec ses classes filles. Cela permet de centraliser le code dans une classe, mais de pouvoir l'utiliser ou d'ajouter des variantes dans les autres.

Exercice

Qu'est-ce que le polymorphisme ?

- ☐ Le fait qu'une classe puisse posséder les propriétés et les méthodes d'une autre classe
Ce principe est "l'héritage". Bien que le polymorphisme fasse partie de l'héritage, c'est un concept à part entière.
- ☒ Le fait que plusieurs méthodes de plusieurs objets différents puissent avoir le même nom, mais un corps différent
- ☐ Le fait que plusieurs variables puissent avoir le même nom dans des fonctions différentes
-  Le polymorphisme désigne le mécanisme permettant à différentes classes d'avoir le même nom de méthode, mais une implémentation différente. Il existe plusieurs manières d'utiliser le polymorphisme : l'héritage et les interfaces sont les deux exemples les plus courants.

Exercice

Qu'est-ce qu'une interface ?

- ☐ L'ensemble des éléments visuels avec lesquels un utilisateur peut interagir sur une application
Bien que ça ait le même nom, nous parlons ici des interfaces au sens de la Programmation Orientée Objet. C'est pour éviter cette ambiguïté que l'on parle plus volontiers de "UI", pour "User Interface", lorsque l'on parle de l'aspect front d'un site.
- ☒ Un "contrat" forçant les développeurs à implémenter les méthodes qui sont dans l'interface
- ☐ Une classe qui ne peut pas être instanciée

- Q Une interface est un ensemble de signatures de méthodes qui devront être implémentées dans toutes les classes qui souhaitent implémenter l'interface.

Exercice

Quelle est la différence entre une classe abstraite et une classe normale ?

- ☐ Une classe abstraite implémente une interface
Une classe normale peut également implémenter une interface.
 - ☒ Une classe abstraite ne peut pas être instanciée
 - ☐ Une classe abstraite ne possède pas de code, uniquement des définitions de méthodes
On parle d'interface dans ce cas.
- Q Une classe abstraite est une classe qui ne peut pas être instanciée : essayer de faire un `new` dessus soulèvera une erreur.

Exercice

Soit le code suivant :

```
1 <?php
2
3 class Drink {
4     public function getDescription(): string
5     {
6         return 'Une boisson';
7     }
8 }
9
10 class Soda extends Drink {
11     public function getDescription(): string
12     {
13         return parent::getDescription().' pétillante';
14     }
15 }
```

Que vont afficher les instructions suivantes ?

```
1 <?php
2
3 $soda = new Soda();
4 echo $soda->getDescription();
```

- ☐ Une boisson
 - ☐ pétillante
 - ☒ Une boisson pétillante
- Q Nous avons instancié un objet de type `Soda`, donc c'est la méthode `getDescription` de la classe `Soda` qui est appelée. Cette méthode fait appel à la méthode parente, qui retourne "Une boisson", et y concatène "pétillante". Cela donne "Une boisson pétillante".

Exercice

Avec le code de la question précédente, que vont afficher les instructions suivantes ?

```
1 <?php
2 $drink = new Drink();
3 echo $drink->getDescription();
```

☒ Une boisson

☐ pétillante

☐ Une boisson pétillante

Q Cette fois, nous avons créé un objet de type `Drink`, c'est donc la méthode `getDescription` de la classe `Drink` qui est appelée. Elle retourne simplement "Une boisson".

Exercice

Au code des questions précédentes, nous rajoutons le code suivant :

```
1 <?php
2 class EnergyDrink extends Drink {
3     public function getDescription(): string
4     {
5         return 'énergisante';
6     }
7 }
```

Qu'affiche le code suivant ?

```
1 <?php
2 $energyDrink = new EnergyDrink();
3 echo $energyDrink->getDescription();
```

☐ Une boisson

☒ énergisante

☐ Une boisson énergisante

☐ Une boisson pétillante

Q En ayant créé un objet de type `EnergyDrink`, c'est la méthode `getDescription` de la classe `EnergyDrink` qui est appelée. Cette méthode retourne "énergisante", donc c'est ce qui est affiché. De toute évidence, il manque un appel à une méthode parente.

Exercice

Au code des questions précédentes, nous rajoutons le code suivant :

```
1 <?php
2 class Water extends Drink {}
```

Qu'affiche le code suivant ?

```
1 <?php
2 $water = new Water();
3 echo $water->getDescription();
```

☒ Une boisson

☐ Une boisson pétillante

☐ Cela provoque une erreur

Q En ayant créé un objet de type `Water`, c'est la méthode `getDescription` de la classe `Water` qui est appelée. Or, comme cette classe ne possède pas de méthode `getDescription`, c'est celle de la classe mère qui est appelée. Ce code affiche donc : "Une boisson".

```

1 <?php
2
3
4 // Fichier Descriptable.php
5 interface Descriptable
6 {
7     public function getTitle(): string;
8
9     public function getDescription(): string;
10 }
11
12
13 // Fichier Course.php
14 require_once 'Descriptable.php';
15
16 class Course implements Descriptable
17 {
18     public string $title;
19     public string $description;
20
21     public function __construct(string $title, string $description)
22     {
23         $this->title = $title;
24         $this->description = $description;
25     }
26
27     public function getTitle(): string
28     {
29         return $this->title;
30     }
31
32     public function getDescription(): string
33     {
34         return $this->description;
35     }
36 }
37
38 //Fichier Person.php
39 require_once 'Descriptable.php';
40
41 abstract class Person implements Descriptable
42 {
43     public string $firstName;
44     public string $lastName;
45
46     public function __construct(string $firstName, string $lastName)
47     {
48         $this->firstName = $firstName;
49         $this->lastName = $lastName;
50     }
51
52     public function getFullName()
53     {
54         return $this->lastName.' '.$this->firstName;
55     }
56
57     public function getTitle(): string
58     {

```

```
59         return $this->getFullName();
60     }
61 }
62
63 // Fichier Student.php
64 require_once 'Person.php';
65 require_once 'Course.php';
66
67 class Student extends Person
68 {
69     public array $courses;
70
71     public function __construct(string $firstName, string $lastName, array $courses)
72     {
73         parent::__construct($firstName, $lastName);
74         $this->courses = $courses;
75     }
76
77     public function getDescription(): string
78     {
79         $coursesName = [];
80         foreach ($this->courses as $course)
81         {
82             $coursesName[] = $course->title;
83         }
84
85         return 'Etudie '.implode(', ', $coursesName);
86     }
87 }
88
89 // Fichier Teacher.php
90 require_once 'Person.php';
91 require_once 'Course.php';
92
93 class Teacher extends Person
94 {
95     public Course $course;
96
97     public function __construct(string $firstName, string $lastName, Course $course)
98     {
99         parent::__construct($firstName, $lastName);
100         $this->course = $course;
101     }
102
103     public function getFullName()
104     {
105         return 'Professeur '.parent::getFullName();
106     }
107
108     public function getDescription(): string
109     {
110         return 'Enseigne '.$this->course->title;
111     }
112 }
113
114
115 // Fichier functions.php
116 require_once 'Descriptable.php';
```

```
117  
118 function displayDescription(Descriptable $descriptable)  
119 {  
120     echo '<h3>'.$descriptable-&gtgetTitle().'</h3><p>'.$descriptable-&gtgetDescription().'</p>';  
121 }
```