

# **La Programmation Orientée**

## **Objet : l'encapsulation et la visibilité**

# Table des matières

<b>I. Contexte</b>	<b>3</b>
<b>II. L'encapsulation</b>	<b>3</b>
<b>III. Exercice : Appliquez la notion</b>	<b>5</b>
<b>IV. La visibilité</b>	<b>6</b>
<b>V. Exercice : Appliquez la notion</b>	<b>10</b>
<b>VI. Les accesseurs et les mutateurs</b>	<b>11</b>
<b>VII. Exercice : Appliquez la notion</b>	<b>15</b>
<b>VIII. Les classes finales</b>	<b>15</b>
<b>IX. Exercice : Appliquez la notion</b>	<b>20</b>
<b>X. Auto-évaluation</b>	<b>21</b>
A. Exercice final .....	21
B. Exercice : Défi .....	23
<b>Solutions des exercices</b>	<b>24</b>

## I. Contexte

**Durée :** 1 h

**Environnement de travail :** Local (via XAMPP)

**Pré-requis :** Connaître les aspects de base et avancés de la POO

### Contexte

Grâce à la Programmation Orientée Objet, il est possible de créer des architectures à base de classes permettant de gérer tous les aspects de notre projet, en évitant la duplication de code.

C'est donc un outil très puissant, mais, comme pour tous les outils, il faut imposer des limites et ne pas permettre de l'utiliser n'importe comment.

Dans ce cours, nous allons voir comment il est possible de protéger nos classes d'une mauvaise utilisation de ses méthodes ou de ses propriétés.

## II. L'encapsulation

### Objectifs

- Comprendre la notion d'encapsulation
- Savoir déterminer l'interface de nos classes

### Mise en situation

Lorsque l'on crée une classe, il faut toujours garder à l'esprit que l'on ne sait jamais qui va l'utiliser, ni comment est-ce qu'il va la manipuler. Si l'on travaille en équipe, les autres développeurs ne savent pas forcément comment utiliser nos objets et pourraient appeler des méthodes qui ne sont pas censées l'être. Si on travaille seul, rien ne nous dit que le projet que l'on fait pour un client ne sera pas donné à une autre équipe. Même le "nous du futur", qui reprendra le code dans quelques années, est à voir comme un autre développeur qui n'aura pas forcément en tête toutes les subtilités de nos classes.

Or, modifier une propriété que l'on n'est pas censé toucher, ou appeler une méthode à un moment qui n'est pas prévu, peut provoquer des problèmes dans nos applications. Il est donc indispensable de protéger nos classes afin de s'assurer que la seule manière possible de les utiliser est bien celle que l'on a prévue.

### Définition L'encapsulation

Un des principaux intérêts des classes est de regrouper dans un même endroit les données que l'on va manipuler et les actions que l'on va réaliser sur ces données.

Ce principe s'appelle l'**encapsulation** et c'est ce qui différencie les classes des simples fonctions.

### Exemple Encapsuler une ressource

Prenons l'exemple d'une classe permettant d'écrire dans un fichier : on ouvre le fichier et on stocke la ressource vers ce fichier dans une propriété, qui sera réutilisée par les autres méthodes.

```

1 <?php
2
3 class File
4 {
5     public $fileResource;
6
7     public function __construct(string $filename)
8     {
9         $this->fileResource = fopen($filename, 'a');
10    }
11
12    public function write(string $content)
13    {
14        return fwrite($this->fileResource, $content);
15    }
16
17    public function close(): bool
18    {
19        return fclose($this->fileResource);
20    }
21 }
```

Nous avons ainsi **encapsulé** la ressource du fichier avec les actions que l'on va faire sur cette ressource.

### Définition Les interfaces

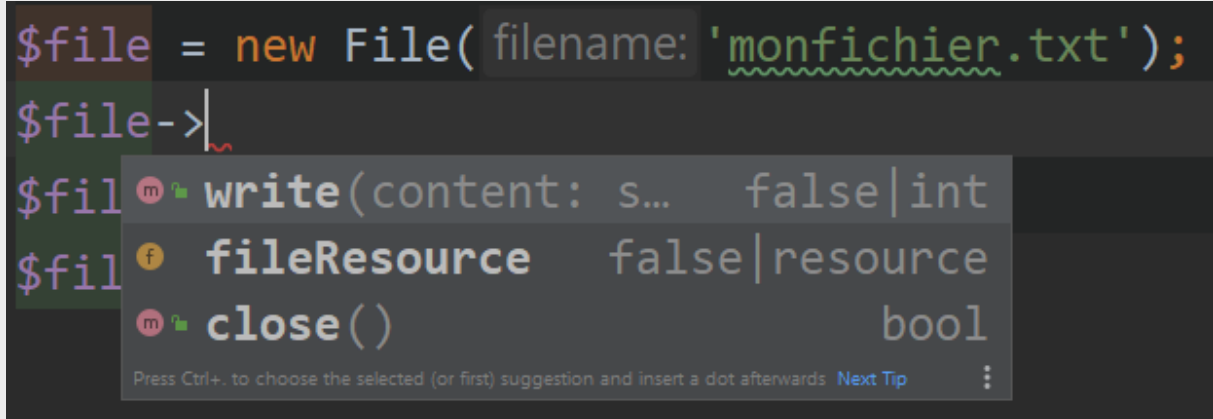
Le fait d'avoir encapsulé ces données dans une classe fait que nous avons défini un ensemble de propriétés et de méthodes que les développeurs utilisant notre classe pourront utiliser : on parle d'**interface** d'une classe.

Nous avons déjà vu ce terme lorsque nous avons créé des interfaces permettant de forcer l'implémentation de certaines méthodes, mais c'est en réalité un terme un peu plus général qui permet de désigner l'ensemble des méthodes que l'on peut utiliser lorsque l'on instancie un objet.

**Exemple**

L'interface de notre classe `File` est constituée des méthodes `write()` et `close()`. Nous pouvons également accéder à la propriété `fileResource`.

L'interface de notre classe est ce qui nous est proposé par notre IDE lorsque l'on veut agir sur notre objet :



```
$file = new File( filename: 'monfichier.txt');
$file->
$file write(content: s... false|int
$file fileResource false|resource
$file close() bool
Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards. Next Tip
```

**Complément** **Se poser les bonnes questions**

L'encapsulation et l'interface sont des notions qui ne sont pas nouvelles : nous les manipulons depuis le début de la POO. Cependant, elles permettent de se poser une question à laquelle nous allons répondre : faut-il que tout ce qui est **encapsulé** dans notre classe fasse partie de son **interface** ? Ou, autrement dit, est-ce qu'un développeur utilisant notre classe doit avoir accès à toutes ses méthodes et tous ses attributs ?

**Syntaxe** **À retenir**

- Le fait de centraliser, au même endroit, des données et les actions effectuées sur ces données s'appelle l'**encapsulation**.
- Lorsque l'on crée une classe, on met à disposition des développeurs une **interface**, c'est-à-dire un ensemble de méthodes qui pourront être utilisées.

**Complément**

L'encapsulation<sup>1</sup>

**III. Exercice : Appliquez la notion**

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



2

<sup>1</sup> [https://fr.wikipedia.org/wiki/Encapsulation\\_\(programmation\)](https://fr.wikipedia.org/wiki/Encapsulation_(programmation))

<sup>2</sup> <https://repl.it/>

## Question

[solution n°1 p.25]

Pour une application de gestion d'articles de journaux, nous avons besoin d'encapsuler la gestion des textes dans une classe. Vous devez donc créer une classe `Text` qui sera utilisée par l'équipe de développement et qui comportera l'interface suivante :

- Une méthode `getSentences()` : `array` qui retourne un tableau de phrases composant le texte. Pour simplifier le problème, on considère que seuls les points seront utilisés dans nos textes pour séparer les phrases, et que toutes les phrases seront bien formées (une phrase à la fin de chacune).
- Une méthode `getTextLength()` : `int` qui retourne le nombre de caractères de notre texte.
- Une méthode `getSentencesCount()` : `int` qui retourne le nombre de phrases dans notre texte.

Voici comment la classe pourrait être utilisée :

```
1 $text = new Text('Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec lacinia quam
  sed lacinia blandit. In et viverra elit. Nullam molestie quam eget porta venenatis. Aenean
  blandit auctor turpis, eu rhoncus libero pulvinar id. Aenean euismod enim ac sagittis
  accumsan. Fusce venenatis purus orci, in euismod erat porttitor in. Morbi semper dignissim
  felis a tincidunt.');
```

```
2 print_r($text->getSentences());
```

```
3 echo 'Le texte possède ' . $text->getSentencesCount() . ' phrases et ' . $text->getTextLength() . '
  caractères.';
```

### Indice :

Il existe une fonction PHP permettant de réaliser simplement les deux premières fonctions. Pour la troisième, il suffit d'appeler la première et de compter le nombre de cases : une fonction existe également en PHP pour ça.

### Indice :

Attention, la méthode `explode()` est pratique pour séparer les phrases, mais le dernier point de la dernière phrase va provoquer une case vide à la fin du tableau. Il faut penser à l'enlever.

## IV. La visibilité

### Objectifs

- Comprendre les risques liés à l'interface de nos classes
- Modifier l'interface en gérant la visibilité de nos méthodes et propriétés

### Mise en situation

Les méthodes et propriétés qui sont accessibles depuis l'extérieur de notre classe s'appellent les méthodes **publiques**. Jusqu'à présent, nous n'avons déclaré que des propriétés et des méthodes publiques, d'où le mot-clé `public`. Cependant, il est parfois préférable que certains éléments restent internes à la classe et ne soient pas accessibles depuis l'extérieur.

### Sécuriser les éléments mis en place

Dans notre exemple, concernant la ressource du fichier, n'importe qui pourrait la modifier à tout moment. On pourrait ainsi écrire dans un fichier qui n'est pas celui attendu, voire avoir une ressource qui ne correspond pas du tout à celle d'un fichier, ce qui provoquerait une erreur :

```
1 <?php
2
3 // Fichier File.php
4 class File
5 {
6     public $fileResource;
7
8     public function __construct(string $filename)
```

```
9      {
10         $this->fileResource = fopen($filename, 'a');
11     }
12
13     public function write(string $content)
14     {
15         return fwrite($this->fileResource, $content);
16     }
17
18     public function close(): bool
19     {
20         return fclose($this->fileResource);
21     }
22 }
23
24 // Fichier index.php
25 require_once 'File.php';
26
27 $file = new File('monfichier.txt');
28 $file->fileResource = null; // Rien ne m'empêche de rendre la ressource nulle
29 $file->write('Hello world!'); // Provoque une erreur
30 $file->close(); // Provoque une erreur
```

Lorsque l'on crée une classe, il est donc de notre responsabilité de vérifier l'interface de notre classe et de s'assurer qu'il ne soit pas possible d'utiliser des méthodes et des propriétés qui ne sont pas prévues pour. Pour cela, on utilise la **visibilité**.

#### Définition La visibilité

La **visibilité** est le fait d'indiquer, pour chaque attribut et chaque méthode, qui est capable de l'utiliser. Il existe trois niveaux de visibilité : publique, protégée et privée.

La **visibilité publique** permet à n'importe qui d'accéder aux propriétés et aux méthodes de notre classe. C'est la visibilité que nous utilisons jusqu'à présent.

La **visibilité protégée** permet d'interdire l'utilisation d'une méthode ou d'une propriété en dehors de notre classe ou de ses classes filles. C'est-à-dire que n'importe quel objet instancié à l'extérieur de notre classe ne pourra pas les utiliser. En d'autres termes, cela signifie que les éléments protégés ne seront accessibles que depuis la variable `$this`.

Enfin, la **visibilité privée** permet d'interdire l'utilisation d'une méthode ou d'une propriété strictement en dehors de notre classe. Même les classes filles n'auront pas accès à ces éléments.

#### Syntaxe La visibilité publique

La visibilité se définit indépendamment au niveau de chaque élément de notre classe. Jusqu'à présent, nous avons utilisé le mot-clé `public` pour définir une propriété ou une méthode. Ce mot-clé sert en réalité à définir une visibilité publique pour nos éléments.

#### Exemple

```
1 <?php
2
3 class User
4 {
5     public string $username; // Cette propriété est publique
6 }
```

```

7     public function __construct(string $username)
8     {
9         $this->username = $username;
10    }
11
12    public function getName(): string // Cette méthode est publique
13    {
14        return $this->username;
15    }
16 }
17
18 $john = new User('John');
19
20 $john->username = 'Johnny'; // Je peux donner une valeur à cette propriété parce qu'elle est
    publique
21 echo $john->username.PHP_EOL; // Je peux afficher cette propriété parce qu'elle est publique
22 echo $john->getName(); // Je peux appeler cette méthode parce qu'elle est publique

```

## Syntaxe La visibilité protégée

La visibilité protégée se déclare grâce au mot-clé `protected`.

## Exemple

```

1 <?php
2
3 class User
4 {
5     protected string $username; // Cette propriété est protégée
6
7     public function __construct(string $username)
8     {
9         $this->username = $username;
10    }
11
12    protected function getName(): string // Cette méthode est protégée
13    {
14        return $this->username;
15    }
16 }
17
18 class Admin extends User
19 {
20     public function getAdministratorName(): string // Cette méthode est publique
21     {
22         return 'Administrateur ' . $this->getName(); // La classe Admin a accès à la méthode
    getName car elle est protégée
23    }
24 }
25
26 $john = new Admin('John');
27
28 $john->username = 'Johnny'; // Retourne une erreur car la propriété est protégée
29 echo $john->username.PHP_EOL; // Retourne une erreur car la propriété est protégée
30 echo $john->getName(); // Retourne une erreur car la méthode est protégée
31 echo $john->getAdministratorName(); // On peut appeler cette méthode parce qu'elle est
    publique, même si elle fait appel à des éléments protégés

```



**Syntaxe**    **La visibilité privée**

La visibilité privée se déclare grâce au mot-clé `private`.

**Exemple**

```
1 <?php
2
3 class User
4 {
5     private string $username; // Cette propriété est privée
6
7     public function __construct(string $username)
8     {
9         $this->username = $username;
10    }
11
12    private function getName(): string // Cette méthode est privée
13    {
14        return $this->username;
15    }
16 }
17
18 class Admin extends User
19 {
20     public function getAdministratorName(): string // Cette méthode est publique
21     {
22         return 'Administrateur ' . $this->getName(); // La classe Admin n'a pas accès à la
23         méthode getName car elle est privée
24     }
25 }
26 $john = new Admin('John');
27
28 $john->username = 'Johnny'; // Retourne une erreur car la propriété est privée
29 echo $john->username.PHP_EOL; // Retourne une erreur car la propriété est privée
30 echo $john->getName(); // Retourne une erreur car la méthode est privée
31 echo $john->getAdministratorName(); // Retourne une erreur car la méthode est buguée
```

Vous pouvez vous représenter très facilement ces trois niveaux de visibilité en faisant un parallèle avec les niveaux de protection de ce que vous postez sur les réseaux sociaux.

Premier niveau, vos posts sont publics : toutes les personnes sur internet peuvent voir vos publications, qu'elles soient ou non enregistrées sur la plateforme.

Deuxième niveau, « seulement moi » ou `private` : vous et vous seul avez accès à vos publications.

Troisième niveau, « visible pour mes amis » ou `protected` : seulement les personnes ayant un compte sur l'application et faisant partie de votre réseau peuvent accéder à vos publications. C'est ce processus de réflexion sur le niveau de visibilité qui doit accompagner chaque variable ou fonction que vous déclarez.

**Syntaxe**    **À retenir**

La **visibilité** permet de modifier l'interface d'une classe en masquant certaines propriétés et méthodes.

Il existe trois niveaux de visibilité :

- `private`, qui masque pour tout le monde sauf pour la classe elle-même.
- `protected`, qui masque pour les appels extérieurs à la classe, mais autorise les appels depuis les classes filles.
- `public`, qui autorise tous les appels, d'où qu'ils soient.

## V. Exercice : Appliquez la notion

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



### Question

[solution n°2 p.25]

Un hôpital vous demande d'améliorer le code de son application de gestion de patients pour éviter au maximum que les données des utilisateurs soient mal manipulées. Voici la classe actuelle :

```

1 <?php
2
3
4 class Patient
5 {
6     public int $id;
7     public string $firstName;
8     public string $lastName;
9
10    public function __construct(int $id)
11    {
12        $this->loadPatient($id);
13    }
14
15    public function loadPatient(int $id)
16    {
17        // Nous simulons l'appel à une base de données en créant un tableau en dur
18        $data = [
19            'id' => $id,
20            'firstName' => 'John',
21            'lastName' => 'Doe',
22        ];
23
24        $this->id = $data['id'];
25        $this->firstName = $data['firstName'];
26        $this->lastName = $data['lastName'];
27    }
28
29    public function getFullName()
30    {
31        return $this->firstName . ' ' . $this->lastName;

```

1 <https://repl.it/>

```

32     }
33 }

```

Protégez cette classe en effectuant les modifications suivantes :

- La propriété `id` ne doit pouvoir être manipulée que par notre classe : on ne doit pas pouvoir modifier l'`id` sous peine de modifier un autre patient que celui que l'on a chargé.
- Les propriétés `firstName` et `lastName` ne doivent jamais être récupérées directement par les scripts externes, mais toujours via la méthode `getFullName()`. En revanche, elles peuvent être manipulées par les classes filles.
- La méthode `loadPatient()` ne doit pouvoir être appelée que par notre classe. Un objet représentant un patient, on ne doit pas pouvoir appeler cette méthode ailleurs que dans le constructeur de notre classe.

## VI. Les accesseurs et les mutateurs

### Objectifs

- Comprendre ce que sont les accesseurs et les mutateurs
- Utiliser des méthodes pour gérer la visibilité des propriétés

### Mise en situation

La visibilité permet de gérer l'accès à nos méthodes et propriétés. Mais, si les niveaux de visibilité sont suffisants pour les méthodes, une problématique se pose pour les propriétés : étant donné qu'il est possible à la fois de leur attribuer une valeur et de la lire, il peut être intéressant de gérer ces deux opérations indépendamment. Autoriser la lecture d'une valeur, mais pas son écriture. Nous allons voir comment les accesseurs nous permettent de répondre à cette problématique.

#### Méthode Différencier le type d'accès autorisé

Pour gérer indépendamment l'accès à une variable en lecture et écriture en PHP, la technique est simple : il suffit de déclarer la propriété en privé, puis de créer des méthodes qui vont permettre de récupérer la valeur de notre propriété ou de la modifier.

Ainsi, c'est la visibilité de ces méthodes qui vont décider des opérations autorisées sur notre propriété.

#### Définition Les accesseurs

Un **accesseur**, appelé **getter**, est une méthode permettant d'accéder à la valeur d'une propriété. Le nom d'un accesseur est toujours préfixé par `get`, suivi du nom de la propriété.

### Lire un nom d'utilisateur

Voici une classe définissant un utilisateur de base, donc avec son nom d'utilisateur, et l'accesseur `getUsername` associé :

```

1 <?php
2
3 class User
4 {
5     private string $username; // La propriété est privée pour éviter de la manipuler
    directement
6
7     public function __construct(string $username)
8     {
9         $this->username = $username;

```

```

10     }
11
12     public function getUsername(): string // L'accesseur est public : nous avons le droit
d'accéder à la valeur de notre propriété
13     {
14         return $this->username;
15     }
16 }
17
18 $john = new User('John');
19
20
21 echo $john->getUsername(); // Pour afficher le nom de l'utilisateur, nous passons par son
accesseur

```

### Définition Les mutateurs

Un **mutateur**, appelé **setter**, est une méthode permettant de modifier la valeur d'une propriété. Le nom d'un mutateur est toujours préfixé par **set**, suivi du nom de la propriété. Un mutateur permet également de contrôler la validité des données que l'on essaie de faire passer à notre propriété.

### Définir un nom d'utilisateur

Ajoutons un mutateur `setUsername` à notre la classe `User` :

```

1 <?php
2
3 class User
4 {
5     private string $username;
6
7     public function __construct(string $username)
8     {
9         $this->username = $username;
10    }
11
12    public function getUsername(): string
13    {
14        return $this->username;
15    }
16
17    public function setUsername(string $username): void // Le mutateur est public : nous avons
le droit de modifier la valeur de notre propriété
18    {
19        if (ctype_alnum($username)) { // On vérifie que notre nom d'utilisateur soit bien
alphanumérique
20            $this->username = $username;
21        }
22    }
23 }
24
25 $john = new User('John');
26 $john->setUsername('JohnDoe'); //Pour modifier le nom de l'utilisateur, nous passons par son
mutateur
27
28 echo $john->getUsername(); // Affiche "JohnDoe"

```

**void ( : void )**

Une fonction de type void ne retourne aucune valeur, elle se termine soit par une déclaration soit retour vide ( return; ) soit sans déclaration de retour.

**Remarque** **Différencier constructeurs et mutateurs**

Les mutateurs **ne remplacent pas les constructeurs**. Bien que les deux servent à initialiser les valeurs de nos propriétés, un constructeur permet de définir quelles propriétés sont nécessaires au bon fonctionnement de notre objet. En effet, si une propriété n'est pas présente dans le constructeur d'une classe, alors sa valeur sera celle par défaut (potentiellement nulle) tant que l'on n'a pas appelé son mutateur. Il faudra donc potentiellement modifier le code en conséquence.

**Définition** **Les fluent setters**

Si notre classe comporte beaucoup de propriétés, appeler nos mutateurs un à un peut s'avérer répétitif et verbeux. Pour pallier ce problème, nous pouvons utiliser des **fluent setters**. Cela consiste à retourner l'objet courant dans notre mutateur plutôt que de ne rien retourner. Grâce à cette technique, nous pouvons ainsi enchaîner les modifications de nos propriétés sans avoir à répéter le nom de la variable.

Le type de retour d'un fluent setter est le type de la classe actuelle : par exemple, les fluent setters de notre classe `User` vont retourner l'objet `User` actuel.

Cependant, mettre `User` en type de retour pourrait poser des problèmes lors de l'héritage : cela signifie que, si une classe `Admin` hérite de `User`, alors la méthode `setUsername` de notre classe `Admin` devra retourner un `User` au lieu d'un `Admin`.

Pour éviter ces problèmes, il existe un mot-clé permettant d'utiliser le type de la classe actuelle : `self`.

**self ( : self ou self:: )**

Les références statiques avec `self` sont résolues en utilisant la classe à laquelle appartiennent les fonctions, celle où elles ont été définies.

**Une classe complète**

Agrémentons notre classe `User` de quelques propriétés et des accesseurs/mutateurs associés en utilisant la technique des fluent setters :

```
1 <?php
2
3 class User
4 {
5     private int $id;
6     private string $username;
7     private string $firstName;
8     private string $lastName;
9
10    public function __construct(int $id, string $username, string $firstName, string
11    $lastName)
12    {
13        $this->id = $id;
14        $this->username = $username;
15        $this->firstName = $firstName;
16        $this->lastName = $lastName;
17    }
18 }
```

```

18     public function getUsername(): string
19     {
20         return $this->username;
21     }
22
23     public function setUsername(string $username): self // On remarque l'utilisation de self
pour définir le type de retour
24     {
25         if (ctype_alnum($username)) {
26             $this->username = $username;
27         }
28
29         return $this;
30     }
31
32     public function getId(): int
33     {
34         return $this->id;
35     }
36
37     public function setId(int $id): self
38     {
39         $this->id = $id;
40
41         return $this;
42     }
43
44     public function getFirstName(): string
45     {
46         return $this->firstName;
47     }
48
49     public function setFirstName(string $firstName): self
50     {
51         $this->firstName = $firstName;
52
53         return $this;
54     }
55
56     public function getLastName(): string
57     {
58         return $this->lastName;
59     }
60
61     public function setLastName(string $lastName): self
62     {
63         $this->lastName = $lastName;
64
65         return $this;
66     }
67 }
68
69 $john = new User(1, 'JohnDoe123', 'John', 'Doe');
70 // Il est possible de tout mettre sur une ligne, mais, pour des raisons de lisibilité, il est
préférable d'utiliser un setter par ligne
71 $john
72     ->setUsername('JohnDoe') // Cette méthode retourne $john, donc on peut enchaîner les
appels
73     ->setFirstName('Johnny')

```

```

74     ->setLastName('Doe-Dupont')
75 ;
76
77 echo $john->getUsername(); // Affiche "JohnDoe". On aurait également pu rajouter cet appel aux
    précédents, mais cela pourrait nuire à la clarté du code.

```

### Syntaxe À retenir

- Les **accesseurs** (getters) sont des méthodes permettant de récupérer la valeur d'une propriété. Leur nom est sous la forme `getProperty`.
- Les **mutateurs** (setters) sont des méthodes permettant d'assigner une valeur à une propriété et de vérifier sa validité. Leur nom est sous la forme `setProperty`.
- Ces méthodes permettent de gérer la visibilité des opérations de lecture et d'écriture d'une propriété. Pour cela, il faut que la propriété ait une visibilité privée et que toutes les manipulations soient faites avec les méthodes.
- Les **fluent setters** sont des mutateurs qui retournent l'objet courant : cela permet d'enchaîner les appels.

## VII. Exercice : Appliquez la notion

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



### Question

[solution n°3 p.26]

Un restaurant souhaiterait avoir une application permettant de gérer les plats commandés par ses clients.

Un plat possède les caractéristiques suivantes :

- Un nom, qui est une chaîne de caractères. Il doit être défini dans le constructeur, être accessible depuis l'extérieur en lecture, mais ne doit pas être modifiable.
- Une liste d'ingrédients en supplément, qui est un tableau de chaînes de caractères. Ce sont les ingrédients que les utilisateurs veulent rajouter dans leur plat. Ils doivent être modifiables et accessibles depuis l'extérieur de notre classe.
- Un prix, qui est un nombre à virgule. Il doit être défini dans le constructeur et ne doit pas être modifiable. On doit pouvoir cependant récupérer le prix à l'extérieur de la classe, mais en prenant en compte la majoration de 1 € par ingrédient supplémentaire. Il doit être manipulable directement par les classes filles.

Créez une classe permettant de gérer ces contraintes.

Voici un exemple d'utilisation :

```

1 $meal = new Meal('Salade Caesar', 10.50);
2 $meal->setIngredients(['Olives', 'Cheddar']);
3 echo $meal->getPrice(); // Affiche 12.5

```

## VIII. Les classes finales

1 <https://repl.it/>

## Objectifs

- Implémenter des classes finales
- Comprendre la logique de composition en PHP

## Mise en situation

Grâce à la visibilité, nous pouvons protéger nos classes des utilisations non voulues lors de l'utilisation de nos objets dans des scripts externes.

Cependant, actuellement, rien n'empêche un développeur mal averti ou mal intentionné de faire hériter notre classe pour pouvoir arriver à ses fins : créer une méthode permettant de modifier une propriété dont le mutateur a été défini comme étant privé, par exemple, ce qui pourrait provoquer des anomalies lors de l'exécution du reste des méthodes.

Pour éviter cela, il est possible d'empêcher l'héritage de nos classes en les définissant comme des classes finales.

### Définition Classe finale

Une **classe finale** est une classe qui ne peut pas avoir de classe fille. Essayer d'étendre une classe finale provoquera une erreur fatale de PHP.

Cela permet de s'assurer que notre classe ne sera utilisée que dans les cas qui ont été définis, et qu'il ne sera pas possible de passer outre en utilisant l'héritage.

Pour déclarer une classe finale, il suffit d'utiliser le mot-clé `final` devant la déclaration de la classe :

```
1 <?php
2
3 final class User
4 {
5     // Le contenu de la classe ne change pas
6 }
```

## Composition over inheritance

L'utilisation des classes finales met en valeur une philosophie qui est très importante en PHP : *composition over inheritance*. Cela signifie que nous allons préférer composer nos classes en utilisant d'autres classes, quand c'est possible, plutôt que d'utiliser l'héritage.

### Exemple

Pour expliquer ce concept, nous allons prendre l'exemple un site e-commerce qui propose des produits à destination du monde entier.

Ainsi, le prix de nos produits devra être affiché avec la bonne monnaie et le bon format : si les prix sont en euros, le symbole "€" est placé après le prix, alors qu'en dollars, le symbole "\$" est placé avant.

Implémentons cela en utilisant l'héritage grâce à une classe abstraite `Product` permettant de définir la logique de base, et deux classes filles `EUPRODUCT` et `USPRODUCT` permettant de faire le formatage.

```
1 <?php
2
3 abstract class Product
4 {
5     private string $name;
6     private float $price;
7
8     public function __construct(string $name, float $price)
```



```
9      {
10          $this->name = $name;
11          $this->price = $price;
12      }
13
14      public function getName(): string
15      {
16          return $this->name;
17      }
18
19      public function setName(string $name): void
20      {
21          $this->name = $name;
22      }
23
24      public function getPrice(): float
25      {
26          return $this->price;
27      }
28
29      public function setPrice(float $price): void
30      {
31          $this->price = $price;
32      }
33
34      public abstract function formatPrice(): string;
35 }
36
37 class EUProduct extends Product
38 {
39     public function formatPrice(): string
40     {
41         return $this->getPrice().' €';
42     }
43 }
44
45 class USProduct extends Product
46 {
47     public function formatPrice(): string
48     {
49         return '$'.$this->getPrice();
50     }
51 }
52
53 $hardDrive = new EUProduct('Disque dur', 140.00);
54 $UShardDrive = new USProduct('Hard Drive', 140.00);
55
56 echo $hardDrive->formatPrice(); // Affiche 140.00 €
57 echo '<br>';
58 echo $UShardDrive->formatPrice(); // Affiche $140.00
```

Cela fonctionne bien, mais ajoutons une autre contrainte et considérons que chaque type de produit possède ses propres caractéristiques : un disque dur possède une marque et une capacité, un livre possède un nombre de pages et un nom d'auteur, etc.

Nous devrions donc créer une classe fille par type de produit, mais nous ne voulons pas perdre le formatage apporté par les classes `EUProduct` et `USProduct`.

Pour un type de produit, nous serions donc obligés de créer une classe par formatage. Nous aurions ainsi un `EUBook` et un `EUHardDrive` (qui héritent de `EUProduct`), et un `USBook` et un `USHardDrive` (qui héritent de `USProduct`), chaque classe de type étant un duplicata des autres.

Cela complexifierait inutilement le code et créerait de la duplication, qui rendrait l'application très difficile à maintenir et à faire évoluer.

Le problème principal vient du fait que nous avons intégré à la classe `Product` une logique qui ne la concernait pas directement : le formatage du prix.

Pour résoudre ce problème, nous pouvons traiter l'affichage dans d'autres classes spécialisées que nous allons passer à nos produits. Créons une interface `PriceFormatter` et ses classes filles dérivées :

```
1 <?php
2
3 interface PriceFormatter
4 {
5     // Tous nos PriceFormatter devront implémenter cette méthode. Cela nous permet de
    // l'utiliser dans nos produits.
6     public function format(float $price): string;
7 }
8
9 // On remarque l'utilisation de classes finales
10 final class EUFormatter implements PriceFormatter
11 {
12     public function format(float $price): string
13     {
14         return $price.' €';
15     }
16 }
17
18 final class USFormatter implements PriceFormatter
19 {
20     public function format(float $price): string
21     {
22         return '$'.$price;
23     }
24 }
```

À partir de là, il nous suffit maintenant de composer nos produits avec le bon `PriceFormatter`. Créons donc une propriété de type `PriceFormatter`, qui sera injectée dans le constructeur :

```
1 <?php
2
3 require_once 'PriceFormatter.php';
4
5 class Product
6 {
7     private string $name;
8     private float $price;
9     // On déclare une propriété de type PriceFormatter
10    private PriceFormatter $formatter;
11
12    public function __construct(string $name, float $price, PriceFormatter $formatter)
13    {
14        $this->name = $name;
15        $this->price = $price;
16        // On injecte notre PriceFormatter
17        $this->formatter = $formatter;
18    }
19 }
```

```

19
20 public function getName(): string
21 {
22     return $this->name;
23 }
24
25 public function setName(string $name): void
26 {
27     $this->name = $name;
28 }
29
30 public function getPrice(): float
31 {
32     return $this->price;
33 }
34
35 public function setPrice(float $price): void
36 {
37     $this->price = $price;
38 }
39
40 public function formatPrice(): string
41 {
42     // Nous faisons appel a notre PriceFormatter pour formater notre prix. Ce n'est plus
43     // utile de créer des classes filles.
44     return $this->formatter->format($this->getPrice());
45 }
46
47 $hardDrive = new Product('Disque dur', 140.00, new EUFormatter());
48 $UShardDrive = new Product('Hard Drive', 140.00, new USFormatter());
49
50 echo $hardDrive->formatPrice();
51 echo '<br>';
52 echo $UShardDrive->formatPrice();

```

Nous pouvons maintenant créer nos types de produits sous forme de sous-classes sans problème.

### Quand utiliser la composition ?

Il serait possible d'utiliser la composition pour les types de produits : créer une interface `ProductType` et des sous-classes `Book` et `HardDrive` que l'on injecterait dans notre produit. Ainsi, la classe `Product` resterait sans classe fille (donc serait une classe finale) et toute la logique serait séparée dans d'autres sous-classes plus petites.

Cependant, cela peut paraître moins pertinent : les livres et les disques durs **sont** des produits, il n'est pas illogique d'avoir des classes dérivées.

Pour déterminer s'il est préférable d'utiliser la composition ou l'héritage, une solution simple serait de se poser la question : « Est-ce que je suis susceptible d'utiliser ce bout de code dans un autre contexte ? ».

Dans notre cas, on peut facilement imaginer des cas où on aurait besoin de formater des prix (pour des frais de port, par exemple, qui ne seraient pas des produits à proprement parler), mais moins où on aurait besoin uniquement du nombre de pages et de l'auteur d'un livre.

Bien évidemment, chaque cas dépend de l'application que vous souhaitez réaliser : c'est avec l'expérience et la pratique que des concepts comme "Composition over inheritance" prennent tout leur sens.

### Fondamental

Une bonne pratique de développement est de toujours rendre finales les classes qui implémentent une interface. Ainsi, la structure des classes d'un projet doit être, dans la plupart des cas : `interface > classes abstraites (facultatives) > classes filles finales`. Cela permet d'éviter d'avoir trop de sous-classes et de rendre le code trop complexe.

De plus, si l'héritage possède l'avantage de centraliser du code, il peut être dangereux de faire dépendre trop de classes d'une même base. Cela signifie que, si la base est modifiée, alors **il faudra tester l'intégralité de nos classes filles** pour s'assurer qu'il n'y a pas d'effets indésirables.

La composition permet d'éviter ce problème en séparant notre code en petites "briques" que l'on assemble au besoin, mais que l'on pourra tester séparément.

### Syntaxe À retenir

- Les **classes finales** sont des classes qui ne peuvent pas avoir de classes filles.
- Il est préférable d'utiliser la **composition** plutôt que l'héritage pour garder une structure simple et tester plus facilement notre code.
- Les classes finales permettent ainsi de contrôler l'évolution de notre projet.

### Complément

Un article de Marco "Ocranius" Pivetta, un des développeurs de Doctrine, sur les classes finales<sup>1</sup>

Documentation officielle<sup>2</sup>

## IX. Exercice : Appliquez la notion

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



### Question 1

[solution n°4 p.27]

Un musée est en charge de peintures et de sculptures. Ces deux œuvres ont des caractéristiques communes : un titre, et le nom de l'artiste qui l'a réalisée. Cependant, les peintures ont aussi un type (aquarelle, peinture à l'huile, gouache...). Les sculptures, quant à elles, ont une hauteur en centimètres.

Le logiciel actuel du musée permet déjà de gérer la base des informations avec la classe abstraite suivante :

```
1 <?php
2
3 abstract class Artwork
4 {
5     private string $title;
6     private string $artist;
7
8     public function __construct(string $title, string $artist)
9     {
```

1 <https://ocranius.github.io/blog/when-to-declare-classes-final/>

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

3 <https://repl.it/>

```

10     $this->title = $title;
11     $this->artist = $artist;
12 }
13
14 public function getTitle(): string
15 {
16     return $this->title;
17 }
18
19 public function setTitle(string $title): void
20 {
21     $this->title = $title;
22 }
23
24 public function getArtist(): string
25 {
26     return $this->artist;
27 }
28
29 public function setArtist(string $artist): void
30 {
31     $this->artist = $artist;
32 }
33 }

```

Implémentez cette classe abstraite en créant deux classes finales représentant les peintures et les sculptures.

## Question 2

[solution n°5 p.28]

Les statues sont posées sur des socles, ce qui augmente leur taille finale. Le musée dispose de deux types de socles : des socles fixes de 10 cm et des socles extensibles, qui peuvent avoir une taille entre 5 et 15 cm, avec une position de base à 5 cm.

Intégrez les socles dans votre code pour calculer correctement la taille de la statue dans son accesseur.

Attention : normalement, votre classe représentant les sculptures est une classe finale. Interdiction donc de créer des classes filles `SculptureWithFixedPedestal` et `SculptureWithVariablePedestal` pour gérer ces nouveaux cas !

### Indice :

Il va falloir utiliser la composition : créez une interface `Pedestal` qui sera implémentée par deux autres classes, et ajoutez une propriété de type `Pedestal` à vos sculptures.

## X. Auto-évaluation

### A. Exercice final

#### Exercice 1

[solution n°6 p.29]

Exercice

Qu'est-ce que l'encapsulation ?

- ☐ Le fait de centraliser au même endroit les données et les actions à réaliser dessus
- ☐ Le fait de masquer certaines propriétés et méthodes
- ☐ Le fait d'utiliser des méthodes pour accéder aux propriétés

Exercice

Qu'est-ce qu'un accesseur ?

- ☐ Une méthode permettant de modifier une propriété
- ☐ Une méthode permettant de récupérer la valeur d'une propriété
- ☐ Une méthode permettant d'initialiser les valeurs de nos propriétés

Exercice

Qu'est-ce que la visibilité privée ?

- ☐ Le fait d'empêcher l'instanciation d'une classe
- ☐ Le fait de dresser la liste des méthodes qui devront être implémentées dans une classe
- ☐ Le fait d'empêcher l'utilisation de propriétés ou de méthodes en dehors de la classe

Exercice

Quels sont les différents niveaux de visibilité ?

- ☐ Public
- ☐ Réservé
- ☐ Protégé
- ☐ Abstrait
- ☐ Disponible
- ☐ Privé
- ☐ Dupliqué
- ☐ Interne

Exercice

Pourquoi utiliser des méthodes pour manipuler nos propriétés ?

- ☐ Pour pouvoir gérer la visibilité de la lecture et de l'écriture indépendamment
- ☐ Appeler une méthode est plus rapide en termes de performance que de manipuler la propriété
- ☐ Pour pouvoir effectuer des vérifications sur les données avant de les utiliser
- ☐ Un nom de méthode est plus lisible qu'un nom de propriété

Exercice

Qu'est-ce qu'une classe finale ?

- ☐ Une classe qui ne peut pas avoir de classes filles
- ☐ Une classe qui ne peut être instanciée qu'une seule fois
- ☐ Une classe qui ne peut pas être instanciée

Exercice

À quoi correspond la visibilité protégée ?

- ☐ Visible uniquement depuis la classe en cours
- ☐ Visible uniquement depuis la classe en cours ou les classes filles
- ☐ Visible depuis tout le script

Exercice

Qu'est-ce que la composition ?

- ☐ Le fait d'utiliser des classes finales
- ☐ Le fait d'utiliser un gestionnaire de dépendances
- ☐ Le fait d'avoir des objets spécialisés dans un seul traitement et de construire nos classes à partir de ces objets

Exercice

La notion d'héritage ne sert à rien si on utilise la composition.

- ☐ Vrai
- ☐ Faux

Exercice

Quel est le contraire d'une classe finale ?

- ☐ Une classe avec uniquement des méthodes publiques
- ☐ Une classe mère
- ☐ Une classe abstraite

## B. Exercice : Défi

Dans ce défi, vous allez devoir utiliser vos nouvelles connaissances pour créer le code d'une application de gestion immobilière.

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



### Question 1

[solution n°7 p.31]

Une agence immobilière a besoin d'une application pour gérer ses biens immobiliers. Il existe deux types de biens immobiliers : les appartements et les maisons. Un appartement possède une adresse, un étage (0 pour rez-de-chaussée, 1 pour premier étage, etc.), un prix et une surface habitable. Une maison, quant à elle, possède une adresse, une surface habitable, un prix et un nombre de niveaux (1 pour plain-pied, 2 s'il y a un étage, etc.). Le nombre de niveaux doit toujours être strictement positif (en cas de donnée invalide, on peut le mettre à 1), tandis que le numéro de l'étage peut être négatif (pour les appartements en sous-sol).

L'adresse et la surface ne doivent pas pouvoir être modifiées une fois l'objet créé. En revanche, le prix peut être négocié.

Écrivez le code de cette application en utilisant une structure de code facilement maintenable.

---

1 <https://repl.it/>

**Question 2**

[solution n°8 p.33]

Chaque bien immobilier peut être associé à une annexe, mais ce n'est pas obligatoire. Il existe plusieurs types d'annexes : les jardins, et les places de parking. Les deux possèdent une surface qui ne doit pas être modifiable, mais seule celle des jardins doit être comprise dans la surface habitable. Les jardins possèdent également une caractéristique permettant de définir si une piscine est présente ou non, tandis que les stationnements ont un numéro de place.

Utilisez la composition pour ajouter des informations dans votre application immobilière.

**Indice :**

Pour dire qu'un type peut être `null`, il faut utiliser le symbole "?" devant le nom du type. Par exemple, `?int` signifie : "un entier ou null".

**Indice :**

Il est possible de spécifier une valeur par défaut dans la signature de fonction en utilisant le symbole "=" suivi de la valeur.

**Solutions des exercices**



## p. 6 Solution n°1

```

1 <?php
2
3
4 class Text
5 {
6     public string $text;
7
8     public function __construct(string $text)
9     {
10         $this->text = $text;
11     }
12
13     public function getSentences(): array
14     {
15         $sentences = explode('.', $this->text);
16         array_pop($sentences); //On enlève le dernier élément, toujours vide
17
18         return $sentences;
19     }
20
21     public function getTextLength(): int
22     {
23         return strlen($this->text);
24     }
25
26     public function getSentencesCount(): int
27     {
28         return count($this->getSentences());
29     }
30 }
31
32 $text = new Text('Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec lacinia quam
    sed lacinia blandit. In et viverra elit. Nullam molestie quam eget porta venenatis. Aenean
    blandit auctor turpis, eu rhoncus libero pulvinar id. Aenean euismod enim ac sagittis
    accumsan. Fusce venenatis purus orci, in euismod erat porttitor in. Morbi semper dignissim
    felis a tincidunt.');
```

```

33 print_r($text->getSentences());
34 echo 'Le texte possède ' . $text->getSentencesCount() . ' phrases et ' . $text->getTextLength() . '
    caractères.';

```

## p. 10 Solution n°2

```

1 <?php
2
3
4 class Patient
5 {
6     private int $id;
7     protected string $firstName;
8     protected string $lastName;
9
10    public function __construct(int $id)
11    {
12        $this->loadPatient($id);
13    }

```

```

14
15     private function loadPatient(int $id)
16     {
17         // Nous simulons l'appel à une base de données en créant un tableau en dur
18         $data = [
19             'id' => $id,
20             'firstName' => 'John',
21             'lastName' => 'Doe',
22         ];
23
24         $this->id = $data['id'];
25         $this->firstName = $data['firstName'];
26         $this->lastName = $data['lastName'];
27     }
28
29     public function getFullName()
30     {
31         return $this->firstName.' '.$this->lastName;
32     }
33 }

```

p. 15 Solution n°3

```

1 <?php
2
3
4 class Meal
5 {
6     private string $name;
7     private array $ingredients;
8     protected float $price;
9
10    public function __construct(string $name, float $price)
11    {
12        $this->setName($name);
13        $this->setPrice($price);
14        $this->ingredients = [];
15    }
16
17    public function getName(): string
18    {
19        return $this->name;
20    }
21
22    public function setName(string $name): void
23    {
24        $this->name = $name;
25    }
26
27    public function getIngredients(): array
28    {
29        return $this->ingredients;
30    }
31
32    public function setIngredients(array $ingredients): void
33    {
34        $this->ingredients = $ingredients;

```

```

35     }
36
37     public function getPrice(): float
38     {
39         return $this->price + count($this->ingredients);
40     }
41
42     public function setPrice(float $price): void
43     {
44         $this->price = $price;
45     }
46 }
47
48 // Exemple d'utilisation :
49
50 $meal = new Meal('Salade Caesar', 10.50);
51 $meal->setIngredients(['Olives', 'Cheddar']);
52 echo $meal->getPrice(); // Affiche 12.5

```

#### p. 20 Solution n°4

```

1 <?php
2
3 final class Painting extends Artwork
4 {
5     private string $type;
6
7     public function __construct(string $title, string $artist, string $type)
8     {
9         $this->type = $type;
10        parent::__construct($title, $artist);
11    }
12
13    public function getType(): string
14    {
15        return $this->type;
16    }
17
18    public function setType(string $type): void
19    {
20        $this->type = $type;
21    }
22 }
23
24 final class Sculpture extends Artwork
25 {
26     private int $height;
27
28     public function __construct(string $title, string $artist, int $height)
29     {
30         $this->height = $height;
31         parent::__construct($title, $artist);
32     }
33
34     public function getHeight(): int
35     {
36         return $this->height;

```

```

37     }
38
39     public function setHeight(int $height): void
40     {
41         $this->height = $height;
42     }
43 }

```

**p. 21 Solution n°5**

```

1 <?php
2
3 interface Pedestal
4 {
5     public function getPedestalHeight(): int;
6 }
7
8 class FixedPedestal implements Pedestal
9 {
10     public function getPedestalHeight(): int
11     {
12         return 10;
13     }
14 }
15
16 class VariablePedestal implements Pedestal
17 {
18     private int $height = 5;
19
20     public function __construct(int $height)
21     {
22         $this->setHeight($height);
23     }
24
25     public function getHeight(): int
26     {
27         return $this->height;
28     }
29
30     public function setHeight(int $height): void
31     {
32         if ($height < 15 && $height > 5) {
33             $this->height = $height;
34         }
35     }
36
37     public function getPedestalHeight(): int
38     {
39         return $this->getHeight();
40     }
41 }
42
43 final class Sculpture extends Artwork
44 {
45     private int $height;
46     private Pedestal $pedestal;
47

```

```


48     public function __construct(string $title, string $artist, int $height, Pedestal
    $pedestal)
49     {
50         $this->height = $height;
51         $this->pedestal = $pedestal;
52         parent::__construct($title, $artist);
53     }
54
55     public function getHeight(): int
56     {
57         return $this->height + $this->pedestal->getPedestalHeight();
58     }
59
60     public function setHeight(int $height): void
61     {
62         $this->height = $height;
63     }
64 }

```

### Exercice p. 21 Solution n°6


#### Exercice

Qu'est-ce que l'encapsulation ?

- ☒ Le fait de centraliser au même endroit les données et les actions à réaliser dessus
- ☐ Le fait de masquer certaines propriétés et méthodes
- ☐ Le fait d'utiliser des méthodes pour accéder aux propriétés
-  L'encapsulation est le fait de regrouper, dans une seule classe, des données et les actions à réaliser dessus.

#### Exercice

Qu'est-ce qu'un accesseur ?

- ☐ Une méthode permettant de modifier une propriété  
*C'est un mutateur.*
- ☒ Une méthode permettant de récupérer la valeur d'une propriété
- ☐ Une méthode permettant d'initialiser les valeurs de nos propriétés  
*C'est un constructeur.*
-  Un accesseur est une méthode permettant de récupérer la valeur d'une propriété.

#### Exercice

Qu'est-ce que la visibilité privée ?

- ☐ Le fait d'empêcher l'instanciation d'une classe  
*Ce sont les classes abstraites.*
- ☐ Le fait de dresser la liste des méthodes qui devront être implémentées dans une classe  
*On parle plutôt d'interface.*

- ⦿ Le fait d'empêcher l'utilisation de propriétés ou de méthodes en dehors de la classe
- 🔍 La visibilité permet de masquer les méthodes et propriétés pour empêcher leur utilisation hors de la classe.

### Exercice

Quels sont les différents niveaux de visibilité ?

- ☒ Public
- ☐ Réservé
- ☒ Protégé
- ☐ Abstrait
- ☐ Disponible
- ☒ Privé
- ☐ Dupliqué
- ☐ Interne

🔍 Il existe trois niveaux de visibilité : public, protégé et privé.

### Exercice

Pourquoi utiliser des méthodes pour manipuler nos propriétés ?

- ☒ Pour pouvoir gérer la visibilité de la lecture et de l'écriture indépendamment
  - ☐ Appeler une méthode est plus rapide en termes de performance que de manipuler la propriété
  - ☒ Pour pouvoir effectuer des vérifications sur les données avant de les utiliser
  - ☐ Un nom de méthode est plus lisible qu'un nom de propriété
- 🔍 Les accesseurs et les mutateurs permettent de gérer plus finement la visibilité des attributs en nous permettant de séparer la lecture et l'écriture de nos propriétés. Ils permettent aussi de vérifier une valeur avant de la stocker dans une propriété.

### Exercice


Qu'est-ce qu'une classe finale ?

- ⦿ Une classe qui ne peut pas avoir de classes filles
  - Une classe qui ne peut être instanciée qu'une seule fois
  - Une classe qui ne peut pas être instanciée
- 🔍 Une classe finale est une classe qui ne peut pas être étendue, donc qui ne peut pas avoir de classe fille.

### Exercice


À quoi correspond la visibilité protégée ?

- Visible uniquement depuis la classe en cours
- C'est la visibilité privée.*

- ☒ Visible uniquement depuis la classe en cours ou les classes filles
- ☐ Visible depuis tout le script  
*C'est la visibilité publique.*
-  La visibilité protégée permet de rendre une propriété ou une méthode visible uniquement depuis la classe en cours ou une de ses sous-classes.


### Exercice

Qu'est-ce que la composition ?

- ☐ Le fait d'utiliser des classes finales
- ☐ Le fait d'utiliser un gestionnaire de dépendances
- ☒ Le fait d'avoir des objets spécialisés dans un seul traitement et de construire nos classes à partir de ces objets
-  Grâce à la composition, notre classe est **composée** d'autres objets, spécialisés dans un traitement particulier.


### Exercice

La notion d'héritage ne sert à rien si on utilise la composition.

- ☐ Vrai
- ☒ Faux
-  Les notions d'héritage et de composition ne sont pas incompatibles : les éléments composant une classe implémentent souvent une interface, et peuvent parfois utiliser des classes abstraites pour centraliser du code. La composition permet d'éviter l'**abus d'héritage**, pas de proposer une alternative.

### Exercice

Quel est le contraire d'une classe finale ?

- ☐ Une classe avec uniquement des méthodes publiques
- ☐ Une classe mère
- ☒ Une classe abstraite
-  Une classe finale est une classe qui ne peut pas avoir de classe fille. Or, une classe abstraite est une classe qui doit avoir une classe fille. Ces deux notions sont donc complètement opposées. Il n'est d'ailleurs pas possible en PHP de créer une classe `abstract final`.

### p. 23 Solution n°7

```
1 <?php
2
3 abstract class RealEstate {
4     private string $address;
5     private float $price;
6     private float $surface;
7
8     public function __construct(string $address, float $price, float $surface)
9     {
10         $this->address = $address;
11         $this->price = $price;
12         $this->surface = $surface;
```

```

13     }
14
15     public function getAddress(): string
16     {
17         return $this->address;
18     }
19
20     public function setAddress(string $address): void
21     {
22         $this->address = $address;
23     }
24
25     public function getPrice(): float
26     {
27         return $this->price;
28     }
29
30     public function setPrice(float $price): void
31     {
32         $this->price = $price;
33     }
34
35     public function getSurface(): float
36     {
37         return $this->surface;
38     }
39
40     public function setSurface(float $surface): void
41     {
42         $this->surface = $surface;
43     }
44 }
45
46 final class House extends RealEstate
47 {
48     private int $levelCount;
49
50     public function __construct(string $address, float $price, float $surface, int
51     $levelCount)
52     {
53         parent::__construct($address, $price, $surface);
54         $this->setLevelCount($levelCount);
55     }
56
57     public function getLevelCount(): int
58     {
59         return $this->levelCount;
60     }
61
62     public function setLevelCount(int $levelCount): void
63     {
64         if ($levelCount > 0) {
65             $this->levelCount = $levelCount;
66         } else {
67             $this->levelCount = 1;
68         }
69     }

```



```

70
71 final class Flat extends RealEstate
72 {
73     private int $floor;
74
75     public function __construct(string $address, float $price, float $surface, int $floor)
76     {
77         parent::__construct($address, $price, $surface);
78         $this->floor = $floor;
79     }
80
81     public function getFloor(): int
82     {
83         return $this->floor;
84     }
85
86     public function setFloor(int $floor): void
87     {
88         $this->floor = $floor;
89     }
90 }
91
92 $house = new House('102 rue des noyers', 350000.00, 120, 2);
93 echo $house->getAddress().'\n';
94 $flat = new Flat('45 rue de la république', 150000, 30, 3);
95 echo $flat->getFloor();

```

## p. 24 Solution n°8

```

1 <?php
2
3 abstract class RealEstate {
4     private string $address;
5     private float $price;
6     private float $surface;
7     private ?Annex $annex;
8
9     public function __construct(string $address, float $price, float $surface, ?Annex $annex
= null)
10     {
11         $this->setAddress($address);
12         $this->setPrice($price);
13         $this->setSurface($surface);
14         $this->setAnnex($annex);
15     }
16
17     public function getAddress(): string
18     {
19         return $this->address;
20     }
21
22     public function setAddress(string $address): void
23     {
24         $this->address = $address;
25     }
26
27     public function getPrice(): float

```

```

28     {
29         return $this->price;
30     }
31
32     public function setPrice(float $price): void
33     {
34         $this->price = $price;
35     }
36
37     public function getSurface(): float
38     {
39         if ($this->annex !== null) {
40             return $this->surface + $this->annex->getAdditionalSurface();
41         }
42
43         return $this->surface;
44     }
45
46     public function setSurface(float $surface): void
47     {
48         $this->surface = $surface;
49     }
50
51     public function getAnnex(): ?Annex
52     {
53         return $this->annex;
54     }
55
56     public function setAnnex(?Annex $annex): void
57     {
58         $this->annex = $annex;
59     }
60 }
61
62 final class House extends RealEstate
63 {
64     private int $levelCount;
65
66     public function __construct(string $address, float $price, float $surface, int
67 $levelCount, ?Annex $annex = null)
68     {
69         parent::__construct($address, $price, $surface, $annex);
70         $this->setLevelCount($levelCount);
71     }
72
73     public function getLevelCount(): int
74     {
75         return $this->levelCount;
76     }
77
78     public function setLevelCount(int $levelCount): void
79     {
80         if ($levelCount > 0) {
81             $this->levelCount = $levelCount;
82         } else {
83             $this->levelCount = 1;
84         }
85     }
86 }

```

```

85 }
86
87 final class Flat extends RealEstate
88 {
89     private int $floor;
90
91     public function __construct(string $address, float $price, float $surface, int $floor, ?
Annex $annex = null)
92     {
93         parent::__construct($address, $price, $surface, $annex);
94         $this->setFloor($floor);
95     }
96
97     public function getFloor(): int
98     {
99         return $this->floor;
100     }
101
102     public function setFloor(int $floor): void
103     {
104         $this->floor = $floor;
105     }
106 }
107
108 interface AnnexInterface
109 {
110     public function getAdditionalSurface(): float;
111 }
112
113 abstract class Annex implements AnnexInterface
114 {
115     private float $surface;
116
117     public function __construct(float $surface)
118     {
119         $this->setSurface($surface);
120     }
121
122     public function getSurface(): float
123     {
124         return $this->surface;
125     }
126
127     public function setSurface(float $surface): void
128     {
129         $this->surface = $surface;
130     }
131 }
132
133 final class Garden extends Annex
134 {
135     private bool $hasPool;
136
137     public function __construct(float $surface, bool $hasPool)
138     {
139         parent::__construct($surface);
140         $this->setPool($hasPool);
141     }

```

```

142
143     public function hasPool(): bool
144     {
145         return $this->hasPool;
146     }
147
148     public function setPool(bool $hasPool): void
149     {
150         $this->hasPool = $hasPool;
151     }
152
153     public function getAdditionalSurface(): float
154     {
155         return $this->getSurface();
156     }
157 }
158
159 final class Parking extends Annex
160 {
161     private int $number;
162
163     public function __construct(float $surface, int $number)
164     {
165         parent::__construct($surface);
166         $this->setNumber($number);
167     }
168
169     public function getNumber(): int
170     {
171         return $this->number;
172     }
173
174     public function setNumber(int $number): void
175     {
176         $this->number = $number;
177     }
178
179     public function getAdditionalSurface(): float
180     {
181         return 0;
182     }
183 }
184
185 $garden = new Garden(20, true);
186 $parking = new Parking(6, 2);
187 $house = new House('102 rue des noyers', 350000.00, 120, 2);
188 echo $house->getSurface().'\n'; // Affiche 120, car il n'y a pas d'annexe
189 $house->setAnnex($garden);
190 echo $house->getSurface().'\n'; // Affiche 140 car le jardin est compris dans l'annexe
191 $flat = new Flat('45 rue de la république', 150000, 30, 3, $parking);
192 echo $flat->getSurface().'\n'; // Affiche 30 car le parking n'est pas compris dans le
    calcul

```