

La programmation Orientée Objet : concepts de base

Table des matières

I. Contexte	3
II. Classes et objets	3
A. Classes et objets	3
B. Exercice : Quiz	5
III. Instanciation, constructeurs, accesseurs et mutateurs	6
A. Instanciation, constructeurs, accesseurs et mutateurs	6
B. Exercice : Quiz	10
IV. Essentiel	11
V. Auto-évaluation	11
A. Exercice	11
B. Test	12
Solutions des exercices	13

I. Contexte

Durée : 60 minutes

Prérequis : les bases en PHP

Environnement de travail : PHP 8.2.5, onlinePHP.io

Contexte

Les classes et les objets sont deux concepts au centre de la programmation orientée objet. Ensemble, ils nous permettent de représenter des données de manière instinctive, et d'optimiser le traitement. Il est donc important de maîtriser ces deux concepts.

Aujourd'hui, la programmation orientée objet est un incontournable de la programmation, certains langages, tels que C# ou Java, sont construits autour de ce paradigme tandis que d'autres ont introduit ce paradigme plus tard, c'est le cas de PHP.

Pour bien comprendre la programmation orientée objet, nous allons aborder les différences entre une classe et un objet, ainsi que des principes de fonctionnement, tels que l'instanciation, les *getters* et *setters*, ainsi que les méthodes.

Pour ce faire nous allons utiliser un exemple concret que nous allons représenter à l'aide de la programmation orientée objet.

II. Classes et objets

A. Classes et objets

Définition

Les objets

Qu'est-ce qu'un objet ? Cela peut paraître une question simple, mais il est important de pouvoir définir précisément ce qu'est un objet, si l'on veut pouvoir le représenter.

Prenons par exemple une voiture, qu'est-ce qui compose cet objet ?

- Des attributs : une voiture, comme tout autre objet, possède différents attributs (nom du modèle, marque, puissance, consommation, etc.).
- Son fonctionnement : chaque pièce dans la voiture a un fonctionnement précis, pris ensemble, ils permettent à la voiture de se comporter comme prévu par le créateur (les freins freinent, l'accélérateur accélère). On peut associer cela aux méthodes composant notre objet.
- Un constructeur : chaque voiture est construite de manière à répondre à des critères prédéfinis, dans le cas de la programmation orientée objet, c'est le rôle du constructeur.
- Un modèle : les voitures, aujourd'hui, sont produites en masse, à partir d'un modèle ou d'un schéma. Dans notre cas, ce rôle est rempli par la classe.

Un objet est donc composé de multiples attributs, son fonctionnement est décrit par les différentes méthodes le composant, cet objet sera construit par notre constructeur en utilisant le modèle défini par notre classe.

Les classes

Comme nous avons pu le voir, une classe est un modèle, et elle va nous permettre de définir le comportement de notre objet. La première étape est de définir les attributs de notre objet, en PHP une classe, et ses attributs sont définis comme ceci :

```
1 class Voiture
2 {
3     public string $modele;
4     public string $marque;
5     public int $vitesse = 0;
6 }
```

Ici, nous avons une classe **Voiture**, avec 3 attributs :

- Modèle de type chaîne de caractères
- Marque de type chaîne de caractères
- Vitesse de type entier

Il y a 4 parties dans la déclaration d'un attribut :

- La visibilité : **public**, **protected**, **private**. Un attribut « *public* » peut être accessible et modifié de n'importe où, c'est-à-dire à l'intérieur ou à l'extérieur de la classe. Un attribut « *protected* » ne peut être accessible qu'à l'intérieur de la classe où il est défini ou dans une classe qui l'hérite, tandis qu'un attribut « *private* » ne peut être accessible que dans la classe où il a été déclaré.
- Le type : il est possible de fortement typer les attributs d'une classe.
- Le nom de l'attribut : tout comme une variable, il est nécessaire de nommer l'attribut en suivant la syntaxe \$nomDeLattribut.
- L'initialisation (optionnel) : toujours comme une variable, il est possible d'assigner une valeur par défaut à notre attribut, ici la vitesse de notre voiture est mise à 0 par défaut.

Attention Syntaxe

Dans notre exemple, nos attributs sont typés, mais cela est optionnel.

Typer fortement ses attributs a le mérite d'augmenter la lisibilité du code, et donc sa maintenabilité, mais aussi de facilement détecter les erreurs de type.

Dans le cas où l'on souhaite déclarer cette classe de manière non typée, il est possible de le faire de cette manière :

```
1 class Voiture
2 {
3     public $modele;
4     public $marque;
5     public $vitesse = 0;
6 }
```

Cette vidéo montre la différence de comportement entre une classe fortement typée, et une classe non typée :

Les méthodes

Comme nous l'avons vu plus tôt, les méthodes définissent le fonctionnement de notre objet. À l'heure actuelle, notre classe n'a aucune méthode définie, nous allons donc définir une méthode « *accélérer* » qui changera la vitesse de notre voiture.

```
1 class Voiture
2 {
3     public string $modele;
4     public string $marque;
5     public int $vitesse = 0;
6
7     public function accélérer(int $vitesse){
8         $this->vitesse += $vitesse;
9     }
10 }
```

Nous avons donc déclaré une fonction **Accélérer()** qui prend un paramètre **\$vitesse**. Lorsque nous appelons cette méthode, elle ajoutera la vitesse que nous passons en paramètre à la vitesse de notre voiture.

La déclaration d'une fonction se fait en 4 étapes :

- Visibilité : tout comme les attributs, cela permet de définir le degré d'accessibilité de la méthode.
- Mot clé **function** : il est nécessaire d'utiliser ce mot clé pour laisser savoir à PHP que nous sommes en train de déclarer une méthode, et non un attribut.
- Nom de la méthode : contrairement aux attributs ou variables, le nom d'une méthode ne nécessite pas de « \$ ».
- Paramètre : dans le cas où notre méthode prend des paramètres (comme dans notre exemple ci-dessus, où nous avons un paramètre « \$vitesse »), nous précisons ces paramètres dans la parenthèse. Dans le cas où nous n'avons pas de paramètre, il suffit de laisser les parenthèses vides.
- Le bloc de code : placés entre « {} » il s'agit des actions effectuées par notre méthode, dans notre exemple nous ajoutons la vitesse demandée à la vitesse de notre voiture.

Maintenant que nous avons défini notre méthode, il faut donc l'appeler, mais pour cela il est nécessaire d'avoir un objet sur lequel travailler, c'est là qu'intervient le concept d'instanciation.

B. Exercice : Quiz

[solution n°1 p.15]

Question 1

Une méthode sert à :

- ☐ Définir les attributs d'un objet
- ☐ Définir comment l'objet doit être utilisé
- ☐ Définir le fonctionnement de l'objet

Question 2

Quelle est la différence entre un objet et une classe ?

- ☐ Aucune, ce sont des synonymes
- ☐ On parle d'objet dans le monde réel, et de classe en programmation
- ☐ Une classe est un modèle, à partir duquel on crée des objets

Question 3

Quelle visibilité empêche toute modification d'un attribut en dehors de la classe dans laquelle il est déclaré ?

- ☐ Private
- ☐ Public
- ☐ Protected

Question 4

Laquelle de ces déclarations d'attribut est correcte ?

- ☐ string \$modele;
- ☐ public string \$modele;
- ☐ public string modele;

Question 5

Laquelle de ces déclarations de fonction est correcte ?

- ☐ public function accelerer(int \$vitesse)
- ☐ public function accelerer()
- ☐ Les deux

III. Instanciation, constructeurs, accesseurs et mutateurs

A. Instanciation, constructeurs, accesseurs et mutateurs

Comme nous l'avons vu plus tôt, une classe est un modèle à partir duquel nous pouvons créer un objet. En effet notre classe seule ne peut pas faire grand-chose, mais si nous utilisons cette classe pour créer un objet, nous obtenons quelque chose de tangible dont nous pouvons manipuler les données. Ce processus est ce que l'on appelle « instancier un objet ».

Instanciation

Nous allons donc utiliser la classe Voiture que nous avons créée précédemment. Pour rappel, la classe Voiture est composée ainsi :

```
1 class Voiture
2 {
3     public string $modele;
4     public string $marque;
5     public int $vitesse = 0;
6
7     public function accelerer(int $vitesse){
8         $this->vitesse += $vitesse;
9     }
10 }
```

À l'heure actuelle, notre classe, ses attributs et sa méthode « *accélérer* » sont définis, mais ne sont pas utilisés. Nous allons donc instancier notre objet, pour ce faire nous allons utiliser le mot clé « **new** », comme dans l'exemple ci-dessous :

```
1 $maVoiture = new Voiture();
```

Dans cet exemple de code, nous avons :

- Un objet nommé maVoiture.
- Le mot clé « *new* » suivi du nom de notre classe, c'est ce qui permet à PHP de savoir que notre variable maVoiture est un objet de type Voiture.

Maintenant que nous avons un objet sur lequel nous pouvons travailler, nous pouvons modifier les données de cet objet. Modifions la marque de notre voiture :

```
1 $maVoiture->marque = "Mercedes";
```

Dans notre exemple, nous avons attribué la chaîne de caractères « *Mercedes* » à l'attribut `marque` de notre voiture. Mais ce n'est pas tout, nous avons aussi créé une méthode accélérée, qui nous permet d'ajouter une valeur entière à notre vitesse, qui est de 0 par défaut, essayons de l'appeler :

```
1 $maVoiture->accelerer(10);
```

Cet exemple va donc rajouter « 10 » à la valeur de notre vitesse.

Comme nous pouvons le voir dans les deux précédents exemples, lorsque nous essayons d'accéder à quelque chose qui est interne à notre objet, que ça soit un attribut ou une méthode, nous utilisons « `->` », ainsi écrire `$maVoiture->marque` revient à dire « *la marque de mon objet maVoiture* ».

Constructeurs

À l'heure actuelle, lorsque nous créons un objet `Voiture`, les attributs de cet objet sont laissés par défaut. Dans le cadre de notre vitesse, il s'agit de 0, mais ni la marque, ni le modèle ne sont renseignés par défaut. Or, toute voiture possède une marque et un modèle, mais ceux-ci diffèrent de voiture en voiture.

Pour répondre à ce problème, il existe ce que l'on appelle les constructeurs.

Définition Les constructeurs

Qu'est-ce qu'un constructeur ? En programmation orientée objet, les constructeurs sont tout simplement des méthodes. Ces méthodes sont appelées automatiquement à la création d'un objet. C'est-à-dire que lorsque nous utilisons le mot clé « *new* », nous faisons appel à un constructeur.

Nous allons donc définir un constructeur, qui nous permettra de définir une marque et un modèle à chaque initialisation d'objet.

La syntaxe de déclaration d'un constructeur est très proche de celle d'une méthode, il suffit de déclarer une méthode dans notre classe qui sera nommée `__construct`, comme ceci :

```
1 public function __construct()  
2 {  
3 }
```

Cette différence de syntaxe est due au fait que le constructeur est une méthode dite « *magique* » et qu'elle possède un comportement différent, implicite, des méthodes que vous créerez. Comme ici le fait qu'elle soit appelée à chaque fois qu'un objet est créé via `new Object()`. Il existe une multitude d'autres méthodes magiques.

Nous avons donc notre constructeur défini, sauf que celui-ci ne fait actuellement rien. Comme nous l'avons établi précédemment, nous avons besoin de ce constructeur pour définir la marque et le modèle de notre voiture à l'initialisation. Pour ce faire, nous allons donc lui ajouter deux paramètres :

- `$marque`
- `$modele`

Ce qui nous donne ceci :

```
1 public function __construct(string $marque, string $modele)  
2 {  
3 }
```

Nous avons désormais deux paramètres définis. Maintenant, il ne reste qu'à dire au constructeur ce qu'il doit faire de ces données. Dans notre cas c'est assez simple, nous souhaitons simplement affecter ces données à nos attributs :

```
1 public function __construct(string $marque, string $modele)  
2 {  
3     $this->marque = $marque;  
4     $this->modele = $modele;  
5 }
```

Nous sommes donc désormais en capacité de définir la marque et le modèle de notre voiture lorsque nousinstancions notre objet. Imaginons que nous souhaitions instancier une Mercedes de modèle GLS-Coupé, il suffirait de faire :

```
1 $maVoiture = new Voiture("Mercedes", "GLS-Coupé");
```

Avec ces nouvelles additions, notre classe devrait ressembler à cela :

```
1 class Voiture
2 {
3     public string $modele;
4     public string $marque;
5     public int $vitesse = 0;
6
7     public function accélérer(int $vitesse = 0){
8         $this->vitesse += $vitesse;
9     }
10
11     public function __construct(string $marque, string $modele){
12         $this->marque = $marque;
13         $this->modele = $modele;
14     }
15 }
```

Nous avons nos attributs : marque, modèle, vitesse, une méthode « accélérer », et un constructeur.

Mais il existe encore un concept important que nous n'avons pas abordé, celui des accesseurs et des mutateurs.

Définition Accesseurs, mutateurs

Les accesseurs et mutateurs, plus communément appelés getter et setter, sont des méthodes qui nous permettent d'accéder aux données dans nos attributs, et de les modifier.

Getter, setter

Mais alors, quelle est leur utilité concrètement ? En effet, nous avons vu qu'il est tout à fait possible de modifier les données de notre classe sans passer par une méthode. Pour reprendre notre exemple plus tôt, si on veut changer la marque de notre voiture il nous suffirait d'utiliser :

```
1 $maVoiture->marque = "Mercedes";
```

Et effectivement, c'est le cas.

Pour observer l'intérêt de ces méthodes, nous allons utiliser un exemple concret : imaginons que notre attribut vitesse ne puisse jamais être inférieur à 0, comment pouvons-nous reproduire cela ?

La première méthode s'agirait de simplement rajouter une condition à chaque fois que l'on veut modifier la vitesse, pour s'assurer que la nouvelle valeur ne soit pas inférieure à 0. Le problème est que cela nous forcerait à dupliquer du code, ce qui est non seulement une mauvaise pratique, mais aussi extrêmement difficile à maintenir.

C'est là qu'interviennent les getters et les setters, ils nous permettent de définir un comportement qui sera appliqué à chaque fois que l'on accède ou que l'on modifie notre attribut.

Essayons donc de mettre en place notre setter, comme expliqué plus haut, les getter et setter, ne sont ni plus ni moins que des méthodes :

```
1 public function setVitesse(int $vitesse)
2 {
3
4 }
```

Par convention, le nom que l'on donne à notre setter se compose de cette manière : setNomDeAttribut.

Comme nous pouvons le voir, notre setter accepte un paramètre de type entier, maintenant rajoutons la logique dans notre méthode :

```
1 public function setVitesse(int $vitesse){
2     if($vitesse<0){
3         $this->vitesse = 0;
4     }else{
5         $this->vitesse = $vitesse;
6     }
7 }
```

Comme nous l'avons vu plus tôt, nous souhaitons que notre vitesse ne descende jamais en dessous de 0, donc il nous suffit d'utiliser une simple condition :

- Si le paramètre \$vitesse possède une valeur inférieure à 0, on met la valeur 0 dans notre attribut vitesse,
- Dans le cas contraire, nous utilisons la valeur passée en paramètre.

Ajoutons aussi notre getter :

```
1 public function getVitesse(){
2     return $this->vitesse;
3 }
```

Tout comme notre setter, la convention veut que l'on appelle notre getter getNomDeAttribut. Dans notre cas, nous n'avons pas de logique particulière à ajouter lorsque l'on récupère la donnée, donc un simple return suffit.

Nous pouvons aussi observer que nous utilisons le mot clé « \$this » lorsqu'on utilise ce mot clé, cela signifie que nous faisons référence à l'instance de notre classe.

Avec les getters et setters rajoutés pour notre vitesse, notre classe devrait ressembler à ceci :

```
1 class Voiture
2 {
3     public string $modele;
4     public string $marque;
5     public int $vitesse = 0;
6     public function setVitesse(int $vitesse){
7         if($vitesse<0){
8             $this->vitesse = 0;
9         }else{
10             $this->vitesse = $vitesse;
11         }
12     }
13     public function getVitesse(){
14         return $this->vitesse;
15     }
16     public function accélérer(int $vitesse = 0){
17         $this->vitesse += $vitesse;
18     }
19
20     public function __construct(string $marque, string $modele){
21         $this->marque = $marque;
22         $this->modele = $modele;
23     }
24 }
```

Si on regarde notre méthode « accélérer », celle-ci n'utilise pas notre setter, nous allons donc changer cela :

```
1 class Voiture
2 {
3     public string $modele;
4     public string $marque;
5     public int $vitesse = 0;
```

```

6   public function setVitesse(int $vitesse){
7       if($vitesse<0){
8           $this->vitesse = 0;
9       }else{
10          $this->vitesse = $vitesse;
11      }
12  }
13  public function getVitesse(){
14      return $this->vitesse;
15  }
16  public function accélérer(int $vitesse = 0){
17      $this->setVitesse($this->vitesse += $vitesse);
18  }
19
20  public function __construct(string $marque, string $modele){
21      $this->marque = $marque;
22      $this->modele = $modele;
23  }
24 }

```

Les getters et les setters sont donc les outils parfaits lorsque l'on veut s'assurer que chaque appel à nos attributs suit une logique bien précise, et nous permettent d'être en contrôle complet de nos attributs.

B. Exercice : Quiz

[solution n°2 p.16]

Question 1

Quel mot clé permet d'instancier un objet ?

- ☐ create
- ☐ spawn
- ☐ new

Question 2

Parmi ces déclarations de constructeurs, laquelle est correcte ?

- ☐ public function construct()
- ☐ public function __construct()
- ☐ public __construct()

Question 3

Parmi ces déclarations, laquelle est conventionnellement correcte pour un setter ?

- ☐ public function setVitesse()
- ☐ public function Vitesse()
- ☐ public function getVitesse()

Question 4

Parmi ces options, laquelle permettrait de modifier l'attribut marque ?

- ☐ \$maVoiture->marque =
- ☐ \$maVoiture ::marque =
- ☐ \$maVoiture.marque =

Question 5

À quel moment est appelé le constructeur ?

- ☐ Jamais
- ☐ À l'instanciation d'un objet
- ☐ Lorsque l'on appelle un getter

IV. Essentiel

Au fil de ce cours, nous avons pu étudier ce que sont les classes et les objets, ainsi que leurs différences. Nous avons pu voir que les classes servent de moule, et que les objets étaient donc créés à partir de ce moule. Nous savons aussi que chaque objet est composé d'attributs et de méthodes, tous prédéfinis par la classe utilisée pour l'instancier.

Le mot clé « *class* » nous permet de définir notre classe, pour le nom de celle-ci. Par convention, la première lettre est en majuscule, ainsi la syntaxe pour notre classe Voiture est comme ceci : « *class Voiture*{} ».

Pour instancier un objet, nous utilisons le mot clé « *new* », l'instanciation fait appel à une méthode particulière, nommée constructeur. Par défaut une classe possède un constructeur vide, mais il est possible de définir un constructeur spécifique, pour cela il est nécessaire d'utiliser cette syntaxe : « *public function __construct()* ».

Lorsque l'on veut lire ou modifier un attribut, nous pouvons passer directement par son nom, mais il est conseillé d'utiliser des méthodes appelées getter et setter. Ces méthodes permettent de centraliser la logique nécessaire lors de l'accès à nos attributs, facilitant la maintenance de notre code. Par convention ces méthodes sont nommées ainsi : « *getNomDeAttribut* » et « *setNomDeAttribut* ».

Ces concepts que nous avons explorés forment la base de la programmation orientée objet, il est important d'avoir une bonne maîtrise de ces principes avant d'aller plus loin.

V. Auto-évaluation

A. Exercice

Vous travaillez pour une jeune entreprise qui souhaite développer une application de gestion des employés pour une entreprise.

Dans cette application, il serait nécessaire de stocker les informations personnelles des employés, telles que leur nom, prénom et âge, dans une base de données. Pour cela, vous allez créer une classe nommée **Personne**, chacune de ces personnes ayant les attributs suivants : **nom**, **prénom**, **âge**. Au strict minimum, une personne est représentée par son nom et prénom.

Question 1

[solution n°3 p.17]

Définissez la classe **Personne** en suivant les consignes citées précédemment.

Question 2

[solution n°4 p.17]

Créez une instance de cette classe et une fois cette instance créée, modifiez l'âge de l'utilisateur via un setter, puis utilisez une méthode nommée **afficherInformations** pour afficher le nom, prénom, âge de cette personne.

B. Test

Exercice 1 : Quiz

[solution n°5 p.18]

Question 1

En prenant une instance \$monObjet de cette classe :

```
1 class MaClasse{
2     private string $monAttribut;
3     public function getMonAttribut(){
4         return $this->monAttribut;
5     }
6 }
```

Laquelle de ces options permettrait de lire la valeur présente dans l'attribut \$monAttribut ?

- ☐ \$monObjet->monAttribut;
- ☐ \$monObjet->getMonAttribut();
- ☐ Les deux

Question 2

Quelle est l'utilité du mot clé static ?

- ☐ Il est utilisé pour montrer qu'un attribut ne change jamais de valeur
- ☐ Il est simplement décoratif
- ☐ Il est utilisé pour désigner des méthodes et attributs accessibles hors instance d'objet

Question 3

Combien de fois peut-on définir un constructeur dans une même classe en utilisant **__construct** ?

- ☐ 0
- ☐ 1
- ☐ Autant que l'on veut

Question 4

Si l'on exécute cet exemple de code, que se passe-t-il ?

```
1 class MaClasse{
2
3     private string $monAttribut = "succès";
4
5     public static function getMonAttribut(){
6         return self::$monAttribut;
7     }
8 }
9
10 echo(MaClasse::getMonAttribut());
```

- ☐ On affiche succès
- ☐ Il ne se passe rien
- ☐ Nous obtenons une erreur

Question 5


Quelle est l'utilité des getters et setters ?

- ☐ Ils permettent de centraliser la logique utilisée lorsque l'on accède à un attribut
- ☐ Ils permettent de donner des noms plus longs à nos attributs
- ☐ Il s'agit simplement d'une convention

Solutions des exercices


Exercice p. 5 Solution n°1**Question 1**

Une méthode sert à :

- ☐ Définir les attributs d'un objet
- ☐ Définir comment l'objet doit être utilisé
- ☒ Définir le fonctionnement de l'objet
-  Les méthodes permettent de définir les actions qu'un objet peut effectuer.


Question 2

Quelle est la différence entre un objet et une classe ?

- ☐ Aucune, ce sont des synonymes
- ☐ On parle d'objet dans le monde réel, et de classe en programmation
- ☒ Une classe est un modèle, à partir duquel on crée des objets
-  La classe sert de modèle, chaque objet que l'on créera sera créé à partir de ce modèle.


Question 3

Quelle visibilité empêche toute modification d'un attribut en dehors de la classe dans laquelle il est déclaré ?

- ☒ Private
- ☐ Public
- ☐ Protected
-  En visibilité « *Public* », l'attribut peut être modifié de n'importe où. En visibilité « *Protected* », l'attribut peut être modifié soit depuis la classe où il est déclaré, soit depuis ses sous-classes. Seule la visibilité « *Private* » empêche toute modification en dehors de la classe où l'attribut est déclaré.


Question 4

Laquelle de ces déclarations d'attribut est correcte ?

- ☐ `string $modele;`
- ☒ `public string $modele;`
- ☐ `public string modele;`
-  Le premier exemple ne déclare pas de visibilité. PHP a besoin de savoir quel niveau d'accès donner à un attribut. Dans le second exemple, il manque le « `$` » nécessaire à toute déclaration de variable en PHP.

Question 5

Laquelle de ces déclarations de fonction est correcte ?


- ☐ public function accelerer(int \$vitesse)
- ☐ public function accelerer()
- ☒ Les deux
-  Dans le premier cas, il s'agit d'une déclaration d'une méthode avec paramètres. Dans le second cas il s'agit d'une méthode sans paramètres, les deux sont valides et servent différents besoins.

Exercice p. 10 Solution n°2

Question 1

Quel mot clé permet d'instancier un objet ?


- ☐ create
- ☐ spawn
- ☒ new

 La syntaxe correcte est la suivante : `$monObjet = new myClasse();`

Question 2

Parmi ces déclarations de constructeurs, laquelle est correcte ?


- ☐ public function construct()
- ☒ public function __construct()
- ☐ public __construct()

 __construct est le mot clé réservé en PHP pour définir un constructeur.

Question 3

Parmi ces déclarations, laquelle est conventionnellement correcte pour un setter ?


- ☒ public function setVitesse()
- ☐ public function Vitesse()
- ☐ public function getVitesse()

 Conventionnellement, il convient de nommer son setter setNomDeAttribut.

Question 4


Parmi ces options, laquelle permettrait de modifier l'attribut marque ?

- ☒ `$maVoiture->marque =`
- ☐ `$maVoiture ::marque =`
- ☐ `$maVoiture.marque =`

 En PHP on utilise « -> » pour accéder au contenu d'un objet, que ce soient des attributs ou des méthodes.

Question 5

À quel moment est appelé le constructeur ?

- ☐ Jamais
- ☒ À l'instanciation d'un objet
- ☐ Lorsque l'on appelle un getter
-  Le constructeur est automatiquement appelé lorsque l'on instancie un objet avec « new ».

p. 11 Solution n°3

Voici un exemple de code qui fonctionne :

```
1 class Personne{
2     public string $nom;
3     public string $prenom;
4     public int $age;
5
6     public function getNom(){
7         return $this->nom;
8     }
9     public function getPrenom()
10    {
11        return $this->prenom;
12    }
13    public function getAge()
14    {
15        return $this->age;
16    }
17    public function setPrenom(string $prenom)
18    {
19        $this->prenom = $prenom;
20    }
21    public function setNom(string $nom){
22        $this->nom = $nom;
23    }
24    public function setAge(int $age){
25        $this->age = $age;
26    }
27    public function __construct(string $nom, string $prenom){
28        $this->setNom($nom);
29        $this->setPrenom($prenom);
30    }
31 }
```

p. 11 Solution n°4

Voici un exemple de code qui fonctionne :

```
1 class Personne{
2     public string $nom;
3     public string $prenom;
4     public int $age;
5 }
```

```

6 public function getNom(){
7     return $this->nom;
8 }
9 public function getPrenom()
10 {
11     return $this->prenom;
12 }
13 public function getAge()
14 {
15     return $this->age;
16 }
17 public function setPrenom(string $prenom)
18 {
19     $this->prenom = $prenom;
20 }
21 public function setNom(string $nom){
22     $this->nom = $nom;
23 }
24 public function setAge(int $age){
25     $this->age = $age;
26 }
27 public function __construct(string $nom, string $prenom){
28     $this->setNom($nom);
29     $this->setPrenom($prenom);
30 }
31 public function afficherInformations(){
32     return($this->nom." ".$this->prenom." ".$this->age." ans");
33 }
34 }
35 $maPersonne = new Personne("John", "Smith");
36 $maPersonne->setAge(35);
37 echo($maPersonne->afficherInformations());

```

Exercice p. 12 Solution n°5

Question 1

En prenant une instance \$monObjet de cette classe :


```

1 class MaClasse{
2     private string $monAttribut;
3     public function getMonAttribut(){
4         return $this->monAttribut;
5     }
6 }

```


Laquelle de ces options permettrait de lire la valeur présente dans l'attribut \$monAttribut ?

- ☐ \$monObjet->monAttribut;
- ☒ \$monObjet->getMonAttribut();
- ☐ Les deux

 \$monAttribut est privé, et n'est donc pas accessible depuis l'extérieur de la classe, il est donc nécessaire de passer par son getter.


Question 2

Quelle est l'utilité du mot clé static ?

- ☐ Il est utilisé pour montrer qu'un attribut ne change jamais de valeur
- ☐ Il est simplement décoratif
- ☒ Il est utilisé pour désigner des méthodes et attributs accessibles hors instance d'objet
-  Le mot clé static est utilisé lors de la définition d'attributs et de méthodes ne dépendant pas d'un objet pour être accessible.

Question 3


Combien de fois peut-on définir un constructeur dans une même classe en utilisant `__construct` ?

- ☐ 0
- ☒ 1
- ☐ Autant que l'on veut
-  Par défaut, PHP ne permet de définir `__construct` qu'une seule fois, il est tout de même possible d'obtenir plusieurs constructeurs en utilisant ce que l'on appelle des fonctions de créations.

Question 4


Si l'on exécute cet exemple de code, que se passe-t-il ?

```
1 class MaClasse{
2
3     private string $monAttribut = "succès";
4
5     public static function getMonAttribut(){
6         return self::$monAttribut;
7     }
8 }
9
10 echo(MaClasse::getMonAttribut());
```

- ☐ On affiche succès
- ☐ Il ne se passe rien
- ☒ Nous obtenons une erreur
-  Une méthode statique ne peut pas accéder à un attribut non statique.

Question 5

Quelle est l'utilité des getters et setters ?

- ☒ Ils permettent de centraliser la logique utilisée lorsque l'on accède à un attribut
- ☐ Ils permettent de donner des noms plus longs à nos attributs
- ☐ Il s'agit simplement d'une convention
-  Les getters et setters servent à centraliser la logique utilisée lorsque l'on veut lire ou modifier les données de nos attributs. Cela permet de garder le contrôle sur les données de ces attributs, tout en facilitant la maintenance en évitant la duplication de code.