

La programmation Orientée Objet : concepts avancés

Table des matières

I. Contexte	3
II. Namespaces	3
A. Namespaces	3
B. Exercice : Quiz	5
III. Héritage	6
A. Héritage	6
B. Exercice : Quiz	11
IV. Essentiel	12
V. Auto-évaluation	12
A. Exercice : Exercice	12
B. Test	13
Solutions des exercices	13

I. Contexte

Durée : 1 h

Prérequis : la programmation Orientée Objet : concepts de base

Environnement de travail : PHP 8.2.5, OnlinePHP.io

Contexte

La Programmation Orientée Objet (POO) est un paradigme de programmation qui permet de modéliser le monde réel en utilisant des objets et des classes. PHP est un langage de programmation populaire qui prend en charge la POO, ce qui donne l'opportunité aux développeurs de créer des applications robustes et évolutives.

La POO permet de modéliser le monde réel de manière plus précise et de créer des applications plus évolutives et maintenables. Les concepts avancés de la POO tels que l'héritage, les interfaces, l'encapsulation et l'abstraction permettent de réutiliser le code existant et de faciliter le processus de développement. De plus, les namespaces offrent une meilleure organisation du code source et réduisent les risques de conflits de noms.

La connaissance de ces concepts aide à mieux comprendre les frameworks et les bibliothèques PHP populaires tels que Laravel, Symfony et Yii, qui utilisent largement ces concepts. Les compétences en POO avec PHP sont très demandées dans l'industrie et peuvent augmenter les opportunités de carrière et les salaires des développeurs.

II. Namespaces

A. Namespaces

Définition

Les Namespaces

Les Namespaces permettent d'encapsuler du code, de manière à rassembler des classes, des interfaces, des fonctions et des variables qui sont en lien les unes avec les autres.

Les Variables

Dans le cadre de ce cours, nous allons représenter deux types d'animaux, les chats et les chiens, de manière simplifiée. Pour ce faire, nous allons avoir besoin de déclarer différents namespace. Nous allons commencer avec un namespace pour animal.

Méthode

Déclarer un namespace

Pour déclarer un namespace, voici à quoi ressemble la syntaxe :

```
1 <?php
2 namespace animal;
3
4 ?>
```

La déclaration d'un namespace est assez simple, nous utilisons le mot-clé **namespace** suivi du nom que l'on souhaite donner à notre namespace, ici **animal**.

Méthode

Définir la classe Animal

Maintenant que nous avons notre namespace, nous allons définir la classe Animal. Pour cet exercice, nous allons nous contenter de lui donner un attribut : **nom**, nous allons aussi définir un getter, et un setter pour notre classe ainsi qu'une méthode **dort**.

Une fois tout cela déclaré, notre fichier **animal.php** devrait ressembler à ceci :

```
1 <?php
2 namespace animal;
3 class Animal{
4     protected string $nom;
5
6     public function getNom(){
7         return $this->nom;
8     }
9
10    public function setNom(string $nom = null){
11        $this->nom = $nom;
12    }
13
14    public function dort(){
15        echo('zzz');
16    }
17 }
18
19 ?>
```

Maintenant que nous savons déclarer un namespace, nous allons en profiter pour déclarer 3 autres namespace : **chat**, **chien**, **main**.

Les namespaces **chat** et **chien** contiendront les classes les concernant, le namespace **main** servira à exécuter notre code.

Voici ce que nous avons maintenant que ces namespaces ont été ajoutés :

```
1 <?php
2 namespace animal;
3 class Animal{
4     protected string $nom;
5
6     public function getNom(){
7         return $this->nom;
8     }
9
10    public function setNom(string $nom = null){
11        $this->nom = $nom;
12    }
13    public function dort(){
14        echo('zzz');
15    }
16 }
17 namespace chien;
18 namespace chat;
19 namespace main;
20 ?>
```

Méthode Appeler la classe Animal

Maintenant que notre espace de travail est organisé, regardons comment nous pouvons faire appel à notre classe **Animal**.

Plaçons-nous en dessous de « *main* » et essayons d'instancier un nouvel objet comme ceci :

```
1 namespace main;
2 $myAnimal = new Animal();
```

Si nous exécutons ce code, nous recevons une erreur. En effet, nous sommes actuellement dans le namespace **main**, dans lequel nous n'avons pas déclaré de classe **Animal**. Il faut donc prévenir PHP que nous faisons référence à une classe existant dans un namespace différent.

Pour cela, il est nécessaire de rajouter un `\namespace\` devant le nom de notre classe, où namespace est remplacé par le nom du namespace en question.

Dans notre exemple, nous aurions donc :

```
1 namespace main;  
2 $myAnimal = new \animal\Animal();
```

Maintenant que PHP sait où aller chercher la classe **Animal**, il arrive à instancier notre objet sans soucis.

Pour le reste de ce cours, nous allons continuer à définir nos namespace dans un seul fichier, mais il est conseillé de ne déclarer qu'un seul namespace par fichier, cette vidéo montre précisément comment cela fonctionne :

Pour les namespace **chat** et **chien**, nous reviendrons dessus une fois que nous en saurons davantage sur l'héritage.

B. Exercice : Quiz

[solution n°1 p.15]

Question 1

Que trouve-t-on dans les Namespaces ?

- ☐ Des noms uniquement
- ☐ Des variables
- ☐ Des classes, interfaces, fonctions et variables

Question 2

Laquelle de ces syntaxes est correcte ?

- ☐ namespace animal;
- ☐ animal namespace;
- ☐ public namespace animal;

Question 3

Laquelle de ces syntaxes est correcte ?

- ☐ \$monAnimal = new animal.Animal();
- ☐ \$monAnimal = new \animal\Animal();
- ☐ \$monAnimal = new "animal"Animal();

Question 4

Laquelle de ces phrases est correcte ?

- ☐ On ne peut définir qu'un seul namespace par fichier.
- ☐ On peut définir autant de namespace par fichier que l'on veut.
- ☐ On peut définir autant de namespace par fichier que l'on veut, mais il est préférable de n'en définir qu'un seul.

Question 5

Laquelle de ces phrases est correcte ?

- ☐ Il est impossible d'accéder aux classes définies dans un namespace en étant hors de ce namespace.
- ☐ PHP sait automatiquement à quel namespace nous faisons référence.
- ☐ Il est nécessaire de préciser à quel namespace nous faisons référence.

III. Héritage

A. Héritage

Maintenant que nous avons défini notre classe **Animal**, nous souhaitons définir nos classes **Chat** et **Chien**, pour ce faire nous allons utiliser le concept d'héritage.

Lorsqu'une classe hérite d'une autre, elle hérite de ses attributs, méthodes, getter et setters, tant que ceux-ci ne sont pas privés. Dans notre cas, cela veut dire que si nous héritons de la classe **Animal**, nous héritons de l'attribut **nom** et des getters et setters, ainsi que de la méthode **Dort**.

L'héritage nous permet donc d'étendre la fonctionnalité de la classe **Animal** à nos classes **Chat** et **Chien**, ceci nous permet d'éviter la duplication de code. De plus, dans le cas où nous modifions la classe **Animal**, ces modifications se répercutent sur les classes qui en héritent automatiquement.

En programmation, pour obtenir les attributs et les méthodes d'une classe existante dans une nouvelle classe, on utilise le mot-clé « *extends* » suivi du nom de la classe à hériter. Si la classe se trouve dans un namespace, on doit également inclure le namespace avant le nom de la classe.

Exemple

Prenons notre exemple, dans le namespace **chien** nous allons déclarer une nouvelle classe **Chien** qui hérite d'**Animal**, comme ceci :

```
1 <?php
2 namespace animal;
3 class Animal{
4     protected string $nom;
5
6     public function getNom(){
7         return $this->nom;
8     }
9
10    public function setNom(string $nom = null){
11        $this->nom = $nom;
12    }
13    public function dort(){
14        echo("zzz");
15    }
16 }
17 namespace chien;
18 class Chien extends \animal\Animal{
19 }
20 namespace chat;
21 namespace main;
22 ?>
```

Méthode Initialiser un objet

Pour nous assurer que notre héritage a bien fonctionné, essayons d'initialiser un objet de type **chien**, et d'appeler la méthode **dort** que nous devrions avoir hérité comme ceci :

```
1 namespace main;
2 $monChien = new \chien\Chien();
3 $monChien->dort();
```

Si tout se passe bien, nous affichons « zzz », ce qui veut dire que l'héritage se passe sans problème.

Faisons la même chose pour notre classe **Chat** :

```
1 <?php
2 namespace animal;
3 class Animal{
4     protected string $nom;
5
6     public function getNom(){
7         return $this->nom;
8     }
9
10    public function setNom(string $nom = null){
11        $this->nom = $nom;
12    }
13    public function dort(){
14        echo('zzz');
15    }
16 }
17 namespace chien;
18 class Chien extends \animal\Animal{
19 }
20 namespace chat;
21 class Chat extends \animal\Animal{
22 }
23 namespace main;
24 ?>
```

Nous avons maintenant nos deux classes, **Chien** et **Chat**, chacune héritant d'**Animal**.

Méthode Créer des méthodes

Maintenant que nous avons deux classes distinctes, créons des méthodes uniques à l'une et à l'autre. Nous allons créer une méthode **aboie()** qui retournera « Wouf » pour la classe Chien et une méthode **miaule()** qui retournera « Miaou » pour Chat, comme ceci :

```
1 <?php
2 namespace animal;
3 class Animal{
4     protected string $nom;
5
6     public function getNom(){
7         return $this->nom;
8     }
9
10    public function setNom(string $nom = null){
11        $this->nom = $nom;
12    }
13    public function dort(){
14        echo("zzz");
15    }
16 }
```

```

16 }
17 namespace chien;
18 class Chien extends \animal\Animal{
19     public function aboie(){
20         echo("Wouf");
21     }
22 }
23 namespace chat;
24 class Chat extends \animal\Animal{
25     public function miaule(){
26         echo("Miaou");
27     }
28 }
29 namespace main;
30 ?>

```

Si nous testons nos méthodes comme ceci :

```

1 namespace main;
2 $monChien = new \chien\Chien();
3 $monChat = new \chat\Chat();
4 $monChien->aboie();
5 $monChat->miaule();

```

Nous affichons effectivement « Wouf » et « Miaou », et si on inverse, et que l'on essaie d'appeler Miaule sur **\$monChien**, et Aboie sur **\$monChat**, nous recevons une erreur. Les deux classes sont donc bien distinctes malgré leur tronc commun.

Attention PHP et héritage de multiples classes

PHP ne supporte pas l'héritage de classes multiples par défaut. Dans les cas où on l'on veut étendre la fonctionnalité de plusieurs classes, il est possible d'utiliser les **Traits**, cette vidéo démontre ce fonctionnement :

Il ne s'agit pas de la seule utilité des Traits. L'intérêt principal des Traits est de permettre à plusieurs classes de partager des fonctions similaires, dont aucune logique métier n'est liée entre chacune de ces classes.

À l'inverse de l'héritage qui permet aux classes filles d'avoir un parent partageant une logique similaire.

Exemple

Prenons par exemple un employé et un magasin, ces deux entités possèdent une adresse. Or, il n'existe pas de corrélation entre un employé et un magasin. Un Trait nous permet donc de représenter une adresse qui s'agit de celle de notre magasin, ou de notre employé.

Concrètement, nous pourrions représenter l'exemple ci-dessus par ce code :

```

1 trait Adresse{
2     private string $adresse;
3
4     public function getAdresse(){
5         return $this->adresse;
6     }
7
8     public function setAdresse(string $adresse){
9         $this->adresse = $adresse;
10    }
11 }
12
13 class Employe{
14     use Adresse;

```



```

15
16 //Logique métier Employe
17 //...
18 }
19
20 class Magasin{
21     use Adresse;
22
23     //Logique métier Magasin
24     //...
25 }

```

Les Traits sont aussi largement utilisés par les différents frameworks PHP, nous pouvons citer par exemple Symfony, notamment les EntityTraits (id, uuid, createdAt, UpdatedAt, isDeleted...) partagés entre chaque entité, mais n'ayant pas de logique similaire.

Interface 1

Nous allons aborder un concept similaire à l'héritage, mais pourtant bien différent, celui des **interfaces**.

Définition Les interfaces

Une interface est un bloc de code dans lequel nous pouvons déclarer des méthodes. À la grande différence des classes, dans les interfaces, il est impossible de déclarer des attributs, ou bien d'avoir de la logique.

Pour mieux visualiser cela, comparons notre classe Animal, à une interface Animal :

```

1 <?php
2 namespace animal;
3 class Animal{
4     protected string $nom;
5
6     public function getNom(){
7         return $this->nom;
8     }
9
10    public function setNom(string $nom = null){
11        $this->nom = $nom;
12    }
13    public function dort(){
14        echo('zzz');
15    }
16 }
17
18 interface AnimalInterface{
19     public function getNom();
20     public function setNom(string $nom = null);
21     public function dort();
22 }

```

Nous pouvons voir que nous avons déclaré les mêmes méthodes pour notre interface que pour notre classe, nous pouvons aussi voir que ces méthodes sont vides. En effet, les interfaces ne contiennent pas de logique.

Au même titre qu'une classe sert de moule pour un objet, une interface va servir de moule pour une classe. Une classe qui implémente une interface va devoir implémenter la logique de chacune des méthodes définies dans l'interface.

Quand utiliser une interface ?

Il est préférable d'utiliser une interface dans les cas où nous savons quelle fonctionnalité nos classes doivent avoir, mais que l'implémentation de cette fonctionnalité doit se faire au cas par cas.

Cette vidéo montre plus en détail le fonctionnement et les avantages des interfaces :

Remarque

Une interface est une forme de contrat.

Définition

Contrats

Un contrat, en informatique, est un ensemble d'obligations qui doivent être respectées au moment de l'exécution de notre code. Les contrats permettent de spécifier la manière dont doit être utilisé un composant, mais aussi comment le composant doit se comporter.

Si nous reprenons l'exemple de notre interface, où l'on veut utiliser une interface, nous sommes dans l'obligation d'implémenter les méthodes ayant été définies par l'interface. Dans le cas contraire, le code ne fonctionnera pas et retournera une erreur.

Ce concept de contrat est essentiel au bon fonctionnement de beaucoup de framework, tel que Symfony, où de nombreuses classes dépendent de la bonne implémentation de ces contrats. Nous pouvons par exemple citer la classe User, qui doit disposer des méthodes getRoles, getPassword, etc., et ce, en implémentant l'interface UserInterface.

Comme nous l'avons vu plus haut, les interfaces sont une forme de contrats, mais il ne s'agit pas des seuls contrats à notre disposition, il existe aussi ce que l'on appelle les classes abstraites.

Classes abstraites

Les classes abstraites sont à mi-chemin entre une classe et une interface.

Au même titre qu'une interface, il n'est pas possible d'instancier une classe abstraite.

Additionnellement, là où une interface nous permet de définir des méthodes dont l'implémentation dépendra des classes implémentant notre interface, une classe abstraite permet de définir des méthodes abstraites, donc l'implémentation dépendra des classes qui héritent de notre classe abstraite.

Tout comme une classe concrète, il est possible d'hériter d'une classe abstraite, mais PHP ne supporte l'héritage que d'une classe, qu'il s'agisse d'une classe abstraite ou non. Il est aussi possible de définir des attributs. On peut également définir et d'implémenter des méthodes concrètes.

Exemple

Utilisons un exemple concret. Reprenons notre classe **Animal**, et nos classes **Chien** et **Chat**. Tout comme nous l'avons fait pour illustrer le concept des interfaces, nous allons laisser le soin aux classes **Chien** et **Chat** d'implémenter la méthode **cri()** cette méthode sera déclarée dans notre classe abstraite, comme ceci :

```
1 abstract class Animal{
2     protected string $nom;
3
4     public function getNom(){
5         return $this->nom;
6     }
7
8     public function setNom(string $nom = null){
9         $this->nom = $nom;
10    }
11    public function dort(){
```

```
12     echo("zzz");
13 }
14
15 public abstract function cri();
16
17 }
```

Si nous observons la classe abstraite **animal**, nous pouvons voir que pour déclarer une classe abstraite, il suffit d'utiliser le mot-clé « *abstract* » avant *class*.

De même en préfixant fonction par *abstract*, nous déclarons une méthode abstraite, qui devra obligatoirement être implémentée par les classes qui héritent de notre classe abstraite. Dans notre cas, les classes **Chien** et **Chat** devront donc implémenter la méthode **cri()**.

Passons donc à nos classes **Chien** et **Chat** :

```
1 namespace chien;
2 class Chien extends \animal\Animal{
3     public function cri(){
4         echo("Wouf");
5     }
6 }
7 namespace chat;
8 class Chat extends \animal\Animal{
9     public function cri(){
10         echo("Miaou");
11     }
12 }
```

Nous pouvons voir que l'héritage d'une classe abstraite se fait de la même manière qu'une classe concrète, via le mot-clé « *extends* ». Quant à l'implémentation de **cri()**, le fonctionnement est identique à celui de l'implémentation d'une interface. Il suffit de définir une méthode du même nom que la méthode que l'on implémente, et de définir la logique de cette méthode. Dans notre cas, afficher « *Wouf* » pour **Chien**, et « *Miaou* » pour **Chat**.

Les classes abstraites nous permettent donc de définir une fondation pour les différentes classes qui en hériteront, tout en leur laissant la tâche d'implémenter certaines des fonctionnalités.

B. Exercice : Quiz

[solution n°2 p.16]

Question 1

Quelle syntaxe est utilisée pour déclarer une interface ?

- ☐ new interface MonInterface{}
- ☐ interface MonInterface{}
- ☐ class MonInterface{}

Question 2

Quel mot-clé permet d'hériter d'une classe ?

- ☐ inherit
- ☐ implement
- ☐ extends

Question 3

Que peut contenir une interface ?

- ☐ Des méthodes
- ☐ Des attributs
- ☐ Les deux

Question 4

Quel est le mot-clé permettant d'implémenter une interface ?

- ☐ implements
- ☐ extends
- ☐ possess

Question 5

De ces noms d'interface, lequel respecte la convention ?

- ☐ ExempleInterface
- ☐ InterfaceExemple
- ☐ Aucune

IV. Essentiel

En conclusion, les concepts avancés en POO avec PHP tels que les namespaces, l'héritage, les interfaces et les classes abstraites sont des éléments clés pour la création d'applications robustes et évolutives, au même titre que les namespaces pour l'organisation du code source. Ils permettent de diviser les classes, les fonctions et les constantes en groupes logiques, afin de faciliter la réutilisation du code, de minimiser les conflits de noms et de mieux gérer la complexité du code.

L'héritage est un concept important en POO qui permet à une classe enfant d'hériter des propriétés et des méthodes d'une classe parent. Cela permet aux développeurs de réutiliser le code existant et de simplifier le processus de développement. La classe enfant peut étendre ou modifier les fonctionnalités de la classe parent en ajoutant ou en surchargeant des méthodes. L'héritage permet également une meilleure organisation du code, en créant des hiérarchies de classes.

Les interfaces sont un autre concept clé en POO avec PHP. Les interfaces définissent un ensemble de méthodes que les classes doivent implémenter. Les interfaces permettent de garantir une meilleure cohérence dans la conception de l'application et facilitent la collaboration entre les développeurs. Les classes peuvent implémenter plusieurs interfaces pour fournir différentes fonctionnalités et garantir une plus grande flexibilité.

En tant que tel, il est important de les maîtriser pour devenir un développeur PHP compétent et efficace.

V. Auto-évaluation

A. Exercice : Exercice

Pour ce cas pratique, nous allons définir deux classes, une classe **Homme**, et une classe **Femme**. Ces deux classes vont hériter de la classe **Personne**. Ces classes devront posséder les attributs **Prenom** et **Age**.

De plus, ces deux classes devront implémenter la méthode "sePresenter" qui donnera ces résultats :

- Je suis un Homme de \$age ans et je m'appelle \$prenom
- Je suis une Femme age de \$age et je m'appelle \$prenom

Question 1

[solution n°3 p.17]

Dans un premier temps, définissez la classe **Personne**.

Question 2

[solution n°4 p.17]

Définissez les classes **Homme** et **Femme**, en suivant les consignes précédentes, modifiez l'âge, le prénom et utilisez la méthode **SePresenter**. Assurez-vous d'utiliser les concepts d'héritage et d'interface.

B. Test**Exercice 1 : Quiz**

[solution n°5 p.18]

Question 1

Laquelle de ces phrases est vraie ?

- ☐ Il est possible d'hériter d'une classe et d'implémenter plusieurs interfaces à la fois.
- ☐ Il n'est pas nécessaire d'implémenter toutes les méthodes d'une interface.
- ☐ Il est possible d'hériter de plusieurs classes.

Question 2

Quelle syntaxe est correcte ?

- ☐ class Enfant extends Parent implements FamilleInterface
- ☐ class Enfant implements FamilleInterface extends Parent
- ☐ class Enfant extends FamilleInterface extends Parent

Question 3

Quel mot-clé permet d'hériter d'un Trait ?

- ☐ extends
- ☐ get
- ☐ use

Question 4

Lequel de ces concepts ne peut pas être instancié ?

- ☐ Traits
- ☐ Interface
- ☐ Les deux

Question 5


Quelle affirmation est fausse ?

- ☐ Les Traits peuvent définir de la logique dans leurs méthodes.
- ☐ Les interfaces peuvent définir des attributs.
- ☐ Les classes ne peuvent pas être héritées.

Solutions des exercices


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

Que trouve-t-on dans les Namespaces ?

- ☐ Des noms uniquement
- ☐ Des variables
- ☒ Des classes, interfaces, fonctions et variables
-  Namespaces encapsule des classes, interfaces, fonctions et variables en lien les unes avec les autres.


Question 2

Laquelle de ces syntaxes est correcte ?

- ☒ namespace animal;
- ☐ animal namespace;
- ☐ public namespace animal;
-  Pour déclarer un namespace, la syntaxe correcte est **namespace** suivi du nom.


Question 3

Laquelle de ces syntaxes est correcte ?

- ☐ \$monAnimal = new animal.Animal();
- ☒ \$monAnimal = new \animal\Animal();
- ☐ \$monAnimal = new "animal"Animal();
-  Pour appeler du code venant d'un namespace, il est nécessaire de faire appel à la classe en utilisant son nom complet, qui inclut le namespace.


Question 4

Laquelle de ces phrases est correcte ?

- ☐ On ne peut définir qu'un seul namespace par fichier.
- ☐ On peut définir autant de namespace par fichier que l'on veut.
- ☒ On peut définir autant de namespace par fichier que l'on veut, mais il est préférable de n'en définir qu'un seul.
-  En PHP, il est possible de définir plusieurs namespace par fichier, mais pour des raisons de maintenabilité, il est préférable de n'en définir qu'un seul.

Question 5


Laquelle de ces phrases est correcte ?

- ☐ Il est impossible d'accéder aux classes définies dans un namespace en étant hors de ce namespace.
- ☐ PHP sait automatiquement à quel namespace nous faisons référence.
- ☒ Il est nécessaire de préciser à quel namespace nous faisons référence.
-  Il est possible de référencer des classes, fonctions, variables, etc., définies dans des Namespaces différents, mais il faut préciser le Namespace en question.

Exercice p. 11 Solution n°2


Question 1

Quelle syntaxe est utilisée pour déclarer une interface ?

- ☐ new interface MonInterface{}
- ☒ interface MonInterface{}
- ☐ class MonInterface{}
-  La déclaration d'une interface se fait avec le mot-clé interface.


Question 2

Quel mot-clé permet d'hériter d'une classe ?

- ☐ inherit
- ☐ implement
- ☒ extends
-  On utilise « *extends* » pour hériter d'une classe.


Question 3

Que peut contenir une interface ?

- ☒ Des méthodes
- ☐ Des attributs
- ☐ Les deux
-  En PHP, les interfaces ne peuvent contenir que des méthodes uniquement.


Question 4

Quel est le mot-clé permettant d'implémenter une interface ?

- ☒ implements
- ☐ extends
- ☐ possess
-  En PHP, on utilise « *implements* » pour implémenter une interface.

Question 5

De ces noms d'interface, lequel respecte la convention ?

- ☒ ExempleInterface
- ☐ InterfaceExemple
- ☐ Aucune
-  La convention pour le nommage en PHP demande de suffixer le nom de ses interfaces avec « *Interface* ».

p. 13 Solution n°3

Voici un exemple qui fonctionne :

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

p. 13 Solution n°4

Voici un exemple qui fonctionne :

```
1 interface PresentationInterface{
2     public function sePresenter();
3 }
4
5 class Homme extends Personne implements PresentationInterface{
6
7     public function sePresenter(){
8         echo("Je suis un Homme de ".$this->getAge()." ans et je m'appelle ".$this->getPrenom());
9     }
10 }
11
```

```

12 class Femme extends Personne implements PresentationInterface{
13
14     public function sePresenter(){
15         echo("Je suis une Femme age de ".$this->getAge()." et je m'appelle ".$this->getPrenom());
16     }
17 }
18
19 $monsieur = new Homme("john", "28");
20 $monsieur->sePresenter();
21
22 $femme = new Femme("Marie", "27");
23 $femme->sePresenter();

```

Exercice p. 13 Solution n°5

Question 1

Laquelle de ces phrases est vraie ?

- ☒ Il est possible d'hériter d'une classe et d'implémenter plusieurs interfaces à la fois.
- ☐ Il n'est pas nécessaire d'implémenter toutes les méthodes d'une interface.
- ☐ Il est possible d'hériter de plusieurs classes.
- ☒ Il est nécessaire d'implémenter toutes les méthodes définies dans une interface, et il n'est pas possible d'hériter de plusieurs classes. Pour cela, il faut utiliser des Traits.

Question 2

Quelle syntaxe est correcte ?

- ☒ class Enfant extends Parent implements FamilleInterface
- ☐ class Enfant implements FamilleInterface extends Parent
- ☐ class Enfant extends FamilleInterface extends Parent
- ☒ L'implémentation se fait toujours avec le mot-clé **implements** et toujours après l'héritage des classes.


Question 3

Quel mot-clé permet d'hériter d'un Trait ?

- ☐ extends
- ☐ get
- ☒ use
- ☒ Le mot-clé **use** est utilisé lorsque l'on veut hériter d'un Trait.


Question 4

Lequel de ces concepts ne peut pas être instancié ?

- ☐ Traits
- ☐ Interface
- ☒ Les deux
-  Ni les Traits ni les interfaces ne peuvent être instanciés.

Question 5

Quelle affirmation est fausse ?

- ☐ Les Traits peuvent définir de la logique dans leurs méthodes.
- ☒ Les interfaces peuvent définir des attributs.
- ☐ Les classes ne peuvent pas être héritées.
-  Les interfaces ne peuvent définir ni attribut ni logique dans leurs méthodes.