



Programmation orientée objet en PHP

[Accueil](#) ▶ [Mes cours](#) ▶ [Développement logiciel](#) ▶ [POO PHP](#) ▶ [L'héritage](#) ▶ [Introduction](#)

Introduction

Nous allons créer trois classes :

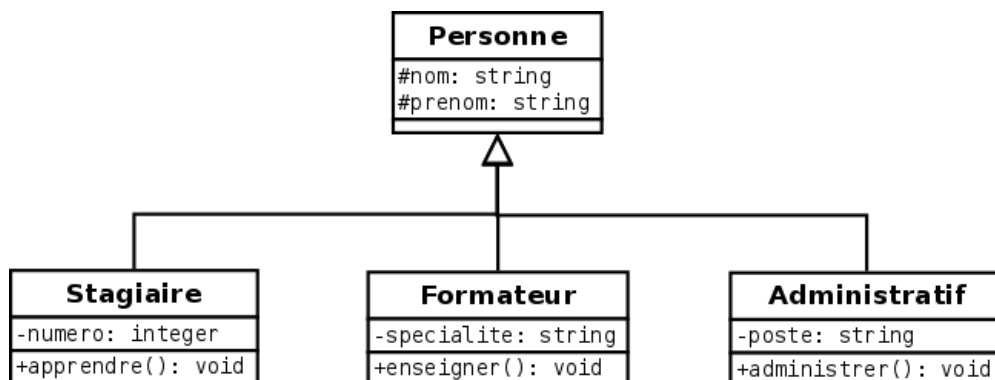
- une classe **Stagiaire** possédant les attributs nom, prénom et numéro ainsi qu'une méthode apprendre()
- une classe **Formateur** possédant les attributs nom, prénom et spécialité ainsi qu'une méthode enseigner()
- une classe **Administratif** possédant les attributs nom, prénom et poste ainsi qu'une méthode administrer().

| Stagiaire | Formateur | Administratif |
|---|--|---|
| -nom: string -prenom: string -numero: integer +apprendre(): void | -nom: string -prenom: string -specialite: string +enseigner(): void | -nom: string -prenom: string -poste: string +administrer(): void |

Le Stagiaire, le Formateur et l'Administratif ont tous des attributs communs : le nom et le prénom. Puisque ces trois éléments ont en commun d'être tous des personnes, et comme une personne possède un nom et un prénom, il est normal que le Stagiaire, le Formateur et l'Administratif possède un nom et un prénom. Mais le Stagiaire, le Formateur et l'Administratif ne sont pas de "simples personnes", ils possèdent en plus de la personne des attributs et des compétences propres, ce sont en fait des personnes "spécialisées" : ces entités possèdent un rôle spécial et des caractéristiques propres. En plus du nom et du prénom, le Stagiaire possède un attribut numéro et une méthode apprendre, le Formateur un attribut spécialité et une méthode enseigner, l'Administratif un attribut poste et une méthode administrer. Chacun a son rôle, mais tous sont des personnes, et possèdent donc un nom et un prénom.

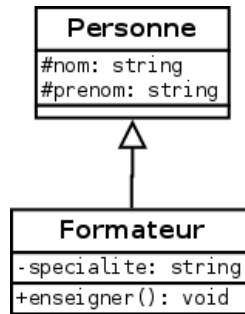
Nous pourrions réunir en une classe les attributs et méthodes communes aux entités Stagiaire, Formateur et Administratif : une classe **Personne**, par exemple, qui posséderait un attribut nom et un attribut prénom. Puis faire en sorte que les classes Stagiaire, Formateur et Administratif récupèrent les attributs et méthodes de la classe **Personne**. Cela nous éviterait de dupliquer les attributs nom et prénom dans chaque classe. Nous pourrions dire alors que la classe **Personne** est la classe "mère" ou super-classe des classes Stagiaire, Formateur et Administratif et que les classes Stagiaire, Formateur et Administratif héritent de la classe **Personne**. La classe **Personne** contient les attributs, et les méthodes, communs aux trois classes Stagiaire, Formateur et Administratif. La relation qui relie la superclasse, ici **Personne**, à ses classes filles, ici Stagiaire, Formateur et Administratif, est une relation d'héritage, ou encore une relation is-a ("est un"). Le Formateur est une **Personne**, il hérite donc de la classe **Personne** ; de même pour les classes Stagiaire et Administratif. Nous pouvons aussi dire que la classe fille (ou sous-classe) est une spécialisation de la super-classe.

La relation d'**héritage** permet à la classe fille de posséder les attributs et méthodes accessibles de la classe mère.



La relation d'héritage est représentée par la flèche. Le caractère "#" devant les attributs nom et prénom de **Personne** signifie que ces attributs ont une visibilité "protected" : ils sont accessibles à la classe elle-même et à ses classes filles. Rien n'empêche une super-classe de posséder des attributs et méthodes privées, mais ces attributs et méthodes ne seront pas accessibles aux sous-classes. Les attributs et méthodes publics sont

accessibles à tous, donc aux sous-classes.



Ce diagramme signifie que la classe Formateur hérite de la classe Personne, que la classe Formateur est une sous-classe ou classe fille de Personne, que la classe Personne est la super-classe ou la classe mère de la classe Personne.

En Java, pour indiquer qu'une classe B hérite d'une classe A, on utilise le mot clé `extends` dans la déclaration de la classe B.

Personne.php (minimaliste : avec des attributs publics et sans constructeur personnalisé pour l'instant)

```

1 <?php
2
3 // Personne.php
4 class Personne {
5
6     public $nom;
7     public $prenom;
8
9 }
  
```

Formateur.php (minimaliste aussi : avec un attribut public et sans constructeur personnalisé pour l'instant)

```

1 <?php
2
3 // Formateur.php
4 class Formateur extends Personne {
5
6     public $specialite;
7
8     public function enseigner() {
9         return 'j\'enseigne';
10    }
11
12 }
  
```

Le mot clé `extends` signifie que Formateur hérite de Personne, ce qui signifie que Formateur possède les attributs accessibles définis dans Personne, c'est-à-dire le nom et le prénom. À partir d'une référence à un objet de type Formateur, nous allons donc pouvoir accéder aux attributs nom, prénom et spécialité.

```

1 <?php
2
3 // lanceur.php
4 include './Personne.php';
5 include './Formateur.php';
6
7 $formateur = new Formateur();
8 $formateur->nom = 'Sparrow';
9 $formateur->prenom = 'Jack';
10 $formateur->specialite = 'Développement Logiciel';
11 echo "nom : $formateur->nom prénom : $formateur->prenom spécialité : $formateur->specialite";
  
```

Dans ce lanceur nous avons créé un objet de type Formateur via le constructeur par défaut et nous avons accédé

directement en lecture et écriture à ses attributs. Nous n'avons donc pas respecté le principe de l'encapsulation (accès aux attributs de l'objet uniquement via des méthodes de l'objet).

Nous allons modifier notre classe `Personne` pour protéger ses attributs tout en les rendant accessibles à ses classes filles : nous donnerons aux attributs la visibilité `protected`. Et nous créerons des getters et setters sur ces attributs.

```

1  <?php
2
3  // Personne.php
4  class Personne {
5
6      protected $nom;
7      protected $prenom;
8
9      public function getNom() {
10         return $this->nom;
11     }
12
13     public function getPrenom() {
14         return $this->prenom;
15     }
16
17     public function setNom($nom) {
18         $this->nom = $nom;
19     }
20
21     public function setPrenom($prenom) {
22         $this->prenom = $prenom;
23     }
24
25 }
```

Nous modifions aussi la classe `Formateur` pour protéger ces attributs avec une visibilité `private` et des getters et setters :

```

1  <?php
2
3  // Formateur.php
4  class Formateur extends Personne {
5
6      private $specialite;
7
8      public function getSpecialite() {
9         return $this->specialite;
10     }
11
12     public function setSpecialite($specialite) {
13         $this->specialite = $specialite;
14     }
15
16     public function enseigner() {
17         return 'j\'enseigne';
18     }
19
20 }
```

Nous allons aussi devoir modifier notre lanceur puisque les attributs ne sont plus accessibles directement. Il faudra maintenant utiliser les getters et setters.

```

1  <?php
2
3  // lanceur.php
4  include './Personne.php';
5  include './Formateur.php';
```

```

6
7 $formateur = new Formateur();
8 $formateur->setNom('Sparrow');
9 $formateur->setPrenom('Jack');
10 $formateur->setSpecialite('Développement Logiciel');
11 echo "nom : {$formateur->getNom()} prénom : {$formateur->getPrenom()} "
12 . "spécialité : {$formateur->getSpecialite()}";

```

Dans les cours précédents nous avons vu que pour initialiser les valeurs des attributs il était très pratique d'utiliser un constructeur personnalisé. Pour initialiser les attributs nom, prénom et spécialité de Formateur nous pourrions donc créer un constructeur.

```

1 <?php
2
3 // Formateur.php
4 class Formateur extends Personne {
5
6     private $specialite;
7
8     function __construct($nom, $prenom, $specialite) {
9         $this->nom = $nom;
10        $this->prenom = $prenom;
11        $this->specialite = $specialite;
12    }
13
14    public function getSpecialite() {
15        return $this->specialite;
16    }
17
18    public function setSpecialite($specialite) {
19        $this->specialite = $specialite;
20    }
21
22    public function enseigner() {
23        return 'j\'enseigne';
24    }
25
26 }

```

Notre lanceur devient :

```

1 <?php
2
3 // lanceur.php
4 include './Personne.php';
5 include './Formateur.php';
6
7 $formateur = new Formateur('Sparrow', 'Jack', 'Développement Logiciel');
8 echo "nom : {$formateur->getNom()} prénom : {$formateur->getPrenom()} "
9 . "spécialité : {$formateur->getSpecialite()}";

```

Les attributs nom et prénom sont déclarés dans la classe Personne. Il serait donc intéressant de créer un constructeur personnalisé (ou surchargé) dans la classe Personne qui initialiserait les attributs nom et prénom :

```

1 <?php
2
3 // Personne.php
4 class Personne {
5
6     protected $nom;
7     protected $prenom;
8
9     function __construct($nom, $prenom) {
10        $this->nom = $nom;

```

```

11         $this->prenom = $prenom;
12     }
13
14     public function getNom() {
15         return $this->nom;
16     }
17
18     public function getPrenom() {
19         return $this->prenom;
20     }
21
22     public function setNom($nom) {
23         $this->nom = $nom;
24     }
25
26     public function setPrenom($prenom) {
27         $this->prenom = $prenom;
28     }
29
30 }

```

Nous pourrions donc créer une instance de *Personne* et l'initialiser avec ce constructeur. Et nous pourrions aussi ré-utiliser ce constructeur dans la classe *Formateur* : dans le constructeur de la classe fille (*Formateur*) nous allons appeler le constructeur de la classe mère (*Personne*). Pour cela nous allons utiliser le mot clé *parent* et l'opérateur *::*.

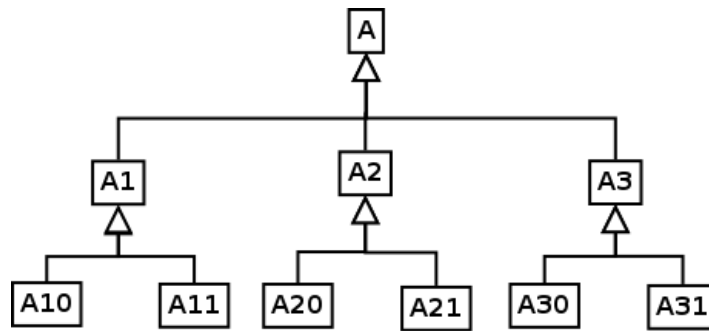
```

1  <?php
2
3  // Formateur.php
4  class Formateur extends Personne {
5
6      private $specialite;
7
8      function __construct($nom, $prenom, $specialite) {
9          parent::__construct($nom, $prenom);
10         $this->specialite = $specialite;
11     }
12
13     public function getSpecialite() {
14         return $this->specialite;
15     }
16
17     public function setSpecialite($specialite) {
18         $this->specialite = $specialite;
19     }
20
21     public function enseigner() {
22         return 'j\'enseigne';
23     }
24
25 }

```

Le lanceur n'est pas modifié.

En PHP on ne peut hériter que d'une seule classe, mais rien n'empêche une classe d'hériter d'une classe qui elle-même hérite d'une classe, formant ainsi une hiérarchie de classes assez importante :



Les classes A1, A2, A3 héritent directement de A, et possèdent donc les attributs et méthodes accessibles de A. Les classes A10 et A11 héritent directement de A1, et possèdent donc les attributs et méthodes accessibles de A1, et donc de A puisque A1 hérite de A.

Fin

NAVIGATION



Accueil

- [Ma page](#)
- Pages du site
- Mon profil
- Cours actuel
 - [POO PHP](#)
 - Participants
 - Généralités
 - La programmation orientée objet : premiers pas
 - L'héritage
 - Introduction**
 - T.P. héritage
 - Redéfinition de méthodes
 - T.P. redéfinition
 - Les classes abstraites
 - T.P. véhicule
 - Le mot clé final
 - T.P. employés
 - Les interfaces
 - Le typage
 - Les namespaces
 - Les exceptions
 - Les bases de données avec PDO
 - Les tests avec PHPUnit
 - Petite application version 2
 - Petite application version 3

Mes cours

ADMINISTRATION



- Administration du cours

- Réglages de mon profil

Connecté sous le nom « [Arnaud Lemais](#) » ([Déconnexion](#))
[POO PHP](#)



Programmation orientée objet en PHP

[Accueil](#) ► [Mes cours](#) ► [Développement logiciel](#) ► [POO PHP](#) ► [L'héritage](#) ► [T.P. héritage](#)

T.P. héritage

Implémentez les classes Stagiaire et Administratif héritant de Personne. Vous ré-utiliserez la classe Personne de la leçon précédente. Vous créerez pour chaque classe un constructeur personnalisé, des getters et setters pour les attributs sur le modèle de la dernière classe Formateur présentée dans la leçon précédente. Dans un lanceur, vousinstancierez ces deux classes.

Vous remettrez vos scripts dans une archive zip dont le nom sera de la forme "heritage_*nom_prenom*.zip".

État du travail remis

| | |
|--------------------------|------------------------------|
| Numéro de tentative | Ceci est la tentative 1. |
| Statut des travaux remis | Aucune tentative |
| Statut de l'évaluation | Pas évalué |
| Dernière modification | vendredi 27 mars 2015, 15:24 |

Ajouter un travail

Modifier votre travail remis

NAVIGATION



[Accueil](#)

■ [Ma page](#)

[Pages du site](#)

[Mon profil](#)

[Cours actuel](#)

[POO PHP](#)

[Participants](#)

[Généralités](#)

[La programmation orientée objet : premiers pas](#)

[L'héritage](#)

[Introduction](#)

[T.P. héritage](#)

[Redéfinition de méthodes](#)

[T.P. redéfinition](#)

[Les classes abstraites](#)

[T.P. véhicule](#)

[Le mot clé final](#)

[T.P. employés](#)

[Les interfaces](#)

[Le typage](#)

[Les namespaces](#)

[Les exceptions](#)

[Les bases de données avec PDO](#)

Les tests avec PHPUnit
Petite application version 2
Petite application version 3

[Mes cours](#)

ADMINISTRATION



Administration du cours

Réglages de mon profil

Connecté sous le nom « [Arnaud Lemais](#) » ([Déconnexion](#))
[POO PHP](#)



Programmation orientée objet en PHP

[Accueil](#) ► [Mes cours](#) ► [Développement logiciel](#) ► [POO PHP](#) ► [L'héritage](#) ► [Redéfinition de méthodes](#)

Redéfinition de méthodes

La redéfinition de méthodes permet à une sous-classe de modifier le comportement d'une méthode héritée de sa super-classe.

Dans la classe `Personne`, nous allons ajouter une méthode publique `parler()`. La classe `Formateur` héritera donc de cette méthode.

```
1 <?php
2
3 // Personne.php
4 class Personne {
5
6     protected $nom;
7     protected $prenom;
8
9     function __construct($nom, $prenom) {
10         $this->nom = $nom;
11         $this->prenom = $prenom;
12     }
13
14     public function getNom() {
15         return $this->nom;
16     }
17
18     public function getPrenom() {
19         return $this->prenom;
20     }
21
22     public function setNom($nom) {
23         $this->nom = $nom;
24     }
25
26     public function setPrenom($prenom) {
27         $this->prenom = $prenom;
28     }
29
30     public function parler() {
31         return 'je parle...';
32     }
33
34 }
```

Dans le lanceur, nous créons une instance de `Formateur` et nous appelons la méthode `parler()`.

```
1 <?php
2
3 // lanceur.php
4 include './Personne.php';
5 include './Formateur.php';
6
7 $formateur = new Formateur('Sparrow', 'Jack', 'développement logiciel');
8 echo $formateur->parler();
```

Lorsque nous appelons la méthode `parler()` avec la référence `formateur`, le comportement est bien celui défini

dans la super-classe, c'est-à-dire la classe Personne.

Maintenant si je souhaite donner à Formateur une façon de parler qui lui sera propre, je peux redéfinir la méthode parler() dans la classe Formateur.

```
1 <?php
2
3 // Formateur.php
4 class Formateur extends Personne {
5
6     private $specialite;
7
8     function __construct($nom, $prenom, $specialite) {
9         parent::__construct($nom, $prenom);
10        $this->specialite = $specialite;
11    }
12
13    public function getSpecialite() {
14        return $this->specialite;
15    }
16
17    public function setSpecialite($specialite) {
18        $this->specialite = $specialite;
19    }
20
21    public function enseigner() {
22        return 'j\'enseigne';
23    }
24
25    public function parler() {
26        return 'le PHP est un langage...';
27    }
28
29 }
```

Le lanceur n'est pas modifié. La classe Personne a gardé sa façon de parler, la classe Formateur possède une façon de parler bien à elle.

Si je souhaite ré-utiliser la méthode de la super-classe dans la méthode de la sous-classe, je peux une fois de plus utiliser le mot clé parent et l'opérateur ::.

Formateur.php (modification de la méthode parler())

```
1 <?php
2
3 // Formateur.php
4 class Formateur extends Personne {
5
6     private $specialite;
7
8     function __construct($nom, $prenom, $specialite) {
9         parent::__construct($nom, $prenom);
10        $this->specialite = $specialite;
11    }
12
13    public function getSpecialite() {
14        return $this->specialite;
15    }
16
17    public function setSpecialite($specialite) {
18        $this->specialite = $specialite;
19    }
20
21    public function enseigner() {
22        return 'j\'enseigne';
23    }
24 }
```

```
25     public function parler() {  
26         $msg = parent::parler();  
27         return "$msg le PHP est un langage...";  
28     }  
29  
30 }
```

Le lanceur n'est toujours pas modifié. La chaîne renvoyée par la méthode `parler()` de la classe `Personne` est assignée à la variable `msg` puis la méthode `parler()` de `Formateur` renvoie une concaténation de chaînes.

Fin

NAVIGATION



Accueil

■ Ma page

Pages du site

Mon profil

Cours actuel

POO PHP

Participants

Généralités

La programmation orientée objet : premiers pas

L'héritage

Introduction

T.P. héritage

Redéfinition de méthodes

T.P. redéfinition

Les classes abstraites

T.P. véhicule

Le mot clé final

T.P. employés

Les interfaces

Le typage

Les namespaces

Les exceptions

Les bases de données avec PDO

Les tests avec PHPUnit

Petite application version 2

Petite application version 3

[Mes cours](#)

ADMINISTRATION



Administration du cours

Réglages de mon profil

Connecté sous le nom « [Arnaud Lemais](#) » ([Déconnexion](#))
[POO PHP](#)



Programmation orientée objet en PHP

[Accueil](#) ► [Mes cours](#) ► [Développement logiciel](#) ► [POO PHP](#) ► [L'héritage](#) ► [T.P. redéfinition](#)

T.P. redéfinition

Redéfinissez dans vos classes Stagiaire et Administratif la méthode parler() de Personne. La méthode parler() de Stagiaire renverra "Je parle... et j'apprends", la méthode parler() d'Administratif renverra "Je parle... et je travaille". Les méthodes parler() de Stagiaire et Administratif ré-utiliseront la méthode parler() de Personne.

Vous remettrez vos scripts dans une archive zip dont le nom sera de la forme "redefinition_*nom_prenom*.zip".

État du travail remis

| | |
|--------------------------|------------------------------|
| Numéro de tentative | Ceci est la tentative 1. |
| Statut des travaux remis | Aucune tentative |
| Statut de l'évaluation | Pas évalué |
| Dernière modification | vendredi 27 mars 2015, 15:24 |

Ajouter un travail

Modifier votre travail remis

NAVIGATION



Accueil

■ Ma page

Pages du site

Mon profil

Cours actuel

POO PHP

Participants

Généralités

La programmation orientée objet : premiers pas

L'héritage

Introduction

T.P. héritage

Redéfinition de méthodes

T.P. redéfinition

Les classes abstraites

T.P. véhicule

Le mot clé final

T.P. employés

Les interfaces

Le typage

Les namespaces

Les exceptions

Les bases de données avec PDO

Les tests avec PHPUnit

Petite application version 2

Petite application version 3

[Mes cours](#)

ADMINISTRATION



Administration du cours

Réglages de mon profil

Connecté sous le nom « [Arnaud Lemais](#) » ([Déconnexion](#))
[POO PHP](#)



Programmation orientée objet en PHP

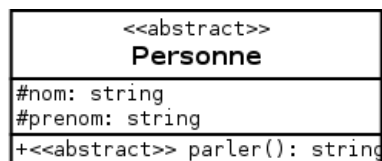
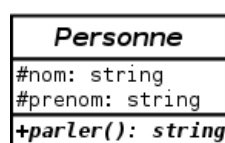
[Accueil](#) ▶ [Mes cours](#) ▶ [Développement logiciel](#) ▶ [POO PHP](#) ▶ [L'héritage](#) ▶ [Les classes abstraites](#)

Les classes abstraites

Une [classe abstraite](#) est une classe qui ne servira qu'à l'héritage, on ne pourra pas instancier cette classe. Le rôle de la classe abstraite est principalement d'être une super-classe et de créer une classe "socle" à l'héritage. Une classe abstraite peut très bien hériter d'une autre classe, et posséder des attributs et des méthodes public, protected et même private.

En PHP, pour indiquer qu'une classe est abstraite, nous utilisons le mot clé "abstract" dans la déclaration de la classe.

Dans un diagramme de classe, les classes abstraites peuvent être représentées des deux façons :



Les noms des classes et des méthodes abstraites peuvent être écrits en italique ou porter le "stéréotype" abstract ("**<<abstract>>**").

```
1 <?php
2
3 // Personne.php
4 abstract class Personne {
5
6     protected $nom;
7     protected $prenom;
8
9     function __construct($nom, $prenom) {
10         $this->nom = $nom;
11         $this->prenom = $prenom;
12     }
13
14     public function getNom() {
15         return $this->nom;
16     }
17
18     public function getPrenom() {
19         return $this->prenom;
20     }
21
22     public function setNom($nom) {
23         $this->nom = $nom;
24     }
25
26     public function setPrenom($prenom) {
27         $this->prenom = $prenom;
28     }
29
30     public function parler() {
31         return 'je parle...';
32     }
33
34 }
```

Maintenant que nous avons rendu cette classe abstraite, nous ne pouvons plus écrire ceci :

```
1 <?php
2
3 // lanceur.php
4 include './Personne.php';
5 $pers = new Personne('Wayne', 'Bruce');
```

Par contre nous pouvons toujours utiliser cette classe dans le cadre de l'héritage :

```
1 <?php
2
3 // Formateur.php
4 class Formateur extends Personne {
5
6     private $specialite;
7
8     function __construct($nom, $prenom, $specialite) {
9         parent::__construct($nom, $prenom);
10        $this->specialite = $specialite;
11    }
12
13    public function getSpecialite() {
14        return $this->specialite;
15    }
16
17    public function setSpecialite($specialite) {
18        $this->specialite = $specialite;
19    }
20
21    public function enseigner() {
22        return 'j\'enseigne';
23    }
24
25    public function parler() {
26        $msg = parent::parler();
27        return "$msg le PHP est un langage...";
28    }
29
30 }
```

lanceur.php

```
1 <?php
2
3 // lanceur.php
4 include './Personne.php';
5 include './Formateur.php';
6
7 $formateur = new Formateur('Sparrow', 'Jack', 'développement logiciel');
8 echo $formateur->parler();
```

Dans une classe abstraite, il est aussi possible de déclarer des méthodes abstraites. Ces méthodes ne comporteront pas d'implémentation (de code) dans la classe abstraite qui les déclare, mais elles devront forcément être implémentées dans les classes filles non-abstraites.

Personne.php (modification de la méthode parler())

```
1 <?php
2
3 // Personne.php
4 abstract class Personne {
5
6     protected $nom;
```

```

7     protected $prenom;
8
9     function __construct($nom, $prenom) {
10         $this->nom = $nom;
11         $this->prenom = $prenom;
12     }
13
14     public function getNom() {
15         return $this->nom;
16     }
17
18     public function getPrenom() {
19         return $this->prenom;
20     }
21
22     public function setNom($nom) {
23         $this->nom = $nom;
24     }
25
26     public function setPrenom($prenom) {
27         $this->prenom = $prenom;
28     }
29
30     public abstract function parler();
31 }

```

La méthode abstraite ne contient pas d'implémentation et donc pas de bloc {...}. Seuls les noms de la méthode et de ses paramètres sont indiqués, on appelle parfois cela la signature de la méthode.

Dans la classe Formateur nous ne pouvons plus faire appel à la méthode de la super-classe puisque la méthode de la super-classe n'a plus d'implémentation. La classe Formateur doit donc fournir une implémentation à la méthode parler(), ou devenir elle-même une classe abstraite.

Formateur.php (modification de la méthode parler())

```

1 <?php
2
3 // Formateur.php
4 class Formateur extends Personne {
5
6     private $specialite;
7
8     function __construct($nom, $prenom, $specialite) {
9         parent::__construct($nom, $prenom);
10        $this->specialite = $specialite;
11    }
12
13    public function getSpecialite() {
14        return $this->specialite;
15    }
16
17    public function setSpecialite($specialite) {
18        $this->specialite = $specialite;
19    }
20
21    public function enseigner() {
22        return 'j\'enseigne';
23    }
24
25    public function parler() {
26        return 'je parle comme un formateur...';
27    }
28
29 }

```


Le lanceur ne change pas.

Fin

NAVIGATION



[Accueil](#)

■ [Ma page](#)

[Pages du site](#)

[Mon profil](#)

[Cours actuel](#)

[POO PHP](#)

[Participants](#)

[Généralités](#)

[La programmation orientée objet : premiers pas](#)

[L'héritage](#)

 [Introduction](#)

 [T.P. héritage](#)


 [Redéfinition de méthodes](#)

 [T.P. redéfinition](#)

 [Les classes abstraites](#)

 [T.P. véhicule](#)

 [Le mot clé final](#)

 [T.P. employés](#)

[Les interfaces](#)

[Le typage](#)

[Les namespaces](#)

[Les exceptions](#)

[Les bases de données avec PDO](#)

[Les tests avec PHPUnit](#)

[Petite application version 2](#)

[Petite application version 3](#)

[Mes cours](#)

ADMINISTRATION



[Administration du cours](#)

[Réglages de mon profil](#)

Connecté sous le nom « [Arnaud Lemais](#) » ([Déconnexion](#))
[POO PHP](#)



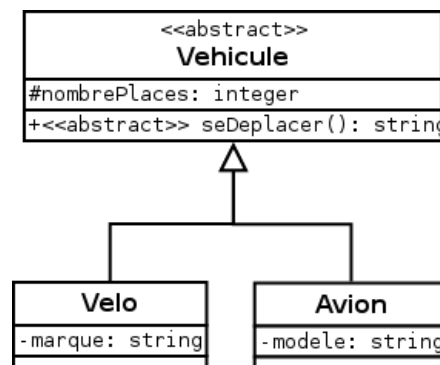
Programmation orientée objet en PHP

[Accueil](#) ▶ [Mes cours](#) ▶ [Développement logiciel](#) ▶ [POO PHP](#) ▶ [L'héritage](#) ▶ [T.P. véhicule](#)

T.P. véhicule

Créez une classe abstraite `Vehicule` possédant un attribut `nombrePlaces` et une méthode abstraite `seDeplacer()`. Créez une classe `Velo` qui héritera de la classe `Vehicule`, qui possédera un attribut `marque` et qui implémentera la méthode `seDeplacer()` : cette méthode renverra une chaîne de caractères du type "Je me déplace comme un vélo et je peux emmener " + `nombrePlaces` + " personnes".

Créez une classe `Avion` qui héritera de la classe `Vehicule`, qui possédera un attribut `modele` et qui implémentera la méthode `seDeplacer()` : cette méthode renverra une chaîne de caractères du type "Je me déplace comme un avion et je peux emmener " + `nombrePlaces` + " personnes".



Dans un lanceur, instanciez les classes `Velo` et `Avion` et appelez les méthodes `seDeplacer()` de ces instances.

Les fichiers seront à remettre dans une archive zip dont le nom sera de la forme "vehicule_*nom_prenom*.zip".

État du travail remis

| | |
|--------------------------|------------------------------|
| Numéro de tentative | Ceci est la tentative 1. |
| Statut des travaux remis | Aucune tentative |
| Statut de l'évaluation | Pas évalué |
| Dernière modification | vendredi 27 mars 2015, 15:25 |

Ajouter un travail

Modifier votre travail remis

NAVIGATION

[Accueil](#)

▪ [Ma page](#)

Pages du site

Mon profil

Cours actuel

[POO PHP](#)

Participants


Généralités




La programmation orientée objet : premiers pas

L'héritage

 [Introduction](#)

 [T.P. héritage](#)

 [Redéfinition de méthodes](#)

 [T.P. redéfinition](#)

 [Les classes abstraites](#)

 [T.P. véhicule](#)

 [Le mot clé final](#)

 [T.P. employés](#)

Les interfaces

Le typage

Les namespaces

Les exceptions

Les bases de données avec PDO

Les tests avec PHPUnit

Petite application version 2

Petite application version 3

[Mes cours](#)

ADMINISTRATION



Administration du cours

Réglages de mon profil

Connecté sous le nom « [Arnaud Lemais](#) » ([Déconnexion](#))

[POO PHP](#)



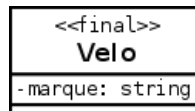
Programmation orientée objet en PHP

[Accueil](#) ► [Mes cours](#) ► [Développement logiciel](#) ► [POO PHP](#) ► [L'héritage](#) ► [Le mot clé final](#)

Le mot clé final

Le mot clé **final** utilisé dans la déclaration de la classe signifie que l'on ne peut pas hériter de cette classe. Utilisé dans une déclaration de méthode, le mot clé **final** signifie que les classes filles ne pourront pas redéfinir cette méthode.

Dans un diagramme de classe, le stéréotype <<final>> indique qu'une classe ou qu'une méthode est finale.

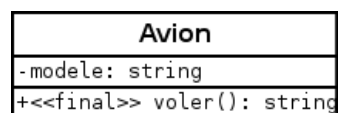


Une classe Velo rendue finale : héritage impossible.

```

1 <?php
2
3 // Velo.php
4 final class Velo {
5
6     private $marque;
7
8     function __construct($marque) {
9         $this->marque = $marque;
10    }
11
12    public function getMarque() {
13        return $this->marque;
14    }
15
16    public function setMarque($marque) {
17        $this->marque = $marque;
18    }
19
20    public function __toString() {
21        return "je suis un vélo de marque $this->marque";
22    }
23
24 }
```

Une classe Avion avec une méthode final :



```

1 <?php
2
3 // Avion.php
4 class Avion {
5
6     private $modele;
7
8     function __construct($modele) {
9         $this->modele = $modele;
10    }
```

```
11
12     public function getModele() {
13         return $this->modele;
14     }
15
16     public function setModele($modele) {
17         $this->modele = $modele;
18     }
19
20     final public function voler() {
21         return 'je vole...';
22     }
23
24 }
```

[Fin](#)

NAVIGATION





Accueil

■ Ma page

[Pages du site](#)[Mon profil](#)[Cours actuel](#)

POO PHP

[Participants](#)[Généralités](#)[La programmation orientée objet : premiers pas](#)[L'héritage](#) [Introduction](#) [T.P. héritage](#) [Redéfinition de méthodes](#) [T.P. redéfinition](#) [Les classes abstraites](#) [T.P. véhicule](#) [Le mot clé final](#) [T.P. employés](#)[Les interfaces](#)[Le typage](#)[Les namespaces](#)[Les exceptions](#)[Les bases de données avec PDO](#)[Les tests avec PHPUnit](#)[Petite application version 2](#)[Petite application version 3](#)[Mes cours](#)

ADMINISTRATION

[Administration du cours](#)[Réglages de mon profil](#)

Connecté sous le nom « [Arnaud Lemais](#) » ([Déconnexion](#))
[POO PHP](#)



Programmation orientée objet en PHP

[Accueil](#) ► [Mes cours](#) ► [Développement logiciel](#) ► [POO PHP](#) ► [L'héritage](#) ► [T.P. employés](#)

T.P. employés

Une société emploie trois types de personnel :

- les permanents au salaire fixe
- les vacataires : salaire à l'heure selon un taux horaire défini
- les commerciaux : salaire calculé à partir d'un pourcentage sur leurs ventes

Les vacataires ne peuvent travailler plus de 150 heures dans le mois. Au delà de 125 heures, leur taux horaire est majoré de 15%. Par exemple, si un vacataire a travaillé 140 heures, les 125 heures sont payées au taux horaire normal, les heures au delà (15 heures) sont payées au taux horaire normal majoré de 15 %. Chaque employé est qualifié par son nom, son prénom ainsi que son numéro de sécurité sociale.

Dans un script lanceur.php, créez un employé de chaque type et« affichez » chaque employé.
L'application devra permettre l'affichage suivant :

permanent : Michel Vaillant

num sec soc : 156045600714523

salaire mensuel : 2500,00€

payé : 2500,00€

vacataire : Astérix Le Gaulois

num sec soc : 156046602714456

taux horaire: 35,00€ ; heures travaillées: 40,00

payé : 1400,00€

commercial : Jack Sparrow

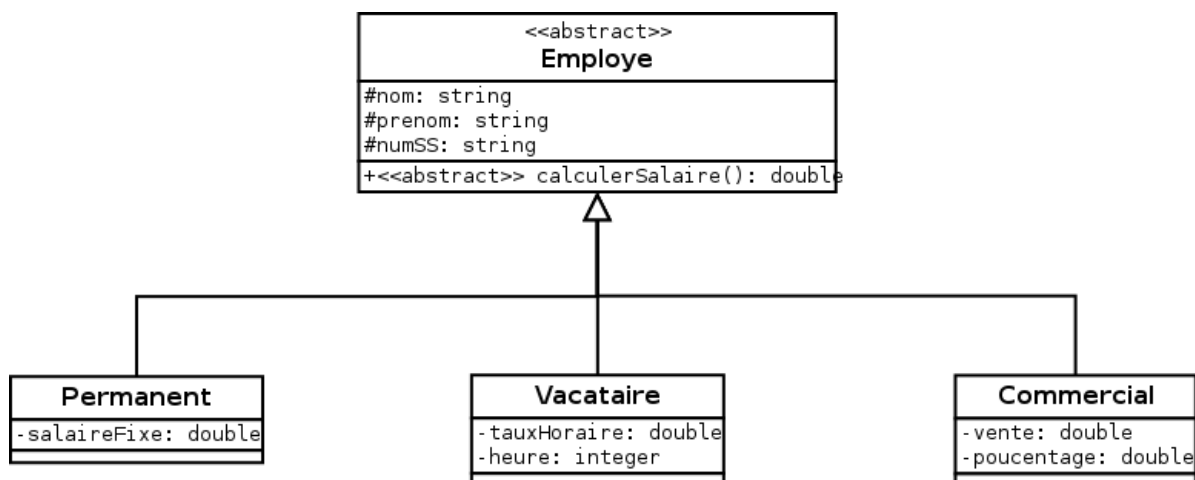
num sec soc : 145781212345696

Ventes : 450000,00€ Pourcentage : 5,00%

payé : 22500,00€

Note : Utilisez la méthode `__toString()` pour générer la chaîne de caractères à afficher, utilisez la méthode `calculerSalaire()` à l'intérieur de cette méthode `__toString()`.

Voici le diagramme de classes :



Les fichiers seront remis dans une archive zip dont le nom sera de la forme "employees_nom_prenom.zip".

État du travail remis

| | |
|--------------------------|------------------------------|
| Numéro de tentative | Ceci est la tentative 1. |
| Statut des travaux remis | Aucune tentative |
| Statut de l'évaluation | Pas évalué |
| Dernière modification | vendredi 27 mars 2015, 15:25 |

[Ajouter un travail](#)[Modifier votre travail remis](#)

NAVIGATION




Accueil

■ Ma page

[Pages du site](#)[Mon profil](#)[Cours actuel](#)

POO PHP

[Participants](#)[Généralités](#)[La programmation orientée objet : premiers pas](#)[L'héritage](#) [Introduction](#) [T.P. héritage](#) [Redéfinition de méthodes](#) [T.P. redéfinition](#) [Les classes abstraites](#) [T.P. véhicule](#) [Le mot clé final](#) [T.P. employés](#)[Les interfaces](#)[Le typage](#)[Les namespaces](#)[Les exceptions](#)[Les bases de données avec PDO](#)[Les tests avec PHPUnit](#)[Petite application version 2](#)[Petite application version 3](#)[Mes cours](#)

ADMINISTRATION

[Administration du cours](#)[Réglages de mon profil](#)

Connecté sous le nom « [Arnaud Lemais](#) » ([Déconnexion](#))
POO PHP