Héritage

Programmation Orientée Object (POO) en PHP

Chris Chevalier – chevalier@chris-freelance.com

Introduction à l'héritage

Chris Chevalier – chevalier@chris-freelance.com

L'héritage en quelques mots… (1/2)

- L'**héritage** fait partie des concepts importants de la **POO**
- **L'héritage** consiste à faire **hériter** une **classe B** d'une **classe A** - La **classe B** représente la **classe fille** et que la **classe A**, la **classe mère**
- Ainsi, la **classe fille hérite** de tous les **attributs** et **méthodes publiques** (*public*) de la **classe mère**
- Ces **méthodes** conservent leur comportement d'origine
- L'**héritage** est donc très utile pour définir des **méthodes** communes à plusieurs **classes** tout en laissant la liberté d'en créer de nouvelles dans les **classes enfants**

L'héritage en quelques mots… (2/2)

- Soit **2 classes A et B** où B **hérite** de A - Pour qu'un **héritage** soit pertinent, on doit pouvoir dire que B est un A
- Plus concrètement, on peut imaginer une **classe** *Guerrier* héritant de la **classe** *Personnage* puisqu'on peut effectivement dire qu'un guerrier est un personnage
- Dans la même idée, on pourrait faire hériter la **classe** *Chien* de la **classe** *Animal* - Un chien étant, bien entendu, un animal

L'héritage en image…

```php
<?php

class Personnage {

}

class Guerrier extends Personnage {
    /**
     * La classe Guerrier hérite des
     * attributs et méthodes publiques
     * de la classe Personnage
     */
}

?>
```

Explications

- Pour faire **hériter** une **classe** des **attributs** et **méthodes** d'une autre **classe** en **PHP**, il suffit de placer le mot-clé *extends* suivi du nom de la **classe mère** après la définition de la **classe fille** (*class MaClasseFille extends MaClasseMere { }*)

- Comme dans la réalité, une mère peut avoir plusieurs filles mais une fille ne peut avoir qu'une unique mère

- Ainsi, nous pouvons créer plusieurs autres **classes** héritant de la **classe** *Personnage* - Cette dernière servira ainsi de modèle à toutes les autres puisqu'elles **hériteront** de tous les **attributs** et **méthodes** de la **classe** *Personnage*

L'héritage en image…

```
3   class Personnage {
4
5   }
6
7   class Guerrier extends Personnage {
8
9   }
10
11  class Magicien extends Personnage {
12
13  }
14
15  class Voleur extends Personnage {
16
17  }
```

La force de l'héritage en quelques mots…

- Tout l'intérêt de l'**héritage** réside dans le fait que chaque **classe fille** peut créer des **attributs** et **méthodes** qui lui sont propres en plus de celles **héritées** de la **classe mère**

- En pratique, il suffit de créer des **attributs** et **méthodes** au sein de la **classe fille** comme nous le faisions jusqu'ici

- Imaginons maintenant que nous souhaitons permettre à un magicien de pouvoir lancer un sort pour infliger des dégâts proportionnels à sa puissance magique

- Bien sûr, il ne s'agit pas de caractéristiques et d'actions que peut faire un personnage donc l'héritage prend tout son sens

La force de l'héritage en image…

```php
class Magicien extends Personnage {
  private $magie;

  public function lancerUnSort($perso)
  {
    $perso->recevoirDegats($this->magie);
  }
}
```

- En plus des **attributs** et **méthodes** de la **classe** *Personnage*, la **classe** *Magicien* a maintenant accès à l'**attribut** *$magie* et à la **méthode** *lancerUnSort()*

La redéfinition de méthodes (1/2)

- Une autre force de **l'héritage** réside dans la possibilité de **réécrire** une **méthode héritée** afin de modifier son comportement

- En pratique, il suffit de déclarer et implémenter à nouveau la **méthode** dans la **classe fille**

- Imaginons que nous souhaitons **récrire** la **méthode** *attaquer()* dans la **classe** *Magicien* afin d'y ajouter des spécificités propres aux magiciens - Pour autant, on veut conserver le contenu de la méthode de la classe parente

- Nous ne pouvons pas réécrire manuellement le contenu de la **méthode** *attaquer()* car elle utilise l'**attribut privé** *$nom*

La redéfinition de méthodes (2/2)

- Heureusement, il existe une solution permettant d'appeler une **méthode** d'une **classe parente** dans une **classe fille**

- Pour cela, il suffit d'utiliser le mot-clé *parent* suivi de l'**opérateur de résolution de portée** (« :: ») puis de la **méthode** issue de la **classe parente** à appeler (*parent::methodeDeLaClasseMere()*)

```php
class Personnage {
  private $nom;
  private $pv;

  public function attaquer($perso) {
    echo $this->nom . " attaque " . $perso->nom;
  }

  public function recevoirDegats($degats) {
    $this->pv -= $degats;
  }
}

class Magicien extends Personnage {
  private $magie = 50;

  public function attaquer($perso) {
    parent::attaquer($perso);
    $this->lancerUnSort($perso);
  }

  public function lancerUnSort($perso) {
    $perso->recevoirDegats($this->magie);
  }
}
```

Explications

- Ici, on souhaite utiliser et **réécrire** la **méthode** *attaquer()* de la **classe** *Personnage* dans sa **classe fille** *Magicien*

- Pour ce faire, on déclare à nouveau la **méthode** au sein de la **classe fille**, on fait appel à la **méthode** issue de la **classe mère** grâce au mot-clé *parent* puis on ajoute les instructions propres à la **classe fille**

- Ainsi, appeler la **méthode** *attaquer()* depuis un **objet** de type *Magicien* va afficher à l'écran un message du type « ***Gandalf attaque Aragorn*** » avant de lui infliger des dégâts grâce à l'appel de la **méthode** *lancerUnSort()*

- Notez que si la **méthode parente** retourne une valeur, on peut la récupérer comme on le ferait avec un appel sur une **méthode** standard

```php
class A
{
    public function test()
    {
        return 'test';
    }
}

class B extends A
{
    public function test()
    {
        $retour = parent::test();
        echo $retour;
    }
}

$b = new B;
$b->test(); // Affiche "test"
```

Complément sur la visibilité

- Notez qu'une **méthode parente** redéfinit dans une **classe enfant** doit avoir la même **visibilité** ou une **visibilité plus permissive** que la **méthode** d'origine

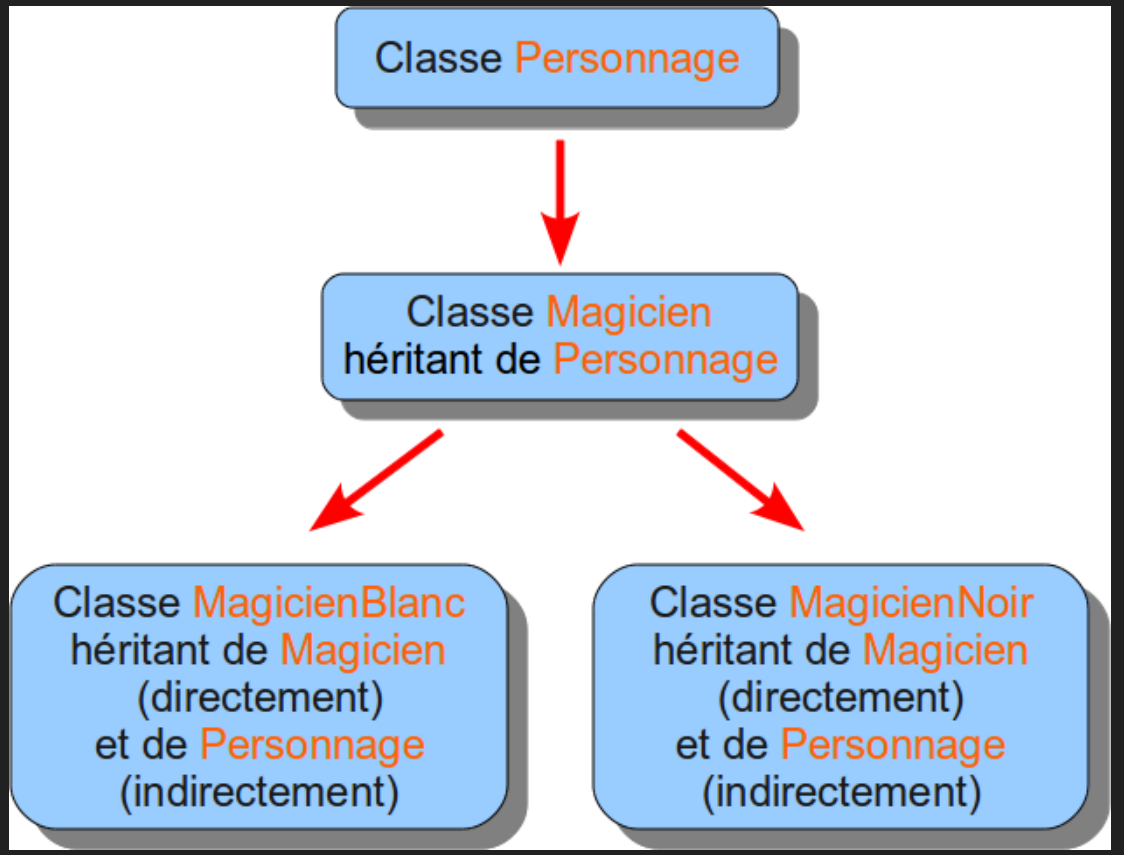
- Ainsi, il est possible de **redéfinir** une **méthode parente privée** en **publique** mais l'inverse n'est pas permis

Un arbre généalogique

- En **POO**, toutes les **classes** peuvent être **héritées**
- On peut ainsi créer un véritable **arbre généalogique** avec de très nombreuses **classes héritant** les unes des autres
- Dans notre cas, on pourrait imaginer une division de la **classe** *Magicien* en 2 nouvelles **classes** : *MagicienNoir* et *MagicienBlanc*
- Ainsi, ces 2 **classes héritent** à la fois des **attributs** et **méthodes** des **classes** *Magicien* et *Personnage*

Un arbre généalogique

```
3   class Personnage {
4
5   }
6
7   class Magicien extends Personnage {
8
9   }
10
11  class MagicienNoir extends Magicien {
12
13  }
14
15  class MagicienBlanc extends Magicien {
16
17  }
```



La visibilité protégée

Héritage

La visibilité protected en quelques mots…

- Jusqu'ici, nous avons parlé uniquement de **2 types de visibilité** : *public* et *private*
- Il en existe pourtant une troisième : *protected*
- Au niveau restrictif, *protected* se place entre *public* et *private*
- En effet, les **attributs** et **méthodes** *protected* d'une **classe** ne sont accessibles que depuis celle-ci ET ses **classes filles**

```php
class Personnage {
  protected $magie;
  private $nom;

  public function __construct() {
    $this->magie = 50;
    $this->nom = 'Gandalf';
  }
}

class Magicien extends Personnage {
  public function afficherInfos() {
    echo $this->magie; // L'attribut est protégé, on y a donc accès
    echo $this->nom; // L'attribut est privé, on n'y a pas accès donc rien ne s'affichera
  }
}

$gandalf = new Magicien();

echo $gandalf->magie; // Erreur fatale
echo $gandalf->nom; // Rien ne s'affiche

$gandalf->afficherInfos(); // Affiche uniquement « 50 »
```

Abstraction

Héritage

Les classes abstraites en quelques mots…

- Il est possible d'imposer certaines contraintes dans **l'héritage** - On parlera alors **d'abstraction** ou de **finalisation** en fonction de la contrainte à mettre en place

- Nous avons découvert précédemment, que toutes les **classes** que nous déclarons peuvent être **instanciées** pour exploiter leurs **attributs** et **méthodes**

- Définir une **classe** comme **abstraite** va nous permettre d'empêcher toute **instanciation** de cette **classe**

- On ne pourra donc pas se servir directement de la **classe** en tant que telle et il nous faudra la faire **hériter** pour exploiter ses **attributs** et **méthodes** à travers sa **classe fille**

Les classes abstraites en quelques mots…

- Reprenons notre exemple avec la **classe** *Personnage* et ses **classes filles** *Guerrier*, *Magicien* et *Voleur*
- La **classe** *Personnage* représentant un modèle pour ses 3 **classes filles**, nous ne créerons jamais d'**objet** de type *Personnage* mais directement de type *Guerrier*, *Magicien* ou *Voleur*
- Quand on parle de modèle, il peut être intéressant de déclarer la **classe** comme **abstraite**
- Pour déclarer une **classe abstraite**, il suffit de faire précéder le mot-clé *class* par *abstract* (*abstract class MaClass { }*)

Les classes abstraites en image…

```
3   abstract class Personnage { // La classe Personnage est abstraite
4
5   }
6
7   class Magicien extends Personnage { // La classe Magicien hérite de la classe Personnage
8
9   }
10
11  $magicien = new Magicien;
12  $perso = new Personnage; // Cette ligne produit une erreur car on instancie une classe abstraite
```

Les méthodes abstraites en quelques mots…

- Déclarer une **méthode abstraite** permet de forcer toutes les **classes filles** à écrire cette **méthode**
- En cas d'oubli, une **erreur PHP** est levée
- Étant donnée que l'on force ses **classes filles** à implémenter la **méthode abstraite**, elle n'attend aucune instruction
- Syntaxiquement, il suffit d'écrire uniquement le **prototype de la méthode** suivi d'un point-virgule pour conclure l'instruction (*asbtract public function maFonction($monParametre);*)

Les méthodes abstraites en image…

```php
abstract class Personnage {
  // On force toutes les classes filles à écrire cette méthode
  abstract public function attaquer(Personnage $perso);

  public function recevoirDegats() { // Cette méthode n'a pas besoin d'être réécrite
    // Instructions
  }
}

class Magicien extends Personnage {
  // La méthode frapper() a le même type de visibilité que la méthode abstraite
  public function attaquer(Personnage $perso) {
    // Instructions
  }
}
```

- Notez qu'une **méthode abstraite** ne peut être définie que dans une **classe abstraite**

Finalisation

Héritage

Les classes finales en quelques mots…

- La **finalisation** est le concept inverse de **l'abstraction**
- En effet, une **classe** déclarée **finale** ne pourra pas être **héritée** de **classe fille**
- En pratique, une **classe finale** est déclarée à l'aide du mot-clé *final* (*final class MaClasse { }*)

Les classes finales en image…

```
3   abstract class Personnage { // La classe abstraite sert de modèle.
4
5   }
6
7   final class Magicien extends Personnage { // On ne peut pas créer de classe héritant de Magicien
8
9   }
10
11  class MagicienNoir extends Magicien { // Une erreur est levée car on ne peut hériter d'une classe finale
12
13  }
```

Les méthodes finales en quelques mots…

- Inversement aux **méthodes abstraites**, les **méthodes finales** bloquent leur **redéfinition** depuis les **classes filles**

- En d'autres termes, une **méthode** déclarée **finale** ne peut pas être **redéfinie**

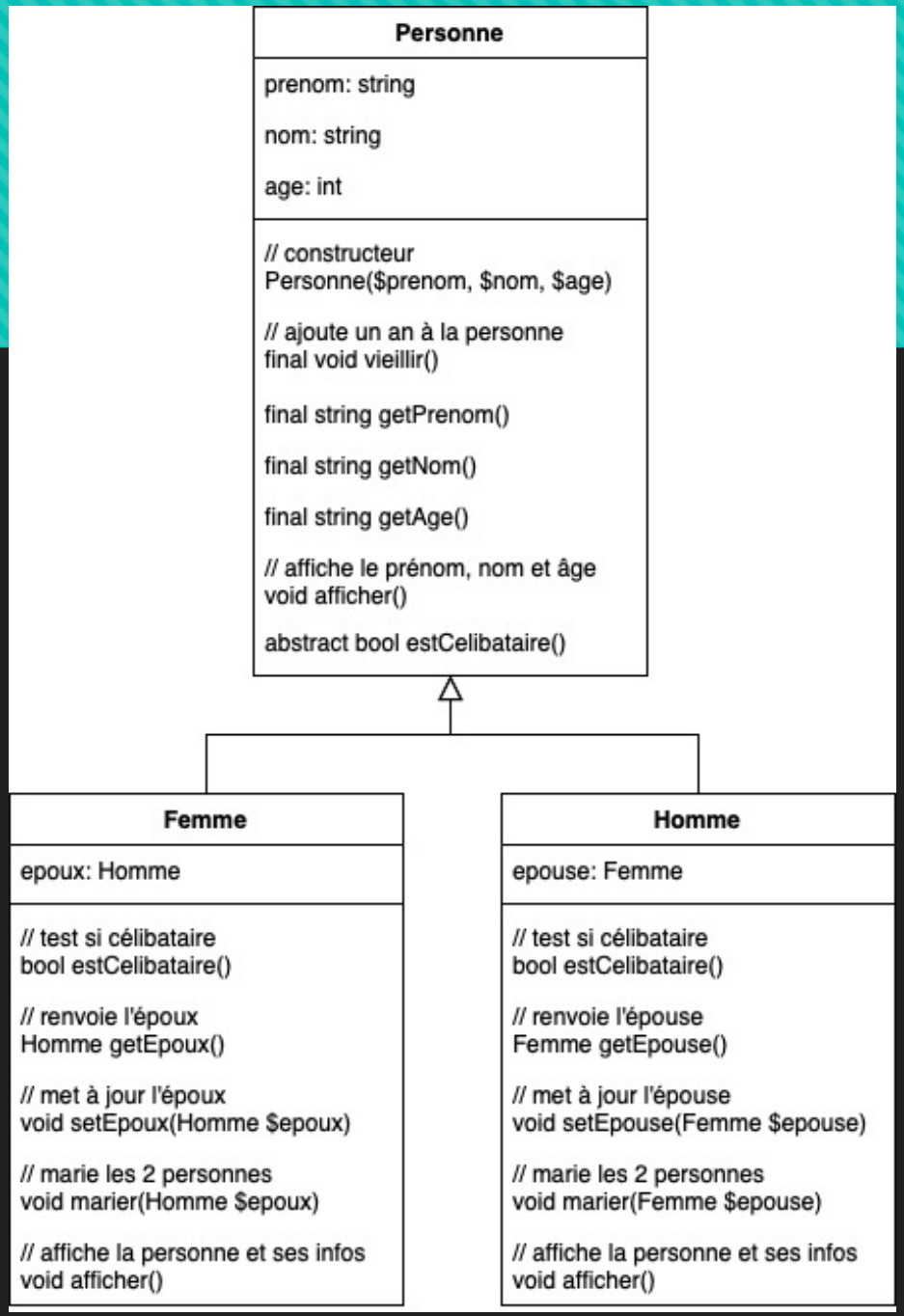
```php
abstract class Personnage {
  public function attaquer(Personnage $perso) {
    // Instructions
  }

  final public function recevoirDegats($degats) {
    // Instructions
  }
}

class Magicien extends Personnage {
  // Aucun problème pour redéfinir une méthode standard
  public function attaquer(Personnage $perso) {
    // Instructions
  }

  // Une erreur est levée car la méthode est finale dans la classe parente
  public function recevoirDegats($degats) {
    // Instructions
  }
}
```

TP - Mariage heureux

Héritage

TP - Mariage heureux

- Créez une **classe abstraite** *Personne* et 2 **classes finales** *Homme* et *Femme* en respectant la structure suivante


Personne
prenom: string
nom: string
age: int
// constructeur
Personne($prenom, $nom, $age)
// ajoute un an à la personne
final void vieillir()
final string getPrenom()
final string getNom()
final string getAge()
// affiche le prénom, nom et âge
void afficher()
abstract bool estCelibataire()
Femme
epoux: Homme
// test si célibataire
bool estCelibataire()
// renvoie l'époux
Homme getEpoux()
// met à jour l'époux
void setEpoux(Homme $epoux)
// marie les 2 personnes
void marier(Homme $epoux)
// affiche la personne et ses infos
void afficher()
Homme
epouse: Femme
// test si célibataire
bool estCelibataire()
// renvoie l'épouse
Femme getEpouse()
// met à jour l'épouse
void setEpouse(Femme $epouse)
// marie les 2 personnes
void marier(Femme $epouse)
// affiche la personne et ses infos
void afficher()

TP - Mariage heureux

- La **méthode** *afficher()* des **classes filles** *Homme* et *Femme* **redéfinit** la **méthode parente** en conservant son contenu (indice : *parent*)

- Ainsi cette **méthode** doit afficher une chaîne de caractère de type : « **Meghan a 37 ans. Elle est célibataire.** » ou « **Meghan a 37 ans. Elle est mariée à Harry.** »

- La **méthode** *setEpoux()* sert à marier les 2 personnes entre elles en l'appelant au sein de la **méthode** *marier()*

- Utilisez l'**auto-chargement de classes** (Cf. cours **2 - Classes et objets**) pour tester les différentes **méthodes** sur des **objets** issus des **classes** *Homme* et *Femme* dans un fichier *Main.php*

TP - Mariage heureux

- Envoyez vos sources dans un dossier compressé à l'adresse email chevalier@chris-freelance.com

Des questions ?