II. Programmation Orientée Objet - Suite

1. Les Méthodes et Propriétés Statiques

Les méthodes et propriétés statiques sont des éléments particuliers d'une classe. Contrairement aux méthodes et propriétés classiques, qui appartiennent à une instance (un objet) de la classe, les méthodes et propriétés statiques appartiennent à la classe elle-même. Cela signifie que vous pouvez y accéder sans avoir besoin de créer une instance de la classe.

1.1. Définition d'une propriété ou méthode statique

Pour définir une propriété ou une méthode statique, il suffit d'ajouter le mot-clé static après l'instruction de visibilité (public, private, protected) :

```
public static $property = "value";
```

Les méthodes et propriétés statiques sont particulièrement utiles lorsque vous avez des valeurs ou des comportements qui ne sont pas spécifiques à une instance d'objet, mais qui doivent être partagés par toutes les instances de la classe. Par exemple, si vous souhaitez compter le nombre d'objets créés d'une certaine classe, une propriété statique serait idéale.

Exemple :

Voici un exemple qui montre comment utiliser une propriété et une méthode statique dans une classe :

```
class Compteur {
   // Propriété statique qui garde une trace du nombre d'objets créés
   public static $compteur = 0;

   // Constructeur : chaque fois qu'un objet est créé, on incrémente le compteur
   public function __construct() {
       self::$compteur++;
   }

   // Méthode statique : on peut accéder au compteur sans créer d'objet
   public static function obtenirCompteur() {
       return self::$compteur;
   }
}

// Création d'objets
$obj1 = new Compteur();
$obj2 = new Compteur();
```

```php
$obj3 = new Compteur();

// Accès à la propriété statique via la méthode statique
echo "Nombre d'objets créés : " . Compteur::obtenirCompteur(); // Affiche 3
```

Dans cet exemple, la propriété compteur est statique, donc elle est partagée entre toutes les instances de la classe Compteur. La méthode obtenirCompteur() est également statique, ce qui permet d'accéder au nombre d'objets créés sans avoir besoin de créer un objet spécifique pour cela.

1.2. Accéder à une propriété ou méthode statique

Il existe deux manières d'accéder à une méthode ou propriété statique, selon que vous y accédez depuis la classe elle-même ou depuis une instance de la classe :

Depuis la classe elle-même :
Vous utilisez le mot-clé self, suivi de :: puis du nom de la méthode ou de la propriété. Par exemple :

```php
self::$compteur;
self::obtenirCompteur();
```

Depuis une instance :
Dans ce cas, vous utilisez le nom de la classe suivi de ::

```php
Compteur::$compteur;
Compteur::obtenirCompteur();
```

2. L'Héritage en Programmation Orientée Objet

L'héritage est un concept fondamental de la programmation orientée objet qui permet à une classe d'hériter des propriétés et des méthodes d'une autre classe. Dans ce mécanisme, la classe qui hérite est appelée la classe dérivée (ou classe enfant), et la classe dont elle hérite est appelée la classe parente. Cela permet de réutiliser le code existant sans avoir à le réécrire, et de créer des relations entre des classes. Une classe dérivée peut alors ajouter ses propres caractéristiques ou modifier certaines méthodes de la classe parente.

Par exemple, précédemment nous avons créer une classe Voiture. Imaginons, que l'on veuille maintenant créer une classe Moto, on pourrait la créer comme nous l'avons fait avec la classe Voiture. Mais on voit bien que ces deux classes possèdes des caractéristiques identiques tel que la couleur, la marque et la vitesse maximale. Au lieu de réécrire ce code, nous pouvons

créer une classe parente appelé Véhicule, qui implémente les propriété et méthodes qui sont communes aux voitures et aux motos.

2.1. Création de la classe parente

La classe parente définit des propriétés et des méthodes communes. C'est dans cette classe que l'on place tout ce qui est commun à plusieurs autres classes.

```php
// Classe parente : Vehicule
class Vehicule {
   public $couleur;
   public $marque;
   public $vitesseMax;


   // Constructeur pour initialiser les propriétés de la classe
   public function __construct($couleur, $marque, $vitesseMax) {
       $this->couleur = $couleur;
       $this->marque = $marque;
       $this->vitesseMax = $vitesseMax;
   }

   // Méthode commune à tous les véhicules
   public function demarrer() {
       echo "Le véhicule démarre.<br>";
   }
}
```

2.2. Création des classes dérivées

Le mot-clé extends

Lorsqu'une classe dérivée hérite d'une classe parente, nous utilisons le mot-clé extends. Ce mot-clé permet à la classe dérivée d'accéder aux propriétés et aux méthodes de la classe parente. En d'autres termes, la classe dérivée étend la classe parente, en ajoutant ses propres fonctionnalités tout en réutilisant celles de la classe parente.

Dans notre exemple, nous avons une classe parente Vehicule et nous créons des classes dérivées comme Voiture et Moto en utilisant extends pour dire à PHP que ces classes vont hériter des caractéristiques de Vehicule.

```php
class Voiture extends Vehicule { }
```

L'appel au constructeur de la classe parente

En programmation orientée objet, lorsque vous créez une classe dérivée, cette classe hérite des propriétés et des méthodes de la classe parente. Cependant, si la classe parente possède un constructeur, vous devez explicitement l'appeler dans la classe dérivée pour que les propriétés de la classe parente soient correctement initialisées.

Dans notre exemple, la classe parente Vehicule possède un constructeur qui permet d'initialiser les propriétés communes à tous les véhicules, comme couleur, marque et vitesseMax.

```
public function __construct($couleur,$marque, $vitesseMax) {
  $this->couleur = $couleur;
  $this->marque = $marque;
  $this->vitesseMax = $vitesseMax;
}
```

Lorsque l'on crée la classe dérivée Voiture, cette dernière hérite des propriétés de Vehicule. Cependant, pour que ces propriétés soient correctement initialisées, nous devons appeler explicitement le constructeur de la classe parente dans le constructeur de la classe dérivée. Cela garantit que les propriétés communes à tous les véhicules seront bien initialisées avant de gérer les propriétés spécifiques à la classe dérivée.

```
public function __construct($couleur, $marque, $vitesseMax) {
   // Appel du constructeur de la classe parente
   parent::__construct($couleur, $marque, $vitesseMax);
}
```

Le mot-clé parent:: permet d'accéder aux méthodes et aux constructeurs de la classe parente. Ici, nous l'utilisons pour appeler le constructeur de la classe Vehicule et lui transmettre les paramètres couleur, marque, et vitesseMax.

Cela garantit que le constructeur de Vehicule s'exécute et initialise les propriétés héritées avant que la classe dérivée Voiture ne gère ses propres propriétés spécifiques.

Le constructeur de la classe dérivée doit toujours recevoir en paramètre les propriétés de la classe parente, puis les transmettre au constructeur de la classe parente à l'aide de parent::__construct().

Une fois cela fait, le constructeur de la classe dérivée peut initialiser des propriétés supplémentaires qui lui sont spécifiques.

Les classes Voiture et Moto

Grâce à l'héritage, nous avons vu que les classes dérivées héritent des propriétés et méthodes de la classe parente et que lorsqu'elle dispose d'un constructeur, les classes dérivées doivent l'appeler explicitement.

En plus d'hériter, il est possible aux classes dérivées d'implémenter leurs propre propriétés et méthodes.

Par exemple, la classe Voiture pourrais avoir une propriété supplémentaire nombreDePortes qui sera initialisé dans son constructeur et une méthode klaxonner :

```
class Voiture extends Vehicule {
   public $nombreDePortes;

   public function __construct($couleur, $marque, $vitesseMax, $nombreDePortes) {

       parent::__construct($couleur, $marque, $vitesseMax);

       $this->nombreDePortes = $nombreDePortes;
   }

   public function klaxonner() {
       echo "La voiture klaxonne !<br>";
   }
}
```

La classe Moto fonctionne de manière similaire. Elle hérite des propriétés communes à tous les véhicules, mais ajoute une propriété spécifique : typeDeMoteur et une méthode faireUnTour :

```
class Moto extends Vehicule {
   public $typeDeMoteur;

   public function __construct($couleur, $marque, $vitesseMax, $typeDeMoteur) {

       parent::__construct($couleur, $marque, $vitesseMax);

       $this->typeDeMoteur = $typeDeMoteur;
   }

   public function faireUnTour() {
       echo "La moto fait un tour.<br>";
   }
}
```

3. Les classes Abstraites

3.1. Qu'est-ce qu'une classe abstraite ?

Une classe abstraite est un type particulier de classe en programmation orientée objet. Ce qui la distingue, c'est qu'elle ne peut pas être instanciée directement. En d'autres termes, vous ne pouvez pas créer d'objets à partir d'une classe abstraite.

Une classe abstraite sert de modèle de base pour d'autres classes. Elle vous permet de définir des comportements communs qui seront partagés entre plusieurs classes dérivées. Cependant, certains comportements devront être complétés ou personnalisés par les classes qui héritent de la classe abstraite. Cela nous permet de garantir que certaines méthodes doivent être présentes dans les classes dérivées, mais avec une logique spécifique pour chaque sous-classe.

3.2. Méthodes abstraites

Une **méthode abstraite** est une méthode déclarée dans une classe abstraite qui n'a pas de corps. Cela signifie que cette méthode ne contient aucune implémentation dans la classe abstraite. Les classes dérivées doivent obligatoirement implémenter cette méthode avec leur propre logique.

Cela permet de définir un contrat : les classes dérivées doivent définir cette méthode. Une méthode abstraite permet de dire : "Toutes les classes qui héritent de cette classe devront définir cette méthode, mais c'est à elles de décider comment elle sera implémentée."

3.3. Quand utiliser une classe abstraite ?

Les classes abstraites sont utiles dans les situations où vous voulez définir des comportements généraux dans une classe de base, mais que vous souhaitez forcer les classes dérivées à compléter certains comportements spécifiques.

Cela permet de structurer votre code de manière claire, en définissant des méthodes essentielles que toutes les classes dérivées devront avoir, tout en laissant chaque classe dérivée choisir comment ces méthodes doivent fonctionner.

3.4. Exemple

Imaginons que nous avons une classe abstraite Animal, qui définit un comportement commun pour tous les animaux. Chaque type d'animal, comme un chien ou un chat, devra implémenter sa propre version de la méthode parler().

Classe abstraite Animal

```php
abstract class Animal {
   // Méthode abstraite : chaque sous-classe doit l'implémenter
   abstract public function parler();

   // Méthode normale : elle est déjà définie et peut être utilisée par les sous-classes
   public function dormir() {
       echo "L'animal dort.<br>";
   }
}
```

Pour créer une classe abstraite, on place le mot-clé abstract avant le mot-clé class. cette classe nous définissions une méthode parler mais qui n'a pas d'implémentation, elle n'a pas de corps. Chaque classe qui héritera de cette classe abstraite devra implémenter cette méthode comme elle le souhaite.

En revanche, la méthode dormir() est une méthode normale : elle contient déjà une implémentation. Toutes les classes dérivées pourront l'utiliser telle quelle, sans avoir besoin de la réécrire.

Classes dérivées : Chien et Chat

Maintenant, créons deux classes dérivées de Animal, une pour Chien et une pour Chat. Ces classes vont **implémenter la méthode parler()** de manière spécifique à chaque type d'animal.

```php
class Chien extends Animal {
   public function parler() {
       echo "Le chien aboie !<br>";
   }
}

class Chat extends Animal {
   public function parler() {
       echo "Le chat miaule.<br>";
   }
}
```

3.5. Règles d'implémentation des classes abstraites

1. Une classe abstraite ne peut pas être instanciée directement. Vous ne pouvez pas créer un objet à partir d'une classe abstraite, mais vous pouvez créer des objets à partir des classes qui héritent de cette classe abstraite.

2. Les méthodes abstraites doivent obligatoirement être implémentées dans les classes dérivées. Cela garantit que chaque sous-classe définira son propre comportement pour ces méthodes.

3. Une classe abstraite peut aussi contenir des méthodes normales, c'est-à-dire des méthodes qui ont un corps. Les classes dérivées peuvent réutiliser ces méthodes ou les remplacer.

4. Les interfaces

4.1. Qu'est-ce qu'une interface ?

Une interface définit un contrat que les classes doivent respecter. Contrairement à une classe abstraite, une interface ne contient pas d'implémentation. Elle ne fait que déclarer des méthodes, sans fournir de corps pour ces méthodes.

Les classes qui implémentent cette interface doivent obligatoirement fournir une implémentation pour toutes les méthodes qu'elle déclare. Cependant, les classes sont libres de définir leur propre logique pour chaque méthode. Cela signifie que l'interface garantit qu'une classe va respecter un certain contrat, mais sans imposer la manière de le faire.

4.2. Quand utiliser une interface ?

Les interfaces sont particulièrement utiles lorsque vous voulez garantir qu'un ensemble de classes possède certaines méthodes communes, mais que vous ne voulez pas imposer de détails sur la façon de les implémenter. Elles permettent de créer des structures flexibles où différentes classes partagent une interface, tout en conservant leur propre logique d'implémentation.

Cela favorise une approche modulaire et permet de définir des comportements communs dans des contextes différents, sans se soucier des détails spécifiques de chaque classe.

4.3. Mise en œuvre

Créer une interface

Pour créer une interface, on procède de la même manière que lors de la création d'une classe. Au lieu d'utiliser le mot class on utilise le mot interface :

```php
interface AnimalInterface {
   public function parler();
}
```

À l'intérieur de l'interface, nous déclarons des méthodes sans fournir d'implémentation. Ces méthodes devront obligatoirement être implémentées par les classes qui vont implémenter cette interface.

Utiliser une interface

On utilise une interface au moment de créer une classe comme pour l'héritage, mais cette fois il faut utiliser le mot-clé implements :

```php
class Chien implements AnimalInterface {
   public function parler() {
       echo "Le chien aboie !<br>";
   }
}
```

4.4. Règles d'implémentation des interfaces

Une classe qui implémente une interface doit implémenter toutes ses méthodes. Cela garantit que toutes les classes qui respectent une interface fournissent un comportement cohérent pour les méthodes spécifiées.

Les interfaces ne peuvent pas contenir de propriétés ni d'implémentations de méthodes. Une interface déclare uniquement des méthodes. Elle ne peut pas avoir de propriétés ni de logique dans les méthodes. Ce rôle est laissé aux classes qui implémentent l'interface.

5. TP – SUITE

Dans cette deuxième partie de votre jeu de rôle, vous allez approfondir les concepts vus dans ce cours.

5.1. Étapes à suivre

1. **Modifier la classe Personnage pour la rendre abstraite**
   - Transformez la classe Personnage en une classe abstraite.
   - Déplacez toutes les propriétés et méthodes communes à tous les personnages dans cette classe.
   - Identifiez les méthodes qui peuvent être utilisées différemment selon le type de personnage et déclarez-les comme abstraites :

2. **Ajouter des propriétés et méthodes statiques**
   - Dans la classe abstraite Personnage, ajoutez une propriété statique pour compter le nombre total de personnages créés.
   - Ajoutez une méthode statique pour afficher le nombre total de personnages.

3. **Créer des classes dérivées pour Guerrier et Orc**
   - Créez deux classes dérivées, Guerrier et Orc, qui héritent de la classe abstraite Personnage.
   - Implémentez les méthodes abstraites dans chaque classe en fonction de leurs comportements spécifiques.

4. **Ajouter de nouveaux types de personnages : Magicien et Troll**
   - **Classe Magicien** :
     - Ajoutez une propriété spécifique pour définir le type de magie utilisé par le magicien (par exemple : feu, glace, télépathie).
   - **Classe Troll** :
     - Ajoutez une capacité spéciale de régénération pour le troll. Après chaque tour, le troll regagne un certain pourcentage de sa santé, sauf si sa santé est déjà au maximum.

5. **Créer une interface pour les magiciens**
   - Créez une interface MagieInterface qui définit une méthode lancerSort().
   - Faites en sorte que la classe Magicien implémente cette interface.
   - La méthode lancerSort() devra être implémentée dans la classe Magicien avec une logique spécifique.

Scénarios

- Créez un Guerrier, un Orc, un Magicien, et un Troll en utilisant les nouvelles classes dérivées.
- Utilisez les propriétés et méthodes statiques pour afficher le nombre total de personnages créés.
- Organisez un combat entre les personnages, en mettant en œuvre leurs capacités spécifiques :
  - Le magicien peut utiliser lancerSort pour infliger des dégâts supplémentaires.
  - Le troll peut se régénérer après chaque tour.
  - Les guerriers et les orcs utilisent leur force brute pour attaquer.

Bonus
- Ajoutez méthode sortDeSoin() à la classe Magicien qui permet de se soigner.
- Faites en sorte que la classe Troll puisse infliger des dégâts supplémentaires lorsqu'il est attaqué comme capacité de riposte.