PHP

OOP Avancé

Dans ce module

- Parents et enfants
- Surcharge
- Types de classe
- Namespace
- Namespace et Use
- Pratique

Parents et enfants (extension de classe)

- On peut étendre une classe "enfant" avec une classe "parente" pour limiter les doublons et pour avoir tout un système hiérarchique pyramidal.
- **Héritage** : les parents partages tout leur attributs avec leur enfants.
- "**extends**" défini la relation de parenté.
- **Un enfant peut lui-même être parent.** Il n'y pas de limite à cette hiérarchique pyramidal.
- Un enfant ne peut avoir qu'**un seul parent** direct.

```php
class Parent {
    public $property_1;
    protected $property_2;
}

class FirstChild extends Parent {
    public $property_3;
}

class SecondChild extends Parent {
    public $property_4;
}
```

→ On parle aussi de lien :
classe fille -> classe mère

Parents et enfants - Exemple

```php
class Animal {
    public static $type;
    public $name;

    public function getName()
    {
        return $this->name;
    }
}

class Dog extends Animal {}

class Cat extends Animal {}
```

```php
$animal = new Animal();
$animal->name = 'Floppy';
$animal::$type = 'animal';

$dog = new Dog();
$dog->name = 'Jango';
$dog::$type = 'chien';

$cat = new Cat();
$cat->name = 'Chipie';
$cat::$type = 'chat';

echo $animal->getName(); // Floppy
echo $dog->getName(); // Jango
echo $cat->getName(); // Chipie

echo Animal::$type;    // chat
echo Dog::$type;       // chat
echo Cat::$type;       // chat
echo $animal::$type;   // chat
echo $dog::$type;      // chat
echo $cat::$type;      // chat
```

Surcharge (**Override**)

- Un enfant peut redéfinir une propriété d'un parent pour le surcharger (constantes inclus).
- Mise à part modifier la valeur de démarrage, il n'y a pas d'intérêt à surcharger un attribut statique. Il y en a pour les méthodes statiques.
- En plus de la propriété qu'il a surchargée, un enfant peut toujours accéder à la version d'origine du parent.

```php
class Parent {
    public $property = Value';
    public function foo()
    {
        // CODE A
    }
}

class  FirstChild  extends Parent {
    public $property = 'NewValue';
    public function  foo()
    {
         // CODE B
    }
}
```

```php
class Parent {
    public function foo()
    {
        // CODE A
    }
}

class  FirstChild  extends Parent {
    public function  foo()
    {
         parent::foo(); // CODE A
         // CODE B
    }
}
```

Surcharge – Exemple 1

```php
class Animal {
    public $name = 'Default';

    public function getName()
    {
        return strtolower($this->name);
    }
}

class Dog extends Animal {
    public function getName()
    {
        return strtoupper($this->name);
    }
}

class Cat extends Animal {
    //
}

class Rabbit extends Animal {
    public $name = 'Napoléon';
}
```

→ Une surcharge peut servir à **supplanter** le comportement d'une propriété parente

```php
$dog = new Dog();
$dog->name = 'Jango';

$cat = new Cat();
$cat->name = 'Chipie';

$rabbit = new Rabbit();

echo $dog->getName(); // (Jango =>) JANGO
echo $cat->getName(); // (Chipie =>) chipie
echo $rabbit->getName(); // (Default =>) napoléon
```

Surcharge – Exemple 2

```php
class Mother {
    public $version = 1;

    public function text()
    {
        return 'avant';
    }

    public function message()
    {
        return $this->text().' '.$this->version;
    }
}

class Child extends Mother {
    public $version = 2;

    public function text()
    {
        return 'après';
    }
}
```

→ Une surcharge peut servir à **supplanter** le comportement d'une propriété parente

```php
$mother = new Mother();
$child = new Child();

echo $mother->message(); // avant 1
echo $child->message(); // après 2
```

Surcharge – "final"

→ L'instruction « final » permet d'empêcher la surcharge d'une variable ou d'une fonction

```php
class Animal {
    final public $name = 'Default';

    final public function getName()
    {
        return strtolower($this->name);
    }
}

class Dog extends Animal {
    public function getName() // !! ERROR !! \\
    {
        return strtoupper($this->name);
    }
}
```

Surcharge et parenté

```php
class Animal {
    public function hello()
    {
        return 'Hello';
    }
}

class Dog extends Animal {
    public function hello()
    {
        return parent::hello().' dog !';
    }
}

class Cat extends Animal {
    public function hello()
    {
        return 'Hey, '.lcfirst(parent::hello()).' cat !';
    }
}
```

→ Une surcharge peut servir à **compléter** le comportement d'une propriété parente

```php
$dog = new Dog();
echo $dog->hello(); // Hello dog !

$cat = new Cat();
echo $cat->hello(); // Hey, hello cat !
```

Surcharge des méthodes magiques et parenté

```php
class Animal {
    protected $name;

    public function __construct($name)
    {
        $this->name = $name;
    }
}

class Dog extends Animal {
    public $age;

    public function __construct($name, $age)
    {
        $this->age = $age;
        parent::__construct($name)
    }
}
```

→ L'enfant hérite aussi des méthodes magiques du parent.

→ Les méthodes magiques d'un enfant peuvent surcharger celles du parent.

→ Si l'on souhaite conserver le comportement des méthodes magiques du parent, il convient de les appeler dans la méthode surchargé de l'enfant.

→ Attention, les propriétés non surchargé d'un parent peuvent être dépendantes de méthodes, comme le constructeur.

Types de classe

- Il existe d'autres types de classe ("sous-classe") :
  - ***abstract*** *(abstraite): C'est une classe, à usage parental uniquement, apportant des restrictions de structure au développement permettant une bonne utilisation de ces dernières.*
    - Ne peut pas être directement instancié.
    - Définie des signatures de méthodes (avec l'instruction abstract) que tout enfant doit obligatoirement implémenter.
  - ***interface*** *: Définie des méthodes de classe que toute classe qui l'**implémente** doit posséder.*
    - Ne peut pas être directement instancié.
    - Pas de relation parent enfant. On parlera plutôt d'une **contractualisation** entre l'interface et la classe qui l'implémente.
    - Définie des signatures de méthodes que toutes classe qui l'implémente doit obligatoirement créer.
    - Ne possède rien d'autre que des signatures et des constantes de classe (non surchargée).
    - Une classe peut implémenter autant d'interface que l'on souhaite.
  - ***trait*** *: Définie des attributs et méthodes comme le fait une classe mais "importables" par plusieurs autres classe.*
    - Ne peut pas être directement instancié.
    - Une classe peut "importer" autant de trait que l'on souhaite.
    - Ne peut pas posséder de constantes (sauf après PHP8.2).
    - Rarement utilisé mais un trait peut aussi définir des signatures de méthodes abstraite.

Type de classe : Abstraite

```php
abstract class Animal {
    public static $age;
    public $name;

    abstract public function getName();

    public function __construct()
    {
        echo "Animal's constructor";
    }

    public function common()
    {
        echo 'common code';
    }
}

class Dog extends Animal {
    public function getName()
    {
        return $this->name;
    }
}

// ERROR: getName() not defined
class Cat extends Animal {}
```

```php
$dog = new Dog(); // Animal's constructor
echo $dog->common(); // common code
echo $dog->getName(); // Jango
echo $dog::$age; // 0

$animal = new Animal(); // ERROR: Animal is abstract
```

Type de classe : Interface

```php
interface Canine {
    public function getNumberTeeth();
}

interface Coat {
    public function getCoatColor();
}

abstract class Animal {
    public $name = 'Jango';

    abstract public function getName();
    abstract public function setName($name);

    public function common()
    {
        // CODE
    }
}
```

```php
class Dog extends Animal implements Canine, Coat {
    public function getName($name)
    {
        // CODE
    }

    public function setName($name)
    {
        // CODE
    }

    public function getNumberTeeth()
    {
        // CODE
    }

    public function getCoatColor()
    {
        // CODE
    }
}
```

```php
class Cat implements Coat {
    public function getCoatColor()
    {
        // CODE
    }
}
```

Type de classe : Trait

```php
trait Canine {
  public $numberTeeth;

  public function setNumberTeeth($number)
  {
    $this->numberTeeth = $number;
  }
}

trait ColorConverter {
  public function getHexa($color)
  {
    return self::COLORS[$color];
  }
}

class Animal {
  const COLORS = [
    'black' => '000000',
    'white' => 'ffffff',
  ];
}
```

```php
class Dog extends Animal {
  use Canine, ColorConverter;

  public $color;

  public function setColor($color)
  {
    if (array_key_exists($color, self::COLORS)) {
      $this->color = $color;
      return true;
    }

    return false;
  }
}
```

```php
class Cat {
  use Canine;
}
```

Types de classe - Exemple

```php
trait Canine {
  public $numberTeeth;

  public function setNumberTeeth($number)
  {
    $this->numberTeeth = $number;
  }
}

trait ColorConverter {
  public function getHexa($color)
  {
    return self::COLORS[$color];
  }
}

interface Coat {
    const COLORS = [
      'black' => '000000',
      'white' => 'ffffff',
    ];

    public function getCoatColor();
}

class Animal {
   // ...
}
```

```php
class Dog extends Animal implements Coat {
    use Canine, ColorConverter;

    public $color;

    public function setColor($color)
    {
        if (array_key_exists($color, self::COLORS)) {
          $this->color = $color;
          return true;
        }

        return false;
    }

    public function getCoatColor() {
       return $this->getHexa($this->color);
    }
}

class Cat {
    use Canine;
}
```

```php
$dog = new Dog();
$dog->setColor('white');
echo $dog->getCoatColor(); // ffffff

$cat = new Cat();
$cat->setNumberTeeth(24);
echo $cat->numberTeeth; // 24
```

Namespace (Espace de nom)

- Un **namespace** est un peu comme un "**alias virtuel**" d'une classe.
- Cet alias virtuel permet un **rangement des classes encore plus structuré.**
- Pour que ce rangement soit utile réellement utile il faudra suivre quelques conventions. On va imposer les règles suivante (entre la classe et le fichier) :
  - *Rien d'autre qu'une classe (class, abstract, interface ou trait) par fichier*
  - *Le fichier porte le nom exacte de la classe + ".php" : class Animal { } => Animal.php*
  - *Le namespace doit être défini en premier avant tout autre code PHP.*

…/Animal/**Animal.php**

```php
<?php

namespace Animal;

class Animal {
  //
}
```

…/Animal/**Dog.php**

```php
<?php

namespace Animal\Canine;

class Dog {
  //
}
```

…/Animal/**Cat.php**

```php
<?php

namespace Animal\Feline;

class Cat {
  //
}
```

Namespace et Use

- Dans ce cadre "1 fichier 1 classe" le namespace n'est qu'un alias de rangement/classification.
- Il sera nécessaire d'importer les fichiers de classe entre eux (Pour simplifier ces importations, un import automatique (auto-loader) est réalisé dans la plus part des gros Framework de développement).
- Après importation, pour utiliser une classe dans une autre on l'importera avec "use" entre la définition du namespace et la définition de la classe.
- Le chemin d'importation (use) se compose comme suit : NAMESPACE\CLASSNAME

…/Animal/**Animal.php**

```php
<?php
namespace Animal;

class Animal {
  //
}
```

…/Animal/**Dog.php**

```php
<?php
namespace Animal\Canine;

use Animal\Animal;

require_once 'Animal.php';

class Dog extends Animal {
  //
}
```

…/Animal/**Cat.php**

```php
<?php
namespace Animal\Feline;

use Animal\Animal;

require_once 'Animal.php';

class Cat extends Animal {
  //
}
```

Namespace et Use

- On constate très vite qu'import dans chaque classe va vite devenir compliqué et ingérable.
- En même temps, il faut bien comprendre que les fichiers de classe ne doit pas contenir de code procédural, c'est-à-dire sans OOP (Excepté les require/include de fichiers).
- Donc on peut créer un fichier d'importation général à la racine du site. Il suffira d'inclure "autoloader.php" dans tout script PHP où on souhaite instancier les classes.

…/**autoloader.php**

```php
<?php
require_once 'animal/Animal.php';
require_once 'animal/Dog.php';
require_once 'animal/Cat.php';
```

…/Animal/**Animal.php**

```php
<?php
namespace Animal;

class Animal {
  //
}
```

…/**index.php**

```php
<?php

use Animal\Feline\Cat;
use Animal\Canine\Dog;

require_once 'autoloader.php';

$cat1 = new Cat();
$cat2 = new Cat();
$dog1 = new Dog();
$dog2 = new Dog();
```

…/Animal/**Cat.php**

```php
<?php
namespace Animal\Feline;

use Animal\Animal;

class Cat extends Animal {
  //
}
```

…/Animal/**Dog.php**

```php
<?php
namespace Animal\Canine;

use Animal\Animal;

class Dog extends Animal {
  //
}
```

Pratique

- Créer un dossier « Models » s'il n'existe pas.
- Dans le dossier « Models » : Créer une classe mère appelée « Model » dans un fichier « Model.php ».
- Dans le dossier « Models » : Reprenez les différentes classes « model » des l'exercices précédents.
- Ajouter un namespace à toutes les classes du dossiers Models.
- Étendez (extends) les classes enfants Oder, Product et User avec la classe mère « Model ». Vous pouvez soit importer (avec require) directement Model dans les classes enfants soit utiliser un fichier « autoloader.php » pour regrouper l'importation des classes.
- Faites de même pour les contrôleurs. Créez un dossier Controllers, ajoutez un fichier/classe parent Controller et ajoutez le/les controllers des exercices précédents. Etendez les enfants avec le parent et ajoutez les namspaces correspondant.

```
|-- autoloader.php (optionnel)
|-- Models
        |-- Product.php
        |-- Order.php
        |-- Model.php
|-- Controllers
        |-- ProductController.php
        |-- Controller.php
```

Pratique

- Pour les classes mères Model et Controller, n'y a-t-il pas un meilleur type de classe qui leur corresponde ? Si vous le trouvais, faites les modifications nécessaires dans les deux classes.
- Dans le dossier Models, créez un dossier « Contracts » et ajoutez-y une interface « OrderProductInterface » (sans oublier d'y ajouter le bon namespace).
- Ajoutez-y une fonction signature abstraite « public function totalPriceTTC(int $quantity) » où cette méthode est la signature de son homonyme de votre modèle Product.
- Ajoutez ensuite l'interface au modèle Product.
- Puis dans la fonction addProductPriceToOrder du modèle Order, modifiez le type du premier paramètre de Product à OrderProductInterface.
- Voyez-vous les intérêts d'utiliser une interface plus tôt que la classes Product elle-même ?

```
|-- Models
        |-- Contracts
                |-- OrderProductInterface.php
        |-- Product.php
        |-- Order.php
        |-- Model.php
|-- ...
```

Pratique

- Maintenant que vous avez une organisation Controller et Model, le but va être de compléter les méthodes CRUD (Create, Read, Update et Delete) da la ressource Product.
- Il faudra avoir les méthodes suivantes dans le ProductController (ajouter les page php/html nécessaire):
  - CREATE GET: create(): Affiche le formulaire/page de création d'un produit
  - CREATE POST: store(): (action de la page create) Insert un produit dans la base de données
  - READ GET: index(): Affiche la liste des produits avec une pagination
  - READ GET: show(): Affiche un page produit précis via un ID
  - UPDATE GET: edit(): Affiche le formulaire/page d'édition d'un produit précis via un ID
  - UPDATE POST: update(): (action de la page edit) Update un produit précis via un ID
  - DELETE POST: delete(): Delete un produit précis via un ID
- Ajouter une colonne booléenne (mysql: int(1)) « enable » dans votre table de produits. Puis adapter votre code pour faire que dans les méthodes READ, les produits avec « enable = 0 » ne soit plus affichés. On peut toujours les « update » ou « delete ».
- Ajouter, ensuite, dans « update » la possibilité d'activé ou de désactivé un produit.

PHP

OOP Avancé

Fin du module