Le modèle objet de PHP5

Alexandre Niveau

GREYC — Université de Caen

Adapté du cours de Jean-Marc Lecarpentier

Programmation objet en PHP

- La programmation objet a été ajoutée à PHP3, améliorée dans PHP4, et complètement revue dans PHP5
- Syntaxe et fonctionnement lourdement inspirés de Java
- Remarque : les variables d'instance, appelées attributs en Java, sont appelées « propriétés » en PHP

Une classe en PHP5

- Définition et utilisation d'une classe :

```php
<?php
class Rectangle {
  private $longueur;
  private $largeur;

  public function __construct($L, $l) {
    $this->longueur = $L;
    $this->largeur = $l;
  }

  public function getLongueur() {
     return $this->longueur;
  }
  public function getLargeur() {
     return $this->largeur;
  }
}

$c = new Rectangle(5, 2);
echo "Dimensions : {$c->getLongueur()}x{$c->getLargeur()}\n";
echo "Erreur fatale :\n";
echo "Dimensions : {$c->longueur}x{$c->largeur}";
```

Pièges des objets

- Il faut obligatoirement utiliser `$this` dans une classe pour faire référence aux propriétés et méthodes

```php
<?php
class Toto {
    public $x = 3;
    function afficherX() {
        echo $x;  // (au lieu de $this->x)
    }
}
$toto = new Toto();
$toto->afficherX();  // lève une Notice: undefined variable $x
```

- Lors de la déclaration de la visibilité des propriétés, on leur met des `$`, mais pas ailleurs. Cependant, mettre un `$` n'est pas une erreur de syntaxe, ça fait juste autre chose que ce qu'on voudrait...

```php
<?php
class Toto {
    public $x = 3;
}
$toto = new Toto();
echo $toto->x;  // affiche 3
echo $toto->$x;  // lève une Notice: undefined variable $x

$variable = "x";
echo $toto->$variable;  // affiche 3 : $toto-
>$variable est interprété comme $toto->x !
```

- Un foreach sur un objet parcourt ses propriétés :

```php
<?php
class Toto {
    public $x = 3;
    public $y = 'coucou';
}
$toto = new Toto();
foreach ($toto as $k => $v) {
    echo "clef $k, valeur $v\n";
}
// résultat :
//     clef x, valeur 3
//     clef y, valeur coucou
```

Constructeur, destructeur, conversion en chaîne

- Fonction `__construct()`
- Fonction `__destruct()` : appelée lorsque l'instance est détruite (utilisation de `unset`, ou fin de l'exécution du script)
- Fonction `__toString()` : appelée lorsque l'objet doit être converti en chaîne de caractères
- *Attention au double underscore du constructeur*, crée des bugs difficiles à trouver !

```php
<?php
```

```php
class Point {
  public $x;
  public $y;
  public function ___construct($x, $y) {
    $this->x = $x;
    $this->y = $y;
  }

  public function __toString() {
    return "({$this->x}, {$this->y})";
  }
}

$p = new Point(3, 2);
echo $p;
```

```
(, )
```

- Que s'est-il passé ?

Membres statiques

- Utilisation du mot clé `static`
- Une propriété statique a la même valeur pour toutes les instances de la classe
- Une méthode statique peut être appelée sans avoir instancié la classe
- Différences avec Java :
  - pas la même syntaxe pour l'accès aux propriétés et méthodes statiques : *double deux-points* à la place de la flèche
  - on n'appelle pas les propriétés et méthodes statiques avec `$this`, mais avec le mot-clef `self`, qui représente « la classe de l'instance courante »

```php
<?php

// Exemple de classe avec des membres statiques

class Toto {
  private static $compteur = 0;
  private $dummy;

  public function __construct($contenu) {
    $this->dummy = $contenu;
    self::$compteur++;
  }

  public function printInstanceStats() {
    // $this->compteur n'est pas défini => donne une chaine vide
    return $this->dummy . " et " . $this->compteur . "\n";
  }

  public static function printClassStats() {
    //$this->dummy = "toto";  // Fatal error
```

```
    return "Il y a actuellement " . self::$compteur . " instances de la classe
  }

  public function __destruct() {
    echo "Destruction d'une instance\n";
    self::$compteur--;
  }
}

echo Toto::printClassStats();

$A = new Toto("instance A");
echo "\$this->compteur n'est pas défini => donne une chaine vide et une Notice
echo "A : " . $A->printInstanceStats();
echo Toto::printClassStats();

$B = new Toto("instance B");
echo "\$this->compteur n'est pas défini => donne une chaine vide et une Notice
echo "B : " . $B->printInstanceStats();
echo Toto::printClassStats();

echo "unset(\$A)\n";
unset($A);
echo Toto::printClassStats();

unset($B);
```

```
Il y a actuellement 0 instances de la classe Toto
$this->compteur n'est pas défini => donne une chaine vide et une Notice
A : instance A et
Il y a actuellement 1 instances de la classe Toto
$this->compteur n'est pas défini => donne une chaine vide et une Notice
B : instance B et
Il y a actuellement 2 instances de la classe Toto
unset($A)
Destruction d'une instance
Il y a actuellement 1 instances de la classe Toto
Destruction d'une instance
```

Constante d'un objet

- Définition avec `const`
- Comme pour les propriétés statiques, accès avec `::`

```
<?php

class MyClass {
  const ma_constante = 'toto';

  public function showConstant() {
    echo  "Voici ma constante: " .self::ma_constante . "\n";
  }

  public function chgeConstant() {
```

```
        // Fatal error
        // self::ma_constante = "test d'erreur";
    }
}

$obj = new MyClass();

$obj->showConstant();

echo "MyClass::ma_constante : " . MyClass::ma_constante . "\n";  // Fonctionne
echo "\$obj->ma_constante : " . $obj->ma_constante . "\n";  // Ne fonctionne p
echo "\$obj::ma_constante : " . $obj::ma_constante . "\n"; // Marche avec PHP
```

Dérivation

- Définition d'une classe dérivée avec `extends`

```
<?php
class Rectangle {
  protected $longueur;
  protected $largeur;

  public function __construct($longueur, $largeur) {
    $this->longueur = $longueur;
    $this->largeur = $largeur;
  }

}

class Carre extends Rectangle {

  public function __construct($largeur) {
    parent::__construct($largeur, $largeur);
  }
```

- Contrairement à Java, constructeurs et destructeurs sont hérités : si non définis dans une sous-classe, ce sont ceux du parent qui sont utilisés
- En revanche, s'ils sont redéfinis, ils n'appellent pas automatiquement ceux du parent : il faut le faire explicitement avec `parent::__construct()`

Late static binding

- Il existe une alternative à `self` : `static`
- Permet de résoudre « tardivement » une référence à un champ statique

  ➤ `static::toto` n'est pas forcément le `toto` de la classe courante, mais celui de la classe qui a effectivement été instanciée à l'exécution
- voir la doc [https://www.php.net/manual/en/language.oop5.late-static-bindings.php]

- permet de faire de l'héritage sur les méthodes/propriétés statiques… Utile par exemple pour faire des *factory methods* statiques [https://www.php.net/manual/en/language.oop5.late-static-bindings.php#114005]

Visibilité des membres

- 3 niveaux :
  - public : accessible de partout
  - protected : accès limité à la classe et ses classes dérivées
  - private : accès seulement dans la classe

```php
<?php
class MyClass
{
    public $public = 'Public';
    protected $protected = 'Protected';
    private $private = 'Private';

    function printHello()
    {
        echo $this->public;
        echo $this->protected;
        echo $this->private;
    }
}

$obj = new MyClass();
echo $obj->public; // Works
//echo $obj->protected; // Fatal Error
//echo $obj->private; // Fatal Error
$obj->printHello(); // Shows Public, Protected and Private


// classe dérivée
class MyClass2 extends MyClass
{
    // We can redeclare the public and protected method, but not private
    protected $protected = 'Protected2';

    function printHello()
    {
        echo $this->public;
        echo $this->protected;
        echo $this->private;
    }
}

$obj2 = new MyClass2();
echo $obj2->public; // Works
echo $obj2->private; // Undefined
//echo $obj2->protected; // Fatal Error
$obj2->printHello(); // Shows Public, Protected2, not Private
```

```
PublicPublicProtectedPrivatePublicPublicProtected2
```

Encapsulation des données

- Données membres non visibles de l'extérieur
- Utilisation d'accesseurs et de mutateurs

```php
<?php

class Rectangle {
  protected $longueur;
  protected $largeur;

  public function __construct($longueur, $largeur) {
    $this->longueur = $longueur;
    $this->largeur = $largeur;
  }

  // Accesseurs

  // Getters
  public function getLongueur() { return $this->longueur; }
  public function getLargeur() { return $this->largeur; }

  // Setters
  public function setLongueur($longueur) { $this->longueur = $longueur; }
  public function setLargeur($largeur) { $this->largeur = $largeur; }

}

class Carre extends Rectangle {

  public function __construct($largeur) {
    parent::__construct($largeur, $largeur);
  }

  public function setLargeur($largeur) {
    parent::setLongueur($largeur);
    parent::setLargeur($largeur);
  }
}


echo "Définition d'un rectangle 5x2 :\n";
$R = new Rectangle(5, 2);
// echo $R->longueur; produit Fatal error
// Utiliser les accesseurs :
echo "Longueur : " . $R->getLongueur() . "\n";
echo "Largeur : " . $R->getLargeur() . "\n";

echo "Définition d'un carré de largeur 4\n";
$C = new Carre(4);
echo "Largeur :  " . $C->getLargeur() . "\n";

echo "Changement de la largeur : \n";
$C->setLargeur(8);
echo "Largeur :  " . $C->getLargeur() . "\n";
```

```php
echo "Longueur :  " . $C->getLongueur() . "\n";
```

```
Définition d'un rectangle 5x2 :
Longueur : 5
Largeur : 2
Définition d'un carré de largeur 4
Largeur :  4
Changement de la largeur :
Largeur :  8
Longueur :  8
```

- Au passage, noter le mot-clef `parent` pour appeler une méthode (ou le constructeur) de la superclasse

Exceptions : principe

- Quand on rencontre un problème dans son programme, au lieu de simplement l'arrêter, on envoie une exception
- L'envoi d'une exception arrête l'exécution du code en cours
- L'intérêt est qu'il est possible de *récupérer* des exceptions qui ont été envoyées par une partie de son programme, afin de traiter la source du problème (si c'est possible)
- Une exception non récupérée remonte à la fonction appelante, qui elle-même peut avoir mis en place une récupération, etc.
- Si l'exception remonte « jusqu'en haut », le programme s'arrête avec une *Fatal error: Uncaught exception*

Renvoyer une exception

- `throw new Exception("Houston, we have a problem");`
- Classe Exception prédefinie dans PHP5
- Méthodes ("final") :
  - `getMessage()` : message d'erreur de l'exception
  - `getCode()` : code d'erreur de l'exception
  - `getFile()` : fichier dans lequel l'exception a été renvoyée
  - `getLine()` : n° de ligne où l'exception a été renvoyée
  - `getTrace()` : tableau contenant la trace
  - `getTraceAsString()` : chaîne de caractères avec la trace

Essayer du code

- Pour pouvoir récupérer des exceptions, il faut placer le code PHP à exécuter dans un bloc `try`

- Tout envoi d'exception dans ce bloc interrompt l'exécution…
- … mais le programme passe alors à la gestion de l'exception, qui suit

Attraper l'exception

- Un bloc `try` doit être suivi d'un bloc `catch`
- Ce bloc "attrape" les exceptions éventuellement qui ont été renvoyées dans le bloc `try`
- ```
  try {
  // code à exécuter
  }
  catch (Exception $e) {
     echo "Erreur, exception renvoyée : " . $e->getMessage();
  }
  ```

Personnaliser ses exceptions

- Dériver la classe `Exception`
- Permet de créer différents types d'exceptions en utilisant plusieurs blocs `catch`
- Une exception non gérée dans un bloc `catch` est passée au bloc suivant

Affichage debug d'un objet

```php
<?php

class Titi {
    private $x = false;
    protected $y = "";
    public $z = null;
    function methode() { }
}

echo "---- var_export ----\n";
var_export(new Titi());
echo "\n";

echo "---- print_r ----\n";
print_r(new Titi());
echo "\n";

echo "---- var_dump ----\n";
var_dump(new Titi());
echo "\n";
```

```
---- var_export ----
\Titi::__set_state(array(
   'x' => false,
   'y' => '',
   'z' => NULL,
```
```

```
))
---- print_r ----
Titi Object
(
    [x:Titi:private] =>
    [y:protected] =>
    [z] =>
)

---- var_dump ----
/users/ensweb/www-prod/TW4b/pres/oophp/demo/exPrintDebug.php:19:
class Titi#4 (3) {
  private $x =>
  bool(false)
  protected $y =>
  string(0) ""
  public $z =>
  NULL
}
```

Mot clé final

- `final` pour une méthode : la méthode ne peut pas être redéfinie dans une classe dérivée
- `final` pour une classe : la classe ne peut pas être dérivée

```php
<?php
class finalEx {
  private $donnee = "exemple";

  final function maj() {
    return  strtoupper($this->donnee);
  }
}

final class finalExtend extends finalEx {

  /* Fatal error : impossible de redéfinir maj()
  public function maj() {
    return  $this->donnee;
  } */
}

/* Fatal error : impossible de dériver finalExtend
class erreur extends finalExtend {
  // impossible
}
*/

$class = new finalEx();
echo $class->maj();
echo "\n";
$classExtend = new finalExtend();
echo $classExtend->maj();
```

```
?>
```

```
EXEMPLE
EXEMPLE
```

Classe et méthode abstraite

- Mot clé `abstract`
- Une classe abstraite ne peut pas être instanciée
- Une méthode abstraite définit sa signature, pas son implémentation
- Une méthode abstraite doit être dérivée avec une visibilité supérieure

```
<?php

abstract class AbstractClass
{
   // Force la classe étendue à définir cette méthode
   abstract protected function getValue();

   // méthode commune
   public function printOut() {
     print $this->getValue();
   }
}

class ConcreteClass1 extends AbstractClass
{
   protected function getValue() {
     return "ConcreteClass1";
   }
}

class ConcreteClass2 extends AbstractClass
{
   protected function getValue() {
     return "ConcreteClass2";
   }
}

$class1 = new ConcreteClass1;
$class1->printOut();

$class2 = new ConcreteClass2;
$class2->printOut();


?>
```

```
ConcreteClass1ConcreteClass2
```

Interfaces

- Permettent de créer du code spécifiant les signatures des méthodes
- Sans avoir à définir leur implémentation
- Définir avec le mot clé `interface`
- Une classe implémente une interface :
  `class maClasse implements monInterface`
- Existence d'interfaces "internes" prédéfinies dans PHP5

Déclaration du type des paramètres

- « Type hinting » de PHP 5 :
  - On peut déclarer un type pour les paramètres des fonctions
  - Limités à des noms de classes (et `array` et `callable`)

    ➤ déjà très utile notamment avec des interfaces
  - `function MaFonction(MaClasse $param)` : $param doit être une instance de la classe MaClasse
  - Vérification faite à l'exécution. Erreur fatale (*catchable*…) émise si le paramètre n'est pas une instance de la classe indiquée
- En PHP 7, on peut utiliser aussi des types primitifs, comme `int` ou `string` : voir la liste [https://www.php.net/manual/en/functions.arguments.php#functions.arguments.type-declaration]
- Attention : par défaut, une `TypeError` est envoyée si le type n'est pas respecté, **sauf** si la conversion est possible !

  ➤ il faut ajouter `declare(strict_types=1);` pour bénéficier du « typage strict », qui renverra bien une `TypeError` dans tous les cas où le paramètre donné ne respecte pas la déclaration. Sauf pour les `int`, qui pourront toujours être convertis en `float`.

Déclaration du type de retour

- On peut aussi depuis PHP 7 déclarer un type de retour pour les fonctions et méthodes
- Syntaxe :

```
function toto(): float {
    return 3.14;
}
```

- On y met les mêmes types que pour les paramètres, avec `void` en plus
- Même piège du « mode strict » non activé par défaut (la valeur de retour, par défaut, est simplement convertie si c'est possible)

Types nullifiables

- Très souvent en PHP les paramètres et les valeurs de retour sont mises à `null` pour certains cas particuliers (par exemple des erreurs)

  ➤ ennuyeux si on veut typer la fonction

- Depuis PHP 7.1, on peut utiliser des *nullable types*, en ajoutant un point d'interrogation au début du type : par ex `function titi(?int $x)`
- Signifie qu'on attend un int, ou null

Héritage vs composition

- Un principe important en POO est de favoriser la *composition* plutôt que l'*héritage* pour mutualiser du code
- En effet l'héritage a des conséquences pénibles, notamment le fait qu'à partir du moment où on a sous-classé une classe, il devient difficile de la faire évoluer (tout changement a des conséquences sur les classes filles)

  ➤ cela contrevient à la modularisation du code — les classes sont trop dépendantes les unes des autres, alors que le but de la POO est plutôt d'essayer de créer des « boîtes noires » dont le comportement interne n'a pas d'influence à l'extérieur

- En utilisant la composition, on garde l'indépendance entre les classes

  ➤ le prix à payer est une syntaxe un peu plus lourde dans la classe, et l'obligation parfois d'écrire des méthodes de « délégation »

Traits

- Les *traits* de PHP [https://www.php.net/manual/en/language.oop5.traits.php] sont (principalement) un moyen de faire de la composition, sans les inconvénients

```
trait MonTrait {
    function uneMethodeUtile() {
}

class Toto {
    use MonTrait;
}
```

- Dans l'exemple ci-dessus, la classe `Toto` contient la méthode `uneMethodeUtile`, comme si elle avait hérité de `MonTrait`, sauf qu'il n'y a pas eu d'héritage :
  - Toto peut très bien étendre une autre classe
  - Toto peut utiliser d'autres traits, pas de problème d'héritage multiple (en cas de conflit de nommage, il faut *obligatoirement* définir explicitement la méthode concernée dans `Toto`
  - On peut utiliser une même méthode dans plusieurs classes, sans qu'elles soient liées par le typage
- Les inconvénients de la composition disparaissent, puisque tout se passe comme si la méthode était définie dans la classe :
  - syntaxe simple

- pas besoin d'écrire des méthodes de délégation

Méthodes magiques

- Méthodes `__sleep` et `__wakeup` : code à exécuter lors de la sérialisation et désérialisation d'un objet
- Utile pour les ressources (connection BDD, ressource fichier)
- Méthodes `__get` `__set` `__call` pour gérer la surcharge objet (voir le manuel [http://fr.php.net/manual/fr/language.oop5.overloading.php])

  ➤ à utiliser pour gérer les erreurs, pas « pour de vrai » !

[http://creativecommons.org/licenses/by-nc-sa/4.0/]