




Le langage PHP (orienté objet)

Développement web
HENALLUX — IG3


Au programme...

- 
- **Le langage PHP : langage de base**
 - Caractéristiques générales du langage
 - Données, instructions, éléments du langage, tableaux...
 - **Le langage PHP : aspects orientés objets** 
 - **Échanges de données entre le client et le serveur**

PHP orienté objet

- 
- **Classes et objets en PHP**
 - Comment définir une classe en PHP
 - Comment manipuler des objets en PHP
 - Visibilité, propriétés de classe, constructeur...
 - **Héritage en PHP**
 - Héritage entre classes
 - Classes abstraites et interfaces
 - **Particularités du PHP orienté objet**
 - Méthodes magiques
 - Traits
 - Namespaces
 - Late static binding

Classes et objets en PHP

- 
- **Création de classe et d'objets**
 - Syntaxe de base
 - Attributs et méthodes
 - **Manipuler des objets en PHP**
 - **Plus en détail...**
 - Visibilité des propriétés (public/private)
 - Constructeur et destructeur
 - Propriétés statiques
 - La fonction `__autoload`

Ensuite : *l'héritage en PHP*

Classes et objets

- L'orienté objet en PHP :
 - introduit à partir de la version 5
 - basé sur les classes (comme Java) <-> Javascript (oo prototypal)

- Définir une classe

```
class Etudiant {  
    public $nom, $cote;  
    function affiche () {  
        echo $this->nom, ' a eu ', $this->cote, ' /20.';  
    }  
}
```

- Créer un objet

```
$e = new Etudiant ();
```

Classes et objets : attributs

- Propriétés : déclaration optionnelle

```
class Etudiant {  
    public $nom, $cote;  
}
```

Ces déclarations sont optionnelles : on peut ajouter des propriétés à la volée !

- Accès aux propriétés (-> au lieu de .)

```
$e = new Etudiant ();
```

```
$e->nom = 'Lisa';
```

Pas de \$ devant le nom des propriétés !

```
$e->cote = 18; // $e->Cote créerait une autre propriété !
```

```
echo $e->nom;
```

- Remarque : `$nom = 'cote';`
`$e->$nom` donne 18 !

Classes et objets : attributs

- Initialisation par défaut (la valeur doit être constante)

```
class Etudiant {  
    public $nom, $cote = 10;  
}
```

- Pour tester le contenu d'un objet : print_r

```
$e = new Etudiant ();  
$e->nom = 'Lisa';  
$e->cote = 18;  
print_r ($e);
```

```
Etudiant Object  
(  
    [nom] => Lisa  
    [cote] => 18  
)
```

Classes et objets : méthodes

- Définition d'une méthode

```
class Etudiant {  
    public $nom, $cote;  
    function affiche () {  
        echo $this->nom, ' a eu ', $this->cote, ' /20.';  
    }  
}
```

Accès aux champs
via `$this->nom`

Pour les méthodes, éviter les
noms qui commencent par `__`

- Utilisation

```
$e = new Etudiant ();  
$e->nom = 'Lisa';  
$e->cote = 18;  
$e->affiche(); // $e->Affiche() produit le même résultat !
```


Classes et objets : méthodes

- Accès au sein de la classe : même dans la classe elle-même, les noms des propriétés et des méthodes doivent toujours être qualifiés

```
class Etudiant {  
    public $nom, $cote;  
    function affiche () {  
        echo $this->nom, ' a eu ', $this->pourcent(), '%';  
    }  
    function pourcent () {  
        return 100 * $this->cote / 20;  
    }  
}
```

nom ne suffit pas !

pourcent() ne suffit pas !

- En PHP, pas de surcharge (= même nom, nombres d'arguments différents).

Manipuler des objets en PHP

- Dans la plupart des langages, tester l'égalité de deux objets revient à tester l'égalité des références.
- En PHP, `$o1 == $o2` est vrai si
 - `$o1` et `$o2` sont de même classe, et
 - les propriétés de `$o1` et de `$o2` ont les mêmes valeurs.

Note : `$o1 = $o2` donne bien lieu à un partage de référence.
- Par contre, `$o1 === $o2` teste bien l'égalité des références.

```
class A { public $mot; }  
$a1 = new A (); $a1->mot = "Hello";  
$a2 = new A (); $a2->mot = "Hello";  
echo 'Test == : ', $a1==$a2, "\n";  
echo "Test === : \n", $a1=== $a2;
```

```
Test == : 1  
Test === :
```

Manipuler des objets en PHP

- Tester le « type » d'un objet : `$e instanceof Etudiant`
- On peut parcourir le contenu d'un objet comme s'il s'agissait d'un tableau associatif.
- On peut également ajouter de nouvelles propriétés à la volée.

```
$e = new Etudiant();  
$e->nom = "Lisa";  
$e->cote = 18;  
$e->famille = "Simpsons";
```

```
foreach ($e as $prop => $val)  
    echo "$prop contient : $val\n";
```

```
nom contient : Lisa  
cote contient : 18  
famille contient : Simpsons
```

Visibilité des propriétés

- Pour les propriétés et les méthodes :
 - `public` (par défaut) : visible partout
Note : on peut également utiliser `var` pour les propriétés
 - `protected` : visible dans la classe et dans ses sous-classes
 - `private` : visible uniquement dans la classe

```
class Etudiant {  
    var $nom; // synonyme de public $nom  
    private $cote;  
    private function pourcentage () {return $this->cote*5;}  
}
```

Constructeur et destructeur

- Constructeur : ancienne syntaxe (similaire à Java)

```
class Etudiant {  
    public $nom, $cote;  
    function Etudiant ($nom, $cote) {  
        $this->nom = $nom; $this->cote = $cote;  
    }  
}  
$e = new Etudiant('Lisa', 18);
```

- Constructeur : nouvelle syntaxe (fonctions "magiques")

```
class Etudiant {  
    public $nom, $cote;  
    function __construct ($nom, $cote) {  
        $this->nom = $nom; $this->cote = $cote;  
    }  
}  
$e = new Etudiant('Lisa', 18);
```

Constructeur et destructeur

- Destructeur (appelé juste avant la destruction d'un objet)

```
class Etudiant {  
    public $nom, $cote;  
    function __destruct () { ... }  
}
```

- Détruire un objet : supprimer toutes les références vers lui.

```
class A { function __destruct () { echo 'Objet détruit.'; } }  
$a = new A();  
unset($a); // déclenche le destructeur  
$b = new A();  
$c = $b;  
unset($b); // ne déclenche pas le destructeur  
           // car l'objet est encore accessible
```

Propriétés de classe

- Membres attachés à la classe plutôt qu'à chaque instance

```
class Etudiant {  
    public $nom, $cote;  
    static $maxCote = 0;  
    function afficheCoteEtMax () {  
        echo $this->cote, ' [max=', self::$maxCote, ']';  
    }  
}
```

Référence à un
membre statique
via self::

- Accès à une propriété statique : uniquement via la classe !
 - `Etudiant::$maxCote` (depuis l'extérieur ou l'intérieur)
 - `self::$maxCote` (uniquement depuis l'intérieur de la classe)
 - Mais `$e->maxCote`, `$this->maxCote` ne fonctionnent pas !
 - Méthodes statiques : `$obj->appel()` génère un warning

Propriétés de classe

- Membres attachés à la classe plutôt qu'à chaque instance

```
class Utils {  
    static function afficheSomme ($x, $y, $z) {  
        echo $x + $y + $z;  
    }  
}  
Utils::afficheSomme(3,4,5);
```

- Accès à la méthode statique : uniquement via la classe !
 - `Utils::afficheSomme` (depuis l'extérieur ou l'intérieur)
 - `self::afficheSomme` (uniquement depuis l'intérieur de la classe)

Propriétés de classe

- Définitions de constantes propres à une classe ("final static" en Java)

```
class Etudiant {  
    const MAX = 20;  
    public $nom, $cote;  
}
```

- Accès aux constantes
 - `Etudiant::MAX` depuis l'intérieur ou l'extérieur de la classe
Exemple : `echo Etudiant::MAX;`
 - `self::MAX` depuis l'intérieur de la classe (pas `$this->` !)
Exemple :

```
function pourcentage () {  
    return 100*$this->cote/self::MAX;  
}
```


La fonction `__autoload`

- PHP permet de définir une fonction qui est automatiquement appelée lorsqu'il rencontre une classe encore inconnue.
- On peut l'utiliser pour lui faire charger automatiquement la définition de la classe en question si les fichiers sont bien ordonnés :

```
function __autoload($classe) {  
    require_once "classes/$classe.class.php";  
}
```

si toutes les classes sont définies dans des fichiers nommés `nomClasse.class.php` rangés dans le répertoire `classes`.

Héritage en PHP

- 
- **Héritage entre classes en PHP**
 - ... extends ...
 - **Classes abstraites et interfaces**
 - **Héritage et constructeurs**
 - Pas d'appel automatique comme en Java !

Ensuite : *Particularités du PHP orienté objet*

Héritage entre classe

- [Comme Java] Pas d'héritage multiple (mais voir : interface, traits)
- Via le mot `extends` (comme en Java)

```
class Animal {  
    public $nom;  
    function affiche () {echo 'Je suis ', $this->nom;}  
}  
class Chien extends Animal {  
    function affiche () {  
        parent::affiche();  
        echo ' et je suis un chien !';  
    }  
}  
$c = new Chien ();  
$c->nom = "Fido";  
$c->affiche();
```

Référence à la classe mère via
`parent::` (statique ou non)

Héritage entre classe

- Le mot-clef « final » fonctionne comme en Java : il permet de...
 - bloquer la redéfinition d'une méthode

```
class Animal {  
    final function estVivant () {  
        return true;  
    }  
}
```

ne pourra pas être redéfini
dans les classes filles

- bloquer l'extension d'une classe

```
final class Caniche extends Chien {  
    ...  
}
```

ne pourra pas avoir de
classe fille

Héritage : classes abstraites

- Syntaxe proche du Java (abstract, extends)

```
abstract class Mordeur {  
    public $nom;  
    function attaque($cible) { echo $this->nom, ' mord ', $cible; }  
    abstract function description();  
}  
class Loup extends Mordeur {  
    public $couleur;  
    function __construct($couleur) {  
        $this->couleur = $couleur;  
        $this->nom = 'Un loup '.$couleur;  
    }  
    function description() {  
        echo $this->nom, ' surgit devant vous!';  
    }  
}
```

La classe fille doit implémenter toutes les méthodes abstraites (ou être abstraite elle-même).

```
$l = new Loup ("blanc");  
$l->description();  
$l->attaque("Lisa");
```

Héritage : interfaces

- Syntaxe proche du Java (interface, implements)

```
interface IMonstre {  
    function description();  
    function attaque($cible);  
}
```

```
class Loup implements IMonstre {  
    public $couleur;  
    function description() {  
        echo 'Un loup ', $this->couleur, ' surgit devant vous!';  
    }  
    function attaque($cible) {  
        echo 'Un loup ', $this->couleur, ' griffe ', $cible;  
    }  
}
```

Toutes les méthodes doivent
être implémentées !

```
$l = new Loup ();  
$l->couleur = "blanc";  
$l->description();  
$l->attaque("Lisa");
```

Héritage et constructeurs

- En **Java**, le constructeur est une méthode bien à part.
 - Il n'est pas hérité.
 - Quand on construit un objet de la classe-fille, on fait toujours appel à un constructeur de la classe-mère.
 - Si la classe-fille possède un constructeur...
 - Si celui-ci possède un appel explicite au constructeur de la classe-mère, il doit être tout au début du code ;
 - S'il n'y a aucun appel explicite, on ajoute l'instruction `super();` au début.
 - Si la classe-fille ne possède pas de constructeur explicite...
 - On crée un constructeur implicite qui se contente d'appeler `super();`
- **Ce n'est pas la même chose en PHP !**

Héritage et constructeurs


- L'héritage traite la méthode magique `__construct` comme les autres méthodes (par opposition à Java).

```
class Vehicule { public $nbRoues; }
```

```
class Voiture extends Vehicule {  
    public $marque;  
    function __construct() { $this->nbRoues = 4; }  
}
```

```
class VoitureNeuve extends Voiture {  
    public $dateConstruction;  
}
```

```
$vn = new VoitureNeuve();  
print_r($vn);
```



Le constructeur est hérité !

```
VoitureNeuve Object  
(  
    [dateConstruction] =>  
    [marque] =>  
    [nbRoues] => 4  
)
```

Héritage et constructeurs


- L'héritage traite la méthode magique `__construct` comme les autres méthodes (par opposition à Java).

```
class Vehicule { public $nbRoues; }
```

```
class Voiture extends Vehicule {  
    public $marque;  
    function __construct($marque) { $this->nbRoues = 4;  
                                     $this->marque = $marque; }}
```

```
class VoitureNeuve extends Voiture {  
    public $dateConstruction;  
}
```

```
$vn = new VoitureNeuve();  
print_r($vn);
```



Le constructeur est hérité !

Warning: Missing argument 1 for Voiture::__construct()

Héritage et constructeurs

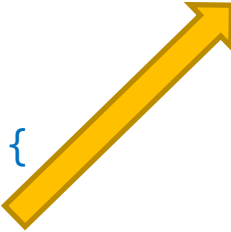
- Il n'y a pas d'appel implicite au constructeur de la classe-mère !

```
class Vehicule { public $nbRoues; }
```

```
class Voiture extends Vehicule {  
    public $marque;  
    function __construct() { $this->nbRoues = 4; }  
}
```

```
class VoitureNeuve extends Voiture {  
    public $dateCons;  
    function __construct($date) { $this->dateCons = $date; }  
}
```

```
$vn = new VoitureNeuve(2015);  
print_r($vn);
```



Pas d'appel implicite au constructeur-mère !

```
VoitureNeuve Object  
(  
    [dateCons] => 2015  
    [marque] =>  
    [nbRoues] =>  
)
```

Héritage et constructeurs

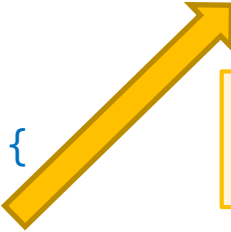
- Il n'y a pas d'appel implicite au constructeur de la classe-mère !

```
class Vehicule { public $nbRoues; }
```

```
class Voiture extends Vehicule {  
    public $marque;  
    function __construct() { $this->nbRoues = 4; }  
}
```

```
class VoitureNeuve extends Voiture {  
    public $dateCons;  
    function __construct($date) { parent::__construct();  
                                   $this->dateCons = $date; }  
}
```

```
$vn = new VoitureNeuve(2015);  
print_r($vn);
```



Si on en veut un, il faut l'écrire explicitement !


```
VoitureNeuve Object  
(  
    [dateCons] => 2015  
    [marque] =>  
    [nbRoues] => 4  
)
```

Syntaxe : opérateurs `->` et `::`

- Opérateur *objet*->
 - référence à une propriété : `$a->nom`
 - référence à une méthode (statique ou pas) : `$a->agit()`
 - sur l'objet courant : `$this->...`
- Opérateur *classe*::
 - référence à une constante : `Etudiant::COTEMAX`
 - référence à une propriété statique : `Etudiant::$nombre`
 - référence à une méthode statique : `Etudiant::incremNombre()`
 - sur la classe courante : `self::...`
 - référence à la classe-mère : `parent::meth()`

Note : si `$c1 = 'Etudiant'`, on peut aussi utiliser `$c1::COTEMAX...`

Particularités du PHP oo

- 
- **Les « méthodes magiques »**
 - `__construct`, `__destruct` et les autres
 - **Les traits**
 - Pour éviter le copier/coller entre les classes
 - **Les namespaces**
 - Les « packages » Java à la sauce PHP
 - **Le late static binding**
 - Le polymorphisme étendu aux méthodes de classe

Les méthodes magiques

- Les méthodes magiques sont des méthodes...
 - dont le nom commence par « `__` »
 - dont le nom est fixé à l'avance
 - qui sont appelées automatiquement dans certaines situations.
- Deux exemples :
 - `__construct()` : constructeur de la classe
 - appelé automatiquement lorsque « `new` » est utilisé
 - `__destruct()` : destructeur de la classe
 - appelé automatiquement lorsqu'un objet est détruit (lorsque sa place réservée en mémoire est libérée)
 - ... mais il y en a d'autres !

Les méthodes magiques

- Méthodes magiques relatives à la conversion automatique :
 - `__toString()` : affichage de l'objet
 - appelé automatiquement lorsqu'on doit convertir l'objet en une chaîne de caractères (ex : concaténation ou echo)
 - `__invoke($arg)` : l'objet comme une fonction
 - appelé automatiquement lorsqu'on doit convertir l'objet en une fonction (ex : on écrit `$obj(1)`)

Les méthodes magiques

- Méthodes magiques relatives aux propriétés inexistantes ou inaccessibles :
 - `__set($nom,$val)` : mutateur
 - appelé automatiquement lorsqu'on tente d'affecter une valeur à une propriété inexistante/inaccessible `$obj->prop`
 - `__get($nom)` : sélecteur
 - appelé automatiquement lorsqu'on tente de lire la valeur d'une propriété inexistante/inaccessible `$obj->prop`
 - `__isset($nom)` : vérificateur d'existence d'une propriété
 - appelé automatiquement lorsqu'on doit évaluer une expression `isset($obj->prop)` ou `empty($obj->prop)` où « prop » est une propriété inexistante ou inaccessible
 - `__unset($nom)` : destructeur de propriété
 - appelé automatiquement lorsqu'on demande la destruction d'une propriété inexistante/inaccessible via `unset($obj->prop)`

Les méthodes magiques

- Exemple d'utilisation :

```
class ToutPrivate {
    private $valeur;
    function __get($name) {
        if ($name == "valeur") {
            echo "Demande de valeur\n";
            return $this->valeur;
        }
    }
    function __set($name,$val) {
        if ($name == "valeur") {
            if ($val > 0) {
                $this->valeur = $val;
            }
        }
    }
}
```

```
$tp = new ToutPrivate();
$tp->valeur = 3;
echo "Valeur = $tp->valeur !\n";
$tp->valeur = -7;
echo "Valeur = $tp->valeur !\n";
```

```
Demande de valeur
Valeur = 3 !
Demande de valeur
Valeur = 3 !
```

Sans __get/__set :

Fatal error, cannot access private property

Traits

- Les traits sont aux classes ce que les fonctions/modules sont aux programmes.
= moyen de mettre des éléments communs en évidence.

- Déclaration

```
trait aUnNom {  
    public $nom;  
    function afficheNom () { echo $this->nom; }  
}
```

- Utilisation

```
class Etudiant {  
    use aUnNom;  
}
```



Traits

- On peut combiner plusieurs traits dans une même classe.

```
class Etudiant {  
    use aUnNom, aUneAdresse;  
    ...  
}
```

- On peut définir des traits à partir d'autres.

```
trait aUnNomEtUnAge {  
    use aUnNom;  
    public $age;  
    ...  
}
```

Traits

- Priorité (ordre d'utilisation) des membres

```
class Mere {  
    function f () { echo 'fMere'; }    3  
}  
trait monTrait {  
    function f () { echo 'fTrait'; }    2  
}  
class Fille extends Mere {  
    use monTrait;  
    function f() { echo 'fFille'; }    1  
}  
$o = new Fille ();  
$o->f();
```

Traits

- Renommage des membres d'un trait

```
trait multiplication {  
    public $nb, $val;  
    function produit () { return $this->nb * $this->val; }  
}
```

```
class Achat {  
    use multiplication {  
        multiplication::produit as prixTotal;  
    }  
}
```

```
$a = new Achat(); $a->nb = 3 ; $a->val = 10.5;  
echo $a->prixTotal();
```


Traits

- Une classe ne peut pas utiliser deux traits qui déclarent des méthodes de même nom.

```
trait t1 { function f() { echo "f de t1"; } }  
trait t2 { function f() { echo "f de t2"; } }  
class A { use t1, t2; } // Erreur
```

- Déclaration de préférence

```
class A {  
    use t1, t2 {  
        t2::f insteadof t1;  
        t1::f as g;  
    }  
}  
$a = new A();  
$a->f();  
$a->g();
```



Permet de tout de même
avoir accès à t1::f

Namespaces

- Pour éviter les conflits de noms, on peut ranger les déclarations dans des espaces de noms distincts.

- Deux syntaxes possibles :

```
namespace nom;  
... code ...
```

porte sur tout le fichier

```
namespace nom {  
    ... code ...  
}
```

permet d'avoir plusieurs espaces de noms dans un même fichier

- Note : les définitions correspondant à un espace de noms peuvent être réparties sur plusieurs fichiers.

Namespaces

- Structure des espaces de noms :
 - espace de noms global :
`namespace { ... }`
 - sous-espaces de noms :
`espace1`
`espace1\sousespaceA`
`espace1\sousespaceA\soussousespaceUn`


Namespaces

- Résolution de noms :
 - nom non qualifié
 1. On recherche dans l'espace de noms courant.
 2. S'il ne s'y trouve pas, on va voir dans l'espace de nom global.
 - nom complètement qualifié `\ns\nom`
Ex : `$a = new \espace1\MonObjet();`
 - nom qualifié `ns\nom`
 - On recherche dans les sous-espaces de l'espace de noms courant.Ex : `sousespaceB\MaClasse::methStat();`

- Déclaration d'alias :

```
use \espace1\MonObjet as MonObjet;  
use sousespaceB\MaClasse as Classe3;  
use ClasseGlobale as CB; // dans l'espace de noms global
```

Optionnel vu qu'on garde le même nom



Late static binding

- Un problème d'héritage et de méthodes de classe...

```
class Mere {  
    static function parle() {  
        echo 'Je suis la classe ' . self::nom() . "\n";  
    }  
    static function nom() {  
        return "Mere";  
    }  
}  
class Fille extends Mere {  
    static function nom() {  
        return "Fille";  
    }  
}  
$f = new Fille();  
$f->parle();  
Fille::parle();
```

```
Je suis la classe Mere  
Je suis la classe Mere
```

Les appels à des méthodes statiques sont généralement résolus à la compilation (idem en Java).

Late static binding

- Un problème d'héritage et de méthodes de classe...

```
class Mere {  
    static function parle() {  
        echo 'Je suis la classe ' . static::nom() . "\n";  
    }  
    static function nom() {  
        return "Mere";  
    }  
}  
class Fille extends Mere {  
    static function nom() {  
        return "Fille";  
    }  
}  
$f = new Fille();  
$f->parle();  
Fille::parle();
```

Permet de retarder (« late... ») la résolution du lien (« ...binding ») jusqu'au moment de l'exécution, où on se base sur le « type » du contenu.

```
Je suis la classe Fille  
Je suis la classe Fille
```