Chapitre 10
Compléments au modèle objet

Surcharge

- Le langage PHP n'accepte pas une vrai surcharge des méthodes
  - Pas de typage des paramètres : surcharge difficile
  - Nombre de paramètre et typage : simuler une surcharge
- Surcharge entre les noms d'attributs et les noms de méthodes
  - Différence syntaxique claire
  - $obj->titre et $obj->titre()
- Surcharge des getter et setter
  - Méthode surchargé de lecture __get($name)
  - Méthode surchargé d'écriture __set($name,$value)
- Surchage de l'opérateur d'appel de fonction
  - _invoke()

Getter du même nom que l'attribut

```
class Exemple {
   private $valeur;

   function __construct($valeur) {
      $this->valeur = $valeur;
   }
   function valeur() {
       return $this->valeur;
    }
}

$e = new Exemple(21);
echo $e->valeur()."\n";
```

- Sécurité d'accès
- Eviter la production de noms inutiles : facilité de lecture

Getter du même nom que l'attribut

```
class Exemple {
  private $valeur;

  function valeur() {
    if (! isset($this->valeur)) {
        $this->valeur = 0;
    }
    return $this->valeur;
  }
}

$e = new Exemple();
echo $e->valeur()."\n";
```

- Déclenchement de code supplémentaire : accès "paresseux"
  - Utile car PHP ne s'exécute que très peu de temps, donc limiter la construction et l'initialisation à juste ce qui est nécessaire.
- Sécurité, mais du code en plus à exécuter

Exemple surcharge des Mutateurs (Mutators)

```
class test {
   public function __set($name, $value) {
       echo "Définition de '$name' à la valeur '$value'\n";
        $this->$name = $value;
   }
   public function __get($name){
        echo "Récupération de '$name'\n";
        return $this->$name;
        }
    }
$a = new test();
$a->attrib = 10;
echo $a->attrib;
```

- Affiche les messages à chaque opération
- Traitement supplémentaires pour tous les attributs

Héritage des méthodes

```
class Mere {
        function affiche() {
            echo "Dans la mère\n";
        }
    }
class Fille extends Mere {

    }
$f = new Fille();
$f->affiche();
```

- PHP : héritage simple.
- Toutes les méthodes héritées
- Méthode identifiée par son nom et par ses paramètres.

Affichage : "Dans la mère"

Héritage des méthodes : remplacement

```
class Mere {
        function affiche() {
            echo "Dans la mère\n";
        }
    }
class Fille extends Mere {
        function affiche() {
            echo "Dans la fille\n";
        }
    }
$f = new Fille();
$f->affiche();
```

- Masque la méthode héritée

Affichage : "Dans la fille"

Héritage des méthodes : enrichissement

```
class Mere {
        function affiche() {
            echo "Dans la mère\n";
        }
    }
class Fille extends Mere {
        function affiche() {
            parent::affiche();
            echo "Dans la fille\n";
        }
    }
$f = new Fille();
$f->affiche();
```

- La re-définiton d'un méthode :
  - appeler **explicitement** la méthode parent
- Opérateur :: pour la résolution de la portée

Affiche : Dans la mère

Dans la fille

Héritage des méthodes : portée

```
class GrandMere {
  function affiche() { echo "Dans la grand mère\n"; }
}

class Mere extends GrandMere {
   function affiche() { echo "Dans la mère\n"; }
}

class Fille extends Mere {
    function affiche() { echo "Dans la fille\n"; }
    function appel() {
       self::affiche();     // la classe : équivalent à $this->
       parent::affiche();   // la première super méthode
       GrandMere::affiche();// une méthode particulière
    }
}
    $f = new Fille();
    $f->appel();
```

Héritage du constructeur

```
class Base {
    function __construct() {
        print "Contructeur de Base\n";
    }
}

class SouClasse extends Base {
    function __construct() {
        parent::__construct();
        print "Constructeur de la sous classe\n";
    }
}
```

- Si non redéfinition : utilise le constructeur hérité
- Si redéfinition : nécessité d'appel **explicite** au constructeur de la classe mère
- Même fonctionnement que les autres méthodes

Foncteur : surcharge de l'opérateur ()

```
class Droite {
  private $a; // Pente de la droite
  private $b; // Valeur initiale

  function __construct($a,$b) { $this->a = $a; $this->b = $b; }

  function __invoke($x) {return $this->a * $x + $this->b; }
}

    $d = new Droite(2,3);
    echo $d(1).' '.$d(2)."\n";
```

- **Foncteur** : objet fonction
- Conserve ses valeurs entre les appels
- Ex: une droite définie comme un foncteur

Attributs de classes

```
class Exemple {
        private static $nbAcces = 0;
        private $nom;

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

        function nom() {
            self::$nbAcces += 1;
            return $this->nom;
        }

        function nbAcces() {
            return self::$nbAcces;
        }
    }
    $moi = new Exemple("Moi");
    echo $moi->nom().' '.$moi->nbAcces()."\n";
    $lui = new Exemple("Lui");
    echo $lui->nom().' '.$lui->nbAcces()."\n";
```

Classe abstraite

```
abstract class Article {
     protected static $TVA = 19.6;
     abstract protected function prixTTC();
}

class TV extends Article {
  function __construct($prix) {$this->prix = $prix;}
  function prixTTC() { return $this->prix  * (1 + parent::$TVA/100);}
}

    $tv = new TV(299);
    echo $tv->prixTTC()."\n";
```

- Mélange méthodes abstraites et concrètes
- Solution (mauvaise) pour l'absence de généricité
- Non instanciable, racine d'héritage
- Mot clée "final" : inverse d'abstract, empêche de surcharger une méthode dans les classes filles.

Interface

```
interface Article {
   public function prixTTC();
}

class TV implements Article {
   static $TVA = 19.6;
   function __construct($prix) {$this->prix = $prix;}

    function prixTTC() { return $this->prix  * (1 + self::$TVA/100);}
}

    $tv = new TV(299);
    echo $tv->prixTTC()."\n";
```

- En plus de l'héritage simple, vient de Java
- Ne remplace pas l'héritage multiple
- Un **contrat passé** : être sûr que les méthodes sont implantés
- Que des définitions de méthodes publiques (rien d'autre)

Sérialisation : usage

- Mécanisme mis en œuvre dans la transformation des objets à préserver lors d'une session.
  - Tout objet stocké dans $_SESSION est sérialisé
  - Possibilité de contrôler en détail
- Permet de ne pas tout sauvegarder :
  - que ce qui est indispensable (ex: très gros objets)
  - opération de nettoyage de l'objet avant arrêt
- Permet de retrouver un état précédent
  - coder des tâches de réinitialisation
  - Ex: ouvrir à nouveau un fichier
  - Ex: ouvrir à nouveau une connexion à un BD

Sérialisation

```
class Article {
  static $TVA = 19.6;
  private $prix;
  private $prixTTC;

  function __construct($prix) {
     $this->prix = $prix; $this->computeTTC();
  }

  function computeTTC() {
    $this->prixTTC = $this->prix * (1 + self::$TVA/100);}
  function prixTTC() { return $this->prixTTC;}

   // Nom des attributs à sauvegarder pour serialize()
   function __sleep() { return array('prix');}
   // Action à faire à la suite d'un unserialize()
   function __wakeup() { $this->computeTTC();}
}
    $tv = new Article(299);
    echo $tv->prixTTC()."\n";
```

le PHP 8 ... plus proche de Java

- Les paramètres peuvent (enfin !) être contraints à un type

  function f1 (int $i, MaClasse $c) { ...}
- Liste de paramètres variables mais du même type

  function f2(int ...$intList) { ...}
- Les méthodes et fonction peuvent (enfin !) avoir un type de retour

  fonction f3(int $i) : int { ... }
- Langage plus "fonctionnel"
  - Fonctions dans un variable !
  - Fermeture (closure) d'une fonction autour d'un objet
  - Permet l'ajout dynamique de méthodes !
  - fonctionnel : style de codage populaire en Javascript et en Java 9

```
class A {private $x = 1;}
$getX = function() {return $this->x;};
echo $getX->call(new A);
```

Vers le PHP 8 : améliorations à la marge

- Nouvel opérateur ?? : "sucre syntaxique" pour remplacer le isset() et un test

  ```
  $username = $_GET['user'] ?? 'nobody';
  ```

- Manipuler des codes UTF-8

  ```
  echo "\u{9999}";
  ```

  affiche 香

- Sécurité du codage avec la vérification par des **assert**()
- Espace de nom comme C++ (**namespace**), avec le **use**
- Opérateur de comparaison **<=>** comme le strcmp() en C
- Définir une constante tableau
- Définir des classe anonymes à la volée et leur instances
- ....

A retenir

- Modèle objet inspiré de Java
  - Portée des attributs et méthodes : public, protected, private
  - Attributs de classe
  - Héritage simple
  - classe **interface**
  - classe **abstraite**
- Gestion précise de la sérialisation
- PHP 7 : plus de contraintes de types dans les paramètres
- PHP 8 : typage des variables ? syntaxe 100% compatible Java ?