

PHP : LA POO

PROGRAMMATION

ORIENTÉES

OBJETS

Avertissement



- ⦿ **Ce cours n'est pas un cours de programmation orientée objets, je rappellerai très brièvement les principes sans approfondir.**
- ⦿ **Ce cours est destiné aux personnes qui connaissent les principes et concepts objet.**
- ⦿ **Ce cours fait le tour de la syntaxe objet pour PHP**

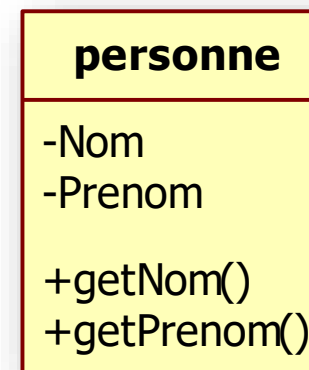
La programmation orientée objets

QU'EST QUE LA POO?

Qu'est-ce que la POO

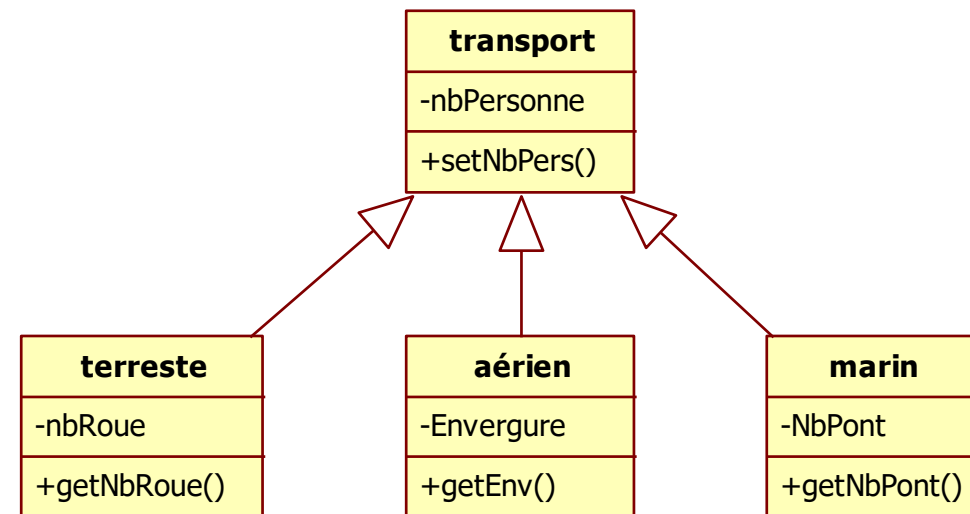
Principe

- ◉ La programmation orientée objets est une autre façon de programmer que la programmation procédurale :
 - Programmation procédurale
 - Séparation le traitement des données
 - Lors de la phase d'analyse on cherche les fonctions du de l'application
 - POO
 - On encapsule données et traitements dans une même boîte, l'objet
 - Lors de la phase d'analyse on cherche les objets qui compose notre application et les interaction entre ces objets



Terminologie

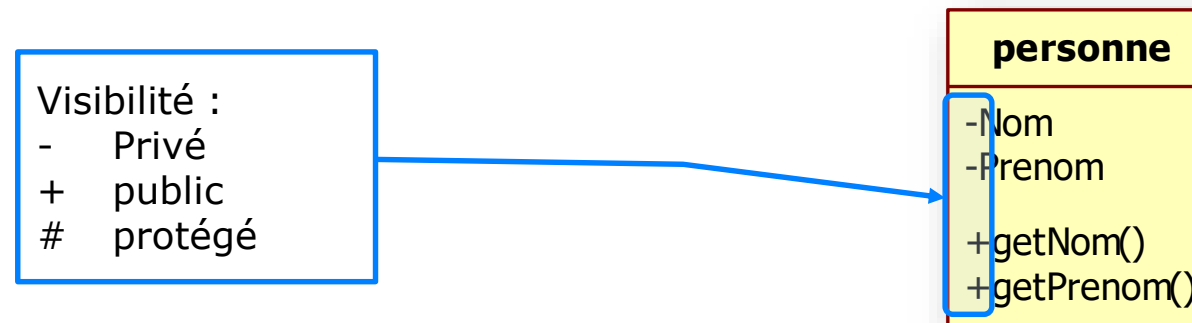
- ⊙ **Une classe** est une représentation abstraite d'un objet. C'est la somme des propriétés de l'objet et des traitements dont il est capable. Une Chaise, un Livre, un Humain, une Rivière sont autant d'exemples possibles de classes. Une classe peut généralement être rendu concrète au moyen d'une instance de classe, on parle alors d'*instance* d'une classe.
- ⊙ On parle **d'héritage** lorsque l'on définit une hiérarchie de classes. On peut par exemple définir la classe *transport* mais elle nous semble trop généraliste: on pourrai ajouter une donnée à cette classe qui définit le type de transport (marin, terrestre, aérien..) mais chaque type de transport à ces propres caractéristiques et fonctions et d'autres qui sont communes à tous ces types de transport, l'héritage permet de créer une classe générique (*transport*) et des classes spécifiques (*marin*, *terrestre*, *aérien*...)



Terminologie

Le principe d'encapsulation

- ⦿ l'encapsulation permet de masquer les données et certaines fonctions à l'utilisateur de la classe. Il pourrait être dangereux de laisser l'utilisateur manipuler ces objets sans aucune restriction.
- ⦿ Les données membres d'une classe doivent donc être invisibles à l'utilisateur de la classe et seules les fonctions membres de la classe peuvent manipuler ces données.
- ⦿ La notion de visibilité nous permet d'instaurer de telles contraintes.



LA POO AVEC PHP

Bref historique de la POO avec php

- ⊙ La programmation orientée objet (POO) a fait son apparition dans la version 3 de PHP. C'était alors simplement un moyen d'autoriser la syntaxe OO (par opposition au procédural), mais les principes de base de la POO ne sont pas implémentés : encapsulation, héritage, visibilité des données membres.
- ⊙ PHP4 a continué dans la lancée, proposant de nouveaux mots clefs, mais toujours sans proposer une syntaxe proche des langages ayant une plus grande maturité comme C++ ou Java.
- ⊙ PHP5, introduit de véritables concepts OO : le constructeur est plus clairement identifié, le destructeur fait son apparition, les objets sont tous pris en charge comme des références, de nouveaux mots clefs font leur apparition (public, protected et private) ainsi que des interfaces et des classes abstraites...

Création d'une classe et manipuler cet objet

Syntaxe de création d'une classe

```
<?php
class Personnage
{
    private $_force = 50;           // La force du personnage, par défaut à 50
    private $_experience = 1;       // Son expérience, par défaut à 1
    private $_degat = 0;            // Ses dégâts, par défaut à 0

    public function deplacer() // déplace le personnage (modifiera sa localisation)
    { // instructions ... }

    public function frapper() // frappe un personnage (suivant la force qu'il a)
    { // instructions ... }
}
?>
```

personnage

- force
- localisation
- experience
- degat

- +frapper()
- +deplacer()

Instancier une classe et manipuler cet objet

- ⦿ Déclaration et définition d'une classe

```
<?php
```

```
class Personnage
```

Définition du nom de la classe

```
{
```

```
private $_force = 50;
```

```
private $_experience = 1;
```

```
private $degat = 0;
```

```
//
```

```
//
```

```
//
```

Déclaration des données membres de la classe:

On définit la visibilité de chaque données membres qui doit être privée pour respecter le principe d'encapsulation

```
public function deplacer() // dépla  
{ // instructions ... }
```

```
public function frapper() // frappe  
{ // instructions ... }
```

Déclaration et définition des fonctions membres de la classe

```
}
```

```
?>
```

Instancier une classe et manipuler cet objet

- Créer un objet

```
<?php
    $perso = new Personnage();
?>
```

vous devez faire précéder le nom de la classe à
instancier du mot-clé *new*

- Appeler une méthode

```
<?php
class Personnage
{
    private $_force = 50;
    private $_experience = 1;
    private $_degat = 0;
    public function parler()
    { echo 'je suis un personnage'; }
}

// appelle de méthode
$perso = new Personnage();
$perso -> parler();
?>
```

l'opérateur *->* permet d'accéder aux membres de la
classe

Instancier une classe et manipuler cet objet

⊙ Accéder à un attribut : l'accesseur

- Les données membres privées ne sont pas accessibles directement, il faut utiliser les accesseurs

```
<?php
class Personnage
{
    private $_force = 50;
    private $_experience = 1;
    private $_degat = 0;
    public function getDegat()
    { return $this->_degat; }
}

// appelle de méthode
$perso = new Personnage();
$perso->_experience;
$perso->getDegat();
?>
```

\$this est une référence à l'objet appelant :

On accède aux membres de la classe à l'aide de cette référence, **on ne doit pas préfixer une variable avec le \$ derrière l'opérateur ->**
\$this->_force

Cet accès à la variable provoque une erreur : Fatal error:
Cannot access private property Personnage::\$_experience
La visibilité est *private*

Pour accéder aux données membres il faut utiliser les méthodes dédiées, ce sont des accesseurs

Instancier une classe et manipuler cet objet

◉ Modifier la valeur d'un attribut : les mutateurs

- Les données membres peuvent être modifiées à travers les mutateurs
- Un mutateur est une méthode qui valorise une donnée membre

```
<?php
```

```
class Personnage
```

```
{
```

```
    private $_force = 50;
```

```
    .....
```

```
    public function setForce(int $force):boolean
```

```
    {if ((!is_int($force)) || ($force >=100) )
```

```
        return false;
```

```
        $this->_force = $force;
```

```
        return true;}  
}
```

```
// appelle de méthode
```

```
$perso = new Personnage();
```

```
if (!$perso ->setForce(50))
```

```
    echo 'erreur sur la valeur de la force';
```

```
?>
```

setForce() est un mutateur, il permet de valoriser une données membres d'un objet

depuis la vesion 7 de php
Il est possible de typer les paramètres
des fonctions et de typer leur sortie

Instancier une classe et manipuler cet objet

◉ Exiger des objets en paramètres

- On peut lors de la définition d'une fonction exiger que le type du paramètre soit un objet

```
<?php
class Personnage
{
    private $_force = 50;
    private $_experience = 1;
    Protected $_degat = 0;
    public function frapper(Personnage $persoAFrapper)
    { $persoAFrapper->_degat += $this->_force; }
}
```

ajouter le nom de la classe dont le paramètre doit être un objet, alors la méthode frapper() ne sera exécutée que si le paramètre passé est de type Personnage, sinon PHP génère une erreur
Le type de la variable à spécifier doit obligatoirement être un nom de classe ou alors un tableau

```
// appelle de méthode
$perso = new Personnage();
$perso2 = new Personnage();
$perso -> frapper(40);
$perso -> frapper($perso2);
$perso -> getDegat();
?>
```

Cet appel génère une erreur:

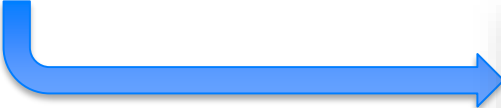
Catchable fatal error: Argument 1 passed to Personnage::frapper() must be an instance of Personnage ...

Cloner des objets

- ⦿ Il est parfois utile de cloner un objet
- ⦿ Un clone créer une copie de l'objet indépendante de l'original
- ⦿ Exemple :

```
class Personnage  
{ ..... }
```

```
$soldat = new Personnage();  
$soldat1 = $soldat ; // copie de la référence de $soldat dans $soldat1  
$soldat2 = clone $soldat; // création d'un clone indépendant de l'originale  
$soldat-> SetForce(80);  
$soldat2 -> SetForce(120);  
echo 'force de perso:'. $soldat -> getForce() .'<br/>';  
echo 'force de perso1 :'. $soldat1 -> getForce() .'<br/>';  
echo 'force de perso2 :'. $soldat2 -> getForce() .'<br/>';
```



```
force de perso :80  
force de perso1 :80  
force de perso2 :120
```

- ⦿ **Remarque** : En faisant `$objet = new Classe;`, `$objet` ne contient pas l'objet lui-même, mais une références vers l'objet

Le constructeur

- ⦿ Le constructeur est la méthode appelée dès que vous instanciez un objet.
- ⦿ Cette méthode accepte des paramètres.

```
<?php
class Personnage
{
    private $_force;
    private $_experience;
    private $_degat;

    public function __construct($force=50,$degat=0)
    { $this->_force=$force; // Initialisation de la force
      $this->_degat=$degat; // Initialisation des dégâts
      $this->_experience = 1; // Initialisation de l'expérience à 1
    }
    .....
}

$perso = new Personnage(60,0);
?>
```


Le destructeur

- La méthode destructeur est appelée dès qu'il n'y a plus de référence sur un objet donné

```
<?php
class Personnage
{
    private $_force;
    private $_experience;
    private $_degat;
    public function __construct($force=50,$degat=0)
    { $this->_force=$force; // Initialisation de la force
      $this->_degat=$degat; // Initialisation des dégâts
      $this->_experience = 1; // Initialisation de l'expérience à 1
    }
    public function __destruct()
    { echo 'je suis mort <br/>'; }}
```

```
$perso = new Personnage(60,0);
unset ($perso);
?>
```

affichera : *je suis mort*
A la place de unset() vous pouvez écrire :
`$perso=NULL;`

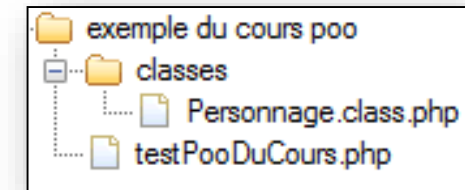
L'auto-chargement de classes

- Il est possible de charger la classe de manière automatique lors de l'instanciation d'un objet de cette classe.
 - Il faut pour cela placer la définition de la classe dans un fichier et le nommer comme la classe, ou du moins le nom de la classe doit apparaitre dans le nom du fichier
 - Pour finir il faut définir une fonction `__autoload($classe)` qui se chargera d'inclure la classe invoquée
- Exemple :

```
2.. Personnage.class.php
1 <?php
2 class Personnage
3 {
4     private $_force ;
5     private $_localisation;
6     private $_experience;
7     private $_degat;
8
9     public function __construct($force=50,$degat=0)
10    { $this->_force=$force; // Initialisation de la force
11      $this->_degat=$degat; // Initialisation des dégâts
12      $this->_experience = 1; // Initialisation de l'expérience à 1
13    }
14 ?>
```

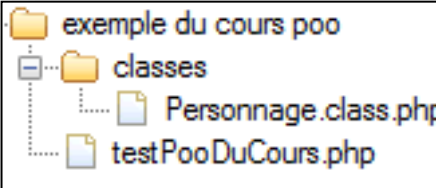
```
<?php
function __autoload($classe) {
    include './classes/'.$classe . '.class.php';
}

//.....
```



L'auto-chargement de classes

- Si on ne nomme pas la fonction d'auto-chargement `__autoload($classe)` il faut alors enregistrer la fonction en autoload avec la fonction `spl_autoload_register('nom de la fonction')`
- la fonction d'auto-chargement `__autoload($classe)` est de fait la fonction `autoload` qui sera appelé lors de l'instanciation
- Exemple :



```
2.. Personnage.class.php
1 <?php
2 class Personnage
3 {
4     private $_force ;
5     private $_localisation;
6     private $_experience;
7     private $_degat;
8
9     public function __construct($force=50,$degat=0)
10     { $this->_force=$force; // Initialisation de la fo
11       $this->_degat=$degat; // Initialisation des dégâ
12       $this->_experience = 1; // Initialisation de l'e
13     }
14 ?>
```

```
1.. testPooDuCours.php
1 <?php
2 function chargerClasse($classe) {
3     include './classes/'.$classe . '.class.php';
4 }
5 spl_autoload_register ('chargerClasse');
6
7 $perso = new Personnage(60,0);
8 $perso2 = new Personnage(40);
9 $perso ->frapper($perso2);
10 $perso ->frapper($perso2);
11 ?>
```

Voir la doc pour savoir comment gérer les erreurs de chargement de classes
<http://php.net/manual/fr/language.oop5.autoload.php>

L'opérateur de résolution de portée (::)

- ⊙ L'opérateur de résolution de portée (aussi appelé *Paamayim Nekudotayim*¹) ou, en termes plus simples, le symbole "double deux-points" (::), fournit un moyen d'accéder aux membres static ou constant, ainsi qu'aux propriétés ou méthodes surchargées d'une classe.

```
<?php
```

```
class Personnage
```

```
{
```

```
    private $_force;
```

```
    ..... •
```

```
    const FORCE_PETITE = 20;
```

```
    const FORCE_MOYENNE = 50;
```

```
    const FORCE_Grande = 80;
```

```
    ..... •
```

```
}
```

```
echo Personnage::FORCE_MOYENNE;
```

Accède aux constantes de la classe sans avoir à l'instancier

```
$perso = new Personnage(Personnage::FORCE_MOYENNE,0);
```

```
?>
```

La définition d'une constante dans une classe est précédé du mot clé *const*

¹ signifie "deux point" en hébreu

Les attributs et méthodes statiques

- Les variables et méthodes "static" sont communes à l'ensemble des objets d'une même classe au moyen de l'opérateur "::".
- On parle de propriétés ou de méthodes "de classe" puisqu'elles n'appartiennent pas à un objet en particulier.

```
<?php
class Personnage
{
    private static $_nbPersonnage;
    .....
    public function __construct()
    { ++self::$_nbPersonnage; }
    public static function getNbPersonnage()
    { return self::$_nbPersonnage; }
    .....
}

$perso = new Personnage();
$perso2 = new Personnage();
echo Personnage::getNbPersonnage(); // affiche 2
?>
```

Le mots clé "**self**" combiné à l'opérateur "::" permet d'accéder aux données "static" de la classe

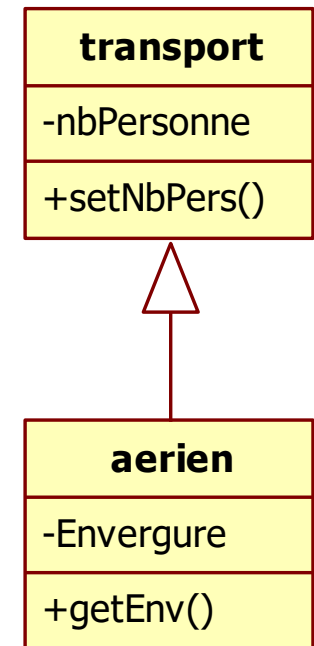
Appel de la méthode sans passer par l'instance

Notion d'héritage

- ◉ Quand on parle d'héritage, c'est qu'on dit qu'une classe B hérite d'une classe A.
- ◉ La classe A est donc considérée comme la classe mère et la classe B est considérée comme la classe fille.
- ◉ La classe fille hérite de toutes les données et méthodes de la classe mère
- ◉ Une classe ne peut hériter que d'une seule autre classe, et ***l'héritage multiple n'est pas supporté***
- ◉ Dans l'exemple ci-contre la classe *aérien* est une classe fille de La classe *transport*, la classe *aérien* hérite de toutes les propriétés et méthodes la classe *transport*.

Syntaxe

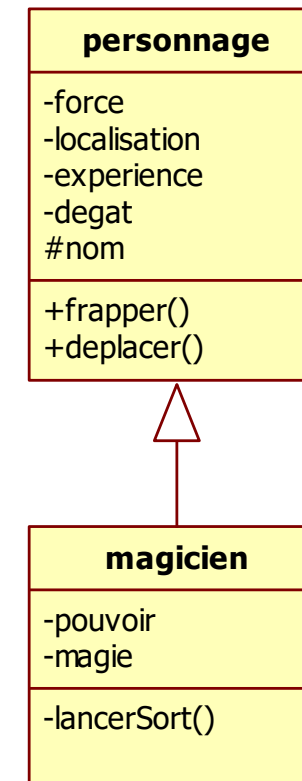
```
<?php
class transport // Création de la classe mère.
{ ..... }
class aerien extends transport // Création de la classe fille
{ ..... }
?>
```



Notion d'héritage

- Opérateur de portée parent couplé au :: permet d'accéder aux données membres et aux méthodes de la classe mère
- Il faut que les données accessibles de la classe fille soient déclarées comme protégées (protected) dans la classe mère.

```
1 <?php
2 class magicien extends Personnage
3 {
4     private $_magie ;
5     private $_pouvoir;
6
7     public function __construct($force=50,$degat=0,$pouvoir=2) {
8         parent::__construct($force,$degat); // appel du constructeur parent
9         $this->_magie=parent::FORCE_Grande; // Initialisation de la force du magicien
10        $this->_pouvoir=$pouvoir; // coefficient multiplicateur de force
11    }
12
13    public function getForce() // redéfinition de la fonction GetForce()
14    { return $this->_magie;}
15
16    public function getPouvoir()
17    { return $this->_pouvoir;}
18
19    public function lancerUnSort(Personnage $persoAFrapper) {
20        parent::frapper($persoAFrapper); // appel de la méthode parent frapper
21        $persoAFrapper->_degat += $this->_magie*$this->_pouvoir;
22    }
23 }
24 ?>
```



Notion d'héritage

- Opérateur de portée parent couplé au :: permet d'accéder aux données membres et aux méthodes de la classe mère
- Il faut que les données accessibles de la classe fille soient déclarées comme protégées (protected) dans la classe mère.

```
1 <?php
2 class magicien extends Personnage
3 {
4     private $_magie ;
5     private $_pouvoir;
6
7     public function __construct($force=50,$degat=0,$pouvoir=2) {
8         parent::__construct($force,$degat); // appel du constructeur parent
9         $this->_magie=parent::FORCE_Grande; // Initialisation de la force du magicien
10        $this->_pouvoir=$pouvoir; // coefficient multiplicateur de force
11    }
12
13    public function getForce() // redéfi
14    { return $this->_magie;}
15
16    public function getPouvoir()
17    { return $this->
18
19    public function lancer
20        parent::frapper(
21            $persoAFrapper->_degat + $this->_magie * $this->_pouvoir,
22    }
23 }
24 ?>
```

Le mots clé "**parent**" combiné à l'opérateur "::" permet d'accéder aux données "static" et protéger (protected) de la classe mère et aux méthodes de la classe mère

Surcharge du constructeur de la classe mère



Si vous surchargez une méthode, sa visibilité doit être la même que dans la classe parent! Si tel n'est pas le cas, une erreur fatale sera levée. Par exemple, vous ne pouvez surcharger une méthode publique en disant qu'elle est privée.

visibilité protégée : *protected*

- La visibilité des données ou méthodes d'une classe mère doivent être de type protéger (protected) si la classe fille doit y accéder

```
1 <?php
2 class magicien extends Personnage
3 {
4     private $_magie ;
5     private $_pouvoir;
6
7     public function __construct($force=
8         parent::__construct($force,$degat
9         $this->_magie=parent::FORCE_GRAND
10        $this->_pouvoir=$pouvoir; // coé
11    }
12
13    public function getForce() // redéfi
14        { return $this->_magie;}
15
16    public function getPouvoir()
17        { return $this->_pouvoir;}
18
19    public function lancerUnSort(Personnage $persoAFrapper) {
20        parent::frapper($persoAFrapper); // appel de la méthode parent frapper
21        $persoAFrapper->degat += $this->_magie*$this->_pouvoir;
22    }
23 }
24 ?>
```

Dans la classe *magicien* (fille) on accède à la donnée *\$_degat* de l'instance de la classe *Personnage* (mère)
Dans la classe *Personnage* la donnée *\$_degat* doit être *protected*

```
1 <?php
2 abstract class Personnage
3 {
4     private $_force ;
5     private $_localisation;
6     private $_experience;
7     protected $_degat;
8     private static $_nbPersonnage;
```

Classes et méthodes abstraites

Classe abstraite

- ⊙ C'est une classe dont l'implémentation n'est pas complète et qui n'est pas instanciable
- ⊙ Très utile pour déclarer des méthodes dans la classe mère et les définir dans les classes fille
- ⊙ Une classe abstraite doit impérativement être dérivée, elle doit avoir au moins une classe fille

Méthode abstraite

- ⊙ Définir une méthode abstraite force toutes les classes fille à écrire cette méthode
- ⊙ On ne doit spécifier aucune instruction dans une méthode abstraite, puisqu'elle doit être définie dans les classes fille
- ⊙ Pour définir une méthode comme étant abstraite, il faut que la classe elle-même soit abstraite !

Classes et méthodes abstraites

Syntaxe :

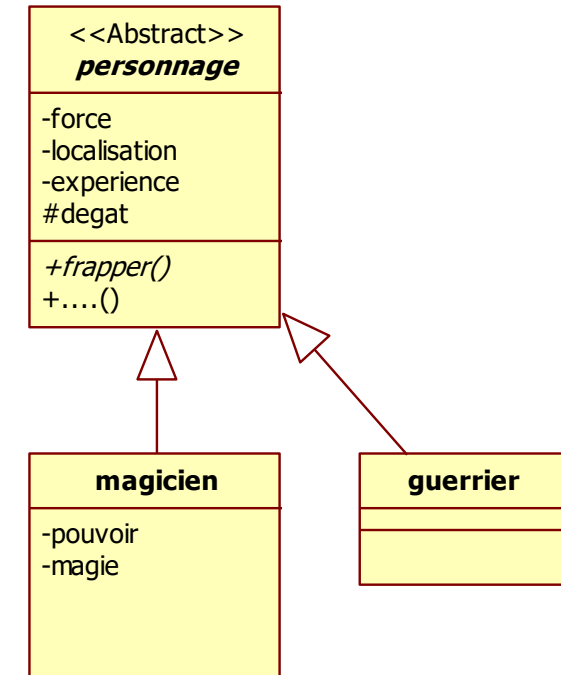
```
2.. Personnage.class.php
1 <?php
2  abstract class Personnage
3  {
4      private $_force ;
5      private $_localisation;
6      private $_experience;
7      protected $_degat;
8      private static $_nbPersonnage;
9      //.....
10
11     public function __construct($force=50,$degat=0)
12     { $this->_force=$force; // Init
13       $this->_degat=$degat; // Init
14       $this->_experience = 1; // In
15       ++self::$_nbPersonnage;
16     }
17
18     abstract public function frapper(Personnage $persoAFrapper) ;
19     //.....
```

Le nom de la classe est précédé du mot-clé *abstract*

Le nom de la méthode est précédé du mot-clé *abstract* et la fonction ne contient pas d'instruction

La méthode est définie dans la classe fille

```
2.. Personnage.class.php
1 <?php
2  class guerrier extends Personnage
3  {
4
5      public function frapper(Personnage $persoAFrapper)
6      { $persoAFrapper->_degat += $this->_force; }
7  }
```



Classes et méthodes finales

- ⊙ Lorsque l'on définit une classe comme « finale », cela signifie qu'elle ne pourra plus être dérivée par une sous-classe. Cela implique également que ses attributs et méthodes ne pourront plus être redéfinis.
- ⊙ En revanche, si l'on applique le mot-clé « *final* » à une méthode d'une classe, alors c'est uniquement cette méthode qui ne pourra plus être redéfinie dans les classes dérivées.
- ⊙ En interdisant la dérivation d'une classe ou la redéfinition (surchage) des méthodes d'une classe, cela vous permet de vous assurer que le développeur ne contournera pas directement la logique que vous avez mise en place

Classes et méthodes finales

Syntaxe

3.. Personnage.class.php

```
1 <?php
2 abstract class Personnage
3 {
4     private $_force ;
5     private $_localisation;
6     private $_experience;
7     protected $_degat;
8     private static $_nbPersonnage;
9     //.....
10
11     public function __construct($force=50,$degat=0)
12     { $this->_force=$force; // Initialisation de la force
13       $this->_degat=$degat; // Initialisation des dégâts
14       $this->_experience = 1; // Initialisation de l'expérience à 1
15       ++self::$_nbPersonnage;
16     }
17
18     final public function getDegat()
19     { return $this->_degat;}
20     //.....
```

4.. magicien.class.php

```
1 <?php
2 final class guerrier extends Personnage
3 {
4     public function frapper(Personnage $persoAFrapper)
5     { $persoAFrapper->_degat += $this->_force; }
6 }
7 ?>
```

Le nom de la classe est précédé du mot-clé *final*

Le nom de la méthode est précédé du mot-clé *final*

La surcharge d'une méthode finale provoque l'erreur :

Fatal error: Cannot override final method Personnage::getDegat()

La dérivation d'une classe finale provoque l'erreur :

Fatal error: Class soldat may not inherit from final class

Les interfaces

Les interfaces

- ⊙ C'est un ensemble de méthodes que les classes doivent définir si elles veulent l'implémenter.
- ⊙ Toutes les méthodes déclarées dans une interface doivent être publiques.
- ⊙ La classe implémentant l'interface doit utiliser exactement les mêmes signatures de méthodes que celles définies dans l'interface
- ⊙ Les interfaces peuvent être étendues comme des classes (notion d'héritage), en utilisant l'opérateur extends.
- ⊙ Les interfaces peuvent contenir des constantes. Les constantes d'interfaces fonctionnent exactement comme les constantes de classe mais ne peuvent pas être écrasées par une classe ou une interface qui en hérite.
- ⊙ Une interface peut étendre plusieurs interfaces, cad que contrairement à l'héritage entre classe qui ne peut être multiple, une interface peut étendre (hériter) plusieurs interfaces
- ⊙ Une classe ne peut implémenter deux interfaces qui partagent des noms de fonctions identique, puisque cela causerait une ambiguïté.
- ⊙ Une interface ne peut pas contenir de variables

Fatal error: Interfaces may not include member variables

Les interfaces

Différence entre interface et classe abstraite

- ⊙ Une **classe abstraite** :
 - Sert de classe de base à des classes plus spécialisées.
 - Une classe abstraite ne sera jamais instanciée.
 - Une classe abstraite contient des méthodes implémentées qui sont communes à d'autres classes filles et qui ont généralement le même comportement.
- ⊙ Une **interface** n'a pas du tout ce but.
 - Une interface sert uniquement à définir les prototypes de méthodes (donc pas de code dans cette classe, juste les prototypes de fonctions) qui devront obligatoirement être redéfinies dans les classes qui implémentent cette interface.
 - Elles regroupent les méthodes communes à plusieurs classes mais dont le comportement est différent.

Les interfaces

Syntaxe

```
1 <?php
2 interface interfaceSoldat
3 {
4     const NB_VIE = 5;
5     public function tuer(Personnage $persoAFrapper) ;
6     public function fuir() ;
7 }
8
9 class soldat implements interfaceSoldat
10 {
11     private $_force;
12     private $_lieu;
13     public function tuer(Personnage $persoAFrapper)
14     { $persoAFrapper->_force = NULL;}
15     public function fuir()
16     { $persoAFrapper->$_lieu = 'loin';}
17 }
18 ?>
```

Une interface est définie par le mot clé *interface*

Seul le prototype des fonctions est défini dans l'interface

La classe qui implémente une interface

Définition des méthodes

- ⦿ **Pour accéder aux constantes définies dans l'interface**

```
18 echo 'nombres de vie :'.soldat::NB_VIE.'<br/>';
19 echo 'nombres de vie :'.interfaceSoldat::NB_VIE.'<br/>';
```


Les interfaces

Ecrasement de constante

```
1 <?php
2 interface interfaceSoldat
3 {
4     const NB_VIE = 5;
5     public function tuer(Personnage $persoAFrapper) ;
6     public function fuir() ;
7 }
8
9 class soldat implements interfaceSoldat
10 {
11     const NB_VIE = 10;
12     private $_force;
13     private $_lieu;
14     public function tuer(Personnage $persoAFrapper)
15     { $persoAFrapper->force = NULL;}
16     public function fuir()
17     { $persoAFrapper->$_lieu = 'loin';}
18 }
19 ?>
```

Écraser la constante dans classe qui implémente l'interface :
provoque une erreur

Fatal error: Cannot inherit previously-inherited or override constant NB_VIE from interface interfaceSoldat :

Les interfaces

Ecrasement de constante

```
1 <?php
2 interface interfaceSoldat
3 {
4     const NB_VIE = 5;
5     public function tuer(Personnage $persoAFrapper) ;
6     public function fuir() ;
7 }
8
9 class soldat implements interfaceSoldat
10 {
11     const NB_VIE = 10;
12     private $_force;
13     private $_lieu;
14     public function tuer(Personnage $persoAFrapper)
15     { $persoAFrapper->_force = NULL;}
16     public function fuir()
17     { $persoAFrapper->$_lieu = 'loin';}
18 }
19 ?>
```

Fatal error: Cannot inherit previously-inherited or override constant NB_VIE from interface interfaceSoldat :

Les interfaces

Héritage de plusieurs interfaces

```
1 <?php
2 interface interfaceSoldat
3 {
4     const NB_VIE = 5;
5     public function tuer(Personnage $persoAFrapper) ;
6 }
7
8 interface interfaceMonstre
9 {
10     public function manger($qte) ;
11 }
12
13 interface interfaceMonstreSoldat extends interfaceSoldat, interfaceMonstre
14 {
15     public function fuir() ;
16 }
17
18 class Soldat implements interfaceMonstreSoldat
19 {
20     private $_force;
21     private $_lieu;
22     public function tuer(Personnage $persoAFrapper)
23     { $persoAFrapper->_force = NULL;}
24     public function fuir()
25     { $persoAFrapper->$_lieu = 'loin';}
26     public function manger($qte)
27     { $this->_force = $qte;}
28 }
29 ?>
```

Interface qui hérite de deux autres interfaces

La classe qui implémente l'interface qui hérite de deux autres interfaces

Il faut définir les fonctions des trois interfaces que la classe implémente

Les interfaces prédéfinies

- ⊙ Php propose des interfaces prédéfinies dont vous devez implémenter leur méthodes :
 - Iterator
 - SeekableIterator
 -
- ⊙ Je vous renvoie à la documentation pour avoir un aperçu des interfaces
 - <http://www.siteduzero.com/tutoriel-3-165923-interfaces-predefinies.html>
 - <http://www.php.net/manual/fr/reserved.interfaces.php>

Les traits

Les traits

- ◉ **Depuis PHP 5.4.0, PHP supporte une manière de réutiliser le code appelée Traits.**
- Les traits sont un mécanisme de réutilisation de code dans un langage à héritage simple tel que PHP.
- Un trait tente de réduire certaines limites de l'héritage simple, en autorisant le développeur à réutiliser un certain nombre de méthodes dans des classes indépendantes. La sémantique entre les classes et les traits réduit la complexité et évite les problèmes typiques de l'héritage multiple.
- Un trait est semblable à une classe, mais il ne sert qu'à grouper des fonctionnalités d'une manière intéressante. Il n'est pas possible d'instancier un Trait en lui-même. C'est un ajout à l'héritage traditionnel, qui autorise la composition horizontale de comportements, c'est à dire l'utilisation de méthodes de classe sans besoin d'héritage.

Les traits

Exemple d'utilisation de Trait

```
<?php
    trait ezReflectionReturnInfo {
        function getReturnType() { /*1*/ }
        function getReturnDescription() { /*2*/ }
    }

    class ezReflectionMethod extends ReflectionMethod {
        use ezReflectionReturnInfo;
        /* ... */
    }

    class ezReflectionFunction extends ReflectionFunction {
        use ezReflectionReturnInfo;
        /* ... */
    }
?>
```

Les traits

Précédence

- Une méthode héritée depuis une classe mère est écrasée par une méthode issue d'un Trait. L'ordre de précédence fait en sorte que les méthodes de la classe courante écrasent les méthodes issues d'un Trait, elles-mêmes surchargeant les méthodes héritées.
- Une méthode héritée depuis la classe de base est écrasée par celle provenant du Trait. Ce n'est pas le cas des méthodes réelles, écrites dans la classe de base.

Les traits

Précédence

- Exemple : Exemple avec l'ordre de précedence.

```
<?php
class Base {
    public function sayHello() {
        echo 'Hello ';
    }
}

trait SayWorld {
    public function sayHello() {
        parent::sayHello();
        echo 'World!';
    }
}

class MyHelloWorld extends Base {
    use SayWorld;
}

$o = new MyHelloWorld();
$o->sayHello();

?>
```

L'exemple va afficher :
Hello World!

Les traits

Précédence

- Autre exemple d'ordre de précedence.

```
<?php
    trait HelloWorld {
        public function sayHello() {
            echo 'Hello World!';
        }
    }

    class TheWorldIsNotEnough {
        use HelloWorld;
        public function sayHello() {
            echo 'Hello Universe!';
        }
    }

    $o = new TheWorldIsNotEnough();
    $o->sayHello();

?>
```

La méthode écrase le comportement
De la méthode *sayHello()* définit dans
le trait

L'exemple va afficher :
Hello Universe!

Les traits

Multiples Traits

- Une classe peut utiliser de multiples Traits en les déclarant avec le mot-clé use, séparés par des virgules.

```
<?php
    trait Hello {
        public function sayHello() {
            echo 'Hello ';
        }
    }

    trait World {
        public function sayWorld() {
            echo 'World';
        }
    }

    class MyHelloWorld {
        use Hello, World;
        public function sayExclamationMark() {
            echo '!!';
        }
    }

    $o = new MyHelloWorld();
    $o->sayHello();
    $o->sayWorld();
    $o->sayExclamationMark();

?>
```

Appel multiples de traits

L'exemple va afficher :
Hello word!

Les traits

Pour d'informations sur les traits je vous renvoie à la documentation :

<http://php.net/manual/fr/language.oop5.traits.php>

Les méthodes magiques

- ⊙ Une méthode magique est une méthode qui, si elle est présente dans votre classe, sera appelée lors de tel ou tel évènement.
- ⊙ Si la méthode n'existe pas et que l'évènement est exécuté, aucun effet "spécial" ne sera ajouté, l'évènement s'exécutera normalement.
- ⊙ Le but des méthodes magiques est d'intercepter un évènement, et de dérouter le comportement normal déclenché par cet événement, par exemple que faire si le développeur essaie d'accéder à une donnée privée de l'objet
- ⊙ Nous avons déjà vu quelques une comme `__construct()` et `__destruct()`
- ⊙ Les méthodes magiques sont toutes préfixées par un double underscore (`__`)
- ⊙ Dans la suite je ne passerai pas en revue toutes les méthodes magiques mais seulement quelques unes, le principe étant le même pour toutes

Les méthodes magiques

⊙ Les méthodes magiques disponibles:

- `__construct()` : Constructeur de la classe ;
- `__destruct()` : Destructeur de la classe ;
- `__set()` : Déclenchée lors de l'accès en écriture à une propriété de l'objet ;
- `__get()` : Déclenchée lors de l'accès en lecture à une propriété de l'objet ;
- `__call()` : Déclenchée lors de l'appel d'une méthode inexistante de la classe (appel non statique)
- `__callstatic()` : Déclenchée lors de l'appel d'une méthode inexistante de la classe (appel statique) : disponible depuis PHP 5.3 et 6.0 ;
- `__isset()` : Déclenchée si on applique `isset()` à une propriété de l'objet ;
- `__unset()` : Déclenchée si on applique `unset()` à une propriété de l'objet ;
- `__sleep()` : Exécutée si la fonction `serialize()` est appliquée à l'objet ;
- `__wakeup()` : Exécutée si la fonction `unserialize()` est appliquée à l'objet ;
- `__toString()` : Appelée lorsque l'on essaie d'afficher directement l'objet : `echo $object;` ;
- `__set_state()` : Méthode statique lancée lorsque l'on applique la fonction `var_export()` à l'objet ;
- `__clone()` : Appelée lorsque l'on essaie de cloner l'objet ;
- `__autoload()` : Cette fonction n'est pas une méthode, elle est déclarée dans le scope global et permet d'automatiser les "include/require" de classes PHP.

Les méthodes magiques

- Les méthodes **__set(\$name,\$val)** et **__get(\$name)**
 - Les paramètres \$name et \$val sont générés automatiquement et contiennent respectivement le nom de la donnée à laquelle on essaie d'accéder et la valeur que l'on essaie de lui donner
 - Vous nommez ces paramètres comme bon vous semble

```
1 <?php
2 final class guerrier extends Personnage
3 {
4     public function frapper(Personnage $perso)
5     { $perso->degat += $this->degat; }
6
7     public function __set($nom, $valeur)
8     { trigger_error("Vous ne pouvez pas valoriser cette attribut($nom)
9       avec la valeur :$valeur utiliser
10       la méthode setForce($valeur) !<br />", E_USER_ERROR); }
11
12     public function __get($nom)
13     { trigger_error("Impossible d'accéder à l'attribut : $nom
14       utiliser la méthode getForce() !<br />", E_USER_ERROR); }
15 }
16 }
17 ?>
```

La méthode magique __set() est appelée dès que l'on essaie de valoriser une donnée de l'objet

La méthode magique __get() est appelée dès que l'on essaie de lire une donnée de l'objet

Les méthodes magiques

⊙ Les méthodes `__set($name,$val)` et `__get($name)`

- L'instruction : `$guerrier->_force=30;` génère l'erreur suivante :

Fatal error: Vous ne pouvez pas valoriser cette attribut(\$_force) avec la valeur :30 utiliser la méthode setForce(30) !
in C:\Document perso\IUT ARLES\0-Cours IUT ARLES\1 - Cours DUT INFO\OMGL S3 - SGBD 3 (PHP+M)

- Le moteur php appelle la méthode magique `__set()` déclenchée par la valorisation directe à une donnée privée.

```
public function __set($nom, $valeur)
{
    trigger_error("Vous ne pouvez pas valoriser cette attribut($$nom)
    avec la valeur :$valeur utiliser
    la méthode setForce($valeur) !<br />", E_USER_ERROR);
}
```

- L'instruction : `echo $guerrier->_force;` génère l'erreur suivante

Fatal error: Impossible d'accéder à l'attribut : \$_force utiliser la méthode getForce() !
in C:\Document perso\IUT ARLES\0-Cours IUT ARLES\1 - Cours DUT INFO\O

- Le moteur php appelle la méthode magique `__get()` déclenchée par l'accès à une donnée privée.

```
public function __get($nom)
{
    trigger_error( "Impossible d'accéder à l'attribut : $$nom
    utiliser la méthode getForce() !<br />", E_USER_ERROR);
}
```

Les exceptions

- ⊙ Le mécanisme des exceptions a été introduit à PHP dans sa version 5 en complément de son nouveau modèle orienté objet.
- ⊙ les exceptions permettent de simplifier, personnaliser et d'organiser la gestion des « erreurs » dans un programme informatique.
- ⊙ Ici le mot « erreurs » ne signifie pas « bug », qui est un comportement anormal de l'application développée, mais plutôt « cas exceptionnel » à traiter différemment dans le déroulement du programme
- ⊙ Les exceptions utilisent la classe native *Exception* du php dont un extrait est donné ici

```
class Exception
{
    protected $message = 'exception inconnu'; // message de l'exception
    protected $code = 0;                      // code de l'exception défini par l'utilisateur
    protected $file;                          // nom du fichier source de l'exception
    protected $line;                          // ligne de la source de l'exception

    function __construct(string $message=NULL, int code=0);

    final function getMessage();              // message de l'exception
    final function getCode();                 // code de l'exception
    final function getFile();                 // nom du fichier source
    final function getLine();                 // ligne du fichier source
    final function getTrace();                // un tableau de backtrace()
    final function getTraceAsString();        // chaîne formatée de trace

    /* Remplacable */
    function __toString();                    // chaîne formatée pour l'affichage
}
```


Les exceptions

Générer, lancer et attraper des exceptions à travers le programme

- Générer une exception (**new**)

```
<?php

// Création de l'objet Exception
$e = new Exception('Une erreur s\'est produite');

// Affiche le message d'erreur
echo $e->getMessage();

?>
```

- La première ligne créer l'objet de type Exception (\$e) et assigne automatiquement le message d'erreur dans le constructeur.
- La seconde ligne de code affiche le message d'erreur enregistré sur la sortie standard.
 - en développement informatique, les programmeurs ont pris l'habitude de nommer une exception uniquement avec la lettre miniscule "e". C'est à la fois une convention de nommage et une bonne pratique très largement répandue. Toutefois, aucune règle n'oblige les développeurs à l'adopter.
 - la classe native Exception est chargée automatiquement par PHP, c'est pourquoi il n'est pas nécessaire d'avoir recours à un quelconque import avant de pouvoir l'utiliser.

Les exceptions

Générer, lancer et attraper des exceptions à travers le programme

- Lancer une exception à travers le programme (***throw***)
 - Le code ne nous sert strictement à rien puisqu'une exception n'est utile que si elle est créée lorsqu'un évènement exceptionnel se déroule pendant l'exécution du programme. Par exemple : une requête SQL qui échoue ..
 - Lorsqu'un tel évènement se produit, c'est que quelque chose d'inhabituel s'est passé. Par conséquent, le programme doit donc être interrompue et signaler l'incident. Pour réaliser cette opération, le programme doit automatiquement « *lancer* » une exception. Le lancement d'une exception provoque immédiatement l'interruption du déroulement normal du programme
 - Le lancement d'une exception à travers le programme est réalisée grâce au mot-clé « *throw* ».

```
function additionner($a, $b)
{
    if (!is_numeric($a) OR !is_numeric($b))
    {
        // On lance une nouvelle exception grâce à throw et on instancie dire
        throw new Exception('Les deux paramètres doivent être des nombres');
    }

    return $a + $b;
}
```

Les exceptions

Générer, lancer et attraper des exceptions à travers le programme

- Lancer une exception à travers le programme (***throw***)

```
function additionner($a, $b)
{
    if (!is_numeric($a) OR !is_numeric($b))
    {
        // On lance une nouvelle exception grâce à throw et on instancie dire
        throw new Exception('Les deux paramètres doivent être des nombres');
    }

    return $a + $b;
}
```

- remarquez que l'exception générée se fait à la volée. Du fait qu'elle est automatiquement renvoyée au programme, nul besoin de stocker l'objet créé dans une variable.
- les exceptions peuvent également être lancées depuis l'intérieur d'une classe.

Les exceptions

Générer, lancer et attraper des exceptions à travers le programme

- ⦿ Interceptor / attraper une exception générée (***try { } catch() { }***)
 - Une fois l'exception lancée il va falloir traiter cette exception
 - Pour cela, il est nécessaire de pouvoir « ***intercepter / attraper*** » l'exception générée pour appliquer le traitement adéquat. C'est là qu'intervient le bloc ***try / catch***.

```
<?php
function additionner($a, $b)
{
    if (!is_numeric($a) OR !is_numeric($b))
    {
        throw new Exception('Les deux paramètres doivent être des nombres');
    }

    return $a + $b;
}

try // Nous allons essayer d'effectuer les instructions situées dans ce bloc.
{
    echo additionner(12, 3), '<br />';
    echo additionner('azerty', 54), '<br />';
    echo additionner(4, 8);
}

catch (Exception $e) // Nous allons attraper les exceptions "Exception" s'il y
{
    echo 'Une exception a été lancée. Message d\'erreur : ', $e->getMessage();
}

echo 'Fin du script'; // Ce message s'affiche, ça prouve bien que le script es
?>
```

Les exceptions

Générer, lancer et attraper des exceptions à travers le programme

- ⦿ Interceptor / attraper une exception générée (***try { } catch() { }***)
 - Le bloc ***try*** « *essaie* » d'exécuter le script entre les deux premières accolades. Si une exception est lancée dans ce bloc, elle est immédiatement « *attraper* » dans le bloc ***catch()*** et les traitements particuliers sont exécutés à la place.

```
<?php
function additionner($a, $b)
{
    if (!is_numeric($a) OR !is_numeric($b))
    {
        throw new Exception('Les deux paramètres doivent être numériques');
    }

    return $a + $b;
}

try // Nous allons essayer d'effectuer les instructions situées dans ce bloc
{
    echo additionner(12, 3), '<br />';
    echo additionner('azerty', 54), '<br />';
    echo additionner(4, 8);
}

catch (Exception $e) // Nous allons attraper les exceptions "Exception" s'il y en a
{
    echo 'Une exception a été lancée. Message d\'erreur : ', $e->getMessage();
}

echo 'Fin du script'; // Ce message s'affiche, ça prouve bien que le script est terminé
?>
```

Le code qui est susceptible de générer le lancement d'une exception doit être placé dans le ***try***

L'exception est attrapée et traitée dans le ***catch***

Quelques liens utiles

Quelques cours de POO en php

- ◉ <http://www.siteduzero.com/tutoriel-3-147180-programmez-en-orientee-objet-en-php.html>
- ◉ <http://g-rossolini.developpez.com/tutoriels/php/cours/?page=poo#LIV-A-7>
- ◉ <http://www.apprendre-php.com/tutoriels/>

Quelques fonctions sur les classes et objets

- ◉ <http://fr2.php.net/manual/fr/function.is-a.php>

Singleton : instance unique d'une classe

- ◉ <http://www.apprendre-php.com/tutoriels/tutoriel-45-singleton-instance-unique-d-une-classe.html>
- ◉ [http://fr.wikipedia.org/wiki/Singleton %28patron de conception%29#PHP 5](http://fr.wikipedia.org/wiki/Singleton_%28patron_de_conception%29#PHP_5)