Héritage

Comme en Java, il n'y a **pas d'héritage multiple** en php.

```php
class chien extends Animal{
    function affiche(){
        parent::affiche(); //référence vers la classe mère (statique ou non)
    }
}
```

Constructeur

Contrairement au java, ou il faut faire un appel explicite au constructeur parent, le php hérite de la méthode constructeur comme n'importe quelle autre :

```php
function __construct($nom,$prenom,$age,$login){
    parent::__construct($nom, $prenom, $age);
    this->login = $login;
}
```

> si le constructeur de la mère à besoin d'un argument, la fille aura besoin des mêmes arguments.

> Si on redéfinit le constructeur. il n'y a pas d'appel implicite à la classe mère. si on en veut un, on doit l'appeler :
>
> ```php
> parent:__construct();
> ```

final

Une **fonction** déclarée dans le parent comme final ne pourra **pas être redéfinie** par les classes filles. Dans le même ordre d'idée une **classe** définie comme finale ne pourra **pas** avoir **de filles**.

final static

(ou constantes propres)

```
const MAX_AGE = 99;
```

Accès à une propriété statique

```
class Etudiant{
    static $nom;
    function hello(){
        echo "hello" . self::$nom;
    }
}
```

On utilisera à l'extérieur/intérieure de la classe :

```
Etudiant::$nom;
```

mais on peut aussi utiliser à intérieur de la classe :

```
self::$nom;
```

> this->nom ne fonctionne pas et les méthodes statiques vont générer un warning

Abstract

Une classe implémenter avec abstract doit implémenter toutes les méthodes abstraites ou être abstraites elles même.

```
abstract class MaClasse{
    abstract function Mafunction{
        //...
    }
}
```

Interfaces

Il est conseillé de faire commencer le nom de la classe par I

```
interface IMonstre{ \\... }

class LoupGarou implements IMonstre{
    // Doit implémenter toutes les fonctions de  IMonstre
}
```

Opérateur ::

il est utilisé dans les exemples ci dessus mais il peut aussi être utilisé comme suit :

```
$nomDeClasse = 'Etudiant';
$nomDeClasse::$AgeMaximum;
```

Traits

Les traits sont des moyens de mettre en commun des éléments commun à une classe ensemble (et aussi de faire du multi-héritage). un trait peut être abstrait.

On peut aussi **utiliser un trait pour en définir un autre**.

Créer

```
trait message {
  public function msg() {
    echo "Bonjour, je suis un message dans un trait";
  }
}
trait monTrait {
  public function bob() {
    echo "Bob";
  }
}
```

Utilisation

```
class Welcome {
  use message,monTrait;
}
```

Priorité des fonctions

Si une fonction est définie dans un trait, dans la classe mère et redéfinie dans une classe, on suivra l'ordre suivant.

1. Redéfinition dans la classe
2. Définition dans le trait
3. Définition dans la classe mère

Renommer une fonction ou propriété d'un trait

```
class Welcome {
  use monTrait{
      monTrait::bob as NomMoinsStupide;
  }
}
```

Une même fonction ou propriété dans Différents traits

C'est interdit, sauf si on renomme l'une des 2 méthodes/propriétés.

```php
trait message {
  public function msg() {
    echo "Bonjour, je suis un message dans un trait";
  }
}
trait message2 {
  public function msg() {
    echo "Bonjour, je suis un message dans un autre trait";
  }
}

class welcome{
    use message,message2{
        message::msg insteadof message2; //cette fonction reste msg
        message2::msg as msg2 //cette fonction devient msg2
    }
}
```

Late State Binding

```php
Class Mere{
    static function parle(){
        echo 'je suis un'.self::nom()."\n";
    }
    static function nom(){
        return 'mère';
    }
}
Class Fille extends Mere{
    static function nom(){
        return 'fille';
    }
}
$f = new Fille();
$f->parle();
Fille::parle(); // retrourne 'je suis mère'
```

Le Problème du Late state binding vient du fait que la résolution de parle de la classe mère est fait à la 'compilation'.

Il faut donc remplacer le **self::** par un **static::**

> Il permets de retarder la résolution du lien jusqu'au moment de l'exécution où on se base sur le typen de contenu.