le cnam

PHP5
Programmation OO

J-F Dazy

J-F Berger

Décembre 2014

Un programme PHP5 : exemple

```
class Sortie {
    function __construct($titre) { // constructeur
       $this->titre = $titre ; }
    function debut(){
       echo "<html><head>$this->titre</head><body>" ; }
   function fin(){
        echo "</body></html>" ; }
} // fin de la classe Sortie

 // application : exécutable/interprétable…
 $s = new Sortie("Bonjour !") ;
 $s->debut() ;
 echo "<p>sur le serveur, il est ".date("H:i:s")."</p>" ;
 $s->fin() ;
```

Programmation Objet : Classes et Objets

Depuis PHP4 php propose une forme de programmation Objet :

L'idée est d'encapsuler dans une structure de données la classe :
- des variables ou propriétés : les attributs
- les fonctions ou méthodes qui agissent sur ces attributs.

La classe est une déclaration, c'est une évolution des déclarations de types des langages NON OO.

L'instance d'une classe est un objet .

Exemple : Classe : class compteBanque{…}
Objet : monCompte = new compteBanque();

Les Classes : définition

- La syntaxe de PhP est identique à celle de Java ou C++.

class NomClasse [**extends** AutreClasse]**{**

[**private** | **protected** | **public**][**static**] $attribut1
[,$attribut2 …] ;

…

[**private**| **protected** | **public**][**static**] nomFonction(
arguments)
{ … }

…

}

L'ordre de définition des champs et des méthodes est indifférent.

Les classes

- **extends** prise en compte de l'héritage.
- **private** le champs ou la méthode n'est visible qu'à l'intérieur de la classe.
- **protected** le champs ou la méthode n'est visible qu'à l'intérieur de la classe et dans les classes dérivées.
- **public** le champ ou la méthode appartient à l'interface utilisateur de la classe, ce sont les seuls objets visibles pour un utilisateur de la classe. C'est l'accès par défaut

Déclaration d'une Classe (2)

ATTENTION :

- *PAS* de définition d'une classe en plusieurs fichiers.
- *NE PAS* couper la définition d'une classe sur plusieurs blocs, sauf si la coupure apparaît à l'intérieur de la déclaration d'une méthode.

```
<?php // interdit
class test {
?>
…
<?php
   function test() {
   }
}
?>
```

```
<?php //possible
class test {
   function test() {
      ?>
…     <?php
      echo 'Ok';
   }
}
?>
```

Déclaration d'une Classe (3)

Pour accéder aux attributs dans les méthodes, il faut utiliser la variable réservée $this suivi de l'opérateur -> puis du nom de l'attribut.

```
// Exemple : fichier compteBanque.class.php

class compteBanque {
  var $solde=100;
  var $numero;

  function crediter($montant){
     $this->solde += $montant;
  }
  function prelever($montant){
     $this-> solde -= $montant;
  }
}
```

Instancier une classe = créer un objet

Création d'un objet avec le mot clé new.
Exécution des méthodes et accès aux attributs de l'objet par à l'opérateur ->

```
<?php
// instanciation d'une classe
$monCompte = new compteBanque();

// modification d'une propriété
$monCompte->numero="1478963";

// appel d 'une méthode
$monCompte->crediter(5000);
?>
```

Nouvelles Functionalités

- Supporte PPP
- Supporte les Exceptions
- Object Iteration
- Object Cloning
- Interfaces
- Autoload
- Etc…

Les Bases

- Les opérations de base sur les objets n'ont pas changé depuis PHP 4.

```
// on peut toujours écrire :
class my_obj {
        var $foo;
        function my_obj() { // constructeur
            $this->foo = 123;
        }
        function static_method($a) {
            return urlencode($a);
        }
    }
    $a = new my_obj; // instantiation d'objet
     $a->my_obj(); // appel de methode
     my_obj::static_method(123); // appel statique
    de methode
```

Similaire, mais pas identique !!!

- syntaxe identique, sémantique assez différente.

- Les Objets sont en PHP5 toujours passés par référence, pas par valeur.
  - PHP 5 $a = new foo(); == PHP4 $a = &new foo();
- Constructeurs, nouveau mécanisme plus consistant :

__construct()

(ancien style de constructeurs toujours supporté)

__construct()

- parent::__construct()

determine automatiquement quel est le constructeur parent et l'appelle.

```
class main {
        function main() { echo "Main Class\n"; }
}
class child extends main {
        function __construct() {
                parent::__construct();
                echo "Child Class\n";
        }
}
$a = new child;
```

__destruct()

- Methodes 'Destructeur' spécifient le code à exécuter à la "dé-initialisation" d'un objet.

```
class fileio {
        private $fp;
        function __construct ($file) {
                $this->fp = fopen($file, "w");
        }
        function __destruct() {
                fflush($this->fp);
                fclose($this->fp);
        }
}
```

PPP modificateurs d'accès

- ATTENTION; le mot-clé VAR pour identifier les attributs de classe est "deprecated" et provoque un `E_STRICT` warning.

  PHP Strict Standards: var: Deprecated. Please use the public/private/protected modifiers in … on line ...

- PPP : il faut utiliser les mots-clé

  PUBLIC, PRIVATE , PROTECTED.

PPP

Enfin(!), comme dans les autres OO langages, on peut spécifier la visibilité des attributs d'un objet (pour restreindre leurs accès).

- PUBLIC – Accessible par tous de partout.
- PROTECTED – peut être utilisé dans la classe et dans les classes dérivées.
- PRIVATE – utilisation dans la classe seulement.

PPP exemples

```php
<?php
class sample {
   public $a = 1; private $b = 2; protected $c = 3;
   function __construct() {
      echo $this->a . $this->b . $this->c;
   }
}
class miniSample extends sample {
    function __construct() {
       echo $this->a . $this->b . $this->c;
    }
}
$a = new sample(); // 123
$b = new miniSample();
// 13 & notice : undefined property miniSample::$b
echo $a->a . $a->b . $a->c;
// fatal error, access to private/protected property
?>
```

PPP Applications

```
<?php
class registrationData {
 public $Login, $Fname, $Lname, $Address, $Country;
 protected $id, $session_id, $ACL;
}

$a = new registrationData();
foreach ($a as $k => $v) {
    if (isset($_POST[$k])) {$a->$k = $_POST[$k];}
}
?>
```

ATTENTION :

Certaines fonctions/constructeurs PHP

Ne respectent pas les règles de visibilité PPP

Static : méthodes statiques

```
<?php
class my_obj {
        public $foo;

        function __construct() { // constructeur
                $this->foo = 123;
        }
//les méthodes statiques doivent être déclarées ‘static’
//sous peine de E_STRICT warning messages.
        static function static_method($a) {
                return urlencode($a);
        }
}

$a = new my_obj;
my_obj::static_method("a b");
?>
```

Static : attributs statiques

- Les objets peuvent contenir des attributs statiques.

```
<?php
class settings {
        static $login = 'NFA017', $passwd = '123456';
}
echo settings::$login; // NFA017
$a = new settings();
echo $a->login; // undefined property warning
$a->login = "Local Value"; // parse error? (NOPE!)
echo $a->login; // "Local Value"
?>
```

Objets par Référence

```
function foo($obj) { $obj->foo = 1; }

$a = new StdClass; foo($a);
echo $a->foo; //  1

class foo2 {
        function __construct() {
                $GLOBALS['zoom'] = $this;
                $this->a = 1;
        }
}

$a = new foo2();
echo ($a->a == $zoom->a); // 1
```

clonage

- Pour copier un objet : mot-clé clone
- *Ce mot-clé réalise*

  $obj2 = $obj; de PHP 4.

Clone : quelle syntaxe ?

```
class A { public $foo; }

$a = new A;
$a->foo = 1;
$a_copy = clone $a;
$a_copy2 = clone($a);
echo "$a->foo,$a_copy->foo,$a_copy2->foo <br />";
//1,1,1
$a_copy->foo = 2;
echo "$a->foo,$a_copy->foo,$a_copy2->foo <br />";
//1,2,1
$a_copy2->foo = 3;
echo "$a->foo,$a_copy->foo,$a_copy2->foo <br />";
//1,2,3
```

Etendre Clone

- __clone() pour modifier la copie en cours.

```
class A {
        public $is_copy = FALSE;

        public function __clone() {
                $this->is_copy = TRUE;
        }
}
$a = new A;
$b = clone $a;
var_dump($a->is_copy, $b->is_copy); // false, true
```

Classes Constantes : 'const'

- En PHP 5 constantes très similaires aux attributs "static", mais leur valeur ne peut être modifiée.

```
class cc {
        const value = 'abc 123';
        function print_constant() {
// accès à une constante de classe depuis la classe
            echo self::value;
      }
}
echo cc::value;
// accès à une constante de classe de l'extérieur de
   la classe i.e. comme static
```

PPP s'applique aussi aux méthodes!

- L'accès aux Méthodes peut aussi être restreint par PPP :
  - Cacher, interdire l'accès aux fonctionnalités internes d'une application.
  - séparation des données.
  - Accroitre la sécurité.
  - Code plus "propre".

Exemple PPP Methods

```
class mysql {
   private $login, $pass, $host;
   protected $resource, $error, $qp;
   private function __construct() {
      $this->resource = mysql_connect($this->host,
                        $this->login, $this->pass);
   }
   protected function exec_query($qry) {
    if (!($this->qp = mysql_query($qry,
                      $this->resource))){
     self::sqlError(mysql_error($this->resource));
    }
   }
   private static function sqlError($str) {
         open_log(); write_to_error_log($str);
         close_log();
   }
}
```

Exemple : PPP Methods

```
class database extends mysql {
   function __construct() {
         parent::__construct();
   }
   function insert($qry) {
         $this->exec_query($qry);
         return mysql_insert_id($this->resource);
   }
    function update($qry) {
         $this->exec_query($qry);
      return mysql_affected_rows($this->resource);
   }
}
```

Final

- définition de classes et de méthodes 'finales' : FINAL.
  - Pour les méthodes cela signifie quelles ne peuvent pas être masquées dans une classe dérivée.

```
class main {
        function foo() {}
        final private function bar() {}
}

class child extends main {
        public function bar() {}
}

$a = new child();  // erreur
```

Final Class

- Une Classe déclarée "final" ne peut être étendue.

```
final class main {
        function foo() {}
        function bar() {}
}
class child extends main { }
$a = new child();
```

```
PHP Fatal error:
  Class child may not inherit from final class
  (main)
```

__autoload()

- *Maintaining class decencies in PHP 5 becomes trivial thanks to the __autoload() function !!!*

```
<?php
function __autoload($class_name) {
        require_once "/php/classes/{$class_name}.inc.php";
}
$a = new Class1;
?>
```

- Si définie, cette fonction est utilisée pour charger automatiquement toute classe NON encore présente.

3 méthodes “Magiques”

- Les Objets en PHP 5 peuvent avoir 3 'magic-methods'.
  - __sleep() –cf. PHP4.
  - __wakeup() –cf. PHP4.
  - __toString() – PHP5.

Sérialisation en PHP5

- Sérialisation : processus de transformation de variables (objets) en une chaine codée qui permettra de recréer ces variables à la désérialisation.
- Utilisée notamment pour les objets complexes ou les tableaux qui ne peuvent pas être écrits de manière simple dans un fichier ou en BD.

- sérialisation : serialize()
- restauration : unserialize()

Exemple sérialisation

```
class test {
        public $foo = 1, $bar, $baz;
        function __construct() {
                $this->bar = $this->foo * 10;
                $this->baz = ($this->bar + 3) / 2;
        }
}
$a = serialize(new test()); // encode
$b = unserialize($a); // restore l'objet dans $b;
```

- Chaîne codée obtenue :

O:4:"test":3:{s:3:"foo";i:1;s:3:"bar";i:10;s:3:"baz";d:6.5;}

__sleep()

- **__sleep()** permet de préciser les attributs à sérialiser :

```
class test {
        public $foo = 1, $bar, $baz;
        function __construct() {
                $this->bar = $this->foo * 10;
                $this->baz = ($this->bar + 3) / 2;
        }
        function __sleep() { return array('foo'); }
}
```

- This makes our serialized data more manageable !!!

```
O:4:"test":1:{s:3:"foo";i:1;}
```

__sleep() (2)

- __*sleep* La fonction serialize() vérifie si la classe a une fonction avec le nom magique __*sleep*. Si c'est le cas, elle l'exécute avant toute linéarisation. Elle peut nettoyer l'objet et elle retourne un tableau avec les noms de toutes les variables de l'objet qui doivent être linéarisées.

- Le but de __*sleep* est aussi de fermer toutes les connexions aux bases de données que l'objet peut avoir ouverte, de valider les données en attente ou d'effectuer des tâches de nettoyage. Cette fonction est utile pour les très gros objets qui n'ont pas besoin d'être sauvegardés en totalité.

__wakeup()

- __wakeup(), si définie, est appelée après dé-sérialisation pour recréer les attributs qui n'ont pas été sérialisés.

```
class test {
        public $foo = 1, $bar, $baz;
        function __construct() {
            $this->bar = $this->foo * 10;
            $this->baz = ($this->bar + 3) / 2;
        }
        function __wakeup() {
             self::__construct();
        }
}
```

__wakeup()

- La fonction unserialize() vérifie la présence d'une fonction de nom 'magique' *__wakeup*. Si elle est présente, cette fonction permet de reconstruire toutes les ressources que l'objet possède.

- Le but de *__wakeup* est aussi de rétablir toute connexion base de données qui aurait été perdue durant la linéarisation et d'effectuer les tâches de réinitialisation.

__toString()

```
<?php
class sample {
        public $foo;

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

        function __toString() {
                return (string)$this->foo;
        }
}
echo new Sample();
?>
```

__toString() quand ?

soit $a = new obj();

- echo "str" . $a;
- echo "str {$a}";
- echo $a{0};
- echo (string) $a;

Dans tous ces cas __toString() ne sera pas appelé

Surcharge

- Aussi bien les appels de méthodes que les accès aux attributs peuvent être surchargés via

  __call() , __get() , __set()

- Cela fournit un mécanisme d'accès à des méthodes et à des attributs "virtuels"

Getter

- __get() permet un accès en lecture à des attributs virtuels.

```
class makePassword {
        function __get($name) {
                if ($name == 'md5')
                        return substr(md5(rand()), 0, 8);
                else if ($name == 'sha1')
                        return substr(sha1(rand()), 0, 8);
                else
                        exit("Invalid Property Name");
                }
}
$a = new makePassword();
var_dump($a->md5, $a->sha1);
```

Setter

- __set() allows **write** access to virtual object properties.

```
<?php
class userUpdate {
        public $user_id;

        function __construct() { db_cect() }

        function __set($name, $value) {
                db_update("UPDATE users SET {$name}='{$value}'
                                        WHERE id={$user_id}");
        }
}
?>
```

Méthodes dynamiques

- The __call() method in a class can be used to emulate any non-declared methods.

```
class math {
        function __call($name, $arg) {
                if (count($arg) > 2) return FALSE;
                switch ($name) {
                        case 'add': return $arg[0] + $arg[1]; break;
                        case 'sub': return $arg[0] - $arg[1]; break;
                        case 'div':  return $arg[0] / $arg[1]; break;
                }
        }
}
```

Important Overloading Reminders

- The name passed to __get(), __set(), __call() is not case normalized.

  $foo->bar != $foo->BAR

- Will only be called if the method/property does not exist inside the object.
- Functions used to retrieve object properties, methods will not work.
- Use with caution, it takes no effort at all to make code terribly confusing and impossible to debug.

Interfaces

- Spécification (syntaxique) des fonctionnalités que les classes qui implantent cette interface doivent respecter.

- ATTENTION, on ne spécifie que ce qui est public. Pas de variable.

Interface Exemples

- Les Interfaces spécifient la syntaxe des api standard.

```
interface webSafe {
        public function encode($str);
        public function decode($str);
}
interface sqlSafe {
        public function textEncode($str);
        public function binaryEncode($str);
}
```

Implémenter des interfaces

- une classe peut implémenter plusieurs interfaces.

```
class safety Implements webSafe, sqlSafe {
        public function encode($str) {
                return htmlentities($str);
        }
        public function decode($str) {
                return html_entity_decode($str);
        }
        public function textEncode($str) {
                return pg_escape_string($str);
        }
        public function binaryEncode($str) {
                return pg_escape_bytea($str);
        }
}
```

ArrayAccess Interface

- L'interface ArrayAccess, permet aux objets d'émuler un tableau
- L'interface requiert les méthodes suivantes :
  - offsetExists($key) - détermine si une valeur existe à la clé $key
  - offsetGet($key) – retourne la valeur à la clé $key
  - offsetSet($key, $value) - assigne value à la clé $key
  - offsetUnset($key) – detruit la valeur à la clé $key

ArrayAccess exemple

```
class changePassword implements ArrayAccess {
 function offsetExists($id) {
    return $this->db_conn->isValidUserID($id);}
 function offsetGet($id) {
    return $this->db_conn->getRawPasswd($id);}
 function offsetSet($id, $passwd) {
   $this->db_conn->setPasswd($id, $passwd);}
 function offsetUnset($id) {
  $this->db_conn->resetPasswd($id);}
}
$pwd = new changePassword;
isset($pwd[123]); // user with an id 123 ?
echo $pwd[123]; // print the user's password
$pwd[123] = "pass"; // change password to "pass"
unset($pwd[123]); // reset user's password
```

Classes abstraites

- "plus que interface" : peuvent contenir des variables et des méthodes…
- Mais, ne sont pas instantiables.
- Idée : déléguer aux sous classes l'implantation de certaines méthodes

Exemple : classe abstraite database

- Les méthodes précédées de abstract doivent être implementées par les sous classes

```
abstract class database {
        public $errStr = '', $errNo = 0;

        abstract protected function init($login,$pass,$host,$db);
        abstract protected function execQuery($qry);
        abstract protected function fetchRow($qryResource);
        abstract protected function disconnect();
        abstract protected function errorCode();
        abstract protected function errorNo();
}
```

Exemple (2)

```
class mysql extends database {
        private $c;
        protected function init($login, $pass, $host, $db) {
                $this->c = mysql_connect($host, $login, $pass);
                mysql_select_db($db, $this->c);
        }
        protected function execQuery($qry) {
                return mysql_query($qry, $this->c);
        }
        protected function fetchRow($res) {
                return mysql_fetch_assoc($res);
        }
  protected function errorCode() {return mysql_error($this->c); }
  protected function errorNo() { return mysql_errno($this->c); }
  protected function disconnect() { mysql_close($this->c); }
}

class pgsql extends database {
   protected function init($login, $pass, $host, $db) {… }
   …
}
```

Iteration : interface iterator

- les classes peuvent implémenter l'interface iterator qui spécifie la manière standard de parcourir un objet de la classe de manière itérative.

- La classe qui implémente l'interface iterator doit alors implanter les méthodes :
  - Rewind()
  - current()
  - key()
  - next()
  - Valid()

Exemple : Iterator (1)

```
class fileI Implements Iterator {
        private $fp, $line = NULL, $pos = 0;

        function __construct($path) {
                $this->fp = fopen($path, "r");
        }

        public function rewind() {
                rewind($this->fp);
        }

        public function current() {
                if ($this->line === NULL) {
                        $this->line = fgets($this->fp);
                }
                return $this->line;
        }
```

exemple Iterator (2)

```
        public function key() {
                if ($this->line === NULL) {
                        $this->line = fgets($this->fp);
                }
                        if ($this->line === FALSE) return FALSE;
                return $this->pos;
        }

        public function next() {
                $this->line = fgets($this->fp);
                ++$this->pos;
                return $this->line;
        }

        public function valid() {
                return ($this->line !== FALSE);
        }
}
```

Exemple Iterator (3)

```
<?php
function __autoload($class_name) {
        require "./{$class_name}.php";
}
foreach (new fileI(__FILE__) as $k => $v) {
    echo "{$k} {$v}";
}
?>
```

Exceptions pour unifier le traitement des erreurs

- Le code du programme (comportement 'normal') est mis dans un bloc :

try {}

- Toutes les erreurs sont envoyées pour traitement dans le bloc

catch () {} correspondant…

PHP5 : classe Exception

```
class Exception
{
  protected $message = 'Unknown exception'; // exception message
  protected $code = 0; // user defined exception code
  protected $file; // source filename of exception
  protected $line; // source line of exception

  function __construct($message = null, $code = 0);

  final function getMessage(); // message of exception
  final function getCode(); // code of exception
  final function getFile(); // source filename
  final function getLine(); // source line
  final function getTrace(); // backtrace array
  final function getTraceAsString(); // trace as a string

  function __toString(); // formatted string for display
}
```

Exception : exemple

```
<?php
                    try {

 $fp = fopen("m:/file", "w");
 if (!$fp) throw new Exception("Cannot open file.");
 if (fwrite($fp, "abc") != 3)
                    throw new Exception("Failed to write data.");
 if (!fclose($fp)) throw new Exception("Cannot close file.");

                    } catch (Exception $e) {
        printf("Error on %s:%d %s\n",
          $e->getFile(), $e->getLine(), $e->getMessage());
        exit;
}
?>
```

Écrire ses propres Exceptions

```
class nfa017Exception extends Exception {

        public function __construct() {
            parent::__construct($GLOBALS['php_errormsg']);
        }

        public function __toString() {
                return sprintf("Error on [%s:%d]: %s\n",
                        $this->file, $this->line, $this->message);
        }
}
```

Utiliser ses propres Exceptions

```
//---------utilisation de nfa017Exception-------------------
ini_set("track_errors", 1); error_reporting(0);
try {
        $fp = fopen("m:/file", "w");
        if (!$fp) throw new nfa017Exception ;
        if (fwrite($fp, "abc") != 3) throw new nfa017Exception ;
        if (!fclose($fp)) throw new nfa017Exception ;
} catch (nfa017Exception $e) { echo $e; }
```

empilage ou nommage des Exceptions ?

```php
<?php
// EMPILAGE
try {
  // Exception $try1
  try {
    // Exception $try2
    } catch (Exception $try2) {

    }
   // Exception $try1
} catch (Exception $try1) {

}
?>
```

```php
<?php
// NOMMAGE
try {
        $a = new dbConnection();
        $a->execQuery();
        $a->fetchData();
} catch (ConnectException $db){

} catch (QueryException $qry) {

} catch (fetchException $dt) {
}
?>
```

Exception Handler

- La fonction exception handler, set_exception_handler() permet de gérer des exception sans try {} catch (){} explicite :

```
function exHndl($e) { trigger_error($e->getLine());}

set_exception_handler('exHndl');

$fp = fopen("m:/file", "w");
if (!$fp) throw new nfa017Exception ;
if (fwrite($fp, "abc") != 3) throw new nfa017Exception;
if (!fclose($fp)) throw new nfa017Exception;
```

'Proposition' de Type

- Alors que PHP est toujours "insensible" aux types on peut maintenant spécifier le type (à respecter) des paramètres d'une méthode.

```
<?php
class Foo {}
function useFoo(Foo $obj) { /* ... */ }

$a = new Foo;
useFoo($a); // OK…

$b = new StdClass;
useFoo($b);
// Fatal error: Argument 1 must be an instance of Foo
?>
```

Constantes magiques PHP5 (1)

__LINE__ : retourne le n° de la ligne courante dans le fichier.

```
<?php // un fichier exemple
echo ">Voici la ligne N°:".__LINE__;
// affichera 2
// si ce programme est le contenu complet du fichier
?>
```

Cette constante (variable pourtant...) très utile lors du debuggage des applications.

Le numéro de ligne correspond bien au numéro de ligne dans le fichier d'origine. Dans l'exemple ci-dessus, en admettant que le fichier soit inclus dans un autre à la ligne 10, le résultat restera le même, c'est à dire 2.

Constantes magiques PHP5 (2)

__FILE__ : retourne le chemin complet et le nom du fichier courant.

Rappel, les chemins d'accès sous Windows sont avec des "\" alors que sous Linux c'est "/". A savoir pour réaliser des applications compatibles pouvant s'installer indépendamment de la plateforme, ou , par exemple développer chez soi sous Windows et proter sur un serveur de production sous Linux…
Depuis Php 4.0.6 la constante ***DIRECTORY_SEPARATOR*** *indique le séparateur utilisé. Donc à l'utiliser absolument.*

Constantes magiques PHP5 (3)

__FUNCTION__ : retourne le nom de la fonction (depuis Php 4.3.0)

```
<?
function test1() {   echo "> : ".__FUNCTION__;}
test1();
?>
```

Cet exemple affichera *> : test1*.

ATTENTION : Afficher le contenu de cette constante sans être dans une fonction, ne retourne rien.

Constantes magiques PHP5 (4)

__CLASS__ : retourne le nom de la classe courante (depuis Php 4.3.0)

```
<?
class MaClasse {
        var $dummy;
        function test() {
                echo "> : ".__CLASS__;
        }
}
$a = new MaClasse();
$a->test();
?>
```

Cet exemple affichera *> : MaClasse*.

Constantes magiques PHP5 (5)

__METHOD__ : retourne le nom de la méthode courante
(depuis Php 5.0.0)

```
<?
class MaClasse {
        var $dummy;
        function test() {
                echo "> : ".__ METHOD__;
        }
}
$a = new MaClasse();
$a->test();
?>
```

Cet exemple affichera *> : MaClasse ::test.*

Constantes magiques PHP5 (6)

%%%%%%ATTENTION --- ATTENTION :%%%%%%

```
<?
echo "> : __FILE__";
?>
```

Affichera > : *__FILE__*

En effet, __FILE__ n'est pas interprété, donc il faut concaténer les chaînes pour obtenir le résultat attendu :

```
<?
echo "> : ".__FILE__;
?>
```

l'API 'Reflection'

- L'API 'Reflection' propose un mécanisme d'introspection pour obtenir des informations détatillées sur les fonctions, méthodes, classes et les exceptions.

- It also offers ways of retrieving doc comments for functions, classes and methods.

Reflection Mechanisms

- L'API fournit des classes distinctes pour l'étude des différentes entitées analysables.

```
<?php
interface Reflector { }
class Reflection { }
class ReflectionException extends Exception { }
class ReflectionFunction implements Reflector { }
class ReflectionParameter implements Reflector { }
class ReflectionMethod extends ReflectionFunction { }
class ReflectionClass implements Reflector { }
class ReflectionObject extends ReflectionClass { }
class ReflectionProperty implements Reflector { }
class ReflectionExtension implements Reflector { }
?>
```