Initiation à la programmation orientée objet

ESECAD
GROUPE SKILL&YOU

Sommaire

Introduction 3
I. Les fonctions 4
A. Rappel du fonctionnement des fonctions 4
B. Utilisation de *true* et *false* 5
C. Opérateur de comparaison 6
II. Découverte de la programmation orientée objet 7
A. Objet, classe et méthodes 7
B. Une classe possède des méthodes 8
C. Instancier une classe 9
D. La pseudo-variable *$this*, une référence à la classe 11
E. Les classes possèdent des propriétés 11
F. Présentation du constructeur 12
G. Les constantes de classe 15
H. Fonctions utiles au développement : *get_object_vars*, *get_class_vars*, *get_class_methods* 15

Les repères de lecture


Retour au chapitre


Définition


Objectif(s)


Espace Élèves


Vidéo / Audio

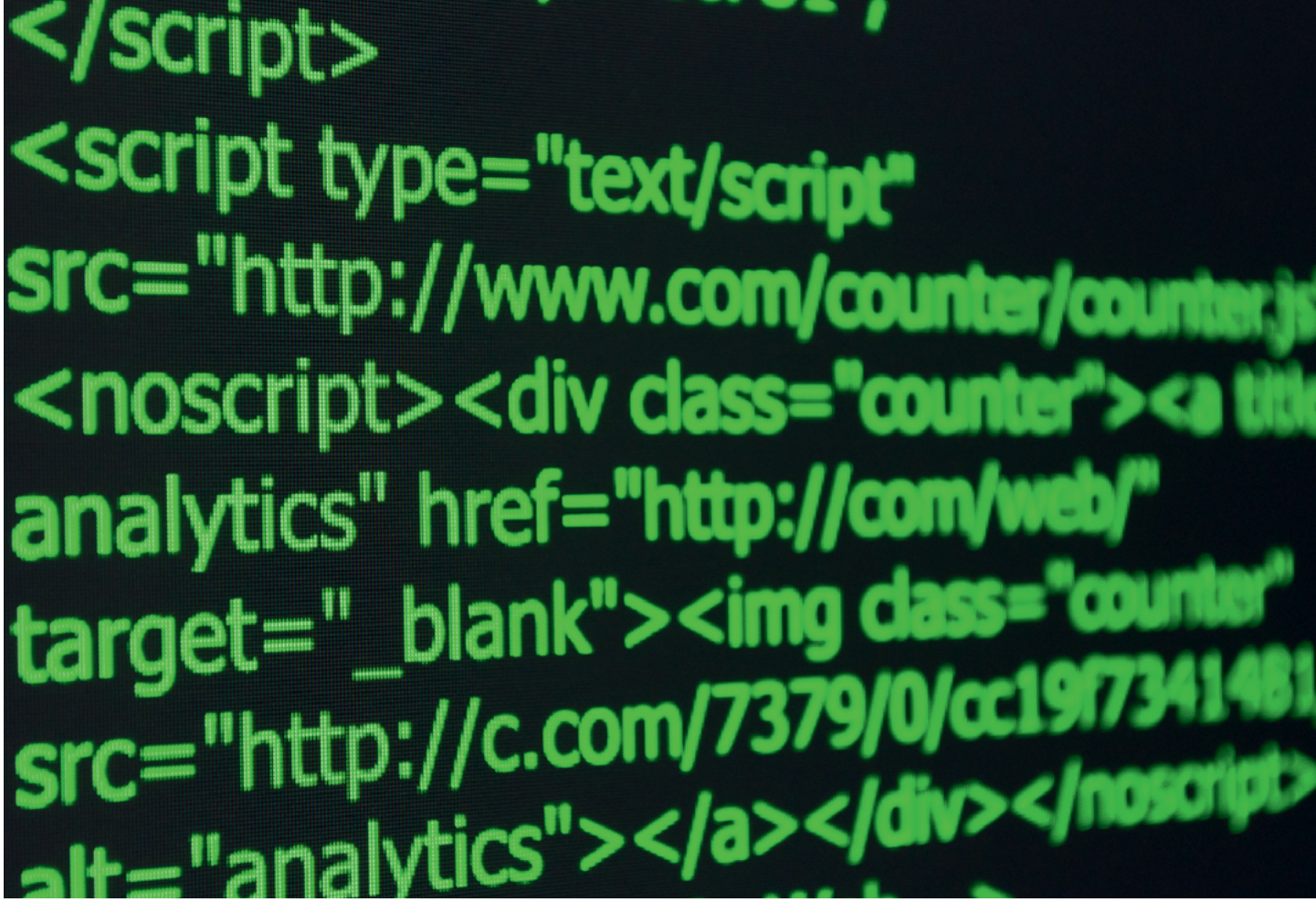

Point important / À retenir


Remarque(s)


Pour aller plus loin


Normes et lois


Quiz



Introduction

Nous avons abordé la notion de fonctions dans le livret précédent comme un moyen utile de gérer du code sans avoir à réécrire des scripts au cours d'un développement. Cette première utilisation était limitée par la nécessité de découvrir d'autres aspects fondamentaux de PHP, comme les tableaux de données. À ce stade de notre apprentissage de PHP, compte tenu de nos connaissances acquises lors de nos premiers développements nous pouvons maintenant aller plus loin et aborder la programmation orientée objet.

I. Les fonctions

A. Rappel du fonctionnement des fonctions

Si nous souhaitons dans un projet utiliser à plusieurs reprises certaines informations enregistrées dans une base de données, il est recommandé de créer une fonction qui contiendra la requête vers la base de données. Ainsi, il nous suffira d'appeler cette fonction à chaque fois que nous en aurons besoin.

***Exemple** : la fonction recupereVilles() destinée à retourner la liste des villes enregistrées dans une table.*

```
<?php
function recupereVilles ()
{
	$mysqli = new mysqli('localhost', 'root', '', 'projet_
villes');
	$result = $mysqli->query('SELECT ville_id
 ville_nom FROM villes');
	while ($row = $result->fetch_array())
	{
		$villes_data[] = $row['ville_nom'];
	}
	return $villes_data;
}
?>
```

Fig. 1

Pour utiliser le résultat fourni par la fonction, il suffit d'appeler cette fonction et de l'affecter à une variable afin de pouvoir en exploiter le contenu :

```
<p>Liste des villes :
<?php $villes = recupereVilles() ?>
<?php foreach ($villes as $ville) :?>
<li><?php echo $ville ?></li>
<?php endforeach?>
```

Fig. 2

En plaçant cette fonction dans un fichier d'inclusion **fonction.php**, nous y aurons accès en permanence et pourrons donc récupérer la liste des villes rapidement et simplement. Nous n'aurons, en effet, pas besoin de réécrire à chaque fois ce code puisque nous pourrons appeler la fonction *recupereVilles()* quel que soit l'endroit du site où nous souhaitons utiliser ce résultat.

Nous pouvons donc créer autant de fonctions que nous le souhaitons selon nos besoins et les inclure pour les réutiliser à volonté. La structure classique d'une application PHP pourrait ainsi être la suivante, par exemple pour le fichier **index.php** :

```
require('fonction.php'); // contient les fonctions
require('top.php'); // en tete html
require('contenu.php'); // affichage du contenu fourni par les
fonctions
require('footer.php'); // pied de page html
```

Fig.3

Le fichier **fonction.php** pourrait contenir par exemple les deux fonctions *recupereVilles()* et *recupereVille($ville_id),* la fonction *recupereVilles* (au pluriel) récupérant toutes les villes et la fonction *recupereVilles* (au singulier) retournant une seule ville (celle dont l'identifiant *$ville_id* est passée en argument).

Notons que si la requête ne récupère aucun résultat, la fonction ne retournera rien. Il est donc nécessaire de vérifier le résultat retourné avant d'utiliser la fonction. Si la valeur fournie pour l'identifiant *ville_id* n'existe pas dans la base (par exemple 8) et que nous cherchons à l'afficher, le message d'erreur sera *warning variables not exist*.

Nous devons vérifier avec un *isset()* que la variable à laquelle nous affectons le résultat de la fonction existe bien.

Exemple

```
$ville = recupereVille(8);
if(isset($ville))
{
        // code
}
```

Fig.4

B. Utilisation de *true* et *false*

Il est également possible d'utiliser le *isset()* au sein de la fonction et de gérer dans l'affichage avec les booléens *TRUE* et *FALSE* (qui peuvent également s'écrire en minuscule *true* et *false*).

Un **booléen** est un type de variable qui n'est pas une chaîne de caractères ou un tableau ou un nombre. Il s'agit d'un type de variable pouvant posséder une valeur de type logique qui ne peut valoir que **vrai ou faux**, *TRUE* signifiant VRAI et *FALSE* signifiant FAUX.

Il est donc très pratique de les utiliser pour déterminer l'échec ou le succès d'une fonction. La façon de gérer les booléens est exposée dans les cas suivants.

Pour un rappel des opérateurs de comparaison, il est recommandé de consulter la page dédiée du manuel PHP : http://www.php.net/manual/fr/language.operators.comparison.php

Affectation à *$foo* de la valeur *TRUE*

```
$foo = TRUE;
```

Fig.5

Affectation à *$foo* de la valeur *FALSE*

```
$foo = FALSE;
```

Fig. 6

Vérification que *$foo* soit *FALSE*

C. Opérateur de comparaison

L'opérateur de comparaison === permet de vérifier que le type de la variable est également un booléen comme *false*.

```
if($foo === false)
```

Fig. 7

Pour que l'opérateur de comparaison soit effectif, le signal « égal » doit être répété 3 fois.

Vérifier que *$foo* est TRUE avec l'opérateur ===.

```
if($foo === true)
```

Fig. 8

Vérifier que *$foo* n'est pas FALSE. L'opérateur !== est le contraire de ===.

```
if ($foo!== false)
```

Fig. 9

Vérifier que *$foo* n'est pas TRUE avec l'opérateur !==.

```
if ($foo!== true)
```

Fig. 10

L'utilisation des fonctions est donc intéressante car les fonctions créées dans le cadre d'un projet peuvent, par exemple, être **réutilisées** dans d'autres projets, soit directement, soit en les adaptant.

Cependant le processus d'industrialisation qui est en cours dans le secteur du développement informatique pose des besoins encore accrus de « **réutilisabilité du code** » ainsi que la nécessité de travailler en équipe sur une application, chaque développeur ayant en charge une partie du développement. Ce qui implique donc que les différents travaux peuvent être développés et de manière autonome, puis assemblés lorsque les projets se terminent, la problématique de faire communiquer les différents codes entre eux devenant alors vitale.

Une bonne réponse à ce besoin est contenue dans la **programmation orientée objet** qui permet justement de définir un code très structuré définissant des règles d'utilisation communes.

La programmation orientée objet, abrégée en **POO**, offre en effet les moyens de travailler, comme son nom l'indique, avec des objets, chaque objet possédant des caractéristiques bien définies.

La programmation orientée objet, basée sur l'usage des **classes**, est une nouvelle manière de développer qui se distingue nettement du code dit procédural que jusqu'à présent nous utilisions.

Notons que PHP propose une véritable programmation orientée objet depuis la version 5 (PHP5).

II. Découverte de la programmation orientée objet

A. Objet, classe et méthodes

1. Un objet est l'instance d'une classe

L'utilisation des objets en développement est fondée sur une logique simple que nous rencontrons dans la vie de tous les jours. Une voiture, un vêtement, un vélo, une table, un verre sont des objets que nous utilisons au quotidien. Une chaise, un avion, un ordinateur, une carafe sont également des objets, plus ou moins complexes. Nous sommes entourés d'objets.

Ces objets possèdent tous des propriétés, par exemple une couleur, un poids, une taille, qui permettent de les identifier et de les utiliser. Tous ces objets sont construits sur la base d'un **modèle initial** qui permet de les reproduire à l'identique. Un objet tel qu'un vélo est construit sur un modèle de vélo prédéfini et possédera les mêmes caractéristiques (ou propriétés) qu'un autre vélo construit sur ce même modèle.

Dans le contexte de la programmation, nous rencontrerons le même principe : un modèle, que nous appelons une **classe**, permet de construire des objets.

Une tasse pourra ainsi être considérée comme un objet au sens développement. Il suffira de créer la classe *Tasse* et de lui affecter des propriétés, comme : une contenance, un poids, une couleur, etc.

Comme nous le voyons, un objet au sens programmation peut être n'importe quel objet courant, qu'il soit matériel ou immatériel. Un panier de site e-commerce pourra être un objet, un compte bancaire pourra être un objet, etc.

Exemple
Une ville, qui naturellement pourra tout à fait être considérée comme un objet. Pour ce faire, nous créons la classe Ville pour laquelle nous définissons les propriétés suivantes : le nom, le nombre d'habitants et le pays.

La classe Ville étant créée, nous pouvons ensuite créer des objets villes de façon illimitée selon les propriétés de la classe Ville :

Tableau n°1

EXEMPLE D'OBJET VILLE :	UN AUTRE OBJET VILLE :	ENCORE UN AUTRE OBJET VILLE :
– nom : Paris ; – nombre d'habitants : 2,2 M ; – pays : France.	– nom : Rome ; – nombre d'habitants : 2,6 M ; – pays : Italie.	– nom : Berlin ; – nombre d'habitants : 3,5 M ; – pays : Allemagne.

Lorsque nous créons l'objet, l'objet concrétise la classe, nous disons qu'il est une instance de la classe (ou qu'il instancie la classe).

L'objet Paris est une instance de la classe Ville, une Twingo est une instance de la classe Voiture, le compte bancaire de Paul est une instance de la classe Compte bancaire, etc.

Ce principe et ce vocabulaire étant posés, penchons-nous maintenant sur la syntaxe de la **POO**.

En toute logique, la création d'une classe s'effectue avec le mot-clef *class* suivi du nom de la classe, que nous écrivons toujours avec une majuscule sur la première lettre et des accolades.

Exemple :
Syntaxe pour créer la classe Ville :

```
class Ville
{
}
```

Fig. 11

B. Une classe possède des méthodes

Instancier : initialiser en programmation, à partir d'un espace mémoire réservé, un objet à partir d'un ensemble de caractéristiques, appelé « classe ». Déclarer un nouvel objet.

Les classes possèdent leurs propres fonctions qui sont appelées des **méthodes**. L'intérêt est que, en instanciant une classe, l'objet peut appeler les méthodes de cette classe.

Pour déclarer une méthode, la syntaxe est la suivante : il faut indiquer un des mots-clefs *public*, *protected*, ou *private*, puis écrire *function* et enfin poser le nom de la méthode (avec les parenthèses suivies des accolades).

```
Class Ville // déclaration de la classe
{
	function recupereVille ($ville_id) // déclaration de la
méthode
	{
		//code de la méthode
	}

	function recupereVilles() // déclaration de la méthode
	{
		//code de la méthode
	}
}
```

Fig. 12

Ayant créé la classe *Ville*, et déclaré deux méthodes, nous pouvons donc créer une instance de cette classe, c'est-à-dire créer un objet *Ville*, avec la syntaxe appropriée.

C. Instancier une classe

Pour instancier la classe, nous utilisons le mot-clef *new* **suivi du nom de la classe**, et nous affectons cet objet à une variable pour laquelle il est recommandé d'utiliser un nommage en adéquation avec le nom de la classe, par exemple :

```
$ville = new Ville ; // instanciation de la classe Ville
```

Fig. 13

La variable *$ville* est donc un objet et lorsque nous créons cet objet, nous pouvons immédiatement accéder aux méthodes et aux propriétés de la classe.

Pour y accéder nous utilisons **la flèche -> suivie du nom de la méthode** comme le montre la syntaxe suivante et qui se lit :

```
$ville->recupereVille(2);
```

Fig. 14

Le résultat sera identique au code utilisé lors de l'appel à la **fonction** *recupereVille* effectuée précédemment dans le rappel des fonctions, car les méthodes sont des fonctions et nous appelons ici la même fonction qui est cette fois intégrée dans la classe *Ville*.

Nous pouvons donc ensuite continuer notre script et afficher les résultats retournés par la méthode comme s'il s'agissait d'une fonction standard.

Voici le code complet :

```
<?php
class Ville // déclaration de la classe
{
      function recupereVilles () // déclaration de la méthode
      {
          $mysqli = new mysqli('localhost','root','','nom_de_la_
base');
          $result = $mysqli->query('SELECT ville_id,
ville_nom FROM villes');
          $row = $result->fetch_array();
          return $row ;
      }

      function recupereVille ($ville_id) // déclaration de la
méthode
      {
            $mysqli = new mysqli('localhost','root','','nom_de_la_
base');
            $result = $mysqli->query('SELECT ville_id,
 ville_nom  FROM villes WHERE ville_id = '. $ville_id);
            $row = $result->fetch_array();
            return $row ;
      }
}
$ville = new Ville; // instanciation de la classe Ville
$ville_data = $ville->recupereVille(2); // affectation du résultat
de la méthode afficheVille lorsque l'argument est 2
?>
<p>le nom de la ville est : <?php echo $ville_data['nom']?> </p>
<p>le nombre d'habitants est : <?php echo $ville_
data['habitants']?> </p>
<p>le pays est : <?php echo $ville_data['pays']?> </p>
```

Fig. 15

La différence avec un appel de fonction dans un code de type procédural tient à l'instanciation de la classe *Ville* avec le mot-clef *new* qui permet d'accéder aux méthodes de la classe avec l'opérateur flèche.

Afin de rationaliser au mieux le code, nous l'organisons avec des inclusions (*require*) de fichiers dédiés aux classes, chaque fichier contenant une seule classe. Ainsi nous créons un fichier **Ville.php** qui contiendra la classe *Ville*, le nom du fichier reprenant le nom de la classe en gardant la majuscule.

Ainsi nous avons un fichier pour la classe et un fichier pour afficher le résultat dans lequel nous incluons le fichier contenant la classe, que nous pouvons ranger dans un répertoire intitulé *class*.

La structure des fichiers est ainsi la suivante :
- **root/class/Ville.php**
- **root/page.php**

Le code devenant donc :

```
require('class/Ville.php');
$ville = new Ville; // instanciation de la classe Ville
//etc..
```

Fig. 16

Concernant le nommage des classes et des méthodes, il est recommandé d'utiliser des termes simples et en anglais comme nous l'avons vu dans les livrets précédents pour les variables, bien que cela ne soit pas une obligation. Dans notre exemple, la classe *Ville* deviendrait simplement la classe *City*.

Pour les méthodes, il est en outre vivement recommandé que leur nom soit le plus explicite possible afin que nous puissions identifier leur rôle le plus rapidement possible. Dans notre exemple, le nom des méthodes *afficheVilles* et *afficheVille* définissent rapidement à quoi elles servent. Si nous optons pour l'anglais, les noms de ces méthodes peuvent être traduits en *getCities* (*get* : obtenir, récupérer) et *getCity*.

Les termes anglais que nous aurons à utiliser seront très souvent les mêmes compte tenu du fait que les méthodes sont fréquemment créées pour effectuer des actions standards telles que : *get* (obtenir), *insert* (insérer, ajouter), *set* (déterminer), *update* (mettre à jour), *delete* ou *remove* (supprimer).

Ainsi si nous appelons *getCities* la méthode restituant la liste des villes et *getCity* la méthode restituant une ville, la méthode qui permettrait de supprimer une ville de la base de données serait *removeCity*, celle permettant d'ajouter une nouvelle ville dans la base serait *setCity* et la méthode dédiée à la mise à jour *updateCity*.

Ainsi comme nous le voyons, nous pouvons créer tout un arsenal de méthodes pratiques et rangées dans une classe pour un objet donné, ce qui permet d'organiser sérieusement un développement, le nombre de méthodes n'étant pas limité.

D. La pseudo-variable *$this*, une référence à la classe

L'appel d'une méthode d'une classe **dans une autre méthode de la même classe** ne s'effectue pas en instanciant la classe avec le mot-clef *new* mais avec la pseudo-variable *$this* suivie de la flèche, de la même manière que pour les attributs. La pseudo-variable *$this* est une référence à la classe dans laquelle elle est utilisée, ce qui permet donc d'accéder aux propriétés et aux méthodes de la classe.

Par exemple, si dans une méthode *method1()* nous souhaitons appeler une méthode *method2()*, sachant que l'ordre des méthodes n'a pas d'importance, la syntaxe serait simplement :

```
function method1()
{
$foo = $this->method2() ;
}
function method2()
{
// code
}
```

Fig. 17

E. Les classes possèdent des propriétés

1. Les propriétés des classes sont des variables

Les attributs ou propriétés des classes sont des variables spécifiques à la classe. Par exemple, un objet vêtement (*clothing* en anglais) peut posséder un attribut couleur, un attribut taille, un attribut prix également si nous évoluons dans un contexte marchand.

Nous pourrions effectuer comme ceci, ce besoin : créer la classe *Clothing* (vêtement) et associer une méthode permettant de récupérer la couleur « rouge » que nous souhaitons définir pour un objet.

```
<?php
class Clothing
{
function getColor()
{
$color = 'rouge';
return $color ;
}
}
$clothing = new Clothing;
var_dump ($clothing); // object(Clothing)[1]
```

Fig. 18

Pour récupérer la couleur rouge nous utilisons l'opérateur flèche **->**.

```
echo $clothing->getColor();// affiche « rouge »
```

Fig. 19

Le modèle objet de PHP permet une méthode plus efficace pour définir cette couleur avec les propriétés. Au lieu de créer une méthode, nous allons donc définir une propriété, ou attribut qui est donc une variable spécifique à la classe. Nous pouvons déclarer directement ces propriétés dont le nombre n'est pas limité. Pour les mettre en œuvre, la syntaxe est simple. Il suffit de les déclarer entre les accolades de la classe, précédées d'un mot-clef, qui peut être soit **public**, soit **protected**, soit **private** :

```
class Clothing
{
public $color;
public $size ;
public $price ;
}
```

Fig. 20

Les mots-clefs *public* pour publique, *protected* pour protégée, ou *private* pour privée, définissent un niveau d'accès sur lequel nous reviendrons.

F. Présentation du constructeur

Pour récupérer directement lors de l'instanciation les attributs d'une classe, il faut faire appel à une méthode spéciale nommée le **constructeur** dont le nom est *__construct()* et dont un des rôles est d'initialiser les attributs de la classe.

Puisqu'il s'agit d'une méthode, et donc d'une fonction, nous pouvons lui passer en argument des variables qui permettront de créer un objet possédant les attributs de la classe et pour lesquelles nous définirons une valeur lors de l'instanciation.

```
public function __construct($init_color, $init_size, $init_price)
```

Fig. 21

Afin que les méthodes, et donc notamment le constructeur, puissent accéder aux attributs, nous pouvons également utiliser la pseudo-variable *$this* puisqu'elle permet, comme nous l'avons vu, d'effectuer une référence à la classe en cours. À l'intérieur d'une classe, *$this* permet donc une référence aux attributs de cette même classe.

Ici la syntaxe consistera à écrire *$this*, l'opérateur flèche puis le nom de l'attribut mais **sans le signe « dollar »**.

Pour accéder à la propriété voulue, la syntaxe est donc *$this->color* (sans le signe dollar !).

```
class Clothing
{
 public $color; // déclaration de l'attribut color
 public $size; // déclaration de l'attribut size
 public $price; // déclaration de l'attribut price
 public function __construct($init_color, $init_size, $init_price)
 {
  // on accède à l'attribut $color par $this
  // on définit la valeur de l'attribut $color par la variable
//$init_color
  $this->color = $init_color;
  // on accède à l'attribut $size par $this
  // on définit la valeur de l'attribut $size par la variable $init_
//size
  $this->size = $init_size;
  // on accède à l'attribut $price par $this
  // on définit la valeur de l'attribut $price par la variable
//$init_price
  $this->price = $init_price;
 }
}
$clothing = new Clothing('blanc', 42, 50); // définition de l'objet
var_dump ($clothing);
```

Fig. 22

Ainsi nous avons donc créé un objet dont les propriétés définies lors de l'instanciation de la classe *Clothing* avec le mot-clef *new* sont la couleur blanche, la taille 42 et le prix de 50 euros.

Le *var_dump* nous donne les informations suivantes sur l'objet :

```
object(Clothing)[1]
 public 'color' => string 'blanc' (length=7)
 public 'size' => int 42
 public 'price' => int 50
```

Fig. 23

Notons que nous pouvons également si besoin définir la valeur des attributs lors de leur création, comme n'importe quelle variable, mais dans ce cas les valeurs seront les mêmes pour tous les objets, à moins de les déclarer à nouveau.

```
class Clothing
{
 public $color = 'blanc';
 public $size = 42;
 public $price = 50;
}
```

Fig. 24

Les propriétés étant définies comme *public*, si nous voulons afficher les valeurs, nous pouvons simplement écrire dans notre script et en dehors de la classe (après instanciation naturellement) le code suivant :

```
echo $clothing->color;
echo $clothing->size;
echo $clothing->price;
```

Fig.25

Cependant, les attributs des classes sont rarement déclarés comme *public* mais généralement en *private* ou *protected*, ce qui ne permet donc pas leur utilisation en dehors de la classe. Si nous définissons leur accès comme *protected* ou *private*, **ces attributs ne seront utilisables qu'à l'intérieur de la classe**, par exemple le code *echo $clothing->color* retournera une erreur PHP.

```
Fatal error: Cannot access protected property Clothing::$color in
chemin_du_fichier ligne 5
```

Fig.26

Le script ne peut en effet accéder à la propriété *$color* de la classe *Clothing* car elle est *protected*. Dans ce cas, le moyen d'accéder à ces informations passe par l'utilisation d'une méthode (constructeur ou méthode dédiée).

Dans la classe :

```
public function getColor ()
{
return $this->color ;
}
```

Fig.27

Dans le script :

```
echo $clothing->getColor(); // affiche blanc
```

Fig.28

Le niveau d'accès doit également être défini pour les méthodes. Nous pouvons hors de la classe utiliser la méthode *getColor()* parce qu'elle est définie comme *public*. Si la fonction était définie en *protected* ou *private*, nous ne pourrions l'utiliser directement en dehors de la classe, mais nous pourrions nous en servir au moyen d'autres méthodes de la classe.

Ajoutons que le constructeur n'est pas toujours utile si nous ne rencontrons pas le besoin d'initialiser les propriétés. Dans ce cas le constructeur est à déclarer mais restera vide :

Bien qu'il doive être dans tous nos développements, le constructeur ne sera pas forcément déclaré s'il ne doit pas être utilisé dans les exemples et démonstrations de ce cours.

```
function __construct()
{
}
```

Fig.29

G. Les constantes de classe

Notons qu'il est également possible de déclarer des constantes de classe, c'est-à-dire des constantes spécifiques à une classe. Les constantes de classe ne se différencient pas des constantes que nous avons déjà rencontrées dans les développements en code procédural, elles contiennent des valeurs qui ne sont pas destinées à changer, s'écrivent en lettres capitales (par convention) mais se déclarent en revanche avec le mot-clef *const* comme nous le voyons dans cet exemple théorique :

```
class Clothing
{
      // déclaration des constantes
      const TAILLE_PETITE = 'S';
      const TAILLE_MOYENNE = 'L';
      const TAILLE_GRANDE = 'XL';
```

Fig.30

La particularité des constantes de classe réside dans la manière dont nous pouvons les appeler. La syntaxe nécessite en effet le mot-clef *self* ainsi que les « **deux doubles points ::** » appelés **opérateur de résolution de portée** comme le montre le code suivant :

```
class Clothing
{
      // déclaration des constantes
      const TAILLE_PETITE = 'S';
      const TAILLE_MOYENNE = 'L';
      const TAILLE_GRANDE = 'XL';
      public function exemple()
      {
         $var = self::TAILLE_PETITE ; // appel de la constante
         return $var ;
      }
}
$clothing = new Clothing();
$value = $clothing->exemple();
var_dump ($value); // affiche string 'S' (length=1)
```

Fig.31

H. Fonctions utiles au développement : *get_object_vars*, *get_class_vars*, *get_class_methods*

Nous pouvons afficher lors du développement la liste des attributs d'un objet au moyen de la fonction PHP *get_object_vars*, comme lorsque nous utilisons *var_dump* ou *print_r*. Cette fonction *get_object_vars* ne récupère en revanche que les informations auxquelles le code a accès et donc cela signifie que si nous l'employons en dehors de la classe nous ne pourrons récupérer que les propriétés définies comme public. Il est alors nécessaire de l'employer au sein de la classe dans une méthode qui serait par exemple spécialement dédiée au débogage du code lors de notre développement.

Dans la classe :

```
public function classProperties()
{
    var_dump(get_object_vars($this));
}
```

Fig.32

Dans le script :

```
echo $clothing->classProperties();
```

Fig.33

Ceci affichera :

```
array (size=3)
  'color' => string 'blanc' (length=7)
  'size' => int 42
  'price' => int 50
```

Fig.34

Remarquons que cette fonction retourne un tableau de données que nous pouvons également récupérer directement au lieu d'utiliser un *var_dump* :

```
public function classProperties()
{
    return get_object_vars($this);
}
```

Fig.35

Il est possible de coupler *get_object_vars* qui retourne les propriétés d'un objet avec la fonction PHP *get_class_vars* qui retourne les valeurs par défaut des propriétés d'une classe.

Dans le même souci de débogage et de vérification des éléments constituants la classe, il est possible de **lister l'ensemble des méthodes de la classe** au moyen de *get_class_methods* qui va retourner un tableau de données contenant les noms des méthodes de la classe.

Ce code peut être effectué simplement dans notre script.

```
$methods = get_class_methods($clothing);
var_dump ($methods);
```

```
array (size=3)
  0 => string '__construct' (length=11)
  1 => string 'getColor' (length=8)
  2 => string 'classProperties' (length=15)
```

Fig. 36

La classe *Clothing* contient donc **trois attributs** (ses variables) et **trois méthodes** (ses fonctions). Rappelons à ce propos que le nombre de propriétés et de méthodes n'est pas limité.

Pour toutes les classes que nous écrirons dans ce livret, les attributs et des méthodes seront définis comme *public*. Nous utiliserons *private* et *protected* dans le prochain cours durant lequel nous approfondirons notre étude de la **POO**.