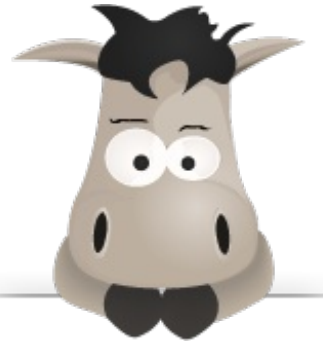

[PHP5] Comment gérer l'absence de surcharge ?

Par Legato

www.siteduzero.com

Dernière mise à jour le 4/07/2010

Sommaire

[PHP5] Comment gérer l'absence de surcharge ?

Par 

Legato

Mise à jour : 04/07/2010
Difficulté : Intermédiaire

[Prérequis]

- Avoir lu le cours **PHP/MySQL** de **M@teo21.**
- Avoir des connaissances en Programmation Orientée Objet.

Il ne s'agit pas de vous enseigner la POO de fond en comble mais de s'intéresser à un point particulier.
Je vais exposer dans ce mini-tutoriel différentes approches pour outrepasser l'absence de surcharge (ici en prenant en exemple les constructeurs) en PHP5. Chacune des 2 approches a ses avantages et ses inconvénients. Votre choix doit dépendre des actions que vous allez effectuer dans la méthode et également de savoir si votre classe est bien définie ou si vous allez la modifier régulièrement.

Précision :
La redéfinition d'une fonction est possible entre une classe mère et une classe fille, c'est-à-dire que l'on peut créer une fonction prenant le même nom et le même nombre de paramètres. Cependant, la surcharge est impossible dans une classe, c'est-à-dire que l'on ne peut pas créer plusieurs fonctions avec le même nom mais avec différents paramètres (tout comme il n'est pas possible de redéfinir une fonction).
Sommaire du tutoriel :

[RAPPEL] La surcharge en PHP5

I. Les constructeurs

Tout d'abord, qu'est-ce qu'un constructeur en POO ?

Un constructeur est une méthode qui permet de créer une instance d'objet.
En PHP5, le constructeur est défini en déclarant la méthode magique `__construct()` .

Code : PHP - Exemple

```php
<?php
class MonObjet
{
    // Constructeur par __construct() == MonObjet().
    public function __construct()
    {
```

```
            echo 'Je construis mon objet.';
        }
    }

    $o = new MonObjet();
    ?>
```

Code : HTML - Résultat

```
Je construis mon objet.
```

Si votre classe hérite d'une classe mère, l'appel du constructeur de celle-ci n'est pas implicite. Il faut alors ajouter la ligne suivante pour appeler son constructeur : `parent::__construct()`

Pour la documentation officielle sur les constructeurs (et destructeurs), c'est par ici.

II. Surcharger une méthode

Si par exemple on souhaite définir plusieurs constructeurs prenant différents paramètres (param1, param2, param3), **on ne peut pas !**

En effet, PHP ne permet pas de déclarer plusieurs fois une fonction du même nom avec différents paramètres. On est obligé d'écrire une fonction ressemblant à celle-ci :

Code : PHP

```
<?php
function meuh($param1,$param2,$param3,...,$paramN)
{
  // Action.
}
?>
```

Et si je veux uniquement les paramètres 1, 3 et 5 ?
Là, c'est le drame !

Code : PHP

```
<?php
meuh('monparam1','','monparam3','','monparam5',...,'');
?>
```

Il faut alors passer tous les paramètres pour éviter une erreur !
Cependant, on peut définir une valeur par défaut à chaque paramètre pour éviter de devoir tous les passer.

Mais à partir du moment où on a défini une valeur par défaut, tous les autres paramètres qui suivent doivent en avoir une, pour éviter un décalage et pour une question de clarté ! Par exemple, si vous avez 4 paramètres, les 2 premiers et le 4^{e} sont obligatoires, le 3^{e} a une valeur par défaut, alors comment savoir si on a précisé le paramètre 3 ou le paramètre 4 ?

On doit d'abord mettre les paramètres essentiels (obligatoires) avant de mettre les non-essentiels.

Cela donne le code suivant :

Code : PHP

```php
<?php
function meuh($param1,$param2 = 'valeur_par_defaut',$param3=
'valeur_par_defaut',...,$paramN = 'valeur_par_defaut')
{
  // Action.
}
?>
```

C'est pourquoi il faut :

- alléger au maximum la fonction ou la découper ;
- trouver une solution alternative.

Nous allons voir maintenant comment pallier à l'absence de la surcharge en PHP5.

[SOLUTION 1] Compter le nombre d'arguments

Pour cette première solution, nous allons :

- passer un nombre indéterminé de paramètres ;
- exécuter le code commun sans distinction de paramètres ;
- compter le nombre d'arguments envoyés au constructeur ;
- appeler une méthode en fonction du nombre de paramètres.

1) Passer un nombre indéterminé de paramètres

On va donc passer à notre constructeur jusqu'à n valeurs de manière ordonnée pour être sûr d'appeler la bonne fonction avec les bons paramètres. Notre constructeur, lui, sera défini sans paramètres dans sa déclaration (mais il peut tout de même en prendre ).

Code : PHP

```php
<?php
class MonObjet
{
    public function __construct()
    {
        //Action(s).
    }
}

 $o = new MonObjet('p1',$p2,array(),$p4,...); // Autant de
paramètres essentiels à envoyer.
?>
```

2) Exécuter le code commun à tous les constructeurs potentiels

Ensuite, avant (ou après, cela dépend de la priorité du code) d'appeler les fonctions correspondant au nombre de paramètres, on déclare les actions à exécuter.

Code : PHP

```php
<?php
class MonObjet
{
    public function __construct()
    {
        //Action(s) commune(s).
        echo 'Je suis commun à tout le monde.';

        //Action(s) spécifique(s).
    }
}
?>
```

3) Compter le nombre d'arguments envoyés au constructeur

Pour cela, on va utiliser une fonction PHP qui s'appelle `func_num_args()` et qui nous renvoie le nombre de paramètres passés au constructeur. Cette donnée doit être stockée dans une variable et non passée directement à une fonction.

Code : PHP

```php
<?php
class MonObjet
{
    public function __construct()
    {
        //Action(s) commune(s).
        echo 'Je suis commun à tout le monde.';

        //Action(s) spécifique(s).
        $cpt = func_num_args();
    }
}
?>
```

4) Faire appel à la fonction associée aux nombres de paramètres

Pour récupérer les paramètres, on fait appel à la fonction `func_get_args()` qui renvoie un tableau contenant l'ensemble des paramètres envoyés à la fonction. Ensuite, grâce à la structure de contrôle `switch($var) ... case` , on va faire appel à la fonction qui correspond à ce nombre.

Code : PHP

```php
<?php
class MonObjet
{
    public function __construct()
    {
        //Action(s) commune(s).
        echo "Je suis commun à tout le monde.";

        //Action(s) spécifique(s).
        $cpt = func_num_args();
 $args = func_get_args();

        switch($cpt)
        {
```

```
                {
                    case '0':
                        //Mon action spécifique ou l'appel de ma fonction
spécifique.
  echo "Je n'ai pas d'argument.";
  $this->fonction_zero_argument();
                    break;

                    case '1':
                        //Mon action spécifique ou l'appel de ma fonction
spécifique.
  echo "J'ai des arguments.";
  $this->fonction_un_argument($args[0]);
                    break;

                    default:
                        //Mon action par défaut ou l'appel de ma fonction
par défaut.
                    break;
                }
            }

 public function fonction_zero_argument()
 {
  echo "Mais pourquoi je n'ai pas d'argument ?";
 }

 public function fonction_un_argument($argument)
 {
  echo "Jaloux ? L'argument c'est [{$argument}] :p.";
 }
}

$o = new MonObjet();
$o2 = new MonObjet("moi le plus fort");
?>
```

Code : HTML - Résultat

```
Je suis commun à tout le monde. Je n'ai pas d'argument. Mais
pourquoi je n'ai pas d'argument ?
Je suis commun à tout le monde. J'ai des arguments. Jaloux ?
L'argument c'est [moi le plus fort] :p.
```

Pertinence de la solution

Les plus malins d'entre vous auront déjà remarqué que cette solution n'est pas utile car elle correspond exactement à la conception d'un constructeur avec le passage de l'ensemble des paramètres. De plus, l'utilisation du switch ainsi que l'appel des fonctions associées sera plus lent que d'avoir directement tout le code dans le constructeur. Ce n'est pas significatif mais il faut le savoir.

Pourquoi nous avoir montré cette solution si elle est équivalente à la conception basique d'un constructeur ?

La réponse est simple : lisibilité du code (et dans la suite du tutoriel).

- On n'a pas un constructeur déclaré avec n paramètres.
- Les tests sur les paramètres s'effectuent dans les fonctions appelées.
- Avec un peu plus de commentaires (normal ou phpDoc), on pourra facilement comprendre l'utilité de la fonction.

[SOLUTION 2] Passage par un tableau de paramètres

Voici grosso modo le schéma d'exécution de cette solution :

- passer un tableau associatif de paramètres ;
- exécuter le code commun sans distinction de paramètres ;
- vérifier l'existence du tableau de paramètres et mettre à jour les propriétés de la classe ;
- exécuter un code particulier.

1) Passer un tableau associatif de paramètres

Certes, un tableau associatif, c'est embêtant à créer.
Certes, ça prend de la place de l'alimenter en données.
Mais vous allez en voir l'utilité d'ici la fin de cette partie.

Code : PHP

```php
<?php
class MonObjet
{
    // Initialiser les valeurs à vide ou à -1 selon le besoin.
    private $nom = "";
    private $type = "";

    // On peut ne pas passer de tableau de paramètres.
    public function __construct($args = null)
    {
        //Action(s).
    }
}

// Déclaration plus lisible mais équivalente à $args = array("nom"
=> "C'est mon nom", "type" => "C'est mon type").
$args = array();
$args['nom'] = "C'est mon nom";
$args['type'] = "C'est mon type";

$o = new MonObjet($args);

?>
```

2) Exécuter le code commun sans distinction de paramètres

On reprend le principe de la solution 1 pour exécuter une action qui doit être effectuée sans tenir compte des paramètres et de leur valeur.

3) Vérifier l'existence du tableau de paramètres et mettre à jour les propriétés de la classe

Tout d'abord, on doit vérifier que l'argument est bien un tableau (grâce à `is_array()`) et ensuite, si ce tableau n'est pas vide (avec `empty()`).
Puis on va mettre à jour les propriétés (existantes) de notre classe.

Code : PHP

```php
<?php
class MonObjet
{
    // Initialiser les valeurs à vide ou à -1 selon le besoin.
```

```
    private $nom = "";
    private $type = "";

    // On peut ne pas passer de tableau de paramètres.
    public function __construct($args = null)
    {
        //Action(s) commune(s)
        echo "Je suis commun à tout le monde. <br />";

        // Si notre paramètre est un tableau non vide.
 if(is_array($args) && !empty($args))
 {
      // Alors pour chaque clé, on récupère sa valeur.
      foreach($args as $key => $value)
      {
  // Si la propriété de la classe existe, alors on met à jour sa
valeur.
  if(isset($this->$key)) $this->$key = $value;
      }

      //Action(s) distincte(s) pour un paramètre.

 }

        //Action(s) distincte(s).

    }
}

 $args = array();
 $args['nom'] = "C'est mon nom";
 $args['type'] = "C'est mon type";

 $o = new MonObjet($args); // Autant de paramètres essentiels à
envoyer.

 var_dump($o);
?>
```

Code : HTML - Résultat

```
Je suis commun à tout le monde.
object(MonObjet)#1 (2) { ["nom:private"]=> string(13) "C'est mon
nom" ["type:private"]=> string(14) "C'est mon type" }
```

4) Exécuter un code particulier

Ensuite, on peut exécuter un code particulier ou faire appel à une fonction selon nos besoins et selon la présence et/ou la valeur d'un paramètre.

Code : PHP

```
<?php
class MonObjet
{
    // Initialiser les valeurs à vide ou à -1 selon le besoin.
    private $nom = "";
    private $type = "";

    // On peut ne pas passer de tableau de paramètres.
    public function __construct($args = null)
    {
```

```
            //Action(s) commune(s)
            echo "Je suis commun à tout le monde. <br />";

        // Si notre paramètre est un tableau non vide.
        if(is_array($args) && !empty($args))
        {
         // Alors pour chaque clé, on récupère sa valeur.
         foreach($args as $key => $value)
         {
          // Si la propriété de la classe existe, alors on met à jour sa
valeur.
          if(isset($this->$key)) $this->$key = $value;
         }

         //Action(s) distincte(s) pour un paramètre.
         if(!empty($args['visibilite']))
         {
          switch($args['visibilite'])
          {
           case 'public':
            echo "Je ne suis pas agoraphobe.";
           break;

           case 'private':
            echo "J'aime pas les gens.";
           break;

           default:
            echo "Je suis indécis.";
           break;
          }

          echo "<br />";
         }
        }

        //Action(s) distincte(s).
        if(!empty($this->type)) echo "Mon type est défini et sa valeur est
: [{$this->type}]<br />";
          }
}

 $args = array();
 $args['nom'] = "C'est mon nom";
 $args['type'] = "C'est mon type";
 $args['visibilite'] = "private";

 $o = new MonObjet($args); // Autant de paramètres essentiels à
envoyer.

 var_dump($o);
?>
```

Code : HTML - Résultat

```
Je suis commun à  tout le monde.
J'aime pas les gens.
Mon type est défini et sa valeur est : [C'est mon type]
object(MonObjet)#1 (2) { ["nom:private"]=> string(13) "C'est mon
nom" ["type:private"]=> string(14) "C'est mon type" }
```

Pourquoi, dans le dump de mon objet, je ne vois pas la propriété `visibilite` ?

C'est simple, elle n'est pas définie donc on ne peut pas la modifier.

Pertinence de la solution

Cette solution est loin d'être optimisée. J'aurais pu directement déclarer l'attribut `$args` du constructeur comme étant un `array`, ce qui aurait évité le test sur le type.

Cependant, en levant son nez et en envisageant le chargement d'un objet sauvegardé dans une base de donnée MySQL par exemple, l'utilité de cette méthode est rapidement trouvée => `mysql_fetch_assoc()`.

En effet, si notre classe a un ensemble de propriétés ayant le même nom que ceux présents dans la table (ou modifiés par la requête), nous avons juste à récupérer le résultat par une requête MySQL et à le transformer en tableau associatif avec `mysql_fetch_assoc()`. Ensuite, on passe notre tableau de paramètres pour initialiser l'objet (on peut ajouter également des paramètres à ce tableau) et le tour est joué.

De plus, même s'il est fastidieux de construire un tableau associatif avec chaque construction d'objet, on peut rapidement voir ce qui est envoyé en paramètre à notre constructeur. De plus, cela facilite le travail de développement et de débogage (même si on peut également afficher les paramètres dans la solution 1 avec les fonctions utilisées, un tableau associatif est un peu plus lisible).

[SOLUTION 3] Objet et Validation

En cours de rédaction...

[CONCLUSION] Ce qu'il faut retenir

Personnellement, je préfère utiliser la solution 2 avec quelques changements dans la classe.

Cependant, il faut adapter la solution à son besoin. Je vous conseille vivement de vous intéresser aux autres méthodes magiques pour compléter la solution 2, afin de l'améliorer largement.

De plus, même si dans le tutoriel on s'est intéressé principalement aux constructeurs, elle s'applique simplement pour la surcharge d'une méthode.

La balle est dans votre camp, prenez soin de vos claviers !

J'espère que ce tutoriel vous a aidé.

Citation

See You Space Codeur...

Partager

Ce tutoriel a été corrigé par les zCorrecteurs.