

## Chapitre 2 : Programmation Orientée Objet

### I. Classe et instance

Les opérations de base pour l'utilisation des objets sont la création d'une classe et la définition de ses propriétés et des méthodes qui vont permettre aux objets créés à partir de la classe d'agir sur leurs propriétés ou de communiquer avec leur environnement. Vient ensuite la création des objets proprement dits en tant qu'instances de la classe de base.

#### I.1. Création d'une classe

Pour créer une classe avec PHP 5, procédez de la façon suivante :

1. Déclarez la classe à l'aide du mot-clé **class** suivi du nom que vous souhaitez lui attribuer.
2. Ouvrez un bloc de code à l'aide d'une accolade ouvrante contenant l'intégralité de la définition de la classe.
3. Déclarez les variables propres de la classe comme des variables ordinaires, avec les mêmes règles de nommage et le caractère **\$** obligatoire. Chaque variable doit être précédée d'un modificateur d'accès précisant les possibilités d'accès à sa valeur. Nous reviendrons sur les choix possibles. Dans un premier temps, faites précéder chaque variable du mot-clé **public**. Les variables peuvent être initialisées avec des valeurs de n'importe quel type reconnu par PHP. En particulier, une variable peut être utilisée comme un tableau créé à l'aide de la fonction `array()`. L'utilisation d'autres fonctions PHP ou d'expressions variables est en revanche interdite pour affecter une valeur à une variable. Le nombre de variables déclarées dans une classe n'est pas limité.
4. Déclarez les fonctions propres de la classe en suivant la même procédure que pour les fonctions personnalisées à l'aide du mot-clé **function**, précédé, comme les variables, d'un spécificateur d'accès (par défaut le mot-clé **public**). Le nom de la fonction ne doit pas non plus commencer par deux caractères de soulignement (`__`), cette notation étant réservée à certaines fonctions particulières de PHP 5.
5. Terminez le bloc de code de la classe par une accolade fermante.

*Syntaxe générale*

```
<?php
class ma_classe
{
//Définition d'une constante
const lang="PHP 5";
//Définition des variables de la classe
public $prop1;
public $prop2 ="valeur";
public $prop3 = array("valeur0", "valeur1");
//Initialisation interdite avec une fonction PHP
//public $prop4= date(" d : m : Y"); Provoque une erreur fatale
//*****
//Définition d'une fonction de la classe
public function ma_fonction($param1, $paramN)
{
//Corps de la fonction
}
}
//fin de la classe
?>
```

L'exemple 1 présente la création d'une classe représentative d'une action boursière contenant des informations générales sur chaque action. La classe nommée **action** contient une constante nommée **PARIS** définissant l'adresse de la Bourse de Paris, deux variables non initialisées, **\$nom**, qui contiendront

l'intitulé de l'action boursière, et **\$cours**, qui en contiendra le prix, ainsi qu'une variable **\$bourse** initialisée à la valeur "Bourse de Paris", qui représentera la place boursière par défaut.

La classe **action** contient également une fonction propre définie par **info()**, qui, selon l'heure d'exécution du script et le jour de la semaine, indique si les bourses de Paris et de New York sont ouvertes ou non.

#### Exemple 1 : Création d'une classe action

```
<?php
class action
{
//Constante
const PARIS="Palais Brognard";
//variables propres de la classe
public $nom;
public $cours;
public $bourse="bourse de Paris ";
//fonction propre de la classe
public function info()
{
echo "Informations en date du ",date("d/m/Y H:i:s"), "<br>";
$now=getdate();
$heure= $now["hours"];
$jour= $now["wday"];
echo "<h3>Horaires des cotations</h3>";
if(($heure>=9 && $heure <=17)&& ($jour!=0 && $jour!=6))
{ echo "La Bourse de Paris est ouverte <br>"; }
else
{ echo "La Bourse de Paris est fermée <br>"; }
if(($heure>=16 && $heure <=23)&& ($jour!=0 && $jour!=6) )
{ echo "La Bourse de New York est ouverte <hr>"; }
else
{ echo "La Bourse de New York est fermée <hr>"; }
}
}
?>
```

Le rôle de la POO étant de créer des bibliothèques de classes réutilisables par n'importe quel script, enregistrez le code de la classe action dans un fichier séparé. Pour l'utiliser, créez un fichier séparé (l'exemple 2) destiné à utiliser cette classe en incorporant son code à l'aide de la fonction **require()**. Cette pratique est fortement recommandée.

#### Exemple 2. Utilisation de la classe action

```
<?php
require("objet2.php");

echo "Constante PARIS =",PARIS,"<br />";
echo "Nom = ",$nom,"<br />";
echo "Cours= ",$cours,"<br />";
echo "Bourse= ",$bourse,"<br />";
//info(); //L'appel de info()Provoque une erreur si vous décommentez la ligne
action::info();//fonctionne

echo "Constante PARIS =",action::PARIS,"<br />";
?>
```

Le résultat relatif à l'appel et l'exécution de la fonction est le suivant :

## Horaires des cotations

La Bourse de Paris est fermée

La Bourse de New York est fermée

---

Constante PARIS =Palais Brognard

Vous constatez en exécutant le script que les variables et les fonctions créées dans le corps de la classe ne sont pas accessibles dans le reste du code du script, même si elles se situent après la définition de la classe. La tentative d’affichage de la constante et des variables de la classe ne produit aucun affichage.

Aussi PHP ne reconnaît pas la fonction appelée et qu’il la considère comme non déclarée. Pour pouvoir l’utiliser en dehors de la création d’un objet instance de la classe action, il vous faut faire appel à la syntaxe particulière suivante :

```
nom_classe::nom_fonction();
```

en précisant éventuellement ses paramètres, s’ils existent. Cela vous permet d’utiliser les fonctions propres d’une classe en dehors de tout contexte objet si elles sont déclarées **public**.

De même, vous pouvez accéder à la constante définie dans la classe en utilisant la même syntaxe en dehors de la classe :

```
echo "Constante PARIS =", action::PARIS, "<br />";
```

qui affichera bien « Palais Brognard ».

## 1.2. Créer un objet

Chaque objet créé à partir de votre classe est appelé une instance de la classe. À chaque instance correspond un objet différent. Un objet particulier est identifié par un nom de variable et est créé à l’aide du mot-clé **new** selon le modèle suivant :

```
$var_objet = new nom_classe()
```

Prenez soin que le code de définition de la classe soit dans le même script ou soit inclus au début du script à l’aide des fonctions **require()** ou **include()**.

À partir de maintenant, le script possède une variable **\$var\_objet** qui a les propriétés et les méthodes définies dans la classe de base.

Vous pouvez le vérifier en écrivant :

```
echo gettype($var_objet)
```

La valeur retournée par cette fonction est bien **object**, Pour créer des objets représentant des actions boursières conformes au modèle de votre classe action, vous écrivez donc :

```
$action1 = new action();
```

```
$action2 = new action();
```

Il est possible de vérifier si un objet particulier est une instance d’une classe donnée en utilisant l’opérateur **instanceof** pour créer une expression conditionnelle, selon la syntaxe suivante :

```
if($objet instanceof nom_classe) echo "OK";
```

Il vous faut maintenant personnaliser vos objets afin que chacun d’eux corresponde à une action boursière particulière. Vous agissez pour cela sur les propriétés d’un objet que vous venez de créer afin de le distinguer d’un autre objet issu de la même classe.

Pour accéder, aussi bien en lecture qu’en écriture, aux propriétés d’un objet, PHP offre la syntaxe suivante :

```
$nom_objet->prop;
```

ou encore :

```
$nom_objet->prop[n];
```

si la propriété **prop** de l’objet est un tableau.

Pour appeler une méthode de l'objet, appliquez la même notation :

```
$nom_objet->nom_fonction();
```

Vous pouvez afficher la valeur d'une propriété ou lui en affecter une autre. Par exemple, pour afficher la valeur de la propriété bourse d'un objet de type action, vous écrivez :

```
echo $action1->bourse;
```

qui affiche la valeur par défaut « Bourse de Paris » définie dans la classe.

Pour lui affecter une nouvelle valeur, vous avez le code :

```
$action1->bourse = "New York";
```

Pour appeler la seule méthode de l'objet :

```
$action1->info();
```

L'exemple 3 donne une illustration de la création d'objets à partir de la classe **action** puis de la définition des propriétés et enfin de l'utilisation de ces propriétés pour un affichage d'informations.

### Exemple 3. Création et utilisation d'objets

```
<?php
```

```
require("objet2.php");
```

```
//Création d'une action
```

```
$action1= new action();
```

```
//Affectation de deux propriétés
```

```
$action1->nom = "Mortendi";
```

```
$action1->cours = 15.15;
```

```
//Utilisation des propriétés
```

```
echo "<b>L'action $action1->nom cotée à la $action1->bourse vaut $action1->cours&euro;</b><hr>";
```

```
//Appel d'une méthode
```

```
$action1->info();
```

```
echo "La structure de l'objet \ $action1 est : <br>";
```

```
//La fonction var_dump() permet d'afficher le nom, le type et la valeur de chaque propriété
```

```
var_dump($action1);
```

```
// Lire l'ensemble des propriétés de l'objet $action1 à l'aide d'une boucle foreach
```

```
echo "<h4>Descriptif de l'action</h4>";
```

```
foreach($action1 as $prop=>$valeur)
```

```
{
```

```
echo "$prop = $valeur <br />";
```

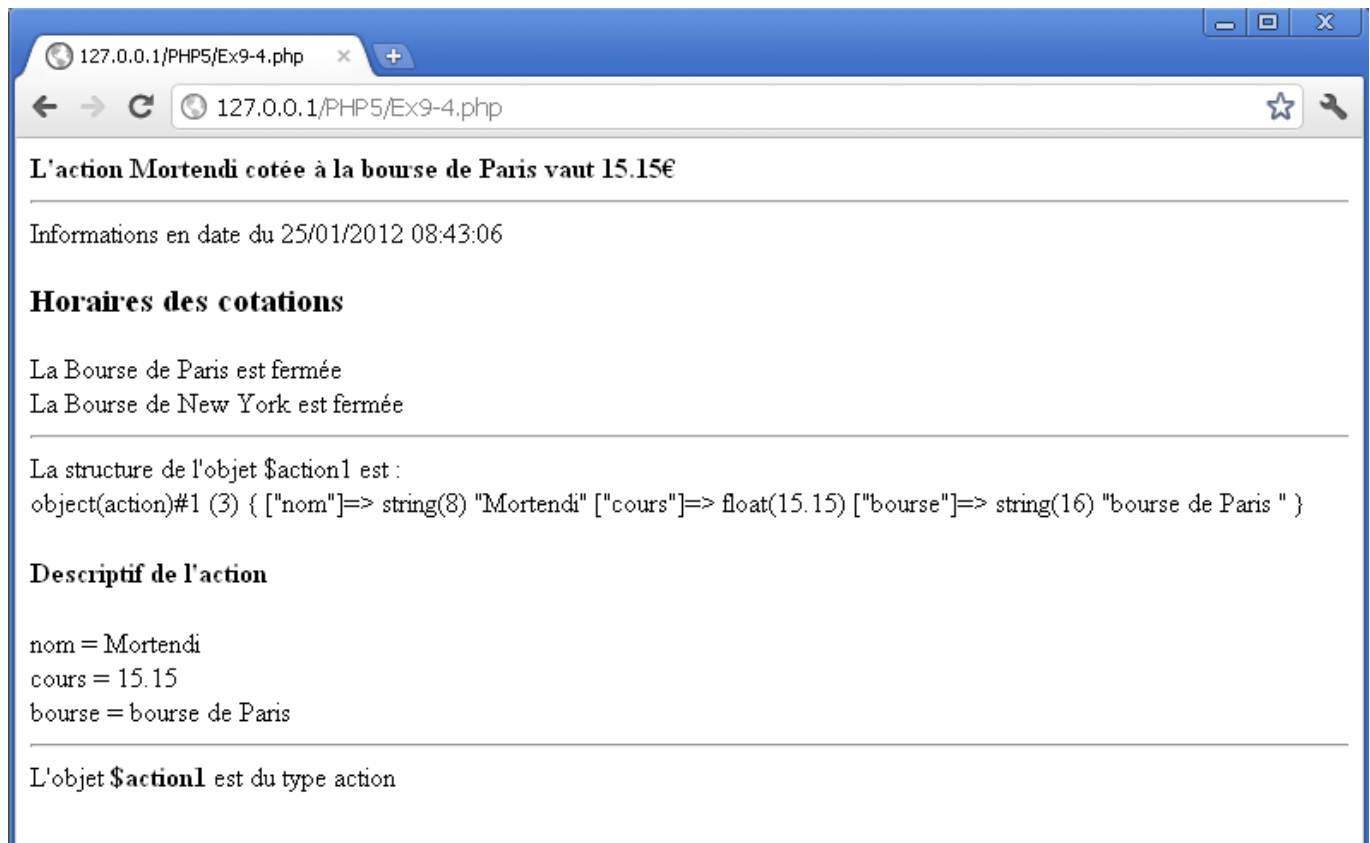
```
}
```

```
//Vérifier si $action1 est une instance de la classe action
```

```
if($action1 instanceof action) echo "<hr />L'objet \ $action1 est du type  
action";
```

```
?>
```

La figure suivante donne le résultat du script.



### I.3. Accès aux variables de la classe

Pour accéder à une constante de classe dans le corps d'une fonction, utilisez la syntaxe suivante :

**self::maconstante**

ou encore :

**nomclasse::maconstante**

Pour qu'une méthode accède aux variables déclarées dans la classe, elle doit y faire appel à l'aide de la syntaxe suivante :

**\$this->mavar**

dans laquelle la pseudo-variable **\$this** fait référence à l'objet en cours, ce qui permet d'utiliser la variable **\$mavar** dans la méthode. La méthode **info()** de votre classe **action** peut maintenant être enrichie et avoir comme fonctionnalité supplémentaire d'afficher toutes les caractéristiques d'un objet action.

Vous pouvez, par exemple, remplacer la ligne de code :

```
echo "<b>L'action $action1->nom cotée à la bourse de $action1->bourse vaut  
$action1->cours _</b><hr>";
```

de l'exemple 3 par le code suivant, qui fera partie du corps de la fonction **info()** :

```
if(isset($this->nom) && isset($this->cours))
```

```
{
```

```
echo "<b>L'action $this->nom cotée à la bourse de {$this->bourse[0]} vaut  
$this->cours &euro;</b><br />";
```

```
}
```

La vérification de l'existence des variables permet de bloquer l'affichage dans le cas où aucun objet n'a été créé, sans pour autant empêcher l'appel de la fonction **info()**.

Cet accès aux variables de la classe est aussi valable si l'une de ces variables est un tableau. Pour accéder à la valeur d'un des éléments du tableau, vous écrirez :

```
$this->montab[1]
```

si la variable **\$montab** a été déclarée dans la classe avec la fonction **array()** selon le modèle :

```
public $montab = array("valeur1", "valeur2");
```

L'exemple 4 permet de modifier la classe **action** qui définit deux constantes utilisées ensuite par la méthode info(). Vous y retrouvez les mêmes variables \$nom, \$cours et \$bourse, qui est un tableau. La méthode info() utilise la variable globale \$client, qui sera définie dans le script créant un objet de type action, ainsi que le tableau superglobal \$\_SERVER pour lire le nom du serveur. La lecture des éléments du tableau \$bourse permet l'affichage des horaires d'ouverture des bourses. En cas de création d'un objet et donc de définition des valeurs de ses propriétés nom et cours, la fonction vous permet d'afficher les informations sur l'action créée.

#### Exemple 4. Utilisation des variables propres par une méthode

```
<?php
class action
{
//Définition d'une constante
const PARIS="Palais Brognard";
const NEWYORK="Wall Street";
//Variables propres de la classe
public $nom ;
public $cours;
public $bourse=array("Paris ","9h00","18h00");

//fonctions propres de la classe
function info()
{
global $client;
//Utilisation de variables globales et d'un tableau superglobal
echo "<h2> Bonjour $client, vous êtes sur le serveur: ",
$_SERVER["HTTP_HOST"],"</h2>";
echo "<h3>Informations en date du ",date("d/m/Y H:i:s"),"</h3>";

echo "<h3>Bourse de {$this->bourse[0]} Cotations de {$this->bourse[1]} à
{$this->bourse[2]} </h3>";

//Informations sur les horaires d'ouverture
$now=getdate();
$heure= $now["hours"];
$jour= $now["wday"];
echo "<hr />";
echo "<h3>Heures des cotations</h3>";
if(($heure>=9 && $heure <=17)&& ($jour!=0 && $jour!=6))
{ echo "La Bourse de Paris ( ", self:: PARIS," ) est ouverte<br>"; }
else
{ echo "La Bourse de Paris ( ", self:: PARIS," ) est fermée <br>"; }
if(($heure>=16 && $heure <=23)&& ($jour!=0 && $jour!=6) )
{ echo "La Bourse de New York ( ", self:: NEWYORK," ) est ouverte<hr>"; }
else
{echo "La Bourse de New York ( ", self:: NEWYORK," ) est fermée <hr>"; }
//Affichage du cours
if(isset($this->nom) && isset($this->cours))
{
echo "<b>L'action {$this->nom} cotée à la bourse de {$this->bourse[0]} vaut
{$this->cours} <euro>;</b><br />";
}
}
?>
```

L'exemple 5 utilise cette classe pour créer des objets après l'inclusion du fichier **objet5.php**. La variable **\$client** est initialisée et sera utilisée par la méthode **info()**. Après la création d'un objet **action** puis la définition de ses propriétés, l'appel de la méthode **info()** de l'objet affiche l'ensemble des informations sur l'action créée et l'ouverture de la Bourse.

#### Exemple 5. Création d'un objet action

```
<?php
```

```
require('objet5.php');
```

```
$client="Geelsen";
```

```
$mortendi = new action();
```

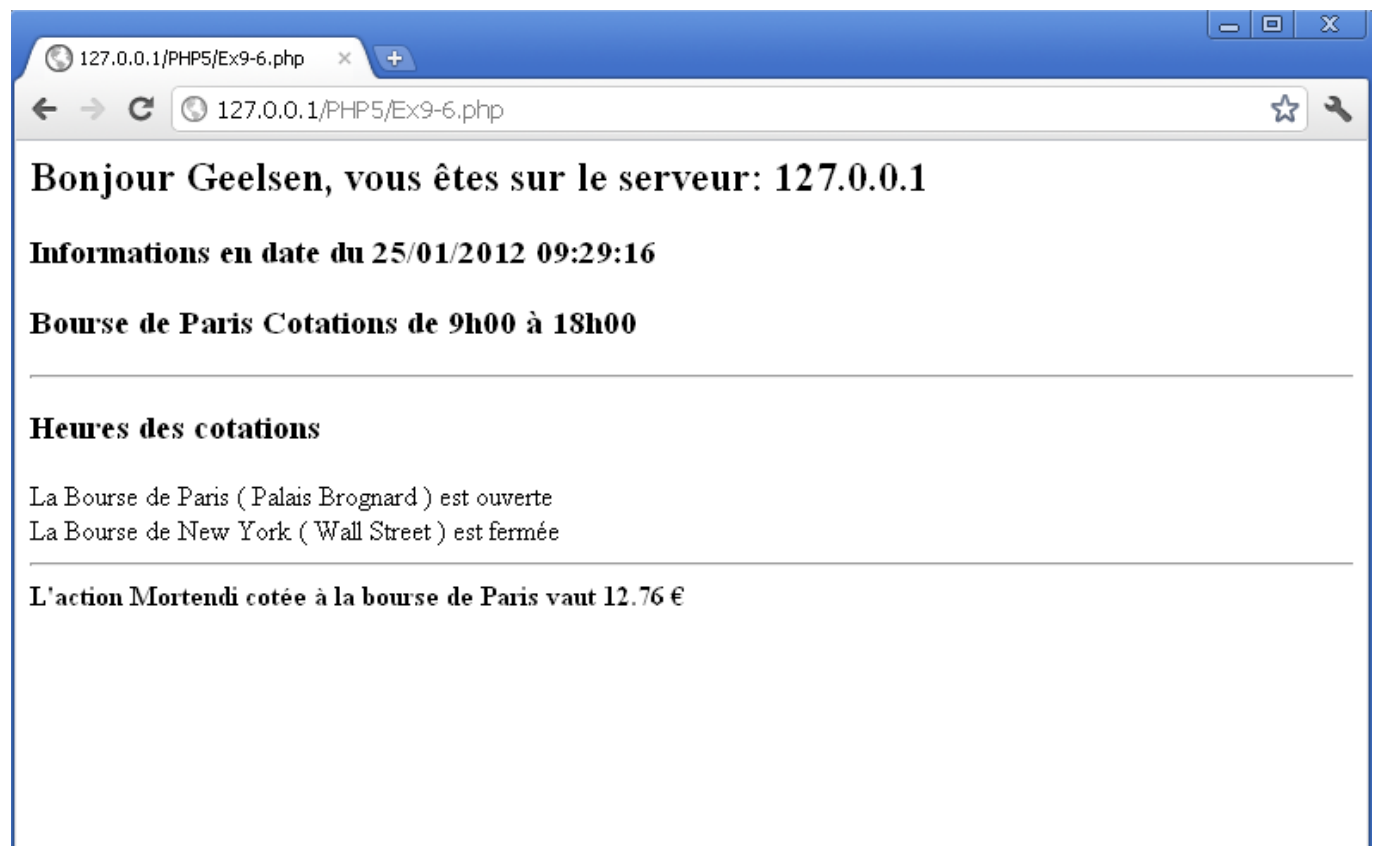
```
$mortendi->nom ="Mortendi";
```

```
$mortendi->cours="12.76";
```

```
$mortendi->info();
```

```
?>
```

La figure suivante illustre le résultat obtenu.



## I.4. Les modificateurs d'accessibilité

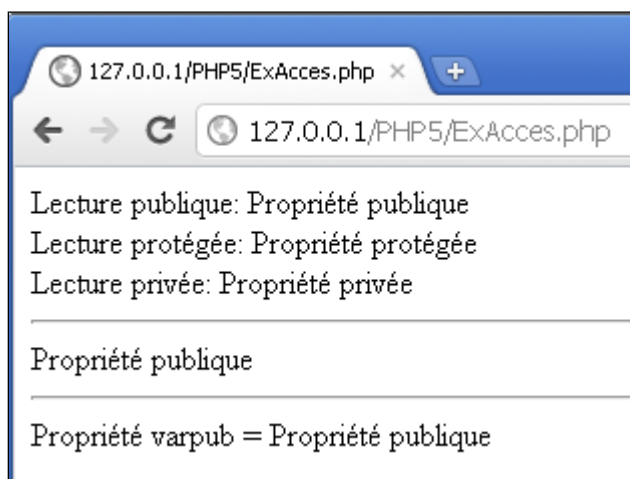
### Accessibilité des propriétés

Il existe trois options d'accessibilité, qui s'utilisent en préfixant le nom de la variable de la classe. Ces options sont les suivantes :

- **public**. Permet l'accès universel à la propriété, aussi bien dans la classe que dans tout le script, y compris pour les classes dérivées, comme vous l'avez vu jusqu'à présent.
- **protected**. La propriété n'est accessible que dans la classe qui l'a créée et dans ses classes dérivées (voir la section « Héritage » de ce chapitre).
- **private**. C'est l'option la plus stricte : l'accès à la propriété n'est possible que dans la classe et nulle part ailleurs.

Le code ci-dessous teste les différents niveaux d'accessibilité aux propriétés.

```
<?php
class acces
{
    //Variables propres de la classe
    public $varpub ="Propriété publique";
    protected $varpro="Propriété protégée";
    private $varpriv="Propriété privée";
    function lireprop()
    {
        echo "Lecture publique: $this->varpub","<br />";
        echo "Lecture protégée: $this->varpro","<br />";
        echo "Lecture privée: $this->varpriv","<hr />";
    }
}
$objet=new acces();
$objet->lireprop();
echo "<hr />";
echo $objet->varpub;
//echo $objet->varpriv; Erreur fatale
//echo $objet->varpro; Erreur fatale
echo "<hr />";
foreach(get_class_vars('acces') as $prop=>$val)
{
    echo "Propriété ",$prop ," = ",$val,"<br />";
}
?>
```





## Accessibilité des méthodes

PHP 5 permet désormais de définir des niveaux d'accessibilité pour les méthodes des objets. Vous retrouvez les mêmes modificateurs que pour les propriétés :

- **public**. La méthode est utilisable par tous les objets et instances de la classe et de ses classes dérivées.
- **protected**. La méthode est utilisable dans sa classe et dans ses classes dérivées, mais par aucun objet.
- **private**. La méthode n'est utilisable que dans la classe qui la contient, donc ni dans les classes dérivées, ni par aucun objet.

Tout appel d'une méthode en dehors de son champ de visibilité provoque une erreur fatale. L'exemple 6 illustre l'emploi de ces modificateurs dans une classe.

### Exemple 6. Accessibilité des méthodes

```
<?php
class accesmeth
{
//Variables propres de la classe
private $code="Mon code privé";
//Méthodes
//Méthode privée
private function lirepriv()
{
echo "Lire privée ",$this->code,"<br />";
}
//Méthode protégée
protected function lirepro()
{
echo "Lire protégée ",$this->code,"<br />";
}
//Méthode publique
public function lirepub()
{
echo "Lire publique : ",$this->code,"<br />";
$this->lirepro();
$this->lirepriv();
}
}
//*****
//Appels des méthodes
$objet=new accesmeth();
$objet->lirepub();
//$objet->lirepro();Erreur fatale
//$objet->lirepriv();Erreur fatale
?>
```

La figure suivante illustre le résultat obtenu.



## I.5. Propriétés et méthodes statiques

PHP 5 introduit la notion de propriété et de méthode statique, qui permet d'accéder à ces éléments sans qu'il soit besoin de créer une instance de la classe. Pour déclarer une propriété ou une méthode statique, vous devez faire suivre le mot-clé définissant l'accessibilité du mot-clé **static**.

Comme les méthodes statiques sont utilisables sans la création d'objet, vous ne devez pas utiliser la pseudo-variable **\$this** pour faire référence à une propriété de la classe dans le corps de la méthode. Vous devez utiliser à la place une des syntaxes suivantes :

- si la méthode est celle de la même classe : **self::\$propriété**
- si la méthode est celle d'une autre classe : **nomclasse::\$propriété**

Notez qu'il faut conserver le signe \$ pour désigner la propriété.

De même, pour appeler une méthode statique de la classe à partir d'une autre méthode, vous utilisez les mêmes syntaxes, avec les mêmes conditions que ci-dessus :

- **self::\$méthode()**
- **nomclasse::\$méthode()**

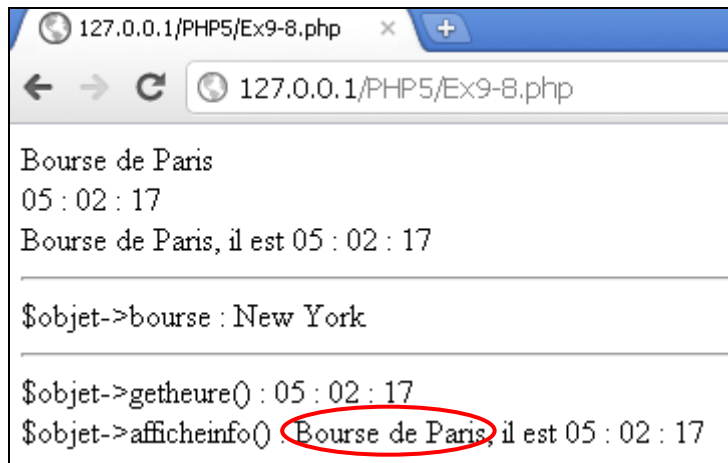
Si vous créez un objet instance de la classe, la propriété déclarée **static** n'est pas accessible à l'objet en écrivant le code **\$objet->propriété**. Par contre, les méthodes statiques sont accessibles par l'objet avec la syntaxe habituelle **\$objet->méthode()**.

Si vous modifiez la valeur d'une propriété déclarée statique à partir d'un objet, cette modification n'est pas prise en compte par les méthodes qui utilisent cette propriété. L'exemple 7 présente ces différentes caractéristiques et leur emploi.

### Exemple 7. Propriété et méthode statiques

```
<?php
class info
{
    //Propriété statique
    public static $bourse="Bourse de Paris";
    //Méthodes statiques
    public static function getheure()
    {
        $heure=date("h : m : s");
        return $heure;
    }
    public static function afficheinfo()
    {
        $texte=info::$bourse.", il est ".self::getheure()";
        return $texte;
    }
}
echo info::$bourse,"<br />";
echo info::getheure(),"<br />";
echo info::afficheinfo(),"<hr />";
//Création d'un objet info
$objet=new info();
$objet->bourse="New York";
echo "\$objet->bourse : ",$objet->bourse,"<hr />";
echo "\$objet->getheure() : ",$objet->getheure(),"<br />";
echo "\$objet->afficheinfo() : ",$objet->afficheinfo(),"<br />";
?>
```

Le résultat obtenu est le suivant :



```
127.0.0.1/PHP5/Ex9-8.php x +
127.0.0.1/PHP5/Ex9-8.php
Bourse de Paris
05 : 02 : 17
Bourse de Paris, il est 05 : 02 : 17
$objet->bourse : New York
$objet->getheure() : 05 : 02 : 17
$objet->afficheinfo() : Bourse de Paris, il est 05 : 02 : 17
```

### Remarque :

L'utilisation d'une propriété statique dans un contexte objet peut présenter un danger. En effet, dans l'exemple précédent, une nouvelle valeur est affectée à la propriété bourse. L'affichage de cette propriété montre que cette affectation est bien réalisée. En revanche, l'appel de la méthode de l'objet qui utilise cette propriété permet de constater que la propriété a toujours la valeur qui a été définie dans la classe. Pour pallier cet inconvénient, il faudrait ajouter à la classe une méthode spéciale qui modifierait la propriété bourse comme l'illustre le code suivant :

```
<?php
class info
{
    //Propriété statique
    public static $bourse="Bourse de Paris";
    //Méthodes statiques
    public static function getheure()
    {
        $heure=date("h : m : s");
        return $heure;
    }

    public function setbourse($val)
    {
        info::$bourse=$val;
    }

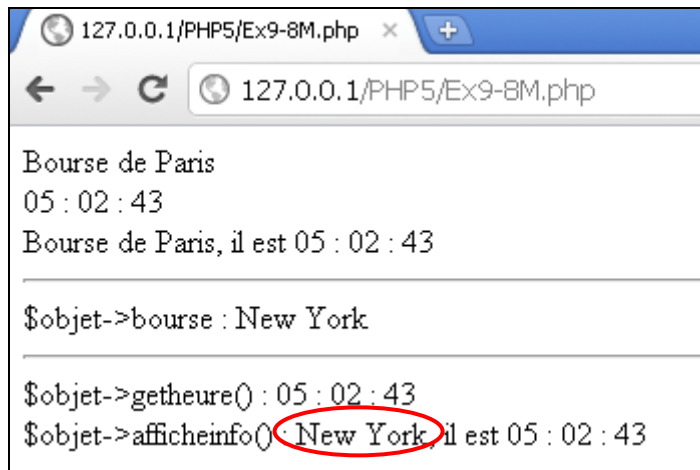
    public static function afficheinfo()
    {
        $texte=info::$bourse.", il est ".self::getheure();
        return $texte;
    }
}

echo info::$bourse,"<br />";
echo info::getheure(),"<br />";
echo info::afficheinfo(),"<hr />";
//Création d'un objet info
$objet=new info();
$objet->setbourse("New York");

echo "\$objet->bourse : ",info::$bourse,"<hr />";

echo "\$objet->getheure() : ",$objet->getheure(),"<br />";
echo "\$objet->afficheinfo() : ",$objet->afficheinfo(),"<br />";
?>
```

Le résultat obtenu est le suivant :



## I.6. Constructeur et destructeur d'objet

Dans ce qui précède, vous avez créé des objets en instanciant la classe action puis avez défini les propriétés des objets ainsi créés. Cette méthode est un peu lourde, car elle implique de définir les propriétés une par une. Il existe une façon plus rapide de créer des objets et de définir leurs propriétés en une seule opération. Elle consiste à créer un constructeur d'objet, qui n'est rien d'autre qu'une fonction spéciale de la classe, dont les paramètres sont les valeurs que vous voulez attribuer aux propriétés de l'objet.

PHP 5 permet de créer des constructeurs avec la méthode `__construct()`, dont la syntaxe est la suivante :

```
void __construct(divers $argument1,...,argumentN)
```

Cette méthode, dite « méthode magique » comme toutes celles qui commencent par deux caractères de soulignement (`__`), porte le même nom, quelle que soit la classe, ce qui permet des mises à jour sans avoir à modifier le nom du constructeur. Elle ne retourne aucune valeur et est utilisée généralement pour initialiser les propriétés de l'objet.

Elle est appelée automatiquement lors de la création d'un objet à l'aide du mot-clé `new` suivi du nom de la classe et des paramètres du constructeur, en utilisant la syntaxe suivante :

```
$mon_objet = new nom_classe(param1,param2,...)
```

Vous avez créé un objet nommé `$mon_objet` et initialisé chacune de ses propriétés avec les valeurs des paramètres passés à la fonction.

De même, vous avez la possibilité avec PHP 5 d'utiliser des destructeurs à l'aide de la fonction `__destruct()`, dont la syntaxe est la suivante :

```
void __destruct()
```

Elle s'utilise sans paramètre car elle n'est généralement pas appelée directement et ne retourne aucune valeur. Elle est appelée automatiquement soit après la destruction explicite de l'objet avec la fonction `unset()`, soit après la fin du script et la disparition de toutes les références à l'objet.

L'exemple 8 illustre ces notions.

### Exemple 8. La classe action munie d'un constructeur et d'un destructeur

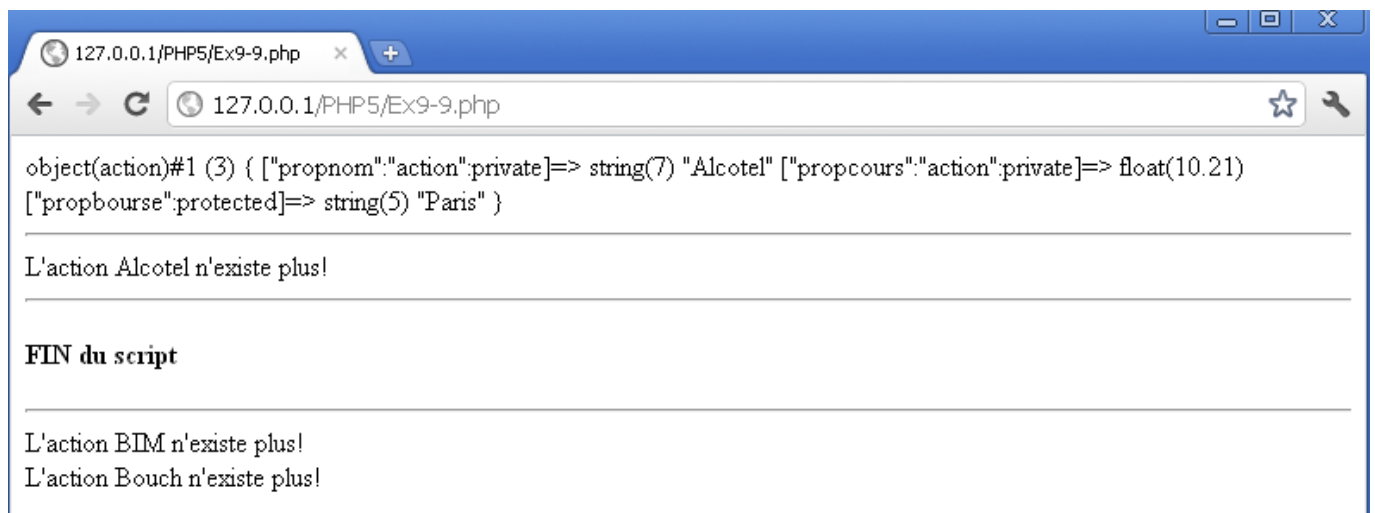
```
<?php
class action
{
    private $propnom;
    private $propcours;
    protected $propbourse;
    function __construct($nom,$cours,$bourse="Paris")
    {
        $this->propnom=$nom;
        $this->propcours=$cours;
        $this->propbourse=$bourse;
    }
}
```

```

function __destruct()
{
echo "L'action $this->propnom n'existe plus!<br />";
}
}
//Création d'objets
$alcotel = new action("Alcotel",10.21);
$bouch = new action("Bouch",9.11,"New York");
$bim = new action("BIM",34.50,"New York");
// Créer une référence à l'objet $bim
$ref=&$bim;
var_dump($alcotel);
echo "<hr />";
unset($alcotel);
unset($bim);
echo "<hr /><h4> FIN du script </h4><hr />";
?>

```

Le résultat obtenu est le suivant :



### Remarque :

La destruction explicite de l'objet Alcotel entraîne l'appel du destructeur et l'affichage d'un message. Par contre, la destruction explicite de l'objet \$bim ne provoque pas l'appel du destructeur car il existe encore une référence à cet objet. En consultant l'affichage réalisé par le script, vous constatez qu'après le message de fin de script, le destructeur est appelé pour les objets \$bim et \$bouch.

## I.7. Déréférencement

Vous avez vu que l'appel d'une méthode d'objet se faisait selon la syntaxe suivante :

```
$varobj->methode();
```

Dans le cas où la méthode appliquée à un objet retourne elle-même un objet et que celui-ci possède ses propres méthodes, il est possible avec PHP 5 de pratiquer le déréférencement. Cela permet d'enchaîner les appels de méthodes les uns à la suite des autres.

Vous pouvez écrire le code suivant :

```
$varobj->methode1()->methode2();
```

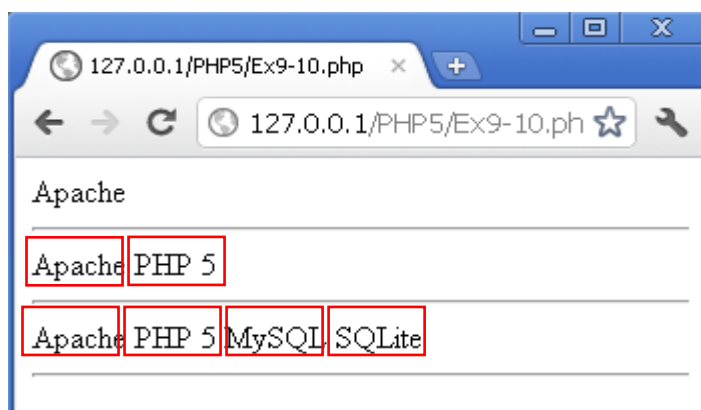
à condition que `methode2()` soit une méthode de l'objet obtenu par l'appel de `methode1()`.

Dans l'exemple 9, vous créez une classe nommée `vvarchar` représentant une chaîne de caractères. Cette classe possède trois méthodes. Le constructeur définit la propriété `chaîne` avec la valeur du paramètre qui lui est passé. La méthode `add()` réalise la concaténation de deux chaînes et retourne l'objet en cours (de type `vvarchar`), dont la propriété `chaîne` est modifiée. La méthode `getch()` retourne la valeur de la propriété `chaîne`. L'exécution de l'exemple 9 illustre l'appel successif de plusieurs méthodes.

### Exemple 9. Déréférencement de méthodes

```
<?php
class vvarchar
{
private $chaîne;
function __construct($a)
{
$this->chaîne= $a;
}
function add($addch)
{
$this->chaîne.=$addch;
return $this;
}
function getch()
{
return $this->chaîne;
}
}
//Création d'objet
$texte=new vvarchar("Apache ");
echo $texte->getch(),"<hr />";
echo $texte->add(" PHP 5 ")>getch(),"<hr />";
echo $texte->add(" MySQL ")>add("SQLite ")>getch(),"<hr />";
?>
```

L'exécution du script affiche le résultat suivant :



## II. Héritage

### II.1. Création d'une classe dérivée

Le mécanisme de l'héritage est fondamental en POO. Il vous permet, en fonction des besoins, de faire évoluer une classe sans la modifier en créant une classe dérivée (on dit aussi une classe enfant, ou une sous-classe) à partir d'une classe de base, ou classe parente. La classe dérivée hérite des caractéristiques (propriétés et méthodes) de la classe parente, et vous lui ajoutez des fonctionnalités supplémentaires. Vous pouvez ainsi créer toute une hiérarchie de classes en spécialisant chaque classe selon vos besoins. Contrairement à d'autres langages, PHP 5 n'autorise que l'héritage simple, une classe ne pouvant hériter que d'une seule classe parente.

Pour créer une classe enfant, faites suivre le nom de la nouvelle classe du mot-clé **extends** puis du nom de la classe parente, selon la forme suivante :

```
class classenfant extends classparent
{
//Propriétés et méthodes nouvelles
}
```

Dans le corps de la classe enfant, il est possible de redéfinir les propriétés et les méthodes de la classe parente, sauf si elles sont déclarées **private** ou **final** (voir plus loin). Il est encore possible d'accéder aux propriétés et aux méthodes redéfinies de la classe parente en les faisant précéder du mot-clé **parent::**. Si elles ne sont pas redéfinies, la classe enfant possède les mêmes propriétés et les mêmes méthodes que la classe parente.

#### Cas particulier des constructeurs et destructeurs

Même si la classe parente possède un constructeur et un destructeur créés avec les méthodes **\_\_construct()** et **\_\_destruct()**, la classe enfant ne possède pas ces méthodes par défaut. Il faut recréer un constructeur et un destructeur propres à la classe enfant. Pour utiliser ceux de la classe parente, il faut les appeler explicitement avec la syntaxe **parent::\_\_construct()** ou **parent::\_\_destruct()**.

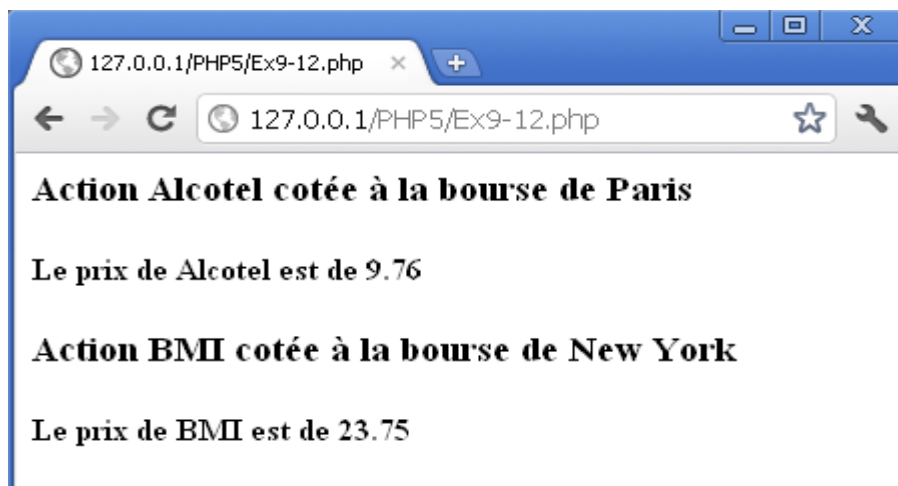
Ces notions se trouvent illustrées dans l'exemple 10. Cet exemple illustre le mécanisme de l'héritage en créant une classe **valeur** représentant une valeur mobilière. À elle seule, cette classe pourrait permettre la création d'objets. Elle contient deux propriétés et deux méthodes, un constructeur et une méthode d'affichage. À partir de cette classe de base, vous créez une classe dérivée **action**, qui possède une propriété supplémentaire. Elle redéfinit un constructeur en utilisant le constructeur parent et enrichit la fonction d'affichage.

#### Exemple 10. Création de classe enfant

```
<?php
//Classe valeur
class valeur
{
protected $nom;
protected $prix;
function __construct($nom,$prix)
{
$this->nom=$nom;
$this->prix=$prix;
}
protected function getinfo()
{
$info="<h4>Le prix de $this->nom est de $this->prix </h4>";
return $info;
}
}
```

```
//Classe action
class action extends valeur
{
public $bourse;
function __construct($nom,$prix,$bourse="Paris")
{
parent::__construct($nom,$prix);
$this->bourse=$bourse;
}
public function getinfo()
{
$info="<h3>Action $this->nom cotée à la bourse de $this->bourse </h3>";
$info.=parent::getinfo();
return $info;
}
}
//Création d'objets
$action1 = new action("Alcotel",9.76);
echo $action1->getinfo();
$action2 = new action("BMI",23.75,"New York");
echo $action2->getinfo() ;
?>
```

L'affichage obtenu est présenté à la figure suivante :



## II.2. Les classes abstraites

Une classe abstraite ne permet pas l'instanciation d'objets mais sert uniquement de classe de base pour la création de classes dérivées. Elle définit un cadre minimal auquel doivent se conformer les classes dérivées.

Une classe abstraite peut contenir des méthodes déclarées public ou protected, qu'elles soient elles-mêmes abstraites ou non. Une méthode abstraite ne doit contenir que sa signature, sans aucune implémentation. Chaque classe dérivée est chargée de créer sa propre implémentation de la méthode. Une classe contenant au moins une méthode abstraite doit obligatoirement être déclarée abstraite, sinon elle permettrait de créer des objets qui auraient une méthode non fonctionnelle.

Pour créer une classe abstraite, faites précéder le mot-clé class du mot-clé abstract, comme ceci :

```
abstract class nomclasse
{
//Définition de la classe
}
```



Pour créer une méthode abstraite, faites également précéder le modificateur d'accès du mot-clé `abstract`, selon le modèle suivant :

```
abstract public function nomfonction() ;
```

Dans la classe qui dérive d'une classe abstraite, vous devez définir les modificateurs d'accessibilité des méthodes avec une visibilité égale ou plus large que celle de la méthode abstraite. Une classe abstraite définie, par exemple, `protected`, est implémentée dans les classes dérivées comme `protected` ou `public`.

L'exemple 11 reprend la création de la classe enfant `action` de l'exemple 10. Il permet de réaliser la même opération mais à partir d'une classe abstraite nommée `valeur`. Cette classe `valeur` contient deux propriétés et deux méthodes abstraites. La classe dérivée doit donc créer sa propre implémentation complète de ces méthodes. Il n'est plus question ici d'utiliser une méthode parente, comme dans l'exemple 10.

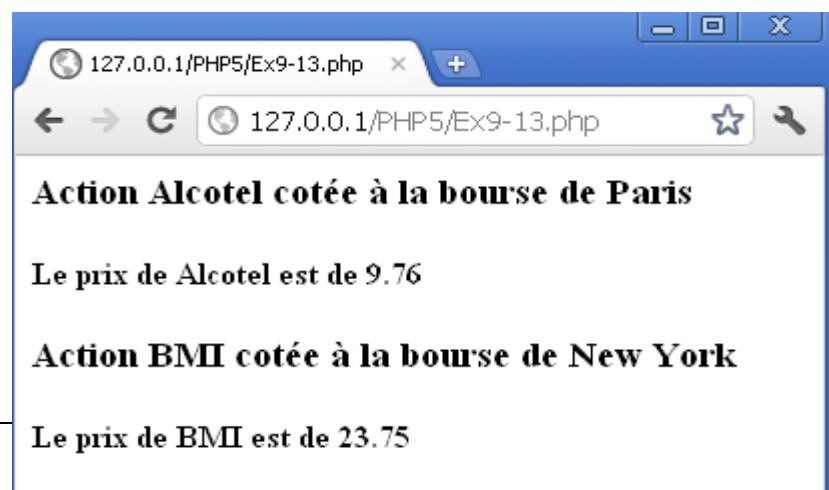
#### Exemple 11. Dérivation de classe abstraite

```
<?php
//Classe abstraite valeur
abstract class valeur
{
    protected $nom;
    protected $prix;
    abstract protected function __construct() ;
    abstract protected function getinfo() ;
}

//Classe action
class action extends valeur
{
    private $bourse;
    function __construct($nom,$prix,$bourse="Paris")
    {
        $this->nom=$nom;
        $this->prix=$prix;
        $this->bourse=$bourse;
    }
    public function getinfo()
    {
        $info("<h3>Action $this->nom cotée à la bourse de $this->bourse </h3>");
        $info.="("<h4>Le prix de $this->nom est de $this->prix </h4>");
        return $info;
    }
}

$action1 = new action("Alcotel",9.76);
echo $action1->getinfo();
$action2 = new action("BMI",23.75,"New York");
echo $action2->getinfo();
?>
```

Le résultat obtenu est identique à celui de l'exemple 10.



### III. Clonage d'objet

La notion de clonage d'objet permet d'effectuer une copie exacte d'un objet mais en lui affectant une zone de mémoire différente de celle de l'objet original. Contrairement à la création d'une simple copie à l'aide de l'opérateur = ou d'une référence sur un objet avec l'opérateur &, les modifications opérées sur l'objet cloné ne sont pas répercutées sur l'original.

Pour cloner un objet, utilisez le mot-clé clone selon la syntaxe suivante :

```
$objetclone = clone $objet ;
```

L'objet cloné a exactement les mêmes propriétés et les mêmes méthodes que l'original.

Après le clonage, les modifications opérées sur la variable \$objet n'ont aucun effet sur le clone, et réciproquement.

L'exemple 13 fait apparaître les différences entre une simple copie, une référence et un clonage. Vous y définissez une classe action ayant une seule propriété et un constructeur, un destructeur, qui affiche un message après la destruction de l'objet.

Pour vérifier les différences de comportement des copies et des clones, vous créez un objet \$alcotel, son clone dans la variable \$clone et une copie et une référence de \$alcotel, respectivement dans \$bim et \$ref. La modification de la propriété nom de l'objet \$bim est répercutée dans les objets \$alcotel et \$ref, comme le prouve l'affichage de leur propriété nom.

L'affichage des caractéristiques des objets \$alcotel, \$bim et \$ref fournit le même résultat (Object id #1), qui montre que ces trois objets font bien référence au même espace mémoire. En revanche, le même affichage pour l'objet \$clone fournit un résultat différent : Object id #2.

La destruction du clone entraîne immédiatement l'appel du destructeur, alors que la destruction successive des autres objets montre que le destructeur n'est appelé qu'après la destruction de la dernière référence à l'objet \$alcotel.

#### Exemple 13. Copie et clonage

```
<?php
//Définition de la classe action
class action
{
public $nom;
function __construct($nom)
{
$this->nom=$nom;
}
function __destruct()
{
echo "L'action $this->nom n'existe plus!<br />";
}
}
//Création d'objets
$alcotel = new action("Alcotel",10.21);
$clone= clone $alcotel;
$bim=$alcotel;
$ref=&$bim;

//Modification d'une propriété
$bim->nom="BIM";
```

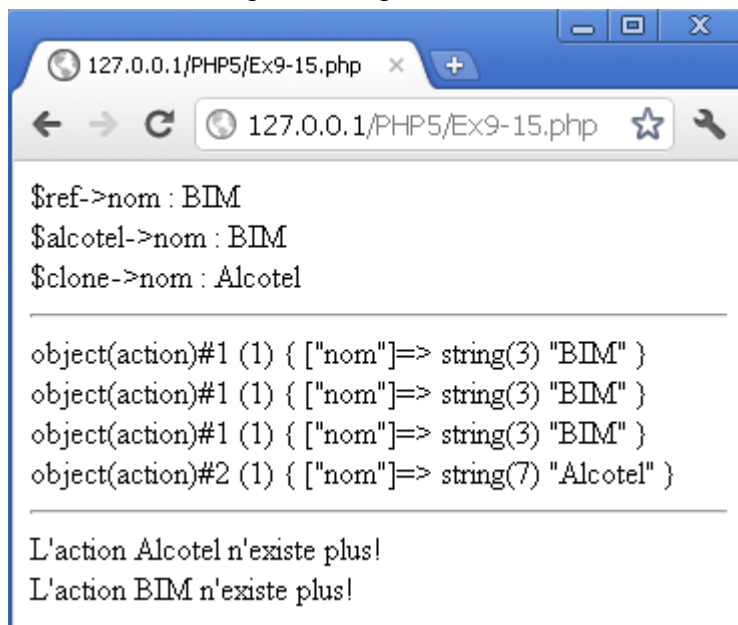
```

echo "\$ref->nom : ".$ref->nom , "<BR>";
echo "\$alcotel->nom : ".$alcotel->nom , "<BR>";
echo "\$clone->nom : ".$clone->nom , "<hr />";
//Affichage des caractéristiques des objets
var_dump($alcotel);echo "<BR>";
var_dump($bim);echo "<BR>";
var_dump($ref);echo "<BR>";
var_dump($clone);echo "<hr>";
//Suppression des objets
unset($clone);
unset($alcotel);
unset($bim);
unset($ref);

?>

```

Le résultat affiché par ce script est le suivant :



Le clonage d'objet peut donc être utile pour préserver l'état d'un objet hors de toute modification intempestive.