La programmation orientée objet (POO)

9

PLAN

9.1 Introduction
9.2 Principe
9.3 Notions de base
9.4 Notions avancées

OBJECTIFS

- Maîtriser la programmation orientée objet et ses caractéristiques.

9.1 Introduction

Le langage PHP est un langage hybride, à la fois procédural et objet. Les chapitres précédents utilisent la syntaxe procédurale afin de laisser la priorité à l'apprentissage des structures de données et à leurs traitements. Avec ce type de programmation, les données sont séparées des traitements, souvent regroupés en fonctions ou procédures.

Ce chapitre porte sur la *Programmation orientée objet* ou POO, pleinement disponible depuis PHP 5, qui constitue une nouvelle approche de la programmation en introduisant les notions de *classe* et d'*objet*.

Les données et les traitements abordés dans les chapitres précédents restent inchangés, seule l'architecture du programme est modifiée et induit de nouvelles fonctionnalités telles que l'*héritage* ou les *exceptions*, modifiant la manière de « penser » le programme.

Le langage PHP propose de nombreuses extensions basées sur la programmation objet. C'est par exemple PDO (*PHP Data Objects*), utilisé dans le chapitre suivant, qui fournit des méthodes d'accès aux bases de données.

9.2 Principe

Définitions

En programmation orientée objet, un *objet* est un ensemble « autonome » contenant des données, les *propriétés*, et des traitements appelés *méthodes*. Chaque objet est créé dynamiquement par l'instruction *new* à partir de la *classe* (modèle de l'objet) dont il est l'*instance*.

Notion de classe et d'objet

Pour comprendre ces notions, prenons un exemple de la vie de tous les jours, celui d'un ordinateur. C'est un objet au sens commun du terme.

Il possède un processeur, une mémoire vive, un disque dur et des périphériques tels qu'un écran, un clavier et une souris. Ce sont ses *propriétés*.

Cet ordinateur permet des traitements tels que : la saisie de données sur son clavier ; la gestion d'éléments graphiques *via* la souris ; l'affichage d'informations sur son écran ; l'exécution d'un logiciel *via* son processeur ; la mémorisation de valeurs grâce à sa mémoire vive ; la sauvegarde de données sur son disque dur. Ces différents traitements qui s'appliquent aux seuls éléments de l'ordinateur sont ses *méthodes*. Elles n'ont aucun sens pour un autre objet comme une imprimante, elles sont spécifiques à la notion d'ordinateur et participent à sa définition.

Ces deux notions, les *propriétés* (éléments de l'ordinateur) et les *méthodes* (traitements propres à l'ordinateur), caractérisent un ordinateur au sens large, c'est la *classe* de l'objet. En effet, il s'agit là d'une description d'un ordinateur que l'on peut écrire sur une feuille de papier, ce n'est pas l'ordinateur lui-même.

À partir de cette description ou de ce modèle, qui est la classe, on peut construire un vrai ordinateur de bureau qui est l'*objet*. C'est la création, représentée par l'instruction *new*, conçue à partir du modèle. On dit que l'ordinateur de bureau qui est l'objet, est l'*instance* de la classe « ordinateur ». À partir d'une même classe, on peut créer de nombreux ordinateurs de bureau. Si tous possèdent les mêmes caractéristiques, chaque élément d'un ordinateur n'agit que sur son ordinateur, c'est la notion de *portée des propriétés*. Ainsi la saisie de données sur le clavier d'un ordinateur n'a aucun effet sur l'ordinateur d'à côté. De plus, d'autres éléments sont totalement invisibles, voire inaccessibles, de l'extérieur comme le processeur. C'est la notion d'*encapsulation*.

On peut également concevoir d'autres types d'ordinateurs, d'autres classes, basés sur la description initiale, avec des éléments et des traitements supplémentaires, c'est la notion d'*héritage*. C'est par exemple la classe « ordinateur portable », qui est un ordinateur, mais possédant une caméra intégrée. Cette nouvelle classe hérite de la classe générique « ordinateur », et possède un nouvel élément ou propriété, la caméra, et de nouveaux traitements ou méthodes, filmer ou effectuer des

conférences. Pour décrire cette nouvelle classe « ordinateur portable », il suffit de rappeler que c'est un ordinateur et qu'en plus il possède une caméra et les fonctionnalités associées. On ne détaille que ces nouvelles propriétés et ces nouvelles méthodes, les autres propriétés et méthodes liées à la classe « ordinateur », sont héritées.

Enfin, cette classe « ordinateur portable » peut modifier localement certaines caractéristiques de la classe dont il hérite. Par exemple, la souris est remplacée par un pavé tactile. Les actions de déplacer la souris sont remplacées par le glisser du doigt sur la surface. Les autres actions de la souris, comme le clic, restent inchangées et s'appliquent au pavé tactile. C'est la notion de *surcharge*.

9.3 Notions de base

Cette section présente les syntaxes de base d'une programmation objet. Le contexte qui sert de support aux exemples est celui de la gestion d'une personne dont les informations sont : son nom, son prénom et son âge.

La classe

Le mot-clef *class* définit une nouvelle classe. Il est suivi par le nom de la classe et d'une paire d'accolades. Le nom de la classe commence par une lettre ou un souligné suivi par un nombre quelconque de lettres majuscules ou minuscules, de chiffres et du souligné.

Une classe peut contenir des *constantes*, des *variables* appelées « *propriétés* » ou « *attributs* », des *fonctions* appelées « *méthodes* ».

```
class Personnel
{
...
} // Fin de la classe
```

Les constantes de classe

Syntaxe

Les constantes sont définies à l'intérieur d'une classe *via* le mot-clef *const*. Leur contenu ne peut être qu'une valeur constante, en aucun cas une variable, le résultat d'un calcul ou de l'appel d'une fonction. Voici un exemple de syntaxe :

```
class Personnel
{  const AGE_MIN = 14 ;  // Déclaration de deux constantes
   const AGE_MAX = 65 ;
   ...
}
```

Les constantes magiques

Le langage PHP fournit un grand nombre de constantes prédéfinies dont certaines appelées *constantes magiques*. Elles n'ont pas besoin d'être définies, leur accès dépend des extensions disponibles ou non dans l'environnement PHP (compilées avec PHP ou chargées dynamiquement). Leur nom est préfixé et suffixé par deux caractères soulignés. En voici quelques-unes :

- `__LINE__` : le numéro de ligne courante dans le fichier PHP ;
- `__DIR__` : le dossier où se trouve le fichier PHP ;
- `__FILE__` : le fichier PHP ;
- `__FUNCTION__` : le nom de la fonction ;
- `__CLASS__` : le nom de la classe courante ;
- `__TRAIT__` : le nom du trait ;
- `__METHOD__` : le nom de la méthode ;
- `__NAMESPACE__` : le nom de l'espace de noms ;

Les propriétés

Principe

Les variables d'une classe se nomment *propriétés* ou *attributs*. On trouve aussi parfois les termes *membres* ou *champs*. Seule leur déclaration et leur visibilité changent par rapport à la programmation procédurale.

Déclaration et utilisation

Les propriétés sont définies *via* les mots-clefs *public*, *protected* ou *private* suivis d'une déclaration classique de variable. Selon le mot-clef utilisé, la propriété (la variable) aura une *visibilité* différente. Par exemple, elle pourra être totalement invisible à l'extérieur de l'objet. Le programme `propriete_declaration_objet_shell.php` présente cette notion. Trois propriétés sont déclarées : `$_nom`, `$prenom`, `$age`, respectivement privée, protégée et publique. Pour accéder à la propriété, on utilise l'opérateur « `->` » suivi de son nom sans le caractère « `$` », par exemple `$this->prenom`.

Listing 9.1 – Programme propriete_declaration_objet_shell.php

```
<?php
 class Personnel
 { // -- Les propriétés --
   private   $_nom="Dupont"  ;
   protected $prenom="Pierre";
   public    $age=55         ;
   // -- Affichage d'une personne --
   public function AffichePersonne()
```

```
    { echo "Personne : ";
      echo $this->_nom."\t".$this->prenom."\t".$this->age;
      echo PHP_EOL ;
    }
  }
  // === Programme ===
  $une_personne = new Personnel();
  $une_personne->AffichePersonne();
 ?>
```

Voici son exécution :

Listing 9.2 – Exécution de propriete_declaration_objet_shell.php

```
$ php propriete_declaration_objet_shell.php
Personne : Dupont   Pierre   55
```

Remarque

Par convention, et en respect de la notation PEAR, les noms des propriétés privées (*private*) sont préfixés par un souligné « _ », comme pour `$_nom`.

Il existe également le mot-clef *var* équivalent à *public* et considéré un temps comme obsolète.

Le mot-clef *static* déclare des propriétés ou des méthodes accessibles sans avoir besoin d'instancier la classe (*via new*). Par défaut, une propriété statique est publique. Voici le programme `propriete_statique_objet_shell.php`.

Listing 9.3 – Programme propriete_statique_objet_shell.php

```
<?php
 class PropStat
 { public static $NbPersonnes=0;
 }
 // === Programme ===
 echo "Valeur : ".PropStat::$NbPersonnes.PHP_EOL;
 PropStat::$NbPersonnes=100;
 echo "Valeur : ".PropStat::$NbPersonnes.PHP_EOL;
?>
```

Voici son exécution :

Listing 9.4 – Exécution de propriete_statique_objet_shell.php

```
$ php propriete_statique_objet_shell.php
Valeur : 0
Valeur : 100
```

Remarque

La propriété statique est accessible, sans création d'objet. Il est nécessaire d'indiquer son contexte via l'opérateur de résolution de portée « :: », comme : `PropStat::$NbPersonnes`.

La pseudo-variable `$this` est l'objet en cours d'utilisation. Cette pseudo-variable n'est pas applicable pour une propriété statique qui est par définition accessible sans qu'un objet soit créé. Cette syntaxe est utilisée dans le programme `propriete_declaration_objet_shell.php` présenté précédemment.

```
class Personnel
{  ...
   public function AffichePersonne()
   { echo "Personne : ";
     echo $this->_nom."\t".$this->_prenom."\t".$this->age;
     echo PHP_EOL ;
   }
}
```

Les propriétés et méthodes accessibles dès lors que l'objet est créé se nomment des *membres d'instance*. Les propriétés et méthodes statiques qui sont accessibles indépendamment de la création d'un objet, sans instancier la classe, se nomment des *membres de classe* ou des *membres statiques*.

Visibilité

Les mots-clefs *public*, *private* ou *protected* induisent la visibilité de la propriété à l'intérieur et à l'extérieur de la classe :

- *public* : la propriété (ou la méthode) publique est accessible depuis n'importe quel contexte, aussi bien à l'intérieur qu'à l'extérieur de sa classe ;
- *private* : la propriété (ou la méthode) privée n'est accessible que depuis la classe où elle est définie. C'est la notion d'*encapsulation* ;
- *protected :* la propriété (ou la méthode) protégée est accessible depuis la classe où elle est définie, les classes qui en héritent, et aux classes parentes.

Le programme `proprietes_visibilites_objet_shell.php` présente la déclaration et la visibilité des propriétés.

Listing 9.5 – Programme proprietes_visibilites_objet_shell.php

```
<?php
 class Personnel
 { public  $matricule = 112233 ; // Les propriétés
   private $_nom      = "Kasma";
   public function AffichePersonne()
   { echo "matricule : ".$this->matricule."  ";
     echo "_nom : ".$this->_nom."  "          ;
     echo "maladies: ".$this->maladies."  "   ;
     echo PHP_EOL ;
   }
 }
 class Medecin extends Personnel
 { protected $maladies="Asthme"  ; // Propriétés
```

```
    public function AffichePatient()
    { echo "matricule  : ".$this->matricule."  ";
      echo "_nom : ".$this->_nom."  "           ;
      echo "maladies: ".$this->maladies."  "    ;
      echo PHP_EOL ;
    }
  }
  // === Programme ===
  $salarie = new Personnel()   ;
  $docteur = new Medecin()     ;
  echo "-- Salarié  --".PHP_EOL;
  $salarie->AffichePersonne()  ;
  $salarie->matricule=998877   ;// Accès en dehors de la classe
  echo "Nouveau matricule : ".$salarie->matricule.PHP_EOL;
  //echo "Nom : ".$salarie->_nom.PHP_EOL; // Erreur
  echo "-- Docteur --".PHP_EOL ;
  $docteur->AffichePatient()   ;
?>
```

Voici son exécution. La propriété `$_nom` (*private*) est invisible pour l'objet `$docteur`, et la propriété `$maladies` (*protected*) est invisible pour l'objet `$salarie`. La propriété `$matricule` (*public*) est accessible dans le programme.

Listing 9.6 – Exécution de proprietes_visibilites_objet_shell.php

```
$ php proprietes_visibilites_objet_shell.php
-- Salarié  --
matricule : 112233  _nom : Kasma  maladies:
Nouveau matricule : 998877
-- Docteur --
matricule  : 112233  _nom :   maladies : Asthme
```

Les méthodes

Les méthodes sont les fonctions internes à une classe. Elles suivent les mêmes règles syntaxiques que celles édictées en programmation procédurale et sont déclarées *via* le mot-clef `function`. Leur visibilité en dehors de leur classe de définition dépend de l'usage des préfixes *public*, *protected*, *private*. Dans le programme `proprietes_visibilites_objet_shell.php`, la méthode `AffichePersonne()` est publique. Elle est visible du programme principal. Son appel dans un programme ou une autre fonction utilise l'opérateur « `->` ».

```
// === Classe Personnel ===
 class Personnel
 { ...
   public function AffichePersonne()
   { ... }
 }
 ...
```

```
// === Programme ===
$salarie = new Personnel()   ;
$salarie->AffichePersonne()  ;
...
```

Les constructeurs et les destructeurs

PHP propose deux méthodes facultatives `__construct()` et `__destruct()` (deux soulignés). La première est le constructeur. Elle est appelée automatiquement lors de la création d'un nouvel objet (*via* `new`). Elle sert à initialiser le contexte d'usage de l'objet. La seconde est le destructeur. Elle est appelée dès qu'il n'existe plus aucune référence à l'objet, par exemple lorsque l'objet est supprimé *via* `unset()`. Dans le premier exemple, le constructeur ne possède aucun paramètre.

```
class Personnel
 {  ...
   private $_nom   ; // -- Les propriétés --
   private $_prenom;
   private $_age   ;
   function __construct()// -- Constructeur --
   { $this->_nom    = "" ;
     $this->_prenom = "" ;
     $this->age     = 0  ;
   }
   function __destruct() // -- Destructeur --
   { unset ($this->_nom)   ;
     unset ($this->_prenom);
     unset ($this->_age)   ;
   }
   ...
 }
```

L'instruction `new` appelle le constructeur, `unset()` appelle le destructeur.

```
$une_personne = new Personnel(); // Appel du constructeur
...
unset ($une_personne); // Appel du destructeur
```

Dans le second exemple, le constructeur possède trois paramètres.

```
function __construct($n,$p,$a) // -- Constructeur --
{ $this->_nom    = $n ;
  $this->_prenom = $p ;
  $this->age     = $a ;
}
```

L'instruction `new` appelle le constructeur avec les données initiales :

```
$une_personne = new Personnel("dupont","jean",45);
```

Remarque

Dans le cas de l'héritage entre classes, une classe enfant hérite du constructeur ou du destructeur de sa classe parent si elle n'en possède pas. Cependant, l'appel du constructeur ou du destructeur de la classe parent n'est pas automatique et doit être explicite. La syntaxe de l'appel du constructeur parent est `parent::__construct()`.

L'objet

Création

L'objet est l'instance d'une classe qui en est le modèle. La déclaration de classe ne crée aucun objet. C'est l'instruction `new` qui effectue cette opération. Par exemple :

```
$salarie = new Personnel()   ;
```

Accès aux propriétés et méthodes

- *Publiques*

L'accès aux propriétés et méthodes d'un objet utilise l'opérateur « `->` ». Il dépend de leur visibilité. Les syntaxes suivantes sont basées sur le programme `proprietes_visibilites_objet_shell.php`.

```
$salarie->matricule=998877 ; // Accès à la propriété publique
echo $salarie->_nom ;    // Accès interdit (private) : Erreur
$salarie->AffichePersonne(); // Accès à la méthode publique
```

Si la propriété est publique, son accès en lecture et modification est direct. Avec les propriétés privées, il faut définir des méthodes spécifiques afin d'en contrôler l'accès et ainsi garantir leur protection *via* le mécanisme d'encapsulation.

- *Privées*

La méthode qui retourne la valeur d'une propriété privée s'appelle *accesseur* ou *getter*. La méthode qui modifie la valeur d'une propriété privée s'appelle *mutateur* ou *setter*. Il est possible d'utiliser des méthodes spécifiques à chaque propriété ou d'utiliser les méthodes magiques `__set()` et `__get()` (deux soulignés).

Par propriété

Dans le cas où chaque propriété possède son propre accesseur et son propre mutateur, l'usage est d'utiliser comme nom `getPropriété()` et `setPropriété()`, où « `Propriété` » est le nom de la propriété. Le programme `getter_setter_objet_shell.php` en présente la syntaxe :

Listing 9.7 – Programme getter_setter_objet_shell.php

```
<?php
 class Personnel
 { const AGE_MIN = 14 ;  // Déclaration de deux constantes
   const AGE_MAX = 65 ;
   private $_nom=""   ; // Les propriétés
   private $_prenom="";
   private $_age=0    ;
   // -- Getter --
   public function getNom() {return $this->_nom;}
   public function getPrenom() {return $this->_prenom;}
   public function getAge() {return $this->_age;}
   // -- Setter --
   public function setNom($n) {$this->_nom=$n;}
   public function setPrenom($p) {$this->_prenom=$p;}
   public function setAge($a) {
     if (($a>=self::AGE_MIN)&&($a<self::AGE_MAX))
     {$this->_age=intval($a);}
   }
 }
 // === Programme ===
 $une_personne = new Personnel() ;
 $une_personne->setNom("dupont") ;
 $une_personne->setPrenom("jean");
 $une_personne->setAge("25")     ;
 echo $une_personne->getNom()." ";
 echo $une_personne->getPrenom()." ";
 echo $une_personne->getAge().PHP_EOL;
?>
```

Voici son exécution :

Listing 9.8 – Exécution de getter_setter_objet_shell.php

```
$ php getter_setter_objet_shell.php
dupont jean 25
```

La définition de méthodes accesseurs et mutateurs pour chaque propriété trouve vite ses limites. Avec un grand nombre de propriétés, la liste des méthodes accesseurs et mutateurs peut devenir importante. De plus, la création de variables en PHP est dynamique. Par conséquent, il est possible à un programme de créer « à la volée » une nouvelle propriété sans qu'elle soit déclarée dans la classe. Si l'on ajoute les lignes suivantes à la fin du programme précédent :

```
// Accès à une propriété inexistante
 $une_personne->telephone="0143451122" ;
 echo $une_personne->telephone.PHP_EOL;
```

L'exécution montre que la propriété « telephone » a été créée.

```
$ php getter_setter2_objet_shell.php
dupont jean 25
0143451122
```

Ceci va à l'encontre de la notion d'encapsulation propre à la programmation objet car le programme interagit directement avec la classe. Les méthodes magiques __get() et __set() résolvent ce problème.

Les méthodes magiques *__get()* et *__set()*

Le langage PHP propose un certain nombre de *méthodes magiques* préfixées par deux soulignés, dont le comportement est prédéfini. La redéfinition explicite de ces méthodes dans la classe réalise des traitements appelés automatiquement.

La méthode __get()est automatiquement appelée lors d'une tentative de lecture d'une propriété privée. De même __set()est automatiquement appelée lors d'une tentative de modification d'une propriété privée. Le programme get_set_magique_objet_shell.php présente ces syntaxes.

Listing 9.9 – Programme get_set_magique_objet_shell.php

```
<?php
 class Personnel
 { const AGE_MIN = 14 ; // Déclaration de deux constantes
   const AGE_MAX = 65 ;
   private $_nom=""   ; // Les propriétés
   private $_prenom="";
   private $_age=0    ;
   // -- __get et __set méthodes magiques -
   public function __get($propriete) {
            if (property_exists($this, $propriete)) {
                return $this->$propriete;
            }
   }
   public function __set($propriete, $valeur) {
     if (property_exists($this, $propriete)) {
       if ($propriete=="age") {
         if (($valeur>=self::AGE_MIN)&&($valeur<self::AGE_MAX))
           $this->_age=intval(trim($valeur));
       }
       else {
           $this->$propriete = strtoupper(trim($valeur));
       }
     }
   }
 }
 // === Programme ===
 $une_personne = new Personnel() ;
 $une_personne->_nom="dupont" ;
 $une_personne->_prenom="jean";
 $une_personne->_age="25"     ;
```

```
 echo $une_personne->_nom." ";
 echo $une_personne->_prenom." ";
 echo $une_personne->_age.PHP_EOL;
 // Aucun accès possible à une propriété inexistante
 $une_personne->Telephone="0143451122" ;
 echo $une_personne->Telephone.PHP_EOL;
?>
```

Son exécution montre qu'il est impossible d'ajouter « Telephone » à la classe.

Listing 9.10 – Exécution de get_set_magique_objet_shell.php

```
$ php get_set_magique_objet_shell.php
DUPONT JEAN 25
```

Les fonctions sur les classes et les objets

Le tableau 9.1 présente les fonctions sur les classes et les objets. Certaines d'entre elles sont détaillées dans les sections suivantes. Cette liste est disponible à l'URL : *http://php.net/manual/fr/ref.classobj.php*.

Tableau 9.1 – Fonctions sur les classes et les objets

Fonction	Signification	Exemple
`__autoload`	Prépare le chargement dynamique d'une classe	`function __autoload($NC) { require $NC.'_Class.php';}`
`class_alias`	Crée un alias de classe	`class_alias('Cadre','CD');`
`class_exists`	Vrai si la classe est définie	`$CE=class_exists('Cadre');`
`class_implements`	Retourne les interfaces implémentées par une classe ou une interface	`$I=class_implements('En');`
`get_called_class`	Retourne le nom de la classe depuis laquelle une méthode statique a été appelée	`class Cadre { static public function NEntrp() { $c=get_called_class(); return $c; } } $Classe=Cadre::NEntrp();`
`get_class_methods`	Retourne la liste des méthodes d'une classe	`$tabM=get_class_methods('Cadre');`
`get_class_vars`	Valeurs par défaut des propriétés d'une classe	`$tabV=get_class_vars('Cadre');`
`get_class`	Classe d'un objet	`$NomC=get_class($UnCadre);`

Fonction	Signification	Exemple
get_declared_classes	Liste des classes définies dans PHP	$TDC=get_declared_classes();
get_declared_interfaces	Liste des interfaces définies dans PHP	$TDI=get_declared_interfaces();
get_declared_traits	Liste des traits définis dans PHP	$TDT=get_declared_traits();
get_object_vars	Retourne les propriétés non statiques d'un objet	$TabV=get_object_vars($UnCadre);
get_parent_class	Retourne le nom de la classe parente d'un objet	$PClasse=get_parent_class($UnCadre);
interface_exists	Retourne vrai si l'interface est définie	if (interface_exists('MonInterface')) {...}
is_a	Vrai si l'objet est une instance d'une classe ou a cette classe en parent	if(is_a($UnCadre,'Personnel')) {...}
is_object	Retourne vrai si la variable est de type objet	if(is_object($UnCadre) {...}
is_subclass_of	Vrai si un objet est une sous-classe d'une classe	$EstSC=is_subclass_of($UnCadre,'Personnel');
method_exists	Vrai si la méthode existe pour une classe	$ME=method_exists($UnCadre,'Affiche');
property_exists	Vrai si l'objet ou la classe possède une propriété	$PE=property_exists('Cadre','_tel');
trait_exists	Vrai si un trait existe	$ET=trait_exists('EntrT');

9.4 Notions avancées

Le chargement dynamique de classes

Principe

Dans les exemples précédents, la classe est définie dans le même fichier que l'objet et le programme l'utilisant. Si le programme devient imposant, il est nécessaire de répartir les divers composants du programme dans différents fichiers .php, comme cela est présenté au chapitre 8 sur les fonctions.

La programmation objet ne déroge pas à cette organisation et nombre de développeurs définissent un fichier .php par classe, ce qui implique l'ajout explicite de nombreuses syntaxes `include` ou `require` précédant la création d'objet. Cela

peut être évité depuis la version 5 de PHP qui autorise le chargement dynamique de classes *via* les fonctions `__autoload()` ou `spl_autoload_register()`.

Syntaxe

Le programme précédent `get_set_magique_objet_shell.php` a été séparé en deux fichiers. Le premier `Personnel_Classe.php` contient la définition de la classe `Personnel`, le second `chargement_classe1_objet_shell.php` contient le chargement dynamique de la classe, déclenché par la création d'un nouvel objet. Les lignes identiques au programme précédent sont remplacées par des « … ».

Listing 9.11 – Programme Personnel_Classe.php

```
<?php
 class Personnel
 {  ...
   private $_nom=""   ; // -- Les propriétés --
   private $_prenom="";
   private $_age=0    ;
   // -- __get et __set méthodes magiques --
   public function __get($propriete) { ... }
   public function __set($propriete, $valeur) { ... }
 }
?>
```

La fonction `__autoload()` ajoute la classe fournie en argument à la pile de chargement dynamique. Elle contient une instruction `require` appliquée au fichier dont le nom est le texte de la variable `$NomClasse`, suivi de `_Classe.php`.

Listing 9.12 – Programme chargement_classe1_objet_shell.php

```
<?php
 function __autoload($NomClasse) {
    require $NomClasse . '_Classe.php';
 }
 // === Programme ===
 $une_personne = new Personnel() ; // Chargement dynamique
 ...
?>
```

Avec la fonction `spl_autoload_register()`le programme précédent devient :

Listing 9.13 – Programme chargement_classe2_objet_shell.php

```
<?php
 function chargeur_classe($NomClasse) {
    require $NomClasse . '_Classe.php';
```

```
  }
  spl_autoload_register('chargeur_classe');
  // === Programme ===
  $une_personne = new Personnel() ; // Chargement dynamique
  ...
?>
```

L'héritage

Principe

L'héritage est un grand principe de la programmation objet. C'est la définition d'une nouvelle classe, la *classe fille* ou *enfant*, à partir d'une classe existante, la *classe mère* ou *parent*. La nouvelle classe hérite des propriétés et des méthodes publiques et protégées de son *parent*. Sa mise en œuvre utilise le mot-clef *extends*.

Ainsi les propriétés et méthodes communes à plusieurs classes peuvent être regroupées dans une classe unique. Ces autres classes héritent de cette classe « commune », elles ne contiennent que la définition de leurs propres propriétés et méthodes. L'héritage ne s'applique qu'aux classes connues. La classe parent doit être définie avant la classe enfant, sauf en cas de chargement dynamique de classe.

Mise en œuvre

Le programme `heritage1_objet_shell.php` présente deux classes. La classe `Cadre` hérite de la classe `Personnel`. L'objet `$un_cadre` est créé. Sa classe hérite, de la classe `Personnel`, donc des méthodes `getNom(), getPrenom(), setNom(), setPrenom()`qui sont `protected` mais pas des propriétés `$_nom` et `$_prenom` qui sont `private`. Cet objet possède la propriété `$_tel` et les méthodes publiques `ModifieCadre()` et `AfficheCadre()`.

Listing 9.14 – Programme heritage1_objet_shell.php

```
<?php
 class Personnel
 { private $_nom=""    ; // -- Les propriétés --
   private $_prenom="" ;
   // -- méthodes --
   protected function getNom() {return $this->_nom;}
   protected function getPrenom() {return $this->_prenom;}
   protected function setNom($v) {
       $this->_nom=strtoupper(trim($v));}
   protected function setPrenom($v) {
       $this->_prenom=strtoupper(trim($v));}
 }
 class Cadre extends Personnel
 { private $_tel=""   ; // -- Propriété --
   // -- méthodes --
   public function ModifieCadre($n,$p,$t) {
```

```
    $this->setNom($n)    ;
    $this->setPrenom($p) ;
    $this->_tel=trim($t) ;
   }
   public function AfficheCadre() {
    echo $this->getNom()." "    ;
    echo $this->getPrenom()." " ;
    echo $this->_tel.PHP_EOL    ;
   }
 }
 // === Programme ===
 $un_cadre = new Cadre()   ;
 $un_cadre->ModifieCadre("dupont","jean","0143451122");
 $un_cadre->AfficheCadre() ;
?>
```

Voici son exécution :

Listing 9.15 – Exécution de heritage1_objet_shell.php

```
$ php heritage1_objet_shell.php
DUPONT JEAN 0143451122
```

La réécriture ou « surcharge »

La classe fille peut *réécrire* une méthode de la classe mère. Cette notion s'apparente à la *surcharge*. Elle s'applique aux propriétés et aux méthodes.

En comparaison avec d'autres langages objets, le terme surcharge n'est pas exact et le terme réécriture est plus approprié. Dans les autres langages, la surcharge permet d'écrire plusieurs fois la même méthode **dans la même classe** avec un nombre différent de paramètres, proposant ainsi un traitement différent selon les paramètres d'appel. Cela s'applique typiquement au constructeur. PHP ne le permet pas.

Dans le programme `surcharge1_objet_shell.php` la méthode `Affiche()` de la classe mère `Personnel` affiche le nom et le prénom. Elle est récrite dans la classe fille `Cadre` pour afficher le téléphone également. Le constructeur de la classe `Cadre` appelle explicitement le constructeur de son parent.

Listing 9.16 – Programme surcharge1_objet_shell.php

```
<?php
 class Personnel
 { protected $nom    ; // -- Les propriétés --
   protected $prenom ;
   function __construct($n,$p) // -- Constructeur --
   { $this->nom    = strtoupper(trim($n)) ;
     $this->prenom = strtoupper(trim($p)) ;
   }
   public function Affiche() {
```

```
    echo "Nom: ".$this->nom;
    echo "  -  Prénom: ".$this->prenom.PHP_EOL;
  }
}
class Cadre extends Personnel
{ private $_tel   ; // -- Propriété --
  function __construct($n,$p,$t) // -- Constructeur --
  { parent::__construct($n,$p);//Appel du constructeur parent
    $this->_tel   = trim($t) ;
  }
  public function Affiche() {  // surcharge
    echo "Nom: ".$this->nom;
    echo "  -  Prénom: ".$this->prenom;
    echo "  -  Tel: ".$this->_tel.PHP_EOL;
  }
}
// === Programme ===
$un_salarie = new Personnel("martin","pierre") ;
$un_salarie->Affiche() ; //Méthode de la Classe Personnel
$un_cadre = new Cadre("dupont","jean","0143451122")   ;
$un_cadre->Affiche() ; // Méthode de la Classe Cadre
?>
```

Voici son exécution :

Listing 9.17 – Exécution de surcharge1_objet_shell.php

```
$ php surcharge1_objet_shell.php
Nom: MARTIN  -  Prénom: PIERRE
Nom: DUPONT  -  Prénom: JEAN  -  Tel: 0143451122
```

L'opérateur de résolution de portée ::

- *Principe*

L'opérateur de résolution de portée « :: » appelé « Paamayim Nekudotayim » (son nom en hébreu) donne accès aux éléments statiques ou constants, ainsi qu'aux propriétés ou méthodes surchargées d'une classe. Il faut indiquer le nom de la classe avant les deux points ou les mots-clefs *self* ou *parent*.

- *Accès aux constantes de classes*

Le programme `operateur_portee1_objet_shell.php` présente l'accès aux constantes de classe `AGE_MIN` et `AGE_MAX` à l'intérieur de la classe *via* le mot-clef *self*, et dans le programme avec le nom de la classe.

Listing 9.18 – Programme operateur_portee1_objet_shell.php

```
<?php
 class OpPortee
 { const AGE_MIN = 14 ; // Déclaration de deux constantes
```

```
    const AGE_MAX = 65 ;
    function __construct()// -- Constructeur --
    { echo "Creation de l'objet : "   ;
      echo "  AGE_MIN=".self::AGE_MIN ;
      echo "  AGE_MAX=".self::AGE_MAX.PHP_EOL;
    }
  }
  // === Programme ===
  $unOperateur = new OpPortee() ;
  echo "Dans le programme AGE_MIN=".OpPortee::AGE_MIN.PHP_EOL;
?>
```

- *Accès aux méthodes*

On peut également indiquer explicitement le contexte d'une méthode utilisée. Le programme `surcharge2_objet_shell.php` est une adaptation du programme `surcharge1_objet_shell.php`. La classe `Personnel` reste inchangée. La méthode `Affiche()` de la classe `Cadre` surcharge celle de la classe `Personnel`, en appelant la première *via* la syntaxe `parent::Affiche()`.

Listing 9.19 – Programme surcharge2_objet_shell.php

```
<?php
 class Personnel { ... } // === Classe Personnel ===
 class Cadre extends Personnel // === Classe Cadre ===
 { private $_tel   ; // -- Propriétés --
   function __construct($n,$p,$t) // -- Constructeur --
   { parent::__construct($n,$p); //constructeur parent
     $this->_tel   = trim($t) ;
   }
   public function Affiche() {  // surcharge
     echo "-- Dans la classe Cadre --".PHP_EOL;
     parent::Affiche();
     echo "  -  Tel: ".$this->_tel.PHP_EOL;
   }
 }
 // === Programme ===
 $un_salarie = new Personnel("martin","pierre") ;
 $un_salarie->Affiche() ; //Méthode de la Classe Personnel
 $un_cadre = new Cadre("dupont","jean","0143451122")   ;
 $un_cadre->Affiche() ; // Méthode de la Classe Cadre
?>
```

Dans cet exemple, la syntaxe :

```
parent::Affiche();
```

peut également se noter :

```
Personnel::Affiche();
```

Voici son exécution :

Listing 9.20 – Exécution de surcharge2_objet_shell.php

```
$ php surcharge2_objet_shell.php
Nom: MARTIN  -  Prénom: PIERRE
-- Dans la classe Cadre --
Nom: DUPONT  -  Prénom: JEAN
  -  Tel: 0143451122
```

- *Accès aux membres statiques*

Le mot-clef *static* déclare des propriétés ou des méthodes accessibles sans avoir besoin d'instancier la classe *via new*. On parle de *membres de classe* ou de *membre statiques* par opposition aux *membres d'instances* qui correspondent aux propriétés et méthodes accessibles uniquement *via* l'objet.

Le programme `operateur_portee2_objet_shell.php` présente la propriété `$NbPersonnes` et la méthode `Message()` déclarées en statique. `Message()` accède à la propriété *via self*. Le programme accède à la propriété `$NbPersonnes` et à la méthode `Message()` en faisant précéder l'opérateur « :: » par le nom de la classe.

Listing 9.21 – Programme operateur_portee2_objet_shell.php

```
<?php
 class OpPortee
 { public static $NbPersonnes=0;
   public static function Message(){
     echo "Nb personnes:".self::$NbPersonnes.PHP_EOL;
   }
 }
 // === Programme ===
 OpPortee::$NbPersonnes=100;
 echo "Valeur : ".OpPortee::$NbPersonnes.PHP_EOL;
 OpPortee::Message();
?>
```

L'opérateur instanceOf

L'héritage n'est pas limité à deux classes et peut aboutir à une *hiérarchie de classes* aussi appelée *hiérarchie de types*, soit un arbre généalogique. Par exemple, la classe `Directeur` hérite de la classe `Cadre` qui hérite de la classe `Personnel`. Un directeur est aussi un cadre qui appartient au personnel de l'entreprise. L'opérateur `instanceOf` retourne une valeur booléenne indiquant si l'objet est une instance de la classe précisée. Le programme `hierarchie_de_classe_objet_shell.php` présente cet opérateur. Les classes `Personnel` et `Cadre` sont identiques à celles du programme `surcharge2_objet_shell.php`, leur corps a été remplacé par des « … ».

Listing 9.22 – Programme hierarchie_de_classe_objet_shell.php

```
<?php
 class Personnel { ... }
 class Cadre extends Personnel { ... }
 class Directeur extends Cadre
 { private $_assistant   ; // -- Propriétés --
   function __construct($n,$p,$t,$a) // -- Constructeur --
   { parent::__construct($n,$p,$t);  //constructeur parent
     $this->_assistant   = trim($a) ;
   }
   public function Affiche() {  // surcharge
     parent::Affiche();
     echo "  -  Assistant: ".$this->_assistant.PHP_EOL;
   }
 }
 // === Programme ===
$un_directeur = new Directeur("lefevre","paul","0600112233","pignon") ;
 $un_directeur->Affiche() ;// Méthode de la Classe Directeur
 $est_instance_D = $un_directeur instanceOf Directeur ;
 $est_instance_C = $un_directeur instanceOf Cadre     ;
 $est_instance_P = $un_directeur instanceOf Personnel ;
 echo '$un_directeur'." instance de Directeur : ";
 var_dump($est_instance_D);
 echo '$un_directeur'." instance de Cadre : ";
 var_dump($est_instance_C);
 echo '$un_directeur'." instance de Personnel : ";
 var_dump($est_instance_P);
?>
```

Voici son exécution :

Listing 9.23 – Exécution de hierarchie_de_classe_objet_shell.php

```
$ php hierarchie_de_classe_objet_shell.php
Nom: LEFEVRE  -  Prénom: PAUL
  -  Tel: 0600112233
  -  Assistant: pignon
$un_directeur instance de Directeur : bool(true)
$un_directeur instance de Cadre : bool(true)
$un_directeur instance de Personnel : bool(true)
```

Le polymorphisme ou la généricité

Principe

Le *polymorphisme*, du grec « plusieurs formes », est le mécanisme par lequel une méthode peut être implémentée à travers plusieurs classes. C'est par exemple le même algorithme de tri implémenté par une classe de personnes ou une classe d'entiers. Cette notion se nomme également *généricité* car la méthode ainsi développée est générique. Le polymorphisme s'appuie sur les comportements prévus par

l'héritage et sur la notion d'*abstraction de classe*. Dans ce cas, le comportement d'une méthode de la classe mère n'est réellement connu que dans la classe fille. On parle de *classe abstraite*.

Les classes abstraites

• *Principe*

Une classe abstraite est définie par le mot-clé *abstract*. Par opposition aux classes concrètes présentées jusque-là, elle ne peut être instanciée pour créer un objet. Elle sert uniquement de modèle aux classes filles qui en héritent. Cette notion participe à la sécurité des développements en programmation objet en fournissant un « canevas » obligatoire aux classes filles.

Tout comme une classe concrète, une classe abstraite possède des propriétés et des méthodes dont la visibilité dépend des mots-clefs *private*, *protected* et *public*. Elle se différencie dans le fait qu'elle peut contenir des *méthodes abstraites* qui doivent obligatoirement être redéfinies (« surchargées ») dans la classe fille, imposant ainsi son comportement aux classes qui en héritent.

Une classe abstraite ne contient aucun constructeur ni destructeur puisqu'elle ne peut être instanciée.

• *Méthodes abstraites*

Une méthode abstraite est également définie par le mot-clé *abstract*. Seul son en-tête ou *signature* est défini dans la classe. Elle ne doit pas posséder de corps. Les classes filles doivent la redéfinir totalement avec la même visibilité ou une visibilité moins restrictive. En plus de l'en-tête qui doit être identique, elles doivent définir son corps donc les traitements à effectuer.

Toute classe possédant au moins une méthode abstraite est une classe abstraite.

• *Exemples*

Le programme `classe_abstraite_objet_shell.php` présente la classe abstraite `Personne`. Elle contient les propriétés `$nom` et `$prenom`, et les méthodes `setNom()`, `getNom()`, `setPrenom()`, `getPrenom()`. Elle possède la méthode abstraite `Affiche()` qui est vide. Les classes `Personnel` et `Client` héritent de la classe abstraite `Personne`. Elles doivent redéfinir la méthode `Affiche()`.

Listing 9.24 – Programme classe_abstraite_objet_shell.php

```
<?php
 // === Classe Abstraite Personne ===
 abstract class Personne
 { protected $nom    ; // -- Les propriétés --
   protected $prenom ;
   // getters et setters
```

```
  public function setNom($nom){$this->nom = $nom;}
  public function getNom(){return $this->nom;}
  public function setPrenom($nom){$this->prenom = $prenom;}
  public function getPrenom(){return $this->prenom;}
  // Méthode abtraite
  abstract function Affiche();
}
class Personnel extends Personne
{ protected $matricule   ; // -- Propriété --
  function __construct($m,$n,$p) // -- Constructeur --
  { $this->matricule = intval(trim($m))     ;
    $this->nom       = strtoupper(trim($n)) ;
    $this->prenom    = strtoupper(trim($p)) ;
  }
  public function Affiche() {
    echo "Nom: ".$this->nom;
    echo "  -  Prénom: ".$this->prenom;
    echo "  -  Matricule: ".$this->matricule.PHP_EOL;
  }
}
class Client extends Personne
{ protected $adresse   ; // -- Propriétés --
  function __construct($n,$p,$a) // -- Constructeur --
  { $this->nom       = strtoupper(trim($n)) ;
    $this->prenom    = strtoupper(trim($p)) ;
    $this->adresse   = strtoupper(trim($a)) ;
  }
  public function Affiche() {
    echo "Nom: ".$this->nom;
    echo "  -  Prénom: ".$this->prenom;
    echo "  -  Adresse: ".$this->adresse.PHP_EOL;
  }
}
// === Programme ===
$UnSalarie = new Personnel(324152,"martin","pierre") ;
echo "--- Salarie ---".PHP_EOL;
$UnSalarie->Affiche() ;//Méthode de la Classe Personnel
$UnClient = new Client("leroy","patrick","12 rue Ordener 75018 Paris") ;
echo "--- Client ---".PHP_EOL;
$UnClient->Affiche() ;//Méthode de la Classe Client
?>
```

Voici son exécution :

Listing 9.25 – Exécution de classe_abstraite_objet_shell.php

```
$ php classe_abstraite_objet_shell.php
--- Salarie ---
Nom: MARTIN  -  Prénom: PIERRE  -  Matricule: 324152
--- Client ---
Nom: LEROY  -  Prénom: PATRICK  -  Adresse: 12 RUE ORDENER 75018 PARIS
```

Si la méthode `Affiche()` n'est pas redéfinie dans la classe fille, par exemple `Personnel`, le message d'erreur suivant apparaît :

```
Fatal error: Class Personnel contains 1 abstract method ...
```

Les interfaces

- *Principe*

Quand toutes les méthodes d'une classe sont abstraites, cette classe devient une *interface*. Elle est définie par le mot-clef *interface*, et propose un ensemble de services visibles à l'extérieur regroupés sous le terme API (*Application Programming Interface*). Alors que les classes abstraites servent à organiser la programmation, les interfaces formalisent les méthodes qu'une classe doit implémenter. Alors qu'il n'est possible d'hériter que d'une seule classe parente, on peut hériter de plusieurs interfaces, en séparant chaque nom par une virgule. Une classe implémente une interface *via* le mot-clef *implements*.

Il ne faut pas confondre la notion d'héritage et d'interface. Par exemple, on peut formaliser les contrôles d'accès d'une entreprise *via* une interface, et cela s'applique aussi bien au personnel, aux clients, qu'aux camions de livraison. Alors que le personnel et le client sont des personnes et peuvent hériter des propriétés et des méthodes d'une classe `Personne`, il est évident qu'un camion n'est pas une personne et ne peut pas en hériter, alors que sa classe implémente pourtant les contrôles d'accès de l'entreprise *via* l'interface.

Les méthodes d'une interface sont publiques et toutes abstraites, le mot-clef *abstract* n'est donc plus utilisé. Une interface ne peut porter le nom d'une classe.

Comme pour les classes, une interface peut hériter d'une autre interface *via* le mot-clef *extends*. Mais il n'est pas possible de réécrire les méthodes héritées de l'interface mère dans l'interface fille. Contrairement à une classe, une interface peut hériter de plusieurs interfaces, en séparant chaque nom par une virgule.

La classe implémentant une interface doit utiliser les mêmes méthodes avec la même signature (entête).

Une interface peut contenir des constantes. Elles se comportent comme des constantes de classe. Néanmoins, elles ne peuvent pas être écrasées par une classe qui l'implémente ou une interface qui en hérite.

- *Exemple*

Le programme `interface1_objet_shell.php` présente la gestion du contrôle d'accès d'une entreprise pour ses salariés et pour ses livraisons *via* l'interface `ContEntreprise`. Il contient également l'interface `SetGetPersonne` de gestion des salariés. La classe `Personnel` implémente les deux interfaces. La classe `Livraison` n'implémente que l'interface `ContEntreprise`.

Listing 9.26 – Programme interface1_objet_shell.php

```
<?php
 // === Interface Contrôle d'accès d'une Entreprise ===
 interface ContEntreprise
 { // -- Les méthodes --
   public function setCont($id,$type,$date);
   public function getCont($id);
 }
 // === Interface pour getteur et setter d'une personne ===
 interface SetGetPersonne
 { // -- Les méthodes --
   public function setVar($id,$nom,$prenom);
   public function getVar($id);
 }
 // === Classe Personnel ===
 class Personnel implements ContEntreprise,SetGetPersonne
 { // -- Les propriétés --
   protected $TabVars = array();
   protected $TabCont = array();
   // Méthodes de l'interface SetGetPersonnes
   public function setVar($id,$nom,$prenom) {
     $id     = intval(trim($id))        ;
     $nom    = strtoupper(trim($nom))   ;
     $prenom = strtoupper(trim($prenom));
     $this->TabVars[$id]=array('nom'=>$nom,'prenom'=>$prenom);
   }
   public function getVar($id) {
     return $this->TabVars[$id];
   }
   // Méthodes de l'interface ContEntreprise
   public function setCont($id,$type,$date) {
     $id   = intval(trim($id))      ;
     $type = strtoupper(trim($type));
     $date = trim($date)            ;
     $this->TabCont[]=array('id'=>$id,'type'=>$type,'date'=>$date);
   }
   public function getCont($id) {
     unset($listeR);
     foreach($this->TabCont as $etiquette=>$unControl){
       if ($unControl['id']==$id) {$listeR[]=$unControl;}
     }
     return $listeR;
   }
   public function AfficheP($id) {
     $une_personne=$this->getVar($id);
     echo $id." : ".$une_personne['nom']." - ".$une_personne['prenom'].
     PHP_EOL;
   }
   public function AfficheC($id) {
    $listeDesControles=$this->getCont($id);
    echo "Liste des accès : ".PHP_EOL;
```

```
    foreach($listeDesControles as $num => $unControl) {
      echo "   ".$unControl['id']." - ".$unControl['type']." -
      ".$unControl['date'].PHP_EOL;
    }
  }
}
// === Classe Livraison ===
class Livraison implements ContEntreprise
{ // -- Propriété --
  protected $TabCont = array();
  // Méthodes de l'interface ContEntreprise
  public function setCont($id,$type,$date) {
    $id   = trim($id)              ;
    $type = strtoupper(trim($type));
    $date = trim($date)            ;
    $this->TabCont[]=array('id'=>$id,'type'=>$type,'date'=>$date);
  }
  public function getCont($id) {
    unset($listeR);
    foreach($this->TabCont as $etiquette=>$unControl){
      if ($unControl['id']==$id) {$listeR[]=$unControl;}
    }
    return $listeR;
  }
  public function AfficheC($id) {
   $listeDesControles=$this->getCont($id);
   echo "Liste des accès : ".PHP_EOL;
   foreach($listeDesControles as $num => $unControl) {
     echo "   ".$unControl['id']." - ".$unControl['type']." -
     ".$unControl['date'].PHP_EOL;
   }
  }
}
// === Programme ===
// Création du salarié et affectation des contrôles d'accès
$UnSalarie = new Personnel() ;
$UnSalarie->setVar(324152,"martin","pierre");
$UnSalarie->setCont(324152,"E","12/11/2015-09:43:00");
$UnSalarie->setCont(324152,"S","12/11/2015-18:30:00");
$UnSalarie->setCont(324152,"E","13/11/2015-08:30:00");
$UnSalarie->setCont(324152,"S","13/11/2015-17:45:18");
// Création d'une livraison+affectation des contrôles d'accès
$UneLivraison = new Livraison() ;
$UneLivraison->setCont("CopieExpress","E","12/11/2015-09:43:00");
$UneLivraison->setCont("CopieExpress","S","12/11/2015-09:45:00");
echo "--- Salarié ---".PHP_EOL; // Affichage salarié
$UnSalarie->AfficheP(324152) ;//Méthode la Classe Personnel
$UnSalarie->AfficheC(324152) ;//Méthode la Classe Personnel
echo "--- Livraison ---".PHP_EOL; // Affichage livraison
$UneLivraison->AfficheC("CopieExpress") ;//Méthode la Classe Personnel
?>
```

Voici son exécution :

Listing 9.27 – Exécution de interface1_objet_shell.php

```
$ php interface1_objet_shell.php
--- Salarié ---
324152 : MARTIN - PIERRE
Liste des accès :
   324152 - E - 12/11/2015-09:43:00
   324152 - S - 12/11/2015-18:30:00
   324152 - E - 13/11/2015-08:30:00
   324152 - S - 13/11/2015-17:45:18
--- Livraison ---
Liste des accès :
   CopieExpress - E - 12/11/2015-09:43:00
   CopieExpress - S - 12/11/2015-09:45:00
```

Les méthodes `setCont()`, `getCont()` et `AfficheC()` ont le même contenu et sont présentes dans les deux classes `Personnel` et `Livraison`. Afin d'éviter cette redondance, on peut les déplacer dans une nouvelle classe `GestCont`. De ce fait, c'est cette classe qui implémente les méthodes de l'interface `ContEntreprise`.

Les deux classes `Personnel` et `Livraison` ne contiennent plus ces trois méthodes. Elles doivent désormais hériter de la nouvelle classe `GestCont`. La classe `Personnel` doit en plus implémenter l'interface `SetGetPersonne`.

La nouvelle classe `Livraison` devient vide puisque tout son contenu a été déplacé dans la classe `GestCont`. Sa syntaxe est :

```
class Livraison extends GestCont {}
```

On voit que la classe `Livraison` ainsi réécrite n'apporte plus rien par rapport à la classe `GestCont` dont elle hérite. Elle peut simplement être supprimée. Dans ce cas, la création d'une nouvelle livraison devient :

```
$UneLivraison = new GestCont() ; // Création d'une livraison
```

Le programme `interface3_objet_shell.php` contient ces modifications.

Listing 9.28 – Programme interface3_objet_shell.php

```
<?php
 // === Interface Contrôle d'accès d'une Entreprise ===
 interface ContEntreprise
 { // -- Les méthodes --
   public function setCont($id,$type,$date);
   public function getCont($id);
 }
 // === Interface pour getteur et setter d'une personne ===
 interface SetGetPersonne
 { // -- Les méthodes --
   public function setVar($id,$nom,$prenom);
   public function getVar($id);
```

```
}
// === Classe GestCont de gestion des contrôles ===
class GestCont implements ContEntreprise
{ protected $TabCont = array();// -- Propriété --
  // Méthodes de l'interface ContEntreprise
  public function setCont($id,$type,$date) {
    $id   = trim($id)              ;
    $type = strtoupper(trim($type));
    $date = trim($date)            ;
    $this->TabCont[]=array('id'=>$id,'type'=>$type,'date'=>$date);
  }
  public function getCont($id) {
    unset($listeR);
    foreach($this->TabCont as $etiquette=>$unControl){
      if ($unControl['id']==$id) {$listeR[]=$unControl;}
    }
    return $listeR;
  }
  public function AfficheC($id) {
   $listeDesControles=$this->getCont($id);
   echo "Liste des acess : ".PHP_EOL;
   foreach($listeDesControles as $num => $unControl) {
     echo "   ".$unControl['id']." - ".$unControl['type']." -
     ".$unControl['date'].PHP_EOL;
   }
  }
}
// === Classe Personnel ===
class Personnel extends GestCont implements SetGetPersonne
{ protected $TabVars = array();// -- Propriété --
  // Méthodes de l'interface SetGetPersonnes
  public function setVar($id,$nom,$prenom) {
    $id     = intval(trim($id))        ;
    $nom    = strtoupper(trim($nom))   ;
    $prenom = strtoupper(trim($prenom));
    $this->TabVars[$id]=array('nom'=>$nom,'prenom'=>$prenom);
  }
  public function getVar($id) {return $this->TabVars[$id];}
  // Les méthodes de l'interface ContEntreprise sont héritées
  // Méthodes de la classe
  public function AfficheP($id) {
    $une_personne=$this->getVar($id);
    echo $id." : ".$une_personne['nom']." - ".$une_personne['prenom'].
    PHP_EOL;
  }
}
// === Classe Livraison === vide : elle est supprimée
//class Livraison extends GestCont {}
// === Programme ===
$UnSalarie = new Personnel() ; // Création d'un salarié
```

```
  $UnSalarie->setVar(324152,"martin","pierre");
  // Affectation des contrôles d'accès
  $UnSalarie->setCont(324152,"E","12/11/2015-09:43:00");
  $UnSalarie->setCont(324152,"S","12/11/2015-18:30:00");
  $UnSalarie->setCont(324152,"E","13/11/2015-08:30:00");
  $UnSalarie->setCont(324152,"S","13/11/2015-17:45:18");
  $UneLivraison = new GestCont() ; // Création d'une livraison
  // Affectation  contrôles d'accès
  $UneLivraison->setCont("CopieExpress","E","12/11/2015-09:43:00");
  $UneLivraison->setCont("CopieExpress","S","12/11/2015-09:45:00");
  echo "--- Salarié ---".PHP_EOL; // Affichage salarié
  $UnSalarie->AfficheP(324152) ;//Méthode la Classe Personnel
  $UnSalarie->AfficheC(324152) ;//Méthode la Classe Personnel
  echo "--- Livraison ---".PHP_EOL; // Affichage livraison
  $UneLivraison->AfficheC("CopieExpress") ;//Méthode la Classe Personnel
?>
```

- *Quelques interfaces prédéfinies*

Le langage PHP propose des interfaces prédéfinies. En voici quelques-unes :

- **`Traversable`** : elle indique si une classe peut être parcourue *via* l'instruction `foreach`. Elle doit être implémentée *via* `IteraorAggregate` ou `Iterator`.
- **`Iterator`** : c'est une extension de `Traversable`. Elle uniformise les méthodes de parcours d'objets par itération. Ses méthodes sont : `current()`, `key()`, `next()`, `rewind()`, `valid()`. Toute classe implémentant cette interface doit les réécrire. Un exemple est présenté avec le parcours d'un objet.
- **`ArrayAccess`** : elle donne accès à l'objet en utilisant une syntaxe de type tableau, en précisant entre crochet la clef à laquelle on souhaite accéder. Ses méthodes sont : `offsetExists()`, `offsetGet()`, `offsetSet()`,`offsetUnset()`.
- **`Serializable`** : elle spécialise le comportement de la fonction `serialize()` présentée plus loin dans ce chapitre.

Les classes et méthodes finales

Avec les classes et méthodes *finales*, PHP introduit un nouveau mécanisme assurant la sécurité du processus de développement. Il interdit la redéfinition (surcharge) des méthodes d'une classe, donc assure que le développeur ne peut contourner les traitements définis dans la classe mère.

Les classes finales

Une classe finale ne peut plus être étendue *via* le mécanisme d'héritage et le mot-clef *extends*. Elle empêche tout héritage, ses propriétés et méthodes ne peuvent être redéfinies dans une classe fille. Une classe abstraite ne peut être finale. La déclaration d'une classe finale utilise le mot-clef *final*.

La déclaration en finale de la classe `GestCont` du programme gérant le contrôle d'accès du personnel d'une entreprise, `interface3_objet_shell.php`, est :

```
final class GestCont implements ContEntreprise { ... }
```

Comme il n'est plus possible d'hériter de cette classe, la syntaxe suivante :

```
class Personnel extends GestCont implements SetGetPersonne
{ ...}
```

génère l'erreur :

```
Fatal error: Class Personnel may not inherit from final class (GestCont)
in /.../classe_finale1_objet_shell.php on line 41
```

Dans le cas présent, pour permettre l'héritage, il suffit de définir les seules méthodes en finale, et non la classe complète.

Les méthodes finales

Une *méthode finale* ne peut être modifiée dans une classe fille qui en hérite. Sa déclaration utilise le mot-clef *final*. Voici la déclaration en finale des méthodes de la classe `GestCont` précédente. Cette classe n'est plus en final, elle peut être héritée, mais ses méthodes ne peuvent être réécrites.

Listing 9.29 – Classe GestCont de interface3_objet_shell.php

```
class GestCont implements ContEntreprise
{ protected $TabCont = array();// -- Propriété --
  // Méthodes de l'interface ContEntreprise
  public final function setCont($id,$type,$date) {
    $id   = trim($id)              ;
    $type = strtoupper(trim($type));
    $date = trim($date)            ;
    $this->TabCont[]=array('id'=>$id,'type'=>$type,'date'=>$date);
  }
  public final function getCont($id) {
    unset($listeR);
    foreach($this->TabCont as $etiquette=>$unControl){
      if ($unControl['id']==$id) {$listeR[]=$unControl;}
    }
    return $listeR;
  }
  public final function AfficheC($id) {
   $listeDesControles=$this->getCont($id);
   echo "Liste des acess : ".PHP_EOL;
   foreach($listeDesControles as $num => $unControl) {
     echo "   ".$unControl['id']." - ".$unControl['type']." -
".$unControl['date'].PHP_EOL;
    }
  }
}
```

Les traits

Les traits proposent un mécanisme de gestion des lignes de codes communes à plusieurs classes, dans le cadre d'un héritage simple, en évitant leur duplication.

Principe

Si deux classes effectuent le même traitement, défini dans une méthode, il est souvent nécessaire de dupliquer cette méthode dans chaque classe. Avec l'héritage simple, cette méthode peut être définie (*factorisée*) dans une classe mère dont les deux classes héritent, la faisant ainsi disparaître des classes filles.

Cela suppose que les deux classes sont « de même nature » au sens de l'héritage, comme un cadre et un directeur qui héritent tous les deux de la classe `Personnel`. Mais si cette méthode correspond plus un outil qu'à un traitement lié à la nature de la classe, alors l'héritage n'a pas lieu d'être. Pour résoudre ce problème, il faudrait hériter de plusieurs classes, ce qui n'est pas possible en PHP.

Le trait autorise cette factorisation de lignes de codes, indépendamment de l'héritage, de manière transverse aux classes.

Syntaxe de base

Prenons l'exemple de l'adresse d'une entreprise qui diffère selon que l'on fait partie du personnel ou qu'on effectue des livraisons. La méthode `AfficheAdresse()` du programme `trait0_objet_shell.php` affiche cette adresse selon la classe de l'objet. Elle est écrite dans les deux classes `Personnel` et `Livraison`.

Listing 9.30 – Programme trait0_objet_shell.php

```
<?php
 class Personnel {
  private $AdresseAcces="21 rue Blum";
  // méthode d'affichage de la porte d'accès
  public function AfficheAdresse() {
   echo get_called_class().'-Accès à l\'entreprise par le: ';
   echo $this->AdresseAcces.PHP_EOL;
  }
 }
 class Livraison {
  private $AdresseAcces="Quai numéro 2";
  // méthode d'affichage de la porte d'accès
  public function AfficheAdresse() {
   echo get_called_class().'-Accès à l\'entreprise par le: ';
   echo $this->AdresseAcces.PHP_EOL;
  }
 }
 // === Programme ===
 $UnSalarie = new Personnel();  // Création salarié
 $UnSalarie->AfficheAdresse();  // Affichage adresse
```

```
 $UneLivraison = new Livraison(); // Création livraison
 $UneLivraison->AfficheAdresse(); // Affichage adresse
?>
```

Voici son exécution :

Listing 9.31 – Exécution de trait0_objet_shell.php

```
$ php trait0_objet_shell.php
Personnel-Accès à l'entreprise par le: 21 rue Blum
Livraison-Accès à l'entreprise par le: Quai numéro 2
```

Cette méthode peut être regroupée dans le trait `TraitControl`. Dans chaque classe, la syntaxe `use TraitControl` remplace la définition de la méthode. Voici le programme `trait1_objet_shell.php` réécriture du programme précédent.

Listing 9.32 – Programme trait1_objet_shell.php

```
<?php
 trait TraitControl {
  public function AfficheAdresse() {
   echo get_called_class().'-Accès à l\'entreprise par le: ';
   echo $this->AdresseAcces.PHP_EOL;
  }
 }
 class Personnel {
  private $AdresseAcces="21 rue Blum";
  use TraitControl;
 }
 class Livraison {
  private $AdresseAcces="Quai de numéro 2";
  use TraitControl;
 }
 // === Programme ===
 $UnSalarie = new Personnel();      // Création salarié
 $UnSalarie->AfficheAdresse();      // Affichage adresse
 $UneLivraison = new Livraison(); // Création livraison
 $UneLivraison->AfficheAdresse(); // Affichage adresse
?>
```

Le programme `trait2_objet_shell.php` présente l'usage de traits multiples dans une classe. Le mot-clef `use` est suivi par la liste des traits séparés par une virgule. Les « … » remplacent les lignes inchangées.

Listing 9.33 – Programme trait2_objet_shell.php

```
<?php
 trait TraitControl {
  public function AfficheAdresse() {...}
 }
 trait TraitAcces {
```

```
 public function setAdresse($ad) {
  $this->AdresseAcces=$ad;
 }
}
class Personnel {
 private $AdresseAcces;
 use TraitControl,TraitAcces;
}
class Livraison {
 private $AdresseAcces;
 use TraitControl,TraitAcces;
}
// === Programme ===
// Création salarié et affichage porte d'accès
$UnSalarie = new Personnel()         ;
$UnSalarie->setAdresse("21 rue Blum");
$UnSalarie->AfficheAdresse()         ;
// Création livraison  et affichage porte d'accès
$UneLivraison = new Livraison()              ;
$UneLivraison->setAdresse("Quai de numéro 2");
$UneLivraison->AfficheAdresse()              ;
?>
```

Traitement des conflits

- *Entre deux traits*

Si deux traits possèdent la même méthode, par exemple `getAdresse()`, l'erreur suivante apparaît et indique une collision :

```
Fatal error: Trait method getAdresse has not been applied, because
there are collisions with other trait methods ...
```

Il devient nécessaire de spécifier dans la syntaxe `use` quelle méthode est à utiliser dans la classe. La syntaxe :

```
use TraitControl,TraitAcces ;
```

est par exemple remplacée par :

```
use TraitControl, TraitAcces {
  TraitAcces::getAdresse insteadof TraitControl;
}
```

qui précise que la méthode `getAdresse()` à utiliser est celle de `TraitAcces`.

- *Entre une classe et un trait*

De la même manière, si une classe définit une méthode déjà déclarée dans le trait qu'elle utilise, la méthode de la classe supplante celle du trait. Si le conflit se produit entre une méthode déclarée dans un trait utilisé dans la classe et une méthode de la classe mère, la méthode du trait sera privilégiée.

Les propriétés

Un trait peut aussi regrouper des propriétés communes aux classes. Le programme `trait3_objet_shell.php` est une réécriture de `trait2_objet_shell.php`. La propriété `$AdresseAcces` est déclarée dans un unique trait `TraitAcces` qui regroupe les méthodes communes aux classes `Personnel` et `Livraison`. Ces classes ne contiennent plus que la syntaxe `use TraitAcces`.

Listing 9.34 – Programme trait3_objet_shell.php

```
<?php
 trait TraitAcces {
  private $AdresseAcces;
  public function AfficheAdresse() {
   echo get_called_class().'-Accès à l\'entreprise par le: ';
   echo $this->AdresseAcces.PHP_EOL;
  }
  public function SetAdresse($ad) {
   $this->AdresseAcces=$ad;
  }
 }
 class Personnel {
  use TraitAcces;
 }
 class Livraison {
  use TraitAcces;
 }
 // === Programme ===
 // Création salarié et affichage porte d'accès
 $UnSalarie = new Personnel()         ;
 $UnSalarie->SetAdresse("21 rue Blum");
 $UnSalarie->AfficheAdresse()         ;
 // Création livraison  et affichage porte d'accès
 $UneLivraison = new Livraison()              ;
 $UneLivraison->SetAdresse("Quai de numéro 2");
 $UneLivraison->AfficheAdresse()              ;
?>
```

Une classe utilisant un trait ne peut redéfinir une propriété du trait.

L'imbrication de traits

Tout comme une classe, un trait peut contenir l'usage d'un autre trait. La syntaxe est identique et la gestion des conflits utilise également la même syntaxe. L'exemple suivant montre que `TraitControl` est utilisé dans `TraitAcces`.

```
trait TraitControl {
 public function AfficheAdresse() {
  echo get_called_class().'-Accès à l\'entreprise par le: ';
  echo $this->AdresseAcces.PHP_EOL;
```

```
 }
}
trait TraitAcces {
 private $AdresseAcces;
 public function SetAdresse($ad) {
  $this->AdresseAcces=$ad;
 }
 use TraitControl;
}
```

Avec cette imbrication, seul le trait contenant l'autre doit être utilisé dans les classes `Personnel` et `Livraison`. La classe `Personnel` devient :

```
class Personnel {
 use TraitAcces;
}
```

La visibilité et les alias

Toute classe utilisant un trait peut changer la visibilité initiale des méthodes du trait *via* l'instruction `use` et l'opérateur `as` associé aux mot-clef `private`, `protected` et `public`. Dans le programme `trait5_objet_shell.php`, la méthode `AfficheAdresse()` est déclarée comme `protected` dans le trait. Son utilisation dans les classes `Personnel` et `Livraison` modifie sa visibilité en `public`.

Listing 9.35 – Programme trait5_objet_shell.php

```
<?php
 trait TraitControl {
  protected function AfficheAdresse() {
   echo get_called_class().'-Accès à l\'entreprise par le: ';
   echo $this->AdresseAcces.PHP_EOL;
  }
 }
 class Personnel {
  private $AdresseAcces="21 rue Blum";
  use TraitControl {
   AfficheAdresse as public;
  }
 }
 class Livraison {
  private $AdresseAcces="Quai de numéro 2";
  use TraitControl {
   AfficheAdresse as public;
  }
 }
 // === Programme ===
 // Création salarié et affichage porte d'accès
 $UnSalarie = new Personnel()   ;
 $UnSalarie->AfficheAdresse()   ;
```

```
 // Création salarié et affichage porte d'accès
 $UneLivraison = new Livraison();
 $UneLivraison->AfficheAdresse();
?>
```

L'opérateur `as` peut également créer un alias avec ou sans modification de la visibilité. La syntaxe suivante crée l'alias `Affichage()` pour l'usage de `AfficheAdresse()` dans la classe `Livraison` avec visibilité en `public`.

```
class Livraison {
 private $AdresseAcces="Quai de numéro 2";
 use TraitControl {
  AfficheAdresse as public Affichage;
 }
}
```

Son usage dans le programme devient :

```
$UneLivraison->Affichage();
```

Mais son utilisation reste inchangée pour la classe `Personnel` dans laquelle aucun alias n'est défini.

```
$UnSalarie->AfficheAdresse();
```

Dans l'exemple précédent, l'usage du nom initial de la méthode pour la classe `Livraison` provoquerait une erreur. Le nom initial est toujours connu, mais la méthode `AfficheAdresse()` sans son alias possède une visibilité `protected`.

```
$UneLivraison-> AfficheAdresse (); // Erreur: protected
```

La syntaxe suivante définit un alias sans modification de visibilité :

```
use TraitControl {
  AfficheAdresse as Affichage;
}
```

Les méthodes abstraites

Un trait peut définir des méthodes abstraites, à condition que la classe qui l'utilise soit abstraite. Les classes qui héritent de cette classe mère doivent implémenter la méthode abstraite, comme le montre le programme `trait7_objet_shell.php`.

Listing 9.36 – Programme trait7_objet_shell.php

```
<?php
 trait Pers {
   protected $nom    ;
   protected $prenom ;
   abstract public function Affiche();// Méthode abtraite
 }
 abstract class Personne {
```

```
  use Pers;
}
class Personnel extends Personne
{ protected $matricule   ; // -- Propriétés--
  function __construct($m,$n,$p){ // -- Constructeur --
    $this->matricule = intval(trim($m))     ;
    $this->nom       = strtoupper(trim($n)) ;
    $this->prenom    = strtoupper(trim($p)) ;
  }
  public function Affiche() {
    echo "Nom: ".$this->nom;
    echo "  -  Prénom: ".$this->prenom;
    echo "  -  Matricule: ".$this->matricule.PHP_EOL;
  }
}
// === Programme ===
$UnSalarie = new Personnel(324152,"martin","pierre") ;
echo "--- Salarie ---".PHP_EOL;
$UnSalarie->Affiche() ;//Méthode de la Classe Personnel
?>
```

Les méthodes et les surcharges magiques

Principe

Les méthodes magiques ont un comportement prédéfini en PHP. Leur nom commence par deux soulignés. Lorsqu'elles sont redéfinies dans la classe, elles sont appelées dès qu'un événement est déclenché. Dans le cas contraire, rien n'est effectué. C'est par exemple le cas du constructeur `__contruct()` qui est activé par l'instruction `new` qui crée un objet. Les méthodes magiques contrôlent le comportement de certaines instructions et de la surcharge magique.

La surcharge magique

C'est la possibilité de créer dynamiquement des propriétés et des méthodes inexistantes dans une classe. Ces membres dynamiques sont traités *via* des méthodes magiques s'ils n'existent pas ou s'ils ne sont pas accessibles.

Les méthodes magiques

- *__set() et __get()*

La méthode `__get()`est automatiquement appelée lors d'une tentative de lecture d'une propriété privée. De même `__set()`est automatiquement appelée lors d'une tentative de modification d'une propriété privée. Elles ont été présentées dans la section détaillant la notion d'objet avec le programme `get_set_magique_objet_shell.php`.

- *__call()*

La méthode __call()est appelée lorsqu'on tente d'appeler une méthode inexistante ou inaccessible (privée) d'une classe. Le programme call_objet_shell.php présente son fonctionnement, avec l'appel de la méthode Modifie() qui n'existe pas dans la classe Personnel.

Listing 9.37 – Programme call_objet_shell.php

```
<?php
 class Personnel
 { private $_nom   ; // -- Les propriétés --
   private $_prenom;
   private $_age;
   function __construct($n,$p,$a) { // -- Constructeur --
    $this->_nom=$n;$this->_prenom=$p;$this->_age=$a;
   }
   public function Affiche() {
    echo "Nom : ".$this->_nom." - "       ;
    echo "Prénom : ".$this->_prenom." - " ;
    echo "Age : ".$this->_age.PHP_EOL     ;
   }
   public function __call($NomF, $ArgsF) {
    echo "Appel de la méthode '$NomF' inexistante ! ".PHP_EOL;
    echo "Avec les arguments : ".implode($ArgsF,', ').PHP_EOL;
   }
 }
 // === Programme ===
 $UnePers = new Personnel("dupont","jean",23);
 $UnePers->Affiche();
 $UnePers->Modifie("dupont","durand","pierre",35);
?>
```

Voici son exécution :

Listing 9.38 – Exécution de call_objet_shell.php

```
$ php call_objet_shell.php
Nom : dupont - Prénom : jean - Age : 23
Appel de la méthode 'Modifie' inexistante !
Avec les arguments : dupont, durand, pierre, 35
```

- *__clone()*

La méthode __clone()est appelée lors du clonage d'un objet *via* l'opérateur clone. Son usage est présenté dans la section sur le clonage d'objet dans le programme clone_objet_shell.php.

- *__sleep() et __wakeup()*

Ces méthodes contrôlent les processus de sérialisation et de désérialisation des données présentées dans la section suivante. Cela autorise, par exemple, la sélection des

propriétés à convertir en chaîne de caractères. Le programme `serialize2_objet_shell.php` présente leur mise en œuvre.

La gestion des objets

Cette section présente le parcours, le clonage ou la comparaison d'objets.

Le parcours

Le langage PHP autorise le parcours des membres d'un objet *via* l'instruction `foreach`. La clef retournée est le nom de la propriété.

Le programme `parcours_foreach_objet_shell.php` présente le parcours de l'objet `$UnePersonne` instance de la classe `Personnel`. Deux boucles `foreach` parcourent l'objet. La première se situe dans le programme et la seconde à l'intérieur de la méthode `ParcoursInterne()`.

Listing 9.39 – Programme parcours_foreach_objet_shell.php

```
<?php
 class Personnel
 { public $nom   ; // -- Les propriétés --
   public $prenom;
   protected $tel;
   private $_age ;
   function __construct($n,$p,$t,$a) { // -- Constructeur --
     $this->nom    = strtoupper(trim($n));
     $this->prenom = strtoupper(trim($p));
     $this->tel    = trim($t)            ;
     $this->_age   = intval($a)          ;
   }
  function ParcoursInterne() {
     echo "-- Parcours interne --".PHP_EOL;
     foreach($this as $propriete => $valeur) {
         echo "$propriete : $valeur".PHP_EOL;
     }
  }
 }
 // === Programme ===
 $UnePersonne = new Personnel("dupont","jean","0143112233",45);
 echo "-- Parcours externe --".PHP_EOL;
 foreach($UnePersonne as $propriete => $valeur) {
    echo "$propriete : $valeur".PHP_EOL;
 }
 $UnePersonne->ParcoursInterne();
?>
```

Seules les propriétés visibles sont affichées. Le programme voit les propriétés publiques. La méthode `ParcoursInterne()` affiche toutes les propriétés.

Listing 9.40 – Exécution de parcours_foreach_objet_shell.php

```
$ php parcours_foreach_objet_shell.php
-- Parcours externe --
nom : DUPONT
prenom : JEAN
-- Parcours interne --
nom : DUPONT
prenom : JEAN
tel : 0143112233
_age : 45
```

L'interface `Iterator` peut être implémentée pour le parcours d'un objet. La boucle `foreach` s'appuie alors implicitement sur les méthodes de cette interface. Le programme `parcours_iterator_objet_shell.php` présente le parcours *via* l'interface `Iterator`, et affiche l'écho de l'appel de chaque méthode.

Listing 9.41 – Programme parcours_iterator_objet_shell.php

```
<?php
 class Personnel implements Iterator
 { private $TabProp = array();// -- Propriété --
   function __construct($TP) { // -- Constructeur --
     $TP['nom']   = strtoupper(trim($TP['nom']))   ;
     $TP['prenom']= strtoupper(trim($TP['prenom']));
     $TP['tel']   = trim(trim($TP['tel']))         ;
     $TP['age']   = intval($TP['age'])             ;
     $this->TabProp=$TP;
   }
   public function rewind(){
     echo "..rembobinage\n";
     reset($this->TabProp);
   }
   public function current() {
     $TabProp = current($this->TabProp);
     echo "..actuel : ".$TabProp.PHP_EOL;
     return $TabProp;
   }
   public function key() {
     $TabProp = key($this->TabProp);
     echo "..clé : $TabProp".PHP_EOL;
     return $TabProp;
   }
   public function next() {
     $TabProp = next($this->TabProp);
     echo "..suivant : $TabProp".PHP_EOL;
     return $TabProp;
   }
   public function valid() {
     $clef = key($this->TabProp);
     $TabProp = ($clef!==NULL) && ($clef!==FALSE);
```

```
      echo "..valide : $TabProp".PHP_EOL;
      return $TabProp;
    }
  }
  // === Programme ===
$prop=array('nom'=>"dupont",'prenom'=>"jean",'tel'=>"0143112233",
'age'=>45);
  $UnePersonne = new Personnel($prop);
  echo "-- Parcours via Iterator --".PHP_EOL;
  foreach($UnePersonne as $propriete => $valeur) {
     echo "-----> $propriete : $valeur".PHP_EOL;
  }
?>
```

La boucle `foreach` peut être remplacée par la boucle `while` suivante utilisant explicitement les méthodes de l'interface `Iterator` :

```
$UnePersonne->rewind();
while ($UnePersonne->valid()) {
  echo "----->".$UnePersonne->key()." : ".$UnePersonne->current().PHP_EOL;
  $UnePersonne->next();
}
```

L'objet et la référence

Alors que les variables sont passées par *valeur*, les objets sont passés par *référence*.

L'affectation d'une variable à une autre transmet une *copie de sa valeur*. Les deux variables sont distinctes et la modification de l'une ne change pas l'autre.

L'affectation d'un objet dans un autre transmet sa *référence* (identifiant de l'objet). Les deux objets ne sont pas distincts, *ils partagent la même zone de données*. Le deuxième objet se comporte comme un alias. La modification d'une propriété commune *via* un objet est visible par l'autre.

Le programme `reference_objet_shell.php` présente ces notions. La variable `$var1` copie sa *valeur* dans la variable `$var2`. La modification de `$var2` est invisible pour `$var1`, les deux variables sont séparées. L'objet `$UnePers1` copie sa *référence* dans l'objet `$UnePers2`. Les deux objets ont le même identifiant. Ils sont le même objet. La modification des propriétés de `$UnePers2` modifie `$UnePers1`.

Listing 9.42 – Programme reference_objet_shell.php

```
<?php
 class Personnel
 { public $nom   ; // -- Les propriétés --
   public $prenom;
   function __construct($n,$p) { // -- Constructeur --
     $this->nom=$n;$this->prenom=$p;
   }
   function ParcoursInterne() {
```

```
    foreach($this as $propriete=>$valeur) {echo "$valeur ";}
    echo PHP_EOL;
   }
}
// === Programme ===
$var1 = "CONTENU1"     ;
$var2 = $var1          ;
echo '=== Transmission par valeur ===='.PHP_EOL;
echo '-- Après copie $var2=$var1'.PHP_EOL     ;
echo '$var1='.$var1.' - $var2='.$var2.PHP_EOL ;
$var2 = "AUTRE CONTENU";
echo '-- Après modification de $var2'.PHP_EOL ;
echo '$var1='.$var1.' - $var2='.$var2.PHP_EOL ;
echo '=== Transmission par référence ===='.PHP_EOL;
$UnePers1 = new Personnel("DUPONT","JEAN");
$UnePers2 = $UnePers1          ;
echo '-- Parcours $UnePers1 : ';
$UnePers1->ParcoursInterne()   ;
echo '-- Parcours $UnePers2 : ';
$UnePers2->ParcoursInterne()   ;
echo '-- Après modification de $UnePers2'.PHP_EOL;
$UnePers2->nom="MARTIN"        ;
$UnePers2->prenom="PIERRE"     ;
echo '-- Parcours $UnePers1 : ';
$UnePers1->ParcoursInterne()   ;
echo '-- Parcours $UnePers2 : ';
$UnePers2->ParcoursInterne()   ;
?>
```

Le clonage

La section précédente montre qu'une affectation ne duplique pas l'objet. Il faut utiliser une méthode de clonage pour produire deux objets séparés, ne partageant pas le même espace en mémoire, *via* le mot-clef *clone*. La syntaxe est :

```
$UnPers2 = clone $UnePers1 ;
```

Cet opérateur appelle la méthode magique `__clone()` pour le nouvel objet créé. Il est possible de modifier cette méthode pour effectuer des traitements. Le programme `clone_objet_shell.php` clone deux objets et réécrit la méthode `__clone()`pour transformer le nom et le prénom de l'objet créé, en majuscules. En l'absence de réécriture, les noms et prénoms restent inchangés.

Listing 9.43 – Programme clone_objet_shell.php

```
<?php
 class Personnel
 { private $_nom   ; // -- Les propriétés --
   private $_prenom;
   function __construct($n,$p) { // -- Constructeur --
    $this->_nom=$n;$this->_prenom=$p;
```

```
    }
    public function __clone() {
     $this->_nom    = strtoupper($this->_nom)   ;
     $this->_prenom = strtoupper($this->_prenom);
    }
    function ParcoursInterne() {
     foreach($this as $propriete=>$valeur) {echo "$valeur ";}
     echo PHP_EOL;
    }
  }
  // === Programme ===
  $UnePers1 = new Personnel("dupont","jean");
  $UnePers2 = clone $UnePers1    ;
  echo '-- Parcours $UnePers1 : ';
  $UnePers1->ParcoursInterne()   ;
  echo '-- Parcours $UnePers2 : ';
  $UnePers2->ParcoursInterne()   ;
?>
```

Son exécution montre que les deux objets diffèrent.

Listing 9.44 – Exécution de clone_objet_shell.php

```
$ php clone_objet_shell.php
-- Parcours $UnePers1 :   dupont jean
-- Parcours $UnePers2 :   DUPONT JEAN
```

La comparaison

La comparaison d'objet utilise l'opérateur « == » (ou « != » pour différent) comme pour les variables. La syntaxe est :

```
if ($UnPers2 == $UnePers1) ...
```

Pour que le test précédent soit vrai il faut que les objets aient les *mêmes propriétés* avec les *mêmes valeurs*, et qu'ils soient *l'instance de la même classe*.

La comparaison *via* l'opérateur identique « === » (ou « !== » pour non identique) vérifie que les deux objets sont la même référence, donc qu'il sont tous les deux le même objet. Le test suivant est vrai :

```
$UnPers2 = $UnePers1 ; // Affectation : même objet
if ($UnPers2 === $UnePers1) ...
```

Le typage

Le typage d'objet, également appelé *type_hinting* permet aux méthodes (ou fonctions) d'imposer le type du paramètre. Les types autorisés sont `class`, `interface`, `array` et `callable` (fonctions de rappel). Il suffit d'indiquer le type attendu devant le nom du paramètre lors de la déclaration de la méthode.

Toute classe fille d'une classe autorisée par ce typage sera aussi autorisée. Cela est également vrai pour les classes implémentant une interface autorisée. En PHP 5.6 ce typage n'est pas applicable à `int`, `float`, `string`, `ressource` ou `trait`. Une des évolutions apportées par PHP 7.0 est l'application du typage strict aux types scalaires int, float, string, etc.

Le programme `typage_objet_shell` présente quatre méthodes utilisant le typage : `AffAssistant()` (typage `class`), `AffEquipe()` (typage `interface`), `ChiffreAffaire()` (typage `array`) et `AffPrev()` (typage `callable`).

Listing 9.45 – Programme typage_objet_shell.php

```
<?php
 class Directeur
 { public $nom ; // Propriété
   function __construct($n) {$this->nom=$n;} // Constructeur
   // Méthodes
   public function AffAssistant(Assistant $unA) {
     echo "Assistant : ".$unA->nom.PHP_EOL;
   }
   public function ChiffreAffaire(array $res) {
     echo "-- Chiffre d'affaires --".PHP_EOL;
     foreach($res as $a => $val) {
       $valF=number_format($val,2,","," ")." €";
       echo "$a : $valF".PHP_EOL               ;
     }
   }
   public function AffEquipe(Iterator $E) {
     echo "-- Equipe des ingénieurs --".PHP_EOL;
     foreach($E->P as $num => $valeur) {$NumI=$num+1;
       echo "Ingénieur ".$NumI." : ".$valeur.PHP_EOL;
     }
   }
   public function AffPrev(callable $fonction,array $tab) {
     call_user_func($fonction,$tab);
   }
 }
 class Assistant {
  public $nom ; // Propriété
  function __construct($n) {$this->nom=$n;} // Constructeur
 }
 class Equipe implements Iterator
 { public $P = array();// Propriété
   function __construct($T){$this->P=$T;} // Constructeur
   public function rewind(){reset($this->P);}
   public function current(){$R=current($this->P);return $R;}
   public function key(){$R=key($this->P);return $R;}
   public function next(){$R=next($this->P);return $R;}
   public function valid(){$clef=key($this->P);
     $R=(($clef!==NULL) && ($clef!==FALSE));
```

```
      return $R;
    }
  }
  // = Fonction utilisateur: Projection du chiffre d'affaires =
  function projection($resC) {
     echo "-- Projections --".PHP_EOL;
     foreach($resC as $a => $val) {
       $valF=number_format($val*2,2,","," ")." €";
       $as=$a+2                                ;
       echo "$as : $valF".PHP_EOL              ;
     }
  }
  // === Programme ===
  $equipe      = array("dupont", "martin", "durand")   ;
  $ResAnnuels  = array(2014=>12345.76, 2015=>21340.88) ;
  $UnDirecteur = new Directeur("Bigboss")              ;
  $UnAssistant = new Assistant("Pignon")              ;
  $UneEquipe   = new Equipe($equipe)                  ;
  $UnDirecteur->AffAssistant($UnAssistant)       ;
  $UnDirecteur->AffEquipe($UneEquipe)            ;
  $UnDirecteur->ChiffreAffaire($ResAnnuels)      ;
  $UnDirecteur->AffPrev('projection',$ResAnnuels);
?>
```

Voici son exécution :

Listing 9.46 – Exécution de typage_objet_shell.php

```
$ php typage_objet_shell.php
Assistant : Pignon
-- Equipe des ingénieurs --
Ingénieur 1 : dupont
Ingénieur 2 : martin
Ingénieur 3 : durand
-- Chiffre d'affaires --
2014 : 12 345,76 €
2015 : 21 340,88 €
-- Projections --
2016 : 24 691,52 €
2017 : 42 681,76 €
```

La sérialisation et la sauvegarde

La sérialisation est la représentation linéaire d'une donnée quelconque PHP sous la forme d'une chaîne de caractères. Elle s'applique également aux objets. La fonction `serialize()` traduit l'objet en une chaîne de caractères qui peut ensuite être sauvegardée. La fonction `unserialize()`recrée l'objet original à partir de cette chaîne, à condition que la classe de l'objet soit définie. Le programme `serialize1_objet_shell.php` sauvegarde un objet dans un fichier, puis le restaure.

Listing 9.47 – Programme serialize1_objet_shell.php

```
<?php
 class Personnel
 { private $_nom   ; // -- Les propriétés --
   private $_prenom;
   private $_age;
   function __construct($n,$p,$a) { // -- Constructeur --
    $this->_nom=$n;$this->_prenom=$p;$this->_age=$a;
   }
   public function Affiche() {
    echo "Nom : ".$this->_nom." - "       ;
    echo "Prénom : ".$this->_prenom." - " ;
    echo "Age : ".$this->_age.PHP_EOL     ;
   }
 }
 // === Programme ===
 $UnePers1 = new Personnel("dupont","jean",23);
 // Sérialisation et sauvegarde
 $ChainePers1 = serialize($UnePers1);
 file_put_contents('sauve1_objet.txt', $ChainePers1);
 // Lecture du fichier et reconstruction de l'objet
 $ChainePers2 = file_get_contents('sauve1_objet.txt');
 $UnePers2    = unserialize($ChainePers2);
 // Affichage de l'objet restauré
 echo "-- Objet restauré ---".PHP_EOL;
 $UnePers2->Affiche();
?>
```

Voici son exécution :

Listing 9.48 – Exécution de serialize1_objet_shell.php

```
$ php serialize1_objet_shell.php
-- Objet restauré ---
Nom : dupont - Prénom : jean - Age : 23
```

Voici le contenu du fichier `sauve1_objet.txt` :

Listing 9.49 – Fichier sauve1_objet.txt

```
$ cat sauve1_objet.txt
O:9:"Personnel":3:{s:15:"Personnel_nom";s:6:"dupont";s:18:"Personnel_
prenom";s:4:"jean";s:15:"Personnel_age";i:23;}
```

Le programme `serialize2_objet_shell.php` est une adaptation du programme précédent. Il utilise les méthodes magiques `__sleep()` appelée par `serialize()`, et `__wakeup()` appelée par `unserialize()`.

La première méthode `__sleep()` limite la sérialisation aux seules propriétés `$_nom` et `$_prenom`. La propriété `$_age` n'est pas sérialisée, donc pas sauvegardée.

La seconde méthode `__wakeup()` effectue un post-traitement sur le nom et le prénom en mettant la première lettre en majuscule, et convertit l'âge en entier. Comme cette propriété n'est pas restaurée, sa valeur initiale vide, est convertie en numérique 0 par `intval()`. Les lignes inchangées sont remplacées par des « ... ».

Listing 9.50 – Programme serialize2_objet_shell.php

```
<?php
 class Personnel
 { ... // -- Les propriétés --
   function __construct($n,$p,$a) { ... } // Constructeur
   public function Affiche() { ... }
   // Méthode magique __sleep() Appelée par serialize()
   public function __sleep() {
    echo '-- Exécution de __sleep --'.PHP_EOL;
     return array('_nom','_prenom');
   }
   // Méthode magique __sleep() Appelée par unserialize()
   public function __wakeup() {
    echo '-- Exécution de __wakeup --'.PHP_EOL;
    $this->_nom    = ucwords($this->_nom)   ;
    $this->_prenom = ucwords($this->_prenom);
    $this->_age    = intval($this->_age)    ;
   }
 }
 // === Programme ===
 $UnePers1 = new Personnel("dupont","jean",23);
 ...
?>
```

Voici son exécution :

Listing 9.51 – Exécution de serialize2_objet_shell.php

```
$ php serialize2_objet_shell.php
-- Exécution de __sleep --
-- Exécution de __wakeup --
-- Objet restauré ---
Nom : Dupont - Prénom : Jean - Age : 0
```

Les espaces de noms ou namespaces

Principe

Les espaces de noms ou *namespaces* regroupent les noms des classes, méthodes, propriétés ou constantes dans des espaces de nommage séparés, évitant ainsi tout conflit et autorisant la réutilisation de noms existants dans d'autres espaces. Cela résout le problème de l'importation de modules existants, ayant des noms de classe ou de membres pouvant être utilisés dans notre programme.

Déclaration

Un espace de noms est déclaré *via* la syntaxe `namespace`, en première ligne du programme PHP. Aucune instruction ne doit la précéder sauf éventuellement `declare`. Tous les noms déclarés par la suite sont regroupés dans cet espace, jusqu'à la déclaration d'un nouvel espace de noms. Si aucun espace de noms n'est déclaré, les noms sont regroupés dans l'espace *global*. L'usage est de déclarer un unique espace de noms par fichier .php, mais rien n'empêche d'en avoir plusieurs.

Le programme `namespace1_objet_shell.php` utilise une syntaxe procédurale. Il déclare `EspacePerso`. La fonction `strtoupper()` est réécrite, elle ne rentre pas en conflit avec l'espace global. Cette fonction appelle `ucword()` qui passe l'initiale de la chaîne en majuscule. L'appel de `strtouper()` exécute la fonction de l'espace de noms `EspacePerso`. L'appel de `\strtoupper()` (avec le caractère `\` devant) appelle explicitement la fonction de l'espace global.

Listing 9.52 – Programme namespace1_objet_shell.php

```
<?php
 namespace EspacePerso; // Déclaration d'un espace de noms
 function strtoupper($nom) { // Déclaration d'une fonction
   return ucwords($nom);
 }
 // == Programme ==
 // Appel dans l'espace EspacePerso
 echo strtoupper("dupont").PHP_EOL;
 // Appel dans l'espace global
 echo \strtoupper('dupont').PHP_EOL;
?>
```

Voici son exécution :

Listing 9.53 – Exécution de namespace1_objet_shell.php

```
$ php namespace1_objet_shell.php
Dupont
DUPONT
```

Le programme `namespace2_objet_shell.php` présente la déclaration de deux espaces de noms dans le même fichier avec une programmation objet.

Listing 9.54 – Programme namespace2_objet_shell.php

```
<?php
 namespace EspacePerso1; // -- Espace de noms EspacePerso1 --
 class Personnel
 { private $_nom   ; // Propriété
   function __construct($n) {$this->_nom=$n;} // Constructeur
   public function Affiche() {
     echo "Nom:".strtoupper($this->_nom).PHP_EOL;}
 }
```

```
$UnePers1 = new Personnel("dupont");
$UnePers1->Affiche();

namespace EspacePerso2; // -- Espace de noms EspacePerso2 --
class Personnel
{ private $_nom   ; // Propriété

  function __construct($n) {$this->_nom=$n;} // Constructeur
  public function Affiche() {
    echo "Nom:".$this->_nom.PHP_EOL;}
}
$UnePers1 = new Personnel("martin");
$UnePers1->Affiche();
?>
```

Voici son exécution :

Listing 9.55 – Exécution de namespace2_objet_shell.php

```
$ php namespace2_objet_shell.php
Nom:DUPONT
Nom:martin
```

Le programme `namespace3_objet_shell.php` présente une autre syntaxe dans laquelle les éléments de l'espace de noms sont entourés d'accolades `{}`. Le libellé du troisième espace de noms n'est pas indiqué, c'est l'espace de noms global. Les lignes identiques au programme précédent sont remplacées par des « ... ».

Listing 9.56 – Programme namespace3_objet_shell.php

```
<?php
 namespace EspacePerso1 // -- Espace de noms EspacePerso1 --
 {
  class Personnel { ... }
  $UnePers1 = new Personnel("dupont");
  $UnePers1->Affiche();
 }
 namespace EspacePerso2 // -- Espace de noms EspacePerso2 --
 {
  class Personnel { ... }
  $UnePers1 = new Personnel("martin");
  $UnePers1->Affiche();
 }
 namespace             // -- Espace de noms global --
 {
  class Personnel
  { private $_nom   ; // Propriété
    function __construct($n) {$this->_nom=$n;} // Constructeur
    public function Affiche() {
      echo "Nom:".ucwords($this->_nom).PHP_EOL;}
  }
```

```
  $UnePers1 = new Personnel("durand");
  $UnePers1->Affiche();
 }
?>
```

La constante __NAMESPACE__

La constante magique __NAMESPACE__ indique le nom de l'espace de noms utilisé. Le programme `namespace4_objet_shell.php` présente sa syntaxe. Les lignes identiques au programme précédent sont remplacées par des « ... ».

Listing 9.57 – Programme namespace4_objet_shell.php

```
<?php
 namespace EspacePerso1 // -- Espace de noms EspacePerso1 --
 {class Personnel { ... }
  echo "Espace de noms : ".__NAMESPACE__.PHP_EOL;
  $UnePers1 = new Personnel("dupont");
  $UnePers1->Affiche();
 }
 namespace EspacePerso2 // -- Espace de noms EspacePerso2 --
 {class Personnel { ... }
  echo "Espace de noms : ".__NAMESPACE__.PHP_EOL;
  $UnePers1 = new Personnel("martin");
  $UnePers1->Affiche();
 }
 namespace              // -- Espace de noms global --
 {class Personnel { ... }
  echo "Espace de noms : ".__NAMESPACE__.PHP_EOL;
  $UnePers1 = new Personnel("durand");
  $UnePers1->Affiche();
 }
?>
```

Son exécution montre que l'espace global ne possède pas de nom.

Listing 9.58 – Exécution de namespace4_objet_shell.php

```
$ php namespace4_objet_shell.php
Espace de noms : EspacePerso1
Nom:DUPONT
Espace de noms : EspacePerso2
Nom:martin
Espace de noms :
Nom:Durand
```

Sous-espaces

Il est possible de définir un sous-espace de noms à l'intérieur d'un autre *via* le caractère « \ ». La syntaxe est de la forme :

```
namespace EPerso1
{
 ...
}
namespace EPerso1\ECadre
{
 ...
}
?>
```

Remarque

Les espaces de noms sont déclarés dans n'importe quel ordre. Si l'espace supérieur n'est pas préalablement déclaré, il l'est automatiquement à la déclaration du sous-espace.

Hiérarchie d'espaces

Dans les exemples précédents, les méthodes étaient appelées sans préciser l'espace de noms. Le nom de la classe ou de la méthode était *non qualifié*. Les appels étaient faits en relatif, c'est-à-dire à partir de l'espace où se trouve l'instruction. Il est possible d'appeler des fonctions (en procédural), ou des classes (en objet) en indiquant explicitement l'espace de noms à utiliser. Le nom de la fonction ou de la classe est alors dit *qualifié*. La syntaxe utilise le caractère hiérarchique « \ » et l'arborescence des espaces de noms permettant d'atteindre l'élément.

La syntaxe suivante est relative. Elle indique que la classe `Personnel` est celle du sous-espace `ECadre` de l'espace d'exécution de cette instruction.

```
$UnePers2=new ECadre\Personnel("martin");
```

La syntaxe suivante qualifie la classe `Personnel` qui se trouve dans l'arborescence indiquée en absolu : `\EPerso\ECadre`.

```
$UnePers2=new \EPerso\ECadre\Personnel("dabert");
```

Le programme `namespace5_objet_shell.php` présente ces syntaxes.

Listing 9.59 – Programme namespace5_objet_shell.php

```
<?php
 namespace EPerso // -- Espace de noms EPerso --
 {
  class Personnel
  { private $_nom   ; // Propriété
    function __construct($n) {$this->_nom=$n;} // Constructeur
    public function Affiche() {
      echo "Personnel:".$this->_nom.PHP_EOL;}
  }
  echo "Espace de noms : ".__NAMESPACE__.PHP_EOL;
  $UnePers1 = new \EPerso\Personnel("dupont"); // Absolue
  $UnePers1->Affiche();
```

```
  $UnePers2 = new ECadre\Personnel("martin"); // Relative
  $UnePers2->Affiche();
 }
 namespace EPerso\ECadre // -- Espace de noms EPerso\ECadre --
 {
  class Personnel
  { private $_nom   ; // Propriété
    function __construct($n) {$this->_nom=$n;} //Constructeur
    public function Affiche() {
      echo "Cadre:".strtoupper($this->_nom).PHP_EOL;}
  }
  echo "Espace de noms : ".__NAMESPACE__.PHP_EOL;
  $UnePers1 = new Personnel("leroy"); //Non qualifié
  $UnePers1->Affiche();
  $UnePers2 = new \EPerso\ECadre\Personnel("dabert");//Absolu
  $UnePers2->Affiche();
 }
?>
```

Voici son exécution :

Listing 9.60 – Exécution de namespace5_objet_shell.php

```
$ php namespace5_objet_shell.php
Espace de noms : EPerso
Personnel:dupont
Cadre:MARTIN
Espace de noms : EPerso\ECadre
Cadre:LEROY
Cadre:DABERT
```

Alias d'espaces

La syntaxe d'une hiérarchie d'espaces de noms peut être simplifiée *via* la définition d'*alias* avec les mots-clefs `use` et `as`. La syntaxe précédente :

```
$UnePers1 = new \EPerso\Personnel("dupont");
```

se réécrit :

```
use \EPerso\Personnel as AP_Personnel;
$UnePers1 = new AP_Personnel("dupont");
```

Le programme `namespace6_objet_shell.php` réécrit le programme précédent avec les alias d'espaces. Les lignes identiques sont remplacées par des « ... ».

Listing 9.61 – Programme namespace6_objet_shell.php

```
<?php
 namespace EPerso // -- Espace de noms EPerso --
 {use \EPerso\Personnel as AP_Personnel;
  use \EPerso\ECadre\Personnel as AC_Personnel;
```

```
  class Personnel { ...  }
  echo "Espace de noms : ".__NAMESPACE__.PHP_EOL;
  $UnePers1 = new AP_Personnel("dupont"); // Alias
  $UnePers1->Affiche();
  $UnePers2 = new AC_Personnel("martin"); // Alias
  $UnePers2->Affiche();
 }
 namespace EPerso\ECadre // -- Espace de noms EPerso\ECadre --
 {use \EPerso\ECadre\Personnel as AC_Personnel;
  class Personnel { ...  }
  echo "Espace de noms : ".__NAMESPACE__.PHP_EOL;
  $UnePers1 = new Personnel("leroy");
  $UnePers1->Affiche();
  $UnePers2 = new AC_Personnel("dabert"); // Alias
  $UnePers2->Affiche();
 }
?>
```

Les exceptions

Principe

Une exception est générée lorsqu'un traitement ne s'est pas déroulé correctement, par exemple une division par zéro. Dans ce cas, une exception est « lancée » (`throw`), et il faut alors « essayer » (`try`) de la « capturer » (`catch`). L'objet « lancé » doit être de la classe `Exception` ou d'une classe qui étend celle-ci.

Les exceptions sont produites par différentes méthodes de PHP, comme avec l'outil de gestion des accès aux bases de données PDO (*PHP Data Objetcs*), étudié au chapitre 10, qui génère des exceptions de la classe *PDOException*. Mais le développeur peut lui-même « lancer » ses propres exceptions *via* `throw`.

Syntaxe

Une exception est lancée *via* la syntaxe `throw`. Par exemple :

```
if (!$reponse){
   throw new Exception('Erreur de requête.'.PHP_EOL);
}
```

Les lignes d'instructions pouvant générer une exception doivent être dans un bloc `try{}`. Chaque `try` doit avoir au moins un bloc `catch(){}` ou `finally(){}`. Voici un exemple de syntaxe.

```
try {
 // --- Connexion de la base de données ---
 $bdd=new PDO('mysql:host=localhost;dbname=CoursPHP', ...);
 // --- Exécution de la requête ---
 $reponse = $bdd->query('SELECT * FROM personnes');
 ...
```

```
 $reponse->closeCursor();
}
catch(Exception $e){
 echo 'Erreur : '.$e->getMessage();
}
```

Plusieurs blocs `catch` peuvent capturer différentes classes d'exception.

Lorsqu'une exception est lancée dans le bloc `try` par une instruction, les instructions suivantes ne sont pas exécutées. PHP passe le contrôle aux instructions du premier bloc `catch` correspondant.

Si aucune exception n'est levée dans le bloc `try`, les instructions des blocs `catch` correspondants ne sont pas exécutées. Le programme se poursuit par les instructions situées après le dernier bloc `catch` correspond au `try`.

Si une exception est lancée et non capturée, une erreur fatale est émise avec un message tel que « *Uncaught Exception ...* ».

Il est possible de relancer une exception dans le bloc `catch`.

Si un bloc `finally` est spécifié après le bloc `catch`, les instructions s'y trouvant seront toujours exécutées après celle du `try` ou du `catch` indépendamment du fait qu'une exception ait été lancée. Cela permet d'exécuter du code même quand une exception, par exemple fatale, n'est pas gérée dans un bloc `try`. On peut y trouver, par exemple, la fermeture de la connexion à une base de données.

Remarque

La gestion des exceptions est applicable en programmation procédurale.

Si une exception est lancée dans un espace de nom, il faut indiquer l'espace de noms global pour la classe Exception. La syntaxe de la classe devient `\Exception`.

Exemples

- *Génération d'une exception*

Le programme `exception1_objet_shell.php` contrôle le constructeur de la classe `Personnel` afin de générer une exception (`new Exception`) si le nom fourni est vide. Le message d'erreur est indiqué en paramètre à la classe `Exception`, ainsi qu'un numéro défini par le développeur. Le bloc `catch` appelle les méthodes de la classe `Exception` : `getMessage()`, `getCode()`, `getFile()`, `getLine()`, `getTrace()`, `getTraceAsString()`et `__toString()`.

Listing 9.62 – Programme exception1_objet_shell.php

```
<?php
 class Personnel
 {private $_nom   ; // -- Propriété --
  function __construct($n) { // -- Constructeur --
   try {
```

```
      if (empty($n))
         throw new Exception('Pas de création: Nom vide',15);
      $this->_nom=$n;
      echo "Création de ".$this->_nom.PHP_EOL;
    }
    catch (Exception $e) {
      echo $e->getMessage().PHP_EOL;
      echo "Erreur numéro   : ".$e->getCode().PHP_EOL;
      echo "Fichier         : ".$e->getFile().PHP_EOL;
      echo "Ligne           : ".$e->getLine().PHP_EOL;
      echo "Tableau de trace: ".PHP_EOL;
      print_r($e->getTrace());
      echo "Trace formaté   : ".$e->getTraceAsString().PHP_EOL;
      echo "Chaîne formatée  : ".$e->__toString().PHP_EOL;
    }
   }
 }
 $UnePers1 = new Personnel("dupont");
 $UnePers2 = new Personnel("");
?>
```

Son exécution montre que la création de « dupont » ne génère aucune exception, alors que la création avec une chaîne vide génère une exception.

Listing 9.63 – Exécution de exception1_objet_shell.php

```
$ php exception1_objet_shell.php
Création de dupont
Pas de création: Nom vide
Erreur numéro    : 15
Fichier          : /.../exception1_objet_shell.php
Ligne            : 8
Tableau de trace :
Array
(
    [0] => Array
        (
            [file] => /.../exception1_objet_shell.php
            [line] => 24
            [function] => __construct
            [class] => Personnel
            [type] => ->
            [args] => Array
                (
                    [0] =>
                )

        )

)
```

```
Trace formaté    : #0 /Users/lery/Sites/CoursPHP/9_POO/exception1_
objet_shell.php(24): Personnel->__construct('')
#1 {main}
Chaîne formatée : exception 'Exception' with message 'Pas de
création:Nom vide' in /.../exception1_objet_shell.php:8
Stack trace:
#0 /.../exception1_objet_shell.php(25): Personnel->__construct('')
#1 {main}
```

- *Nouvelle classe exception et catch multiples*

L'exemple suivant présente la création d'une nouvelle classe `MonException`. Une exception `MonException` est lancée si la taille du nom est inférieure à 2. Deux blocs `catch` capturent respectivement une exception `MonException` pour les noms trop courts, et une exception `Exception` pour les noms vides.

Listing 9.64 – Programme exception2_objet_shell.php

```
<?php
 class MonException extends Exception {
   public function __toString() {
     $m = $this->message ;
     $c = $this->code    ;
     return "Problème : ".$m." => Taille=".$c;
   }
 }
 class Personnel
 { private $_nom   ; // -- Propriété --
   function __construct($n) { // -- Constructeur --
   try {
     if (empty($n))
       throw new Exception('Pas de création: Nom vide');
     $taille = strlen($n)                    ;
     $MonMessage="Nom '".$n."' trop petit"  ;
     if ($taille<2)
       throw new MonException($MonMessage,$taille);
     $this->_nom=$n                          ;
     echo "Création de ".$this->_nom.PHP_EOL;
   }
   catch (MonException $me) {
     echo $me->__toString().PHP_EOL;
   }
   catch (Exception $e) {
     echo $e->getMessage().PHP_EOL;
   }
  }
 }
 $UnePers1 = new Personnel("dupont");
 $UnePers2 = new Personnel("T")     ;
 $UnePers3 = new Personnel("")      ;
?>
```

Voici son exécution :

Listing 9.65 – Exécution de exception2_objet_shell.php

```
$ php exception2_objet_shell.php
Création de dupont
Problème : Nom 'T' trop petit => Taille=1
Pas de création: Nom vide
```

- *Gestion d'un finally*

Cet exemple montre que le bloc `finally` est toujours exécuté. Le constructeur de la classe `Famille` calcule le revenu par personne. `$UneFamille1` est correctement créée. `$UneFamille2` est créée avec un nombre de personnes erroné, soit 0, ce qui provoque une division par 0 du calcul de revenu par personne.

Listing 9.66 – Programme exception3_objet_shell.php

```
<?php
 class Famille
 {private $_nom        ; // -- Les propriétés --
  private $_nbpers     ;
  private $_revenus    ;
  private $_r_par_pers ;
  function __construct($n,$r,$nbp) { // -- Constructeur --
   try {
     if (empty($n))
       throw new Exception('Pas de création: Nom vide');
     $this->_nom        = $n     ;
     $this->_revenus    = $r     ;
     $this->_nbpers     = $nbp   ;
     $this->_r_par_pers = $r/$nbp;
     echo "Famille ".$this->_nom.PHP_EOL;
     echo "   Nombre de personnes : ".$this->_nbpers.PHP_EOL ;
     echo "   Revenus : ".$this->_revenus.PHP_EOL            ;
     echo "   Revenus/personne : ".$this->_r_par_pers.PHP_EOL;
   }
   catch (Exception $e) {
     echo $e->getMessage().PHP_EOL;
   }
   finally {
     echo "Fin de création pour la famille : ".$n.PHP_EOL;
   }

  }
 }
 $UneFamille1 = new Famille("dupont",4000,2);
 $UneFamille2 = new Famille("martin",2000,0);
?>
```

Son exécution montre que le message du bloc `finally` est toujours affiché.

Listing 9.67 – Exécution de exception3_objet_shell.php

```
$ php exception3_objet_shell.php
Famille dupont
   Nombre de personnes : 2
   Revenus : 4000
   Revenus/personne : 2000
Fin de création pour la famille : dupont

Warning: Division by zero in /.../exception3_objet_shell.php on line 15
Famille martin
   Nombre de personnes : 0
   Revenus : 2000
   Revenus/personne :
Fin de création pour la famille : martin
```

Exceptions prédéfinies

Le langage PHP propose un ensemble de classes d'exceptions prédéfinies. En voici quelques-unes :

- `BadFunctionCallException` : la fonction de rappel est inexistante ;
- `BadMethodCallException` : la méthode de rappel est inexistante ;
- `DomainException` : la valeur n'est pas dans le domaine de données valides ;
- `InvalidArgumentException` : l'argument n'est pas du type attendu ;
- `LengthException` : la taille est invalide ;
- `LogicException` : erreurs dans la logique du programme ;
- `OutOfBoundsException` : la valeur n'est pas une clé valide ;
- `OutOfRangeException` : l'index n'est pas dans la plage de valeurs autorisées ;
- `RuntimeException` : erreur d'exécution ;
- `UnexpectedValueException` : la valeur n'est pas du type attendu.

Exercices

9.1 Cet exercice porte sur les classes et les objets. Nous prenons comme support le programme de gestion d'une liste de personnes donné à l'exercice n° 1 du chapitre 8. Adapter ce programme afin de gérer une classe `Personne` représentant une personne et une classe `LPers` gérant une liste de personnes. Cette liste sera un tableau d'objet de la classe `Personne`. Chaque personne aura les propriétés privées nom, prénom et âge. Ces deux classes seront chargées dynamiquement. Le programme principal présentera le menu suivant :

```
-1- Saisie d'une liste de personnes
-2- Affichage de toutes les personnes
-3- Sauvegarde dans un fichier
-4- Chargement d'un fichier
-0- Quitter
Choix (1,2,3,...) :
```

Chaque choix appellera une méthode de la classe LPers qui effectuera le traitement demandé.

Solutions

9.1 Le programme `exo_liste_personnes1_objet_shell.php` contient le chargement dynamique des classes `Personne` et `LPers`. Il crée l'objet `$ListePersonnes` instance de la classe `LPers`, puis affiche le menu. Chaque choix appelle une ou plusieurs méthodes de cette classe.

Listing 9.68 – Programme exo_liste_personnes1_objet_shell.php

```
<?php
 // -- Chargement dynamique des classes ---
 function __autoload($NomClasse) {
    require $NomClasse . '_Classe.php';
 }
 // === Programme ===
 define("NON_TROUVE",-1)         ;
 $choix = NON_TROUVE              ;
 // Création d'une liste de personnes
 $ListePersonnes = new LPers()   ;
 // --- Menu ---
 while ($choix !=0) {
```

```
    $choix=NON_TROUVE;
    echo "-1- Saisie d'une liste de personnes".PHP_EOL   ;
    echo "-2- Affichage de toutes les personnes".PHP_EOL ;
    echo "-3- Sauvegarde dans un fichier".PHP_EOL        ;
    echo "-4- Chargement d'un fichier".PHP_EOL           ;
    echo "-0- Quitter".PHP_EOL                            ;
    echo "Choix (1,2,3,...) : ";
    fscanf(STDIN,"%d",$choix);
    switch($choix) {
      case 1 : $ListePersonnes->SaisieLPers()  ; break   ;
      case 2 : $ListePersonnes->AfficheLPers() ; break   ;
      case 3 : $ListePersonnes->SauveLPers()   ; break   ;
      case 4 : $ListePersonnes->ChargeLPers()  ; break   ;
      case 0 : echo "Au revoir".PHP_EOL         ; break  ;
      default : echo "Choix incorrect !".PHP_EOL; break  ;
    }
  }
?>
```

La classe `Personne` contient trois propriétés privées `$_nom`, `$_prenom` et `$_age`, les méthodes magiques `__get()` et `__set()`, les méthodes `Saisie()` et `Affiche()`.

Listing 9.69 – Fichier Personne_Classe.php

```
<?php
 class Personne {
   const AGE_MIN = 1     ; // Déclaration de deux constantes
   const AGE_MAX = 120   ;
   private $_nom         ; // Les propriétés
   private $_prenom      ;
   private $_age         ;
   // -- __get et __set méthodes magiques --
   public function __get($propriete) {
     if (property_exists($this, $propriete)) {
       return $this->$propriete;
     }
   }
   public function __set($propriete, $valeur) {
     if (property_exists($this, $propriete)) {
       if ($propriete=="age") {
         if (($valeur>=self::AGE_MIN)&&($valeur<self::AGE_MAX))
           $this->_age=intval(trim($valeur));
       }
       else {
         $this->$propriete = strtoupper(trim($valeur));
       }
     }
   }
   // -- Méthode de saisie d'une personne --
   public function Saisie() {
     try {
```

```
        echo "Entrez un nom (vide pour fin)    : "   ;
        $this->_nom    = fgets(STDIN);
        $this->_nom=trim($this->_nom);
        if (empty($this->_nom)) throw new Exception();
        echo "Entrez un prénom (vide pour fin) : "   ;
        $this->_prenom = fgets(STDIN)      ;
        $this->_prenom=trim($this->_prenom);
        if (empty($this->_prenom)) throw new Exception();
        echo "Entrez un âge (vide pour fin)    : "      ;
        $this->_age = fgets(STDIN)        ;
        $this->_age = intval($this->_age) ;
        if ($this->_age <= 0) throw new Exception();
        // traitement des données
        $this->_nom = strtoupper($this->_nom)      ;
        $this->_prenom = strtoupper($this->_prenom);
       }
       catch(Exception $e){
         $this->_nom    = null ;
         $this->_prenom = null ;
         $this->_age    = 0    ;
       }
    }
    // -- Méthode d'affichage d'une personne --
    public function Affiche() {
      echo $this->_nom."\t".$this->_prenom."\t".$this->_age;
      echo PHP_EOL ;
    }
 }
?>
```

La classe `LPers` contient la propriété privée `$_LP` qui est la liste des personnes. Elle contient également les méthodes suivantes : `SaisieLPers()`, `AfficheLPers()`, `AfficheTabPersonnes()`, `SauveLPers()`, `ChargeLPers()`.

Listing 9.70 – Fichier LPers_Classe.php

```
<?php
 class LPers {
  const NON_TROUVE = '-1' ;
  const REP_SAUVE  = "../../Sauvegardes/" ;
  private $_LP = array();// Propriété : Liste des personnes
  // Méthode de saisie de la liste des personnes
  public function SaisieLPers() {
   $nom="";
   while(!(is_null($nom))) {
     $Pers = new Personne();
     $Pers->Saisie()       ;
     $nom = $Pers->_nom    ;
     if(!(is_null($nom))) $this->_LP[]=$Pers;
   }
```

```
} // Fin de SaisieLPers
// Méthode d'affichage de la liste des personnes
public function AfficheLPers() {
 if (count($this->_LP) == 0) {
  echo "Aucune personne à afficher !".PHP_EOL;
 }
 else {
  $this->AfficheTabPersonnes($this->_LP);
 }
} // Fin de AfficheLPers
// Méthode outils d'affichage d'une liste de personnes
private function AfficheTabPersonnes($ListeP) {
  // entête de l'affichage
  echo "----------------------------\n";
  echo "ID :\tNom\tPrénom\tAge\n";
  echo "----------------------------\n";
  foreach($ListeP as $indice => $Pers) {
   echo "$indice :\t";
   $Pers->Affiche();
  }
  echo "----------------------------\n";
} // Fin de AfficheTabPersonnes
// Méthode de sauvegarde de la liste des personnes
public function SauveLPers() {
 try {
  $nbpers=count($this->_LP);
  if ($nbpers == 0)
     throw new Exception("Aucune personne à sauvegarder !");
  echo "Nom du fichier de sauvegarde : ";
  $NomF = fgets(STDIN);
  $NomF = trim($NomF) ;
  if (empty($NomF))
     throw new Exception("Erreur : Fichier inaccessible !");
  // Construction du nom du fichier
  $NomF = self::REP_SAUVE.$NomF;
  // Sérialisation et sauvegarde
  $ChaineLPers1 = serialize($this->_LP) ;
  file_put_contents($NomF,$ChaineLPers1);
  echo $nbpers." personne(s) sauvegardée(s) dans le fichier ".$NomF.
  PHP_EOL;
 }
 catch(Exception $e){
  echo $e->getMessage().PHP_EOL;
 }
} // Fin de SauveLPers
// Méthode de chargement de la liste des personnes
public function ChargeLPers() {
 try {
  echo "Nom du fichier à charger : ";
  $NomF = fgets(STDIN);
```

```
    $NomF = trim($NomF) ;
    if (empty($NomF))
       throw new Exception("Erreur : Fichier non trouvé !");
    // Construction du nom du fichier
    $NomF = self::REP_SAUVE.$NomF;
    // Sérialisation et sauvegarde
    if (file_exists($NomF)) {
     $ChaineLPers = file_get_contents($NomF) ;
     $this->_LP   = unserialize($ChaineLPers);
     $nbpers=count($this->_LP);
     echo $nbpers." personne(s) lue(s) à partir du fichier ".$NomF.PHP_EOL;
    }
    else
     echo "Fichier $NomF non trouvé !".PHP_EOL;
   }
   catch(Exception $e){
    echo $e->getMessage().PHP_EOL;
   }
  } // Fin de ChargeLPers
 } // Fin de la classe LPers
?>
```