

PHP

Et

POO

TABLE DES MATIERES

Chapitre 1 LES CLASSES ET LES OBJETS.....	4
1.1 Présentation générale.....	5
1.1.1 Généralités.....	5
1.1.2 Représentation graphique.....	5
1.1.3 Les objets.....	5
1.1.4 Principales caractéristiques objet de PHP5.....	5
1.2 Les classes en PHP 5.....	6
1.2.1 Syntaxe de définition d'une classe.....	6
1.2.2 Les propriétés et les méthodes.....	6
1.2.3 Instantiation d'un objet.....	8
1.2.4 Exemple complet.....	9
1.2.5 Portée des attributs et des méthodes.....	10
1.2.6 Le polymorphisme en PHP.....	11
1.3 Les accesseurs (__set et __get).....	13
1.4 Les constructeurs (__construct).....	14
1.4.1 Définition.....	14
1.4.2 Exemple.....	14
1.5 Les destructeurs (__destruct).....	15
1.6 L'héritage.....	16
1.6.1 Définition.....	16
1.6.2 Représentation graphique.....	16
1.6.3 Syntaxe.....	16
1.6.4 Exemple complet.....	17
1.6.5 Héritage et surcharge.....	18
1.6.6 Héritage et transtypage.....	20
1.7 Compléments.....	23
1.7.1 Les constantes de classe.....	23
1.7.2 Les opérateurs parent, self et ::.....	24
1.7.3 Méthode statique, variable statique.....	25
1.7.4 Classe abstraite.....	33
1.7.5 Classes et méthodes Final.....	36
1.8 Linéariser les objets (Sérialiser, désérialiser).....	38
1.8.1 La sérialisation.....	38
1.8.2 La désérialisation.....	38
1.8.3 Sérialisation et désérialisation Fichier.....	39
1.8.4 Sérialisations et désérialisations Fichier.....	40
1.8.5 Les méthodes __sleep() et __wakeup().....	43
1.9 Les interfaces.....	46
1.10 Parcourir les membres d'une classe.....	51
1.10.1 Via une méthode interne.....	51
1.10.2 Via la fonction get_object_vars(\$objet).....	52
1.10.3 Via l'interface Iterator de PHP.....	53
1.11 DIVERS.....	56
1.11.1 Les méthodes magiques.....	56
1.11.2 La méthode __toString().....	56
1.11.3 Cloner un objet.....	57
1.11.4 La méthode __call(méthode, arguments).....	58
1.11.5 L'héritage et les méthodes __set() et __get().....	62
1.11.6 La fonction __autoload().....	63
1.11.7 Méthodes diverses.....	65
1.11.8 La comparaison d'objets.....	69
1.12 Relations Inter-Classes : Association, Agrégation et Composition.....	70
1.12.1 Association binaire de type Père-Fils.....	71
1.12.2 Association binaire.....	73
1.12.3 Association n-aires.....	76
1.12.4 Association avec une classe d'association.....	77

1.12.5 Comparaison Agrégation-Composition.....	78
1.12.6 Agrégation.....	79
1.12.7 Composition.....	82
1.13 POO ET GESTION DES EXCEPTIONS.....	86
1.13.1 Rappel sur la gestion des erreurs avec PHP.....	86
1.13.2 Introduction à la gestion d'exception.....	91
1.13.3 Créez votre propre classe d'Exception.....	93
1.13.4 Levée d'exception automatisée.....	96
1.13.5 Levée d'exception centralisée.....	100

CHAPITRE 1 LES CLASSES ET LES OBJETS

1.1 PRÉSENTATION GÉNÉRALE

1.1.1 Généralités

Les classes sont des représentations abstraites des objets du monde.

Une classe doit représenter les caractéristiques statiques et dynamiques des objets.

Elle les **encapsule**.

Les caractéristiques statiques sont représentées au moyen **d'attributs** ou de **propriétés**.

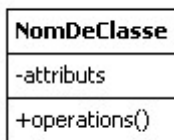
Les caractéristiques dynamiques sont représentées au moyen **d'opérations** ou de **méthodes**.

La POO met en place les mécanismes **d'héritage**.

Pour explorer tous les concepts cf la POO (Programmation Orientée Objet) et la méthode UML (Unified Modeling Language).

1.1.2 Représentation graphique

La représentation graphique (cf UML) est réalisée via un rectangle à 3 compartiments.



1.1.3 Les objets

Un objet est une instantiation d'une classe.

1.1.4 Principales caractéristiques objet de PHP5

Class et new,

This, ::Self, ::parent,

Portées : public, protected, private,

Méthodes __construct et __destruct,

Héritage simple via extends,

Surcharge - éventuellement avec polymorphisme - mais pas de polymorphisme de même niveau,

Propriétés et méthodes static,

Classes abstraites et classes *final*,

Interfaces via Interface et implements,

Méthodes __set, __get pour affecter et récupérer des valeurs d'attributs ... et autres méthodes magiques.

NB : Pour les différences entre PHP4 et PHP5 cf annexe.

1.2 LES CLASSES EN PHP 5

1.2.1 Syntaxe de définition d'une classe

Par convention les noms des classes commencent par une majuscule et sont camélisés.
La classe est enregistrée dans un fichier de même nom.

```
class NomDeClasse
{
}
```

1.2.2 Les propriétés et les méthodes

- Les propriétés

Les propriétés sont déclarées avec le mot réservé **public** ou **private** ou **protected**.

```
QualificateurDePortée $nomDePropriété;
```

This est le pronom qui représente l'objet instancié.

La référence à une propriété ou à une méthode est réalisée avec l'opérateur -> (Notation pointée ☺).

- Les méthodes

Les méthodes reprennent la syntaxe des fonctions avec un qualificateur de portée.

- Remarques

Par convention les noms des attributs/propriétés et des opérations/méthodes commencent par une minuscule et sont camélisés.

- Exemple de classe

Personne
-nom -age
+affecterNom(nom) +affecterAge(age) +recupererNom() +recupererAge()

```
class Personne
{
    // --- Propriétés
    private $nom;
    private $age;

    // --- Méthodes
    public function affecterNom($nom) { $this->nom = $nom; }
    public function recupererNom() { return $this->nom; }
    public function affecterAge($age) { $this->age = $age; }
    public function recupererAge() { return $this->age; }
}
```

1.2.3 Instantiation d'un objet

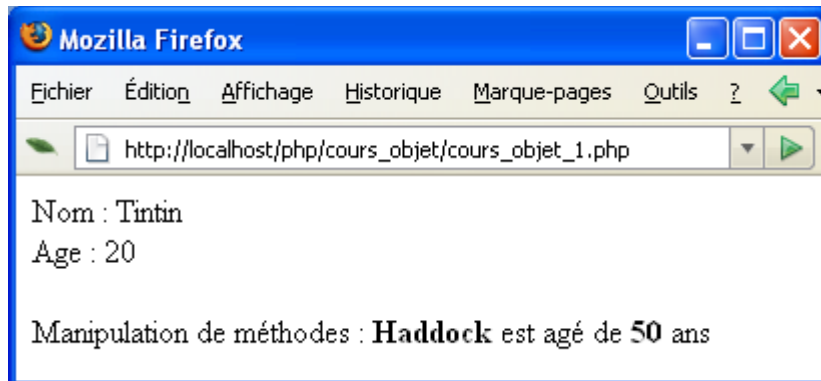
L'instantiation d'un objet à partir d'une classe s'effectue avec le mot clé **new**.
On affecte le nouvel objet à une variable.

```
$variable = new NomDeClasse();
```

- Exemple d'instantiation

```
$tintin = new Personne();
```


1.2.4 Exemple complet



Par convention (Sans aucune obligation) une classe sera stockée dans un fichier php. Le nom du fichier sera identique au nom de la classe.

```
<?php
// --- Personne.php
class Personne
{
    // --- Propriétés
    private $nom;
    private $age;

    // --- Méthodes
    public function affecterNom($nom) { $this->nom = $nom; }
    public function recupererNom() { return $this->nom; }
    public function affecterAge($age) { $this->age = $age; }
    public function recupererAge() { return $this->age; }
}
?>
```

```
<?php
// --- personneUse1.php
require_once("Personne.php");
// --- Instantiation d'un objet et utilisation
$tintin = new Personne();
$tintin->affecterNom("Tintin");
$tintin->affecterAge(20);
echo "Nom : " . $tintin->recupererNom() . "<br />";
echo "Age : " . $tintin->recupererAge() . "<br />";
?>
```

- **Exercice**

Complétez ce code pour qu'il corresponde à l'écran.

1.2.5 Portée des attributs et des méthodes

Les attributs (variables d'instance) peuvent être privés, publics ou protégés.

Les mots réservés correspondants sont les suivants : `private`, `public`, `protected`.

Un attribut ou une méthode **public** est accessible partout : classe, descendants, objets et scripts.

Un attribut ne devrait jamais être public (Principe d'encapsulation).

Un attribut ou une méthode **protected** est accessible au sein de la classe et de ses descendants.

(cf le paragraphe sur l'héritage).

Un attribut ou une méthode **private** est accessible seulement au sein de la classe elle-même.

Exemple :

```
private $nom;  
protected $age;
```

Il en est de même des méthodes.

Aussi bien pour les déclarations que pour la portée. (cf le paragraphe sur l'héritage).

Une méthode private ne sera accessible qu'à l'intérieur de la classe.

Exemple :

```
private function majuscules($asChaine) { return strToUpper($asChaine); }
```

elle sera appelée ainsi dans la classe au niveau de la méthode `affecterNom()` :

```
$this->nom = $this->majuscules($nom);
```

1.2.6 Le polymorphisme en PHP

- Rappel de la définition

Le mot polymorphisme est formé à partir du grec ancien πολλοί (polloí) qui signifie «plusieurs» et μορφος (morphos) qui signifie «forme» (Wikipedia).

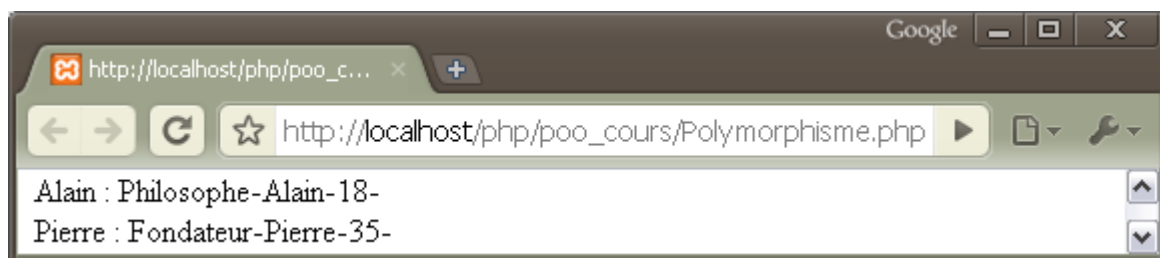
En informatique, le polymorphisme permet d'avoir des signatures de méthodes différentes.

En PHP l'implémentation standard du polymorphisme est impossible. Deux méthodes de même nom ne peuvent co-exister au sein de la même classe.

L'implémentation du polymorphisme en PHP joue sur les paramètres optionnels des fonctions.

- Exemple

Polymorphisme
-nom -prenom -age
+__construct(nom, prenom, age = init)



```
<?php
// --- Polymorphisme.php
// -----
class Polymorphisme
// -----
{
    private $nom, $prenom, $age;

    public function __construct($nom, $prenom, $age=18)
    {
        $this->nom      = $nom;
        $this->prenom   = $prenom;
        $this->age      = $age;
    }
    public function __toString()
    {
        $lsMembres = "";
        foreach($this as $valeur)
        {
            $lsMembres .= "$valeur-";
        }
        return $lsMembres;
    }
}

// -----
// --- Tests
// -----
$alain = new Polymorphisme("Philosophe","Alain");
$pierre = new Polymorphisme("Fondateur","Pierre",35);

echo "<br/>Alain : $alain";
echo "<br/>Pierre : $pierre";
?>
```

1.3 LES ACCESSEURS (__SET ET __GET)

Deux méthodes magiques permettent d'affecter et de récupérer les valeurs des propriétés.

Souvent les set() sont appelés accesseurs et les get() des modificateurs.

Ce sont les méthodes magiques suivantes : __set() et __get().

Attention au double \$\$!

- Syntaxes

```
public function __set($var, $valeur) { $this->$var = $valeur; }
public function __get($var) { return $this->$var; }
```

- Exemple

Personne
#nom -age +instanciations
+ __construct() + __set(valeur) + __get() + __toString() + __clone() + getMembres()

Peu à peu nous allons créer cette classe. Mais pour l'instant seulement les accesseurs.

```
<?php
// --- Personne.php
class Personne
{
    private $nom;
    private $age;

    public function __set($var, $valeur) { $this->$var = $valeur; }
    public function __get($var) { return $this->$var; }
}
?>
```

```
<?php
// --- personneUse2.php
require_once('Personne.php');

// --- Instantiation d'un objet
$tintin = new Personne();

// --- Affectation des valeurs
$tintin->nom = "Tintin";
$tintin->age = "33";

// --- Récupération des valeurs
echo "Monsieur <strong>$tintin->nom</strong> a <strong>$tintin->age</strong>";
?>
```

1.4 LES CONSTRUCTEURS (__CONSTRUCT)

1.4.1 Définition

Le constructeur est une méthode nommée `__construct([args])` qui est appelée automatiquement au moment de l'instantiation d'un objet lorsque l'on utilise l'opérateur **new**.

Il est impossible en PHP pour une classe de posséder plusieurs constructeurs.

Comme d'ailleurs pour n'importe quelle méthode.

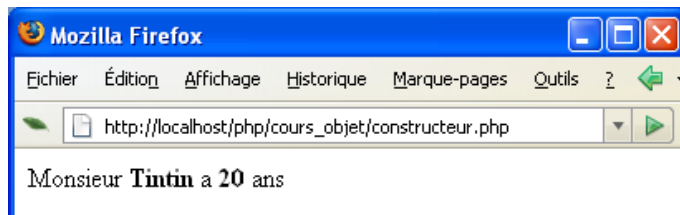
En revanche cela sera possible à des niveaux différents (cf le paragraphe sur l'héritage). Donc pas de polymorphisme possible mais des surcharges oui.

Pour pallier cette absence de polymorphisme, il est possible, comme pour n'importe quelle fonction PHP, d'initialiser les paramètres à des valeurs par défaut.

```
public function __construct($nom = 'Dupont', $age = 18)
```

1.4.2 Exemple

On ajoute un constructeur à la classe `Personne`. Les paramètres de cette méthode sont initialisés.



```
<?php
// --- Personne.php
class Personne
{
    private $nom;
    private $age;
    // --- Le constructeur
    public function __construct($nom = 'Dupont', $age = 18)
    {
        $this->nom = $nom;
        $this->age = $age;
    }
    // --- Autres méthodes
    public function __set($var, $valeur) { $this->$var = $valeur; }
    public function __get($var) { return $this->$var; }
}
?>
```

```
<?php
// --- personneUse3.php
require_once("Personne.php");
// --- Instantiation d'un objet et utilisation
$tintin = new Personne("Tintin", 20);
echo "Monsieur <strong> $tintin->nom</strong> a <strong> $tintin->age</strong> ans<br />";
?>
```

1.5 LES DESTRUCTEURS (__DESTRUCT)

Comme pour les constructeurs une méthode **__destruct()** fera office de destructeur. Elle sera automatiquement appelée au moment de la désinstanciation de l'objet (Fin de script) ou bien lors de l'utilisation de l'instruction **unset(\$objet)**. Cf plus loin un exemple avec les classes composées et les classes agrégées.

Ville
-cp -ville
+__construct(cp, ville) +__destruct() +__set(valeur) +__get()

```
<?php
// --- Ville.php
class Ville
{
    private $cp;
    private $ville;

    public function __construct($cp, $ville)
    {
        $this->cp    = $cp;
        $this->ville = $ville;
        echo "<br />Constructeur";
    }

    public function __get($var) { return $this->$var; }

    public function __destruct()
    {
        echo "<br />Destructeur";
    }
}
?>
```

```
<?php
// --- villeUse1.php
require_once("Ville.php");
$paris = new Ville("75", "Paris");
echo "<br />$paris->ville";

unset($paris);

echo "<br />Il se passe encore quelque chose ...";
?>
```

1.6 L'HÉRITAGE

1.6.1 Définition

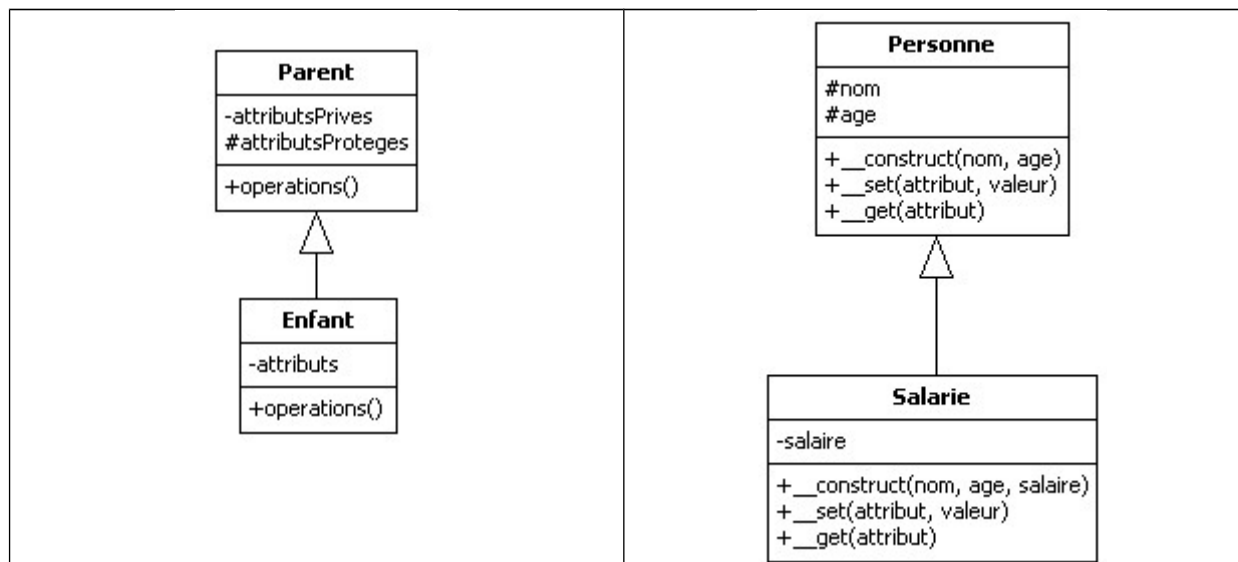
L'héritage est cette possibilité pour une classe d'hériter des attributs et méthodes d'une classe parent.

L'héritage est une spécialisation.

L'héritage simple est supporté en PHP. L'héritage multiple non.

La classe enfant hérite donc des attributs et méthodes du parent (mais seuls **les attributs public et protected sont accessibles directement à partir des descendants**) et possède elle-même ses propres attributs et méthodes.

1.6.2 Représentation graphique



1.6.3 Syntaxe

C'est avec le mot clé **extends** que l'on spécifie l'héritage.

```
Class Enfant extends Parent { .... }
```

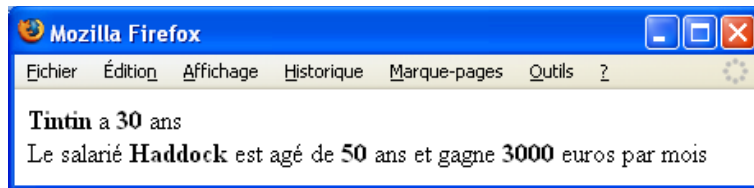
Remarque :

Chaque classe doit posséder un constructeur.

Pour exécuter le constructeur du parent à partir du constructeur de l'enfant (ou une autre méthode magiques) :

```
parent::__construct($paramètre1 [, $paramètre2]);
```


1.6.4 Exemple complet



```
<?php
// --- Personne.php
class Personne
{
    // --- Propriétés
    protected $nom;
    protected $age;

    // --- Méthodes
    public function __construct($nom, $age)
    {
        $this->nom = $nom;
        $this->age = $age;
    }
    public function __set($var, $valeur) { $this->$var = $valeur; }
    public function __get($var) { return $this->$var; }
}
?>
```

```
<?php
// --- Salarie.php
require_once("Personne.php");
class Salarie extends Personne
{
    private $salaire;
    public function __construct($nom, $age, $salaire)
    {
        $this->nom = $nom;
        $this->age = $age;
        $this->salaire = $salaire;
    }
    public function __set($var, $valeur) { $this->$var = $valeur; }
    public function __get($var) { return $this->$var; }
}
?>
```

salarieUse.php

```
<?php
header("Content-Type: text/html; charset=UTF-8");
require_once("Salarie.php");
// --- Instantiation d'un objet et utilisation
$tintin = new Personne("Tintin", 30);
echo "$tintin->nom a $tintin->age ans";

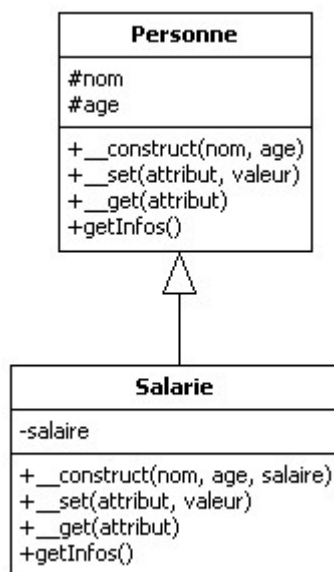
$haddock = new Salarie("Haddock", 50, 3000);
echo "<br />Le salarié $haddock->nom est âgé de $haddock->age ans et gagne
$haddock->salaire euros";
?>
```

1.6.5 Héritage et surcharge

Il est possible dans la chaîne d'héritage d'avoir plusieurs méthodes de même nom, de signatures identiques ou différentes.



Reprenons les classes précédentes et ajoutons les méthodes **getInfos()** :



La méthode **getInfos()** est présente aux deux niveaux.

La méthode de l'enfant peut-être de même signature ou de signature différente.

C'est toujours la méthode de classe la plus proche de l'objet qui est exécutée même si les signatures sont différentes donc il n'y a pas de polymorphisme possible même à des niveaux différents.

La méthode `getInfos()` pourrait solliciter la méthode du parent selon la syntaxe **parent::nomDeMethode()**.

NB : la méthode `__construct()` peut aussi être surchargée.

- Script

```
<?php
// --- Personne.php
class Personne
{
    // --- Propriétés
    protected $nom;
    protected $age;

    // --- Méthodes
    public function __construct($nom, $age)
    {
        $this->nom = $nom;
        $this->age = $age;
    }
    public function __set($var, $valeur) { $this->$var = $valeur; }
    public function __get($var) { return $this->$var; }
    public function getInfos() { return "$this->nom a $this->age ans"; }
}
?>
```

```
<?php
// --- Salarie.php
require_once("Personne.php");
class Salarie extends Personne
{
    private $salaire;
    public function __construct($nom, $age, $salaire)
    {
        $this->nom      = $nom;
        $this->age       = $age;
        $this->salaire = $salaire;
    }
    public function __set($var, $valeur) { $this->$var = $valeur; }
    public function __get($var) { return $this->$var; }
    public function getInfos() { return "$this->nom a $this->age ans et
gagne $this->salaire euros"; }
}
?>
```

```
<?php
// --- personneUse5.php
header("Content-Type: text/html; charset=UTF-8");

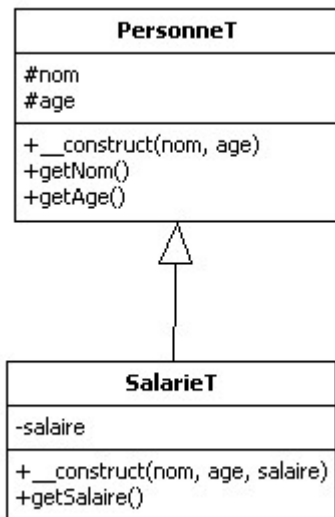
require_once("Salarie.php");

$tintin = new Personne("Tintin", 30);
echo "<br />", $tintin->getInfos();

$haddock = new Salarie("Haddock", 50, 3000);
echo "<br />", $haddock->getInfos();
?>
```

1.6.6 Héritage et transtypage

Admettons - et simplifiées - les classes Personne et Salarie.



Leurs codes :

```

<?php
class PersonneT
{
    protected $nom;
    protected $age;

    public function __construct($nom, $age)
    {
        $this->nom = $nom;
        $this->age = $age;
    }
    public function __set($var, $valeur) { $this->$var = $valeur; }
    public function __get($var) { return $this->$var; }
}
?>
  
```

```

<?php
require_once("PersonneT.php");
class SalarieT extends PersonneT
{
    private $salaire;

    public function __construct($nom, $age, $salaire)
    {
        $this->nom      = $nom;
        $this->age      = $age;
        $this->salaire = $salaire;
    }
    public function __set($var, $valeur) { $this->$var = $valeur; }
    public function __get($var) { return $this->$var; }
}
?>
  
```

Copies : possible de Salarie vers Personne, impossible de Personne vers Salarie.

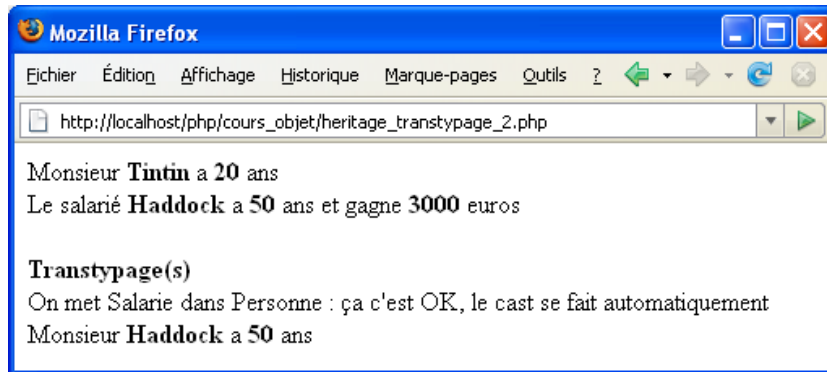
PersonneT		SalarieT
(+)nom (+)age	--- KO -->	(+)nom (+)age (+)salaire
(+)_construct(nom, age) (+)getNom() (+)getAge() (+)toSalarie(oSalarie)	<- OK --	(+)_construct(nom, age) (+)getNom() (+)getAge() (+)_construct(nom, age, salaire) (+)getSalaire() (+)setSalaire()

Codes de base

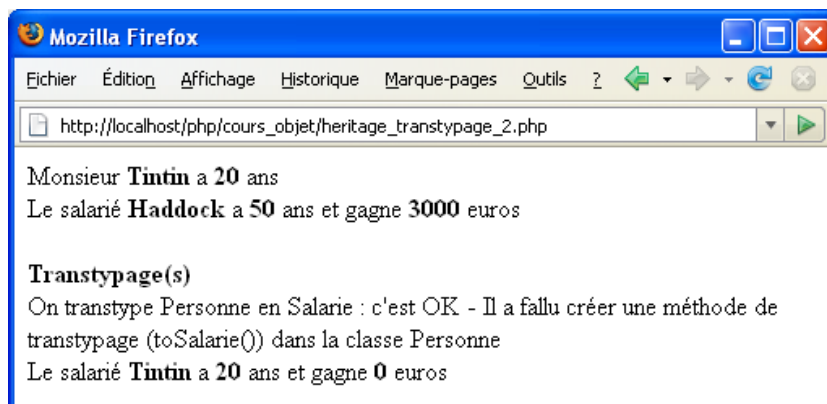
```
<?php
    require_once("PersonneT.php");
    require_once("SalarieT.php");
    // --- On met PersonneT dans SalarieT : erreur
    // --- Il faut creer une méthode pour transtyper
    // $haddock = $tintin;
    // print("<br />Le salarié " . $haddock->getNom() . " a " . $haddock->getAge()
    . " ans" . " et gagne " . $haddock->getSalaire() . " euros");

    // --- On met Salarie dans Personne : ca c'est OK, le cast se fait
    automatiquement
    $tintin = $haddock;
    print("<br />On met Salarie dans Personne : ça c'est OK, le cast se fait
    automatiquement");
    print("<br />Monsieur " . $tintin->getNom() . " a " . $tintin->getAge() . "
    ans<br />");
?>
```

De SalarieT vers PersonneT : natif.



De PersonneT vers SalarieT : via une méthode supplémentaire dans la classe PersonneT!



La méthode toSalarie() dans PersonneT

```
public function toSalarie($aoSalarie)
{
    $aoSalarie->nom = $this->nom;
    $aoSalarie->age = $this->age;
    $aoSalarie->setSalaire(0);
}
```

avec la méthode setSalaire() dans SalarieT

```
public function setSalaire($salaire) { $this->salaire = $salaire; }
```

et le code de transfert

```
print("<br />On transtype Personne en Salarie : c'est OK - Il a fallu créer une
méthode de transtypage (toSalarie()) dans la classe Personne");
$tintin->toSalarie($haddock);
print("<br />Le salarié <strong>" . $haddock->getNom() . " </strong> a <strong> " .
$haddock->getAge() . " </strong>ans" . " et gagne <strong>" . $haddock-
>getSalaire() . " </strong> euros");
```

1.7 COMPLÉMENTS

1.7.1 Les constantes de classe

Une constante de classe est une constante utilisable par tout script sans instantiation d'objet. C'est une valeur de niveau applicatif.

Alors que pour une constante de script on utilise la syntaxe suivante :

define("NOM_DE_CONSTANTE",valeur);

Pour une constante de classe vous la déclarez ainsi : **const** NOM_DE_CONSTANTE = valeur;

Vous l'utilisez ainsi : NomDeLaClasse::NOM_DE_CONSTANTE.

Elle est publique et c'est une statique.

Mais il faut que tous les éléments de la classe soient statiques ainsi la classe est statique autrement l'utilisation s'effectue selon les règles exposées au paragraphe suivant.

```
<?php
// --- Constante.php
class Constante
{
    const PII = 3.14;
}
?>
```

```
<?php
// --- ConstanteUse.php
echo "<br />Constante (Appel statique) : " . Constante::PII;
?>
```

1.7.2 Les opérateurs parent, self et ::

L'opérateur ::, appelé opérateur de résolution de portée, permet d'accéder à des éléments **static** externes ou internes.

Externes :

L'opérateur de résolution :: permet de d'accéder à un élément **static** (constante, variable statique ou méthode statique) d'une classe.

```
NomClasse::NOMDECONSTANTE  
NomClasse::$nomDeVariableStatique  
NomDeClasse::nomDeMethodeStatique()
```

Internes :

Le mot **self** et l'opérateur de résolution permettent d'accéder aux éléments **static** de la classe elle-même.

```
self::NOMDECONSTANTE  
self::$nomDeVariableStatique  
self::nomDeMethodeStatique()
```

Cf les exemples à la page suivante.

Le mot **parent** et l'opérateur de résolution permettent d'accéder aux **attributs** et **méthodes static** du parent.

```
parent::NOMDECONSTANTE  
parent::$nomDeVariable  
parent::nomDeMethode()
```

Cf les exemples au paragraphe suivant.

1.7.3 Méthode statique, variable statique

Les éléments statiques (**static**) sont des éléments de classe et non pas des éléments d'instances.

L'espace mémoire utilisé est un espace local-global commun à toutes les instances de la classe.

Les éléments statiques (**static**) d'une classe sont accessibles sans instantiation.

Souvent on implémente des méthodes statiques dans des classes techniques (Classes arithmétiques, classes String, ...). Cf la classe Java.Lang.Math de Java.

Un attribut statique permet de récupérer une valeur commune à plusieurs instances (nombre d'instances d'une classe par exemple).

La syntaxe de déclaration d'une méthode statique est la suivante :

```
public static function méthode() ...
```

La syntaxe d'appel est la suivante :

```
NomDeClasse::méthode();
```

Une propriété statique est une locale-globale. Elle peut être private ou public. Le qualificateur est **static**.

Une statique privée est utilisée dans une méthode pour des appels récursifs.

La syntaxe de déclaration d'une propriété statique est la suivante :

```
public static $nomDePropriete [ = valeurInitiale];
```

La syntaxe d'appel est la suivante :

```
NomDeClasse::$nomDePropriete;
```

- Exemple self et static

PII est une constante, surfaceCercle est un attribut statique, calculerSurfaceCercle() et calculerPerimetreCercle() sont des méthodes statiques.

Geometrie
+PII +surfaceCercle
+calculerSurfaceCercle(rayon) +calculerPerimetreCercle(rayon)

```
<?php
// --- Geometrie.php
class Geometrie
{
    const PII = 3.14; // --- Une constante
    public static $surfaceCercle; // --- Une propriété statique

    // --- Une méthode statique
    public static function calculerSurfaceCercle($rayon)
    {
        self::$surfaceCercle = $rayon * $rayon * self::PII;
    }

    // --- Une méthode statique
    public static function calculerPerimetreCercle($rayon)
    {
        return $rayon * 2 * self::PII;
    }
}
?>
```

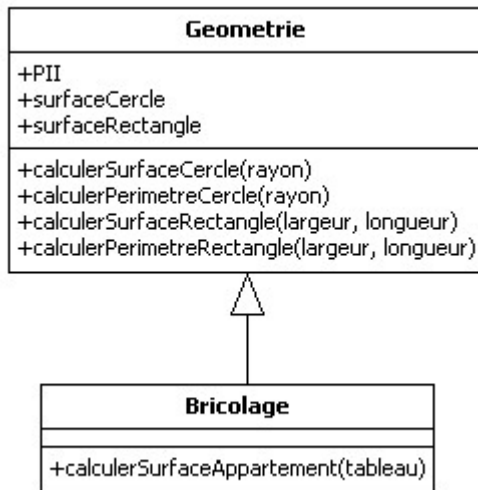
```
<?php
// --- geometrieUse.php
require_once("Geometrie.php");
echo "<br />PII (Constante) : ", Geometrie::PII;

echo "<br />Perimetre (Methode statique) : ",
Geometrie::calculerPerimetreCercle(10);
Geometrie::calculerSurfaceCercle(10);

echo "<br />Surface (Attribut statique) : ", Geometrie::$surfaceCercle;
?>
```

La classe Bricolage : un héritage d'une "statique".

Elle hérite de la classe [Geometrie].



En premier lieu on ajoute les méthodes statiques `calculerSurfaceRectangle()` et `calculerPerimetreRectangle()` puis on ajoutera une classe `Bricolage` qui héritera de celle-ci et qui utilisera les méthodes statiques de la classe parent.

Ajoutez ceci à la classe `Geometrie` :

```

public static $surfaceRectangle;

public static function calculerSurfaceRectangle($largeur, $longueur)
{
    self::$surfaceRectangle = $largeur * $longueur;
}
public static function calculerPerimetreRectangle($largeur, $longueur)
{
    return ($largeur * 2) + ($longueur * 2);
}
  
```

- Exemple parent et static

```
<?php
// --- Bricolage.php
require_once("Geometrie.php");

class Bricolage extends Geometrie
{
    public static function calculerSurfaceAppartement($tableau)
    {
        $surfaceTotale = 0;
        foreach($tableau as $largeur => $longueur)
        {
            parent::calculerSurfaceRectangle($largeur, $longueur);
            $surfaceTotale += parent::$surfaceRectangle;
        }
        return $surfaceTotale;
    }
}

?>
```

```
<?php
// --- BricolageUse.php
require_once("Bricolage.php");

echo "<br />Perimetre (Methode statique) : ",
Bricolage::calculerPerimetreRectangle(5,10);
Bricolage::calculerSurfaceRectangle(5,10);
echo "<br />Surface (Variable statique) : ", Bricolage::$surfaceRectangle;
echo "<br />Surface Appartement : ",
Bricolage::calculerSurfaceAppartement(array(5=>10, 4=>6));

?>
```

- Comptage du nombre d'instances d'une classe.

L'objectif est d'afficher le nombre d'instances d'une classe.

Reprenons la classe `Personne` et ajoutons une **variable statique** nommée `[instanciations]`.

Dans le constructeur la variable statique est incrémentée : **`self::$instanciations ++`**;

Lors de la récupération les valeurs les attributs sont récupérées avec la notation fléchée appliquée à un objet : **`$p1->prenom`**.

Le nombre d'instances est récupéré via l'opérateur de résolution de portée appliqué à la classe : **`Personne::$instanciations`**.

Cf le pattern Singleton.

Personne
#nom -age +instanciations
+__set(valeur) +__get()

```
<?php
// --- Personne.php
class Personne
{
    private $nom, $prenom, $age;
    public static $instanciations;

    public function __construct($prenom, $nom, $age)
    {
        $this->prenom = $prenom;
        $this->nom     = $nom;
        $this->age     = $age;
        self::$instanciations++;
    }
    public function __set($var, $valeur) { $this->$var = $valeur ; }
    public function __get($var) { return $this->$var ; }
}
?>
```

```
<?php
// --- personneUseStatic.php
require_once("Personne.php");

$p1 = new Personne("Albert", "Tintin", 30);
$p2 = new Personne("Laetitia", "Casta", 32);

echo "$p1->prenom $p1->nom a $p1->age ans<br/>";
echo "$p2->prenom $p2->nom a $p2->age ans<br/>";

echo "Il y a ", Personne::$instanciations, " personne(s)";
?>
```

Affichage

Albert Tintin a 30 ans
 Laetitia Casta a 32 ans
 Il y a 2 personne(s)

- **Exercice**

Créez une classe **Statistique** avec des **méthodes statiques** qui calculent le compte, le max, le min, la moyenne, la somme, la variance et l'écart type des éléments d'un tableau de numériques.

Cf la classe Java.Lang.Math de Java.

Statistique
+somme(t) +moyenne(t) +max(t) +min(t) +compte(t) +variance(t) +ecartType(t)

La variance et l'écart-type sont deux indices de dispersion.

La variance : c'est la moyenne des carrés des écarts à la moyenne.

La formule est la suivante : moyenne des carrés des écarts à la moyenne

Variance = Moyenne($\sum (Moyenne(x) - x)^2$)).

L'écart-type : la racine carrée de la variance : $\sqrt{\text{Variance}}$.

- Corrigé

```

<?php
// --- Statistique.php
class Statistique
{
    // -----
    public static function somme($aiT)
    // -----
    {
        return array_sum($aiT);
    }
    // -----
    public static function moyenne($aiT)
    // -----
    {
        return self::somme($aiT) / count($aiT);
    }
    // -----
    public static function max($aiT)
    // -----
    {
        $liMax = $aiT[0];
        for($i=0; $i<count($aiT); $i++)
        {
            if($aiT[$i] > $liMax) $liMax = $aiT[$i];
        }
        return $liMax;
    }
    // -----
    public static function min($aiT)
    // -----
    {
        $liMin = $aiT[0];
        for($i=0; $i<count($aiT); $i++)
        {
            if($aiT[$i] < $liMin) $liMin = $aiT[$i];
        }
        return $liMin;
    }
    // -----
    public static function compte($aiT)
    // -----
    {
        return count($aiT);
    }
    // -----
    public static function variance($aiT)
    // -----
    {
        // Cf float stats_variance ( array $a [, bool $sample = false ] )
        // C'est la moyenne des carrés des écarts à la moyenne
        $tEcartCarres = array();
        $moyenne = self::moyenne($aiT);
        for($i=0; $i<count($aiT); $i++)
        {
            $ecart = abs($aiT[$i] - $moyenne);
            $tEcartCarres[$i] = $ecart * $ecart;
        }
        $variance = self::moyenne($tEcartCarres);
        return $variance;
    }
    // -----
    public static function ecartType($aiT)
    // -----
    {

```

```
        return sqrt(self::variance($aiT));
    }
}
?>
```

```
<?php
// --- statUse.php
require_once("Statistique.php");

$t = array(3,5,7);

echo "<br/>Compte : "      . Statistique::compte($t);
echo "<br/>Somme : "       . Statistique::somme($t);
echo "<br/>Moyenne : "    . Statistique::moyenne($t);
echo "<br/>Min : "        . Statistique::min($t);
echo "<br/>Max : "        . Statistique::max($t);
echo "<br/>Variance : "   . Statistique::variance($t);
echo "<br/>Ecart type : " . Statistique::ecartType($t);
?>
```


1.7.4 Classe abstraite

- Définition

Une classe abstraite est une classe racine non instanciable. L'objectif est de représenter une factorisation, une généralisation. Dans de nombreux cas il est souhaitable de mettre certaines méthodes magiques (par exemple les `__get` et les `__set`) dans une classe abstraite.

La syntaxe est la suivante :

```
abstract class Abstraite { ... }
```

- Remarques

Si vous essayez d'instancier une classe abstraite vous aurez le message suivant :

Fatal error: Cannot instantiate abstract class PersonneA in
C:\xampp\htdocs\php\cours_objet\abstraite.php on line **59**

La classe possède quand même une méthode `__construct()` - qui est protected - qui sera utilisée par les classes héritières.

Vous noterez la déclaration des méthodes `__set()` et `__get()` au niveau de la classe abstraite et l'utilisation directe par les objets issus de classes dérivées.

```
print("<br />$prof->prenom $prof->nom est spécialiste en " . $prof->getSpecialisation());
```

NB : en PHP5 les méthodes d'une classe abstraite peuvent être utilisées par des appels statiques !!!

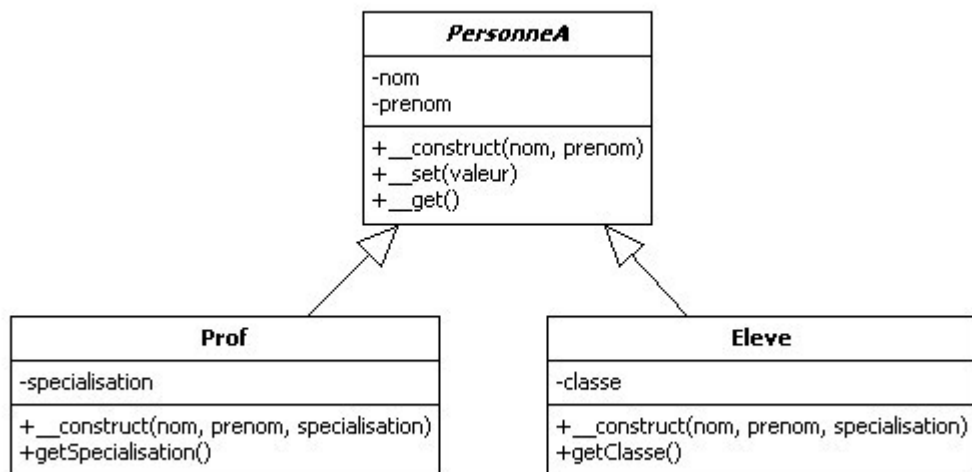
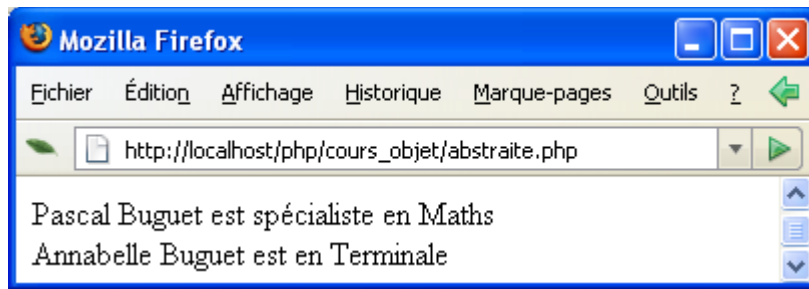
Une méthode aussi peut-être abstraite avec la syntaxe suivante.

abstract protected function nomDeMethode();

Elle ne peut pas être implémentée dans la classe de base, elle fournit une signature, comme les interfaces, et **devra** l'être dans chaque classe descendante.

Une classe qui possède au moins une méthode abstraite doit être déclarée abstraite.

- Exemple



```

<?php
// --- PersonneA.php
abstract class PersonneA
{
    private $prenom;
    private $nom;

    protected function __construct($prenom, $nom)
    {
        $this->prenom = $prenom;
        $this->nom     = $nom;
    }
    public function __set($var, $valeur) { $this->$var = $valeur; }
    public function __get($var) { return $this->$var; }
}

?>

<?php
// --- Prof.php
require_once("PersonneA.php");
class Prof extends PersonneA
{
    private $specialisation;

    public function __construct($prenom, $nom, $specialisation)
    {
        $this->specialisation = $specialisation;
        parent::__construct($prenom, $nom);
    }
    public function getSpecialisation() { return $this->specialisation; }
}

?>

<?php
// --- Eleve.php
require_once("PersonneA.php");
class Eleve extends PersonneA
{
    private $classe;

    public function __construct($prenom, $nom, $classe)
    {
        $this->classe = $classe;
        parent::__construct($prenom, $nom);
    }
    public function getClasse() { return $this->classe; }
}

?>

<?php
// --- profEleveTest.php
require_once("Prof.php");
require_once("Eleve.php");

// $personne = new PersonneA("P","P"); // --- Interdit
$prof  = new Prof("Pascal","Buguet","Maths");
$eleve = new Eleve("Annabelle", "Buguet", "Terminale");

echo "<br />$prof->prenom $prof->nom est specialiste en " . $prof-
>getSpecialisation();
echo "<br />$eleve->prenom $eleve->nom est en " . $eleve->getClasse();

?>

```

1.7.5 Classes et méthodes *Final*

- Classe *Final*

Une classe *final* est une classe qui n'est pas héritable. C'est la fin d'une chaîne d'héritage.

```
final class NomDeClasse { ... }
```

Si vous essayez d'étendre une classe finale vous obtiendriez ce message :

Fatal error: Class HeriteFinal may not inherit from final class (Finale) in
C:\wamp\www\php\cours_objet\finale.php on line **12**

- Méthode *Final*

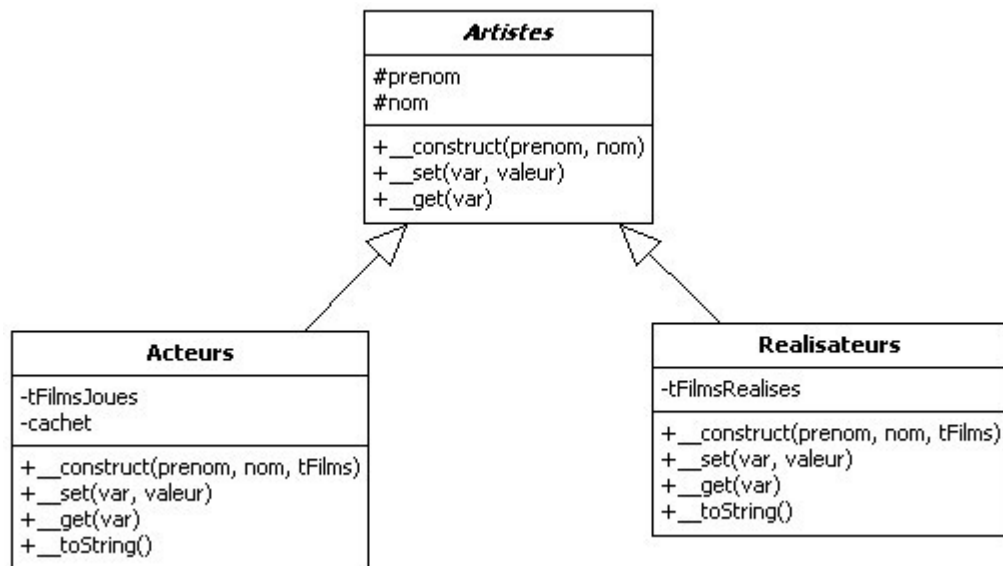
Une méthode *final* n'est pas surchargeable.

```
final public function NomDeFonction( ... ) { ... }
```

Note : lorsque nous étudierons les Exceptions, nous verrons que dans la classe Exception de PHP il y a un certain nombre de méthodes "final".

- **Exercice**

Implémentez le schéma suivant.



- **Corrigé**

Cf ExoAbstractFinal.php.

1.8 LINÉARISER LES OBJETS (SÉRIALISER, DÉSÉRIALISER)

Linéariser un objet c'est le transformer en une chaîne de caractères.

Il est possible de linéariser un objet dans une variable, dans un enregistrement d'un fichier texte, dans un élément d'un fichier XML, dans une table d'une base de données, dans une variable de session (pour pouvoir ainsi passer un objet d'une page à une autre), dans un cookie,

La sérialisation sert aussi à transporter les objets sur le réseau.

Si l'on stocke les linéarisations dans un fichier ou une BD on rend **persistants** les objets.

Pour le faire il faut au préalable transformer chaque objet en une chaîne de caractères (**sérialiser**) et faire l'inverse (**désérialiser**) pour récupérer les objets stockés.

La sérialisation stocke les propriétés pas les méthodes.

La sérialisation s'effectue avec la fonction **serialize(objet)**.

La désérialisation s'effectue avec la fonction **unserialize("chaîne")**.

Les méthodes serialize() et unserialize() font appel – si elles existent – aux méthodes magiques __sleep() et __wakeup(). Cf plus loin la sérialisation BD.

1.8.1 La sérialisation

Serialize() : objet -> chaîne

La sérialisation c'est la transformation d'un objet en une chaîne de caractères (avec un format particulier correspondant à la structure de la classe).

```
$chaîne = serialize($objet).
```

Exemple de chaîne de caractères suite à une sérialisation :

O:8:"personne":2:{s:3:"nom";s:6:"Tintin";s:3:"age";i:20;}

Notes :

O:8 caractères du nom de la classe

2 : 2 propriétés

s:3 propriété 1 dont le nom est de 3 caractères

s6 valeur de la propriété 1 dont le type est String

s:3 propriété 2 dont le nom est de 3 caractères

i:20 valeur de la propriété 2 dont le type est Integer

1.8.2 La désérialisation

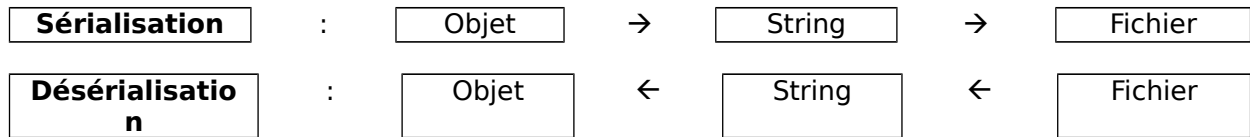
unserialize() : chaîne -> objet

La désérialisation c'est la transformation d'une chaîne de caractères en un objet.

```
$objet = unserialize($chaîne).
```

Note : si vous gérez plusieurs objets dans le même script PHP "reconnaît automatiquement la classe de l'objet désérialisé.

1.8.3 S rialisation et d s rialisation Fichier



Dans cet exemple il y a l'include avec la d finition de la classe, le script de s rialisation, le script de d s rialisation et le fichier de sauvegarde.

La classe contient un constructeur.

La s rialisation se fait par transformation puis stockage dans un fichier texte ouvert en cr ation.

La d s rialisation se fait par lecture du fichier dans une string puis par transformation. Ensuite l'appel d'une m thode est effectu  pour tester la bonne r cup ration.

- La s rialisation : Serialisation.php

```
<?php
// --- Serialisation.php
header("Content-Type: text/html; charset=UTF-8");
require_once("Personne.php");

$p = new Personne("Herg ",50);
file_put_contents("sauve_personne.txt", serialize($p));
echo "S rialisation r alis e";
?>
```

- La d s rialisation : Deserialisation.php

```
<?php
// --- Deserialisation.php
header("Content-Type: text/html; charset=UTF-8");
require_once("Personne.php");
// --- File lit un fichier texte dans un tableau et Implode construit une
cha ne   partir d'un tableau en concat nant les diff rents  l ments
// --- Comme il n'y a qu'une seule ligne cela renvoie une seule cha ne.
// --- Mettre "\n" comme argument est  quivalent car il y a une seule ligne.
// $nr = implode("",file("sauve_personne.txt"));
$nr = file_get_contents("sauve_personne.txt");
$p = unserialize($nr);
echo "<br />Nom : $p->nom";
?>
```

- Le contenu du fichier de sauvegarde : sauve_personne.txt

```
O:8:"Personne":2:{s:13:" Personne nom";s:6:"Herg ";s:13:" Personne age";i:50;}
```

1.8.4 S rialisations et d s rialisations Fichier

Les deux scripts de s rialisation et de d s rialisation stockent puis affichent le contenu du fichier de sauvegarde.

La sauvegarde est effectu e par une s rialisation suivie par une sauvegarde dans un fichier texte. La cha ne est stock e et suivie d'un retour chariot.

La r cup ration est effectu e par une lecture du fichier dans un tableau avec la fonction `File()` puis par une boucle sur le tableau en d s rialisant et enfin par un affichage des propri t s via des m thodes.

- La s rialisation : `Serialisations.php`

```
<?php
// --- Serialisations.php
header("Content-Type: text/html; charset=UTF-8");
require_once("Personne.php");
// --- Instantiation d'objets, serialisations et sauvegardes
$p1 = new Personne("Haddock",50);
$p2 = new Personne("Tintin",25);
$p3 = new Personne("Casta",33);

$lsContenu = serialize($p1) . "\r\n";
$lsContenu .= serialize($p2) . "\r\n";
$lsContenu .= serialize($p3) . "\r\n";

file_put_contents("sauve_personnes.txt", $lsContenu);

print("Les personnes sont sauvegard es");
?>
```

- La d s rialisation : `Deserialisations.php`

```
<?php
// --- Deserialisations.php
require_once("Personne.php");
$TPersonnes = file("sauve_personnes.txt");
for($i=0; $i<count($TPersonnes); $i++)
{
    $p = unserialize($TPersonnes[$i]);
    echo "$p->nom a $p->age ans<br />";
}
?>
```

- Le contenu du fichier de sauvegarde : `sauve_personnes.txt`

```
O:8:"Personne":2:{s:13:" Personne nom";s:7:"Haddock";s:13:" Personne age";i:50;}
O:8:"Personne":2:{s:13:" Personne nom";s:6:"Tintin";s:13:" Personne age";i:25;}
O:8:"Personne":2:{s:13:" Personne nom";s:5:"Casta";s:13:" Personne age";i:33;}
```


- Exercice

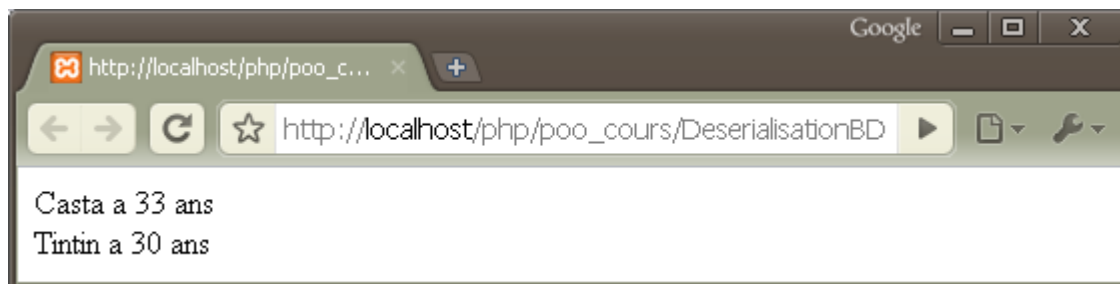
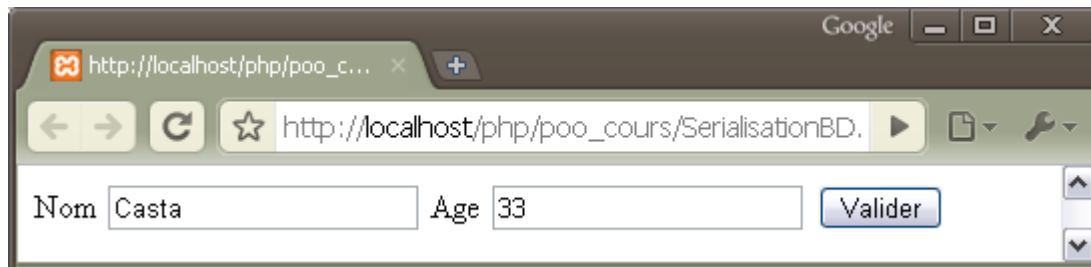
Sérialisations et désérialisations BD.

Faites saisir dans un formulaire des données (Nom et âge) puis stockez le prénom et l'âge dans une table d'une BD (**personnes(personne varchar(255))**).

Répétez l'opération.

Visualisez ensuite.

La classe Personne est utilisée.

Les écrans

A vous !

- **Corrigé**

SerialisationsBD.php

```
<?php
// --- SerialisationBD.php
header("Content-Type: text/html; charset=UTF-8");
if(isset($_GET["nom"]))
{
    require_once("Personne.php");

    $p    = new Personne($_GET["nom"],$_GET["age"]);
    $o    = serialize($p);
    mysql_connect("localhost","root","");
    mysql_select_db("cours");
    $sql  = "INSERT INTO personnes(personne) VALUES('$o')";
    mysql_query($sql);
    echo "Serialisation realisee";
}

?>

<form action="" method="get">
    <label>Nom </label>
    <input name="nom" type="text" value="Casta" />
    <label>Age </label>
    <input name="age" type="text" value="33" />
    <input type="submit" />
</form>
```

DeserialisationsBD.php

```
<?php
// --- DeserialisationBD.php
header("Content-Type: text/html; charset=UTF-8");
require_once("Personne.php");

$lien = mysql_connect("localhost","root","");
mysql_select_db("cours");
$sql  = "SELECT personne FROM personnes";
$curseur = mysql_query($sql);
while($enr=mysql_fetch_row($curseur))
{
    $p = unserialize($enr[0]) ;
    echo "$p->nom a $p->age ans<br />";
}

?>
```

1.8.5 Les méthodes `__sleep()` et `__wakeup()`.

Les méthodes `__sleep()` et `__wakeup()` permettent, lors d'opérations de sérialisation/désérialisation, d'effectuer des opérations particulières.

- `__sleep()`

La fonction **`serialize()`** s'assure que votre classe possède la méthode magique `__sleep()`. Si c'est le cas, cette méthode est appelée **avant** toute linéarisation.

Elle peut alors nettoyer l'objet et sélectionner les propriétés qui doivent être sauvées. Cette fonction est pratique si vous avez de grands objets qui n'ont pas besoin d'être sauvé entièrement.

Dans ce cas la méthode `__sleep()` doit renvoyer un tableau contenant les noms des propriétés à sauvegarder.

Le but de la méthode `__sleep()` est aussi de permettre de se connecter à la BD ou de fermer proprement la connexion à une base de données, de valider les requêtes, de finaliser toutes les actions commencées.

Dans l'exemple qui suit la sérialisation fait appel à la méthode `__sleep()` pour assurer la connexion à la base et le "nettoyage" de l'objet.
La propriété "age" ne sera pas sauvegardée.

- `__wakeup()`

A l'inverse, **`unserialize()`** s'assure de la présence de la méthode magique `__wakeup()`. Si elle existe, cette méthode est exécutée pendant la désérialisation.

Dans l'exemple qui suit la méthode `__wakeup()` les variables qui n'ont pas été sauvées sont initialisées.

```
<?php
class Personne
{
    private $prenom, $nom, $age;

    public function __construct($prenom, $nom, $age)
    {
        $this->prenom = $prenom;
        $this->nom     = $nom;
        $this->age     = $age;
    }
    public function __set($var, $valeur) { $this->$var = $valeur; }
    public function __get($var) { return $this->$var; }
    public function __sleep() // --- Appelée avant de la sérialisation
    {
        $sauver = array();
        $sauver[0] = "prenom";
        $sauver[1] = "nom";
        // $sauver[2] = "age"; // La propriété ne sera pas sérialisée
        $this->connexion();
        return $sauver;
    }
    public function __wakeup() // --- Appelée pendant la désérialisation
    {
        $this->age = "Age inconnu";
    }
    private function connexion()
    {
        mysql_connect("localhost","root","") or die("Erreur connexion
serveur");
        mysql_select_db("cours") or die("Erreur sélection BD");
    }
}
?>
```

SerialisationsPlusBD.php

```
<?php
header("Content-Type: text/html; charset=UTF-8");
if(isset($_GET["prenom"]))
{
    require_once("Personne.php");

    $p    = new Personne($_GET["prenom"], $_GET["nom"], $_GET["age"]);
    $os   = serialize($p);
    $sql  = "INSERT INTO personnes(personne) VALUES('$os')";
    mysql_query($sql);
    echo "Sérialisation réalisée";
}
?>
<form action="" method="get">
    <label>Prénom </label>
    <input name="prenom" type="text" value="Laetitia" />
    <label>Nom </label>
    <input name="nom" type="text" value="Casta" />
    <label>Age </label>
    <input name="age" type="text" value="30" />
    <input type="submit" />
</form>
```

DeserialisationsPlusBD.php

```
<?php
header("Content-Type: text/html; charset=UTF-8");
require_once("Personne.php");

mysql_connect("localhost", "root", "");
mysql_select_db("cours");
$sql      = "SELECT personne FROM personnes";
$curseur = mysql_query($sql);
while($enr=mysql_fetch_row($curseur))
{
    $p = unserialize($enr[0]);
    echo "$p->prenom $p->nom a $p->age ans<br />";
}
?>
```

- [Exercice](#)

"Internaliser" l' action de sérialisation en créant une nouvelle méthode nommée `serialiser()`.

- [Corrigé](#)

SerialisationsDeserialisationsBD.php.

1.9 LES INTERFACES

L'objectif des interfaces est de permettre à plusieurs classes de partager le même protocole de comportement, sans avoir de liens d'héritage entre elles. C'est un contrat.

Une interface est un ensemble de spécifications de méthodes qui **devra** être implémenté par une classe.

Une interface **ne contient que des signatures de méthodes**. Les méthodes doivent être publiques. Elles correspondent à des méthodes abstraites.

Les interfaces permettent de créer un modèle que les classes qui l'implémentent doivent respecter. On peut comparer une interface à une fiche électrique, elle définit un standard de "connexion" sans savoir quels types d'appareils vont venir s'y connecter. Les interfaces sont très pratiques lorsque plusieurs personnes développent un même projet ou lorsqu'on développe un site modulaire.

Les interfaces permettent également de définir des **constantes**, il suffit de les y déclarer à l'aide du mot clé "const".

Si une classe implémente une interface les méthodes de l'interface doivent être implémentées dans le code autrement vous aurez le message suivant :

```
Fatal error: Class Personne contains 2 abstract methods and must therefore be declared abstract or implement the remaining methods (iModeleOrganisation::trier, iModeleOrganisation::filtrer) in c:\Inetpub\wwwroot\php\cours_objet\interface_1.php on line 19
```

Cette technique permet aussi de pallier l'absence d'héritage multiple.
Une interface ne contient pas de variables.

- Syntaxes

Création de l'interface.

```
Interface NomInterface
{
    methodeA();
    methodeB();
}
```

Création de la classe avec implémentation d'une interface.

```
class NomDeClasse implements NomInterface
{
    methodeGet() { ... }
    methodeSet() { ... }
    methodeA() { ... }
    methodeB() { ... }
}
```

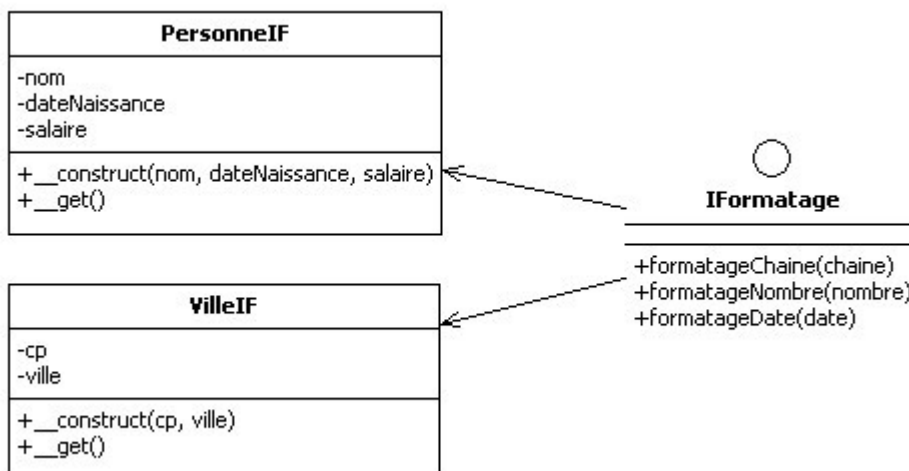
Une classe peut implémenter plusieurs interfaces. Le séparateur est la , (virgule).

```
class NomDeClasse implements NomInterfaceA, NomInterfaceB
```

- Exemple

Créons une interface *IFormatage* qui définit trois méthodes techniques : `formatageChaine(chaine)`, `formatageNombre(nombre)` et `formatageDate(date)`. Les classes *PersonneIF* et *VilleIF* implémenterons cette interface.

```
<?php
interface IFormatage
{
    public function formatageChaine($chaine);
    public function formatageNombre($nombre);
    public function formatageDate($date);
}
?>
```



On peut imaginer des interfaces pour gérer des listes, des DAO, ...

Cf [poo_supplements.doc](#).



```
<?php
require_once("IFormatage.php");
class PersonneIF implements IFormatage
{
    private $nom;
    private $dateNaissance;
    private $salaire;

    public function __construct($nom, $dateNaissance, $salaire)
    {
        $this->nom          = $this->formatageChaine($nom);
        $this->dateNaissance = $this->formatageDate($dateNaissance);
        $this->salaire       = $this->formatageNombre($salaire);
    }
    public function __get($var) { return $this->$var; }
    public function formatageChaine($chaine)
    {
        return strToUpper($chaine);
    }
    public function formatageNombre($nombre)
    {
        return number_format($nombre, 2, ",", " ");
    }
    public function formatageDate($date)
    {
        return ereg_replace("(.*)/(.*)/(.*)", "\\3-\\2-\\1", $date);
    }
}
?>
```

```
<?php
$p = new PersonneIF("tintin","03/10/1920",3000.34);
echo "<br />" . $p->nom;
echo "<br />" . $p->dateNaissance;
echo "<br />" . $p->salaire;
?>
```



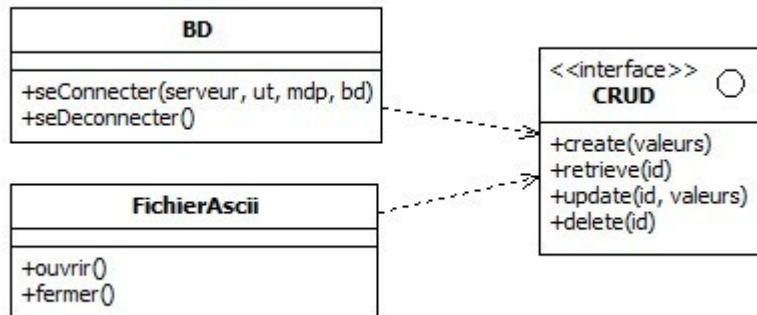

```
<?php
require_once("IFormatage.php");
class VilleIF implements IFormatage
{
    private $cp;
    private $ville;

    public function __construct($cp, $ville)
    {
        $this->cp = $this->formatageNombre($cp);
        $this->ville = $this->formatageChaine($ville);
    }
    //public function __set($var, $valeur) { $this->$var = $valeur; }
    public function __get($var) { return $this->$var; }
    public function formatageChaine($chaine)
    {
        return UCWords($chaine);
    }
    public function formatageNombre($nombre)
    {
        return number_format($nombre, 0, "", " ");
    }
    public function formatageDate($date)
    {
    }
}
?>

<?php
$v = new VilleIF("94130", "nogent sur marne");
echo "<br />" . $v->cp;
echo "<br />" . $v->ville;
?>
```

- [Exercice](#)

Interface CRUD



La table de test :

Le fichier de test :

- [Corrigé](#)

Cf InterfaceCRUDExo.php

1.10 PARCOURIR LES MEMBRES D'UNE CLASSE

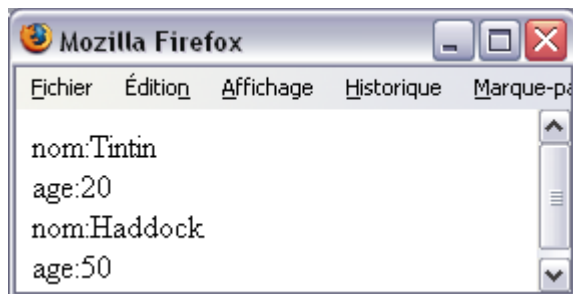
Voici 3 techniques :

- ✓ Implémenter une méthode `getMembres()`,
- ✓ Utiliser la méthode **`get_object_vars($objet)`**.
- ✓ Implémenter l'interface `Iterator` de PHP,

1.10.1 Via une méthode interne

Cette méthode liste toutes les propriétés (private, protected, public) de la classe ainsi que les valeurs des propriétés.

- Exemple



Ajoutez la méthodes `getMembres()` à la classe `Personne`.

```
// -----
public function getMembres()
// -----
{
    $lsMembres = "";
    foreach($this as $attribut => $valeur)
    {
        $lsMembres .= "<br />$attribut:$valeur";
    }
    return $lsMembres;
}
```

```
<?php
// --- getMembresTest.php
require_once("Personne.php");

$tintin = new Personne("Tintin",20);
$haddock = new Personne("Haddock",50);

echo $tintin->getMembres();
echo $haddock->getMembres();
?>
```

Nb : ceci peut être utile pour la méthode `__toString()`. Cf plus loin dans le paragraphe sur les méthodes magiques.

1.10.2 Via la fonction `get_object_vars($objet)`

Cette fonction renvoie un tableau associatif. Les clés sont les attributs, les valeurs sont les valeurs des attributs.

Une ligne et une fonction de plus !!!

```
// -----  
public function getMembres()  
// -----  
{  
    $t = get_object_vars($this);  
    $lsMembres = "";  
    foreach($t as $attribut => $valeur)  
    {  
        $lsMembres .= "<br/>$attribut:$valeur";  
    }  
    return $lsMembres;  
}
```

1.10.3 Via l'interface Iterator de PHP

- Objectif

Parcourir les attributs d'un objet en implémentant sur une classe perso l'interface **Iterator** de PHP.

Cette interface possède **5 méthodes** : **rewind()**, **next()**, **valid()**, **key()**, **current()**.
Au développeur de coder ce qui lui semble nécessaire et en particulier par rapport à la méthode **getMembres()**.

Dans l'exemple suivant les propriétés reçues en paramètres sont stockées dans un attribut nommé \$objet.

Lors de l'instanciation le nombre d'attributs est compté.

La méthode **rewind()** positionne l'index à 0.

La méthode **next()** incrémente la valeur de l'index.

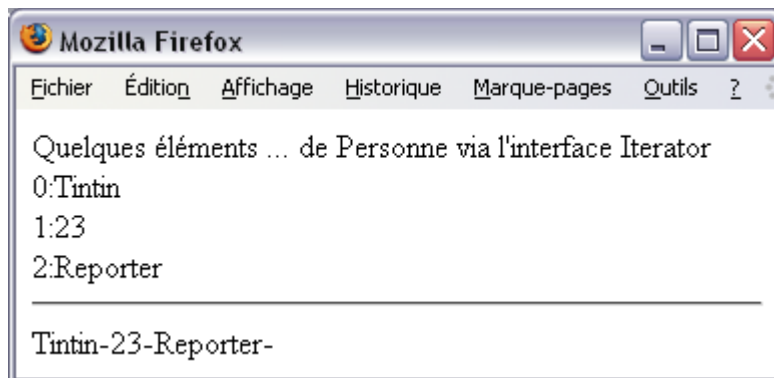
La méthode **key()** renvoie la valeur de l'index.

La méthode **current()** renvoie la valeur de la propriété correspondant à l'index.

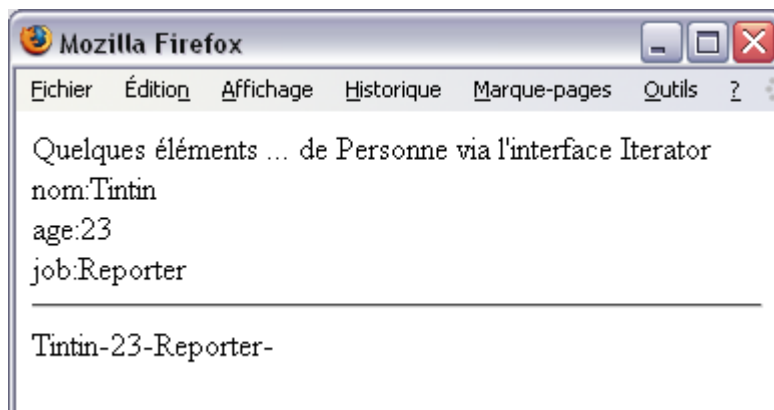
La méthode **valid()** teste la valeur de l'index courant vis-à-vis du nombre d'attributs.

A la différence de la méthode précédente et de la méthode **get_object_vars(\$this)** que nous verrons plus loin l'implémentation de **Iterator** permet de sélectionner les propriétés que l'on veut scruter **MAIS** pour récupérer les clés il faut ajouter un tableau (cf script 2).

- Ecran



Version 2



- Script

```
<?php
header("Content-Type: text/html; charset=UTF-8");
// --- ObjetEtIterator.php
class Personne implements Iterator
{
    private $objet, $nb, $indexCourant;

    function __construct($nom, $age, $job)
    {
        $this->objet = array($nom, $age, $job);
        $this->nb = count($this->objet);
    }
    // --- Méthodes à implémenter de ITERATOR
    public function rewind() { $this->indexCourant = 0; }
    public function next() { $this->indexCourant++; }
    public function key() { return $this->indexCourant; }
    public function current() { return $this->objet[$this->indexCourant]; }
    public function valid() { return $this->indexCourant < $this->nb; }
    public function __toString()
    {
        $chaine = "";
        for($this->rewind(); $this->valid(); $this->next())
        {
            $chaine .= $this->current() . "-";
        }
        return $chaine;
    }
}

// -----
// --- Implémentation
// -----
$p = new Personne("Tintin", 23, "Reporter");

print ("Quelques éléments ... de Personne via l'interface Iterator");
for($p->rewind(); $p->valid(); $p->next()) echo "<br />", $p->key(), ":", $p->current();

echo "<hr />", $p;
?>
```

Version 2

```

<?php
header("Content-Type: text/html; charset=UTF-8");
// --- ObjetEtIterator_2.php
class Personne implements Iterator
{
    private $structure, $objet, $nb, $IndexCourant;

    function __construct($nom, $age, $job)
    {
        $this->structure = array("nom", "age", "job");
        $this->objet      = array($nom, $age, $job);
        $this->nb         = count($this->objet);
    }
    // --- Méthodes à implémenter de ITERATOR
    public function rewind() { $this->indexCourant = 0; }
    public function next()   { $this->indexCourant++; }
    public function key()    { return $this->structure[$this->indexCourant]; }
    public function current() { return $this->objet[$this->indexCourant]; }
    public function valid()  { return $this->indexCourant < $this->nb; }
    public function __toString()
    {
        $chaine = "";
        for($this->rewind(); $this->valid(); $this->next())
        {
            $chaine .= $this->current() . "-";
        }
        return $chaine;
    }
}

// -----
// --- Implémentation
// -----
$p = new Personne("Tintin", 23, "Reporter");

print("Quelques éléments ... de Personne via l'interface Iterator");
for($p->rewind(); $p->valid(); $p->next()) echo "<br />", $p->key(), ":", $p->current();
echo "<hr />", $p;
?>

```

1.11 DIVERS

1.11.1 Les méthodes magiques

Voici quelques-unes des méthodes magiques de PHP :

Méthode/Fonction	Description
<code>__construct([arguments])</code>	Constructeur.
<code>__destruct()</code>	Destructeur.
<code>get(membre)</code>	Pour récupérer une propriété.
<code>set(membre, valeur)</code>	Pour affecter la valeur d'une propriété.
<code>sleep()</code>	Sollicitée lors de la sérialisation.
<code>wakeup()</code>	Sollicitée lors de la désérialisation.
<code>__toString()</code>	Permet grâce à <code>echo</code> , <code>print</code> d'afficher le retour de cette méthode (une chaîne de caractères) et de ne mentionner comme argument de <code>echo</code> ou de <code>print</code> que la variable objet.
<code>__clone()</code>	Permet de personnaliser le clonage. Cf l'instruction <code>clone</code> .
<code>__call(methode, arguments)</code>	Gestion du polymorphisme surcharge de méthode.
<code>__autoload(classe)</code>	Ceci n'est pas une méthode mais une fonction de chargement automatique d'inclusions de classes à utiliser dans un script utilisateur.

Nous avons déjà vu les sept premières.

1.11.2 La méthode `__toString()`

Cette méthode doit renvoyer une chaîne de caractères et est automatiquement sollicitée par `print($objet);` ou par `echo $objet;`.

A ajouter dans la classe `Personne`.

```
public function __toString()
{
    return "$this->prenom $this->nom a $this->age ans";
}
```

```
$tintin = new Personne("Albert", "Tintin", 20);
echo $tintin, "<br/>";
```


1.11.3 Cloner un objet

Cloner un objet c'est faire une copie d'un objet. Depuis la version 5 cela est possible avec le mot clé **clone**.

Si la méthode magique `__clone()` est présente elle est automatiquement sollicitée. Elle permet de personnaliser le clonage. Le clone n'est plus un clone!

Exemple :

Si vous ajoutez une méthode **`__clone()`** à la classe `Personne`, vous modifiez le comportement du clonage.

```
| public function __clone() { $this->nom = ucWords($this->nom); }
```

ainsi le test d'égalité renverra `False` (`ucWords` met en nom propre).

```
<?php
// --- clonage.php
require_once("Personne.php");

// --- Instanciation
$original = new Personne("tintin", 30);
// --- Clonage
$clone     = clone($original);

echo "<br />Original : " . $original;
echo "<br />Clone    : " . $clone;

if($original == $clone) echo "<br />Les deux sont egaux";
else echo "<br />Les deux sont differents";
?>
```



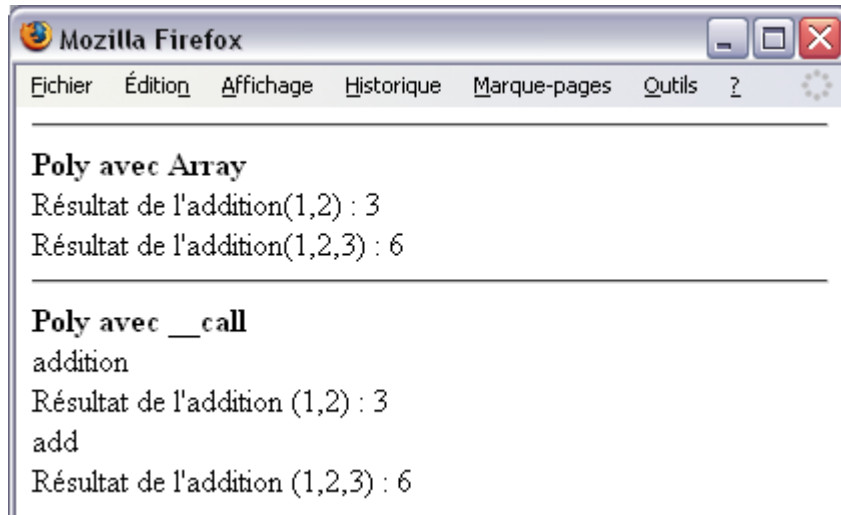
La méthode `__clone()` permet aussi dans le cadre du Singleton d'interdire le clonage.

```
| public function __clone()
| {
|     trigger_error('Le clônage n\'est pas autorisé.', E_USER_ERROR);
| }
```

1.11.4 La méthode `__call(methode, arguments)`

Elle permet de "générer" du polymorphisme ... à la volée.

C'est-à-dire que vous pouvez passer n'importe quel nom de méthode et n'importe quel type d'argument.



<?php

```
// --- Calculs.php
header("Content-Type: text/html; charset=UTF-8");
// -----
class Calculs
// -----
{
    public function addition($arguments)
    {
        $r = 0;
        foreach($arguments as $element) $r += $element;
        return $r;
    }
}

// -----
class CalculsCall
// -----
{
    public function __call($methode, $arguments)
    {
        echo "<br />", $methode;
        $r = 0;
        foreach($arguments as $element) $r += $element;
        return $r;
    }
}

// --- TEST
echo "<hr /><strong>Poly avec Array</strong>";

$operation = new Calculs();

$r = $operation->addition(array(1, 2));
echo "<br />Résultat de l'addition(1,2) : ", $r;
$r = $operation->addition(array(1, 2, 3));
echo "<br />Résultat de l'addition(1,2,3) : ", $r;
unset($operation);

echo "<hr /><strong>Poly avec __call</strong>";

$operation = new CalculsCall();

$r = $operation->addition(1, 2);
echo "<br />Résultat de l'addition (1,2) : ", $r;
$r = $operation->add(1, 2, 3);
echo "<br />Résultat de l'addition (1,2,3) : ", $r;
```

?>

Autre exemple avec `__call()`

Dans cet exemple le nom de la méthode sollicitée est testé ainsi que le type d'argument. La redirection est faite selon ces critères. Le typage faible de PHP permet ce type de "surcharge polymorphique".



```

<?php
// --- OperationCall.php
header("Content-Type: text/html; charset=UTF-8");
// -----
class OperationCall
// -----
{
    // -----
    public function __call($methode, $arguments)
    {
        if($methode == 'affichage' && !is_array($arguments[0]) && !
is_object($arguments[0]))
        {
            return $arguments[0];
        }
        if($methode == 'affichage' && is_array($arguments[0]))
        {
            $r = "";
            foreach($arguments[0] as $valeur) $r .= $valeur . "-";
            return $r;
        }
        if($methode == 'affichage' && is_object($arguments[0]))
        {
            return $arguments[0]->__toString();
        }
    }
}

// -----
class Personne
// -----
{
    private $nom, $prenom;
    public function __construct($nom, $prenom)
    {
        $this->nom = $nom; $this->prenom = $prenom;
    }
    public function __toString()
    {
        $chaine = "";
        foreach($this as $valeur) $chaine .= $valeur . "-";
        return $chaine;
    }
}

// -----
// --- Instanciations et tests
// -----
echo "<br /><strong>Poly avec __call</strong>";
$operation = new OperationCall();

echo "<br />Variable scalaire : ", $operation->affichage("10");

echo "<br />Tableau : ", $operation->affichage(array(1,2));

$p = new Personne("Bianca","Castafiore");
echo "<br />Un objet : ", $operation->affichage($p);
?>

```

1.11.5 L'héritage et les méthodes __set() et __get().

Pour que les méthodes magiques __set() et __get() soient actives dans le cas d'héritage il faut les "dupliquer" et que les propriétés des parents soient protected.

```
<?php
// --- L'héritage
class Personne
{
    protected $nom;
    protected $age;
    public function __construct($nom, $age) { $this->nom = $nom; $this-
>age = $age; }
    public function __set($var, $valeur) { $this->$var = $valeur; }
    public function __get($var) { return $this->$var; }
}

class Salarie extends Personne
{
    private $job;
    public function __construct($nom, $age, $as_job) { $this->job =
$as_job; parent::__construct($nom, $age); }
    public function __set($var, $valeur) { $this->$var = $valeur; }
    public function __get($var) { return $this->$var; }
}

// -----
// --- Instantiation d'un objet et utilisation
// -----

$tintin = new Personne("Tintin",20);
print("Monsieur <b>$tintin->nom</b> a <b>$tintin->age</b> ans<br />");

$haddock = new Salarie("Haddock",50,"Capitaine");
print("Monsieur <b>$haddock->nom</b> a <b>$haddock->age</b> ans et bosse
comme <b>$haddock->job</b><br />");
?>
```

NB : pour assouplir la méthode __toString() utilisez la méthode perso getMembres().

1.11.6 La fonction `__autoload()`

- Objectif

La fonction `__autoload($classe)` permet de charger automatiquement le fichier classe correspondant lors de l'instanciation d'un objet.

Cela évite les interminables `require_once()` lorsque chaque classe est stockée dans un fichier distinct.

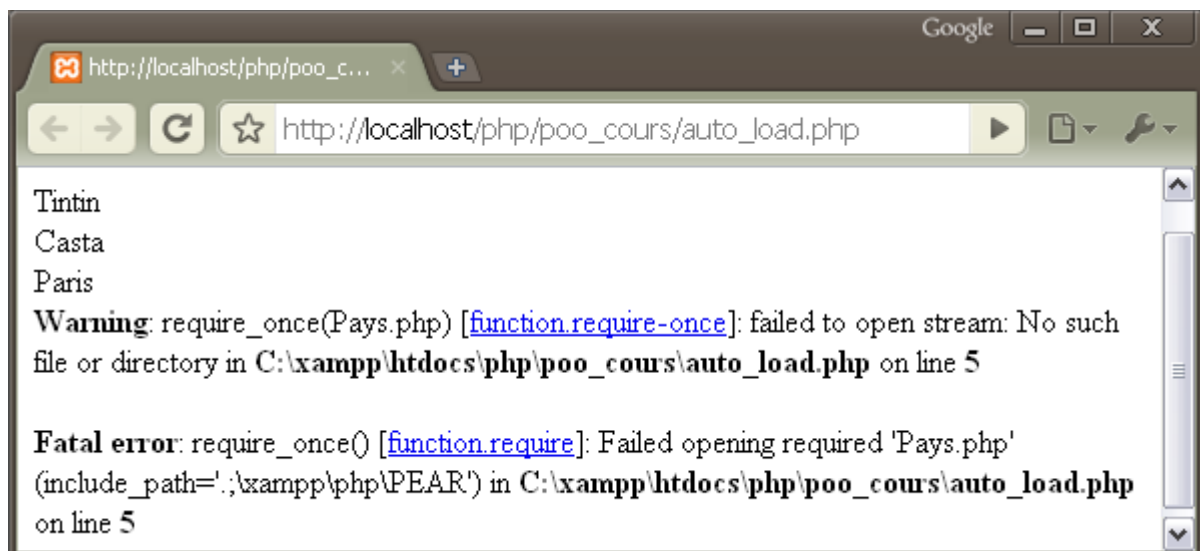
Le nom du fichier doit être identique au nom de la classe.

Les fichiers classes doivent se trouver dans le même dossier ou dans le même chemin de recherche.

Sans gestion d'exception.

Cet exemple instancie quatre objets. Deux de la classe [Personne], un de la classe [Ville] et un dernier de la classe [Pays] qui n'existe pas.

Un Warning et une erreur fatale sont générés.



- Script

```
<?php
// --- auto_load.php
function __autoload($classe)
{
    require_once("$classe.php");
}

$tintin = new Personne("Albert", "Tintin", 20);
echo "<br/>", $tintin->nom;

$casta = new Personne("Laetitia", "Casta", 32);
echo "<br/>", $casta->nom;

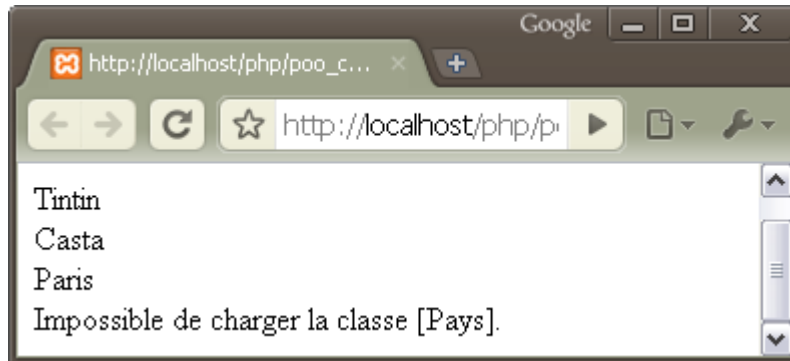
$paris = new Ville("75", "Paris");
echo "<br/>", $paris->ville;

// --- La classe Pays n'existe pas
$france = new Pays("France");
?>
```

Avec une gestion d'exception.

Dans la fonction `__autoload()` on teste l'existence du fichier de classe. S'il n'existe pas on lève une exception.

Les tentatives d'instanciations de classe sont dans un bloc Try,Catch.



```
<?php
// --- auto_load.php
function __autoload($classe)
{
    if(file_exists("$classe.php")) require_once("$classe.php");
    else throw new Exception("<br/>Impossible de charger la classe
[ $classe ].");
}

try
{
    $tintin = new Personne("Albert", "Tintin", 20);
    echo "<br/>$tintin->nom";

    $casta = new Personne("Laetitia", "Casta", 32);
    echo "<br/>$casta->nom";

    $paris = new Ville("75","Paris");
    echo "<br/>$paris->ville";

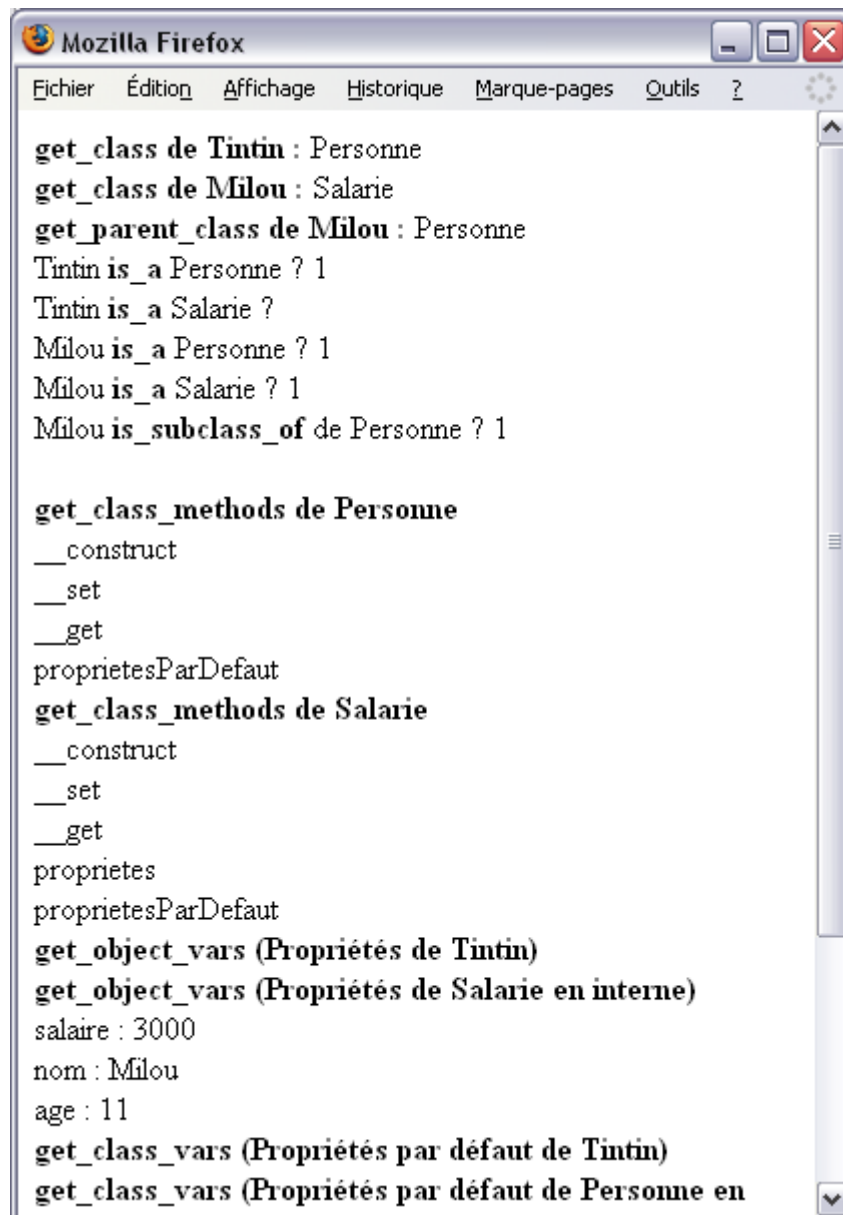
    // --- La classe Pays n'existe pas
    $france = new Pays("France");
}
catch(Exception $e)
{
    echo $e->getMessage();
}

?>
```


1.11.7 Méthodes diverses

Méthodes	Description
get_class(\$objet)	Retourne le nom de la classe d'un objet
get_class_vars('NomDeClasse')	Retourne les noms des attributs et les valeurs par défaut des attributs d'une classe (Pour être appelée de l'extérieur il faut que les propriétés soient publiques, ou à partir d'un dérivé il faut qu'elles soient protected ...)
get_object_vars(\$objet)	Retourne un tableau associatif (noms et valeurs) des propriétés d'un objet (Même remarque)
get_class_methods('NomDeClasse')	Retourne les noms des méthodes d'une classe (Les siennes et celles dont elle hérite)
method_exists('MonDeClasse','méthode')	Vérifie que la méthode existe pour une classe
property_exists('MonDeClasse', 'propriété')	Vérifie que la propriété existe pour la classe
get_parent_class(\$objet) get_parent_class('NomDeClasse')	Retourne le nom de la classe parent d'un objet ou d'une classe
is_subclass_of(\$objet, 'NomDeClasse') is_subclass_of('ClasseEnfant, 'ClasseParent')	Détermine si un objet ou une classe est une sous-classe
is_a(\$objet,'NomDeClasse')	Retourne TRUE si un objet a pour classe ou pour parent une classe donnée
class_exists('NomDeClasse')	Vérifie qu'une classe a été définie
get_declared_classes()	Liste toutes les classes définies (Celles du noyau et les vôtres)
interface_exists('NomDInterface')	Vérifie qu'une interface a été définie
get_declared_interfaces()	Liste toutes les interfaces définies (Celles du noyau et les vôtres)
call_user_method('méthode',\$objet)	Appelle une méthode utilisateur d'un objet (Obsolète depuis PHP4.1.0)
call_user_method_array('méthode',\$objet, \$t)	Appelle une méthode utilisateur avec un tableau de paramètres (Obsolète depuis PHP4.1.0)

- Exemple



The screenshot shows a Mozilla Firefox browser window with the following content:

```
get_class de Tintin : Personne
get_class de Milou : Salarie
get_parent_class de Milou : Personne
Tintin is_a Personne ? 1
Tintin is_a Salarie ?
Milou is_a Personne ? 1
Milou is_a Salarie ? 1
Milou is_subclass_of de Personne ? 1

get_class_methods de Personne
__construct
__set
__get
proprietesParDefaut
get_class_methods de Salarie
__construct
__set
__get
proprietes
proprietesParDefaut
get_object_vars (Propriétés de Tintin)
get_object_vars (Propriétés de Salarie en interne)
salaire : 3000
nom : Milou
age : 11
get_class_vars (Propriétés par défaut de Tintin)
get_class_vars (Propriétés par défaut de Personne en
```

```

<?php
// -----
class Personne
{
    private    $age = 18;
    protected $nom = "Default";

    public function __construct($age, $nom) { $this->age = $age; $this->nom =
$nom; }
    public function __set($var, $valeur) { $this->$var = $valeur; }
    public function __get($var) { return $this->$var; }

    public function proprietesParDefaut()
    {
        print("<br /><strong>get_class_vars (Propriétés par défaut de Personne
en interne)</strong>"); // --- Il faut que le propriétés soient protected
        $tProprietes = get_class_vars('Personne');
        foreach($tProprietes as $propriete) echo "<br />$propriete";
    }
}

// -----
class Salarie extends Personne
{
    private $salaire;

    public function __construct($age, $nom, $salaire) { $this->age = $age; $this-
>nom = $nom; $this->salaire = $salaire;}
    public function __set($var, $valeur) { $this->$var = $valeur; }
    public function __get($var) { return $this->$var; }

    public function proprietes()
    {
        print("<br /><strong>get_object_vars (Propriétés de Salarie en
interne)</strong>");
        // --- Il faut que les propriétés soient protected
        $tProprietes = get_object_vars($this);
        foreach ($tProprietes as $propriete => $valeur) echo "<br />$propriete
: $valeur";
    }
}
// -----
interface iOrdonnancement
{
    public function trier();
}

.../...

```

// --- METHODES SPECIALES

```

    $tintin = new Personne(20,"Tintin");
    $milou  = new Salarie(11,"Milou");

    print("<b>get_class de Tintin : </b>" . get_class($tintin) . "<br />");
    print("<b>get_class de Milou : </b>" . get_class($milou) . "<br />");

    print("<b>get_parent_class de Milou : </b>" . get_parent_class($milou) . "<br
/>");
    // --- Is_a renvoie True si l'objet est d'une classe donnée mais aussi
    descendant d'une classe donnée
    // --- Donc renvoie True pour $milou is_a Personne, is_a Salarie
    // --- et renvoie True pour $tintin is_a Personne et false pour $tintin is_a
    Salarie
    print("Tintin <b>is_a</b> Personne ? " . is_a($tintin,'Personne') . "<br
/>");
    print("Tintin <b>is_a</b> Salarie ? " . is_a($tintin,'Salarie') . "<br />");
    print("Milou <b>is_a</b> Personne ? " . is_a($milou,'Personne') . "<br />");
    print("Milou <b>is_a</b> Salarie ? " . is_a($milou,'Salarie') . "<br />");

    print("Milou <b>is_subclass_of</b> de Personne ? " .
is_subclass_of($milou,'Personne') . "<br />"); // --- Renvoie 1 si True et Null si
false

    print("<br /><b>get_class_methods de Personne</b>");
    $tMethodes = get_class_methods('Personne'); // ou $t_methodes =
get_class_methods(new Personne());
    foreach ($tMethodes as $methode) echo "<br />$methode";

    print("<br /><b>get_class_methods de Salarie</b>");
    $tMethodes = get_class_methods('Salarie');
    foreach ($tMethodes as $methode) echo "<br />$methode";

    print("<br /><b>get_object_vars (Propriétés de Tintin)</b>");
    // --- Il faut que les propriétés soient publiques
    $tProprietes = get_object_vars($tintin);
    foreach ($tProprietes as $propriete = $valeur) echo "<br />$propriete:
$valeur";
    echo $milou->proprietes();

    print("<br /><b>get_class_vars (Propriétés par défaut de Tintin)</b>");
    // --- Il faut que le propriétés soient publiques
    $tProprietes = get_class_vars('Personne');
    foreach ($tProprietes as $propriete) echo "<br />$propriete";
    echo $milou->proprietesParDefaut();

    print("<br /><b>method_exists (La méthode ... existe-t-elle ?)</b> " .
method_exists('Personne','affecter_age'));
    print("<br /><b>property_exists (La propriété ... existe-t-elle ?)</b> ");
    var_dump(property_exists('Personne', 'age'));

    print("<br /><b>class_exists (La classe ... existe-t-elle ?)</b> ");
    if (class_exists('Personne')) { $p = new Personne('Haddock',50); echo "La
classe existe"; }
    else echo "La classe n'existe pas";

    print("<br /><b>interface_exists (L'interface ... existe-t-elle ?)</b> ");
    if (interface_exists('iOrdonnancement')) echo "L'interface existe";
    else echo "L'interface n'existe pas";
?>

```

1.11.8 La comparaison d'objets

- Objectif

Comparer avec l'opérateur d'égalité et d'identité.

- Exemple

Admettons l'instanciation d'objets de la même classe.

```
<?php
// --- 00Comparaisons.php
require_once("Personne.php");

$p1 = new Personne("Tintin",30);
$p2 = new Personne("Tintin",30);

if($p1 == $p2) print("<br/>p1 egal p2");
else print("<br />p1 different de p2"); // Egalité
if($p1 === $p2) print("<br/>p1 identique a p2");
else print("<br />p1 n'est pas identique a p2"); // Identité
?>
```

Si les valeurs sont égales : `$p1 == $p2` renverra True.

Si les valeurs sont différentes : `$p1 == $p2` renverra False.

Dans les deux cas : `$p1 === $p2` renverra False.

`$p1 === $p2` renverra True seulement si on a affecté `$p2` à `$p1` (ou vice-versa) car dans ce cas ce sont les "pointeurs" qui sont identiques ainsi que les valeurs.

1.12 RELATIONS INTER-CLASSES : ASSOCIATION, AGRÉGATION ET COMPOSITION

Les relations inter-classes peuvent être de type **Association binaire de type Père-fils**, **Association binaire**, **Association n-aires**, **Association avec une classe d'association**, **Agrégation** ou **Composition**. L'**héritage** est aussi une relation inter-classes; elle a été vue dans un paragraphe précédent.

Une **association binaire de type Père-Fils** est une association mono-directionnelle entre deux Classes. Une classe dépend de l'autre classe. La classe dépendante connaît la classe "Parent". L'inverse n'est pas vrai.

Une **association binaire** est une association bi-directionnelle entre deux Classes.

Une **association n-aires** est une association bi-directionnelle entre plusieurs Classes.

Une **association avec une classe d'association** correspond à une association porteurs de propriétés qui est implémentée en tant que classe.

Une **agrégation** est un assemblage **faible** de 2 ou plusieurs classes. Les classes forment un tout mais **ont une existence en soi**. Les objets enfants existent sans l'existence du "parent".

Une **composition** est un assemblage **fort** entre 2 ou plusieurs classes.

Les composants n'ont pas d'existence en soi.

Les objets dépendants ne peuvent exister sans l'objet "parent".

1.12.1 Association binaire de type Père-Fils

- Définition

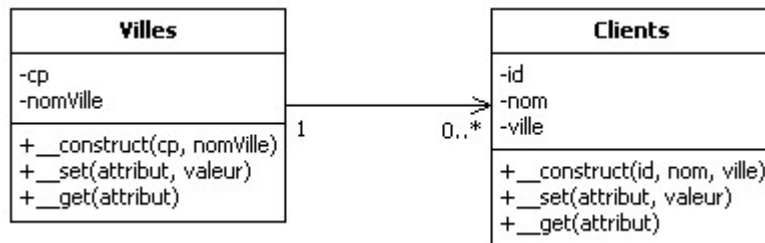
Une association binaire de type "Père-Fils" est une association uni-directionnelle entre deux classes.

Une des classes est indépendante; elle ne connaît pas l'autre classe. L'autre est dépendante; elle connaît la classe dont elle est dépendante. Elle possède une référence vers celle-ci.

- Exemple

L'association Villes-Clients : un client habite une seule ville et une ville concerne éventuellement n clients.

La classe Clients possède un attribut ville.



L'association est matérialisée par la propriété [ville] de la classe [Clients].

```
<?php
// --- AssociationBinairePereFils.php
// --- La classe Villes est indépendante de la classe Clients
// --- C'est la classe Parent (Dans l'optique Association)
class Villes
// -----
{
    private $cp;
    private $nomVille;

    public function __construct($cp, $nomVille)
    {
        $this->cp      = $cp;
        $this->nomVille = $nomVille;
    }
    public function __set($var, $valeur) { $this->$var = $valeur; }
    public function __get($var) { return $this->$var; }
}

// -----
Class Clients
// -----
{
    private $id;
    private $nom;
    private $ville; // --- Ceci représente un objet Villes

    public function __construct($id, $nom, $ville)
    {
        $this->id      = $id;
        $this->nom      = $nom;
        $this->ville    = $ville;
    }
    public function __set($var, $valeur) { $this->$var = $valeur; }
    public function __get($var) { return $this->$var; }
}

// -----
// --- Test
// -----
$paris  = new Villes("75000", "Paris");
$lyon   = new Villes("69000", "Lyon");
$tintin = new Clients("1", "Tintin", $paris);

// --- Interpolations
echo "<br/>$tintin->nom"; // --- Propriete "scalaire"

// --- Acces au nom de la ville de client
echo "<br/>{$tintin->ville->nomVille}";
$tintin->ville = $lyon; // --- Demenagement
echo "<br/>{$tintin->ville->nomVille}"; // --- Structure
?>
```

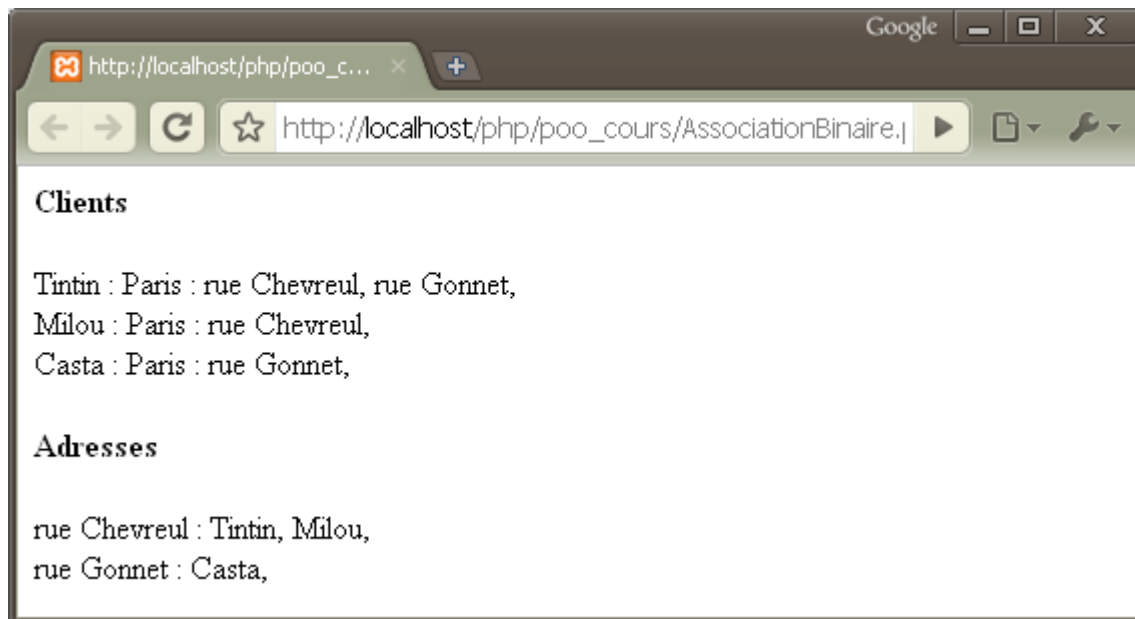
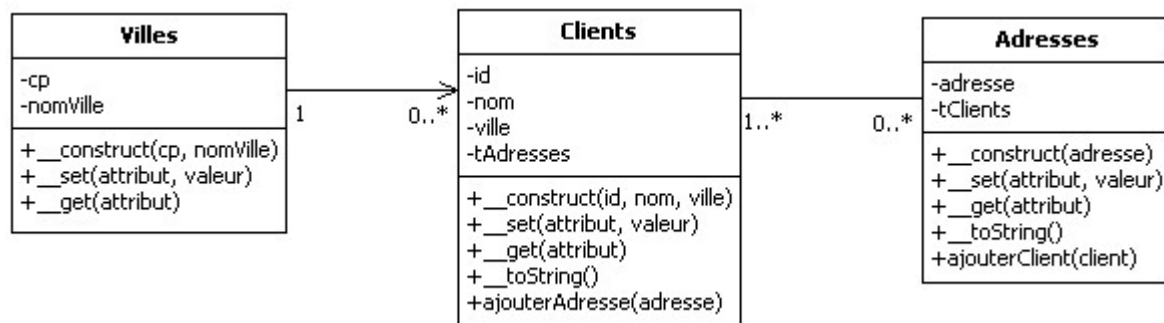

1.12.2 Association binaire

- Définition

Une association binaire est une association bi-directionnelle entre deux classes. Elles sont mutuellement dépendantes. Chaque classe connaît l'autre. Elles possèdent chacune une référence vers l'autre.

- Exemple

La relation Clients-Adresses : Un client peut avoir plusieurs adresses et une adresse peut éventuellement correspondre à plusieurs clients.



```

<?php
// --- AssociationBinaire.php
// -----
class Villes
// -----
{
    private $cp;
    private $nomVille;

    public function __construct($cp, $nomVille)
    {
        $this->cp = $cp;
        $this->nomVille = $nomVille;
    }
    public function __set($var, $valeur) { $this->$var = $valeur; }
    public function __get($var) { return $this->$var; }
}
// -----
Class Clients
// -----
{
    private $id;
    private $nom;
    private $tAdresses = array();
    private $nbAdresses;
    private $ville;

    public function __construct($id, $nom, $adresse, $ville)
    {
        $this->id = $id;
        $this->nom = $nom;
        $this->ville = $ville;
        $this->tAdresses[0] = $adresse;
        $this->nbAdresses = 1;
    }
    public function __set($var, $valeur) { $this->$var = $valeur; }
    public function __get($var) { return $this->$var; }
    public function __toString()
    {
        $adresses = "";
        //for($i=0; $i<count($this->tAdresses); $i++)
        for($i=0; $i<$this->nbAdresses; $i++)
        {
            $adresses .= $this->tAdresses[$i]->adresse . ", ";
        }
        return $this->nom . " : " . $this->ville->nomVille . " : " . $adresses;
    }
    public function ajouterAdresse($adresse)
    {
        $this->tAdresses[$this->nbAdresses] = $adresse;
        $this->nbAdresses++;
    }
}
// -----
class Adresses
// -----
{
    private $adresse;
    private $tClients = array();
    private $nbClients;

    public function __construct($adresse)
    {
        $this->adresse = $adresse;
        $this->nbClients = 0;
    }
    public function __set($var, $valeur) { $this->$var = $valeur; }
}

```

```

    public function __get($var) { return $this->$var; }
    public function __toString()
    {
        $clients = "";
        for($i=0; $i<$this->nbClients; $i++)
        {
            $clients .= $this->tClients[$i]->nom . ",";
        }
        return $this->adresse . " : " . $clients;
    }
    public function ajouterClient($client)
    {
        $this->tClients[$this->nbClients] = $client;
        $this->nbClients++;
    }
}

// -----
// --- Test
// -----
$paris    = new Villes("75000","Paris");

$chevreul = new Adresses("rue Chevreul");
$gonnet   = new Adresses("rue Gonnet");

$tintin   = new Clients("1", "Tintin", $chevreul, $paris);
$chevreul->ajouterClient($tintin);
$tintin->ajouterAdresse($gonnet);

$milou    = new Clients("2", "Milou", $chevreul, $paris);
$chevreul->ajouterClient($milou);

$casta    = new Clients("3", "Casta", $gonnet, $paris);
$gonnet->ajouterClient($casta);

echo "<h4>Clients</h4>";
echo "$tintin";
echo "<br/>$milou";
echo "<br/>$casta";

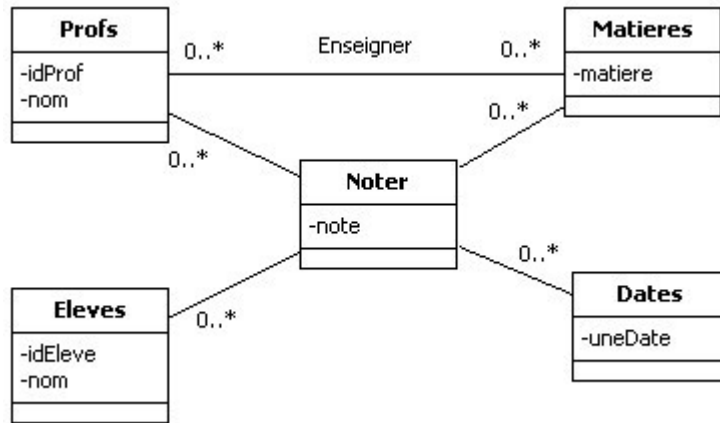
echo "<h4>Adresses</h4>";
echo "$chevreul";
echo "<br/>$gonnet";
?>

```

1.12.3 Association n-aires

Une association binaire existe entre Profs et Matieres.

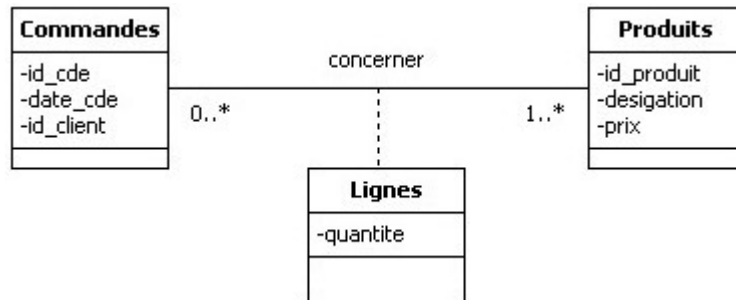
Une association de dimension 4 existe entre Profs, Matieres, Eleves et Dates. La classe Noter est une classe Association.



1.12.4 Association avec une classe d'association

Une association entre deux classes peut posséder des attributs. Ceci est modélisé via une classe-association.

Une commande concerne plusieurs produits et chacun pour une certaine quantité.



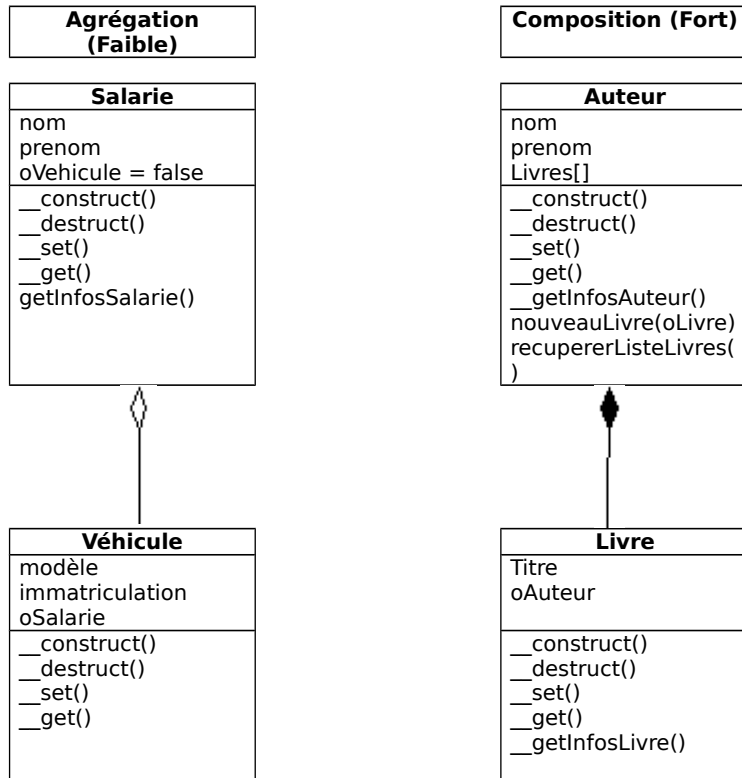
1.12.5 Comparaison Agrégation-Composition

Pour une comparaison prenons deux exemples.

Une agrégation comme relation entre un salarié et un véhicule du parc automobile de sa société.

Une composition entre les livres et leurs auteurs.

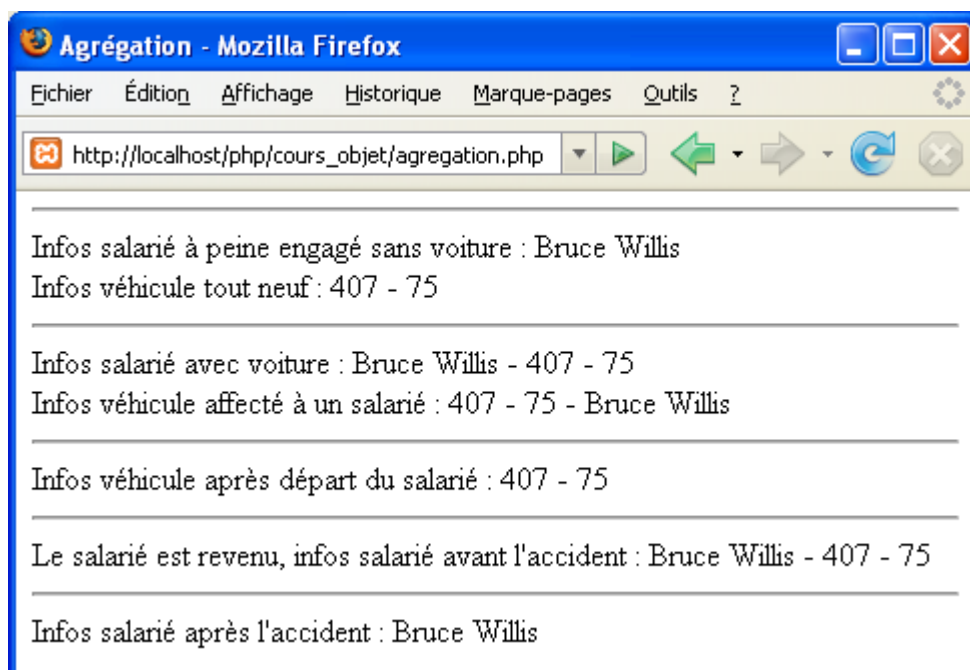
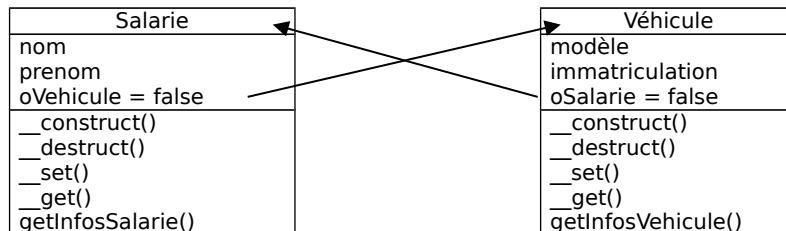
Pour la composition cf aussi le pattern Composite.



1.12.6 Agrégation

- La relation Salarie-Véhicule

Par souci de simplification un véhicule est affecté à un seul salarié et un salarié a au plus un véhicule.



Le script de test crée un salarié, un véhicule, affecte le véhicule au salarié et inversement, détruit le salarié, le recrée, détruit le véhicule.

- Code

```
<?php
// --- Agregation.php
header("Content-Type: text/html; charset=UTF-8");
// -----
class Salarie
// -----
{
    private $nom, $prenom, $oVehicule = false;

    public function __construct($prenom, $nom)
    {
        $this->prenom = $prenom;
        $this->nom      = $nom;
    }
    public function __set($attribut, $valeur) { $this->$attribut = $valeur; }
    public function __get($attribut)         { return $this->$attribut; }
    // --- Les destruction en chaîne c'est loin de rouler
    // --- Il faut le faire manuellement
    // --- ça va être joyeux pour la composition et les relations 1,N !!!
    //public function __destruct() { $this->oVehicule->oSalarie = false; }
    public function __toString()
    {
        if(!$this->oVehicule) $infos = $this->prenom . " " . $this->nom;
        else $infos = $this->prenom . " " . $this->nom . " - " . $this->oVehicule->modele . " - " . $this->oVehicule->immatriculation;
        return $infos;
    }
}

// -----
class Vehicule
// -----
{
    private $modele, $immatriculation, $oSalarie = false;

    public function __construct($modele, $immatriculation, $oSalarie = false)
    {
        $this->modele          = $modele;
        $this->immatriculation = $immatriculation;
        $this->oSalarie        = $oSalarie;
    }
    public function __set($attribut, $valeur) { $this->$attribut = $valeur; }
    public function __get($attribut)         { return $this->$attribut; }
    public function __toString()
    {
        if($this->oSalarie == false)
            $infos = $this->modele . " - " . $this->immatriculation;
        else
            $infos = $this->modele . " - " . $this->immatriculation . " - " . $this->oSalarie->prenom . " " . $this->oSalarie->nom ;
        return $infos;
    }
}
```



```
// -----
// --- Instanciations
// -----
// --- Création d'un salarié
$s1 = new Salarie("Bruce", "Willis");
echo "<hr />Infos salarié à peine engagé sans voiture : ", $s1;

// --- Création d'un véhicule
$v1 = new Vehicule("407", "75");
echo "<br />Infos véhicule tout neuf : ", $v1;

// --- Affectation d'un salarié à un véhicule et vice-versa
$v1->oSalarie = $s1;
$s1->oVehicule = $v1;
echo "<hr />Infos salarié avec voiture : ", $s1;
echo "<br />Infos véhicule affecté à un salarié : ", $v1;

// --- Le salarié s'en va
$s1->oVehicule->oSalarie = false;
unset($s1);
echo "<hr />Infos véhicule après départ du salarié : ", $v1;

// --- Le salarié revient
// --- Reprend la voiture et a un accident avec
$s1 = new Salarie("Bruce", "Willis");
$v1->oSalarie = $s1;
$s1->oVehicule = $v1;
echo "<hr />Le salarié est revenu, infos salarié avant l'accident : ", $s1;
unset($v1);
$s1->oVehicule = false;
echo "<hr />Infos salarié après l'accident : ", $s1;
?>
```

Etant donné que les `__destruct()` ne permettent pas de faire des MAJ en cascade; la solution la plus simple est de créer une méthode nommée `departSalarie()` qui va réinitialiser la valeur dans l'autre objet et détruire l'objet lui-même. Idem pour la classe Vehicule.

- Exemple sur la classe Salarie

```
public function departSalarie()
{
    $this->oVehicule->oSalarie = false;
    unset($this);
}
```

Et plutôt que d'utiliser `unset($v1)` on utilisera `$v1->destruction();`

1.12.7 Composition

- La relation Auteur-Livre

Un auteur a écrit éventuellement plusieurs livres.

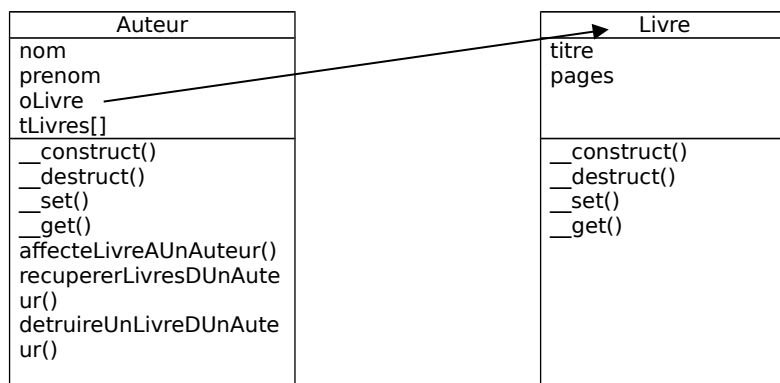
Puisqu'il s'agit d'une relation forte de dépendance, les livres sont référencés directement dans la classe *Auteur*.

Ils sont même instanciés directement à partir de cette classe.

Lorsque l'on détruira un objet *Auteur* automatiquement tous les livres de l'auteur seront détruits.

De la même façon on détruira un livre directement à partir d'un objet auteur.

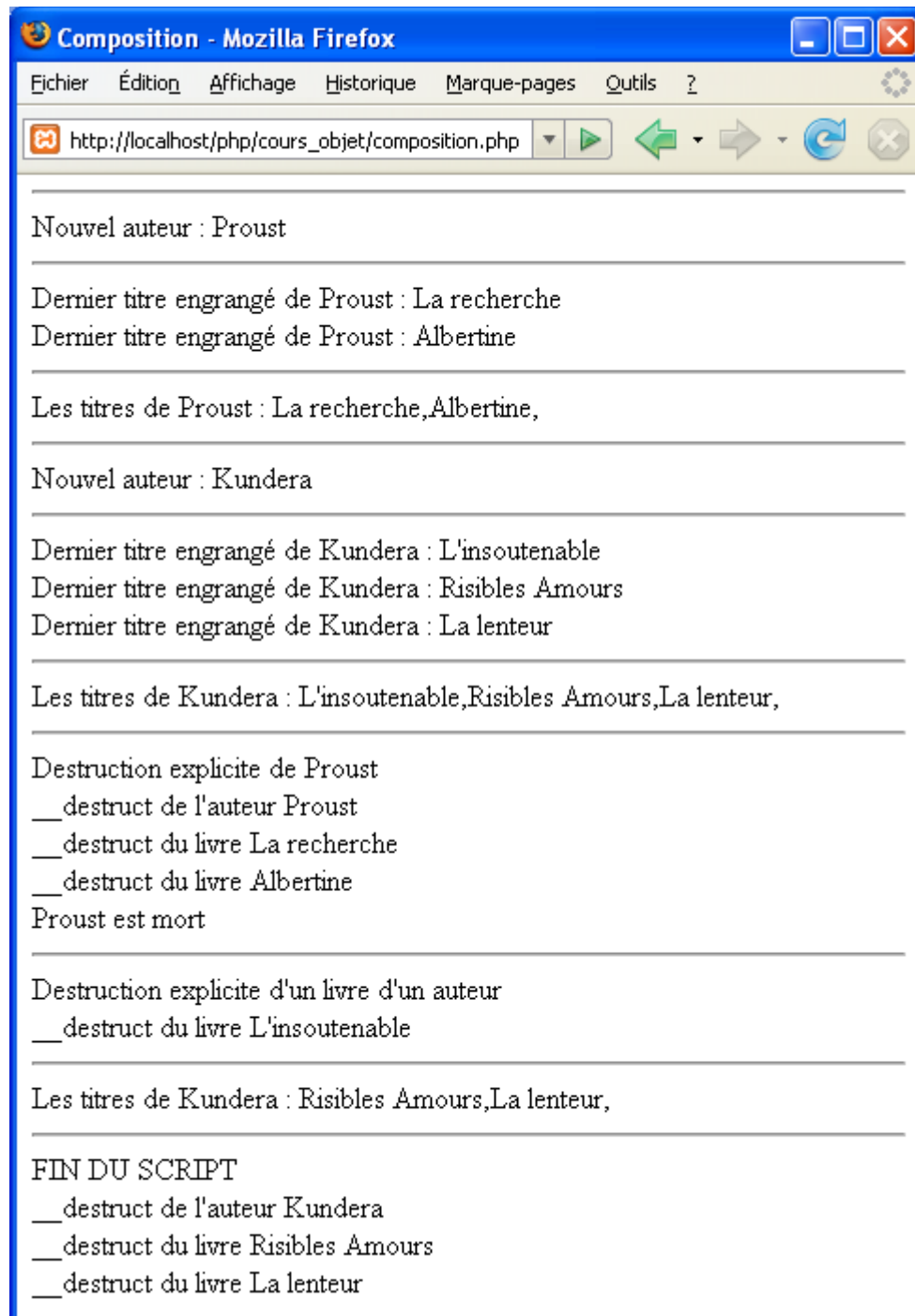
En fait même s'il existe deux classes le comportement se rapproche plus d'une sous-classe.



Le script de test crée des auteurs, des livres, détruit un auteur et détruit un livre.

A titre d'exercice : ajoutez une méthode qui permette de retrouver le nom d'un auteur à partir du livre (cf composition_exo.php).

- Ecran



- Script

```
<?php
// --- Composition.php
header("Content-Type: text/html; charset=UTF-8");
// -----
class Auteur
// -----
{
    private $nom;
    private $oLivre;
    private $tLivres;

    public function __set($attribut, $valeur) { $this->$attribut =
$valeur; }
    public function __get($attribut) { return $this->$attribut; }
    public function __construct($nom) { $this->nom = $nom; }
    public function __destruct() { echo "<br /> __destruct de l'auteur
$this->nom"; }
    public function affecterLivreAUnAuteur($titre, $pages)
    {
        $this->oLivre = new Livre($titre, $pages);
        // --- Surtout pour une destruction future
        // --- mettre comme clé de l'élément tableau
        // --- le titre en majuscule c'est parfait
        // --- avec un code incrémenté à partir d'une static c'est
imparfait
        // --- le premier élément du tableau avait [] comme clé !!! et
non pas [0]
        // --- sauf à le réinitialiser dans le constructeur
        // --- mais bon le code de destruction du livre est quasi plus
simple ainsi
        $this->tLivres[strToUpper($this->oLivre->titre)] = $this-
>oLivre;
    }
    public function recupererLivresDUnAuteur()
    {
        echo "Les titres de $this->nom : ";
        foreach($this->tLivres as $element) echo "$element->titre,";
    }
    public function detruireUnLivreDUnAuteur($titre)
    {
        $titre = strToUpper($titre);
        foreach($this->tLivres as $element)
            if(strToUpper($element->titre) == strToUpper($titre))
                unset($this->tLivres[$titre]);
    }
}
// -----
class Livre
// -----
{
    private $titre;
    private $pages;

    public function __set($attribut, $valeur) { $this->$attribut =
$valeur; }
    public function __get($attribut) { return $this->$attribut; }
    public function __construct($titre, $pages) {$this->titre = $titre;
$this->pages = $pages;}
    public function __destruct() { echo "<br /> __destruct du livre $this-
>titre"; }
}
```

```

// -----
// --- TEST
// -----
$proust = new Auteur("Proust");
echo "<hr />Nouvel auteur : ", $proust->nom;

$proust->affecterLivreAUnAuteur("La recherche",2000);
echo "<hr />Dernier titre engrangé de $proust->nom : ", $proust->oLivre-
>titre;
$proust->affecterLivreAUnAuteur("Albertine",300);
echo "<br />Dernier titre engrangé de $proust->nom : ", $proust->oLivre-
>titre;

echo "<hr />";
$proust->recupererLivresDUnAuteur();

$kundera = new Auteur("Kundera");
echo "<hr />Nouvel auteur : ", $kundera->nom;

$kundera->affecterLivreAUnAuteur("L'insoutenable",350);
echo "<hr />Dernier titre engrangé de $kundera->nom : ", $kundera->oLivre-
>titre;
$kundera->affecterLivreAUnAuteur("Risibles Amours",250);
echo "<br />Dernier titre engrangé de $kundera->nom : ", $kundera->oLivre-
>titre;
$kundera->affecterLivreAUnAuteur("La lenteur",220);
echo "<br />Dernier titre engrangé de $kundera->nom : ", $kundera->oLivre-
>titre;

echo "<hr />";
$kundera->recupererLivresDUnAuteur();

echo "<hr />Destruction explicite de Proust";
unset($proust);
echo "<br />", (isset($proust))?("Proust est vivant")?("Proust est mort");

echo "<hr />Destruction explicite d'un livre d'un auteur";
$kundera->detruireUnLivreDUnAuteur("L'insoutenable");

echo "<hr />";
$kundera->recupererLivresDUnAuteur();

echo "<hr />FIN DU SCRIPT";

```

?>

Pour bien comprendre et visualiser le tableau des livres dans la classe Auteurs vous utiliserez l'instruction suivante dans une méthode de la classe Livres.

```

print_r($this->tLivres);

```

1.13 POO ET GESTION DES EXCEPTIONS

1.13.1 Rappel sur la gestion des erreurs avec PHP

Toute la démonstration sera faite à partir d'un script d'ouverture d'un fichier inexistant. Pour plus de détails cf le support php initiation.

- Pas de gestion d'erreurs

Le script de départ (sans gestion d'erreur). La configuration du php.ini permet l'affichage des erreurs : **display_errors = on.**

```
<?php
    // --- erreurs_1a.php
    // --- Pas de gestion personnalisée des erreurs

    // --- Ouverture fichier inexistant
    $fichier = fopen("fichier.txt","r");
?>
```

Ce message-ci s'affiche :

Warning: fopen(fichier.txt) [[function.fopen](#)]: failed to open stream: No such file or directory in **C:\xampp\htdocs\php\cours_objet\erreurs_1a.php** on line **6**

- Pas de gestion d'erreurs, pas d'affichage d'erreur (encore mieux !)

C'est le @ devant la fonction qui anule l'affichage de l'erreur.

```
<?php
    // --- erreurs_1b.php
    // --- Pas de gestion personnalisée des erreurs et masquage

    $fichier = @fopen("fichier.txt","r");
?>
```

Rien ne s'affiche.

- Gestion d'erreurs au niveau du script et au niveau de chaque instruction

A chaque instruction il faut prévenir l'erreur ou la "rattraper".

A priori : on teste l'existence du fichier sur le disque.

```
<?php
// --- erreurs_1c.php
// --- Gestion locale (Niveau instruction) A priori
$lsFichier = "fichier.txt";
if(!file_exists($lsFichier)) echo "A priori le fichier <b>$lsFichier</b> est
absent ... veuillez ressaisir le nom du fichier";
else $fichier = fopen($lsFichier,"r");
?>
```

A posteriori : si le résultat de la fonction est False on affiche un message d'erreur.

L'erreur a été masquée par le @.

```
<?php
// --- erreurs_1d.php
// --- Gestion locale (Niveau instruction)
$lsFichier = "fichier.txt";
$fichier = @fopen($lsFichier,"r");
if(!$fichier) die("Impossible d'ouvrir '" . $lsFichier . "'");
?>
```

- Gestion d'erreurs au niveau du script

La fonction `set_error_handler()`

```
set_error_handler("gestionnaire_D_Erreur" [, masque])
```

La fonction **`set_error_handler()`** permet de gérer les erreurs au niveau du script. Elle spécifie une fonction utilisateur comme gestionnaire d'erreurs. C'est l'argument obligatoire qu'il faut lui passer. Il est possible de passer un deuxième argument facultatif qui fixe le niveau d'erreur quelque soit la valeur de `error_reporting()`. Si une erreur survient, quelque soit la ligne de code le gestionnaire d'erreurs est appelé. La fonction `set_error_handler()` sollicite une fonction personnalisée d'erreur qui a ce format-ci :

```
Nom de fonction( int $code, string $message [, string $fichier [, int $ligne [, array $contexte]]] )
```

L'argument code et message sont obligatoires.

Sans le masque de `set_error_handler()` la fonction standard de traitement des erreurs de PHP est ignorée. La directive `error_reporting()` du `php.ini` n'aura plus d'effet. Il faut redéfinir le niveau avec la fonction **`error_reporting()`**.

```
Error_reporting(niveau)
```

```
<?php
// --- erreurs_1e.php
// --- Gestion locale (Niveau script)
// --- Gestionnaire personnalisé d'erreur
function monGestionnaireErreur($errNo, $errMsg)
{
    echo "<br /><b>ERREUR</b> [$errNo] $errMsg<br />\n";
}

// --- Paramétrages
error_reporting(E_ERROR | E_WARNING);
set_error_handler("monGestionnaireErreur");

// --- Script générant des erreurs
$canal = fopen("fichier.txt","r");
?>
```


- Gestion centralisée d'erreurs (version simplifiée)

Il faut créer un fichier "bibliothèque" et l'inclure dans chaque script.

Le principe est le même que précédemment mais le code est écrit une fois pour toutes et sera inclus dans chaque script.

Include

```
<?php
// --- erreurs_generales_include.php
// --- Fonction simplifiée de traitement des erreurs
function monGestionnaireErreur($errNo, $errMsg, $errFichier, $errLigne,
$serrContexte)
{
    echo "<br /><b>ALERTE</b> [$errNo] $errMsg\n";
    echo "<br />Dans le fichier <b>$errFichier</b>\n";
    echo "<br />A la ligne <b>$errLigne</b>\n";
}

// --- Fixe le niveau de rapport d'erreur pour ce script : FATALES + ALERTES
error_reporting(E_ERROR | E_WARNING);

// --- Fonction actuelle de traitement des erreurs
set_error_handler("monGestionnaireErreur");
?>
```

Script utilisateur

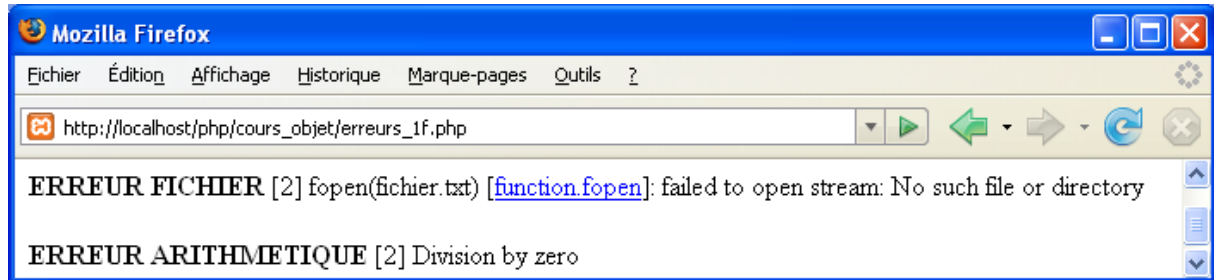
```
<?php
// --- erreurs_1f.php
require_once("erreurs_generales_include.php");

$f = fopen("fichier.text", "r");
?>
```

- Gestion avec une classe

Il est possible de gérer les erreurs avec des classes. La fonction `set_error_handler()` permettra de le spécifier. Dans ce cas il faut passer un tableau comme argument avec comme premier élément le nom de la classe et comme deuxième élément le nom de la méthode.

Les méthodes doivent être statiques.



```
<?php
// --- erreurs_1g.php
// --- Gestionnaire personnalisé d'erreur
// --- avec une classe et deux méthodes statiques
class monGestionnaireErreur
{
    public static function gestionFichiers($errNo, $errMsg)
    {
        echo "<br /><b>ERREUR FICHIER</b> [$errNo] $errMsg<br />\n";
    }
    public static function gestionArithmetiques($errNo, $errMsg)
    {
        echo "<br /><b>ERREUR ARITHMETIQUE</b> [$errNo] $errMsg<br
/>\n";
    }
}

// --- Paramétrages
error_reporting(E_ERROR | E_WARNING);

set_error_handler(array("monGestionnaireErreur", "gestionFichiers"));
$canal = fopen("fichier.txt", "r");

set_error_handler(array("monGestionnaireErreur", "gestionArithmetiques"));
echo 10/0;
?>
```

1.13.2 Introduction à la gestion d'exception

PHP 5 introduit la gestion d'exceptions.

Mais les exceptions PHP ne sont pas levées automatiquement lorsqu'une erreur survient.

Une Exception pour PHP n'est pas une erreur comme dans d'autres langages (Java, PL/SQL, ...).

C'est au développeur de lever éventuellement une exception.

Une exception peut être levée dans d'autres cas. Lorsque survient quelque chose d'exceptionnel ☺.

C'est d'abord en utilisant la classe native **Exception** que nous allons examiner ce type de gestion.

Ensuite nous personnalisons en créant une classe qui héritera de cette classe de base.

La classe Exception de PHP possède :

- 3 attributs Protected : message, line et file,
- 5 méthodes publiques : getMessage(), getLine(), getFile(), getTrace(), getTraceAsString().

Méthode	Description
getMessage()	Renvoie le message d'erreur.
getLine()	Renvoie la ligne de code qui a généré une exception.
getFile()	Renvoie le nom du fichier de script.
getTrace()	Renvoie la trace (un Array).
getTraceAsString()	Renvoie la trace (la pile des fonctions appelées) sous forme de chaîne.

Pour gérer les erreurs via la classe PHP Exception il faut lors de l'apparition d'une erreur lever une exception avec l'instruction **throw**. Ceci doit être fait dans un bloc **try**.

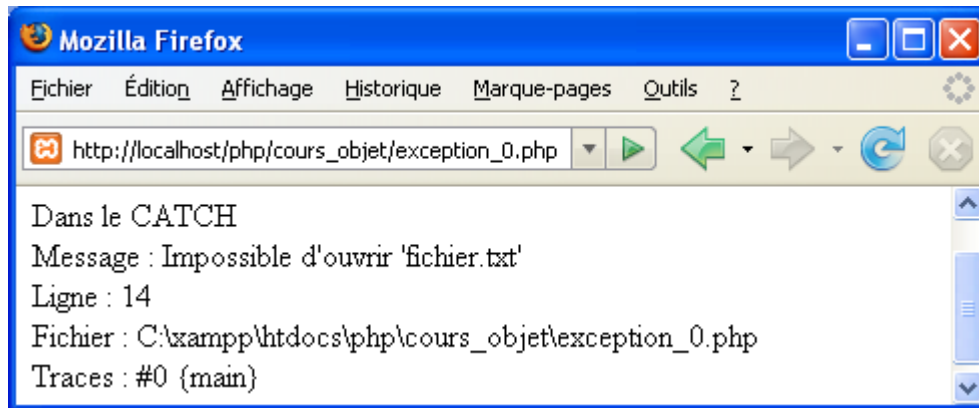
Puis intercepter cette exception dans un bloc **Catch**(Exception \$variable)

Les instructions situées après le catch sont toujours exécutées.

La structure est la suivante :

```
try
{
    Instruction provoquant une erreur;
    if (erreur)
    {
        throw new Exception("Message");
    }
}
catch(Exception $e)
{
    Gestion des exceptions avec les méthodes de la classe Exception;
}
```

- Gestion primitive d'exception



```
<?php
// --- exception_0.php
// --- Le fichier n'existe pas

$lsFichier = "fichier.txt";

try
{
    $fichier = @fopen($lsFichier,"r");
    if(!$fichier) throw new Exception("' $lsFichier' introuvable");
    $lb_close = fclose($fichier);
}
catch(Exception $e)
{
    echo "<br />Dans le CATCH";
    echo "<br />Message : " . $e->getMessage();
    echo "<br />Ligne : " . $e->getLine();
    echo "<br />Fichier : " . $e->getFile();
    echo "<br />Traces : " . $e->getTraceAsString();
}

?>
```

Remarques

Le message est personnalisé. C'est le script (avec l'instruction Throw qui communique le message en levant l'exception). Les autres méthodes renvoient des informations de l'interpréteur.

Dans ce style de gestion à chaque action de votre script pouvant générer une erreur vous devez lever l'exception avec l'instruction Throw.

Donc peu de changement par rapport à une gestion classique.

1.13.3 Créez votre propre classe d'Exception.

- Objectif

En créant une classe héritant de la classe Exception vous pouvez personnaliser la gestion d'erreurs ou bien créer plusieurs gestionnaires selon les fonctions attendues. Attention toutes les méthodes de la classe Exception sont "final" donc non redéfinissables.

Au lieu de lever Exception vous lèverez *ExceptionFichier* ...

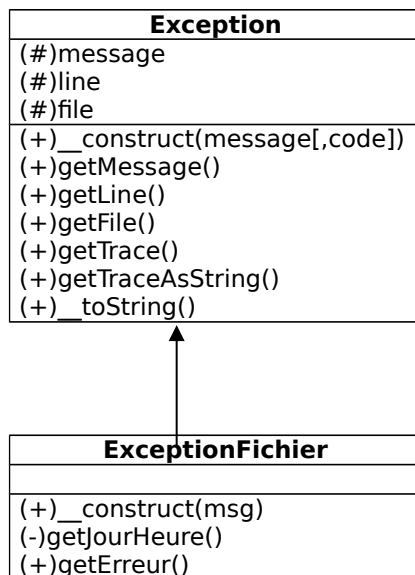
Dans le même ordre d'idées vos classes intégrerons cette classe (ou d'autres spécialisées ExceptionFichier, ExceptionBD, ...) ainsi le développeur d'application utilisateur de vos classes n'a plus de gestion d'erreurs à effectuer.

Dans l'exemple qui suit nous allons créer une classe nommée **ExceptionFichier**

La classe a cette structure-ci :

```
Class ExceptionFichier extends Exception
{
    // --- Méthode(s) personnalisée(s)
    public function méthode()
    {
        // --- du code
        return $qqchose;
    }
}
```

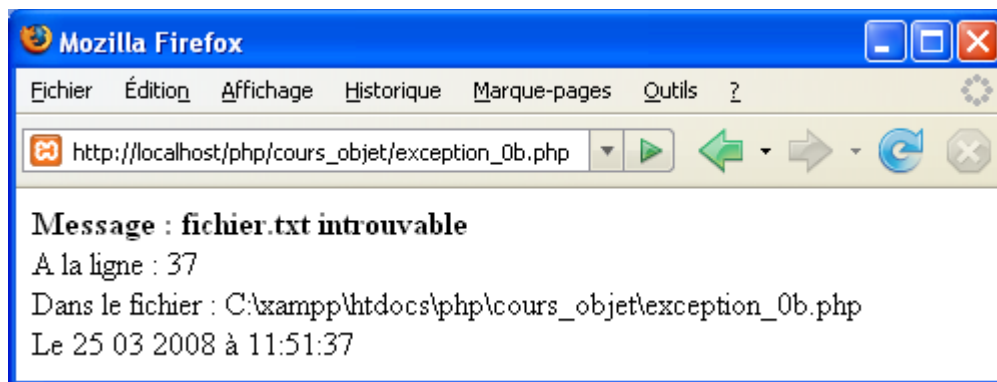
La classe



Le script utilisateur a cette structure-là.

```
try
{
    Instruction;
    throw ExceptionFichier("Message");
}
catch(ExceptionFichier $e)
{
    echo $e->methode();
}
```

- Ecran



- Script

```
<?php
// --- ExceptionFichier.php
// -----
// --- La classe d'Exception personnalisée héritant de Exception
// -----
class ExceptionFichier extends Exception
{
    // --- Méthode privée retournant la date et l'heure
    private function getJourHeure()
    {
        return "Le " . date('d m Y') . " à " . date('H:i:s');
    }

    // --- Méthode retournant un message d'erreur personnalisé
    public function getErreur()
    {
        $msg = '<b>Message : ' . $this->getMessage() . '</b><br/>';
        $msg .= 'A la ligne : ' . $this->getLine() . '<br/>';
        $msg .= 'Dans le fichier : ' . $this->getFile() . '<br/>';
        $msg .= $this->getJourHeure();

        return $msg;
    }
} // --- Fin de la classe

// -----
// --- Le test avec le fichier qui n'existe pas
// -----

$lsFichier = "fichier.txt";
try
{
    $fichier = @fopen($lsFichier,"r");
    if(!$fichier) throw new ExceptionFichier("$lsFichier introuvable");
    $lb_close = fclose($fichier);
}
catch(ExceptionFichier $e)
{
    print $e->getErreur();
}

?>
```

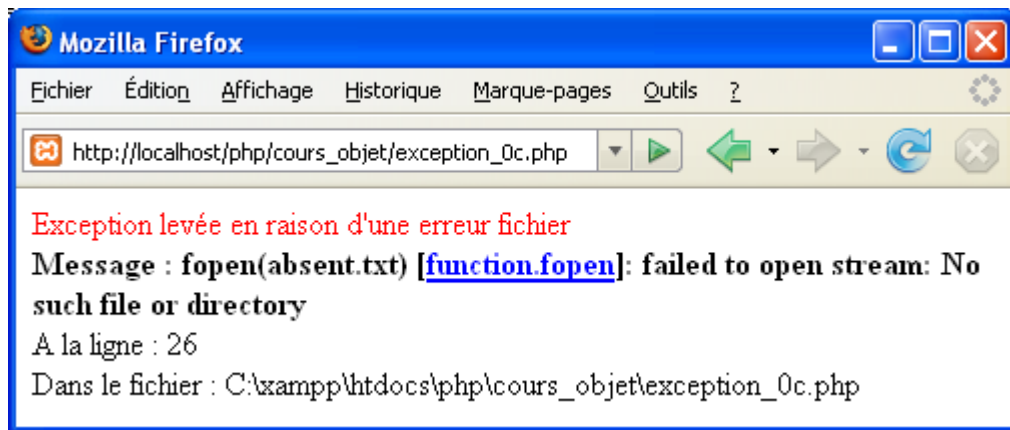
1.13.4 Levée d'exception automatisée

Comme dans le cas de la gestion d'erreurs nous allons préparer la gestion généralisée et centralisée des exceptions.

1.13.4.1 Première étape : utilisation de `set_error_handler()`

- Présentation

Dans le script nous n'aurons plus à lever l'exception en cas d'erreur. La levée d'exception sera automatiquement faite via la fonction `set_error_handler()`. Nous l'avons vu plus haut, il est possible de spécifier à cette fonction un argument de type array spécifiant le nom d'une classe et le nom d'une méthode statique de la classe. C'est cette méthode même qui va lever l'exception et instancier la classe lorsqu'un erreur se produira lors de l'exécution du code.



- Le script

```
<?php
// --- ExceptionFichier.php
// -----
// --- La classe d'Exception personnalisée héritant de Exception
// -----
class ExceptionFichier extends Exception
{
    public function __construct($msg, $code=0, $fichier='', $ligne=0,
    $contexte='')
    {
        parent::__construct($msg, $code);
        if ($fichier != '') $this->file = $fichier;
        if ($ligne != 0) $this->line = $ligne;
        if ($contexte != '') $this->context = $contexte;
    }
    // --- Méthode retournant un message d'erreur personnalisé
    public function getErreur()
    {
        $msg = "<font color='red'>Exception levée en raison d'une
erreur fichier</font><br />";
        $msg .= "<b>Message : ' . $this->getMessage() . '</b><br/>";
        $msg .= "A la ligne : ' . $this->getLine() . '<br/>";
        $msg .= "Dans le fichier : ' . $this->getFile() . '<br/>";
        return $msg;
    }
    public static function erreurTrappeur($code, $msg)
    {
        throw new ExceptionFichier($msg, $code);
    }
}
// --- Fin de la classe

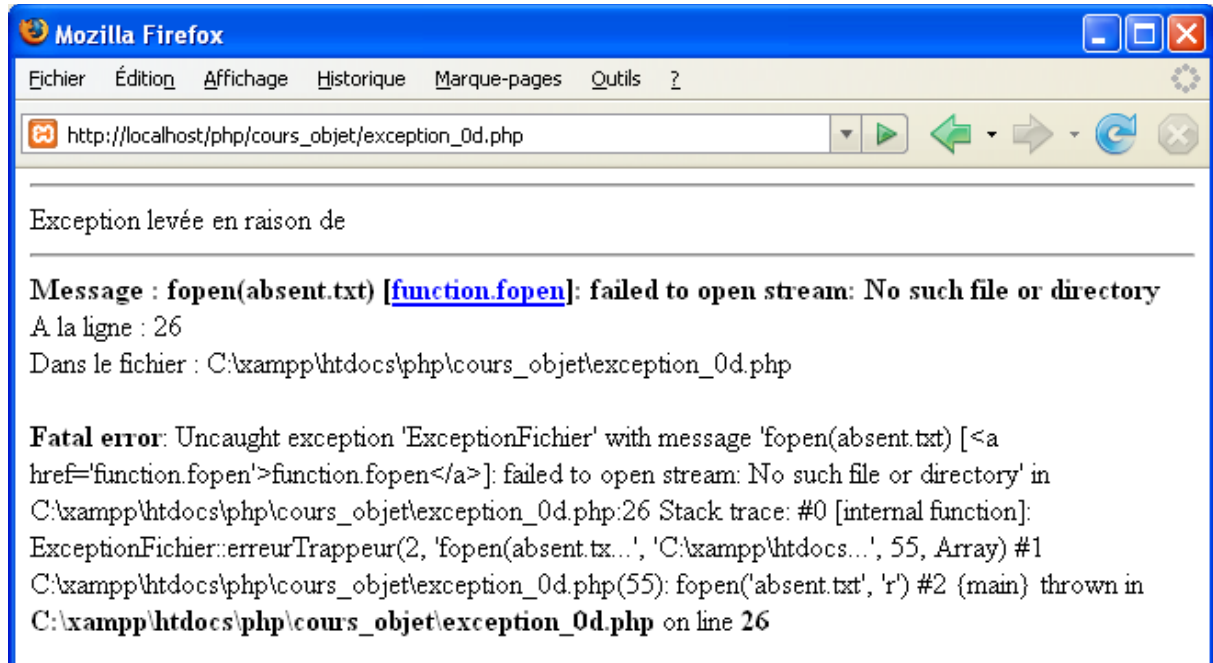
// -----
// --- Le test avec le fichier qui n'existe pas
// -----
set_error_handler(array("ExceptionFichier", "erreurTrappeur"));
try
{
    $fichier = fopen("absent.txt", "r");
}
catch(ExceptionFichier $e)
{
    echo $e->getErreur();
}

?>
```

1.13.4.2 Deuxième étape

- Présentation

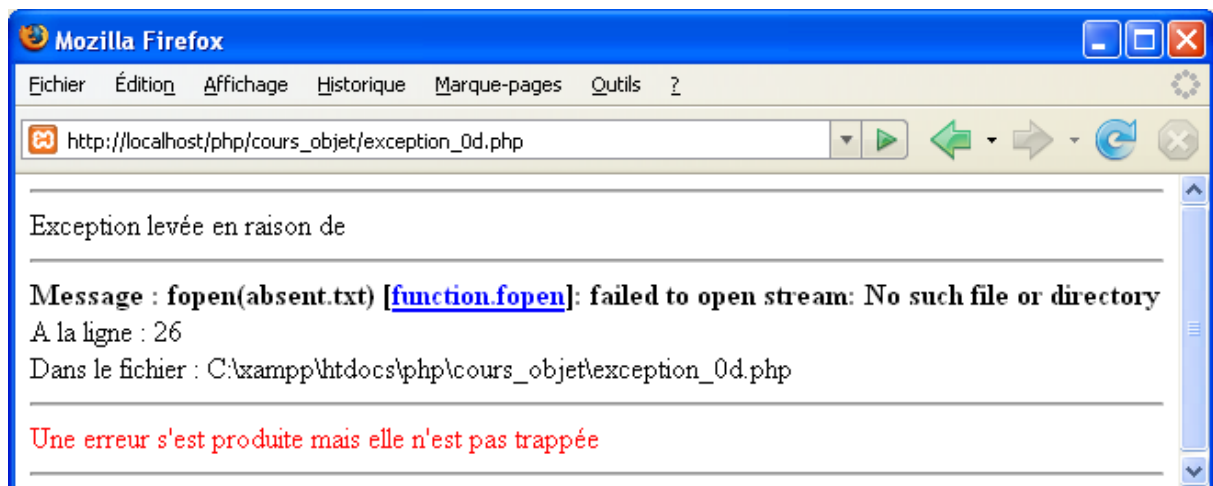
Que se passe-t-il si une erreur se produit et qu'elle n'est pas "trappée" ?
Vous aurez une erreur fatale.



Pour éviter ceci il faut utiliser la fonction **set_exception_handler()**.

```
set_exception_handler("fonction de gestion");
```

L'argument de cette fonction est le nom d'une fonction définie au préalable.
Si une erreur non "trappée" dans le code se produit la fonction est exécutée et le code du script se termine.



- Script

```
<?php
// --- ExceptionFichier.php
// -----
// --- La classe d'Exception personnalisée héritant de Exception
// -----
class ExceptionFichier extends Exception
{
    public function __construct($msg, $code=0, $fichier='', $ligne=0,
    $contexte='')
    {
        parent::__construct($msg, $code);
        if ($fichier != '') $this->file = $fichier;
        if ($ligne != 0) $this->line = $ligne;
        if ($contexte != '') $this->context = $contexte;
    }
    // --- Méthode retournant un message d'erreur personnalisé
    public function getErreur()
    {
        $msg = "<hr />Exception levée en raison de <hr />";
        $msg .= '<b>Message : ' . $this->getMessage() . '</b><br/>';
        $msg .= 'A la ligne : ' . $this->getLine() . '<br/>';
        $msg .= 'Dans le fichier : ' . $this->getFile() . '<br/>';
        return $msg;
    }
    public static function erreurTrappeur($code, $msg)
    {
        throw new ExceptionFichier($msg, $code);
    }
} // --- Fin de la classe

// -----
// --- Fonction définie pour le cas où une exception n'est pas trappée
// --- Pas de Try, Catch
// -----
function ExceptionTrappeur()
{
    echo "<hr /><font color='red'>Une erreur s'est produite mais elle
n'est pas trappée</font><hr />";
}

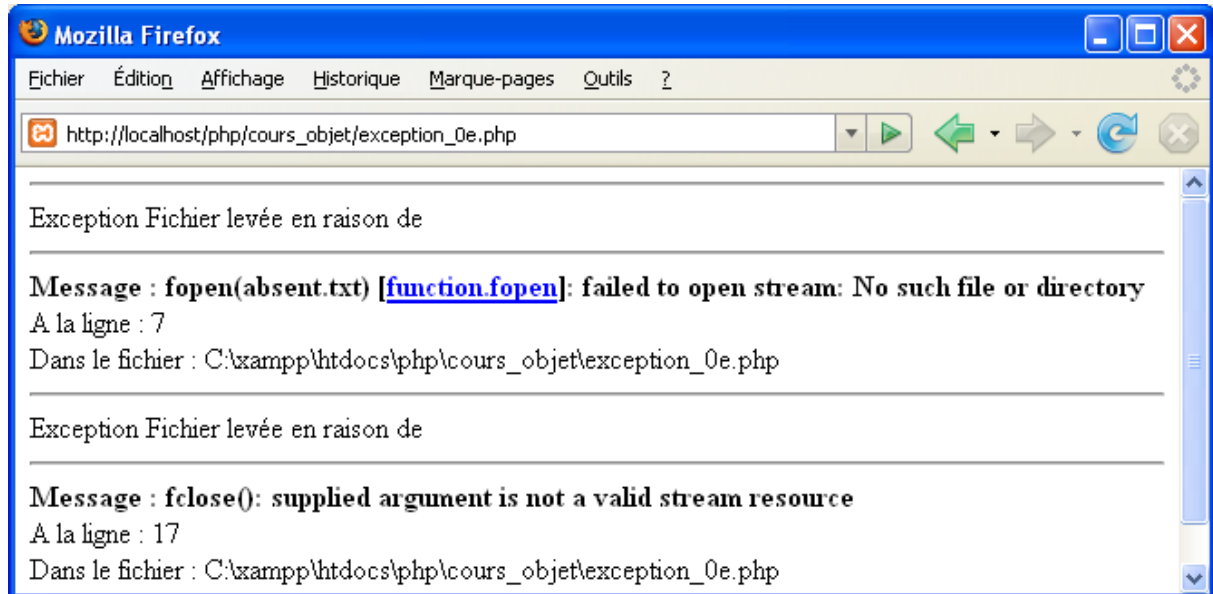
// -----
// --- Le test avec le fichier qui n'existe pas
// -----
set_error_handler(array("ExceptionFichier", "erreurTrappeur"));
set_exception_handler("ExceptionTrappeur");

try
{
    $fichier = fopen("absent.txt", "r");
}
catch(ExceptionFichier $e)
{
    echo $e->getErreur();
}

$fichier = fopen("absent.txt", "r");
// --- Le code qui suit n'est pas exécuté
$fichier = fopen("absent.txt", "r");
?>
```

1.13.5 Levée d'exception centralisée

Nous allons mettre en place le même principe que pour la gestion des erreurs centralisées avec un fichier d'inclusion et un fichier utilisateur.



La classe et les différents `set_error_handler()` et `set_exception_handler()` sont dans le fichier "bibliothèque".

Il n'est pas possible d'avoir plusieurs gestionnaires en même temps et ainsi d'avoir un try et plusieurs catch. Par exemple pour la gestion de fichiers et la gestion BD.

Il faut dans ce cas décentraliser et alternativement appeler un gestionnaire puis l'autre.

```

<?php
// --- ExceptionFichier.php
// -----
// --- La classe d'Exception personnalisée pour les fichiers
// -----
class ExceptionFichier extends Exception
{
    public function __construct($msg, $code=0, $fichier='', $ligne=0,
    $contexte='')
    {
        parent::__construct($msg, $code);
        if ($fichier != '') $this->file = $fichier;
        if ($ligne != 0) $this->line = $ligne;
        if ($contexte != '') $this->context = $contexte;
    }
    // --- Méthode retournant un message d'erreur personnalisé
    public function getErreur()
    {
        $msg = "<hr />Exception Fichier levée en raison de <hr />";
        $msg .= '<b>Message : ' . $this->getMessage() . '</b><br/>';
        $msg .= 'A la ligne : ' . $this->getLine() . '<br/>';
        $msg .= 'Dans le fichier : ' . $this->getFile() . '<br/>';
        return $msg;
    }
    public static function erreurTrappeur($code, $msg, $fichier, $ligne)
    {
        throw new ExceptionFichier($msg, $code, $fichier, $ligne);
    }
} // --- Fin de la classe ExceptionFichier

// -----
function ExceptionTrappeur()
// -----
{
    echo "<hr /><b>Une erreur fatale non trappée</b><hr />";
}

set_error_handler(array("ExceptionFichier", "erreurTrappeur"));
set_exception_handler("ExceptionTrappeur");
?>

```

```

<?php
// --- exception_0e.php
require_once("ExceptionFichier.php");

$fichier = null;
try { $fichier = fopen("absent.txt", "r"); }
catch(ExceptionFichier $e) { echo $e->getErreur(); }

try { fclose($fichier); }
catch(ExceptionFichier $e) { echo $e->getErreur(); }
echo "Jusque là tout va bien";
?>

```

