POO

Programmation orientée objet

Principes de base

PRINCIPES DE BASE

Approche de la POO

Avantages

Les objets

Les classes

- Approche modulaire par opposition à la programmation séquentielle et procédurale.
- Architecture du code est différente, tout est pensé, organisé en objets (utilisateurs, les produits, les commandes...). Modélisation des données.
- Syntaxe spécifique à la POO (classes, objets, méthodes...).
- Approche imposée dans l'utilisation des frameworks.

AVANTAGES DE LA POO

- Un code modulaire: chaque module (type) contient son propre contexte. Il est facile d'isoler un module ou de développer des modules séparément.
- Un code naturel: on applique en informatique la représentation en objets qui est un concept familier et naturel. Il améliore la conceptualisation du projet.
- Un code réutilisable: les types d'objets peuvent être réutilisés ou servir de base pour d'autres types d'objets. Il suffit de rajouter les modifications ou les évolutions nécessaires.
- Un code compréhensible et maintenable: chaque objet regroupe ses propriétés et ses méthodes. On comprend tout de suite la signification générale et le rôle joué par chacun de ses membres.
- Un code modélisable: une application orientée objet peut être représentée sous la forme de différents schémas techniques. Notamment à l'aide de diagrammes de classes UML permettant de modéliser les classes avec leurs attributs et méthodes ainsi que les différentes relations entre elles.

Le concept d’objet en programmation est fait pour être naturel et ressembler à ce que manipule un non-informaticien dans la vie de tous les jours.

Un objet est une chose quelconque, une représentation concrète. La voiture ou la maison de mon voisin est un objet, l’utilisateur de mon site web est un objet, mon compte en banque peut être vu comme un objet.

Un objet peut être également une représentation abstraite ou conceptuelle purement informatique ou non. Une écriture de mon compte bancaire, une session utilisateur ou encore une commande en ligne sont également des objets.

Chaque objet a des caractéristiques qui lui sont propres que l'on appelle des propriétés ou des attributs. Mon compte en banque a une propriété qui définit le numéro de compte, une autre qui définit le solde actuel, une troisième qui donne la date d'ouverture du compte et ainsi de suite. Les propriétés peuvent être vues comme les caractéristiques propres de l'objet.

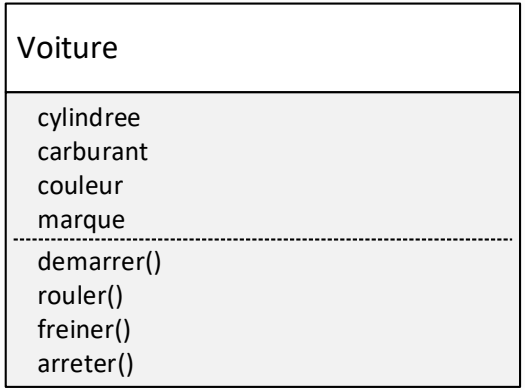
Les objets peuvent aussi avoir des méthodes. Il s'agit des actions que l'on peut appliquer à un objet ou que l'objet peut réaliser. Toujours en prenant mon objet de compte en banque, il existe une méthode pour le solder, une pour réaliser une opération (dépôt ou retrait), une pour transférer de l'argent vers un autre compte, etc.

Vous savez désormais qu'on peut avoir des objets dans une application. Mais d'où sortent-ils ? Dans la vie réelle, un objet ne sort pas de nulle part. En effet, chaque objet est défini selon des caractéristiques et un plan bien précis. En POO, ces informations sont contenues dans ce qu'on appelle des classes.

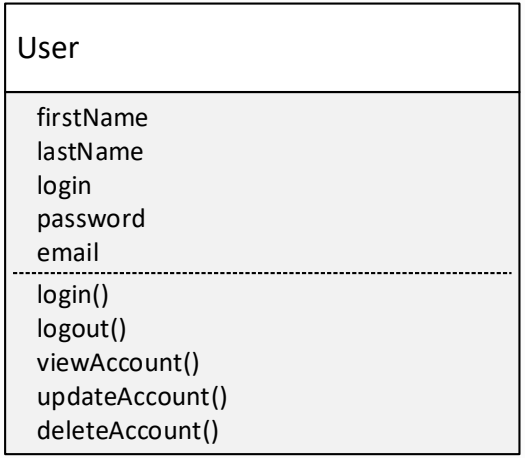
On peut ainsi imaginer une classe représentant les voitures. Toutes les voitures (tous les objets de la classe voiture) ont des plaques d’immatriculation, un moteur avec une certaine puissance et un nombre de portières identifiables (ils ont des propriétés communes). Tous les objets de cette classe ont aussi des méthodes pour démarrer, freiner, accélérer, tourner, etc.

REPRESENTATION D'UNE CLASSE

Vie de tous les jours



Domaine du web



Nom de la classe:

Permet d'identifier la classe

Personnage, *Employé*, *CompteBancaire*, *Utilisateur*...

Ses propriétés ou attributs:

Tout ce qui décrit ou caractérise la classe

Personnage: *nom*, *force*, *vie*, *armes*...

Utilisateur: *id*, *login*, *level*, *mail*

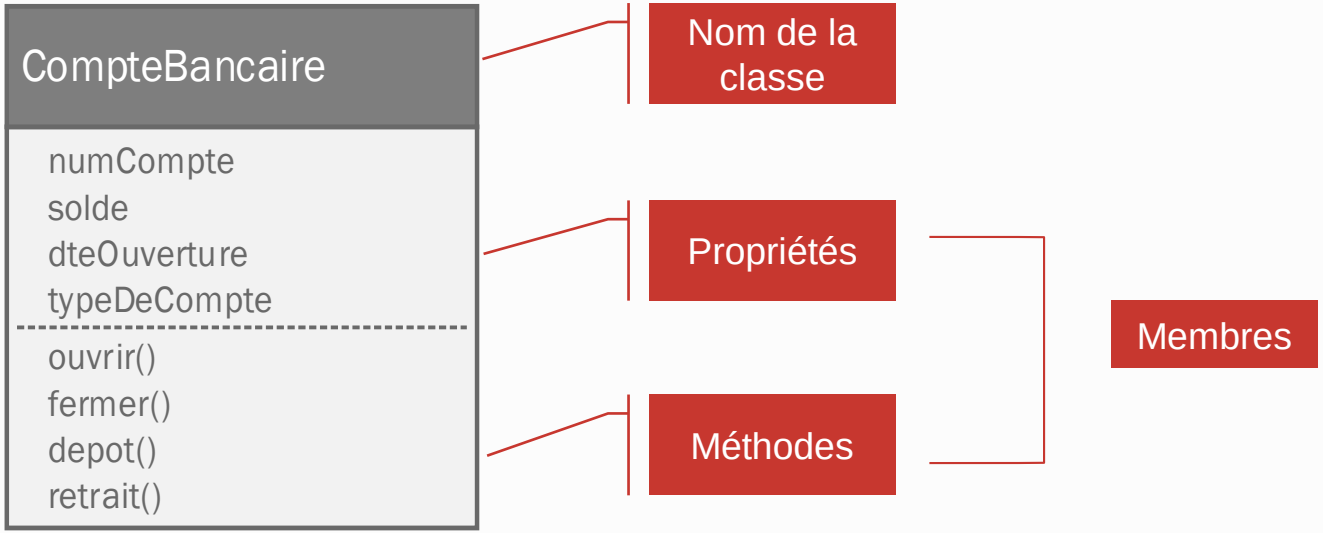
Ses méthodes:

ce que la classe peut faire (actions)

Personnage: *parler*, *attaquer*, *se déplacer*...

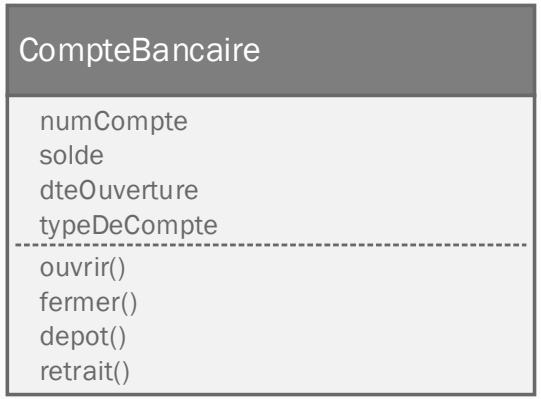
Utilisateur: *s'identifier*, *se déconnecter*, *modifier son compte*...

ORGANISATION D'UNE CLASSE



CLASSE VS OBJET

Classe



Objets

Compte de Pierre

Compte de Marie

Compte de Sophie

Instancier une classe

La classe est le moule qui sert à fabriquer les objets. Le nombre d'objets que l'on peut créer à l'aide d'une classe est illimité. On appelle l'opération qui consiste à créer un objet instanciation et l'objet ainsi créé peut aussi être appelé instance de classe.

Dans le script de l'application ce sont uniquement les objets que nous allons manipuler. Ils sont représenté sous la forme d'une variable de type objet.

CONCEPTS DE LA POO

L'abstraction

L'encapsulation

L'héritage

Le polymorphisme

Le principe de l'abstraction

L’abstraction est un principe qui consiste à ignorer certains aspects d’un sujet qui ne sont pas importants pour le problème dans le but de se concentrer sur ceux qui le sont.

Dans un framework comme Symfony, le développeur utilise les classes qui ont été implémentées par les créateurs. Il se concentre sur son objectif en utilisant les méthodes proposées sans pour cela maitriser ou intervenir dans le noyau du framework. C'est l'idée du développeur de classes et de l'utilisateur des classes.

Par analogie, on peut le comparer au cockpit de l'avion. Le pilote utilise les commandes sans se préoccuper de la manière dont elles ont été conçues.

Le principe de l'encapsulation

L’encapsulation désigne le principe de regroupement des données et du code qui les utilise au sein d’une même unité.

On va très souvent utiliser le principe d’encapsulation afin de protéger certaines données des interférences extérieures en forçant l’utilisateur à utiliser les méthodes définies pour manipuler les données.

Exemple: l'accès au solde du compte (propriété) ne sera accessible dans le script que par la méthode consulterSolde()

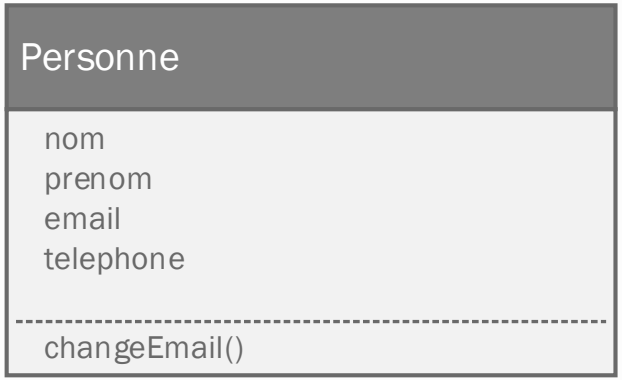
Principe de base: moins vous rendez les données accessibles mieux c’est.

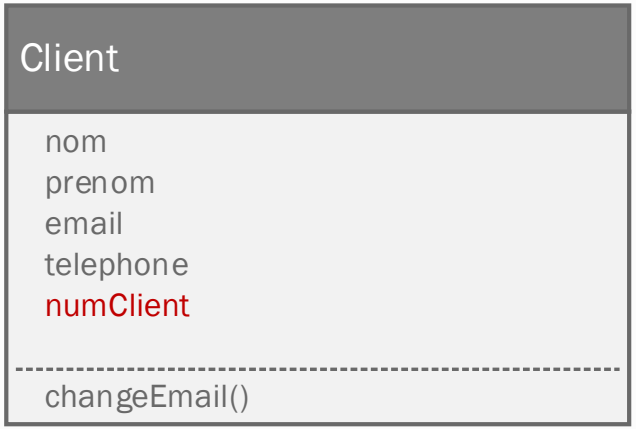
Le principe de l'héritage

Un des grands intérêts de la POO est qu'on va pouvoir rendre notre code très modulable, ce qui va être très utile pour gérer un gros projet ou si on souhaite le distribuer à d'autres développeurs.

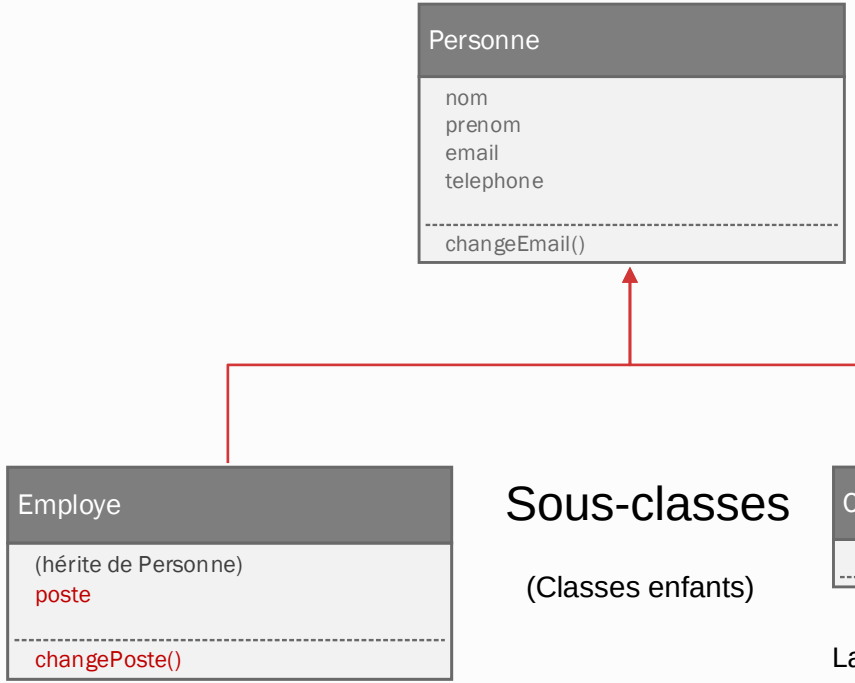
Cette modularité va être permise par le principe de séparation des classes qui est à la base même du PHP et par la réutilisation de certaines classes ou par l'implémentation de nouvelles classes en plus de classes de base déjà existantes.

Sur ce dernier point, justement, il va être possible plutôt que de créer des classes complètement nouvelles d'étendre (les possibilités) de classes existantes, c'est-à-dire de créer de nouvelles classes qui vont hériter des méthodes et propriétés de la classe qu'elles étendent (sous réserve d'y avoir accès) tout en définissant de nouvelles propriétés et méthodes qui leur sont propres.


Personne
nom
prenom
email
telephone
changeEmail()


Client
nom
prenom
email
telephone
numClient
changeEmail()

L'HERITAGE



Superclasse

(Classe parente)

Sous-classes

(Classes enfants)

La classe Client hérite de la classe Personne

La classe Employé hérite de la classe Personne

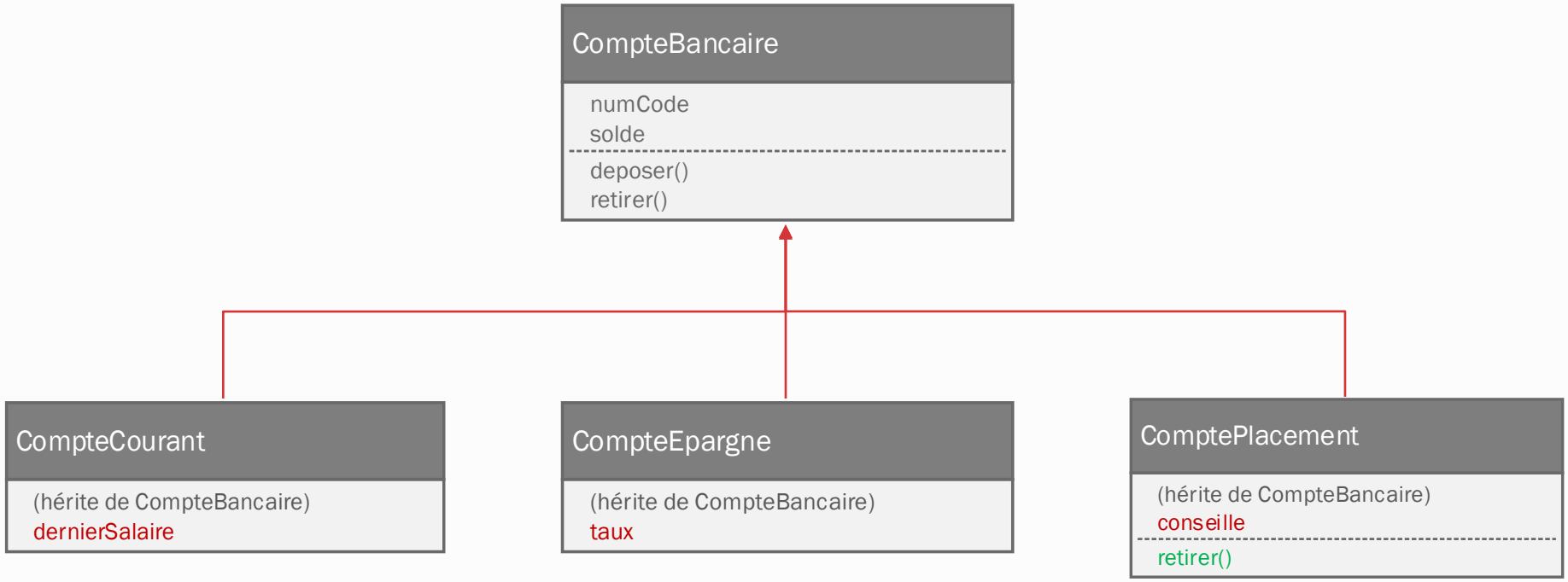
Le principe du polymorphisme

Le terme polymorphisme signe "prendre plusieurs formes" et est utilisé en orienté objet dans le cadre de l'héritage. Il existe plusieurs types de polymorphisme que nous étudierons par la suite mais pour faire simple, il permet de redéfinir l'implémentation d'une méthode de la classe mère dans les classes enfants.

L'intérêt principal de ce procédé est qu'il nous permet d'utiliser une méthode sans ce soucier de son implémentation et de son instanciation.

Exemple:

Le fait de retirer de l'argent sur un compte courant n'entraine pas les mêmes procédures que sur un compte de placement ou là il y a des frais par exemple.



Polymorphisme d'héritage
Surcharge

LA POO EN PHP

1. Introduction
2. Classes et objets
3. Manipuler les objets
4. Encapsulation et la visibilité
5. Le constructeur
6. Constantes de classe
7. Propriétés et méthodes statiques
8. L'héritage
9. L'autoloading
10. La documentation: PHPDoc

Nous allons appréhender les principes fondamentaux de l'orienté objet en PHP à travers l'exemple des utilisateurs d'un site web.
Il existe plusieurs approches d'apprentissage mais je préfère un exemple concret bien que celui-ci ne soit pas fonctionnel. Cela dépasserait largement le cadre du cours.

L'objectif principal est de vous familiariser avec la terminologie et les concepts de l'orienté objet afin d'être capable de finaliser un projet web à l'aide d'un framework dédié.

Définition d'une classe

Les classes sont définies dans des fichiers séparés qui portent le même nom que la classe. Par convention, le système de nommage est le UpperCamelCase: NomDeClasse.

Le nom des classes s'écrit au singulier.

```
class ClassName
{
    // Déclaration des attributs (variables)

    // Déclaration des méthodes (fonctions)
}
```

Instanciation d'une classe (création d'un objet)

La création d'un objet à partir d'une classe se fait à l'aide du mot clé new. Une instance de la classe est créée est sera stockée dans une variable de type objet. C'est cet objet (variable) que nous allons manipuler dans le script.

```
$objectName1 = new ClassName();
$objectName2 = new ClassName();
```

```php
class User
{
    public $name;

    public function sayWelcome(): string
    {
        return 'Bienvenue sur notre site';
    }
}
```

Propriété (publique)

Méthode (publique)

```php
<?php
    $user = new User();
    $user->name = 'John Doe';
    echo '<p>'.$user->sayWelcome().'</p>';
    echo '<p>'.$user->name.' '.$user->sayWelcome().'</p>';
?>
```

Instanciation (objet)

Assignation (valeur)

Invocation méthode

Invocation propriété publique

Explications pour la classe

1. La propriété $name (variable) est précédée d'un mot clé définissant sa visibilité. Nous en reparlerons après. Nous pouvons lui assigner une valeur mais elle la partagera avec tous les objets.

2. La méthode sayWelcome() (fonction) retourne une simple string et possède la même visibilité que la propriété. La sortie HTML est gérée par le script et pas dans la méthode.

Explications pour le script

1. On instancie la classe avec le mot clé new et on enregistre l'objet dans une variable.
2. On affecte une valeur à la propriété $name. Comme nous l'avons définie en publique, nous y avons accès en dehors de la classe. Cette pratique est déconseillée mais nous allons y revenir.
3. Pour accéder aux propriétés et aux méthodes de la classe on utilise l'opérateur objet (->) sans indiquer le signe $ pour les propriétés.
4. On invoque la méthode sayWelcome() sur l'objet de la même manière.

Modificateurs de portée

La déclaration d'attributs ou de méthodes dans une classe se fait en écrivant le nom de l'attribut ou de la méthode à créer, précédé de sa visibilité. Cette visibilité est représentée par les modificateurs de portée que l'on désigne par des mots-clés:

public (propriétés et méthodes accessibles en dehors de la classe)
private (propriétés et méthodes accessibles uniquement au sein de la classe)
protected (identique à private mais accessibles dans les classes héritées)

Ils permettent de restreindre et de masquer ce qui ne doit pas être manipulé en dehors de la classe.

Conventions

Généralement, les propriétés de la classe seront déclarées en « private » et les méthodes en « public ».

Alors, comment afficher ou modifier la valeur d'une propriété sur un objet?
La solution est simple : nous allons implémenter des méthodes dont le seul rôle sera de nous donner l'attribut qu'on leur demande ! Ces méthodes ont un nom bien spécial : ce sont des accesseurs (ou getters).

Pour la modification des valeurs on parle de mutateurs (ou setters). Dans ce cas, la classe devra vérifier l'intégrité des données (type, valeurs…) avant de modifier la propriété. Par convention, ces méthodes commencent par get ou set suivies du même nom que l'attribut dont elles renvoient ou modifient la valeur.

VISIBILITE DES ATTRIBUTS ET DES METHODES

Classe

```php
class User01
{
    private $name;

    public function connect(): string
    {
        return 'Vous êtes connecté en tant que ';
    }
    public function getName(): string
    {
        return $this->name;
    }
    public function setName(string $name): void
    {
        $this->name = $name;
    }
}
```

Objet

```php
require '../src/User01.php';

$user = new User01();

$user->setName('John Doe');

echo $user->connect().' '.$user->getName();
```

La pseudo-variable $this

le mot clef $this est appelé pseudo-variable et sert à faire référence à l'objet couramment utilisé (l'objet en cours).

Cela signifie que lorsqu'on va appeler une méthode depuis un objet, la pseudo-variable $this va être remplacée (substituée) par l'objet qui utilise la méthode actuellement. Dans l'exemple $this fait référence à l'objet $user. Elle permet d'accéder aux propriétés et aux méthodes à l'intérieur de la classe.

Les objets peuvent échanger ou communiquer entre eux. Dans notre exemple un administrateur peut bannir un utilisateur.

Classe

```php
public function ban(User02 $user): string
{
    return $user->getName().' a été banni par '.$this->getName();
}
```

Objets

```php
$user = new User02();
$user->setName('John Doe');
$admin = new User02();
$admin->setName('Vincent Vega');

echo $admin->ban($user); // John Doe a été banni par Vincent Vega
```

La méthode Constructeur

le constructeur d'une classe est une méthode qui va être appelée (exécutée) automatiquement à chaque fois que l'on va instancier une classe.

Le constructeur va ainsi nous permettre d'initialiser des propriétés dès la création d'un objet. Jusqu'à présent, pour assigner une données à une propriété, nous utilisions un setter() après l'instanciation. Avec le constructeur, c'est possible dés l'instanciation de l'objet.

Lors de l'instanciation de notre classe User, PHP va automatiquement rechercher une méthode __construct() dans la classe à instancier et exécuter cette méthode si elle est trouvée.

LE CONSTRUCTEUR

Classe

```php
class User03
{
    private $name;
    private $password;

    public function __construct(string $name, string $password)
    {
        $this->name = $name;
        $this->password = $password;
    }

    public function getName(): string
    {
        return $this->name;
    }
}
```

Objets

```php
$user = new User03('John Doe', 'azerty');
$admin = new User03('Vincent Vega', 'qwerty');

echo $user->getName();
```

On va définir deux paramètres dans notre constructeur qu'on appelle ici $name et $password. Ensuite, nous allons pouvoir passer les valeurs à notre constructeur lors de l'instanciation de notre classe. On va ici passer un nom d'utilisateur et un mot de passe.

Dans le cas présent, vous remarquerez que l'assignation des valeurs se fait directement par affectation des propriétés. Nous n'utilisons pas de setters pour assigner les propriétés. Si les setters existent et qu'ils permettent de contrôler la validité des données vous devrez les utiliser.

```php
class User03
{
    private $name;
    private $password;

    public function __construct(string $name, string $password)
    {
        $this->setName($name);
        $this->setPassword($password);
    }

    public function setName(string $name): void
    {
        $this->name = $name;
    }

    public function setPassword(string $password): void
    {
        // logique de validation
        $this->password = $password;
    }
}
```

Dans cet exemple, l'assignation des valeurs passe par les setters(). Il est préférable d'utiliser cette pratique qui permet plus de contrôle sans surcharger le contrôleur.

Les constantes de classe

En POO vous disposez également des constantes qui reposent sur le même principe. Pour définir une constante de classe, on va utiliser le mot clef const suivi du nom de la constante en majuscules (sans mettre le symbole $) . Le nom d'une constante va respecter les règles communes de nommage PHP.

On définit également la visibilité d'une constante en utilisant les mots clés: public, private et protected.

Par défaut (si rien n'est précisé), une constante sera considérée comme publique et on pourra donc y accéder depuis l'intérieur et depuis l'extérieur de la classe dans laquelle elle a été définie.

Les constantes sont allouées une fois par classe, et non pour chaque instance de classe. Cela signifie qu'une constante appartient à la classe et non pas à un objet en particulier et que tous les objets d'une classe vont donc partager cette même constante de classe.

Définition et utilisation externe

```php
public const BASE_SUBSCRIPTION = 50;
```

Nous créons une constante que nous appelons BASE_SUBSCRIPTION (abonnement de base) et nous lui affectons une valeur de 50. L'assignation d'une valeur est obligatoire dans le cas d'une constante.

```php
$doe = new User04('John Doe', 'azerty');
echo 'Prix abonnement de base: '.$doe::BASE_SUBSCRIPTION;
echo '<br>Ou<br>';
echo 'Prix abonnement de base: '.User04::BASE_SUBSCRIPTION;
```

Pour utiliser une constante à l'extérieur de la classe, nous utilisons l'opérateur de résolution de portée :: (double deux points)

Utilisation interne (dans la classe)

Imaginons que nous souhaitions accorder une réduction de l'abonnement en fonction du poste occupé par l'utilisateur. Nous devons ajouter les propriétés $occupation (poste occupé) et $subscription (l'abonnement avec ou sans la réduction).

```
private $occupation;
private $subscription;

public const BASE_SUBSCRIPTION = 50;
```

Il faut aussi modifier le constructeur pour qu'il prenne en charge la propriété $occupation et l'ajouter lors de l'instanciation de la classe.

Utilisation interne (dans la classe)

Maintenant, ajoutons une méthode permettant de calculer la réduction. Ici une réduction de 10 euros si l'utilisateur est étudiant.

```php
public function setSubscription()
{
    if($this->occupation == 'étudiant') {
        $this->subscription = self::BASE_SUBSCRIPTION - 10;
    } else {
        $this->subscription = self::BASE_SUBSCRIPTION;
    }
}
```

Dans le cas où on tente d'accéder à la valeur d'une constante depuis l'intérieur d'une classe, il faudra utiliser l'un des deux mots clefs self ou parent qui vont permettre d'indiquer qu'on souhaite accéder à une constante définie dans la classe même (self) ou à une constante définie dans une classe mère (parent).

Propriétés statiques

Une propriété statique appartient à la classe dans laquelle elle a été définie et non pas aux objets.

A ne pas confondre avec une constante de classe. La propriété statique contrairement à la constante peut changer de valeur dans le script ou dans la classe. Mais comme elle appartient à la classe et est partagée par tous les objets si un objet modifie la valeur elle le sera pour tous les autres.

Dans notre exemple, nous souhaiterions ajouter la possibilité de bannir des utilisateurs et de les stocker dans une propriété (array). La liste des bannis sera modifiable et partagée par tous les administrateurs.

Dans ce cas nous avons besoin d'une propriété statique qui ne va pas appartenir à un objet mais à la classe entière.

Exemple

Classe

```php
private static array $ban = [];

public function setBan(string $user): void
{
    self::$ban[] .= $user;
}

public function getBan(): string
{
    $val = '';
    foreach(self::$ban as $value) {
        $val .= $value.'<br>';
    }
    return $val;
}
```

Objets

```php
$admin = new User05('John Doe');
$superAdmin = new User05('Vincent
Vega');
$user01 = new User05('Peter Chapman');
$user02 = new User05('Alysa Ford');
$admin->setBan($user01->getName());
$superAdmin->setBan($user02->getName());

echo '<h3>Liste des bannis de ' .
$admin->getName() . '</h3>';
echo $admin->getBan();

echo '<h3>Liste des bannis de ' .
$superAdmin->getName() . '</h3>';
echo $superAdmin->getBan();
```

Analyse du code

1. Pour ne pas surcharger la classe, je n'ai pas indiqué la propriété $name, la constructeur et le getName().
2. Pour stocker les membres bannis j'utilise une propriété statique sous la forme d'un tableau: array $ban.
3. La méthode setBan() prend en paramètre le nom de l'utilisateur que l'on souhaite bannir et l'ajoute à la propriété statique.
4. La méthode getBan() retourne la liste des utilisateurs bannis à l'aide d'une boucle sur la propriété. Cette méthode aurait pu retourner simplement la variable tableau et c'est dans le script que l'on aurait utilisé la boucle.

> La propriété statique $ban va retourner l'ensemble des membres bannis par tous les administrateurs contrairement à une propriété classique qui n'aurait retourné que les membres bannis de l'administrateur concerné.

Méthode statique

Dans le cadre des méthodes, on utilise aussi le mot-clé static pour écrire une méthode qui ne sera pas appelée sur l'objet lui même mais qui sera appelée en dehors du contexte d'instance.

Les méthodes statiques sont des méthodes qui sont faites pour agir sur une classe et non sur un objet. Par conséquent, Il n'y aura aucun $this dans la méthode ! En effet, la méthode n'étant appelée sur aucun objet, il serait illogique que cette pseudo variable existe.

Dans la classe:

```
public static function maFonction(){
    // Code
}
```

Dans le script:

```
MaClasse::maFonction();
```

Exemple

Dans l'exemple précédent, la méthode getBan() doit être appelée sur un objet. Il serait plus logique qu'elle appartient à la classe et non pas à un objet particulier. Pour ce faire il suffit de lui ajouter le mot clé static.

Ainsi, dans le script, nous pourrions l'invoquer sur la classe et non plus sur les objets.

```php
public static function getBan(): string
{
    $val = '';
    foreach(self::$ban as $value) {
        $val .= $value.'<br>';
    }
    return $val;
}
```

```php
echo '<h3>Liste des bannis du site web</h3>';
echo User06::getBan();
```

Notion d'héritage

Il s'agit d'un des principes fondamentaux de la programmation orientée objet. L'héritage va nous permettre de rendre notre code plus modulable, moins répétitif et aussi de permettre à d'autres développeurs de proposer de nouvelles fonctionnalités sans casser la structure du code originel.

Avec l'héritage, une classe hérite des propriétés et des méthodes d'une autre classe. Elle pourra également implémenter ses propres propriétés et méthodes. On dit alors qu'elle "étend" la classe parente.

Comment identifier un héritage:
Soit deux classes A et B. Pour qu'un héritage soit possible, il faut que vous puissiez dire que B est un A. Par exemple, un magicien est un personnage, donc héritage. Un chien est un animal, donc héritage. Un administrateur est un utilisateur héritage aussi.

```php
class User
{
    // propriétés

    // méthodes
}

class Admin extends User
{
    // propriétés

    // méthodes
}

class Customer extends User
{
    // propriétés

    // méthodes
}
```

Classe mère User

Classe fille Admin
Hérite des propriétés et des méthodes de la classe User
Définit ses propres propriétés et méthodes

Classe fille Customer
Hérite des propriétés et des méthodes de la classe User
Définit ses propres propriétés et méthodes

Exemple pour illustrer la théorie

Pour être le plus clair possible, les classes de cet exemple seront simplifiées au maximum.
Dans notre site web, nous avons des utilisateurs spécifiques ou particuliers: des clients. Ils disposent d'un montant (prepaid) qu'ils peuvent dépenser en achetant des formations ou autres...

La classe mère (User) dispose dans notre cas de deux propriétés $name et $password, d'un constructeur pour initialiser et de deux getters getName() et getPassword.

La classe enfant (Customer) va hériter des fonctionnalités de la classe mère et possédera en plus une propriété $prepaid (100 euros pour tout le monde), d'une méthode dépenser spend() et d'une autre getPrepaid() pour obtenir le solde.

```php
class User07
{
    private string $name;
    protected string $password;

    public function __construct(string $name, string $password)
    {
        $this->name = $name;
        $this->password = $password;
    }

    public function getName(): string
    {
        return $this->name;
    }

    public function getPassword(): string
    {
        return $this->password;
    }
}
```

Classe fille Customer

```php
class Customer extends User07
{
    private float $prepaid = 100;

    public function spend($amount): void {
        $this->prepaid -= $amount;
    }

    public function getPrepaid(): float
    {
        return $this->prepaid;
    }
}
```

Script

```php
require '../src/User07.php';
require '../src/Customer.php';

$doe = new Customer('John Doe', 'AZERTY');
echo $doe->getName(); // John Doe
echo '<br>';
echo $doe->spend(25);
echo '<br>';
echo $doe->getPrepaid(); // 75
```

Le cas des propriétés protégées (protected)

La classe fille a hérité de toutes les méthodes et d'aucunes propriétés de la classe mère. Toutes les méthodes sont publiques et toutes les propriétés sont privées. En fait, les propriétés privées ont bien été héritées aussi, mais notre classe fille ne pourra s'en servir. Il n'y a que les méthodes de la classe parente qui auront accès à ces propriétés. C'est comme pour le principe d'encapsulation : ici, les éléments privés sont masqués.

Pour que la classe fille puisse manipuler les propriétés de la classe mère, il faut modifier la portée de visibilité. Elles doivent alors être déclarées en « protected » et non plus en « private ».

Exemple

Dans le même exemple que le précédent, on souhaite que les clients puissent modifier leur password (raison de sécurité). Pour ce faire nous allons ajouter la méthode setPassword() uniquement dans la classe Customer (logique).

Le problème c'est que nous n'aurons pas accès à la propriété $password de la classe mère puisqu'elle est définie en private (uniquement accessible dans la classe ou elle est déclarée). Pour y avoir accès depuis la classe fille nous devrons la déclarer en protected.

Exemple

Classe mère

```
protected string $password;
```

Classe fille

```
public function setPassword($newPassword): void
{
    $this->password = $newPassword;
}
```

Le cas du constructeur

Nous souhaiterions passer par le constructeur de la classe fille pour initialiser la propriété $prepaid. Jusqu'à maintenant tous les objets partagent la même valeur.

Dans la classe mère, le constructeur ne permet pas de le faire et c'est normal car la propriété $prepaid ne lui appartient pas. Nous allons devoir redéfinir le constructeur dans la classe fille. Il s'agit de notions de polymorphisme que nous allons étudier après mais voici un exemple (classe Customer).

```php
private float $prepaid;

public function __construct(string $name, string $password, float $prepaid)
{
    parent::__construct($name, $password);
    $this->prepaid = $prepaid;
}
```

La surcharge des méthodes et des propriétés (overloading)

On dit qu'on « surcharge » une propriété ou une méthode d'une classe mère lorsqu'on la redéfinit dans une classe fille.

Pour surcharger une propriété ou une méthode, il va falloir la redéclarer en utilisant le même nom. Par ailleurs, si on souhaite surcharger une méthode, il faudra également que la nouvelle définition possède le même nombre de paramètres.

La nouvelle définition doit obligatoirement posséder un niveau de restriction de visibilité plus faible ou égal, mais ne doit en aucun cas avoir une visibilité plus restreinte que la définition de base.

Notez qu'il va être relativement rare d'avoir à surcharger des propriétés. Généralement, nous surchargerons plutôt les méthodes d'une classe mère depuis une classe fille.

Exemple

Sur un site web, il est facile d'imaginer un utilisateur de base qui pourrait afficher les articles postés par contre un administrateur aurait en plus accès à ceux qui ne sont pas encore validés.

Le titre des articles et la visibilité (bool) sont stockés dans un tableaux array articles.

Classe mère

```php
class User08
{
    protected $name;
    protected static $articles = [
        'Symfony 5' => true,
        'PHP 8'     => false,
        'HTML 5'    => true
    ];
    public function getArticles()
    {
        foreach(self::$articles as $article => $key) {
            if($key) {
                echo $article.'<br>';
            }
        }
    }
}
```

Pour des raisons de visibilité, j'ai retiré le constructeur et les autres méthodes.

La méthode getArticles() affiche les articles dont la valeur est a true.

Classe fille

```php
class Admin extends User08
{
    public function getArticles()
    {
        foreach(self::$articles as $article => $key) {
            echo $article.'<br>';
        }
    }
}
```

Script

```php
$doe = new User08('John Doe'); // User
$vega = new User09('Vincent Vega'); // Admin
$doe->getArticles(); // User articles
$vega->getArticles(); // Admin articles
```

La méthode getArticles() est surchargée car elle contient un code différent de la méthode homonyme de la classe mère.

Elle affiche tous les articles (true et false).

Polymorphisme d'héritage (overriding)

Il ressemble à la surcharge mais la méthode de la classe fille hérite des fonctionnalités de celle de la classe mère et elle propose en plus ses propres fonctionnalités.

Classe mère

```php
class Admin extends User09
{
    public function getArticles()
    {
        foreach(self::$articles as $article => $key) {
            echo $article.'<br>';
        }
    }

    public function viewRoles(): void
    {
        echo 'Ajouter<br>
              Modifier<br>';
    }
}
```

Classe fille

```php
class SuperAdmin extends Admin
{
    public function getArticles()
    {
        foreach(self::$articles as $article => $key) {
            echo $article.'<br>';
        }
    }

    public function viewRoles(): void
    {
        parent::viewRoles();
        echo 'Supprimer';
    }
}
```

Script

```php
$doe = new Admin('John Doe');
$vega = new SuperAdmin('Vincent Vega');
$doe->viewRoles();
$vega->viewRoles();
```

Les classes abstraites

En POO, il est possible de mettre en place sur nos classes et nos méthodes des contraintes. On parlera alors d'abstraction ou de finalisation suivant la contrainte instaurée.

On a vu jusqu'à maintenant que l'on pouvait instancier n'importe quelle classe afin de pouvoir exploiter ses méthodes. On va maintenant découvrir comment empêcher quiconque d'instancier une classe parente pour n'utiliser que les classes enfants.

Imaginons que dans notre site, nous aillons une classe mère qui regroupe tout ce que les utilisateurs ont en commun et des classes filles par type d'utilisateur (membres, clients, abonnés...). Vu que sur notre site nous n'avons que des membres, des clients et des abonnés il ne sera jamais nécessaire d'instancier la classe utilisateur. Pour empêcher cela, nous allons rendre la classe utilisateur (classe mère) abstraite. Il suffit de commencer la déclaration de la classe par le mot clé abstract

Classe mère

```php
abstract class User10
{
    // Propriétés

    // Méthodes

}
```

Script

```php
require '../src/User10.php';
require '../src/Admin.php';
require '../src/SuperAdmin.php';


$doe = new User10('John Doe');
// Fatal error: Uncaught Error:
Cannot instantiate abstract class
User10
```

La classe mère étant définie comme abstraite, il ne sera plus possible de l'instancier. Vous devrez obligatoirement passer par les classes filles.

Les méthodes abstraites

Une méthode abstraite est une méthode dont seule la signature (c'est-à-dire le nom et les paramètres) va pouvoir être déclarée mais pour laquelle on ne va pas pouvoir déclarer d'implémentation (c'est-à-dire le code dans la fonction ou ce que fait la fonction).

Dès qu'une classe possède une méthode abstraite, il va falloir la déclarer comme classes abstraite.

Une classe abstraite n'est pas structurellement différente d'une classe classique (à la différence de la présence potentielle de méthodes abstraites) et qu'on va donc tout à fait pouvoir ajouter des constantes, des propriétés et des méthodes classiques dans une classe abstraite.

Les méthodes abstraites

Quel intérêt d'avoir des méthodes qui ne font rien ?

Cela va obliger les classes filles à implémenter ces méthodes et à définir leurs fonctionnalités. Cette façon de procéder va nous permettre de mieux contrôler la hiérarchie et la structure de votre code.

Syntaxe:

```php
abstract class User10
{
    protected string $name;

    public function __construct($name)
    {
        $this->name = $name;
    }

    abstract public function viewRoles(): void;
}
```

```php
class Admin extends User10
{
    public function viewRoles(): void
    {
        echo 'Ajouter<br>
              Modifier<br>';
    }
}
```

Les classes et les méthodes finales

Une classe qui est définie avec le mot clé final, ne pourra pas être étendue et une méthode définie de la même manière, ne pourra être surchargé dans une classe fille. Il s'agit ici de mettre un terme à l'héritage et empêcher l'extension et l'emploi abusif des classes et des méthodes (framework, librairies).

L'auto chargement des classes

En PHP orienté objet, nous créons une classe par fichier pour conserver une meilleure visibilité dans l'architecture générale du projet et simplifier la maintenabilité du code en séparant les différentes fonctionnalités.

L'inconvénient est que l'on va devoir écrire dans le script une longue série d'inclusion de classes pour pouvoir les utiliser. De plus, il arrive que certaines d'entre elles ne soit pas nécessaires.

PHP propose une fonction spl_autoload_register() qui prend en paramètre une ou plusieurs fonctions dont le rôle est de n'inclure que les classes dont le script à besoin.

Syntaxe

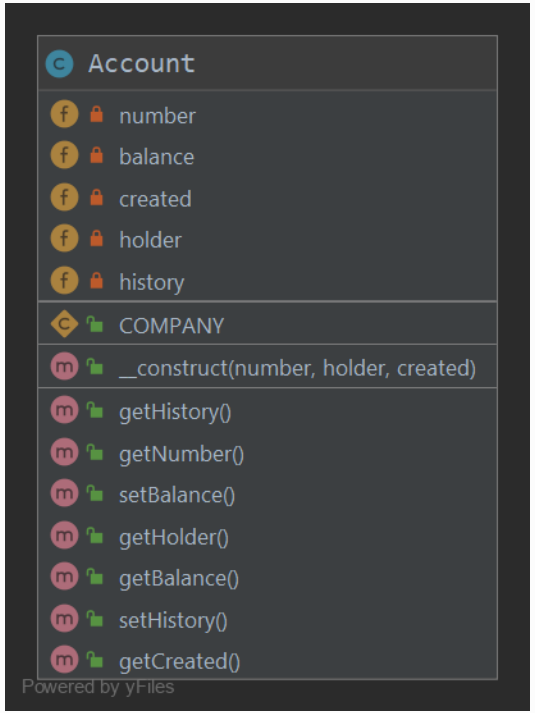
```php
// Fonction d'inclusion des classes
function my_autoloader($class) {
    require '../src/' . $class . '.php';
}

// Fonction d'autoloading
spl_autoload_register('my_autoloader');
```

En deux étapes

```php
// Une seule fonction avec en param une closure
spl_autoload_register(function($class) {
    require '../src/' . $class . '.php';
});
```

En une seule étape avec une fonction anonyme

Gestion de comptes bancaires
Step 01



La documentation - PHPDoc

Les commentaires du code en PHP ainsi que les recommandations d'écriture et de syntaxe sont formulés dans un standard appelé PSR (PHP Standard Recommandations). L'objectif principal étant l'interopérabilité entre les différents frameworks et l'harmonisation des méthodes de développements.

En ce qui concerne la documentation du code il s'agit du PSR-5 qui propose le standard PHPDoc. La documentation réalisée suivant cette norme pourra par la suite être générée et finalisée par des utilitaires dédiés (PHPDocumentor).

Dans le cas des IDE, Les Doc Block peuvent partiellement être générés automatiquement et permettent par la suite une meilleure auto complétion des méthodes et des fonctions.

Toutes les annotations PHPDoc sont contenues dans DocBlocks et sont illustrées par une ligne multiple avec deux astérisques.

Les annotations de fichiers

Les annotations (métadonnées) de niveau fichier s'appliquent à tout le code du fichier et doivent être placées en haut du fichier:

```php
<?php

/**
 * @author John Doe (jdoe@example.com)
 * @copyright MIT
 */
```

Mots clés ou tags:
@author
@copyright

Décrire une variable

Le mot clé @var peut être utilisé pour décrire le type et l'utilisation de:

une propriété de classe
une variable locale ou globale
une classe ou constante globale

Le type peut être l'un des types PHP intégrés ou une classe définie par l'utilisateur, y compris les espaces de noms.

Le nom de la variable doit être inclus, mais peut être omis si le bloc de documents s'applique à un seul élément.

Exemple

```php
/**
 * @var string FirstName and LastName
 */
protected string $name;

/**
 * @var int Min Password length
 */
protected const MINPASSWORD = 8;
```

Mot clé ou tag:
@var
@var type [name] [description]

Décrire une méthode

Lors de la création d'une annotation pour fonction ou méthode on indique dans l'ordre:

La description si nécessaire
Les paramètres
Le return

Pour cela on utilise les tags @param et @return

Exemple

```php
/**
 * Cut a string to create a see more...
 *
 * @param string $txt
 * @param int $length
 * @return string
 */
public function cutString(string $txt, int $length): string {

}
```

Mots clés ou tags:
@param
@return

BIBLIOGRAPHIE

Publications

DELANNOY Claude. 2015. *S'initier à la programmation et à l'orienté Objet*. Eyrolles. 362 p.

GAMBELLI Julien. 2015. *Apprendre à développer un site web responsive et dynamique avec PHP*. ENI. 421 p.

HEURTEL Olivier. 2016. *Développez un site web dynamique et interactif*. ENI. 583 p.

JUNADE Ali. 2015. *Mastering PHP Design Patterns.* Packt Publishing Limited. 270 p.

MARTIN P., PAULI J., PIERRE DE GEYER C., 2016, *PHP 7 avancé*. Eyrolles. 688 p.

ROLLET Olivier. 2015. *Apprendre à développer un site web avec PHP et MySql*. ENI. 576 p.

Sources web

Codecademy - https://www.codecademy.com/learn/php

Développez.com - http://php.developpez.com

Github - https://www.codecademy.com/learn/php

PHP.NET - http://php.net

Stackoverflow - http://stackoverflow.com

Symfony - http://symfony.com/doc/3.1/index.html

W3schools.com - http://www.w3schools.com/php