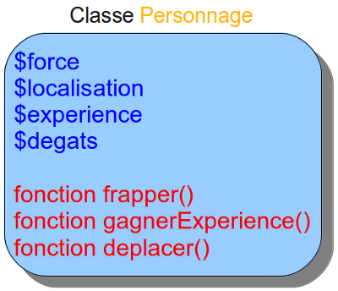
***Object oriented programing***

Ch 1 introduction

In a nutshell, POO is a way of structuring your website for instance, by making objects that will interact with each other. For instance, in a game, there is the object « personnage », qui aura des *caractéristiques* as la force, l’expérience, les dégâts, ….   
All this specs match some values and values are stored in variable.   
  
Besides the specs, a character might also : hit another character, gain experience, move, …   
 Those new possibilities would be what is called « functions ».   
  
In OOP, all these informations are stored in a « class ». A class is an entity wich combine variables and functions. Each function will have acess to the variable of the entity. In the case of the game character here, well have a function called « frapper () », which will only modify the variables « $degats » from the character depending on the variable force. A class is thus a logical gathering of variable and functions and every objects coming from that class will have those.   
  
How to define an Instance ? First, it is the result of an instanciation. Instanciate a class is actualy using a class in order for it to create an object.



As a matter of fact, those a called attributes ( or property or variable ) and methods ( functions). One of the biggest advantage of POO is that it allows the coder to hide some part of the code to the user ( the one who uses the class and not the user that goes to your website from his computer). A big and complex class should have objects within which the user cant modify its attributs. This process is called encapsulation and is there to ensure the validity, types and value about data ‘s objects.   
  
For a quick example : imagine an air pilot which is using loads of buttons and functions but actually never build the said buttons himself. The user of the class should just call method and ignore the attributs. For such a thing to work, attributs must be set to private.

**Créer une classe en php**

<?php

class Personnage // Présence du mot-clé class suivi du nom de la classe.

{

// Déclaration des attributs et méthodes ici.

}

What about the visibility of an attribut or a method ? There are two types : public and private.

If an attribute is public, we can access it from anywhere. From within the object as well as from the outside. The outside of an object is all the code thats not in the class. Whats told to PHP is actualy this : « in that object, give me that attribute » or « in that object, call that method », this is what calling an attribute or method outside of an object is.

If the attribute is private : there will be a few restrictions : we will be able to acess attributes and methodes only from within the class, meaning : only the code that wants to access a private attribut or method within the class, will work. Otherwise, youll get fatal error since the attribut or method is private.   
  
Thanks to the encapsulation principle, we can forbid access to attributes.

To **declare attributes**, here is the way :   
  
<?php

class Personnage

{

private $\_force; // La force du personnage

private $\_localisation; // Sa localisation

private $\_experience; // Son expérience

private $\_degats; // Ses dégâts

}

PEAR notation states that every name of private element must be preceded by an underscore. The value of an attribute cannot be the result of a call to a function.

To declare **a method**

<?php

class Personnage

{

private $\_force; // La force du personnage

private $\_localisation; // Sa localisation

private $\_experience; // Son expérience

private $\_degats; // Ses dégâts

public function deplacer() // Une méthode qui déplacera le personnage (modifiera sa localisation).

{

}

public function frapper() // Une méthode qui frappera un personnage (suivant la force qu'il a).

{

}

public function gagnerExperience() // Une méthode augmentant l'attribut $experience du personnage.

{

}

}

All attributs can be private and for the methods, their visibility do not count. Thats the encapsulation principle.

***How to use the class ?***a) create and manipulate an object :

<?php

$perso = new Personnage;

// here we ware instanciating the class « personnage ».   
to call the method of an object, an operator must be used : « -> ».

At the left of this operator, we place the object that we wanna use. In the exemple above, this object would have been $perso. Right of the operator, we specify the name of the method we wanna call.

<?php

// Nous créons une classe « Personnage ».

class Personnage

{

private $\_force;

private $\_localisation;

private $\_experience;

private $\_degats;

// Nous déclarons une méthode dont le seul but est d'afficher un texte.

public function parler()

{

echo 'Je suis un personnage !';

}

}

$perso = new Personnage;

$perso->parler(); **This line means :** go fetch the object $perso and call the method parler() on this object.

How can a method access the attributes of an object ? When we have an object, we can call methods thanks to the operator -> from which we can, also access the attributes of the class.

<?php

class Personnage

{

private $\_force;

private $\_experience;

private $\_degats;

}

$perso = new Personnage;

$perso->\_experience = $perso->\_experience + 1; // Une erreur fatale est levée suite à cette instruction.

This piece of code leads to a fatal error since its attributes are private.

Dans notre exemple (qui essaye en vain d'augmenter de 1 l'expérience du personnage), il faudra demander à la classe d'augmenter l'expérience. Pour cela, nous allons écrire une méthodegagnerExperience():

<?php

class Personnage

{

private $\_experience;

public function gagnerExperience()

{

// Cette méthode doit ajouter 1 à l'expérience du personnage.

}

}

$perso = new *Personnage*;

$perso->gagnerExperience();

But now, how to access the attribute $\_experience ? This is the time to call the pseudo variable : $this.

In fact, a parameter represents the object and is passed implicitely to each methods of the class, such as in this exemple :   
  
<?php

class Personnage

{

private $\_experience = 50;

public function afficherExperience()

{

echo $this->\_experience;

}

}

$perso = new Personnage;

$perso->afficherExperience();

$ this represent the object we are currently using. . In this script, variable $this and variable $ perso represent the same object. Line 8 to 3 means : display that value in the object used ( here $perso), give me the value of the attributes ( $\_exprience ).

<?php

class Personnage

{

private $\_experience = 50;

public function afficherExperience()

{

echo $this->\_experience;

}

public function gagnerExperience()

{

// On ajoute 1 à notre attribut $\_experience.

$this->\_experience = $this->\_experience + 1;

}

}

$perso = new Personnage;

$perso->gagnerExperience(); // On gagne de l'expérience.

$perso->afficherExperience(); // On affiche la nouvelle valeur de l'attribut.

The method frapper () has an instruction composed of two parts : the first one is to tell PHP that we wanna assign a new value to the attribute $\_degats from the character to hit.   
Second part consists in giving PHP the value that we wanna assign, here we see that this value is reached by $this->\_force.   
  
You have to remember that $this is a variable that represents the object from wichi s called the method. In our case, weve calle the method « frapper () » from the character which is striking and so $this represents the striking character.

<?php

class Personnage

{

private $\_degats; // Les dégâts du personnage.

private $\_experience; // L'expérience du personnage.

private $\_force; // La force du personnage (plus elle est grande, plus l'attaque est puissante).

public function frapper($persoAFrapper)

{

$persoAFrapper->\_degats += $this->\_force;

}

public function gagnerExperience()

{

// On ajoute 1 à notre attribut $\_experience.

$this->\_experience = $this->\_experience + 1;

}

}

How to create two characters, make the first character strike, make the first character gain experience. Make the second character strike and make second character gain experience.

This script is only methods :

<?php

$perso1 = new Personnage; // Un premier personnage

$perso2 = new Personnage; // Un second personnage

$perso1->frapper($perso2); // $perso1 frappe $perso2

$perso1->gagnerExperience(); // $perso1 gagne de l'expérience

$perso2->frapper($perso1); // $perso2 frappe $perso1

$perso2->gagnerExperience(); // $perso2 gagne de l'expérience ?>

But there is a small issue, both characters have the same level of damages, the same experience and the same strenght, in the end, theyll always be equal. To solve that issue, we could assign specific values to both characters in order that the combat can differentiate them.

To solve this issue, use :  **« accesseur » and « mutateur ». in english : getters and setters.**

When implementing methods which are only used to give attribute that we are asking them, we using « **accesseurs »** ( or **getters** ).

By convention, the getters have the same name that the attribute from which they are reflecting the value.

How does it work if we wanna modify an attribute ? Once again, we must ask the class to do it for us. Its by calling methods that we ask our class to modify this or that attribute. The class should contrôle the vlaue in order to ensure its integrity because if it does not do it, we will be able to pass any value to the class and encapsulation would not be respected any more.

The methods uses to do that are called « **Setters »**. they have this form « setNomAttribut () ».

Final exemple :

<?php

$perso1 = new Personnage; // Un premier personnage

$perso2 = new Personnage; // Un second personnage

$perso1->setForce(10);

$perso1->setExperience(2);

$perso2->setForce(90);

$perso2->setExperience(58);

$perso1->frapper($perso2); // $perso1 frappe $perso2

$perso1->gagnerExperience(); // $perso1 gagne de l'expérience

$perso2->frapper($perso1); // $perso2 frappe $perso1

$perso2->gagnerExperience(); // $perso2 gagne de l'expérience

echo 'Le personnage 1 a ', $perso1->force(), ' de force, contrairement au personnage 2 qui a ', $perso2->force(), ' de force.<br />';

echo 'Le personnage 1 a ', $perso1->experience(), ' d\'expérience, contrairement au personnage 2 qui a ', $perso2->experience(), ' d\'expérience.<br />';

echo 'Le personnage 1 a ', $perso1->degats(), ' de dégâts, contrairement au personnage 2 qui a ', $perso2->degats(), ' de dégâts.<br />';

1) creation of two objects : $perso1 and $perso2

2) assignation of value for the strenght and experience of each character

3) $perso1 hit $perso2, $perso2 is taking damages then $perso1 wins experiences.   
4) $perso2 hits $perso 1 : $perso 1 takes damages then $perso2 wins experience.

**What is a constructor ?**

Imagine you have an object for which you need to initialize the attributs as soon as you have created it, without knowing their values yet. For instance, lets say you wish to specify the strenght and damages of a character as soon as you have created it. Actually, the only possibility that is there for you is to modify those attributs manually, once the object is created. In PHP there is something called a « constructor » that will do just that. This constructor is nothing but a  **method written in a class.**

In general, you should avoir using \_\_construct method with private visibility otherwise you wont be able to instanciate the class.

**In a nutshell :**

An object is created thanks to the operator « new »

The access to an attribute or a method of an object is done thanks to the « -> » operator. To read or modify an attribute, we are using getters and setters.

The constructor of a class is used to initialize the object which is currently created. In other words : initialize the value of the attributes.   
  
Classes can be loaded dynamically thanks to the auto load of the class : spl\_autoload\_register.

**Scope resolution operator « :: »**

Is used to call elements belonging to a class and not an object.

So you have to use this operator because a constant belongs to the class and not to an object.

**In a nutshell2 :**

The operator -> is used to access the element of an object

The operator :: is used to access the element of a class

Inside a method, we access the object thanks to the pseudo variable « $this » while we access the class thanks to the keyword « self ».

Attributes and static methods as well as constants of class are elements related to the class, meaning that creating an object to use them is not required

Constant of classes are usefull to avoid « mute code » ie, a coe that, without commentaries, does not inform on how it is actualy working.

**Manipulation of stored datas**

A Table is nothing but a big board where datas are organized.



Exemple of a php scripts that calls the datas stored in the table above here.   
  
<?php

// On admet que $db est un objet PDO

$request = $db->query('SELECT id, nom, forcePerso, degats, niveau, experience FROM personnages');

while ($perso = $request->fetch(PDO::FETCH\_ASSOC)) // Chaque entrée sera récupérée et placée dans un array.

{

echo $perso['nom'], ' a ', $perso['forcePerso'], ' de force, ', $perso['degats'], ' de dégâts, ', $perso['experience'], ' d\'expérience et est au niveau ', $perso['niveau'];

}

What is the composition of a class in php ?

1) a part that declare attributes, they are the « caracteristics » of an object

2) a part that declare methods, those are the functionalities of each objects

3) a part that declare the constants of the class.

There are thus two questions when it comes to creating a class :   
1) what are the characteristics of my objects ?

2) what are the functionalities of my objects ?

How to read a database ?

<?php

// On admet que $db est un objet PDO.

$request = $db->query('SELECT id, nom, forcePerso, degats, niveau, experience FROM personnages');

while ($donnees = $request->fetch(PDO::FETCH\_ASSOC)) // Chaque entrée sera récupérée et placée dans un array.

{

// On passe les données (stockées dans un tableau) concernant le personnage au constructeur de la classe.

// On admet que le constructeur de la classe appelle chaque setter pour assigner les valeurs qu'on lui a données aux attributs correspondants.

$perso = new Personnage($donnees);

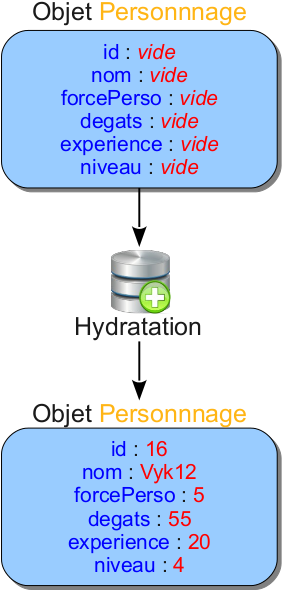
echo $perso->nom(), ' a ', $perso->forcePerso(), ' de force, ', $perso->degats(), ' de dégâts, ', $perso->experience(), ' d\'expérience et est au niveau ', $perso->niveau();

}

**Theory of hydratation**

Mostly used when we are using objects that represent stored data. When we are talking about hydration, we are in fact talking about bringing an object what it needs to work properly. In a nutshell, hydration means bringing to an object datas that are matching its attributes so that it can assign relevant values to those attributes.

The objects has been « hydrated ».



Code snippet of an **hydration in action :**

<?php

class Personnage

{

private $\_id;

private $\_nom;

private $\_forcePerso;

private $\_degats;

private $\_niveau;

private $\_experience;

// Un tableau de données doit être passé à la fonction (d'où le préfixe « array »).

public function hydrate(array $donnees)

{

}

public function id() { return $this->\_id; }

public function nom() { return $this->\_nom; }

public function forcePerso() { return $this->\_forcePerso; }

public function degats() { return $this->\_degats; }

public function niveau() { return $this->\_niveau; }

public function experience() { return $this->\_experience; }

public function setId($id)

{

// L'identifiant du personnage sera, quoi qu'il arrive, un nombre entier.

$this->\_id = (int) $id;

}

public function setNom($nom)

{

// On vérifie qu'il s'agit bien d'une chaîne de caractères.

// Dont la longueur est inférieure à 30 caractères.

if (is\_string($nom) && strlen($nom) <= 30)

{

$this->\_nom = $nom;

}

}

public function setForcePerso($forcePerso)

{

$forcePerso = (int) $forcePerso;

// On vérifie que la force passée est comprise entre 0 et 100.

if ($forcePerso >= 0 && $forcePerso <= 100)

{

$this->\_forcePerso = $forcePerso;

}

}

public function setDegats($degats)

{

$degats = (int) $degats;

// On vérifie que les dégâts passés sont compris entre 0 et 100.

if ($degats >= 0 && $degats <= 100)

{

$this->\_degats = $degats;

}

}

public function setNiveau($niveau)

{

$niveau = (int) $niveau;

// On vérifie que le niveau n'est pas négatif.

if ($niveau >= 0)

{

$this->\_niveau = $niveau;

}

}

public function setExperience($exp)

{

$exp = (int) $exp;

// On vérifie que l'expérience est comprise entre 0 et 100.

if ($exp >= 0 && $exp <= 100)

{

$this->\_experience = $exp;

}

}

}

You should not assign values directly like this :

<?php

// …

public function hydrate(array $donnees)

{

if (isset($donnees['id']))

{

$this->\_id = $donnees['id'];

}

if (isset($donnees['nom']))

{

$this->\_nom = $donnees['nom'];

}

// …

}

// … otherwise you are violating the encapsulation principle.

How to control the integrity of the values ?

<?php

// …

public function hydrate(array $donnees)

{

if (isset($donnees['id']))

{

$this->setId($donnees['id']);

}

if (isset($donnees['nom']))

{

$this->setNom($donnees['nom']);

}

// …

}

// …

**Exemple of how to include SQL querries to a php script :**

<?php

class PersonnagesManager

{

private $\_db; // Instance de PDO.

public function \_\_construct($db)

{

$this->setDb($db);

}

public function add(Personnage $perso)

{

// Préparation de la requête d'insertion.

// Assignation des valeurs pour le nom, la force, les dégâts, l'expérience et le niveau du personnage.

// Exécution de la requête.

}

public function delete(Personnage $perso)

{

// Exécute une requête de type DELETE.

}

public function get($id)

{

// Exécute une requête de type SELECT avec une clause WHERE, et retourne un objet Personnage.

}

public function getList()

{

// Retourne la liste de tous les personnages.

}

public function update(Personnage $perso)

{

// Prépare une requête de type UPDATE.

// Assignation des valeurs à la requête.

// Exécution de la requête.

}

public function setDb(PDO $db)

{

$this->\_db = $db;

}

}

**With the SQL queries instead of the comments :**

<?php

class PersonnagesManager

{

private $\_db; // Instance de PDO

public function \_\_construct($db)

{

$this->setDb($db);

}

public function add(Personnage $perso)

{

$q = $this->\_db->prepare('INSERT INTO personnages(nom, forcePerso, degats, niveau, experience) VALUES(:nom, :forcePerso, :degats, :niveau, :experience)');

$q->bindValue(':nom', $perso->nom());

$q->bindValue(':forcePerso', $perso->forcePerso(), PDO::PARAM\_INT);

$q->bindValue(':degats', $perso->degats(), PDO::PARAM\_INT);

$q->bindValue(':niveau', $perso->niveau(), PDO::PARAM\_INT);

$q->bindValue(':experience', $perso->experience(), PDO::PARAM\_INT);

$q->execute();

}

public function delete(Personnage $perso)

{

$this->\_db->exec('DELETE FROM personnages WHERE id = '.$perso->id());

}

public function get($id)

{

$id = (int) $id;

$q = $this->\_db->query('SELECT id, nom, forcePerso, degats, niveau, experience FROM personnages WHERE id = '.$id);

$donnees = $q->fetch(PDO::FETCH\_ASSOC);

return new Personnage($donnees);

}

public function getList()

{

$persos = [];

$q = $this->\_db->query('SELECT id, nom, forcePerso, degats, niveau, experience FROM personnages ORDER BY nom');

while ($donnees = $q->fetch(PDO::FETCH\_ASSOC))

{

$persos[] = new Personnage($donnees);

}

return $persos;

}

public function update(Personnage $perso)

{

$q = $this->\_db->prepare('UPDATE personnages SET forcePerso = :forcePerso, degats = :degats, niveau = :niveau, experience = :experience WHERE id = :id');

$q->bindValue(':forcePerso', $perso->forcePerso(), PDO::PARAM\_INT);

$q->bindValue(':degats', $perso->degats(), PDO::PARAM\_INT);

$q->bindValue(':niveau', $perso->niveau(), PDO::PARAM\_INT);

$q->bindValue(':experience', $perso->experience(), PDO::PARAM\_INT);

$q->bindValue(':id', $perso->id(), PDO::PARAM\_INT);

$q->execute();

}

public function setDb(PDO $db)

{

$this->\_db = $db;

}

}

**In a nutshell 3**

1) concerning the php script, each recording of the database is represented by an objets that has a list of attributes indentical to the list of the columns from the SQL TABLE.

2) it is possible to hydrate each object thanks to the method « hydrate » which is here to assign values passes in parameters to the corresponding attributes.

3) the communication with the data base is made through the object which is different from the object that is recording ( a class = a role ). Such an object is called a « **manager ».**

4) A manager can store the objects in BDD but can also store them in another support such as an XM file, a text file, ….

**How to think about a big project when we want to code it POO style ?**

Hydration and manager would be two nices concepts to start the work with.

1) we can only speak about heritage between class A and class B if and only if we can say « B is an A, and in that case B inherits from A )

2) a class that herits from another will have access to all its attributes and methods, wether public or protected.

3) The visibility « protected » is equal to the visibility « private ». Exept than one protected element is accessible by the « daughter classes » contrary to the private elements.

4) its possible to forbid the instanciation of a class thanks to the keyword « abstracts

5) its possible to coply « daughter classes » to implement a method thanks to the keyword « abstract » and then to forbid their re writing of a method thanks to the keyword « final »

6) the static resolution allow to know which class has been initially called in order to invoc methods or to access attributs from the called class .

**Magical methods**

Magical methods are methods which are automatically called when a certain event is fired.

Every magical methods starts with two underscores. You should thus avoid to call your method with this same pattern.

MAgical methods which you use most often are : \_\_construct,\_\_set,\_\_getet\_\_call.

***Interfaces***

Interfaces represent behaviour and are nothing but 100% abstract classes.

An interface can be use within a class thanks to the keyword « implements »

Its possible to inherit interfaces thanks to the keyword « extends »

There is a hall pannel of pre defined interfaces that can help when creating a lot of interesting functions such as the creation of an « object array ».

En [programmation orientée objet](https://fr.wikipedia.org/wiki/Programmation_orient%C3%A9e_objet), une **interface** est un ensemble de [signatures](https://fr.wikipedia.org/wiki/Signature_de_type) de [méthodes](https://fr.wikipedia.org/wiki/M%C3%A9thode_(informatique)) publiques d'un [objet](https://fr.wikipedia.org/wiki/Objet_(informatique)).

Il s'agit donc d'un ensemble de méthodes accessibles depuis l'extérieur d'une classe, par lesquelles on peut modifier un objet, ou plus généralement communiquer avec lui. Pour rappel, la différenciation entre méthodes publiques et méthodes privées introduit une [abstraction](https://fr.wikipedia.org/wiki/Abstraction_(informatique)#Programmation_orient%C3%A9e_objet) qui :

* empêche le programmeur d'application (qui emploie une classe) de manipuler l'objet de façon indue, puisque les seules modifications possibles d'une instance sont celles indiquées comme publiques par le concepteur de la classe ;
* permet au programmeur de la classe, de modifier l'implémentation interne de ces méthodes de manière transparente.

Selon le langage de programmation, l'utilité, la portée et la manière de décrire une interface, varient. Néanmoins il est commun à toutes les interfaces de déclarer chacune de leurs [méthodes](https://fr.wikipedia.org/wiki/M%C3%A9thode_(informatique)) sous la forme d'une [signature](https://fr.wikipedia.org/wiki/Signature_de_type) :

* *nomDeFonction (typeArgument1:in/out, typeArgument2:in/out, ... , typeArgumentN:in/out)*.

La sémantique de cette signature dépend aussi du langage.

**Exeptions**

**Exemple of PDOException**

<?php

try

{

$db = new PDO('mysql:host=localhost;dbname=tests', 'root', ''); // Tentative de connexion.

echo 'Connexion réussie !'; // Si la connexion a réussi, alors cette instruction sera exécutée.

}

catch (PDOException $e) // On attrape les exceptions PDOException.

{

echo 'La connexion a échoué.<br />';

echo 'Informations : [', $e->getCode(), '] ', $e->getMessage(); // On affiche le n° de l'erreur ainsi que le message.

}

***Traits are a way to exeternalise code***

They are actualy definig methods that classes can use. A trait is also considered as a mini class.

They are used thanks to the keyword « use »

Its possible to use an infinity of traits within a class by resolving potential conflicts with « insteadof »

A trait can use another trait

Its possible to change the visibility of a method as well as her name by using the keyword « as ».

***Reflectivity API***

How to display on the screen that a class inherit from another ? its a principle known by programmers but not by the computer. Reflectivity API are there to solve that issue.

ReflectionClass : when you instanciate it, you specify the name of the class from which you wanna obtain informations.

In order to know if the class has this or that attribut, you can use : ReflectionClass ::hasProperty($attributeName).

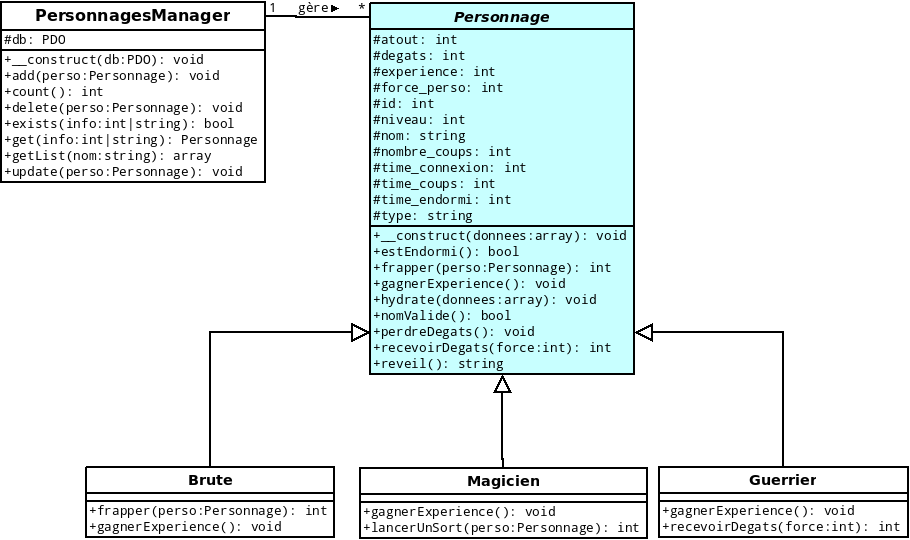
For the methods : ReflectionClass::hasMethod($methodName)

***Unified modeling language***

Its a language that allow to modelise classes and their interactions. Its done through a diagram : youre drawing your classes and you are linking them by following specific conventions. This will allow you to better visualise your application and so to better think about it.

Its rather a methodology than a true « language ».

Thanks to the diagrames of the UML you’ll be able to represent your app : its functionning, how it is run, actions that might be executed by the app, …. Those diagrames will allow the ones that have absolutely no knwoledge in computer science to understand what your app is doing.   
  
exemple :



***Logiciel Dia is used to draw those diagrams***

After drawing those diagrams, you can export them in PHP code or as iamgeS.

***Design patterns***

Those are patterns of solution for specific problems that developers would have already encountered in the profesionnal world when coding solutions.

The usine class for instance is there to instanciante classes so you do not have to do it by yourself.

Moreover, to impose a specific behaviour to your object, you have to force it to implement certain methods.

The observer pattern allow to link some objects to « listeners » which are themself charged to notify the objects to which they are attached

The strategy pattern is there to delocalise the algorithmic of a method in order to rende rit re usable. Its there to avoid duplication of that algorithm.

The singleton pattern allow to instaniate a class only one time.

The injection pattern of dependencies is there to make classes the most independant possible.

***Generators***

Generators are a simple way to create iterators

An interator is a kind of cursor for classes wich contains elements such as lists.

Every function thas has the key word « yield » is automatically considered as a generator

A generator can return a simple value but also a key which will be associated to it

To return a reference via a yield , you must place an « & » before the name of the generators

The « send » method allow to create « co routines » which consist in consuming value and not returning them.

***Closures or « anonymous functions »***

One way to decalre a closure would be the following one :

<?php

function()

{

echo 'Hello world !';

};

Those functions are mainly used as « reminder functions » Which are functions demanded by other functions to execute specific tasks.

It is possible to link a closure to an object or a class thank to the « bindTo » attribute.

***How does an application/website work in php ?***

When the internet user access the website, a php file is executed on the server. In the case here, the file will be executed every time the visitor wants to access a page of the domain.

Launching of the application : there is a php file which will be executed First part of the file will be to deal with autoloader and the second will be to launch the application , this type of file is also known as a bootstrap.

Loading of the client request : this step consist in analysing the request sent by the client. Its at that point that the app will fetch the variables transmitted via form or URL ( the well known GET and POST variables)

Execution of the page : How does the app know which page to execute ? which action the visitor wants to execute ?. The action is determined by a « routeur », by analysing the URL, the routeur is capable of knowing what the visitor wants. For instance, if the URL entered by the visitor is « monsite.com/news-12.html », then the routeur will know that the visitor want the news that has an ID of 12 in the database. The routeur will thus return this action to the application which will execute it.

Response to the client : after having executed the desired action, the app will display the hall thing and the execution of the script will be over. This is when the page will be sent to the user.

***Autoload and PSR-o set of rules***

This set of rules is there to ensure that every developers that follow those rules can use the same autoload.

Classes and spaces of names entirely qualified must dispose of the folllowing structure : **\<Nom du Vendor>\(<Espace de noms>\)\*<Nom de la Classe>**.

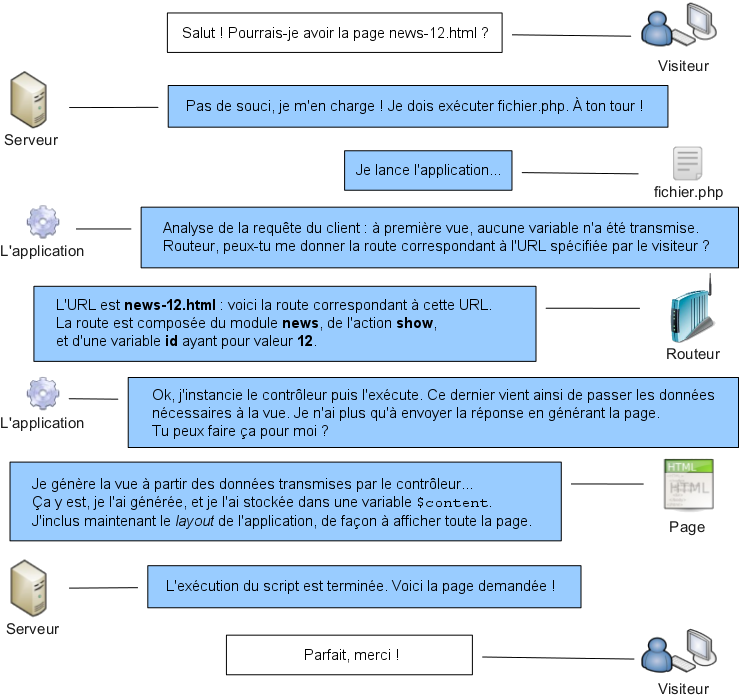
Every spaces of name should have a space of racine name ( name of the vendor)

Every space of names can have as much sub sbace of name that it wishes.

Every seperator of an space of names is converted in DIRECTORY\_SEPARATOR.

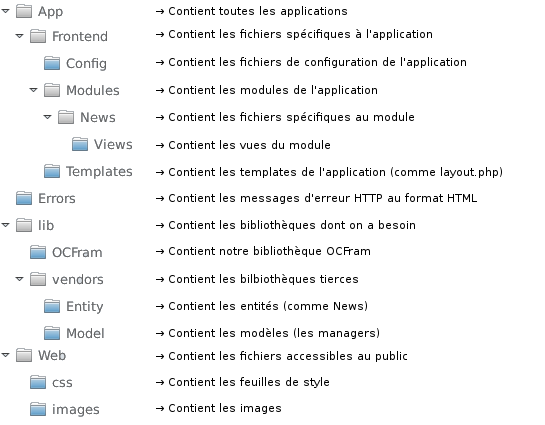
Classes and space of names completely qualified are suffixed with .php when there is a load from the system of file.

Alphabetic character in the name of the vendors, space of names and class names can contain any kind of combinaison of maj and min.



***Exemple of how a full application should be organized and structured :***

<http://www.victorthuillier.com/oc/poo/tp_app/?f=Partie+III>



Concerning the layout for the Front End part, you should use the pack « envision ».

Regarding the **instanceof operator :** use to know if an object is an instance of this or that class.

This operator is used within a condition

For instance :*if* $myObject is *an instance of* MyClass, *then* ….

This operator is used to check the nature of the class from which the object teste dis an instance of it.

This operator allow to check if a certain object is indeed an instance of a child class from this or that class.

It also allow to check if a certain object is an instance of a class that implement this or that interface.

**List of frameworks that are object oriented**

1) Symfony is complete and powerful. It is used by openClassRoom and in the profesional world.

2) Zend framework : often compared to a factory since its very powerful. Massively used in the profesional world because this framework adapt according to the user’s way of coding.

3) code lg nitter : a lot more lighter than the two above but less complete. You will have to fill the library up by yourself.

4) Cake PHP : <https://book.cakephp.org/2/en/index.html>