PHP - OBJET

https://openclassrooms.com/courses/concevez-votre-site-web-avec-php-et-mysql/introduction-a-php

http://php.net/manual/fr/langref.php

Open Class Room

SOMMAIRE

Edition : mai 2016 – mars 2017

PHP OBJET – PARTIE 1 – BASES

https://openclassrooms.com/courses/programmez-en-oriente-objet-en-php/introduction-a-la-poo

Vocabulaire de Programmation orientée objet

Intérêt de la POO

Le code, sans POO, devient vite compliqué et difficile à faire évoluer.

L'intérêt de la POO est d'avoir un code plus facile à faire évoluer

La POO permet de créer des « boites noires », les classes, qui masquent la complexité en donnant des outils simples à utiliser.

Pour créer un objet, il faut d'abord créer une classe : c'est le plan qui permet de créer l'objet.

Classe

Une classe, c'est un type, comme un entier, un réel, un caractère, une string ou un booléen.

En général, une classe correspond à l'équivalent d'un tableau associatif : elle contient plus couples de clé-valeur. Les différentes clés sont appelées « attribut ».

En plus, on associe des fonctions à une classe : on les appelle alors « méthode ».

Une classe est un plan pour un objet.

Objet

Un objet c'est une variable de type Classe.

Quand on crée un objet avec des valeurs pour les couples clé-valeur (pour les attributs), on dit qu'on instancie un objet. Ca passe par la commande « new ».

Les objets sont ce qu'on appelle les instances de la classe.

Instance : une réalisation concrète du plan : un objet de la classe

Instancier : créer un objet à partir d'une classe.

Méthode

Les méthodes sont des fonctions qui sont attachées à une classe.

Elle ne se sont utilisables que par les objets de la classe.

On écrit : objet->methode() pour appeler la méthode pour l'objet en question : c'est comme si on avait passé l'objet en paramètre de la méthode.

Encapsulation

Fait de cacher le contenu technique d'un objet pour ne laisser que l'accès aux fonctionnalités nécessaires pour l'utilisateur (le programme qui utilise l'objet).

Polymorphisme

Fait que, pour une même ligne de code, le comportement soit différent.

Le principe est que, en fonction de l'objet instancié, c'est une méthode ou une autre qui est utilisée.

C'est un mécanisme un peu compliqué qui passe par la notion d'interface ou de classe abstraite mais qui est centrale en programmation objet pour avoir un code facilement adaptable aux changements de cahier des charges.

Exceptions

En cas d'erreur, en programmation objet on passe par des objets de classe Exception.

Ca se fait avec un « try » « catch »

try : on essaie d'exécuter une suite d'instruction

catch : si la suite d'instructions exécutée à générer une erreur sous la forme d'une exception, on passe dans le bloc catch

Le catch précise le nom de l'objet exception qu'on va traiter.

On peut alors accéder à des informations par la méthode getMessage() par exemple.

Guides de style

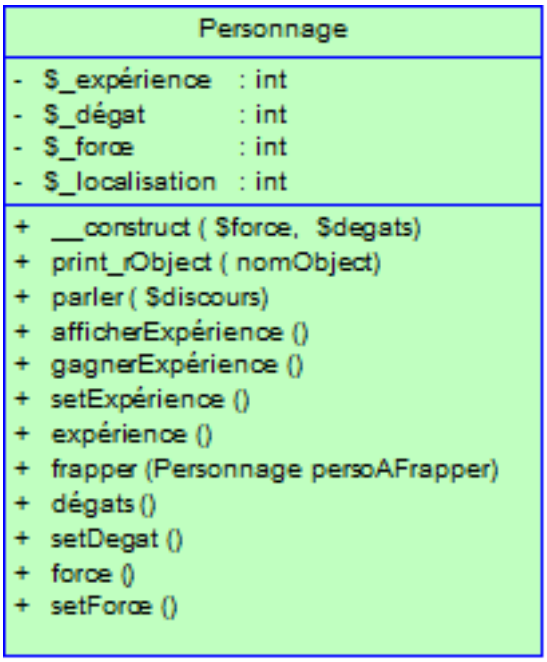
https://fr.wikipedia.org/wiki/CamelCase

https://eilgin.github.io/php-the-right-way/#code_style_guide

notation PEAR

Classe et objet - exemple

Créer une classe : le plan UML



Les attributs privés (symbole – en UML) sont écrit $_nomAttribut.

Les méthodes donc publiques.

Le constructeur s'appelle : __construct( ) : deux _ devant le nom.

Le « this » dans les méthodes permet d'utiliser l'objet en cours.

Les fonctions « set », appelées « setter » permettent de donner une valeur à un attribut : ici les fonctions setDegats(), setForce(), setExperience().

Les fonctions « nomAttribut», appelées « getter » permettent de récupérer la valeur d'un attribut : ici les fonctions degats(), force() et experience().

Créer une classe : le code

```
<?php
class Personnage
{
  private $_force = 50; // La force du personnage, par défaut 50
  private $_localisation; // Sa localisation
  private $_experience;   // Son expérience
  private $_degats;       // Ses dégâts

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

  }

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

  }

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

  }
}
```

Créer une classe : les bons usages

Les 3 règles fondamentales

- Bien nommer ses variables et ses fonctions.
- Bien indenter le programme.
- Bien commenter le programme.

Elément de notation PEAR :

- nom des **classes** commence par une **majuscule**
- **underscore** précédant les noms d'éléments **privés**
- **Constantes** en **majuscules** avec des **_ pour séparer les parties du nom**.
- **double underscore** précédent les noms d'éléments **magiques** (constructeur, etc.)
- **pas d'underscore** pour les élément **protected**.

Utiliser une classe

fichier index

C'est la page HTML : il « require » le fichier contenant la classe puis inclut le fichier avec les usages.

On choisit d'organiser ça comme ça pour séparer les fichiers.

```
<?php
require 'Personnage.php'; // On utilise la classe.
include 'usages.php';     // On inclut les usages.
?>
```

require, include, require_once, include_once

http://php.net/manual/fr/function.require.php

Require produit une **erreur** si le fichier requis n'existe pas. Le require est orienté « inclusion de morceaux PHP » : si on ne lest a pas, on peut considérer que le programme ne pourra pas fonctionner.

Include produit un **warning** si le fichier requis n'existe pas. L'include est orienté « inclusion de morceaux HTML » : si on ne les a pas, alors on peut considérer que ce n'est pas grave.

Si on recharge une classe déjà chargée, ça va planter !

Le _once permet d'éviter de recharger un fichier déjà inclus. C'est utile pour les classes et les fonctions.

En final, on tend à utiliser **include_once** et **require_once** tout le temps.

usages.php

Le fichier d'usages permet de créer les objets et de les utiliser.

Le mot clé « new » permet de créer un nouvel objet. C'est la méthode __construct() qui est appelée à cette occasion.

A partir d'un objet, on peut utiliser les méthodes définies dans la classe : print_rObject(), parler(), gagnerExperience(), etc.

```
<?php
// création de $perso1
$perso1 = new Personnage(50,0);
$perso1->print_rObject('$perso_1');
$perso1->parler('Bonjour, je m\'appelle perso_1');

echo '<br>Expérience de perso_1 apprès gain d\'expérience : ';
$perso1->gagnerExperience();   // On gagne de l'expérience.
$perso1->afficherExperience(); // On affiche la nouvelle valeur
de l'attribut.

// création de $perso2
$perso2 = new Personnage(50,0);
$perso2->print_rObject('$perso_2');
$perso2->parler('Bonjour, je m\'appelle perso_2');

// $perso1 frappe $perso2
$perso1->frapper($perso2);
echo 'Dégats de perso_2 après le coup: '. $perso2-
>degats().'</br>';
```

Test

PHP-06-OBJET-01-CLASSE-PERSONNAGE

indexPersonnage.php : test de base

A noter :

- ***trigger_error***
  https://www.php.net/manual/fr/function.trigger-error.php
- ***E_USER_WARNING***
  https://www.php.net/manual/fr/function.set-error-handler.php

Auto-chargement des classes : spl_autoload_register et pile d'autoload

https://www.php.net/manual/fr/language.oop5.autoload.php

Problème : charger plusieurs fois la même classe

Si on require deux fois la même classe, ça va générer une erreur.

Pour éviter ça, on peut utiliser le require_once.

En général, on met une classe par fichier : ça oblige donc à faire un require_once par classe : c'est lourd !

Pour éviter ça, on peut aussi utiliser la fonction spl_autoload_register qui gère un chargement automatique des classes.

spl_autoload_register et fonction requireClass

La fonction spl_autoload_register permet de remplacer la méthode __autoload() d'une classe.

La méthode __autoload() est une méthode « magique » dont le nom est prédéfinie pour toute classe et qui est appelée automatiquement à chaque instanciation d'un objet d'une classe. En général, on y mettait un include_once ou un require_once du fichier de la classe.

Cette méthode est aujourd'hui obsolète.

spl_autoload_register() est une sorte d'__autoload() généralisé : il permet d'exécuter une méthode à chaque instanciation d'une classe : on parle de pile d'autolaod().

Ainsi, on n'a plus besoin de charger la classe.

```php
<?php
// On crée une pile d'autoload avec le spl_autoload

spl_autoload_register('requireClass');

// On crée une fonction pour faire les require de Classe

function requireClass($classe){
  require $classe . '.php';
}
include 'usages.php';     // On inclut les usages.
?>
```

Test

PHP_TESTPOO_TD02_CLASSE_PERSONNAGE _AUTOLOAD

A noter :

```php
static private $_nbObjects=0;     // attribut de classe
const FORCE_PETITE = 20;          // constante de classe

static public function nbObjects(){ // méthode de classe
self::$_nbObjects;                // accès à un attribut de classe
self::FORCE_PETITE                // accès à un attribut de classe
Personnage::FORCE_MOYENNE         // accès à un attribut de classe
public function __toString()      // méthode magique
echo($objet);                     // <=> echo $objet->__toString()
```

Attribut et fonction de classe : static, self, ::

Principe

Un attribut de classe, ou attribut static, est un attribut qui ne concerne pas un objet en particulier mais toute la classe.

Une fonction de classe, ou fonction static est une fonction qui ne concerne pas un objet en particulier mais toute la classe.

Exemple

On compte le nombre d'instance de la classe Personnage.

On se dote d'une variable de classe : $_nbObjects (variable privée accessible via son getter).

On se dote d'un getter : function nbObjects()

On incrémente $_nbObjects dans le constructeur (à chaque nouvelle instance).

Noter l'usage du mot clé « self ».

```
class Personnage
{
  // $_nombre : variable de classe : static, initialisée à 0
  static private $nbObjects=0;

  etc.

  // function static
  static public function nbObjects(){
     // on accede à la variable static par self et ::
     // self represente la classe, on pourrait metre Personnage
     return self::$nbObjects;
  }

  public function __construct($force, $degats) {
     etc.

     // on compte le nombre d'instances de la classe
     self::$nbObjects++;
}
```

Utilisation d'une fonction statique -- Class ::methode() ou objet->methode()

```
echo Personnage::nbObjects().' personnages instanciés';
```

Personnage:: pour accéder à une fonction statique.

Remarque : on peut aussi passer par un objet pour accéder à la méthode static :

```
echo $perso1->getCompteurCoups();
```

Constante de classe – const <=> public static

Principe : const MA_CONSTANTE = 20 ;

On peut déclarer des constantes (attributs avec une valeur qui ne change pas) dans la classe.

Ces attributs sont toujours public et static. Le mot clé const remplace ces deux attributs.

On les écrit en majuscules avec des _ pour séparer les parties du nom.

Les constantes participent à l'encapsulation du code : l'utilisateur de la classe sait qu'il existe une FORCE_PETITE mais ne connaît pas forcément sa valeur. Ca rend le code plus clair et plus facile à maintenir.

Exemple

```
class Personnage {
  private $_force;

  // Déclarations des constantes en rapport avec la force.
  const FORCE_PETITE = 20;
  const FORCE_MOYENNE = 50;
  const FORCE_GRANDE = 80;

  public function __construct($force){
    // le setter vérifie l'intégrité des données fournies
    // On vérifie que $force vaut « FORCE_PETITE »,
    // ou « FORCE_MOYENNE », ou « FORCE_GRANDE ».
    if (in_array($force, [self::FORCE_PETITE,
            self::FORCE_MOYENNE, self::FORCE_GRANDE])) {
      $this->_force = $force;
    }
    else{ // sinon on donne une petite force par défaut !
      $this->_force = self::FORCE_PETITE;
    }
}
```

Utilisation de constantes : Classe ::CONSTANTE

```
$perso = new Personnage(Personnage::FORCE_MOYENNE);
```

Première méthode magique : __toString()

Principe

http://php.net/manual/fr/language.oop5.magic.php#object.tostring

__toString détermine comment l'objet doit réagir lorsqu'il est traité comme une chaîne de caractères.

Ainsi, __toString retourne une chaine de caractères.

Et par exemple, en faisant un echo($unObjet), on affiche la chaine de caractère retournée.

Dans la classe :

```
    // méthode magique,
    // on return une chaine qui sera traitée par un echo
    public function __toString()
    {
        return
          'Force = '.$this->_force.
          ' - Localisation = '.$this->_localisation.
          ' - Experience = '.$this->_experience.
          ' - Degats = '.$this->_degats.
          '<br>';
    }
```

Dans le programme qui utilise la classe :

```
  $perso1 = new Personnage(Personnage::FORCE_MOYENNE,0);
  echo($perso1); // méthode magique
```

POO – Synthèse de syntaxe

Création de classe et création d'objet (instanciation d'objet)

```
< ?php
class Membre {
    /* ici on décrira la class */
}
$robert = new Membre() ;
$li = new Membre() ;

?>
```

$robert ne s'appellera plus « variable » mais « objet ».

La class Membre contiendra par exemple le nom, le pseudo, le mot de passe, et des fonctions permettant de gérer le membre, par exemple : modifier le mot de passe, changer ses données personnelles, etc.

On peut créer autant d'objets qu'on veut.

Organisation du code : séparation des classes et des objets

On crée un fichier séparé avec la déclaration des classes.

Pour utiliser ce fichier on fait un « require » ou un « require_once » pour qu'il ne soit inclus 2 fois.

On peut aussi utiliser le mécanisme d'auto-load.

Définition d'une classe

Une classe contient des attributs et des méthodes.

Création des variables

```
< ?php
class Membre {
    private $_pseudo ;
    private $_email ;
    private $_actif ;
}
?>
```

private : signifie que la variable n'est pas accessible !!! A quoi servent-elle alors ?

Elles vont servir pour les fonctions qui elles seront accessibles. C'est le principe de la POO : donner des outils simples à utiliser, en l'occurrence les méthodes, en cachant la complexité.

Création des méthodes

```
< ?php
class Membre {
	private $_pseudo ;
	private $_email ;
	private $_actif ;

	public function pseudo(){ // getter pour récupérer l'attribut
		return $this->_pseudo ;
	}
	public function setPseudo($pseudo){ // setter pour mettre à jour l'attribut
		$this->_pseudo=$pseudo ;
}
?>
```

this-> : this c'est l'objet qui utilise la classe. La -> permet d'accéder à une variable de la classe.

La fonction renvoie le pseudo. Pourquoi avoir récupérer directement la variable ? Ca permet, par exemple, de rajouter du code, de faire des vérifications par exemple, ou des transformations, etc.

```
	public function setPseudo($pseudo){
		if (!empty($nouveauPseudo) AND strlen($nouveauPseudo) < 15) {
			$this->_pseudo=$pseudo ;
		}
}
```

Ici on vérifie que le pseudo n'est pas vide et que la taille est <15.

A noter que le $pseudo passé en paramètre dans setPseudo n'a rien à voir avec le $pseudo de la classe. On pourrait l'appeler $varPseudo pour bien distinguer. Mais ce n'est pas l'usage et ça rend le code simple et très lisible, quand on a l'habitude !

Utilisation des fonctions

Dans la page qui utilise la class, ou pourra écrire :

```
< ?php
include_once('Membre.class.php')

$robert = new Membre() ;
$robert->setPseudo(« Robert ») ;
echo &robert->getPseudo() ;
?>
```

Terminologie des fonctions

Les fonctions get sont appelées : getter ou accesseur

Les fonctions set sont appelées : setter ou mutateur

Dans la page qui utilise la class, ou pourra écrire :

Autres fonctions

```
        public function bannir(){
                $this->actif=false ;
        }
        public function debannir(){
                $this->actif=true;
    }
    publid function getActif(){
                return $this->actif ;
    }
```

Utilisation des fonctions

```
$robert = new Membre() ;
$robert->setPseudo(« Robert ») ;
echo &robert->getPseudo() ;
$robert->bannir() ;
```

Avec la POO, le code est beaucoup plus lisible et donc plus facile à mettre à jour.

Les constructeurs

Le constructeur est une fonction qui permet d'instancier un objet.

Par défaut, c'est le nom de la classe : $robert=new Membre() ;

On peut créer ses propres constructeurs, en leur faisant faire des actions particulières.

Par exemple, on veut un constructeur qui donne tout de suite la valeur du pseudo au membre et qui le positionne comme membre actif.

```
        public function __construct ($pseudo){ /* deux _ et construct */
                $this->pseudo=$pseudo ;
                $this->actif=true ;
        }
```

Quand on code un constructeur, il s'appelle toujours : __construct.

Garbage collector

Pour supprimer un objet, il suffit que sa référence (le nom de la variable objet) soit setté à NULL ;

$objet=NULL ;

Le garbage collector (ramasse-miette) gère le fait de libérer la mémoire mais on ne sait pas quand c'est fait.

On peut forcer l'appel avec la fonction : gc_collect_cycles();

Utilisation des fonctions

```
$robert = new Membre(« Robert ») ; /* on crée l'objet directement avec un pseudo */
echo &robert->getPseudo() ;
$robert->bannir() ;
```

PHP OBJET – PARTIE 2 – POO ET BD

Présentation des notions

Les classes métiers

Les classes « métier » sont les classes du modèle.

Les classes « métier » contiennent les objets de base de l'application (dans notre exemple, la classe personnage) correspondent à une table de la BD.

Dans le TP Artiste, le modèle correspond aux 3 tables gérées (Œuvres et Expositions. D'un point de vue métier, les œuvres exposées sont des composants des expositions).

Une classe « métier » contient un attribut « _id » et les autres attributs de la table.

Chaque objet d'une classe métier se retrouve en tant que une ligne (un tuple) dans une table de la BD.

Instanciation d'un objet d'une classe métier : l'hydratation

Un objet d'une classe métier (un personnage dans notre exemple) va être instancié à partir de la BD.

Cette instanciation est appelée : hydratation.

On part du principe que ce qui est dans la BD est « propre ». Donc quand on hydrate, on n'a pas de problème de vérification.

L'objectif est que le processus d'hydratation soit le plus générique possible, quelle que soit la classe.

Cela concernera donc en final le constructeur des classes métier.

Bonnes pratiques : nom de Classe = nom de Table

La classe « Personnage » est écrite au singulier. Il s'agit du modèle d'un personnage.

En BD, on a intérêt à appliquer le même principe pour se faciliter la tâche dans les relations entre la BD et le code serveur. Donc la table « Personnage » sera aussi au singulier.

Manager d'une classe métier

Les relations avec la BD d'une classe métier ne sont pas traitées dans la classe elle-même mais dans une classe à part : la classe « manager ».

Ca permet de séparer la partie BD de la partir calcul, en conformité avec l'architecture MVC.

La classe « manager » appartient au modèle.

Instanciation d'un objet d'une classe métier - Hydratation

Rappels PHP

Classe PDO

constructeur pour la connexion à la BD

pdoStat = pdo->exec($reqSQL) // suivi de rien

pdoStat = pdo->query($reqSQL) // suivi d'un fetch

pdoStat = pdo->prepare ($reqSQL..) // suivi d'un bind et d'un execute

Classe PDOStatement

pdoStat->bindValue // suivi d'un execute

pdoStat->execute // suivi d'un fetch

pdoStat->fetch

pdoStat->fetchAll

Notion d'hydratation

L'hydratation d'un objet (on dit aussi « hydrater un objet ») consiste à donner des valeurs à ses attributs à partir des informations récupérées dans la BD.

Version 1 : récupération des données SANS POO : avec FETCH_ASSOC

1) Une requête à la BD
2) Chaque ligne du résultat de la requête est un tableau correspondant à un personnage
3) On peut travailler sur ce tableau

```
// $pdo est un objet PDO : la connexion à la BD
$reqSQL='
     SELECT id, nom, forcePerso, degats, niveau, experience
     FROM personnage
';

$pdoStat = $pdo->query($reqSQL);

// Chaque entrée sera récupérée et placée dans un array.
while ($personnage = $pdoStat->fetch(PDO::FETCH_ASSOC)){
   echo $personnage['nom'], ' a ',
      $personnage['degats'], ' de dégâts -- id :  ',
      $personnage['id'], '<br>';
}
```

- ***Test***

PHP-06-OBJET-03-CLASSE-PERSONNAGE-BD : INDEX SANS CLASSE

indexTestBD_sans_Classe.php

Version 2 : récupération avec POO : avec FETCH_ASSOC, sans hydratation

https://www.php.net/manual/fr/pdostatement.fetch.php

1) Une requête à la BD
2) Chaque ligne du résultat de la requête est tableau : $donnees.
3) On instancie un objet à partir de ce tableau : new Personnage($donnees);
4) On peut travailler sur cet objet

```
$pdo = connexionBD('projet_jeu');

$reqSQL='
     SELECT id, nom, forcePerso, degats, niveau, experience
     FROM personnage
';

$pdoStat = $pdo->query($reqSQL);

// Chaque entrée sera récupérée et placée dans un array.
while ($donnees = $pdoStat->fetch(PDO::FETCH_ASSOC)){

  // le constructeur appelle les setters qui font le travail
  $personnage = new Personnage($donnees);

  // on peut ensuite faire ce qu'on veut de l'objet :
  $personnage->afficher();
}
```

- ***Le constructeur***

La particularité du code, c'est le constructeur : le **new Personnage($donnees)**

$donnees est un tableau associatif dont les keys correspondent aux attributs de la BD.

Exemple de code :

```
public function __construct(array $donnees) {
     $this->_id=$donnee['id'];
     $this->_nom=$donnee['nom'];
     $this->_degats=$donnee['degats'];
   }
```

- ***Hydratation***

Avec cette technique :

1) Si un attribut change dans la BD, il faut mettre à jour le constructeur.

2) Dans toutes les classes, il y aura un constructeur différent.

Pour remédier à cela, on utilise la technique de l'hydratation ou l'utilisation de FETCH_CLASS.

- ***Test***

PHP-06-OBJET-03-CLASSE-PERSONNAGE-BD : INDEX AVEC CLASSE

indexTestBD_avec_Classe.php

Version 3 : récupération avec POO : avec FETCH_CLASS et hydratation

https://www.php.net/manual/fr/pdostatement.fetch.php

1) Une requête à la BD
2) On fait un setFetchMode en précisant FETCH_CLASS
3) Ainsi chaque fetch renvoit directement un objet de la classe. Il est instancié sans avoir à définir le constructeur : c'est un constructeur sans paramètre.
4) On peut travailler sur cet objet

```
$pdo = connexionBD('projet_jeu');

$reqSQL='
     SELECT id, nom, forcePerso, degats, niveau, experience
     FROM personnage
';

$pdoStat = $pdo->query($reqSQL);
$pdoStat->setFetchMode(PDO::FETCH_CLASS,'Personnage');

// Chaque entrée sera récupérée et placée dans un objet
while ($personnage = $pdoStat->fetch()){
  // on peut ensuite faire ce qu'on veut de l'objet :
  $personnage->afficher();
}
```

- ***Avantages***

Les données sont directement lues comme des objets.

Le constructeur est simplifié : on peut s'en passer. Il fonctionne sans argument et ça marche tout seul : les objets sont instanciés à chaque fetch.

Si les attributs changent dans la BD, ça ne change rien à l'hydratation.

On peut aussi faire un fetchAll :

```
$pdoStat = $pdo->query($reqSQL);
$pdoStat->setFetchMode(PDO::FETCH_CLASS,'Personnage');
$personnages = $pdoStat->fetchAll();
foreach ($personnages as $personnage){
   $personnage->afficher();
}
```

- ***Défaut ?***

Si on veut créer ensuite des objets avec un constructeur avec paramètres, on peut le faire en testant le nombre de paramètres avec func_num_args().

Exemple de code de constructeur :

```
public function __construct() {
     if (func_num_args() == 0) return;

     if (func_num_args() == 3){
       $id=func_get_arg(0);      // 1er argument
       $nom=func_get_arg(1);     // 2ème argument
       $degats=func_get_arg(2); // 3ème argument
       $this->id=$id;
       $this->nom=$nom;
       $this->degats=$degats;
     }
```

```
    }
```

➢ ***Test***

PHP-06-OBJET-03BIS-CLASSE-PERSONNAGE-BD : INDEX AVEC CLASSE

indexTestBD_avec_Classe.php

Version 4 : récupération avec POO : avec FETCH_ASSOC et hydratation

Objectifs

Dans le cas d'un FETCH_ASSOC sans hydratation, le code du constructeur est différent pour chaqe classe et va changer si les attributs changent.

Exemple de code :

```
public function __construct(array $donnees) {
    $this->_id=$donnee['id'];
    $this->_nom=$donnee['nom'];
    $this->_degats=$donnee['degats'];
  }
```

L'objectif est de rendre ce code générique, c'est-à-dire identique pour toutes les classes et pour toutes les évolutions des classes.

Principes techniques

Le constructeur, pour donner une valeur à un attribut, va utiliser le setter de l'attribut.

Il suffit donc de parcourir le tableau $donnees pour récupérer tous les attributs à partir des keys du tableau. Ensuite, on peut faire appel au setter correspondant en utilisant une variable qui contiendra le nom de la fonction (du setter). **Ceci est possible en php car le nom d'une fonction peut être mis dans une variable.**

Pour rendre le constructeur plus lisible, la fonction d'hydratation est appelée par le constructeur.

1er principe de l'hydratation : le constructeur fait appel à la fonction d'hydratation

On se dote d'un constructeur avec un (array $donnees) en paramètre qui correspond à un objet de la BD et de la classe correspondante, quel qu'il soit.

```
public function __construct(array $donnees) {
    $this->hydrate($donnees);
}
```

2ème principe de l'hydratation : une fonction hydrate générique

➢ ***Fonction hydrate non générale***

```
public function hydrate(array $donnees) {
  if (isset($donnees['id'])) $this->setId($donnees['id']);
  if (isset($donnees['nom'])) $this->setNom($donnees['nom']);
  // …
}
```

Le défaut de cette version est qu'il faut une fonction par classe à hydrater et qu'il faut mettre à jour la fonction si les attributs de la classe changent.

➢ ***Fonction hydrate générale***

Pour généraliser, on récupère chaque couple key-value du tableau de données, et à partir de chaque key **on fabrique le nom du setter.**

Reste ensuite à appeler le setter avec la value.

Ceci est possible en php car le nom d'une fonction peut être mis dans une variable.

```
public function hydrate(array $donnees) {
  foreach ($donnees as $key => $value) {
    // On fabrique le nom du setter correspondant à l'attribut :
    $setter = 'set'.ucfirst($key);
    // Si le setter correspondant existe :
    if (method_exists($this, $setter)) {
      // On appelle le setter = $setter contient son nom !!!
      $this->$setter($value);
    }
  }
}
```

3ème principe de l'hydratation : les setters de la classe métier

Les setter doivent avoir un nom standard : setAttribut.

On part du principe que ce qui est dans la BD est « propre ». Donc quand on hydrate, on n'a pas de problème de vérification.

Pour que les données qu'on enregistre dans la BD soient propres, les setters feront toutes les vérifications nécessaires.

```
class Personnage
{
  private $_id;
  private $_nom;
  private $_force;
  etc…

  /* getters compacts ! */
  public function id(){ return $this->_id;}
  public function nom(){ return $this->_nom;}
  public function force(){ return $this->_force;}

  /* setter avec verifications */
  public function setId($id){
    // Cas particuliers : on impose la valeur de id à 0
    if ($id < 0) or ((int)$id == 0) {
      $id=0;
    }

    // Cas general
    $this->_id = $id;
  }

  public function setNom($nom) {
    // Cas particuliers
    if (!is_string($nom)) {
      return; // si on n'a pas une chaîne, on arrête le setter
    }

    // Cas general
    $this->_nom = $nom;
  }
```

```
    public function setForce($force) {
      // Cas particuliers
      if ($force < 1 or $force > 100 or (int)$force == 0 ) {
        return;
      }

      // Cas general
      $this->_force = $force;
  }
```

Défaut

Le défaut de cette version est qu'elle oblige à créer des getters et des setters pour tous les attributs, ce qui n'est pas conforme au principe d'encapsulation et à un usage des classes avec des méthodes conçues comme des services.

C'est pourquoi la technique avec le FETCH_CLASS est plus propre.

Test

PHP_TESTPOO_TD03_BD_ET_CLASSE_PERSONNAGE

indexTestBD_POO_hydratation.php

Services rendues à partir d'une classe métier : les classes manager

Relation avec la BD : la classe Manager

Principes

Les relations avec la BD d'une classe métier ne sont pas traitées dans la classe elle-même mais dans une classe à part : une classe « manager ». Ca permet de séparer la partie BD de la partir calcul, en conformité avec l'architecture MVC.

La classe Manager s'occupera par exemple de :

- ajouter un objet dans la BD : INSERT
- supprimer un objet dans la BD : DELETE
- modifier un objet dans la BD : UPDATE
- récupérer un objet à partir de son id : SELECT … where _id =
- récupérer tous les objets d'une table : SELECT *

Cette classe n'a qu'un attribut : la BD elle-même.

Cette classe travaille avec la BD. Elle pourrait aussi gérer des fichiers XML, des fichiers texte, etc.

Le nom des méthodes publiques doit dont être choisi pour être générique.

Nom de la classe

nomDeLaTableManager

La classe manager s'occupe de la table « Personnage » correspondant à la classe « Personnage » (les deux noms au singuliers, par principe).

Structure de la classe PersonnageManager

Un attribut « bdd » qui est l'objet PDO de connexion à la BD.

```
<?php
class PersonnageManager
{
  private $_bdd; // Instance de PDO.

  public function __construct($pdo) {
    $this->_bdd = $bdd;
  }

  public function insert(Personnage $personnage){ … }
  public function select(){ … }
  public function count(){ … }

}
```

Fonctions insert() : prepare, bind, execute

```
  public function insertInto(Personnage $ personnage) {
    $reqSQL='
      INSERT INTO
      personnage(nom, forcePerso, degats, niveau, experience)
      VALUES(:nom, :force, :degats, :niveau, :experience)
```

```
        ';

        $requete = $this->_bdd->prepare($reqSQL);

        $requete->bindValue
           (':nom', $ personnage ->nom());
        $requete->bindValue
           (':force', $personnage->forcePerso(), PDO::PARAM_INT);
        $requete->bindValue
           (':degats', $personnage->degats(), PDO::PARAM_INT);
        $requete->bindValue
           (':niveau', $personnage->niveau(), PDO::PARAM_INT);
        $requete->bindValue(':experience',
           $ personnage->experience(),PDO::PARAM_INT);
        $requete->execute();
    }
```

Fonction select() : query, fetch

```
public function select(){
    $personnages = [];
    $reqSQL='
       SELECT id, nom, force, degats, niveau, experience
       FROM personnage
       ORDER BY nom'
    ;

    $pdoStat = $this->_pdo->query($reqSQL);

    while ($donnees = $pdoStat ->fetch(PDO::FETCH_ASSOC)) {
      $personnages[] = new Personnage($donnees);
    }

    return $personnages;
  }
```

Fonction count() : query, fetchColumn

```
public function select(){
    $reqSQL='
       SELECT COUNT(*)
       FROM personnages
    ;
    return $this->_pdo->query($reqSQL)->fetchColumn();
}
```

Utilisation de PersonnageManager

```
<?php
// Autoload.
function chargerClasse($classname) { require $classname.'.php'; }
spl_autoload_register('chargerClasse');

// Connexion à la BD : création d'un pdo
include('connexionBD.php');
$pdo = connexionBD('projet_jeu');

// création d'un Manager
```

```
$personnageManager = new PersonnageManager($pdo);

echo '<h3> Affichage des données de la BD</h3>';
echo '<h3> Il y a ', $personnageManager->count(), ' personnage
dans BD</h3>';
$personnages=$personnageManager->select();
foreach($personnages as $personnage){
  $personnage->print_rObject();
}
?>
```

Test

PHP_TESTPOO_TD03_BD_ET_CLASSE_PERSONNAGE

indexTestBD_3_POO_manager.php

Autres fonctions

- ***update: prepare, bind, execute***

```
  public function update(Personnage $perso) {
    $reqSQL='
      UPDATE personnage
      SET   forcePerso = :forcePerso,
            degats = :degats,
            niveau = :niveau,
            experience = :experience
      WHERE id = :id
    ';

    $pdoStat = $this->_pdo->prepare($reqSQL);

    $pdoStat->bindValue(':forcePerso', $perso->forcePerso(),
                                                PDO::PARAM_INT);
    $pdoStat->bindValue(':degats', $perso->degats(),
  PDO::PARAM_INT);
    $pdoStat->bindValue(':niveau', $perso->niveau(),
  PDO::PARAM_INT);
    $pdoStat->bindValue(':experience', $perso->experience(),
                                                PDO::PARAM_INT);
    $pdoStat->bindValue(':id', $perso->id(),
  PDO::PARAM_INT);

    $pdoStat->execute();
  }
```

- ***delete() : exec***

```
public function delete(Personnage $personnage) {
    $reqSQL='
      DELETE FROM personnage
      WHERE id = '.$perso->id()
    ;
    $this->_pdo->exec($reqSQL);
  }
```

- ***selectId() : query, fetch***

```
    public function selectId($id) {
      $id = (int) $id;
      $reqSQL='
        SELECT id, nom, force, degats, niveau, experience
        FROM personnage
        WHERE id = '.$id
      ;

      $pdoStat = $this->_pdo->query($reqSQL);
      $donnees = $pdoStat ->fetch(PDO::FETCH_ASSOC);

      return new Personnage($donnees); // le personnage est hydraté
    }
```

TP - 1

Version 1

Cahier des charges

Chaque visiteur peut créer un personnage avec lequel il peut frapper d'autres personnages.

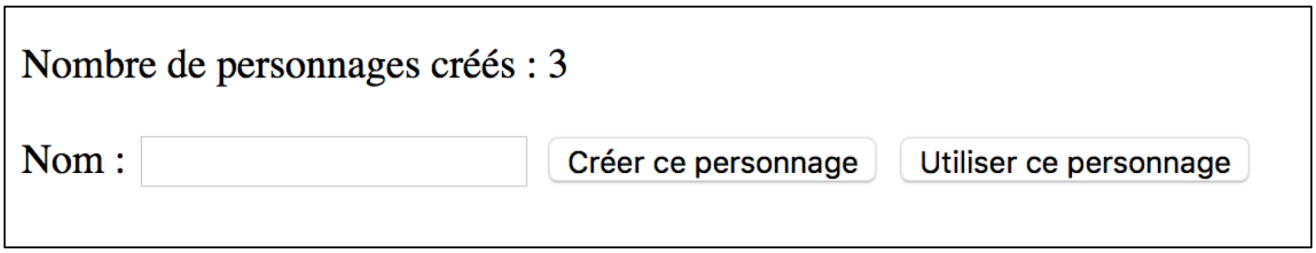
Un personnage a 2 caractéristiques : nom et dégâts. Il a aussi un identifiant.

Les dégâts vont de 0 à 100. Chaque coup porté augmente les dégâts de 5. Arrivé à 100, le personnage est tué et supprimé de la BD.

- ***Vue 1 :***

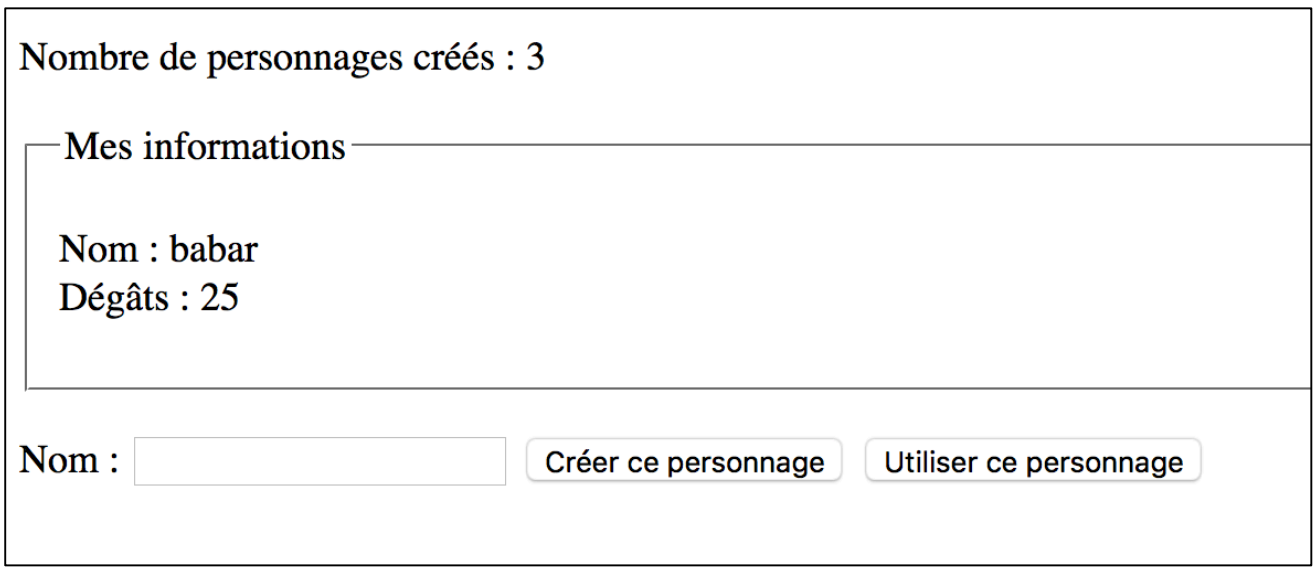
Page d'accueil :

- On affiche le nombre de personnages total
- On permet à l'utilisateur de créer ou de choisir un personnage



Une fois un personnage créé ou choisi :

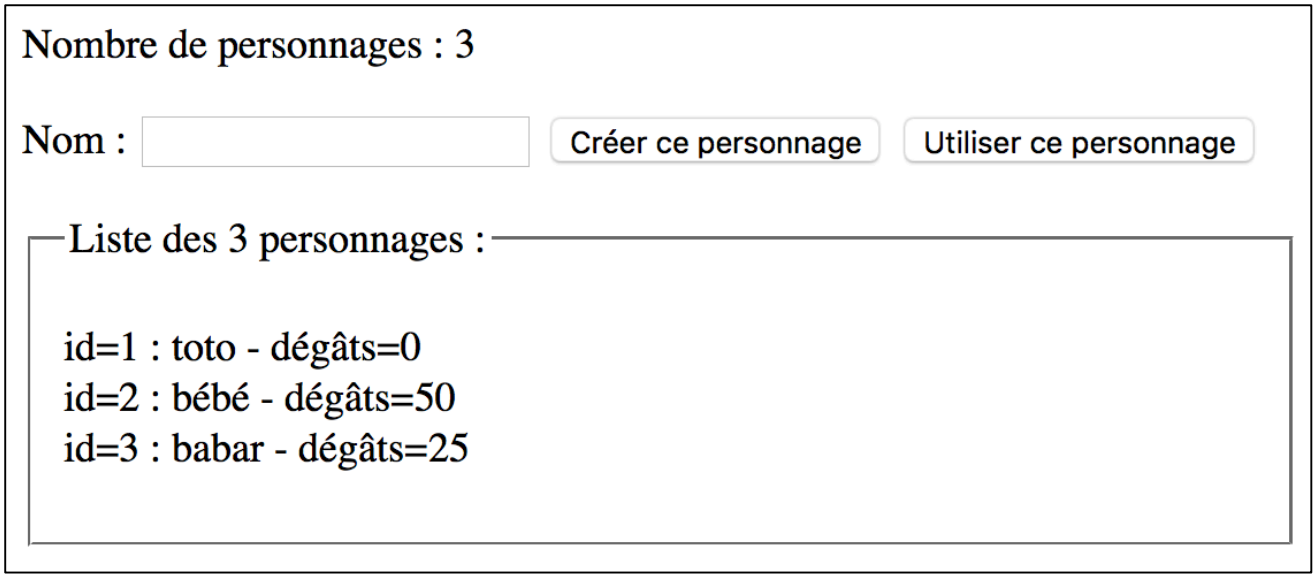
- On affiche toujours le nombre de personnages total
- On affiche les caractéristiques du personnage créé ou choisi



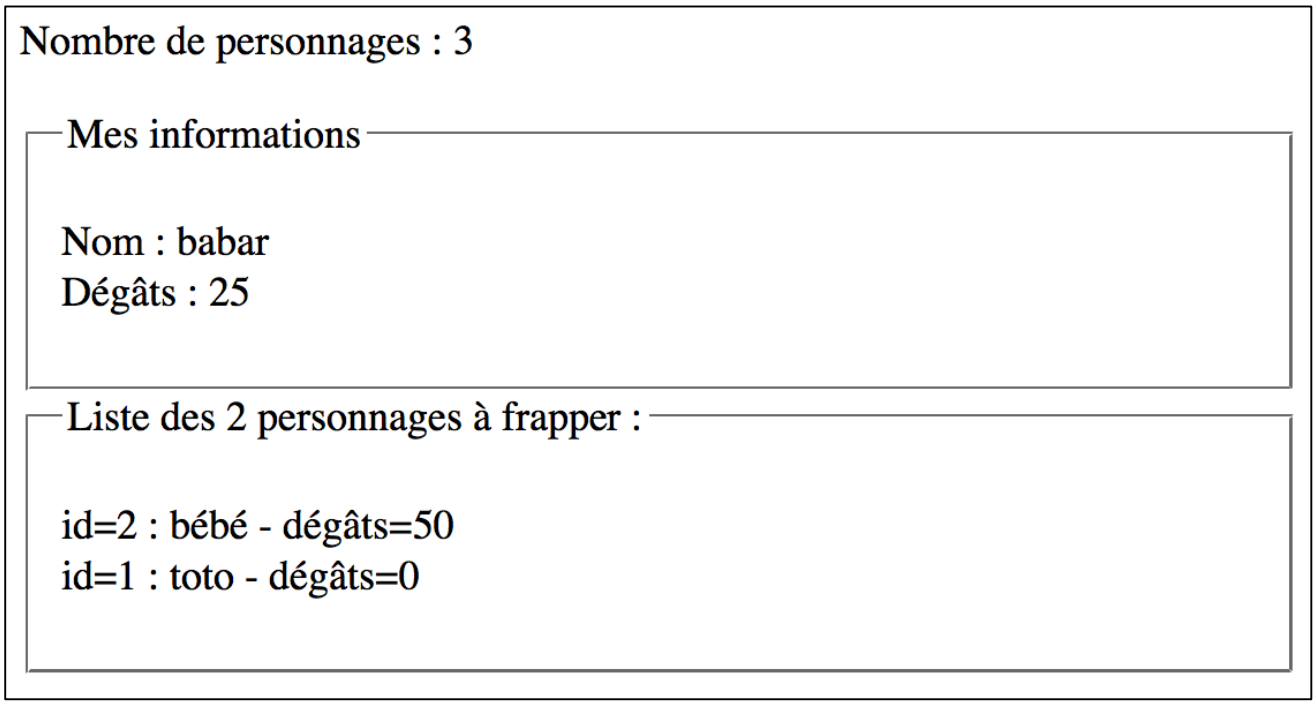
➢ ***Vue 2 :***

On ajoute l'affichage de la liste des personnages :

- Tous les personnages avec leur nombre, quand aucun personnage n'est sélectionné



- La liste des personnages contre lesquels il peut se battre quand un personnage est sélectionné



Méthode

S'appuyer sur les exemples vus en cours et chargés sur le site !

Conception de la BD : table Personnages

Personnages(id, nom, degats)

Le modèle : la classe Personnage

Personnage(id, nom, degats)

Getter, setter, hydrate

Le modèle : la classe PersonnageManager

Insert (nom) : pour ajouter un personnage

Exists(info) : permet, à partir d'une info, un id ou un nom, de savoir si un un personnage est dans la base ou pas.

SelectUn(info) : retourne un personnage à partir d'une info, un id ou un nom

Count() : permet de savoir le nombre de personnages dans la base

Le contrôleur 1

- ***Partie modèle :***

Autoload du chargement des classes

Connexion à la BD

- ***Partie contrôleur :***

Si on a choisi de créer un personnage :

- On crée l'objet
- On vérifie qu'il est valide (nom correct et qui n'existe pas déjà)
- Si ce n'est pas le cas, on crée un message d'erreur, sinon on l'insère dans la BD

Si on a choisi d'utiliser un personnage

- On vérifie qu'il existe dans la BD
- Si ce n'est pas le cas, on crée un message d'erreur, sinon on crée l'objet à partir du personnage de la BD

- ***Partie vue :***

Include de la vue

La vue 1 : HTML

On récupère les variables suivantes du contrôleur :

- $personnageManager : le manager de Personnage pour accéder à la fonction count()
- $perso : le personnage utilisé
- $message : le message à afficher

On affiche le nombre de personnages

Si nécessaire, on affiche le message reçu du contrôleur

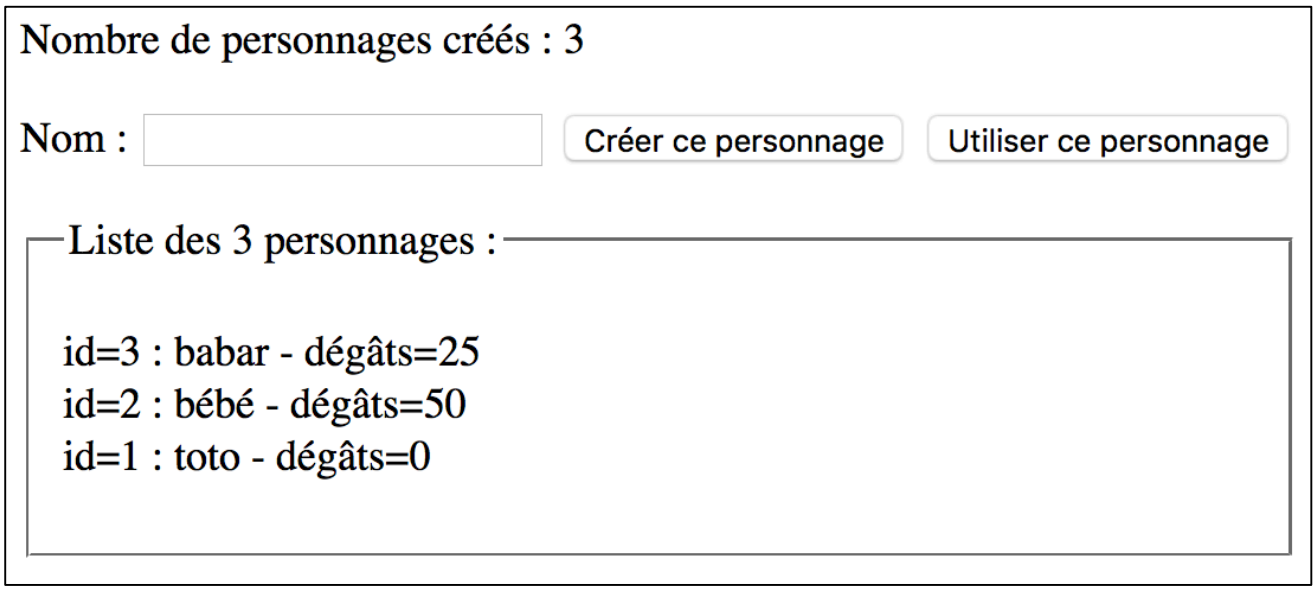
Si nécessaire, on affiche le joueur actuellement sélectionné

On affiche un formulaire pour permettre de créer ou de choisir un joueur

Cahier des charges – version 2

- ***Vue 1 :***

Page d'accueil : on affiche la liste des personnages en plus.

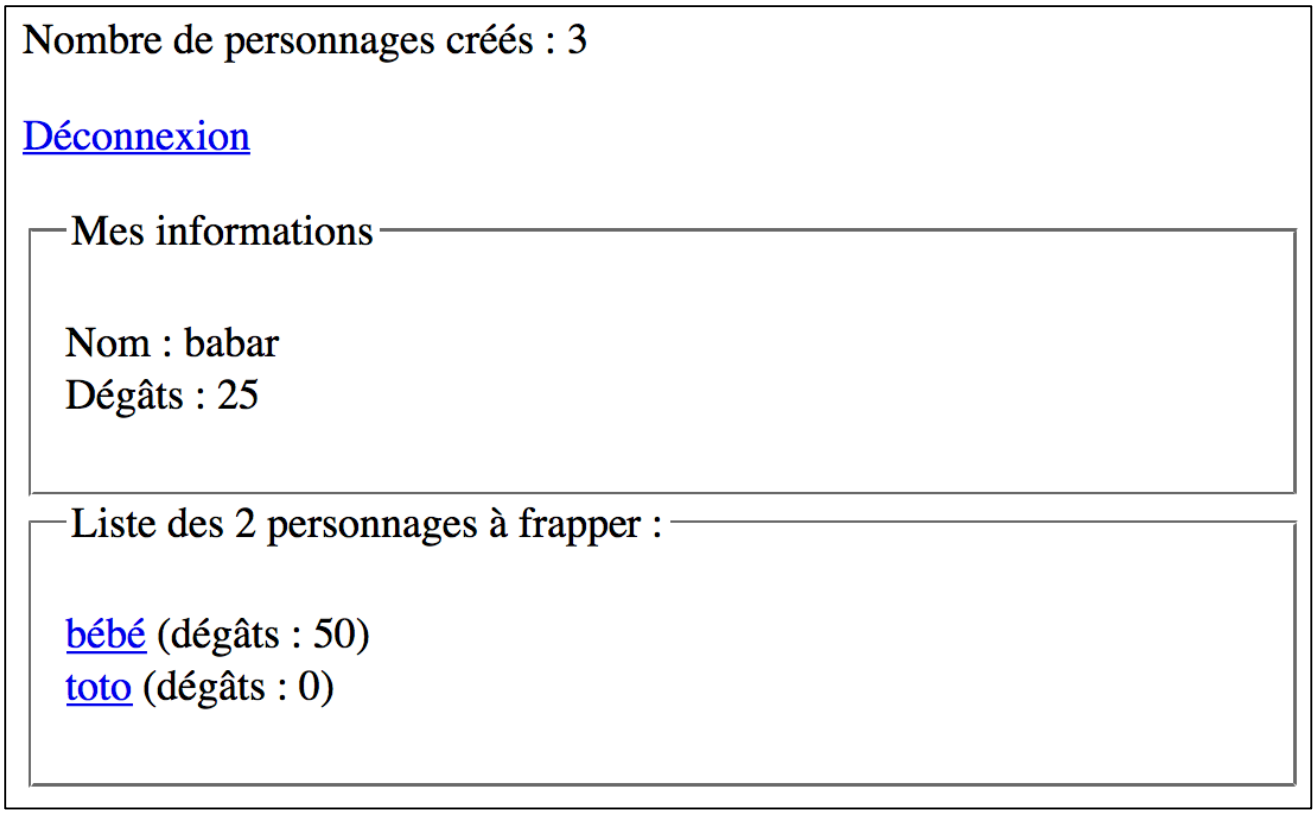


- ***Vue 2 :***

Quand un sélectionne un personnage, on affiche la liste des autres personnages et on permet de les sélectionner pour les frapper.

Un personnage frappé subit 5 points de dégâts. A 100 points de dégâts le personnage est supprimé.

La « déconnexion » permet de revenir sur la page d'accueil.



Cahier des charges – version 3

On peut imaginer toute sortes d'améliorations !

TP2 : Site artiste : classe métier, hydratation, classe manager et MVC

Application au projet Site Artiste : version non MVC -> à compléter

L'exemple donné avec le cours :

01-exemple-SiteArtiste-Objet-base-php-separe dans 02-Exemples-PHP-Objet-Site-Artiste.zip

propose une version du site artiste avec 3 tables : œuvres, exposition et œuvres exposées.

La table de œuvres est gérée avec une classe et une classe manager.

Le code est géré sans MVC mais avec séparation du PHP et du HTML.

Le code gère une classe métier, l'hydratation par FETCH_CLASS et une classe manager.

1) Installez et faites tourner l'exemple. Essayez de bien comprendre sa logique.
2) Ecrivez le code pour gérer les tables exposition et œuvres exposées avec des classes, mais toujours sans MVC.

Application au projet Site Artiste : version MVC -> à compléter

Idem que l'exercice précédent mais avec une architecture MVC.

L'exemple donné avec le cours :

02-exemple-SiteArtiste-Objet-base-MVC dans 02-Exemples-PHP-Objet-Site-Artiste.zip

propose une version du site artiste avec 3 tables : œuvres, exposition et œuvres exposées.

La table de œuvres est gérée avec une classe et une classe manager.

Le code est géré avec MVC.

Le code gère une classe métier, l'hydratation par FETCH_CLASS et une classe manager.

1) Installez et faites tourner l'exemple. Essayez de bien comprendre sa logique.
2) Ecrivez le code pour gérer les tables exposition et œuvres exposées avec des classes, et toujours avec MVC.

On reprend la version MVC du projet Site Artiste.

Le but est maintenant d'ajouter une structuration objet au MVC avec classe métier, hydratation par FETCH_CLASS et classe manager.

PHP OBJET – PARTIE 3 – HERITAGE

La suite du cours présente maintenant les éléments théorique et syntaxiques supplémentaires pour l'utilisation de la POO.

Héritage

est un - extends

Principe 1 : relation est un

Un magicien (classe fille) est un personnage (classe mère) : la classe Magicien va hériter des attributs et des méthodes publiques de la classe Personnage

Principe 2 : mise en commun d'attributs et de méthodes dans une classe mère

Le magicien et le guerrier ont des attributs (nom, dégâts, etc.) en commun. Ils ont aussi des méthodes en commun (frapper, recevoirDégats, getter et setter des attributs communs, etc.)

Ce qui est commun se retrouve dans une classe mère : la classe Personnage dont magicien et guerrier vont hériter.

Code php : class Magicien extends Personnage

```
class Personnage { // classe simple

}

// Magicien hérite des attributs et méthodes publiques de
Personnage.
class Magicien extends Personnage {

}
```

Compléments : protected, redéfinition, abstract, final

visibilité protected

Cette visibilité permet de rendre les attributs et les méthodes visibles dans les classes enfants.

Par défaut, on peut choisir de mettre tous ses attributs et méthodes protected pour permettre un héritage facile.

Les attribus protected s'écrivent sans _ au début

Redéfinir les méthodes de la classe mère dans la classe fille

On peut redéfinir une méthode de la classe mère dans la classe fille.

La visibilité de la méthode redéfinie doit être identique ou plus large que celle de la classe mère.

parent :: appeler une méthode redéfinie de la classe mère dans la classe fille

```
class Personnage { // classe simple
  public function gagnerExperience() {
    …
  }
}

// Magicien hérite des attributs et méthodes publiques de
Personnage.
class Magicien extends Personnage {
  public function gagnerExperience() {
    parent::gagnerExperience();
    …
  }
}
```

Classe abstraite

Une classe abstraite est une classe qui ne pourra jamais être instanciée.

```
abstract class Personnage {
  …
}
```

Méthode abstraite

Une méthode abstraite est une méthode qui n'a pas de corps, mais seulement une en-tête.

Ca permet de forcer la classe fille à écrire la méthode : l'idée est que chaque classe fille doit coder le problème spécifiquement.

Une classe qui contient une méthode abstraite est forcément abstraite.

```
abstract class Personnage {
  abstract public function frapper(Personnage $perso);
```

Classe finale

Une classe finale est une classe dont on ne peut pas hériter.

```
final class Magicien extends Personnage {
  …
}
```

Méthode finale

Une méthode finale est une méthode qu'on ne peut pas redéfinir.

```
abstract class Personnage {
  final public function recevoirDegats(){
    // Instructions.
  }
}
```

Opérateur de résolution de portée « :: » et méthodes « statics »

http://php.net/manual/fr/language.oop5.paamayim-nekudotayim.php

http://php.net/manual/fr/language.oop5.late-static-bindings.php

Principes

Quand on appelle une méthode de classe (static), on peut préciser si on veut celle de la classe en cours (celle de la méthode appelante, self ::), celle de la classe mère de la classe en cours (parent ::), ou celle de la classe de départ de l'appel (static ::).

C'est la même chose pour un attribut de classe.

parent ::

On a déjà vue le parent ::

Il permet de remonter à la classe mère, que ce soit pour une méthode d'objet, une méthode de classe (static) ou des constantes de classe.

```
class Mere{
  public parent function testStatic()
  {
    parent::testRedef();
  }
```

Si une classe Fille appelle le testStatic de la classe Mère (via la classe ou un objet), c'est le testRedef de la classe Grand-mère qui est appelé car il y a le parent::

Si la testRedef n'existe pas cette classe Grand-mère, ça plante même s'il existe dans la classe Mère ou Fille.

self ::

self :: permet d'appeler une méthode de la classe ou une constante de classe pour la classe en cours.

```
class Mere{
  public static function testStatic()
  {
    self::testRedef();
  }
```

Si une classe Fille appelle le testStatic de la classe Mère (via la classe ou un objet), c'est le testRedef de la classe Mère qui est appelée car il y a le self ::

Si la testRedef n'existe pas dans la classe Mère, ça plante, sauf si il existe dans une classe Grand-mère.

static ::

static : permet d'appeler une méthode de la classe ou une constante de classe pour la classe fille en cours (celle de l'objet à l'origine de l'appel de méthode).

```
class Mere{
  public static function testStatic()
  {
    static::testRedef();
  }
```

Si une classe Fille appelle le testStatic de la classe Mère (via la classe ou un objet), c'est le testRedef de la classe Fille qui est appelée car il y a le static::

Si la testRedef n'existe pas dans la classe Fille, ça plante, sauf si il existe dans la Mère.

parent ::

```
class Mere{
  public parent function testStatic()
  {
    parent::testRedef();
  }
```

Si une classe Fille appelle le testStatic de la classe Mère (via la classe ou un objet), c'est le testRedef de la classe Grand-mère qui est appelé car il y a le parent::

Si la testRedef n'existe pas cette classe Grand-mère, ça plante même s'il existe dans la classe Mère ou Fille.

instanceof

L'opérateur instanceof permet de vérifier si tel objet est une instance de telle classe.

instanceof Classe

```
if ($obj instanceof A) echo '$obj est une instance de A';
```

instance of $nomClasse

```
$A = 'A';
if ($obj instanceof $A) echo '$obj est une instance de'. $A;
```

instance of instance

```
$a = new A; $b = new A;
if ($a instanceof $b) echo '$a et $b sont des instances de la
même classe';
```

instance of ClasseMere

```
class A { }
class B extends A { }
$b = new B;
if ($b instanceof A) echo '$b est une instance de A';
```

instance of interface

```
interface iA { }
class A implements iA { }
$a = new A;
if ($a instanceof iA) echo 'Si iA est une instance de iA';
```

TP - 2

Cahier des charges – version 1

On va créer des personnages spécifiques dérivant du personnage de base.

Chaque personnage spécifique a des atouts sous forme d'entier.

Un magicien. Il peut endormir les autres personnages pendant un certain temps égale à 6 heures * ses atouts.

Un guerrier. Lorsqu'un coup lui est porté, il peut parer le coup en fonction de ses atouts.

On doit donc pouvoir savoir si un personnage est endormi ou pas (un personnage endormi ne peut pas être frappé).

On doit pouvoir connaître le délai de réveil d'un personnage personnage sous la forme « XX heures, YY minutes et ZZ secondes », qui s'affichera dans le cadre d'information du personnage s'il est endormi.

Les atouts fonctionnent ainsi :

- en fonction des dégâts, on a une valeurs d'atouts : si les dégâts sont compris entre 0 et 25, l'atout sera de 4, entre 26 et 50 : 3, 51 et 75 : 2, 65 et 90 : 1, 91 et plus : 0.
- En fonction des atouts, l'attaquant augmente la force de ses coups : les dégâts portés sont augmentés de $atouts et la durée d'endormissement est multipliée par $atouts+1 (plus un pour éviter le 0). Le défenseur diminue l'impact des coups à l'inverse (soustraction et division).

Conception de la BD : table Personnage

Personnage(<u>id</u>, <u>nom</u>, degats, type, atouts, dureeSommeil)

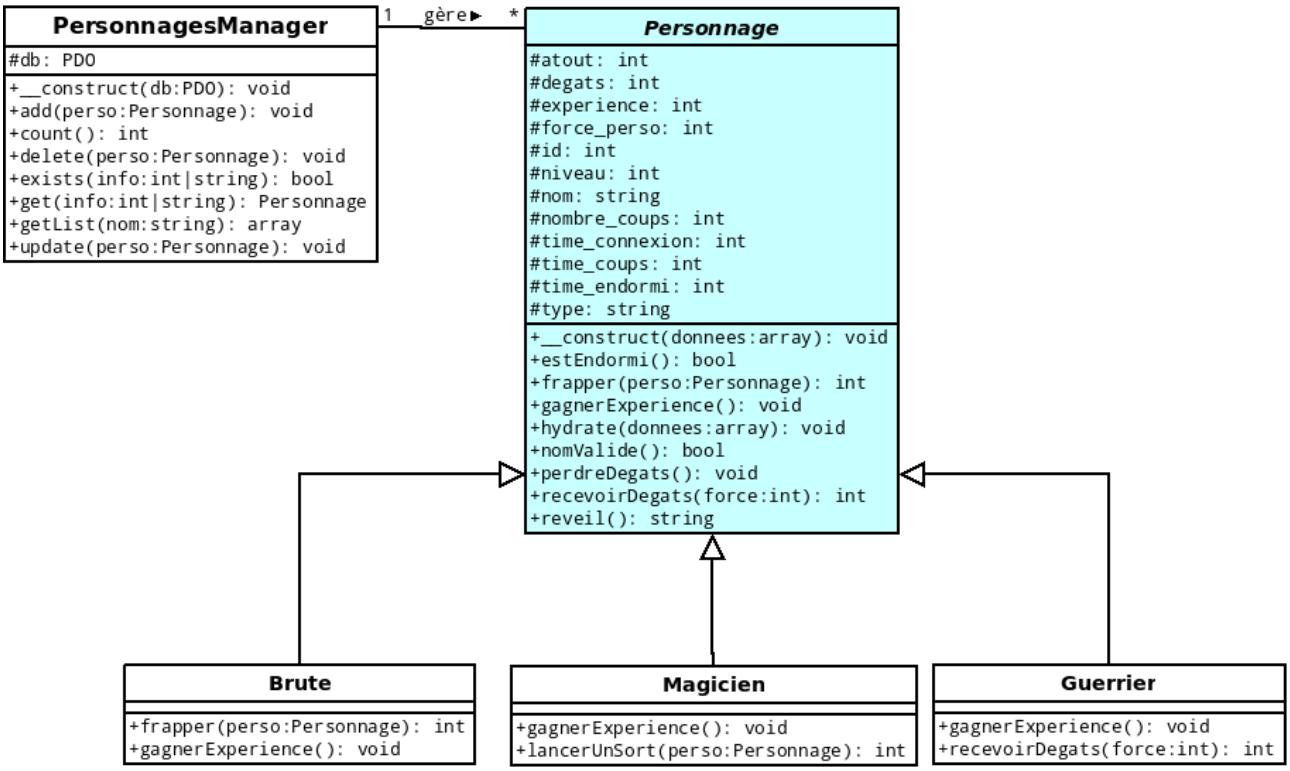
Cahier des charges – version 2

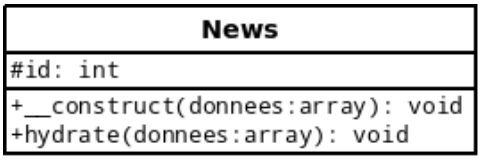
On peut imaginer toute sortes d'améliorations !

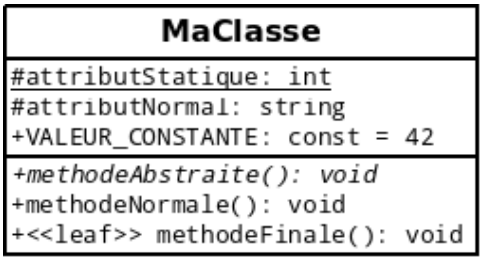
L'UML

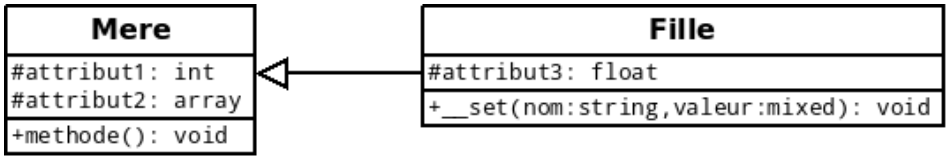
Diagramme de classe

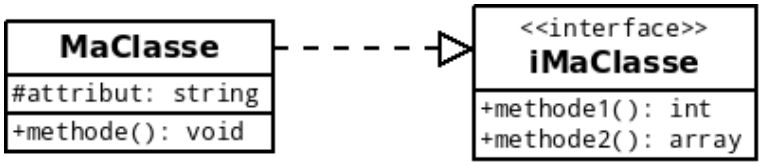
https://openclassrooms.com/courses/programmez-en-oriente-objet-en-php/uml-presentation-1-2

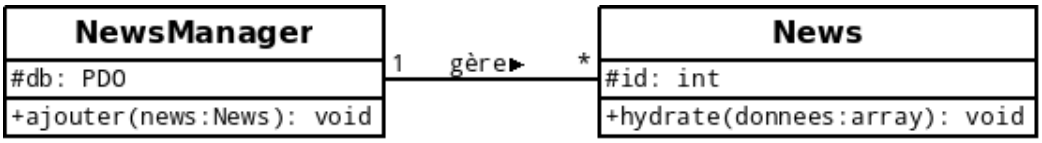


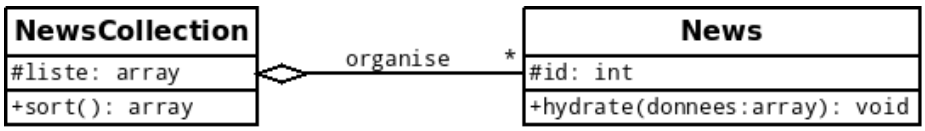


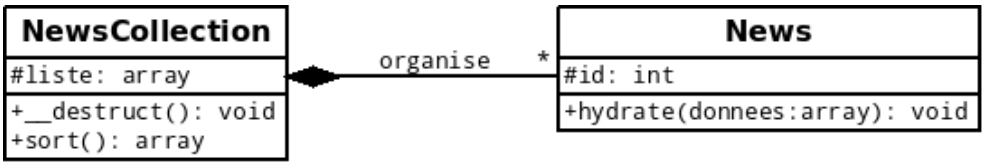












DIA

https://openclassrooms.com/courses/programmez-en-oriente-objet-en-php/uml-modelisons-nos-classes-2-2

Téléchargement sur PC (sur MAC on ne peut pas générer du PHP)

http://dia-installer.de : DIA

http://uml2php5.zpmag.com : Génération de PHP

Pour générer du PHP, il faut commencer par enregistrer le projet.

Ensuite exporter, XSL (*.code), UML-CLASSES-Extended et PHP

On peut aussi exporter en jpg ou en png pour avoir une image

A noter que Power AMC ne génère pas de code PHP

PHP OBJET – PARTIE 4 – COMPLEMENTS

La suite du cours présente maintenant les éléments théorique et syntaxiques supplémentaires pour l'utilisation de la POO.

Méthodes magiques

Présentation

http://php.net/manual/fr/language.oop5.magic.php

Les méthodes magiques sont des méthodes qui sont appelés automatiquement quand certaines actions (new, ->, appel de fonction) sont réalisées.

Ces méthodes commencent toutes par __

Elles peuvent être codées dans chaque classe.

On parle souvent de « surcharge » pour ces méthodes (« overload ») : le terme n'est pas approprié à son usage standard en POO. Il ne s'agit pas vraiment de surcharge puisque ces méthodes ne remplacent pas véritablement une autre méthode. D'ailleurs, la surcharge n'existe pas en PHP.

On en présente quelques-unes ci-dessous. Il y en a d'autres.

constructeur et destructeur

http://php.net/manual/fr/language.oop5.decon.php#object.construct

__construct :

est appelé quand on fait un new. Ca permet les initialisations.

http://php.net/manual/fr/language.oop5.decon.php#object.destruct

__destruct :

est appelée dès qu'il n'y a plus de référence sur un objet donné, ou dans n'importe quel ordre pendant la séquence d'arrêt.

« surcharge » magique de propriété

http://php.net/manual/fr/language.oop5.overloading.php#object.set

__set ($attribut, $value) :

est appelée lors de l'écriture de données vers des propriétés inaccessibles.

est invoquée lorsque unset() est appelée sur des propriétés inaccessibles. __get ($attribut) :

est appelée lors de la lecture de données vers des propriétés inaccessibles.

__isset ($attribut) :

est appelée lorsque isset() ou la fonction empty() sont appelées sur des propriétés inaccessibles.

__unset ($attribut) :

est appelée lorsque unset() est appelée sur des propriétés inaccessibles.

« surcharge » magique de méthode

http://php.net/manual/fr/language.oop5.overloading.php#object.call

__call ($obj->methode) :

est appelée lorsque l'on invoque des méthodes inaccessibles dans un contexte objet : si on appelle une méthode d'objet qui n'existe pas.

__callStatic (ClassObj ::methode) :

est appelée lorsque l'on invoque des méthodes inaccessibles dans un contexte static : si on appelle une méthode de classe (static) qui n'existe pas.

Outils pratiques : __toString(), __debugInfo(), __invoke()

http://php.net/manual/fr/language.oop5.magic.php

Ces trois méthodes permettent de donner des informations sur les variables simples ou complexes.

__toString () :

détermine comment l'objet doit réagir lorsqu'il est traité comme une chaîne de caractères : quand on fait un echo($objet)

__debugInfo :

est appelée par var_dump() lors du traitement d'un objet pour récupérer les propriétés qui doivent être affichées. Si __debugInfo n'existe pas, var_dump affiche toutes les propriétés.

__invoke (…) :

est appelée lorsqu'un script tente d'appeler un objet comme une fonction : quand on fait $objet(5) ;

Objet, pointeur, référence

Présentation

$objet = new Classe;

$objet contient l'identifiant de l'objet réel, sa référence. C'est en réalité l'adresse en mémoire de l'objet réel. $objet est donc en réalité une variable simple qui contient une adresse. Une telle variable est appelé pointeur.

```
$a = new MaClasse;
$b = $a;    // $b contient la même adresse que $a.
            // cette adresse conduit au contenu de l'objet créé
            // il n'y a qu'un seul objet.

$a->attribut1 = 'Hello';
echo $b->attribut1;    // Affiche Hello.
                       // $a et $b référencent le même objet

$b->attribut2 = 'Salut';
echo $a->attribut2;    // Affiche Salut.
                       // $a et $b référencent le même objet
```

Copier un objet = cloner un objet

Présentation

```
$a = new MaClasse;
$copie = clone $a;
```

Dans ce cas, $copie est différent de $a ! Ils n'ont pas la même adresse. Il y a deux objets distincts qui contiennent les mêmes informations.

Méthode magique __clone

La méthode est appelée quand on appelle la méthode clone.

Comparer deux objets

Comparaison classique

if ($a==$b) va vérifier si les deux objets instancient la même classe et s'ils ont les mêmes valeurs d'attributs.

C'est vrai pour 2 clones mais aussi pour 2 références identiques

Comparaison triple =

if($a===$b) va vérifier que les deux références contiennent la même valeur (la même adresse). 2 clones ne vérifient donc plus cette condition.

Il existe un opérateur !== qui est la négation du précédent.

Parcourir les attributs d'un objet : foreach

On peut lister tous les attributs d'un objet, quelle que soit leur visibilité.

foreach ($objet as $valeur) { : $valeur sera la valeur de l'attribut actuellement lu.

foreach ($objet as $attribut => $valeur) {: $attribut aura pour valeur le nom de l'attribut actuellement lu et $valeur sera sa valeur.

Tutoriel sur les variables et les références

https://openclassrooms.com/courses/allez-plus-loin-avec-les-variables#ss_part_6

opérateur &

cf. langage C !

- ***de base***

```
$maVariable = 0;
$monAlias = &$maVariable;
```

- ***paramètre de fonction***

```
function additionne (&$param1, $param2) {
  $param1 += $param2;
}
```

- ***fonction qui retourne une adresse***

```
class MaClasse {
  public $attribut;
  public function &returnAdd(){ //noter le & :renvoie une adresse
    return $this->attribut; // pas de & !!!
  }
}
$monObjet = new MaClasse();
$var = &$monObjet->returnAdd(); // noter le &
```

- ***modification d'un tableau avec un foreach***

```
$texte = 'Voici du texte que l\'on va essayer de transformer.';
$tableau = explode (' ', $texte); // chaque mot dans un tableau.

foreach ($tableau as &$valeur) // Notez le & : on modifie $valeur
  $valeur .= '|'; // .= analogue à +=

echo '<pre>' . print_r ($tableau, true) . '</pre>';
//Voici|, puis du|, etc.
```

variables globales

$GLOBALS : contient toutes les superglobales et les variables globales de la page.

Les globales de la page ne sont visibles que dans la page au moment de son appel.

Toutes les variables de la page correspondent à une clé du tableau $GLOBALS et sont ainsi accessibles dans les fonctions.

Dans une fonction, on peut accéder aux globales par : $GLOBALS['varGlobale'] ou en déclarant : global $varGlobale ;

variable static dans une fonction

une variable static dans une fonction est une variable locale qui conserve sa valeur pour la prochaine utilisation de la fonction.

variable de variable et variable de fonction

le nom d'une variable et celui d'une fonction peuvent être remplacer par une variable.

```
$texte = 'Hello world !';
$variable = 'texte';
echo $$variable; //ou echo ${$variable}; Affiche: Hello world !

$nomFonction = 'strlen';
$chaine = 'Voici une chaîne de caractères';
echo $nomFonction ($chaine); // Affiche « 30 ».
```

cast

cf. langage C !

mixed : tous les types possibles

is_bool, is_int, etc. pour vérifier le type d'une variable

intval($var) : tranforme $var en entier, idem floatval($var)

gettype($var) renvoie le type de la variable

settype($var, 'integer') : $var est désormais de type entier

empty($var) : renvoie vrai si la variable vaut : « », 0, « 0 », NULL, FALSE, array()

écriture octale ou hexadécimale

cf. langage C !

chaîne de caractères

cf. langage C ! : $chaine[3], de 0 à strlen($chaine)-1 : pas de '\0' !!!

Les exceptions et la gestion des erreurs

Présentation

https://openclassrooms.com/courses/programmez-en-oriente-objet-en-php/les-exceptions-10

Les exceptions c'est la technique de la POO pour gérer les erreurs.

L'objectif est qu'une erreur puisse être décelée et traitée comme on le souhaite, et non pas par le système : on redéfinit en quelque sorte les fonctions système de traitement d'erreur.

Ca permettra aussi de traiter l'erreur là où elle apparaît ou ailleurs dans le programme.

Plus généralement, une gestion propre des erreurs est un objectif important pour une application.

Lancer une exception : classe Exception

Exemple :

```
<?php
function additionner($a, $b) {
  if (!is_numeric($a) || !is_numeric($b))
       throw new Exception('Les deux paramètres doivent être des
nombres');
  return $a + $b;
}
function programme(){
  echo additionner(12, 3), '<br />';
  echo additionner('azerty', 54), '<br />';
  echo additionner(4, 8);
}
programme();
```

Message d'erreur

Présentation avec des sauts de lignes, en réalité, tout est à la suite

```
15

Fatal error:
Uncaught Exception:
  Les deux paramètres doivent être des nombres
  in /Users/bertrandliaudet/Sites/06-OBJET/09-Exception/01-
lancerException.php:4

Stack trace:
  #0 /Users/bertrandliaudet/Sites/06-OBJET/09-Exception/01-
lancerException.php(10): additionner('azerty', 54)
  #1 /Users/bertrandliaudet/Sites/06-OBJET/09-Exception/01-
lancerException.php(13): programme()
  #2 {main} thrown in /Users/bertrandliaudet/Sites/06-OBJET/09-
Exception/01-lancerException.php on line 4
```

Analyse du message d'erreur

1 : Fatal error : il s'agit bien d'une erreur fatale ! Le programme s'arrête !

2 : Uncaught Exception : l'exception n'a pas été attrapée, et donc le programme s'arrête.

Deux parties dans l'Uncaught Exception :

- Le message
- Le fichier et la ligne de l'exception

3 : Stack trace : présentation de la pile des appels de fonction, en remontant de la fonction qui plante (en 0) jusqu'au main() : le programme principal, en passant par les fonctions intermédiaires.

Pour chaque fonction, 2 parties :

- Le fichier et la ligne de l'exception
- La fonction avec les paramètres

Attraper une exception : try … catch

Principe

On voit dans le message d'erreur que l'exception dépile les fonctions appelantes.

L'idée est d'attraper l'exception dans la fonction appelante.

Quand on appelle une fonction qui lance une exception, on essaie de la lancer dans un bloc « try » qui est suivi par un bloc « catch » qui récupère l'exception s'il y en a une.

Exemple

```
try{
  programme();
}
catch (Exception $e){
   echo $e->getMessage(), '<br />';
   // echo'<pre>';print_r($e->getTrace());echo'</pre>';
}
echo 'FIN';
```

Résultats

```
15
Les deux paramètres doivent être des nombres
FIN
```

Explication

L'erreur a été attrapée dans la fonction appelante « main » : elle remonte jusqu'au main à travers la fonction programme.

La fonction programme se déroule jusqu'à l'erreur : le 15 est affiché et rien d'autre. Le catch est traité dans le main. Puis la suite du programme.

La classe Exception fournit des méthodes pour accéder au message d'erreur (getMessage), à la pile des fonctions (getTrace), et d'autres : http://php.net/manual/en/class.exception.php

Exception spécialisée

Principe

On peut lancer n'importe quel objet qui hérite de la classe Exception. On peut donc personnaliser nos exceptions.

http://www.php.net/manual/fr/language.exceptions.extending.php

On voit dans la classe Exception que les méthodes sont finales, sauf __toString et __construct.

Exemple de classe spécialisée

On réécrit __toString.

Ainsi, on précise qu'on travaille avec MonException et plus les exceptions en général.

On peut afficher le message directement avec $e.

Le code est plus lisible.

```
class MonException extends Exception {
  public function __toString() {
    return $this->message;
  }
}

function additionner($a, $b) {
  if (!is_numeric($a) || !is_numeric($b))
      throw new MonException('MonException : Les deux paramètres
doivent être des nombres');
  return $a + $b;
}

try{
  programme();
}
catch (MonException $e){
   echo $e, '<br />';
}
```

Plusieurs Catch à la suite

On peut tenter d'attraper une exception spécialisée puis une exception plus générale si on n'a pas réussi. Il suffit d'écrire deux (ou plus) catch à la suite, du plus spécialisé au plus général.

```
try{
  programme();
}
catch (MonException $e){
   echo ' MonException : ', $e, '<br />';
}
catch (Exception $e){
   echo ' Exception : ', $e->getMessage(), '<br />';
}
```

Arbre des exceptions

Présentation

Il existe de nombreuses classes d'exception prédéfinies qui dérivent toutes de la classe Exception.

Notons particulièrement deux enfants : logicalException et RunTimeExcption qui eux-mêmes ont plusieurs enfants.

Les exceptions SPL sont à connaître : http://fr2.php.net/manual/fr/spl.exceptions.php

Notons aussi la classe PDOException qui dérive de RunTimeException

exemple avec PDOException

```
try {
  $db = new PDO(…) ;
}
catch (PDOException $e) {
  echo 'La connexion a échoué.<br />';
  echo 'Informations : [', $e->getCode(), '] ', $e->getMessage();
}
```

Finally : exécuter du code avant une erreur fatale

Si une exception n'est pas attrapée, il y aura une erreur fatale.

Le bloc « finally » qui vient après un try – catch, permet de définir un code à exécuter avant l'arrêt du programme, Ce bloc permet de faire des opérations pour s'assurer que le programme ne s'arrête pas de façon trop sale (nettoyage, fermeture de fichiers, déconnexion, etc.)

Le bloc « finally » peut aussi apparaître directement après un try, sans catch.

Intercepter les erreurs fatales

Principe

Pour intercepter l'exception d'une erreur fatale en exception, il faut un mécanisme qui appelle une fonction chaque fois qu'une erreur fatale est lancée.

Ce mécanisme c'est les « callback ». Sans entrer dans le détail, la fonction set_error_handler() permet d'enregistrer une fonction en « callback ».

Exemple

```
function catchErreurFatale(Exception $e) {
  echo 'Ligne ',$e->getLine(), ' dans ', $e->getFile(), '<br />';
  echo 'Exception lancée : ',$e->getMessage(), '<br />';
}
set_exception_handler('customErreurFatale');
```

On se dote d'une fonction catchErreurFatale. Elle reçoit une Exception en entrée. Cette exception sera l'erreur fatale. On exploite les méthodes de l'exception pour afficher ce qu'on veut.

Compléments

https://openclassrooms.com/courses/programmez-en-oriente-objet-en-php/les-exceptions-10#/id/r-1667288

Les annotations

Présentation

https://openclassrooms.com/courses/programmez-en-oriente-objet-en-php/l-api-de-reflexivite#/id/r-1667551

Les annotations sont une technique de documentation automatique.

Elles utilisent la bibliothèque addendum.

Il faut ensuite documenter ces fichiers selon une syntaxe précise : la syntaxe docblok.

Ensuite, la bibliothèque addendum parcours les classes avec l'API reflection et extrait la domentation.

Les annotations sont surtout utilisées par les frameworks, comme Zend Framework, PHPUnit (framework de tests unitaires), Doctrine (ORM).

Addendum

https://code.google.com/archive/p/addendum/downloads

https://code.google.com/archive/p/addendum/

L'API Reflection

Présentations

http://php.net/manual/fr/intro.reflection.php

L'API Reflection permet d'obtenir des informations de structure sur les objets et les classes, dynamiquement. Par exemple, savoir tel attribut est l'attribut d'une classe particulière, connaître tous les attributs d'une classe, etc.

Principes

Les principales méthodes commencent par has, get, is, new

Les principales méthodes s'appliquent aux classes et interfaces (ReflectionClass ::), d'autres aux attributs (ReflectionProperty ::), d'autres aux methodes (ReflectionMethod ::), d'autre aux objets (ReflectionObject ::), etc.

Les méthodes pour les classes

http://php.net/manual/fr/class.reflectionclass.php

$refMaClasse = new ReflectionClass('MaClasse');

public bool ReflectionClass::hasProperty (string $attribut) // Vérifie si l'attribut existe

public bool ReflectionClass::hasMethode (string $methode) // Vérifie si la methode existe

public bool ReflectionClass::hasConstant (string $CONST) // Vérifie si la CONTANTE existe

echo $ refMaClasse ->getConstant(CONSTANTE) ;

print_r($ refMaClasse ->getConstants(), true) ;

if ($parent = $ refMaClasse ->getParentClass()) {

if ($ refMaClasse ->isSubclassOf('MaClasseMere')) {

if ($refMaClasse ->isAbstract()) {

if ($refMaClasse ->isFinal()) {

if ($refMaClasse ->isInstantiable()) {

Les méthodes pour les interface

$refiMaClasse = new ReflectionClass('iMaClasse');

if ($ refiMaClasse ->isInterface()) {

if ($classeMaClasse->implementsInterface('iMaClasse')) {

ReflectionClass::getInterfaces() // renvoie les interfaces sous formes d'objet ReflectionClass

ReflectionClass::getInterfaceNames() // renvoie les interface sous forme de nom

Les méthodes pour les attributs

http://php.net/manual/fr/class.reflectionproperty.php

$refAttribut = new ReflectionProperty(string $maClasse, string $attribut);

public ReflectionProperty ReflectionClass::getProperty (string $attribut) // un attribut

public array ReflectionClass::getProperties ([int $visibilité]) // Tous les attributs

ReflectionProperty::getName() // nom de l'attribut

ReflectionProperty::getValue($object) // il faut passer un objet en paramètre pour avoir sa valeur

Attention, getValue s'applique aux attributs publics sinon, on a une erreur.

Pour faire un getValue sur un attribut non public, il faut commencer par le rendre accessible par un setAccessible :

ReflectionProperty::->setAccessible(true); // fait comme si l'attribut était public

ReflectionProperty::isPrivate(), isProtected(), isPublic().

ReflectionProperty::isStatic(),

ReflectionProperty::setValue($objet, $valeur) // ATTENTION : contraire à l'encapsulation !!!

Les méthodes pour les attributs static

ReflectionProperty::getValue() // pas besoin d'objet en paramètre .

echo $refMaClasse->getProperty($attribut)->getValue();

ReflectionClass::getStaticPropertyValue($attribut)// équivalent à ce qu'il y a au dessus

ReflectionClass::setStaticPropertyValue($attribut, $value)

ReflectionClass::getStaticProperties(). // retourne un tableau avec les valeurs de tous les attributs static

Les méthodes pour les méthodes

http://php.net/manual/fr/class.reflectionmethod.php

$refMethode = new ReflectionMethod('MaClasse', 'hello'); // méthode hello()

ReflectionClass::getMethod($name).

ReflectionMethod::isPublic(),

ReflectionMethod::isProtected()

ReflectionMethod::isPrivate().

ReflectionMethod::isStatic().

ReflectionMethod::isAbstract()
ReflectionMethod::isFinal().
ReflectionMethod::isConstructor()
ReflectionMethod::isDestructor()

ReflectionMethod::invoke($object, $args) // les arguments en liste

ReflectionMethod::invokeArgs($object, $args) // les arguments dans un tableau
ReflectionMethod::setAccessible($bool). Si $bool vaut TRUE, alors la méthode sera accessible