## Mémento PHP : Programmation orienté objet (POO)

## Class et objet (il faut crée un fichier par class !)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Crée class** | | Class Personnage {attribut et méthode} | | Crée une class, à l'intérieur sont définie des attributs (variable) et méthodes (fonction) (**commencer par une maj**) | | | | | | |
| **visibilité** | | private $\_mavar  private $\_force = 50;  public function afficherForce(){code} | | Accessible seulement depuis l'intérieur de la class (le \_ est pour les attributs ou méthodes private).  Les valeurs données par default doit être une expression scalaire statique (pas de variable, calcule, fonction etc).  Crée une fonction public (accessible depuis l'extérieur) de la class elle-même. | | | | | | |
| **Crée objet** | | $perso = new Personnage; | | | | Crée une instance $perso a partir de la class Personnage // on dit que l'on instancie la classe Personnage | | | | |
| **méthode** | $perso-> afficherForce();  public function frapper(Personnage **$persoAFrapper**)  {**$persoAFrapper**->\_degats += $this->\_force;}  public function frapper(array **$coups**) | | | | | | Utilise la méthode afficherForce() de l'objet $perso (méthode hérité de la class Personnage)  Le fait de préciser la class Personnage oblige a ce que l'argument soit un objet (sera exécuter que si objet)  Affiche la variable private $\_force de l'objet couramment utilisé (avec $this représentant l'objet en cours)  Exige que le paramètre soit un tableau, impossible d'exiger un nombre entier ou une chaîne de caractères de cette façon | | | |
| **L'accesseur (getteur)** | | | public function force() { return $this->\_force; } | | | | | L'accesseur est une fonction qui se charge de renvoyer le contenu d'un l'attribut, ici $\_force. | | |
| **Mutateurs (setter)**  Exemple : vérifie que l'attribut est :  un nombre entier  et qu'il soit <100 | | | public function setExperience(**$experience**){  if (!is\_int(**$experience**)) {  trigger\_error('L\'expérience d\'un personnage doit être un nombre entier', E\_USER\_WARNING); return;}  if (**$experience** > 100) {  trigger\_error('L\'expérience d\'un personnage ne peut dépasser 100', E\_USER\_WARNING);  return;}  $this->\_experience = **$experience**;} | | | | | | | // S'il ne s'agit pas d'un nombre entier.  trigger\_error : affiche ce message lors des erreur du script PHP  // On vérifie bien qu'on ne souhaite pas assigner une valeur supérieure à 100.  // is\_int() et is\_string() pour contrôler |
| **Constructeur**  **instanciation** | | | public function \_\_construct(**$force**, **$degats**) {  echo 'Voici le constructeur !';  $this->setForce(**$force**);  $this->setDegats(**$degats**);  $this->\_experience = 1; }  $perso1 = new Personnage(60, 0); | | | | | | // Constructeur demandant 2 paramètres \_\_construct est appeler a la création  // Message s'affichant une fois que tout objet est créé.  // Initialisation de la force qui appel le mutateurs (ou setteur).  // Initialisation des dégâts.  // Initialisation de l'expérience à 1.  // on crée une instance de la class Personnage $perso1 qui as : 60 de force, 0 dégât | |
| **Auto chargement** | | function chargerClasse(**$classe**){  require **$classe** . '.php';  spl\_autoload\_register('chargerClasse'); | | | // crée une fonction permettant d'inclure les class (il faut qu'elle soit toute dans le même dossier !)  // On inclut la classe correspondante au paramètre passé.}  // On enregistre la fonction en autoload pour qu'elle soit appelée dès qu'on instanciera une classe non déclarée. | | | | | |

## L'operateur de résolution de portée :: (appelé « double deux points » (« Scope Resolution Operator » en anglais))

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Crée constante** | | | const FORCE\_PETITE = 20;  const FORCE\_GRANDE = 80; | Crée des constante (ne pas mettre de $), écrire en majuscule. La constante appartiens à la class et non a l'objet, c'est pourquoi ont ne peux l'appeler via ->, ni $this, ou encore $perso. | |
| **vérification** | if (in\_array($force, [self::FORCE\_PETITE, self::FORCE\_GRANDE])) {  $this->\_force = **$force**; } | | | | // On vérifie qu'on nous donne bien soit une « FORCE\_PETITE », soit une «FORCE\_GRANDE ». // **$force** est le premier paramètre de \_\_construct |
| **création** | | | $perso = new Personnage(Personnage::FORCE\_GRANDE); | | // On envoie une «FORCE\_GRANDE» en guise de force initiale. |
| **Fonction static** | | public static function parler(){echo 'Je vais tous vous tuer !'; } | | | les méthodes statiques sont des méthodes qui sont faites pour agir sur une classe et non sur un objet |
| **Appel fonction static class/objet** | | | Personnage::parler();  $perso->parler(); | Appel la fonction static depuis la class **(préférer cette façon, plutôt que sur l'objet)**  Appel la fonction static depuis un objet de la class Personnage **(ne jamais utiliser $this)** | |
| **Attribut static** | | | private static $\_texteADire = 'Je vais tous vous tuer !'; | Un attribut static appartient a la class et non a un objet si on modifie valeur = la même sr tt objet | |
| **Utiliser attribut** | | | public static function parler() { echo self::$\_texteADire; } | « Dans moi-même, donne-moi l'attribut statique $\_texteADire. » // Self = moi-meme; | |

## Manipulation des données stockées : intégrer les valeur contenu dans une bdd dans un objet

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Requête** | $request = $db->query('SELECT id, nom, forcePerso, degats, niveau, experience FROM personnages'); | | | Requête récupère toute les données Personnages |
| **Boucle** | while ($donnees = $request->fetch(PDO::FETCH\_ASSOC)) | | Chaque entrée sera récupérée et placée dans un array. | |
| **Créa objet** | $perso = new Personnage($donnees); | | Crée un objet $perso | |
| **hydratation** | public function hydrate(array $donnees) {  foreach ($donnees as $key => $value){  $method = 'set'.ucfirst($key);  if (method\_exists($this, $method)) {  $this->$method($value); } } } | Fonction permettant d'insérer les donnée du tableau retourné par la requête dans les attribut de l'objet. il faut passer par les setteur afin de Controller que les données soit valide.  // On récupère le nom du setter correspondant à l'attribut. // ucfirst assemble set + $key  // Si le setter correspondant existe avec : method\_exists(nom-classe-ou-objet,nom-methode)  // On appelle le setter. | | |
| **Astuce** | class A {  public function hello(){  echo 'Hello world !'; }}  $a = new A;  $method = 'hello';  $a->$method(); | Il est possible d'appeler une méthode dynamiquement, c'est-à-dire appeler une méthode dont le nom n'est pas connu à l'avance (en d'autres termes, le nom est connu seulement pendant l'exécution et est donc stocké dans une variable)  // Affiche : « Hello world ! » | | |

## Manipulation des données stockées : manager les donnée avec une class différente, indispensable!

|  |  |  |
| --- | --- | --- |
| **Création**  **PDO** | class PersonnagesManager {  private $\_pdo;  public function \_\_construct($pdo) { $this->setPdo($pdo); }  public function setPdo(PDO $pdo) { $this->\_pdo = $pdo; } | // Crée la class PersonnagesManager (Create, Read, Update, Delete)  // $\_pdo Instance de PDO  // constructeur qui prend en paramètre objet $pdo de class PDO  // setteur pour $\_pdo |
| **Ajouter un personnage**  **Dans la BDD** | public function add(Personnage $perso) {  $req = $this->\_pdo->prepare('INSERT INTO personnages(nom, forcePerso, degats, niveau, experience) VALUES(:nom, :forcePerso, :degats, :niveau, :experience)');  $req ->bindValue(':nom', $perso->nom());  $req ->bindValue(':forcePerso', $perso->forcePerso(), PDO::PARAM\_INT);  $req ->bindValue(':degats', $perso->degats(), PDO::PARAM\_INT);  $req ->bindValue(':niveau', $perso->niveau(), PDO::PARAM\_INT);  $req ->bindValue(':experience', $perso->experience(), PDO::PARAM\_INT);  $req ->execute();} } | // fonction ajoutant une instanciation de l'objet Personnage  // requête préparer pour ajouter le perso dans la BDD |
| **Update un personnage**  **Dans la BDD** | 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->execute(); } | // fonction d'update dans BDD des caractéristiques de l'objet  // requête préparer pour update le perso dans la BDD |

## Héritage, seul les éléments public/protected sont hérité.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Crée class hérité**  **parent::** | Class Magicien extends Personnage{ méthode et attribut nouvel class  Public function gagnerxp() { parent::gagnerxp(); code à la suite } } | | | | // Notre classe Magicien hérite des attributs et méthodes de Personnage. |
| // Ajoute le code à la suite de la méthode gagnerxp() |
| A partir du moment où l'on réécrit une méthode de la classe parente, la méthode appartient à la classe fille et n'a donc plus accès aux éléments private | | | | | |
| **protected** | protected $nom; | | | protected est comme private sauf que les class hérité ont accès aux l'attribut | |
| **Abstration** | abstract class Personnage | Personnage est une class abstraite : on ne peut plus l'utiliser pour crée des objet : il faut faire un héritage pour utiliser | | | |
| abstract public function gagnerxp(); | | Oblige les class fille a écrire la méthode sinon = erreur | | |
| **Finalisation** | final class Personnage | | Personnage est une class final : on ne peut plus crée de class hérité de celle-ci. | | |
| final public function gagnerxp(); | | Méthode hérité, impossible de la modifier dans une instance | | |

## Résolution statique à la volée avec static::

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Exemple** | class Mere{  public static function lancerLeTest() {  **self::**quiEstCe(); }  public function quiEstCe(){  echo 'Je suis la classe <strong>Mere</strong> !';}  }  class Enfant extends Mere{  public function quiEstCe() {  echo 'Je suis la classe <strong>Enfant</strong> !';}  }  Enfant::lancerLeTest();  // ou $e = new Enfant; $e->lancerLeTest(); **(3)** | class Mere{  public static function lancerLeTest() {  **static::**quiEstCe(); }  public function quiEstCe(){  echo 'Je suis la classe <strong>Mere</strong> !';}  }  class Enfant extends Mere{  public function quiEstCe() {  echo 'Je suis la classe <strong>Enfant</strong> !';}  }  Enfant::lancerLeTest();  // ou $e = new Enfant; $e->lancerLeTest();**(3)** | | | // self:: sert à appeler un attribut, une méthode statique ou une constante **de la classe dans laquelle est contenu self::**  // static:: a exactement le même effet que self::, **à l'exception près que static:: appelle l'élément de la classe qui est appelée pendant l'exécution**.  // la résolution statique à la volée a exactement le même effet quand on crée un objet puis qu'on appelle une méthode de celui-ci. Il n'est donc pas du tout obligatoire de rendre les méthodes statiques pour pouvoir y placer static:: **(3)** | |
| **résultat** | Je suis la classe Mere!' | Je suis la classe Fille!' | | |
| **Cas complexe** | class A{  public function appelerQuiEstCe(){ **static::**quiEstCe(); }  public function quiEstCe() { echo 'A';}  }  class B extends A{  public static function test(){ **parent::**appelerQuiEstCe(); }  public function quiEstCe() { echo 'B'; }  }  class C extends B{  public function quiEstCe(){echo 'C';} }  C::test(); | | class A{  public function appelerQuiEstCe(){ **static::**quiEstCe(); }  public function quiEstCe() { echo 'A';}  }  class B extends A{  public static function test(){ **self::**appelerQuiEstCe(); }  public function quiEstCe() { echo 'B'; }  }  class C extends B{  public function quiEstCe(){echo 'C';} }  C::test(); | | | * // Appel de la méthode test de la classe C ; **(3)** * // la méthode n'a pas été réécrite, on appelle donc la méthode test de la classe B ; **(3)** * // on appelle maintenant la méthode**(3)** appelerQuiEstCe de la classe A (avec parent::) ; * // résolution statique à la volée : on appelle la méthode quiEstCe de la classe qui a appelé la méthode appelerQuiEstCe ; **(3)** * // la méthode quiEstCe de la classe C est donc appelée car c'est depuis la classe C qu'on a appelé la méthode test**(3)** |
| **résultat** | C **(3)** | | C | | |
| **Exemple** | class TestParent{  public function \_\_construct() {**static::**qui();}  public **static** function qui(){echo 'TestParent';}  }  class TestChild extends TestParent{  public function \_\_construct() {**static::**qui();}  public function test(){$o = new TestParent();}  public **static** function qui(){echo 'TestChild';}  }  $o = new TestChild;  $o->test(); | | | * Création d'une instance de la classe TestChild ; * appel de la méthode qui de la classe TestChild puisque c'est la méthode \_\_construct de la classe TestChild qui a été appelée ; * appel de la méthode test de la classe TestChild ; * création d'une instance de la classe TestParent ; * appel de la méthode qui de la classe TestParent puisque c'est la méthode \_\_construct de cette classe qui a été appelée. | | |
| **résultat** | À l'écran s'affichera « TestChild » suivi de « TestParent ». | | |

## Methode magique (appeler lors d'événement)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| \_\_destruct | | public function \_\_destruct(){echo 'Destruction de MaClasse';} | | | \_\_destruct sera appelée lors de la **destruction de l'objet**. | | | |
| \_\_set  \_\_get | | public function \_\_set($nom, $valeur){code} | appelée lorsque l'on essaye d'assigner une valeur à un attribut auquel on n'a pas accès ou qui n'existe pas | | | | | |
| public function \_\_get($nom){code} | appelée lorsque l'on essaye d'accéder à un attribut qui n'existe pas ou auquel on n'a pas accès | | | | | |
| Exemple de \_\_set et \_\_get | | class MaClasse{  private $attributs = [];  private $unAttributPrive;  public function \_\_get($nom){if (isset($this->attributs[$nom]))  { return $this->attributs[$nom];}}  public function \_\_set($nom, $valeur) {$this->attributs[$nom] = $valeur;}  public function afficherAttributs(){echo '<pre>', print\_r($this->attributs, true), '</pre>';}  }  $obj = new MaClasse;  $obj->attribut = 'Simple test'; echo $obj->attribut;  $obj->unAttributPrive = 'Autre simple test'; echo $obj->autreAtribut; | | | | | // vérifie si l'attribut auquel on a tenté d'accéder est contenu dans le tableau de stockage d'attributs. Si tel est le cas, on l'affiche, sinon, on ne fait rien.  // Étant donné que tous vos attributs doivent être privés, vous pouvez facilement les mettre en «lecture seule » grâce à \_\_get. L'utilisateur aura accès aux attributs, mais ne pourra pas les modifier. | |
| \_\_isset  \_\_unset | public function \_\_isset($nom){return isset($this->attributs[$nom]);} | | | //est appelée lorsque l'on appelle la fonction [isset](http://fr3.php.net/isset) sur un attribut qui n'existe pas / no accès  //Étant donné que la fonction initiale isset renvoie true ou false, la méthode magique \_\_isset doit renvoyer un booléen  est appelée lorsque l'on appelle la fonction unset sur un attribut qui n'existe pas / no accès | | | | |
| public function \_\_unset($nom){  if (isset($this->attributs[$nom])){unset($this->attributs[$nom]); } | | |
| \_\_call  \_\_callStatic | | public function \_\_call($nom, $arguments){ code } | | | | Appelée lorsque la méthode d'existe pas / private | | |
| public static function \_\_callStatic($nom, $arguments){code} | | | | Idem mais lors d'appel de méthode static / doit obligatoirement être fonction static | | |
| serialize  unserialize | | $\_SESSION['connexion'] = serialize($objetConnexion); | | | | transformation de l'objet en chaîne de caractères | | |
| $objetConnexion = unserialize($\_SESSION['connexion']); | | | | transformation de la chaîne de caractères en objet | | |
| Exemple  \_\_sleep  \_\_wakeup | | protected function connexionBDD() {  $this->pdo = new PDO('mysql:host='.$this->serveur.';dbname='.$this->dataBase, $this->utilisateur, $this->motDePasse); }  public function \_\_sleep(){return ['serveur', 'utilisateur', 'motDePasse', 'dataBase']; }  public function \_\_wakeup(){ $this->connexionBDD();} | | | | | | // Appelée lors de serialize  // Appelée lors de unserialize |
| \_\_toString | | est appelée lorsque l'objet est amené à être converti en chaîne de caractères | | | | | | |
| **\_\_set\_state** | | est appelée lorsque vous appelez la fonction [var\_export](http://fr2.php.net/var_export) en passant votre objet à exporter en paramètre | | | | | | |
| \_\_invoke | | Elle est appelée dès qu'on essaye d'utiliser l'objet comme fonction. Cette méthode comprend autant de paramètres que d'arguments passés à la fonction. | | | | | | |
| \_\_debugInfo | | Cette méthode magique est invoquée sur notre objet lorsqu'on appelle la fonction [var\_dump](https://php.net/manual/fr/function.var-dump.php) | | | | | | |

## Un objet, un identifiant

|  |  |  |  |
| --- | --- | --- | --- |
| **Cas concret** | class MaClasse{  public $attribut1;  public $attribut2; }  $a = new MaClasse;  $b = $a; // On assigne à $b l'identifiant de $a, donc $a et $b représentent le même objet.  $a->attribut1 = 'Hello'; echo $b->attribut1; // Affiche Hello.  $b->attribut2 = 'Salut'; echo $a->attribut2; // Affiche Salut. | | Une variable d'objet ne contient pas l'objet en lui même mais un identifiant qui représente cette objet.  Ainsi dans ce cas la $a et $b font référence au même objet, c'est a dire qu'ont ne peux attribuer des attribut et méthode différente pour $a et $b |
| **copier** | $copie = clone $origine; // On copie le contenu de l'objet $origine dans l'objet $copie. | | Une copie de l'état actuel de l'objet, indépendant de l'origine |
| **\_\_clone** | public function \_\_clone() {self::$instances++;} // \_\_clone est appeler sur l'origine | | Des modification peuvent être apporter lors de la copie |
| **comparer** | il faut que $objet1 et $objet2 aient les mêmes attributs et les mêmes valeurs, mais également que les deux objets soient des instances de la même classe  == comparaison des attribut/méthode/valeur // === comparaison strict : vérifie en plus l'ID | | |
| **Parcourir**  **Un objet** | foreach ($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. | |

## Interface (class abstraite dont le rôle est de décrire un comportement)(constante non abstraite)

|  |  |  |
| --- | --- | --- |
| **crée** | interface Movable{  public function move($dest);  } | 1. Toutes les méthodes présentes dans une interface doivent être publiques. 2. Une interface ne peut pas lister de méthodes abstraites ou finales.   Une interface ne peut pas avoir le même nom qu'une classe et vice-versa. |
| **implémenter** | class Personnage implements Movable{  public function move($dest) {}  } | Implémente la class Movable à Personnage  //l'interface oblige a réécrite méthode sinon:erreur // d'abord spécifier la classe à hériter avec le mot-clé extends**puis** les interfaces à implémenter avec le mot-clé implements |
| **Ex : hérité** | interface iC extends iA, iB{public function test3();} | Ont ne peut réécrire ni une méthode / constante qui a déjà été listée dans l'interface parente |

## Interface prédéfinie

|  |  |
| --- | --- |
| **Iterator** | Cette interface comporte 5 méthodes :   * current : renvoie l'élément courant ; * key : retourne la clé de l'élément courant ; * next : déplace le pointeur sur l'élément suivant ; * rewind : remet le pointeur sur le premier élément ; * valid : vérifie si la position courante est valide. |
| **SeeKableIterator** | ajoute une méthode à la liste des méthodes d'Iterator : la méthode seek. Cette méthode permet de placer le curseur interne à une position précise |
| **ArrayAccess** | L'interface ArrayAccess liste quatre méthodes :   * offsetExists : méthode qui vérifiera l'existence de la clé entre crochets lorsque l'objet est passé à la fonction isset ou empty (cette valeur entre crochet est passé à la méthode en paramètre) ; * offsetGet : méthode appelée lorsqu'on fait un simple $obj['clé']. La valeur 'clé' est donc passée à la méthode offsetGet ; * offsetSet : méthode appelée lorsqu'on assigne une valeur à une entrée. Cette méthode reçoit donc deux arguments, la valeur de la clé et la valeur qu'on veut lui assigner.   offsetUnset : méthode appelée lorsqu'on appelle la fonction unset sur l'objet avec une valeur entre crochets. Cette méthode reçoit un argument, la valeur qui est mise entre les crochets. |
| **Countable** | Ne contiens qu'une méthode : count() // doit renvoyer un nombre qui retourne le nombre d'entré au tableau |
| **ArrayIterator** | Implémente les 4 interfaces ce-dessus // et contient un constructeur qui accepte un tableau en argument |

## Exceptions(attraper une exception permet au script de se dérouler jusqu'a la fin sans arret)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Personnaliser exception** | if (){  throw new Exception('Message a retourner '); } | | Placer throw dans un if (entre les {})  // = if(true) lance throw (hérité d'Exception) | |
| **Attraper une erreur** | try {  code a executer  } catch (Exception $e) {  echo 'Une exception a été lancée. Message d\'erreur : ', $e->getMessage();  } | | //on essaye le code a executer  //si une erreur est levé (que le try a échouer) alors renvois l'exception avec le message personnalisé définie dans throw | |
| **La class Exeptions** | class Exception {  protected $message = 'exception inconnu'; // Message de l'exception.  protected $code = 0; // Code de l'exception défini par l'utilisateur.  protected $file; // Nom du fichier source de l'exception.  protected $line; // Ligne de la source de l'exception.    final function getMessage(); // Message de l'exception.  final function getCode(); // Code de l'exception.  final function getFile(); // Nom du fichier source.  final function getLine(); // Ligne du fichier source.  final function getTrace(); // Un tableau de backtrace().  final function getTraceAsString(); // Chaîne formattée de trace.  /\* Remplacable \*/  function \_\_construct ($message = NULL, $code = 0);  function \_\_toString(); // Chaîne formatée pour l'affichage.} | | | Voici la liste des attributs et méthodes de la classe Exception  Ainsi, nous voyons que l'on a accès aux attributs protégés de la classe et qu'on peut réécrire les méthodes \_\_construct et \_\_toString. Toutes les autres méthodes sont finales, nous n'avons donc pas le droit de les réécrire. |
| **Class hérite de Exception** | class MonException extends Exception{  public function \_\_construct($message, $code = 0) {parent::\_\_construct($message, $code);}  public function \_\_toString() {return $this->message;}}  ... catch (MonException $e) ... | | | Ici nous attrapons les erreur qui serais exécuter dans un try avec les exeption MonException hérité de Exception. On peut enchainer des catch avec plusieurs class Exception |
| **Ex : Exception prédefinie** | InvalidArgumentException | Exception émise si un argument ne correspond pas au type attendu. | | |
| **Exécuter code même si exception pas attrapé** | try{}  finally{ echo 'Action effectuée quoi qu\'il arrive';} | finally est appeler si aucun catch n'a su attrapé l'erreur, finally est donc appeler avant l'erreur fatal (car exception non attraper). | | |

## Les traits (ajout de méthode au class, palliant au problème de l'héritage unique)

|  |  |  |  |
| --- | --- | --- | --- |
| **Crée trait** | trait MonTrait{  public function hello(){  echo 'Hello world !';}  } | | Crée un trait avec la methode hello()  // un trait peut utiliser d'autre traits (use) |
| **Utilisé le trait** | class A{  use MonTrait;  } | | Ajoute (use) le trait qui comporte la méthode hello()  // on peut maintenant faire : $a = new A; $a->hello(); |
| **Resolution de conflit entre trait** | class A{  use Trait1, Trait2{  Trait1:: hello insteadof Trait2;  } | | //permet de résoudre conflit si 2 trait on méthode même nom  Cette ligne signifie donc : « La méthode *hello()* du trait *Trait1* écrasera la méthode du même nom du trait *Trait2* (si elle y est définie). » |
| **entre trait et class** | | Si une classe déclare une méthode et qu'elle utilise un trait possédant cette même méthode, alors la méthode déclarée dans la classe l'emportera sur la méthode déclarée dans le trait | |
| **entre trait et class fille hérité de mère** | | À l'inverse, si une classe utilise un trait possédant une méthode déjà implémentée dans la classe mère de la classe utilisant le trait, alors ce sera la méthode du trait qui sera utilisée (la méthode du trait écrasera celle de la méthode de la classe mère) | |
| **Attribut de trait** | | NE pas utilisé d'attribut qui peuvent rentrer en conflit sinon erreur fatal ou strict ! (+ un attribut ne peut être static contrairement méthode) | |
| **Changer visibilité et nom des méthodes** | use A {  saySomething as protected;}  use A{  saySomething as sayWhoYouAre;}  use A{  saySomething as protected sayWhoYouAre;} | | // definie la fonction saySomething comme protegé  // crée un alias de saySomething en sayWhoAreYou, ainsi on pourra appeler la methode depuis les 2 nom.  //combine les deux exemple ci dessus |

## API de reflexivité (obtenir info class (méthode, attribut...)

|  |  |  |  |
| --- | --- | --- | --- |
| **information** | $classeMagicien = new ReflectionClass('Magicien'); | | Crée un objet hérité de ReflexionClass et en paramètre le nom de la class a analysé |
| **Avec un objet** | $magicien = new Magicien(['nom' => 'vyk12', 'type' => 'magicien']);  $classeMagicien = new ReflectionObject($magicien); | | // instancie un objet hérité de Magicien  // puis instancie un objet de la class ReflectionObject |
| **Info : attribut** | | $classeMagicien->hasProperty('magie');  $classeMagicien->getProperty('magie');  $classeMagicien->getProperties('magie');  $classeA->getStaticPropertyValue('attr');  foreach ($classeMagicien->getProperties() as $attribut)  {  $attribut->setAccessible(true);  echo $attribut->getName(), ' => ', $attribut->getValue($magicien);  }  $attribut->isPublic()// isProtected(), isPublic(), isStatic() | Si l'attribut magie existe renverra true, sinon false  Recupere l'attribut et le renvois dans une instanciation de la classe ReflectionProperty  Renvois un tableau contenant autant d'instance de ReflectionProperty que d'attributs  Recupere la valeur d'un attribut static (le paramètre est le nom de l'attribut)  // crée une boucle afin d'accéder au nom et au valeur des attributs de la classe  //  // prend en paramètre true pour rendre accessible les attribut (si il sont privé,  // ou protected)  // getValue retourne la valeur, prend en paramètre la class  //  Renvois true si l'attribut est public |
| **Info : méthode** | | $classeMagicien->hasMethod('lancerUnSort');  $methode = $classeA->getMethod('hello');  $methode->isPublic(); // isProtected(), isPublic(), isStatic()  $methode->isAbstract(); // isFinal()  $hello->invoke($a, 'test', 'autre test'); | Si la methode lancerUnSort existe renverra true, sinon false  Renvois une instance de ReflectionClass représentant la méthode hello()  Retourne true si la méthode est public, sinon false  Retourne true si la méthode est abstraite, sinon false  Appel methode ($a) depuis l'objet ($hello) et donne deux arguments:' test', 'autretest' |
| **Info : constante**  **Liste complète** | | $classeMagicien->hasConstant('NOUVEAU'); | Si la constante NOUVEAU existe renverra true, sinon false |
| print\_r($classePersonnage->getConstants(), true); | Renvois toute les constante de la classe sous forme de tableau |
| **Classe parente** | | $parent = $classeMagicien->getParentClass();  $parent = $classeMagicien->getName();  $parent = $classeMagicien->isSubclassOf('Personnage'); | Renvois une instance de la class ReflectionClass qui représente classe parente ou false  Renvois le nom de la classe  Renvois true si $classMagicien est hérité de Personnage, sinon false |
| **Info : classe** | | $classePersonnage->isAbstract()// isFinal()  $classePersonnage->isInstantiable() | Renvois true si la classe est abstraite, sinon false  Renvois true si la classe est instanciable, sinon false |
| **interface** | | $classeIMagicien = new ReflectionClass('iMagicien');  $classeIMagicien->isInterface()  $classeMagicien->implementsInterface('iMagicien')  $classeMagicien->getInterfaces('iMagicien')  $classeMagicien-> getInterfacesNames ('iMagicien') | // instancie la classe ReflectionClass en spécifiant une interface iMagicien  Renvois true c'est une interface, sinon false  Renvois true c'est une interface est inplémenter, sinon false  Renvois autant d'instances de la class ReflectionClass qu'il y a d'interfaces  Renvois un tableau contenant le nom de toute les interfaces implémenté |

## Annotations (via addentum et sa class Annotation)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **préparatif** | class Table extends Annotation {}  class Type extends Annotation {} | Si nous voulons ajouter une annotation, Table par exemple, à notre classe pour spécifier à quelle table un objet Personnage correspond, alors il faudra au préalable créer une classe Table | | | |
| **Ex :Table**  **:une valeur**  **Ex : Type**  **:un tableau**  **Clés/valeurs**  **imbrications** | /\*\*  \* @Table("personnages")  \*/  /\*\*  \* @Type({'brute', 'guerrier', 'magicien'})  \*/  @Type({meilleur = 'magicien', 'moins bon' = 'brute', neutre = 'guerrier'})  @UneAnnotation({uneCle = 1337, {uneCle2 = true}}) | | | Précise de quel table correspond une class  Il est aussi possible, pour une annotation, d'avoir un tableau comme valeur. Pour réaliser ceci, il faut mettre la valeur de l'annotation entre accolades et séparer les valeurs du tableau par des virgules  // ou encore avec des clés/valeurs  // un tableau dans un autre | |
| **récupérer** | $reflectedClass = new ReflectionAnnotatedClass('Personnage');  $reflectedClass->getAnnotation('Table')->value; | | Crée une instance de ReflectionAnnotatedClass avec Personnage  Crée une instance de Table (hérité de Annotations) et accède a la valeur avec value | | |
| **Info si class possède** | $reflectedClass = new ReflectionAnnotatedClass('Personnage');  $reflectedClass->hasAnnotation('Table') | | Crée une instance de ReflectionAnnotatedClass avec Personnage  Renvois true si l'annotation 'Table' est présente | | |
| **Annotations valeur multiple**  **utilisation** | /\*\* @Target("class") \*/  class ClassInfos extends Annotation  {  public $author;  public $version;  }  /\*\*  \* @ClassInfos(author = "vyk12", version = "1.0")  \*/ | | // permet d'imposer un type de cible : differente valeur : property, method, nesty  Crée une class ClassInfos hérité de Annotation  // les attributs doivent être public sinon erreur  // les valeurs sont renseigner entre () tel que *clé="valeur",* | | |
| **récupérer** | echo $classInfos->author;  echo $classInfos->version; | | Récupère les valeurs | | |
| **Attribut et méthode** | $reflectedAttr = new ReflectionAnnotatedProperty('Personnage', 'force');  $reflectedMethod = new ReflectionAnnotatedMethod('Personnage', 'deplacer');  $reflectedAttr->getAnnotation('AttrInfos')  $reflectedMethod->getAllAnnotations('ParamInfo') | | | | Le constructeur de ces classes attend en premier paramètre le nom de la classe contenant l'élément et le nom de l'attribut ou de la méthode  //getAllAnnotations() permet de récupérer toutes les annotations d'une entité correspondant au nom donné en argument |

## UML

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| C:\Users\Root\Desktop\403745.png | nom de la classe, puis les attributs et les méthodes(attribut : type).  // : int (=nombre entier) // : array (=tableau)  // : void (=ne renvois rien)  // : mixed = peux renvoyer plusieurs types de résultats différents | | | * Le signe + : l'élément suivi de ce signe est public. * Le signe # : l'élément suivi de ce signe est protégé. * Le signe - : l'élément suivi de ce signe est privé. |
| C:\Users\Root\Desktop\403819.png | * => **Sans style d'écriture particulier**, la méthode est « **normale** », c'est-à-dire ni abstraite, ni finale (comme ici). * =>Si la méthode est en **italique**, cela signifie qu'elle est **abstraite**. * =>Pour montrer qu'une méthode est **finale**, on le spécifie en stéréotype (on place le **mot leaf** **entre chevrons** à côté de la méthode). * => Si l'attribut ou la méthode est **souligné**, cela signifie que l'élément est **statique**. * => Une constante est représentée comme étant un attribut public, statique, de type const et est écrite en **majuscules**. | | | |
| C:\Users\Root\Desktop\403757.png | | | * Parmi les interactions, on peut citer l'**héritage** qui est symbolisé par une simple flèche. | |
| C:\Users\Root\Desktop\405059.png | | | * Une **interface** n'est rien d'autre qu'une classe entièrement abstraite, elle est donc considérée comme tel. * Si une classe doit implémenter une **interface**, alors on utilisera la flèche en pointillés | |
| C:\Users\Root\Desktop\403889.png | | * Le mot écrit au centre, au-dessus de la ligne est la **définition** de la relation. Il est suivi d'un petit symbole indiquant le sens de l'**association.** le chiffre 1 et le symbole \* sont les **cardinalités** qui présentent le nombre d'instances qui participent à l'interaction. On peut désormais lire facilement « 1 NewsManager gère une infinité de News » | | |
| C:\Users\Root\Desktop\403755.png | | * on parlera d'**agrégation** entre deux classes lorsque l'une d'entre elles contiendra au moins une instance de l'autre classe. il n'y a pas de cardinalité du côté du losange. En effet, le côté ayant *le losange signifie qu'il y a obligatoirement une et une seule instance de la classe par relation* | | |
| C:\Users\Root\Desktop\403756.png | | * La **composition** est une agrégation particulière. On parlera de **composition** si, lorsque l'instance de A sera supprimée, toutes les instances de B contenues dans l'instance de A sont elles aussi supprimées (*ce qui n'était pas le cas avec l'agrégation*). | | |