

P.O.O.

La Programmation Orientée Objet en PHP

LES BASES

C'est quoi la POO ?

C'est quoi la POO ?

- La POO est un moyen de penser son code pour en faciliter la maintenance et la distribution
- Il est généralement opposé à la programmation procédurale où les données sont séparées de la façon dont on les traite (par exemple d'un coté vous avez des news, une connexion à la base de données... et de l'autre une suite de fonctions pour modifier ces données)
- En POO, le but est de faire en sorte que tout soit objet et que ces objets interagissent entre eux

C'est quoi un objet ?

- Un objet en POO ressemble un peu à ceux que nous avons dans notre quotidien :
 - *Une lampe*
 - *Une voiture*
 - *Un canapé*
- Un objet a ainsi une suite de propriétés (des attributs) qui lui sont propres (pour une voiture ça pourrait être sa marque, son modèle, sa couleur...)
- Mais aussi une suite de capacités (des méthodes) qui lui sont propres :
 - *Une voiture peut démarrer, tourner, accélérer, ouvrir le coffre...*
 - *Ca serait beaucoup moins pertinent pour désigner les interactions d'un canapé*

C'est quoi une classe ?

- Une classe va être l'équivalent du plan de montage général d'un objet
- Si on prend l'exemple d'une maison, toutes les maisons auront des pièces, des fenêtres, des portes... mais pas forcément le même nombre et pas à la même place
- On va donc utiliser la classe « Maison » pour fabriquer des objets « Maison de Francis », « Maison de Mathilde »...
- On parle alors d'un objet comme étant l'instance d'une classe
- Comme nous habitons dans des maisons et pas dans des plans de maison, nous avons besoin d'instancier les objets pour pouvoir les utiliser (même si il existe des classes dites statiques que l'on peut utiliser directement mais nous aurons le temps d'en parler plus tard)

```
// Je crée ma classe
class Maison
{
    // Ici mes attributs
    private $fenetre;
    protected $porte;
    public $piece;

    // Et mes méthodes
    public function ouvrirFenetre()
    {
        return 'Fenêtre ouverte';
    }

    protected function ouvrirPorte()
    {
        return 'Porte ouverte';
    }

    private function compterPieces()
    {
        return 'Il y a quelques pieces';
    }
}

// J'instancie mes objets
$maisonDeMichel = new Maison();
$maisonDeAnnie = new Maison();

// Et je peux appeler leurs méthodes
$maisonDeMichel->ouvrirFenetre();
```

La syntaxe de base

- Pour créer une classe on utilise simplement le mot clé « class »
- Le nom d'une classe commence par convention par une Majuscule
- Les attributs comme les méthodes ont 3 visibilités différentes :
 - *Public : on peut l'appeler depuis l'intérieur ou l'extérieur de la classe ou de l'objet*
 - *Protected : on ne peut l'appeler que depuis l'intérieur de la classe ou de l'objet ainsi que à l'intérieur des classes ou des objets enfants (on parlera de l'héritage tout à l'heure)*
 - *Private : plus radical, on ne peut l'appeler que depuis l'intérieur de la classe ou de l'objet*
- On instancie des objets avec le mot clé « new »
- On appelle des méthodes avec une flèche « -> »

Le principe d'encapsulation

- C'est un principe très important en POO qui semble alourdir la syntaxe mais qui se prouvera utile avec le temps
- Le principe d'encapsulation dit que tous les attributs d'un objet sont soit « private » soit « protected » et ne sont accessibles que au travers d'accesseurs et de mutateurs (les getters et les setters)
- Cela permet de s'assurer que les propriétés d'un objet ne sont pas manipulables n'importe comment dans le code par l'utilisateur

```
class Personne
{
    private $force;

    public function setForce($force)
    {
        $this->force = $force;
    }

    public function getForce()
    {
        return $this->force;
    }
}

$maurice = new Personne();
$maurice->force; // Va renvoyer une erreur car l'attribut est privé
$maurice->getForce();
```

Le mot clé \$this

- Je fais une petite parenthèse sur ce mot clé qui peut souvent porter à confusion
- « \$this » renvoie à l'objet en cours (et j'insiste, on parle bien de l'objet en cours et pas de la classe en cours)

```
public function setForce($force)
{
    // $this cible l'objet en cours, accède
    // à sa force et lui donne une valeur
    $this->force = $force;
}

public function exemple($uneValeur)
{
    // Et ça marche aussi avec les méthodes
    $this->setForce($uneValeur);
}
```

Les valeurs des attributs

```
class Personne
{
    private $force = 125;
    private $maison = 'Paris';
    private $roles = ['pasteur', 'electricien'];
}
```

- Il est possible de donner une valeur de base aux attributs d'un objet
- Attention cependant, ces valeurs doivent être des expressions scalaires statiques (qui s'accepte qu'un seul type de données : string, int, float ou booleen).
- On ne peut pas leur donner de valeur qui soit un appel à une fonction ou une variable superglobale ou encore une opération...

Les interactions entre objets

- Il est tout à fait possible de faire interagir des objets entre eux
- Ici j'utilise deux objets de la même classe mais la même chose aurait été possible avec des instances de deux classes différentes en passant par « setDegat »

```
class Perso
{
    private $force = 20;
    private $degat;

    // Je peux spécifier ce que je veux en argument
    // de ma méthode, ici une instance de la classe Perso
    public function frapper(Perso $persoAfrapper)
    {
        // Même si l'attribut est privé, je peux
        // y accéder puisque je vais chercher un attribut
        // de la classe Perso depuis une instance de cette
        // même classe
        $persoAfrapper->degat += $this->force;
    }

    public function getDegat()
    {
        return $this->degat;
    }
}

// Je crée mes deux perso
$maurice = new Perso();
$francis = new Perso();

// Et je fais la bagarre
$maurice->frapper($francis);
echo $francis->getDegat();
```

Exercice

- Je vous laisse maintenant faire deux personnages, avec
 - *100 points de vie chacun*
 - *Une force de 20*
 - *Une défense de 10*
- Faites en sorte de pouvoir attaquer un personnage et de changer ses points de vie en fonction de la force de l'adversaire et de sa défense
- Créez aussi une méthode pour afficher les points de vie restants

Correction

- Ici rien de très méchant, on va utiliser le principe d'encapsulation et mettre tous nos attributs en privé et utiliser nos getters et setters pour y accéder
- Je vais aussi utiliser les capacités de PHP 7 et Type Hint tous mes éléments pour m'assurer de recevoir les bonnes informations

```
class Perso
{
    private int $pv = 100;
    private int $force = 20;
    private int $def = 10;

    public function attack(Perso $perso): void
    {
        $newPv = $perso->getPv() - ($this->getForce() - $perso->getDef());
        $perso->setPv($newPv);
    }

    public function getPv(): int
    {
        return $this->pv;
    }

    public function setPv(int $pv): void{...}

    public function getForce(): int{...}

    public function setForce(int $force): void{...}

    public function getDef(): int{...}

    public function setDef(int $def): void{...}
}
```

Le constructeur

```
class Perso
{
    private $force;
    private $degat;

    // Avec deux underscores
    public function __construct($force, $degat)
    {
        $this->force = $force;
        $this->degat = $degat;
    }

    // Maurice aura 20 de force et 0 dégats
    $maurice = new Perso(20, 0);
    // Francis 12 de force et 21 dégats
    $francis = new Perso(12, 21);
```

- Le constructeur est une méthode « magique » qui est lancée automatiquement quand on instancie un objet
- Les arguments passés à cette méthode sont ceux spécifiés lors de l'instance de l'objet
- On l'appelle avec le nom « `__construct` »
- Dans 99,9% des cas, on met le constructeur en publique sinon on ne peut pas instancier notre objet...

Exercice

- En reprenant nos personnages précédents, créez deux personnages :
 - *Un premier avec 100 points de vie, 10 d'attaque et 5 de défense*
 - *Un second avec 50 points de vie 40 d'attaque et 12 de défence*

Correction

- Ici aussi, peu de choses nouvelles, je vais juste devoir revoir ma méthode « naïve » d'attaque car un personnage avec une défense plus forte que l'attaque de son adversaire aurait résulté en une régénération de ses points de vie

```
class Perso
{
    private int $pv;
    private int $force;
    private int $def;

    public function __construct(int $pv, int $force, int $def)
    {
        $this->pv = $pv;
        $this->force = $force;
        $this->def = $def;
    }

    public function attack(Perso $perso): void
    {
        $newPv = $perso->getPv() - ($this->getForce() - $perso->getDef());
        if ($newPv > $perso->getPv()) {
            return;
        }
        $perso->setPv($newPv);
    }
}
```

AUTOLOAD

Un peu de rangement

Le chargement des classes

```
// spl_autoload_register() enregistre une nouvelle
// fonction dans la pile d'auto-load

// Notez que je me suis permis d'utiliser une
// fonction anonyme pour rendre le code lisible
spl_autoload_register(function ($className) {
    require $className . '.php';
});

$maurice = new Perso(20, 0);

$maurice->bonjour();
```

- Comme vous avez pu le remarquer, les classes prennent vite de la place dans un fichier, on va donc ranger tout ça dans des fichiers séparés pour garder notre code propre
- Pour garder les choses propres, on nomme notre fichier avec le nom de la classe qui est dedans
- Enfin on va utiliser une fonction d'autoload qui va nous permettre d'aller chercher le fichier contenant la classe quand on va vouloir instancier un objet avec
- Il existe dans PHP une « pile d'auto-load » avec une liste de fonctions. Quand on essaye d'instancier une classe non déclarée, PHP va les appeler (dans l'ordre de leur déclaration) jusqu'à ce que la classe soit chargée.

ATTRIBUTS ET MÉTHODES STATIQUES

Et puis les constantes de classe aussi tant qu'à faire

Scope Resolution Operator

- Derrière ce nom bien barbare se cache le symbole « :: » aussi appelé « double deux points »
- Il est utilisé pour appeler des méthodes liées à une classe et non pas à un objet.
Ces méthodes sont appelées "statiques"
- Parmi ces éléments statiques on trouve les constantes de classe. Elles sont particulièrement pratiques pour rendre le code plus lisible

```
// 50 quoi ? c'est pas super lisible
$perso = new Personnage(50);

class Personnage
{
    private $force;

    // Déclarations des constantes en rapport avec la force.
    // Notez l'absence de $
    const FORCE_PETITE = 20;
    const FORCE_MOYENNE = 50;
    const FORCE_GRANDE = 80;

    public function __construct($forceInitiale)
    {
        // On assigne la valeur d'un attribut depuis son setter
        $this->setForce($forceInitiale);
    }

    public function setForce($force)
    {
        // On vérifie qu'on nous donne bien une constante valable

        // Contrairement à $this qui renvoie à l'objet en cours,
        // on fait référence à la classe en cours avec self::
        if (in_array($force, [self::FORCE_PETITE, self::FORCE_MOYENNE, self::FORCE_GRANDE])) {
            $this->force = $force;
        }
    }

    // On envoie une « FORCE_MOYENNE » en guise de force initiale.
    // Notez que on nomme la classe à qui appartient la constante
    $perso = new Personnage(Personnage::FORCE_MOYENNE);
```

Méthodes statiques

```
class Perso
{
    public static function parler()
    {
        echo 'Je suis un mec sympa !';
    }
}

Perso::parler();
```

- De la même façon il est possible de créer des méthodes statiques
- Ces méthodes sont donc propre à la classe en général plutôt qu'à un objet en particulier et peuvent être appelées sans avoir besoin d'instancier un objet
- ATTENTION : le mot clé « \$this » faisant référence à l'objet en cours, il n'aura aucun effet si ce n'est produire une erreur si utilisé dans une méthode statique (ce qui est logique puisque il n'y a, par définition, pas d'objet en cours)

Attributs statiques

- Pour finir le tour des éléments statiques, on peut évidemment avoir des attributs statiques
- Ils ont un énorme avantage, c'est que comme ils sont dépendants de la classe et non pas de l'objet, si leur valeur est modifiée, tous les objets instanciant cette classe bénéficieront du changement

```
class Perso
{
    // Un attribut statique
    private static $texte = 'Je suis un mec sympa !';

    // Une méthode statique
    public static function parler()
    {
        // Source d'erreur : ici la variable prend son $
        echo self::$texte;
    }
}

Perso::parler();
```

Exercice

- Pour mettre les attributs et méthodes statiques à profit
- Construisez un compteur qui pourra dire combien d'objets d'une classe ont été instanciés

Correction

- Je vais utiliser un attribut statique pour stocker la valeur du compteur
- J'incrémente ce compteur depuis le constructeur
- Enfin je fais une petite méthode statique pour afficher le compteur

```
class Perso
{
    // Mon compteur, statique
    private static $count = 0;

    // Le constructeur incrémente le compteur
    public function __construct()
    {
        self::$count++;
    }

    // Une méthode pour afficher le résultat
    public static function getCount()
    {
        return self::$count;
    }
}

$a = new Perso();
$b = new Perso();
$c = new Perso();

echo Perso::getCount();
```

BOSSER AVEC UNE BASE DE DONNÉES

Parce-que c'est quand même pratique

1 entité = 1 objet

- Quand on cherche à représenter quelque chose avec des objets, il est important de bien ranger les choses
- Comme une base de données n'est rien d'autre qu'un ensemble de tableaux organisant des données, on va pouvoir représenter ça sous forme d'objet
- Notez que j'ai nommé mes attributs exactement de la même façon que mes colonnes dans ma DB, c'est beau mais surtout, ça a son importance, vous allez voir

```
class Perso
{
    private $id;
    private $nom;
    private $forcePerso;
    private $degats;
    private $niveau;
    private $experience;
}
```

id	nom	forcePerso	degats	niveau	experience
1	Francis	5	55	4	20
2	Maurice	5	5	1	0
3	Robert	5	0	1	20
4	Jean-Paul	78	1	90	76
5	Gneugneugneu	78	1	90	76

```
class Perso
{
    private $id;
    private $nom;
    private $forcePerso;
    private $degats;
    private $niveau;
    private $experience;

    public function setId(int $id)
    {
        $this->id = $id;
    }

    public function setNom(string $nom)
    {
        $this->nom = $nom;
    }

    public function setForcePerso(int $force)
    {
        if ( $force > 0 && $force <= 100 ) {
            $this->forcePerso = $force;
        }
    }

    // etc... vous avez compris l'idée
    // Notez que les setters sont en Camel Case
}
```

Les getters et les setters

- On va bien sur penser à faire ses getters et ses setters pour vérifier l'intégrité de la donnée qui rentre et qui sort de notre objet
- Par convention, les setters et les getters sont écrits en Camel Case (avec la première lettre du premier mot en minuscule et les autres premières lettres de mots en majuscules)

```
class Perso
{
    private $id;
    private $nom;
    private $forcePerso;
    private $degats;
    private $niveau;
    private $experience;

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

    private function hydrate(array $data)
    {
        // Je parse l'array passé au constructeur
        foreach ($data as $key => $value) {
            // Je construit le nom du setter
            // associé à l'attribut passé
            $method = 'set' . ucfirst($key);

            // Et si il existe dans cet objet, je l'appelle
            if (is_callable([$this, $method])) {
                $this->$method($value);
            }
        }
    }

    // Et les setters font le reste
}
```

L'hydratation

- Rien à voir avec les crèmes, l'hydratation en POO est le fait d'apporter à un objet ce dont il a besoin pour fonctionner, en l'occurrence, les valeurs de ses attributs
- C'est d'ailleurs exactement ce que fait Doctrine si jamais vous avez déjà manipulé Symfony mais je m'égare
- Pour hydrater un objet automatiquement, on va se servir de toutes les conventions de nommage (je vous avais dit que c'était important)

L'hydratation en action

- Maintenant, il ne nous reste plus qu'à mettre ça en forme avec une DB
- PDO::FETCH_ASSOC me retourne les résultats sous forme d'un array associatif uniquement (par défaut les résultats sont en double, une version associée à l'id et une version associée au nom de la colonne)

```
// Je récupère un jeu de données en DB
$query = $db->query('SELECT * FROM personnages WHERE nom = maurice');
$data = $query->fetch(PDO::FETCH_ASSOC); // Oh, une constante !

// Et je laisse l'hydratation faire le reste
$maurice = new Perso($data);
```

1 classe = 1 rôle

- C'est un des adages les plus importants en POO.
- Le but premier de la POO est d'organiser correctement son code pour pouvoir le maintenir et travailler à plusieurs sans se marcher sur les pieds, si on commence à mélanger les choses, on nage à contre courant
- Ceci étant dit, on va vouloir manipuler nos objets en relation avec notre base de données, où doit ont ranger tout cela ? Clairement pas dans nos entités, leur rôle est juste de représenter l'entité
- On va donc avoir recours à des objets « Managers » dont le rôle sera de gérer l'interface avec la DB

```
class NewsManager
{
    // Le manager a juste besoin d'une
    // connexion pour fonctionner
    private $db;

    public function __construct(PDO $db)
    {
        $this->setDb($db);
    }

    public function setDb(PDO $db)
    {
        $this->db = $db;
    }

    public function add(News $news)
    {
        // Je vais ajouter en DB
    }

    public function getAll()
    {
        // Je récupère toutes les news
    }

    // Etc...
}
```

```

class Perso
{
    private $id,
            $nom = 'FooBar',
            $forcePerso,
            $degats,
            $niveau,
            $experience;

    public function __construct()
    {
        echo 'Bonjour ', $this->nom, '<br />';
    }
}

$query = $db->query('SELECT * FROM personnages');
$query->setFetchMode(PDO::FETCH_CLASS, 'Perso');
$persos = $query->fetchAll();
// Dans ce cas, tous mes objets ont déjà leurs attributs
// avec une valeur quand le constructeur est appelé
// Le constructeur peut donc réécrire les données de la DB
=====

$q = $db->query('SELECT * FROM personnages');
$q->setFetchMode(PDO::FETCH_CLASS | PDO::FETCH_PROPS_LATE, 'Perso');
$persos = $q->fetchAll();
// Maintenant le constructeur est appelé à l'initialisation de l'objet,
// Les attributs ont encore leur valeur par défaut, le constructeur
// ne peut donc pas réécrire les données de la DB
// (puisque elles arrivent après)

```

PDO::setFetchMode()

- Une autre façon de récupérer des objets en sortie de DB est avec la méthode setFetchMode() qui va nous permettre de passer plusieurs options :
 - *PDO::FETCH_CLASS* qui va dire à PDO d'aller mettre les résultats trouvés en DB dans un objet
 - *PDO::FETCH_PROPS_LATE* qui va dire à PDO de d'abord instancier le constructeur de l'objet avant de coller les résultats trouvés en DB
 - *Le second argument à passer est le nom de la classe à instancier*
- Le jeu de résultats étant retourné sous forme d'un array d'objets, inutile de passer une option à fetchAll()

L'HÉRITAGE EN POO

Sans passer par un huissier

C'est quoi l'héritage ?

- Pour savoir si une classe B doit hériter d'une classe A, on doit pouvoir dire que B est un A
 - *Une maison ou un appartement sont des bâtiments, on peut donc dire « Maison hérite de bâtiment »*
- C'est un des concepts les plus puissants en POO, il va permettre de transmettre tous les attributs et méthodes d'une classe mère vers une ou des classes filles
- La classe fille pourra alors appeler tous les attributs et méthodes « protected » ou « public » de la classe mère
- Il sera aussi possible d'appeler depuis une instance de la classe fille, tous les attributs et méthodes « public » de la classe mère

L'héritage en action

- Pour déclarer un héritage, on va utiliser le mot clé « extends » dans la déclaration de la classe
- Vous vous souvenez de la visibilité « protected » ? C'est dans ce contexte que elle prend tout son sens

```
class Perso
{
    private $force;
    private $degat;
    private $defense;

    public function frapper(Perso $perso)
    {
        // Pour taper
    }
}

class Magicien extends Perso
{
    // Aura en plus des attributs de Perso
    private $magie;

    // Et en plus des méthodes de Perso
    public function lancerSort(Perso $perso)
    {
        // Un truc avec la magie
    }
}

// Jean-Claude va hériter de la classe Perso
$jeanClaude = new Magicien();
$jeanClaude->lancerSort($michel);
// Et hériter des méthodes
$jeanClaude->frapper($michel);
```

```
class A
{
    public function __construct(string $nom)
    {
        echo '<p>Je suis ' . $nom . '</p>';
    }
}

class B extends A
{
    // Je redéclare la méthode
    public function __construct(string $nom)
    {
        echo '<hr />';
        // Mais je vais exécuter la méthode du parent
        parent::__construct($nom);
        echo '<p>Mais je suis un héritier</p>';
    }
}

$francis = new A('Francis');
$maurice = new B('Maurice');
```

Redéfinir les méthodes

- On aura souvent besoin, soit de réécrire complètement une méthode héritée d'un parent ou alors de redéfinir son code
- Pour cela il suffit simplement de redéclarer la méthode en question
- Pour récupérer la méthode du parent on utilisera le mot clé « parent:: » suivi du nom de la méthode

```
class Mere
{
    public function bonjourMaman()
    {
        echo 'Je suis la classe mère <br />';
    }
}

class Fille extends Mere
{
    public function bonjourFille()
    {
        echo 'Je suis la fille <br />';
    }
}

class PetiteFille extends Fille
{
    public function bonjourPetiteFille()
    {
        echo 'Je suis la petite fille';
    }
}

$a = new PetiteFille();
$a->bonjourMaman();
$a->bonjourFille();
$a->bonjourPetiteFille();
```

Héritage à l'infini

- Il est tout à fait possible de faire des héritages d'héritages d'héritages...
- Les classes « petites filles » hériteront des méthodes et attributs de la classe « fille » qui incluent ceux de la classe « mère »

LES CONTRAINTES

Pour forcer le comportement des classes

Classes abstraites

- Les classes abstraites sont des classes qu'on ne pourra pas instancier directement, on ne pourra pas créer d'objet à partir d'elles
- On sera obligé de créer des enfants de cette classe pour pouvoir créer des objets
- Reprenons notre exemple de personnages, on va créer des magiciens, des paladins... mais jamais de « perso ».On va donc rendre cette classe abstraite

```
// Classe abstraite, on ne peut pas créer
// d'objets directement à partir d'elle
abstract class Perso
{
    private $nom;
    private $force;
    private $degats;

    public function frapper(Perso $perso)
    {
        // Un peu de bagarre
    }
}

class Mage extends Perso
{
    // Les caractéristiques du Mage
}

class Paladin extends Perso
{
    // Celles du Paladin
}

$pierre = new Mage();
$robert = new Paladin();
$robert->frapper($pierre);
```

```
abstract class Mere
{
    // Une méthode abstraite
    abstract public function trucAbstrait();

    public function trucNormal()
    {
        return 'je suis une méthode classique';
    }
}

class Fille extends Mere
{
    // Redéclarée dans la classe fille
    public function trucAbstrait()
    {
        return 'Je dois être redéclarée';
    }
}

$fille = new Fille();
$fille->trucAbstrait();
$fille->trucNormal();
```

Méthodes abstraites

- De la même façon que l'on va avoir besoin de classes qui ne seront jamais instanciées directement, on va avoir besoin de méthodes qui devront être implantées dans chaque classe fille mais dont le contenu sera différent pour chaque enfant
- C'est ce que l'on appelle des méthodes abstraites
- Elles se déclarent comme une méthode classique mais n'ont pas d'accolades puisque elles ne contiennent pas de code
- Elles devront être réécrites dans chaque classe fille sans quoi une erreur fatale sera levée
- Les méthodes abstraites n'existent que dans les classes abstraites

Classes finales

```
abstract class Mere
{
    // Des choses ici
}

// Je déclare ma classe finale
final class Fille extends Mere
{
    // D'autres choses ici
}

// Faire ceci va lever une erreur fatale !
// "Class Foo may not inherit from final class"
class Foo extends Fille
{
    // Rien
}
```

- C'est le concept opposé aux classes abstraites :
On ne peut pas hériter d'une classe finale
- Il est plutôt rare de vouloir déclarer une classe finale

Méthodes finales

- Attention, la logique est ici un peu différente.
- Une méthode finale sera bien héritée par les classes enfants mais ne pourra pas être réécrite

```
class Mere
{
    final public function methodeFinale()
    {
        echo 'Je suis une méthode finale ! <hr />';
    }

    public function foo()
    {
        echo 'Je suis une méthode classique !';
    }
}

class Fille extends Mere
{
    public function foo()
    {
        parent::foo();
        echo '<br />Mais je peux être réécrite...';
    }
}

$fille = new Fille();
$fille->methodeFinale();
$fille->foo();
```

Exercice

- Il est temps de se faire un petit jeu naïf histoire de mettre tout ça en pratique
- Faites un jeu de combat simple sur une page d'accueil :
 - *Vous pouvez créer un nouveau personnage selon ces deux types : magicien ou guerrier et lui donner un nom*
 - *Les guerriers ont 100 points de vie, une attaque aléatoire entre 20 et 40 et une défense aléatoire entre 10 et 19 et ne peuvent que attaquer*
 - *Les magiciens ont 100 points de vie aussi, une attaque entre 5 et 10, 0 de défense mais peuvent endormir un joueur pendant 15 secondes toutes les 2 minutes*

Correction : BaseEntity

- Je vous ai fait [un petit repo disponible ici](#)
- Commençons par le début, toutes mes entités (pas seulement mes personnages) auront probablement besoin d'hydratation, je vais donc inclure ceci dans le constructeur d'une première classe : BaseEntity dont toutes mes classes hériteront
- J'assigne une valeur par défaut pour que cet argument soit optionnel et que puisse quand même utiliser l'option PDO::FETCH_CLASS si je le souhaite

```
abstract class BaseEntity
{
    use Hydrator;

    public function __construct(array $data = [])
    {
        $this->hydrate($data);
    }
}
```

```
abstract class Personnage extends BaseEntity
{
    private $id;
    private $nom = 'John Doe';
    private $pv = 100;
    private $force;
    private $defense;
    protected \DateTime $sleep;

    const IS_SLEEPING = 1;
    const NO_MANA = 2;
    const ITS_ME = 3;
    const NO_DEGATS = 4;
    const DEAD = 5;

    public function __construct(array $data = [])
    {
        ...
    }

    public function attack(Personnage $personnage)
    {
        ...
    }

    protected function defendFrom(Personnage $personnage)
    {
        ...
    }

    public function canAttack(Personnage $personnage)
    {
        ...
    }

    public function getPrettyClass()
    {
        ...
    }

    public function isSleeping():bool
    {
        ...
    }

    // Getters & Setters
}
```

Correction : Entité Personnage

- Je vais ensuite créer une classe Personnage, elle aussi abstraite qui sera le socle commun à tous mes personnages
- Je vais utiliser le pattern des « modèles riches » (en opposition aux modèles anémiques) ce qui veut dire que je vais inclure des méthodes utilitaires dans mon entité afin de faciliter sa manipulation
- J'utilise aussi des constantes pour savoir savoir dans quel état est mon personnage

Correction : classes enfants

```
class Guerrier extends Personnage
{
    public function __construct(array $data = [])
    {
        $this->setForce(random_int(20, 40))
            ->setDefense(random_int(10, 19));

        parent::__construct($data);
    }
}
```

- Comme j'assigne des valeurs par défaut à mes attributs de force et de défense lors de la création d'un nouveau personnage, je le fais dans le constructeur
- MAIS, comme je veux aussi que ces valeurs soient celles inscrites en DB quand je récupère un personnage, je fais en sorte que mon Hydratation vienne écraser ces valeurs, j'appelle le constructeur parent APRÈS

Correction : Manager

- Comme pour mes entités, tous mes manageurs auront besoin d'une instance de connexion à la base de donnée pour fonctionner
- Je vais donc créer une classe abstraite dont hériteront tous mes manageurs et dont le principal boulot sera de récupérer cette connexion

```
abstract class BaseManager
{
    protected \PDO $pdo;

    /**
     * @param ConnectionInterface $pdo
     */
    public function __construct(ConnectionInterface $pdo)
    {
        $this->pdo = $pdo->getConnection();
    }
}
```

Correction : Manager

- Le rôle du manageur spécialisé sera de faire la liaison avec la DB et de fournir les méthodes nécessaires (CRUD) à la manipulation des personnages

```
class PersonnageManager extends BaseManager
{
    public function __construct(ConnectionInterface $pdo){...}

    /**
     * @return Personnage[]
     */
    public function getAllPersonnages(): array{...}

    public function addPersonnage(Personnage $personnage): Personnage{...}

    public function getPersonnageById($id): Personnage{...}

    public function updatePersonnage(Personnage $personnage): bool{...}

    public function deletePersonnageById($id): bool{...}
}
```

```
class PersonnageManager extends BaseManager
{
    public function __construct(ConnectionInterface $pdo)
    {
        parent::__construct($pdo);

        // Si la table n'existe pas...
        $createTable = 'CREATE DATABASE IF NOT EXISTS `'. PDOFactory::DATABASE . '`;
        USE `'. PDOFactory::DATABASE . '`;
        CREATE TABLE IF NOT EXISTS `personnage` (
            `id` int(11) NOT NULL AUTO_INCREMENT,
            `className` varchar(255) NOT NULL,
            `nom` varchar(255) NOT NULL,
            `pv` int(11) NOT NULL,
            `force` int(11) NOT NULL,
            `defense` int(11) NOT NULL,
            `sleep` datetime,
            `lastSpell` datetime,
            PRIMARY KEY (`id`)
        ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;';
        $this->pdo->exec($createTable);
    }
}
```

Précision

- Je réécris la méthode constructeur pour y créer la table nécessaire en DB histoire de pas avoir à le faire dans phpmyadmin

Précision

- J'ai fait assez peu de vérifications quant à l'intégrité des données sur cette correction mais j'utilise quand même des requêtes préparées !
- C'est une habitude indispensable à prendre

```
public function getPersonnageById($id): Personnage
{
    $query = 'SELECT * FROM ' . PDOFactory::DATABASE . '.personnage WHERE id = :id';
    $select = $this->pdo->prepare($query);
    $select->bindValue( param: ':id', $id, type: \PDO::PARAM_INT);
    $select->execute();

    $result = $select->fetch( mode: \PDO::FETCH_ASSOC);
    $className = $result['className'];
    return new $className($result);
}
```

Correction :

Le jeu

- Nous avons nos entités et nos outils pour les gérer, il ne nous reste plus qu'à les faire interagir ensemble
- Le plus simple reste de faire des routes spécifiques au traitement des données :
 - *Une route index avec la liste des personnages et un formulaire pour en créer*
 - *Une route pour ajouter un personnage*
 - *Une route pour supprimer*
 - *Une route pour update*
 - *Une route pour jouer*
 - *Une route pour frapper*
 - *Une route pour lancer des sorts*

Correction

- Prenons par exemple les actions à effectuer en cas d'attaque
- Je vais récupérer les ID des personnages : l'attaquant et la cible via les paramètres en \$_GET
- Je vais récupérer les deux instances depuis la base de données avec le manageur
- Je vais utiliser la méthode d'attaque et en fonction du retour, je vais soit rediriger avec un message (je suis endormi... J'essaye de me frapper moi-même...) soit update les personnages en DB
- Je vais ensuite faire une redirection pour retourner à la page de jeu

```
$personnageManager = DIC::autowire( instance: 'PersonnageManager');
/** @var $personnageManager PersonnageManager */

$myPerso = $personnageManager->getPersonnageById($this->params['id']);
$cible = $personnageManager->getPersonnageById($this->params['cible']);

$result = $myPerso->attack($cible);

if ($result === Personnage::DEAD) {...}

if ($result === Personnage::ITS_ME) {...}

if ($result === Personnage::NO_DEGATS) {...}

if ($result === Personnage::IS_SLEEPING) {...}

$personnageManager->updatePersonnage($cible);
Flash::setFlash('Personnage frappé !');

$this->HTTPResponse->redirect( location: '/play/' . $myPerso->getId());
```



Précision

- Nous vous occupez pas de la forme de certaines méthodes, j'ai utilisé un petit framework fait maison pour cette correction afin de gagner du temps
- **DIC::autowire** est un raccourci pour créer mon instance de manager
- « **\$this->params** » est strictement identique à **\$_GET**
- **Flash::setFlash** crée un message dans **\$_SESSION** pour le lire sur une autre page
- « **\$this->HTTPResponse->redirect** » est un raccourci pour écrire « `header('Location...')` »

SELF::
STATIC::
\$THIS

3 salles, 3 ambiances

```
class Mere
{
    public static function qui()
    {
        echo __CLASS__;
    }

    public static function test()
    {
        self::qui();
    }
}

class Fille extends Mere
{
    public static function qui()
    {
        echo __CLASS__;
    }
}

Fille::test(); // Va afficher "Mère"

// En effet, la fonction qui() est bien contenue
// dans la classe Mère, même si elle est appellée
// depuis la classe Fille
```

Résolution statique à la volée | self::

- On va se faire quelques nœuds au cerveau
- Rappelons que « self:: » renvoi à la classe dans laquelle il est contenu
- Dans le cas suivant, on va donc voir s'afficher « Mère »

```
class Mere
{
    public static function qui()
    {
        echo __CLASS__;
    }

    public static function test()
    {
        // J'utilise le mot clé static::
        static::qui();
    }
}

class Fille extends Mere
{
    public static function qui()
    {
        echo __CLASS__;
    }
}

Fille::test(); // Va afficher "Fille"

// Cette fois ci, c'est la classe Fille qui est
// exécutée et donc static fait référence à elle
```

Résolution statique à la volée | static::

- On va maintenant introduire le mot clé « static:: » qui va faire référence à la classe qui a été appelée durant l'exécution
- Cette fois, c'est bien « Fille » qui sera affiché

static:: dans un contexte non statique

- Bien que déprécié, il est possible d'utiliser « static:: » dans un contexte non statique
- Le comportement pourra donc différer de celui de \$this dans certains cas

```
class A
{
    private function foo()
    {
        echo "success! <br />";
    }
    public function test()
    {
        $this->foo();
        static::foo();
    }
}

class B extends A
{
    // foo() sera copiée dans B, par conséquent
    // son contexte sera toujours A
    // et l'appel se fera sans problème
}

class C extends A
{
    private function foo()
    {
        // La méthode originale est remplacée;
        // le contexte est celui de C
    }
}

$b = new B();
$b->test();      // Succes | Succes
$c = new C();
$c->test();      // Succes | Fatal Error
```

```
class A
{
    public static function foo()
    {
        static::qui();
    }

    public static function qui()
    {
        echo __CLASS__ . "<br />";
    }
}

class B extends A
{
    public static function test()
    {
        A::foo();
        parent::foo();
        self::foo();
    }

    public static function qui()
    {
        echo __CLASS__ . "<br />";
    }
}

class C extends B
{
    public static function qui()
    {
        echo __CLASS__ . "<br />";
    }
}

C::test(); // Affichera A | C | C
```

Et parent:: dans tout ça ?

- Histoire de finir, un dernier exemple issu de la doc php.net
- A::foo() est très direct, il appelle la méthode depuis un contexte défini
- Parent:: ou self:: eux, vont transmettre l'information appelante et static:: va donc déterminer que l'appel provient de C
- (et ne prêtez pas attention aux vagues rouge, intelephense est devenu fou)

LES MÉTHODES MAGIQUES

Et la surcharge de propriétés et de méthodes

C'est quoi ces méthodes ?

- Ce sont des méthodes conçues pour être appelées automatiquement lors d'un évènement particulier
- Si la méthode n'existe pas dans votre classe et que l'évènement est exécuté, rien de spécial ne se passera
- Dans cette catégorie de méthodes, nous avons déjà vu `__construct()` qui va s'exécuter à l'instanciation d'un objet

__destruct()

- C'est le partenaire de __construct(), cette méthode est appelée à la destruction d'un objet
- Elle peut être pratique pour effacer quelque chose de la base de données par exemple

```
class Test
{
    public function __construct()
    {
        echo 'coucou le constructeur<br>';
    }

    public function __destruct()
    {
        echo 'je démonte tout<br>';
    }
}

// Je lance __construct()
$a = new Test();

// Lancera __destruct()
unset($a);
```

__set(\$nom, \$valeur)

- Cette méthode est appelée automatiquement quand on essaye d'attribuer une valeur à un attribut qui n'existe pas ou auquel on n'a pas accès
- Il lui sera passé automatiquement deux arguments :
 - *Le nom de l'attribut que l'on cherche à modifier*
 - *La valeur que l'on cherche à lui attribuer*

```
class Test
{
    private $attributPrive;

    public function __set($name, $value)
    {
        echo sprintf('Impossible d\'attribuer la valeur %s à l\'attribut %s <br/>', $value, $name);
    }
}

$a = new Test();
$a->test = 'Foo'; // Impossible d'attribuer la valeur Foo à l'attribut test
$a->attributPrive = 'Bar'; // Impossible d'attribuer la valeur Bar à l'attribut attributPrive
```

__get(\$nom)

```
class Test
{
    private $attributPrive;

    public function __get($name)
    {
        echo sprintf('Impossible d\'accéder à l\'attribut %s <br/>', $name);
    }
}

$a = new Test();
$a->test; // Impossible d'accéder à l'attribut test
$a->attributPrive; // Impossible d'accéder à l'attribut attributPrive
```

- Semblable à __set(), cette fonction sera appelée quand on cherchera à accéder à un attribut inexistant ou auquel on n'a pas accès
- Un seul argument lui est envoyé automatiquement : le nom de l'attribut auquel on cherche à accéder

__isset(\$name)

- Toujours dans la même logique, cette méthode sera appelée quand on va appeler la fonction isset() sur un attribut inexistant ou auquel on n'a pas accès
- Cette méthode doit renvoyer un booleen

```
class Test
{
    private $attributPrive = 12;

    public function __isset($name): bool
    {
        return isset($this->$name);
    }
}

$a = new Test();
var_dump(isset($a->test)); // false
var_dump(isset($a->attributPrive)); // true
```

__unset(\$name)

- Evidemment on va aussi trouver cette méthode qui va être appelée quand on appelle unset() sur un attribut qui n'existe pas ou auquel on n'a pas accès
- Si on combine le tout ça peut donner quelque chose comme ça

```
class Test
{
    private $attributs = [];

    public function __set($name, $value)
    {
        $this->attributs[$name] = $value;
    }

    public function __get($name)
    {
        if (isset($this->attributs[$name])) {
            return $this->attributs[$name];
        }
    }

    public function __isset($name)
    {
        return isset($this->attributs[$name]);
    }

    public function __unset($name)
    {
        if (isset($this->attributs[$name])) {
            unset($this->attributs[$name]);
        }
    }
}

$a = new Test();

$a->francis = 'test';
echo $a->francis; // test
var_dump( isset($a->francis) ); // true
unset($a->francis);
var_dump( isset($a->francis) ); // false
```

```
class MethodTest
{
    public function __call($name, $arguments)
    {
        // Note : la valeur de $name est sensible à la casse.
        echo "Appel de la méthode '$name' "
            . implode(', ', $arguments) . "<br />";
    }

    public static function __callStatic($name, $arguments)
    {
        // Note : la valeur de $name est sensible à la casse.
        echo "Appel de la méthode statique '$name' "
            . implode(', ', $arguments) . "<br />";
    }
}

$obj = new MethodTest;
$obj->runTest('dans un contexte objet');

MethodTest::runTest('dans un contexte static');
```

__call() & __callStatic()

- Nous avons vu les surcharges de propriétés, passons aux surcharges de méthodes
- Ces méthodes vont donc être appelées quand on va vouloir appeler une méthode qui n'existe pas ou à laquelle on n'a pas accès
- __call(\$name, \$arguments) pour une méthode standard
- __callStatic(\$name, \$arguments) pour une méthode statique
- \$arguments est un array

__toString()

- On commence à rentrer dans les méthodes magiques un peu étranges qui sont appelées quand on commence à vouloir tordre les objets
- Celle-ci est appelée quand on cherche à convertir l'objet en chaîne de caractères (avec un echo par exemple)
- Cette situation se rencontre des fois en gestion d'erreurs

```
class Test
{
    private $text;

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

    public function __toString()
    {
        return $this->text;
    }
}

$test = new Test('Raconte moi des choses');
echo $test;
```

__invoke(...\$values)

```
class Test
{
    public function __invoke(string $text)
    {
        echo strrev($text);
    }
}

$test = new Test();
$test('Je suis du texte'); // etxet ud sius ej
```

- On va terminer dans le très étrange, cette méthode sera appelée quand on va vouloir utiliser notre objet comme une fonction
- Elle prend autant d'arguments que vous le souhaitez

PARCOURIR UN OBJET

Un peu à la façon d'un array

```

class MaClasse
{
    public $attributPublic = 'Attribut public';
    protected $attributProtege = 'Attribut protégé';
    private $attributPrive = 'Attribut privé';

    function listeAttributs()
    {
        foreach ($this as $attribut => $valeur) {
            // Directement depuis la classe parent
            echo $attribut, ' => ', $valeur, '<br />';
        }
    }
}

class Enfant extends MaClasse
{
    function listeAttributs()
    {
        // Redéclaration de la fonction pour que ce ne
        // soit pas celle de la classe mère qui soit appelée.
        foreach ($this as $attribut => $valeur) {
            echo $attribut, ' => ', $valeur, '<br />';
        }
    }
}

$classe = new MaClasse;
$enfant = new Enfant;

$classe->listeAttributs();
$enfant->listeAttributs();

foreach ($classe as $attribut => $valeur) {
    echo $attribut, ' => ', $valeur, '<br />';
}

```

Parcourir un objet

- La boucle `foreach` permet de parcourir un objet avec une syntaxe assez semblable à celle des arrays, selon l'endroit où la boucle se trouvera, elle ne sortira pas les mêmes résultats:
 - Exécutée depuis l'intérieur de la classe, tous les attributs seront lus (`public`, `protected` et `private`)
 - Exécutée depuis une classe enfant, seuls les attributs `public` et `protected` seront affichés
 - Enfin, depuis l'extérieur de la classe, seuls les éléments `public` seront affichés

LES INTERFACES

Pour dicter un comportement précis

C'est quoi une interface ?

- Quand on manipule un objet, on est pas toujours certain que certaines méthodes existent, pour imposer une structure à nos classes, il existe un outil : les interfaces
- Les interfaces sont techniquement des classes totalement abstraites (que l'on ne peut pas invoquer et dont on est obligés de réécrire les méthodes)
- Il ne faut cependant pas confondre les interfaces avec l'héritage, là où une classe fille est un sous ensemble de sa classe mère, les interfaces sont juste des points communs :
 - *On ne peut pas dire que une voiture et un personnage peuvent descendre du même héritage mais les deux peuvent quand même bouger*

Comment ça marche ?

- On utilise le mot clé « implements »
- On devra obligatoirement réécrire les méthodes sans quoi une erreur fatale sera levée
- Toutes les méthodes dans une interface doivent être publiques
- 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

```
interface Movable {
    public function move($lieu);
}

class Personnage implements Movable {
    public function move($lieu) {
        // Des trucs
    }
}

class Vehicule implements Movable
{
    public function move($lieu)
    {
        // D'autres trucs
    }
}
```

```
class Magicien extends Personnage implements Movable {  
    public function move($lieu) {  
        // toujours des trucs ici  
    }  
}  
  
//=====  
  
interface iA {  
    public function test1();  
}  
  
interface iB {  
    public function test2();  
}  
  
class A implements iA, iB {  
    // Pour ne générer aucune erreur, il va falloir  
    // réécrire les méthodes de iA et de iB.  
  
    public function test1() {  
        // des trucs  
    }  
  
    public function test2() {  
        // Plus de trucs  
    }  
}
```

Plus sur les interfaces

- Il est possible de cumuler héritage et interfaces, on déclarera d'abord les héritages puis les interfaces
- Il est également possible d'avoir plusieurs interfaces sur un même objet tant que aucune méthode ne rentre en conflit

Les constantes d'interface

- Il est possible d'implémenter des constantes dans les interfaces
- Elles fonctionneront comme des constantes de classe classiques

```
interface iInterface
{
    const MA_CONSTANTE = 'Hello !';
}

echo iInterface::MA_CONSTANTE; // Affiche Hello !

class MaClasse implements iInterface
{
    // Je ne l'ai même pas redéclarée ici
}

echo MaClasse::MA_CONSTANTE; // Affiche Hello !
```

```
interface iA
{
    public function test1();
}

interface iB extends iA
{
    // Erreur fatale : impossible de réécrire cette méthode.
    public function test1($param1, $param2);
}

//=====

interface iiA
{
    public function test1();
}

interface iiB
{
    public function test2();
}

interface iiC extends iiA, iiB
{
    public function test3();
}

// La classe implémentant l'interface iiC devra avoir
// les 3 méthodes : test1(), test2() et test3()
```

Faire hériter ses interfaces

- Il est tout à fait possible d'avoir un système d'héritage entre interfaces
- Contrairement aux classes :
 - *Une interface ne peut pas réécrire une méthode ou un attribut dont elle a hérité*
 - *Une interface peut hériter de plusieurs interfaces*

INTERFACES PRÉDÉFINIES

Pour ne pas réécrire ce qui existe déjà

```

class MaClasseITR implements Iterator
{
    private $position = 0;
    private $tableau = [1, 2, 3, 4, 5];

    // Retourne l'élément courant du tableau.
    public function current()
    {
        return $this->tableau[$this->position];
    }

    // Retourne la clé actuelle (c'est la même que la position dans notre cas).
    public function key()
    {
        return $this->position;
    }

    // Déplace le curseur vers l'élément suivant.
    public function next()
    {
        $this->position++;
    }

    // Remet la position du curseur à 0.
    public function rewind()
    {
        $this->position = 0;
    }

    // Permet de tester si la position actuelle est valide.
    public function valid()
    {
        return isset($this->tableau[$this->position]);
    }
}

```

Iterator

- L'interface « Iterator » nous permet de modifier le comportement d'un objet lorsqu'il est parcouru (par un foreach par exemple). Cette interface comporte 5 méthodes :
 - *current()* : renvoie l'élément courant
 - *key()* : retourne la clé de l'élément courant
 - *next()* : déplace le pointeur vers l'élément suivant
 - *rewind()* : remet le pointeur sur le premier élément
 - *valid()* : vérifie si la position courante est valide
- La boucle foreach va tour à tour déclencher les méthodes *current()* et *key()* puis *next()* et s'arrêter dès que *valid()* retournera false

SeekableIterator

```
class MaClasseSITR implements SeekableIterator
{
    private $position = 0;
    private $tableau = [1, 2, 3, 4, 5];

    public function current() { /*...*/ }
    public function key() { /*...*/ }
    public function next() { /*...*/ }
    public function rewind() { /*...*/ }
    public function valid() { /*...*/ }

    public function seek($position)
    {
        $anciennePosition = $this->position; // Je sauvegarde l'ancienne position
        $this->position = $position; // Je set la nouvelle
        if (!$this->valid()) {
            trigger_error('La position spécifiée n\'est pas valide', E_USER_WARNING);
            $this->position = $anciennePosition;
        }
    }

    $objet = new MaClasseSITR;

    $objet->seek(7);
    echo '<br />', $objet->current();
```

- Cette interface hérite de Iterator, il est donc inutile d'implémenter les deux
- Elle rajoute juste la méthode « seek » qui permet de définir la position du curseur
- Il est conseillé de ne pas directement régler l'attribut \$position mais de vérifier avant que la position est valide

ArrayAccess

- Grâce à cette interface on va pouvoir transformer notre objet en tableau (et même l'appeler avec des crochets à la fin)
- Cette interface amène donc 4 méthodes :
 - *offsetExists(\$key) : appelée quand on fait isset(\$objet[\$key]) ou empty(\$objet[\$key])*
 - *offsetGet(\$key) : appelée quand on va taper : \$objet[\$key]*
 - *offsetSet(\$key, \$value) : appelé quand on va faire : \$objet[\$key] = \$value*
 - *offsetUnset(\$key) : appelé quand on va faire : unset(\$objet[\$key])*

```
class MaClasse implements ArrayAccess
{
    private $tableau = [1, 2, 3, 4, 5];

    public function offsetExists($key): bool
    {
        return isset($this->tableau[$key]);
    }

    public function offsetGet($key)
    {
        return $this->tableau[$key];
    }

    public function offsetSet($key, $value): void
    {
        $this->tableau[$key] = $value;
    }

    public function offsetUnset($key): void
    {
        unset($this->tableau[$key]);
    }
}

$objet = new MaClasse;
echo $objet[2]; // 3
$objet[2] = 'Hello world !';
echo $objet[2]; // Hello world !
unset($objet[3]);

echo isset($objet[3]) ? '$objet[3] existe' : '$objet[3] n\'existe plus !';
// $objet[3] n'existe plus !
```

Countable

```
class MaClasseCount implements Countable
{
    private $tableau = [1, 2, 3, 4, 5];

    public function count(): int
    {
        return count($this->tableau);
    }
}

$objet = new MaClasseCount;

echo count($objet) . ' éléments dans l\'objet';
```

- Ca commence à ressembler à toutes les fonctions d'un array tout ça, il ne manque que cette interface qui va rajouter la méthode count()
- Elle doit renvoyer un entier et sera appelée quand on fera count(\$objet)

ArrayIterator

- Cette interface est en fait un regroupement des 4 interfaces précédentes, elle implémente exactement toutes les méthodes vues précédemment
- Elle rajoute également un constructeur qui accepte un array comme argument

LES EXCEPTIONS

Pour mieux gérer les erreurs

Nos propres Exceptions

- Grâce à cet outil on va pouvoir créer nos propres types d'erreurs mais en plus on va pouvoir les "attraper" pour pouvoir les afficher correctement plutôt que de taper une fatal error
- On peut lancer une Exception de n'importe où dans le code, lancer une exception est en fait instancier un objet de la classe Exception. Cet objet contiendra le message et le code d'erreur
- Même si le message d'erreur est facultatif, il est important de le spécifier sinon... lancer une exception ne sert pas à grand chose
- Cet objet Exception va prendre 3 arguments facultatifs en entrée :
 - *Le message*
 - *Le code d'erreur*
 - *L'exception précédente*

Lancer une Exception

- On lance une Exception avec le mot clé « throw »
- Si l'exception n'est pas attrapée, elle va générer une erreur fatale dans laquelle on aura des informations sur ce qui a déclenché l'exception et d'où l'exception provient
- On aura même le droit d'avoir notre message d'erreur

```
function additionner($a, $b): float
{
    // Si $a ou $b ne sont pas des chiffres
    if (!is_numeric($a) || !is_numeric($b)) {
        // On lance une nouvelle exception grâce à throw et
        // on instancie directement un objet de la classe Exception.
        // Le code d'erreur ici ne correspond à rien de particulier
        throw new Exception('Les deux paramètres doivent être des nombres', 2478687);
    }

    return $a + $b;
}

echo additionner(12.7, 3), '<br />';           // 15.7
echo additionner('azerty', 54), '<br />';      // Fatal error: Uncaught Exception...
echo additionner(4, 8);                         // Pas exécuté
```

try / catch

```
function additionner($a, $b): float
{
    if (!is_numeric($a) || !is_numeric($b)) {
        throw new Exception('Les deux paramètres doivent être des nombres', 2478687);
    }

    return $a + $b;
}

try {
    echo additionner(12.7, 3), '<br />';
    echo additionner('azerty', 54), '<br />'; // Exception lancée ici
    echo additionner(4, 8);
} catch (Exception $err) {
    echo 'Message d\'erreur : ' . $err->getMessage() . ' | Code d\'erreur : ' . $err->getCode();
    // getMessage() récupère le premier argument passé dans le throw new Exception
    // getCode() récupère le second (le code d'erreur)
}

// Le reste du bloc try n'est pas exécuté à cause de l'Exception
// Mais la suite du code, si
echo 'Je suis executé normalement';
```

- Le bloc try / catch va nous permettre d'attraper les Exceptions
- Il est important de comprendre que quand une Exception est lancée, elle va « remonter » à la recherche d'un bloc try / catch et déclencher une fatal error si elle n'en trouve pas
- À la suite de catch, on va spécifier le type d'exception que l'on cherche à attraper et le nom de l'objet que l'on va créer à partir de cette classe
- On va ensuite récupérer le message et le code d'erreur grâce à des méthodes de la classe
- Si une Exception est attrapée, elle va interrompre le code dans le bloc try mais reprendre la suite du code

Une exception custom

- Exception n'est pas une classe finale, il est donc possible d'en hériter et de réécrire quelques unes de ses méthodes
- Voilà la tronche globale de la classe Exception
- Tout est final sauf le constructeur et toString

```
Exception implements Throwable {  
  
    /* Propriétés */  
    protected string $message;  
    protected int $code;  
    protected string $file;  
    protected int $line;  
  
    /* Méthodes */  
    public __construct(string $message = "", int $code = 0, Throwable|null $previous = null)  
    final public getMessage(): string  
    final public getPrevious(): Throwable|null  
    final public getCode(): int  
    final public getFile(): string  
    final public getLine(): int  
    final public getTrace(): array  
    final public getTraceAsString(): string  
    public __toString(): string  
    final private __clone(): void  
}
```

Une exception custom

```
class MonException extends Exception
{
    public function __construct($message, $code = 0)
    {
        parent::__construct($message, $code);
    }
    public function __toString()
    {
        return $this->message;
    }
}
```

- Je peux tout à fait imaginer créer un nouveau type d'Exception qui rendrait le message d'erreur obligatoire et qui retournerait le message grâce à `toString()`

De retour dans notre code

- Je peux maintenant réécrire mon petit bout de code avec mon exception fraîchement créée

```
class MonException extends Exception
{
    public function __construct($message, $code = 0)
    {
        parent::__construct($message, $code);
    }
    public function __toString()
    {
        return $this->message;
    }

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

        return $a + $b;
    }

    try {
        echo additionner(12.7, 3), '<br />';
        echo additionner('azerty', 54), '<br />';
        echo additionner(4, 8);
    } catch (MonException $err) {
        echo $err;
    }
}
```

Petite précision

- Notez que comme « MonException » hérite de la classe Exception, elle aurait été tout aussi bien attrapée par un catch visant la classe Exception
- En revanche, un bloc catche visant la classe « MonException » n'attrapera pas une Exception standard
- Ce mécanisme nous permet une meilleure spécificité dans la gestion d'erreur

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

    return $a + $b;
}

try {
    echo additionner(12.7, 3), '<br />';
    echo additionner('azerty', 54), '<br />';
    echo additionner(4, 8);
} catch (Exception $err) { // Fonctionne
    echo $err;
}
=====

function additionner2($a, $b): float
{
    if (!is_numeric($a) || !is_numeric($b)) {
        throw new Exception('Les deux paramètres doivent être des nombres');
    }

    return $a + $b;
}

try {
    echo additionner2(12.7, 3), '<br />';
    echo additionner2('azerty', 54), '<br />';
    echo additionner2(4, 8);
} catch (MonException $err) { // Ne fonctionne pas
    echo $err;
}
```

Emboiter des catch

- Si vous voulez effectuer des actions différentes en fonction du type d'erreur dans un même bloc try / catch, il est tout à fait possible d'emboiter des catch
- PHP ira d'abord dans le premier bloc catch pour voir si ça marche, puis dans le second et ainsi de suite
- Pensez donc à mettre les catch les plus spécifiques en premier

```
try {
    // Un truc
} catch (MonException $e) {
    echo $e;
} catch (Exception $e) {
    echo $e->getMessage();
}
```

Les exceptions prédefinies

- Il existe [beaucoup d'exceptions prédefinies en PHP](#), donc plutôt que de forcement chercher à créer son exception custom ou, de toujours jeter une exception générique, il peut être intéressant de jeter une exception spécifique du cas rencontré
- Dans notre exemple d'additions, on pourrait lancer un exception spécifique de la situation "argument ne correspond pas au type attendu"

```
function newAdditionner($a, $b): float
{
    if (!is_numeric($a) || !is_numeric($b)) {
        // On lance exception prédefinie spécifique
        throw new InvalidArgumentException('Mauvais type d\'argument');
    }

    return $a + $b;
}
```

Le bloc finally

```
function add($a, $b): float
{
    if (!is_numeric($a) || !is_numeric($b)) {
        // On lance exception prédéfinie spécifique
        throw new Exception('Mauvais type d\'argument');
    }
    return $a + $b;
}

try {
    echo add('francis', 2), '<br />';
} catch (InvalidArgumentException $e) {
    echo $e->getMessage(), '<br />';
} finally {
    echo 'Je suis exécuté quoi qu\'il arrive, erreur fatale ou pas <br />';
}

echo 'C\'est la fin du script <br />';
```

- Ce bloc sera exécuté quoi qu'il arrive, que tout se passe bien, qu'une Exception soit attrapée ou qu'une erreur fatale soit levée
- On s'en sert souvent pour faire des opérations de nettoyage comme la fermeture d'un fichier ou d'une connexion pour s'assurer que tout soit propre même en cas d'erreur imprévue

LES TRAITS

Importer des méthodes sans héritage

C'est quoi un trait ?

- En PHP, une classe ne peut hériter que d'une seule autre classe, les traits vont nous servir à contourner cette limitation et éviter de la duplication de code
- Imaginons le problème suivant : J'ai deux classes « Writer » et « Mailer », l'une va écrire dans un fichier et l'autre envoyer un mail. Dans les deux cas je vais vouloir que mon texte soit formaté en HTML pour être plus lisible
- Je risque donc d'avoir la partie « formateur » qui sera dupliqué dans les deux classes
- Un trait est un bout de code que nous allons pouvoir importer dans une classe et nous en servir comme si nous en avions hérité

En action

- Pour déclarer un trait, on utilise le mot clé « trait »
- C'est une sorte de mini-classe dans laquelle on va pouvoir déclarer des méthodes et des attributs
- Pour utiliser un trait dans une classe on utiliser le mot clé « use »

```
trait MonTrait
{
    // Un trait est une sorte de mini classe,
    // on va y définir nos méthodes
    public function hello()
    {
        echo 'Hello world !';
    }
}

class A
{
    // Avec le mot clé use, je vais
    // pouvoir utiliser ce trait
    use MonTrait;
}

class B
{
    // Et ceci dans deux classes indépendantes
    use MonTrait;
}

$a = new A;
$a->hello(); // Affiche "Hello world !"

$b = new B;
$b->hello(); // Affiche aussi "Hello world !"
```

```
trait t1
{
    // des choses
}

trait t2
{
    // d'autres choses
}

class A
{
    // Il est possible d'utiliser plusieurs
    // traits, il suffit de les déclarer
    // séparés par des virgules
    use t1, t2;
}
```

Plusieurs traits

- Il est tout à fait possible d'utiliser plusieurs traits dans une même classe

Un trait dans un trait

- Il est également possible d'utiliser un trait dans un trait
- La classe utilisant le trait « final » récupérera toutes les méthodes

```
trait t1
{
    public function bonjour()
    {
        echo 'Bonjour ! <br />';
    }
}

trait t2
{
    // J'utilise un trait dans un trait
    use t1;

    public function salut()
    {
        echo 'Salut ! <br />';
    }
}

class A
{
    // Ma classe aura donc les deux méthodes
    use t2;
}

$a = new A();
$a->bonjour();
$a->salut();
```

```
trait t1
{
    public function bonjour()
    {
        echo 'Je suis le Trait 1';
    }
}

trait t2
{
    public function bonjour()
    {
        echo 'Je suis le Trait 2';
    }
}

class A
{
    use t1, t2 {
        t1::bonjour insteadof t2;
    }
}

$a = new A();
$a->bonjour(); // Je suis le Trait 1
```

Résoudre les conflits

- Si on cherche à utiliser deux traits qui ont des méthodes avec le même nom, une erreur fatale sera levée
- Pour éviter cela, il faut préciser quelle méthode utiliser

```
trait t1
{
    public function bonjour()
    {
        echo 'Bonjour !';
    }
}

class A
{
    use t1 {
        bonjour as private;
    }

    public function ditBonjour()
    {
        echo $this->bonjour();
    }
}

$a = new A();
$a->ditBonjour();
\$a->bonjour\(\); // Fatal Error
```

Changer la visibilité

- Il est possible de changer la visibilité d'une méthode importée en utilisant un trait

La classe avant le trait

- Si vous utilisez un trait et que une de vos méthodes de classe porte le même nom, la méthode de classe sera automatiquement utilisée

```
trait t1
{
    public function bonjour()
    {
        echo 'Je suis le Trait';
    }
}

class A
{
    use t1;

    public function bonjour()
    {
        echo 'Je suis la classe';
    }
}

$a = new A();
$a->bonjour(); // Je suis la classe
```

```
trait t1
{
    public function bonjour()
    {
        echo 'Je suis le Trait';
    }
}

class A
{
    public function bonjour()
    {
        echo 'Je suis le parent';
    }
}

class B extends A
{
    use t1;
}

$b = new B();
$b->bonjour(); // Je suis le Trait
```

Le trait avant le parent

- Par contre, si votre classe utilise un trait et hérite en même temps d'un parent et que les deux possèdent des méthodes avec le même nom, les méthodes du trait seront utilisées

Méthodes abstraites et traits

- Pour en finir sur les traits, il est possible d'importer des méthodes abstraites avec un trait et donc forcer la classe utilisant le trait à la redéfinir
- Si vous importez une méthode abstraite avec un trait dans une classe abstraite, ça sera alors aux classes enfant de redéfinir la méthode

```
trait t1
{
    abstract public function bonjour();
}

class A
{
    use t1;

    public function bonjour()
    {
        // Obligation de redéfinir la méthode
    }
}

// Classe abstraite
abstract class Foo
{
    use t1;
}

// classe héritant du trait avec la méthode abstraite
class Bar extends Foo
{
    public function bonjour()
    {
        // Redéfinition de la méthode
    }
}
```

LES DESIGN PATTERNS

Parce-que certains problèmes ont déjà des solutions

C'est quoi un design pattern ?

- Au fil des années, les développeurs se sont rendu compte que certaines problématiques étaient plutôt récurrentes
- Pour éviter de devoir passer des jours à trouver une solution innovante à des problèmes communs, la communauté a adopté des moyens de conceptions standardisés pour les résoudre

Le pattern Factory

- Le but de ce pattern est de ne plus avoir de « new » à placer dans la partie globale du script mais plutôt de déléguer cette tache à une classe qui le fera pour vous
- L'avantage est de centraliser la création des objets dans une usine pour faciliter l'implémentation ou la modification de fonctions
- Un exemple concret est une usine pour les connexion en base de données, vos identifiants de connexion sont centralisés dans l'usine pour tout votre script

```
class PDOFactory
{
    public static function getMysqlConnexion()
    {
        $db = new PDO('mysql:host=localhost;dbname=madb', 'root', 'root');
        $db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

        return $db;
    }

    public static function getPgsqlConnexion()
    {
        $db = new PDO('pgsql:host=localhost;dbname=madb', 'root', 'root');
        $db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

        return $db;
    }

    // et ainsi passer votre instance de DB avec
    PDOFactory::getMysqlConnexion();
}
```

Le pattern Observer

- La problématique est la suivante : vous avez plusieurs objets qui interagissent entre eux avec les autres et quand un des objets fait une action, certains autres doivent réagir. On va avoir un objet observé (le sujet) et un ou plusieurs objets observateurs.
- Chacun de ces objets va implémenter une interface pré définie SPL
- SplSubject (pour le sujet observé) contient 3 méthodes :
 - *attach(SplObserver \$observer)* : *Elle sert à ajouter un objet observateur à notre observé*
 - *detach(SplObserver \$observer)* : *Qui fait exactement le contraire, elle détache un observateur*
 - *notify()* : *C'est la méthode appelée pour prévenir les observateurs*
- SplObserver (pour les objets qui observent) elle ne contient qu'une méthode :
 - *update(SplSubject \$subject)* : *qui est appelée quand un notify est reçu*

Le pattern Observer

```
class Sujet implements SplSubject
{
    private $observers = [];

    public function attach(SplObserver $observer)
    {
        $this->observers[] = $observer;
    }

    public function detach(SplObserver $observer)
    {
        if (is_int($key = array_search($observer, $this->observers, true))) {
            unset($this->observers[$key]);
        }
    }

    public function notify()
    {
        foreach ($this->observers as $observer) {
            $observer->update($this);
        }
    }

    public function test()
    {
        $this->notify();
    }
}
```

```
class Observateur implements SplObserver
{
    public function update(SplSubject $sujet)
    {
        echo 'Je vous ai entendu !';
    }
}

$sujet = new Sujet();
$observateur = new Observateur();
$sujet->attach($observateur);
$sujet->test();
```

Le pattern Observer

- Ce genre de pattern peut être particulièrement utile si on veut envoyer un email de log lors d'une erreur ou une notification de connexion à l'utilisateur
- La fonction `notify()` passant automatiquement l'objet Sujet à la méthode `update()` de l'objet observateur, il est aisément de passer un message de log ou n'importe quoi
- Dans notre exemple un des observateurs serait une classe `Mailer` chargée d'envoyer un mail quand elle reçoit un `update`

L'injection de dépendances

- Lui est un concept un peu plus compliqué à comprendre mais qui est réellement très puissant
- Il repose sur l'inversion du contrôle des dépendances en passant au service « final » les objets dont il a besoin pour fonctionner dans son constructeur et en s'assurant que les méthodes nécessaires dans ses dépendances soient disponibles grâce à des interfaces
- Le but de ce pattern est de rendre les objets indépendants et réutilisables dans d'autres contextes

```
// Sans injection de dépendances
class GoogleMaps
{
    public function getCoordinatesFromAddress($address)
    {
        // calls Google Maps webservice
    }
}
class OpenStreetMap
{
    public function getCoordinatesFromAddress($address)
    {
        // calls OpenStreetMap webservice
    }
}

class StoreService
{
    public function getStoreCoordinates($store)
    {
        $geolocationService = new GoogleMaps();
        // or $geolocationService = GoogleMaps::getInstance() if you use singletons

        return $geolocationService->getCoordinatesFromAddress($store->getAddress());
    }
}

// Si je voulais utiliser OpenStreetMap plutôt que GoogleMaps, je devrais changer
// le code dans StoreService et toutes les autres classes qui utilisent GoogleMaps...
// Si j'en ai 400 ça peut être long !
```

```
class StoreService
{
    private $geolocationService; // J'ai besoin de cet objet pour fonctionner

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

    public function getStoreCoordinates($store)
    {
        return $this->geolocationService->getCoordinatesFromAddress($store->getAddress());
    }
}

// Je m'assure que mes services aient le comportement que je veux grâce à une interface
interface GeolocationService
{
    public function getCoordinatesFromAddress($address);
}

class GoogleMaps implements GeolocationService
{
    public function getCoordinatesFromAddress($address) { /*...*/ }
}

class OpenStreetMap implements GeolocationService
{
    public function getCoordinatesFromAddress($address) { /*...*/ }
}
```

L'injection de dépendances

- Grâce à l'injection de dépendances, mes objets redeviennent indépendants et mon objet StoreService peut être réutilisé avec plusieurs services de géolocalisation différents

LES NAMESPACES

Pour que tout soit bien rangé

C'est quoi les namespaces ?

- Le problème des classes, c'est qu'elles ont un nom et que dans un gros programme, le choix des noms devient vite limité, et si on commence à utiliser les librairies, c'est pire, on va forcément se retrouver avec une classe qui porte le même nom que la notre... et c'est le plantage
- Pour éviter ça, on va utiliser des namespaces. Ils fonctionnent comme des sortes de dossiers virtuels, généralement on fait en sorte que le namespace corresponde au dossier dans lequel il se trouve mais ce n'est pas obligatoire

Créer un namespace

- Je déclarer le namespace de ma classe avec le mot clé « namespace »
- Dans un autre fichier quand je vais demander de charger la classe, il faudra aussi que je précise le namespace complet

```
namespace Acme\Tools;

class Foo
{
    public function doAwesomeThings()
    {
        echo 'Hello Foo !';
    }
}
```

```
require 'BrouillonPOO.php';

// Dans un autre fichier, si je veux
// appeler ma classe je vais devoir
// écrire son namespace

$foo = new \Acme\Tools\Foo();
$foo->doAwesomeThings();
```

Eviter de se répéter

```
use Acme\Tools\Foo;  
  
$foo = new Foo();  
$foo->doAwesomeThings();
```

- Il existe un moyen simple pour éviter de se répéter, c'est de déclarer les namespaces que l'on va utiliser en haut de fichier
- Il faudra utiliser un « use » différent pour chaque classe que vous voudrez utiliser

Utiliser des alias

```
use Acme\Tools\Foo as SomeFooClass;  
  
$foo = new SomeFooClass();  
$foo->doAwesomeThings();
```

- Il est même possible de donner un alias à une classe en utilisant « use »

Petite précision

```
namespace Acme\Tools;

class Foo
{
    public function doAwesomeThings()
    {
        echo 'Hello Foo !';
    }
}

// Pas besoin de spécifier le namespace
// car on est déjà dedans
$foo = new Foo();
$foo->doAwesomeThings();
```

- Quand on déclare le namespace en haut de fichier on le déclare pour toutes les lignes suivantes
- On n'a donc pas besoin d'utiliser le nom de fichier complet pour appeler une classe du même namespace

Effets indésirables

- Si vous déclarez un namespace, faites attention aux classes qui vivent dans la racine (comme PDO ou DateTime par exemple)
- Il faudra alors penser à les déclarer avec un back-slash au début (ce qui correspond à leur nom complet)

```
namespace Acme\Tools;

class PDOFactory
{
    public static function getMysqlConnexion()
    {
        // PDO est dans le namespace racine
        $db = new \PDO('mysql:host=localhost;dbname=tests', 'root', 'root');
        $db->setAttribute(\PDO::ATTR_ERRMODE, \PDO::ERRMODE_EXCEPTION);

        return $db;
    }
}
```