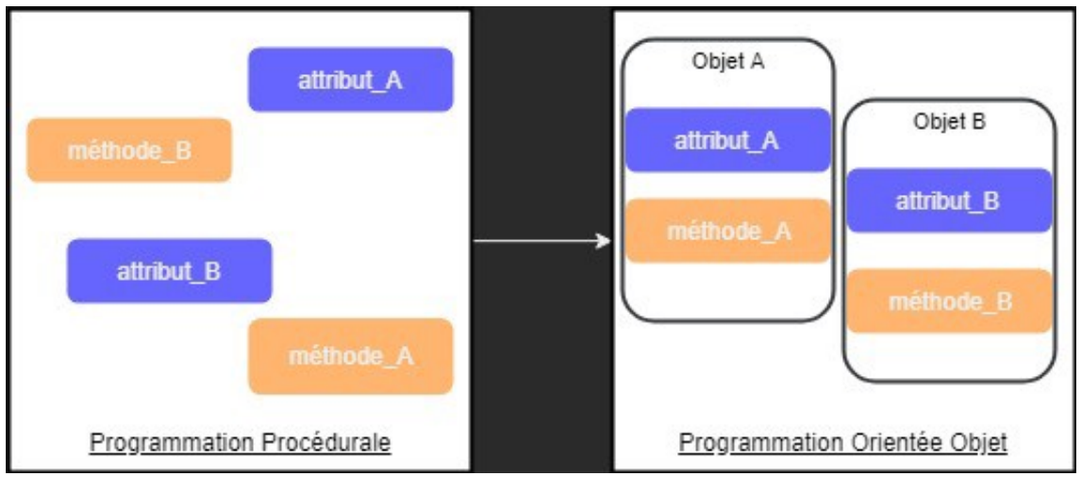
POO

BTS SIO :: SLAM

POO Introduction

- La prog procédurale -> séparation données et traitements
- POO : données & traitements rassemblés dans un même concept
- L’objet est l’élément où l’on retrouve données et traitements



Un objet

- Attention à ne pas confondre avec l'objet de la vie courante. Ici un séjour peut être un objet.

- Un objet est un concept à qui on attribue des valeurs grâce aux variables (on parle d'attributs) mais aussi des traitements ou actions grâce à des fonctions (on parle de méthodes)

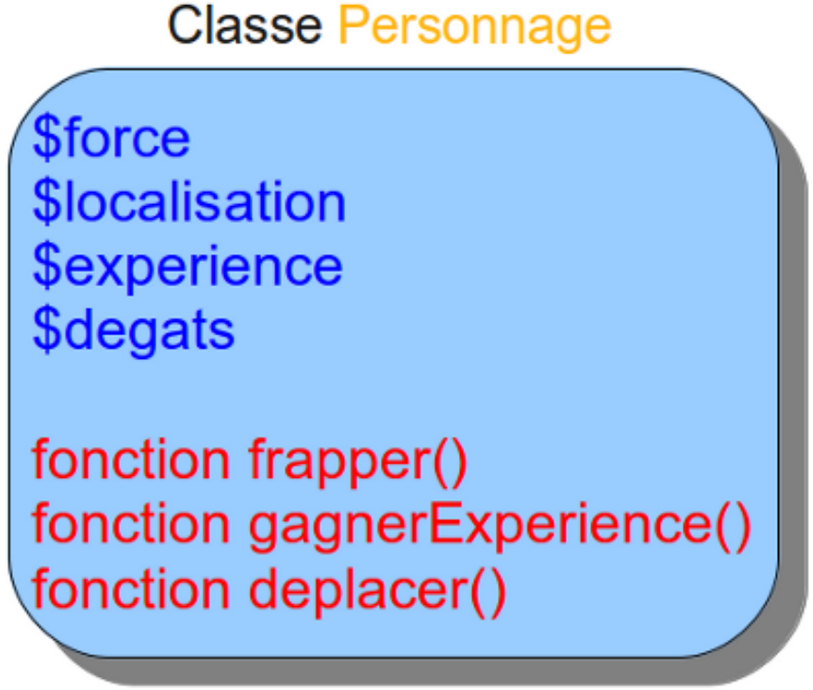
Une classe

- La fabrication de l'objet est confiée à un moule qu'on appelle **une classe** en POO.

- C'est la classe qui contient les éléments (attributs et méthodes) qu'il faudra transmettre pour construire l'objet.

- Pour créer un objet on utilise l'opérateur **new()**

- Le fait de créer un objet à partir d'une classe s'appelle : **instancier**



Une classe

- Comme c’est un objet que l’on va créer on ne doit pourvoir interagir qu’à travers son interface (ses méthodes).

- On va donc rentre inaccessible ses propriétés dans la grande majorité des cas. Pour cela on mettra le mot clef : private.

- On reviendra sur ce principe qu’on appelle l’**encapsulation**.

- Pour accéder à un attribut de notre objet on utilise $this suivi du nom de l’attribut. ex : $this->_nom. Cela évite aussi de confondre avec une variable dans le programme.

Exemple de classe

```php
<?php
class Personnel {
    // -- Propriétés --
    private $_nom = "Dupont";
    protected $prenom = "Pierre";
    public $age = 55;

    // Méthodes
    public function AffichePersonne() {
        echo "Personne : ";
        echo $this->_nom."\t".$this->prenom."\t".$this->age;
        echo PHP_EOL;
    }
}

// Programme
// Instanciation
$salarie = new Personnel();
// Utilisation de la méthode de l'objet
$salarie->AffichePersonne();
?>
```

Norme : notation PEAR -> $_nom pour les données private

L’objet salarie est créé avec le new()
Pour appeler une méthode on utilise le signe ->
On peut donner des valeurs par défaut $_nom = « Dupont »

Utilisation d’une classe

- Si on utilise notre classe comme telle on ferait que des Pierre Dupont
- De plus les attributs sont accessibles.
- On va donc rendre les attributs private
- Découvrir l’intérêt d’un constructeur

Constructeur de classe

```php
<?php
class Personnel {
    // -- Propriétés --
    private $_nom ;
    private $_prenom ;
    private $_age ;

    // Constructeur & destructeur
    function __construct($n, $p, $a) {
        $this->_nom = $n;
        $this->_prenom = $p;
        $this->_age =  $a;
    }

    function __destruct() {
        unset ($this->_nom);
        unset ($this->_prenom);
        unset ($this->_age);
        echo "Destruction de l'objet";

    }

    // Methodes

    public function AffichePersonne() {
        echo "Personne : ";
        echo $this->_nom."\t".$this->_prenom."\t".$this->_age;
        echo PHP_EOL;
    }
}

// Programme
$salarieB = new Personnel("dupond", "jean", 45);
$salarieB->AffichePersonne();
unset($salarieB);
?>
```

Exercices

Exercice 1.1 : Ecrire une classe Point qui permet de représenter un point dans un espace en 2 dimensions grâce à ses coordonnées x et y. Ajouter à la classe **Point** un constructeur prenant en paramètre 2 nombres réels afin d'initialiser ses coordonnées. Ecrire la méthode **affichePoint**() qui permet d'afficher ses coordonnées.

Exercice 1.2 : Pour afficher le point utiliser maintenant la méthode magique : *__toString*().
Regarder ce qu'est une méthode magique en PHP et étudier *__toString*() afin de l'utiliser.

Exercice 1.3 : Ecrire une classe Geométrie qui sera la classe mère des figures géométriques que l'on voudra représenter. Cette classe contient une méthode aire qui retourne l'aire d'une Geométrie sous forme de nombre réel. Par défaut, l'aire d'une Geométrie est 0.

Utilisation d’une classe

- Les méthodes qui permettent d’accéder à la valeur d’un attribut depuis l’extérieur de l’objet sont appelées **accesseurs** ou **getters**.

- Les méthodes qui permettent de modifier les valeurs sont appelées **setters**.

Accès au propriétés

```php
<?php
class Personnel {
    const AGE_MIN = 14;
    const AGE_MAX = 65;

    // -- Propriétés --
    private $_nom="";
    private $_prenom="" ;
    private $_age=0 ;

    // Getter
    public function getNom() {
        return $this->_nom;
    }
    public function getPrenom() {
        return $this->_prenom;
    }
    public function getAge() {
        return $this->_age;
    }

    // Setter
    public function setNom($n) {
        $this->_nom = $n;
    }
    public function setPrenom($p) {
        $this->_prenom = $p;
    }
    public function setAge($a) {
        if (($a>=self::AGE_MIN)&&($a<self::AGE_MAX)) {
            $this->_age=intval($a);
        }
    }

    // Methodes

    public function AffichePersonne() {
        echo "Personne : ";
        echo $this->_nom."\t".$this->_prenom."\t".$this->_age;
        echo PHP_EOL;
    }
}

// Programme
$salarie = new Personnel();
$salarie->setNom("dupont");
$salarie->setPrenom("jean");
$salarie->setAge("25");
echo $salarie->getNom()." ";
echo $salarie->getPrenom()." ";
echo $salarie->getAge().PHP_EOL;
?>
```

Héritage

Lorsqu'on dit que la classe Commercial **hérite** de la classe Personnel, c'est que la classe Commercial **hérite** de tous les attributs et méthodes de la classe Personnel.

Si l'on déclare des méthodes dans la classe Personnel, et qu'on crée une instance de la classe Commercial, alors on pourra appeler n'importe quelle méthode déclarée dans la classe Personnel **du moment qu'elle est publique.**

```php
<?php
class Personnel {
    protected static $NbPersonne ;

    // -- Propriétés --
    private $_nom;
    private $_prenom;
    private $_age;

    public function __construct($n, $p, $a) {
        $this->_nom = $n;
        $this->_prenom = $p;
        $this->_age = $a;
        self::$NbPersonne++;
    }

    // Methode statique
    static function NbrPersonne() {
        return self::$NbPersonne;
    }
    // Methodes

    public function AffichePersonne() {
        echo self::$NbPersonne." Personne : ";
        echo $this->_nom."\t".$this->_prenom."\t".$this->_age;
        echo PHP_EOL;
        echo "<br/>";
    }
}

class Commercial extends Personnel {
    private $_CAArealiser;

    public function __construct($n, $p, $a, $ca) {
        parent::__construct($n, $p, $a);
        $this->_CAArealiser = $ca;
    }

    public function AffichePersonne() {
        parent::AffichePersonne();
        echo $this->_CAArealiser;
    }
}
// Programme
$salarie = new Personnel("Combette","Roland",50);
$salarie1 = new Commercial("Fereira","Antoine",32,5000);
echo $salarie1->AffichePersonne();
?>
```

Exercices

Exercice 1.4 : Ecrire une classe **Rectangle** comme héritant de la classe Geometrie. Un Rectangle est caractérisé par une longueur et une largeur (tous deux des nombres réels). Faire en sorte que la classe redéfinisse correctement la méthode aire et possède un constructeur approprié.
Ajouter à la classe **Rectangle** les méthodes getLongueur et getLargeur qui retournent respectivement la longueur et la largeur du Rectangle.

```php
class Geometrie {
    private $_aire=0;

    public function aire() {
        return $this->_aire;
    }
}
```

Abstraction

- On peut avoir des classes abstraites. Cela veut dire qu'elle ne peuvent servir à instancier un objet. Elles ont un rôle de modèle lors de l'héritage.
  abstract class Personnage

- On peut avoir des méthodes abstraites. Dans la même idée une méthode abstraite à son corps vide. Elle oblige le développeur à écrire cette méthode lors de l'héritage.
  abstract public function()

Exercices

Exercice 1.5 : Ecrire une classe Cercle comme héritant de la classe Géométrie.
La classe géométrie devient abstraite.
Un Cercle est caractérisé par un Point central et un rayon (nombre réel).
Faire en sorte que la classe Cercle redéfinisse correctement la méthode aire et possède un constructeur approprié.
Modifier votre classe Point en la mettant abstraite. Que se passe-t-il ?

Exercices

Exercice 1.5bis : Ne rien changer aux classes mais l'on veut que les classes soient dans des fichiers indépendant donc l'extension est .class.php.
On devra alors utiliser require dans le programme test qui peut s'appeler POOFigures15bis.php

Exercices

Exercice 1.5ter: Si on a 30 ou 40 classes différentes ce qui peut être assez courant cela devient fastidieux de faire pour chaque classe un *require*.
Il existe une manière de les charger automatiquement lors qu'on **intancie** une classe. En cherchant sur le web essayer d'implémenter cette solution.

Exercices

Exercice 1.6 :
Rendre la méthode aire de la classe Geometrie abstraite.

Les constantes de classe

- Comme le nom l'indique ces constantes appartiennent à la classe et non aux objets.

- On utilise l'opérateur de portée :: pour y accéder

- Une constante de classe se déclare avec l'opérateur **const**

- Pour y accéder depuis l'intérieur de la classe on utilise **self::**

- Pour y accéder depuis l'extérieur de la classe on utilise **Classe::CONST**

Les constantes de classe

```php
<?php
class Personnage
{
  private $_force;
  private $_localisation;
  private $_experience;
  private $_degats;

  // Déclarations des constantes en rapport avec la force.

  const FORCE_PETITE = 20;
  const FORCE_MOYENNE = 50;
  const FORCE_GRANDE = 80;

  public function __construct($forceInitiale)
  {
    // N'oubliez pas qu'il faut assigner la valeur d'un attribut uniquement depuis son
setter !
    $this->setForce($forceInitiale);
  }

  public function deplacer()
  {

  }

  public function frapper()
  {

  }

  public function gagnerExperience()
  {

  }

  public function setForce($force)
  {
    // On vérifie qu'on nous donne bien soit une « FORCE_PETITE », soit une « FORCE_MOYENNE
», soit une « FORCE_GRANDE ».
    if (in_array($force, [self::FORCE_PETITE, self::FORCE_MOYENNE, self::FORCE_GRANDE]))
    {
      $this->_force = $force;
    }
  }
}
```

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

Méthodes statiques

- Une méthode statique dépend de la classe et non du ou des objets.

- Pour l'appeler de l'extérieur de l'objet :
  Classe::methode()
  Exemple : **Personnage::parler()**

- Pour l'appeler de l'intérieur de l'objet :
  self::parler()

Méthode static

```php
<?php
class Personnel {
    protected static $NbPersonne ;

    // -- Propriétés --
    private $_nom;
    private $_prenom;
    private $_age;

    public function __construct($n, $p, $a) {
        $this->_nom = $n;
        $this->_prenom = $p;
        $this->_age = $a;
        self::$NbPersonne++;
    }

    // Methode statique
    static function NbrPersonne() {
        return self::$NbPersonne;
    }
    // Methodes

    public function AffichePersonne() {
        echo self::$NbPersonne." Personne : ";
        echo $this->_nom."\t".$this->_prenom."\t".$this->_age;
        echo PHP_EOL;
        echo "<br/>";
    }
}

// Programme
$salarie = new Personnel("Combette","Roland",50);
$salarie1 = new Personnel("Fereira","Antoine",32);
echo Personnel::NbrPersonne();
?>
```

Méthode statique

```php
<?php
class Personnage
{
  private $_force;
  private $_localisation;
  private $_experience;
  private $_degats;

  const FORCE_PETITE = 20;
  const FORCE_MOYENNE = 50;
  const FORCE_GRANDE = 80;

  public function __construct($forceInitiale)
  {
    $this->setForce($forceInitiale);
  }

  public function deplacer(){}

  public function frapper(){}

  public function gagnerExperience(){}

  public function setForce($force)
  {
    // On vérifie qu'on nous donne bien soit une « FORCE_PETITE », soit une « FORCE_MOYENNE », soit une « FORCE_GRANDE ».
    if (in_array($force, [self::FORCE_PETITE, self::FORCE_MOYENNE, self::FORCE_GRANDE]))
    {
      $this->_force = $force;
    }
  }

  // Notez que le mot-clé static peut être placé avant la visibilité de la méthode (ici c'est public).
  public static function parler()
  {
    echo 'Je vais tous vous tuer !';
  }
}
```

$perso = new Personnage(Personnage::FORCE_GRANDE);
$perso->parler();

Attributs statiques

- Un attribut statique dépend de la classe et non du ou des objets.

- Pour l'appeler de l'extérieur de l'objet :
  Exemple : **Personnage::$monAttribut**

- Pour l'appeler de l'intérieur de l'objet :
  self::$monAttribut

Attribut static

```php
<?php
class Personnel {
    protected static $NbPersonne ;

    // -- Propriétés --
    private $_nom;
    private $_prenom;
    private $_age;

    public function __construct($n, $p, $a) {
        $this->_nom = $n;
        $this->_prenom = $p;
        $this->_age = $a;
        self::$NbPersonne++;
    }

    // Methodes

    public function AffichePersonne() {
        echo self::$NbPersonne." Personne : ";
        echo $this->_nom."\t".$this->_prenom."\t".$this->_age;
        echo PHP_EOL;
        echo "<br/>";
    }
}

// Programme
$salarie = new Personnel("Combette","Roland",50);
$salarie->AffichePersonne();
$salarie1 = new Personnel("Fereira","Antoine",32);
$salarie1->AffichePersonne();
?>
```

Exercices

Exercice 1.7 : Ecrire une classe Vehicule qui possède comme propriétés démarré (boolean), une vitesse actuelle et une vitesse maximum pour le véhicule. Cette classe à également 4 méthodes abstraites :

- **demarrer()**
- **eteindre()**
- **ralentir(valeur)**
- **accelerer(valeur)**

Construire une classe Voiture qui hérite de Véhicule. Implémenter les méthodes en essayant d'être le plus près de la réalité. Par exemple :

- **on ne peut accélérer si le moteur n'est pas allumé.**
- **on ne peut ralentir de 30 km/h si on est à 20 km/h.**
- **lorsqu'on accélère on ne peut dépasser la vitesse maximum.**
- **implémenter la méthode de classe (méthode statique) qui donne le nombre de voiture instanciés. On suppose qu'on ne détruit pas de véhicule.**
- **implémenter une constante de classe VITESSE_MAX = 260 qui interdit d'instancier un véhicule avec une vitesse supérieur.**

Pour ceux qui veulent pousser plus loin : on peut interdire ou limiter l'accélération à 20 km/h par appel de fonction ou à 20 % de la vitesse courante.