⬅Revenir à la théorie.

Nouveautés PHP 7 & 8

1. Introduction

PHP 7.x et PHP 8.x ont introduit de sympathiques fonctionalités. Ceci ne rentre pas dans ce cours. Je ne les verrai pas avec vous. Je vous les mets ici. Si cela vous intéresse, lisez et vous verrez comme PHP devient cool. 😀

2. Le typage en PHP

Depuis la version PHP 7.0 fournit la possibilité d'utiliser un typage fort. Chaque nouvelle version de PHP ajoute son lot de nouveautés dans le typage. Il est possible de typer les paramètres des fonctions et des méthodes (fonctions d'une classe) avec int, float, string, bool, array, etc. Il est possible aussi d'indiquer si la fonction retourne un type ou si elle n'en retourne pas (void).

Via le site php.net, voici la liste des types utilisables: Déclarations de type.

Mais afin de ne pas rompre la compatibilité d'anciens scripts, PHP ne force pas le typage par défaut. Mais si on veut vraiment avoir un typage fort, il faut utiliser en haut de nos fichiers PHP la directive suivante:

```
declare(strict_types=1);
```

Avant PHP 7, on devait tester le type d'une variable avec is_int(), is_array(), is_a(), is_subclass_of(),instanceof, etc.

2.1 Typer le paramètre d'une fonction

Avant on devait tester le type pour être certain que cela soit le bon qui était reçu:

```
function times2($entier){
    if(!is_int($entier)){
        throw new exception TypeError();
    }
    return $entier*2;
}
```

Maintenant:

```
function times2(int $integer): int {
    return $entier*2;
}
```

Si la fonction ne reçoit pas un entier, PHP s'arrête. C'est celui qui va utiliser la fonction qui doit faire attention à envoyer le bon type aux paramètres de la fonction.

2.2 Typer la valeur de retour d'une fonction

Il est possible de typer la valeur de retour d'une fonction. Cela permet de directement voir le type qui va être retourné par la fonction.

```php
function getAge(User $user): int {
    return $user->age();
}
```

2.3 Typer les attributs d'une classe

On peut dorénavant typer les attributs d'une fonction. Une propriété peut avoir un type nullable. C'est à dire qu'il peut accepter une valeur nulle. On précède le type d'un point d'interrogation. Dans l'exemple suivant, nous avons deux objets $johnny et $philip qui sont des instances de la classe User:

- $johnny a des enfants stockés dans le tableau $Child: Gabriet et Raphaël.
- $philip n'a pas d'enfant: le tableau $Childs est à null.

```php
class User
{
    public string $Name;
    public string $FirstName;
    public DateTime $Birth;
    public ?array $Childs;

    public function __construct(string $name, string $firstname, DateTime $birth, ?array $
    {
        $this->Name = $name;
        $this->FirstName = $firstname;
        $this->Birth = $birth;
        $this->Childs = $childs;
    }

    public function getAge(): int
    {
        return date_diff($this->Birth, date_create('now'))->y;
    }
}

$johnny = new User("Piette","Johnny", date_create("1974-03-07"),
    array(
        new User("Piette Jacques", "Gabriel", date_create("2014-08-22"), null),
        new User("Piette Jacques", "Raphaël", date_create("2017-03-17"), null)
    )
);

$philip = new User("Dupont","Philip", date_create("1991-01-07"), null);
```

2.4 Fonction sans retour de valeur (void)

void après la définition d'une fonction indique que la fonction ne retourne pas de valeur. Certains m'ont demandé qu'est-ce qu'elle fait si elle ne retourne pas de valeur ? Elle fait tout simplement une action, traîtement.

```php
function foo(): void {
    echo "foo bar";
}
```

Ici l'action, le traîtement est d'afficher quelque chose à l'écran. Mais ça pourrait être un enregistrement dans une base de donnée, un envoie d'emails, etc.

3. Class constructor property promotion

Ici c'est une nouveauté de PHP 8. Il permet dans le constructeur de directement créer les propriétés d'une classe.

Avant:

```php
class Customer {
    public string $nom;
    public string $prenom;
    private ?Address $address;

    public function __construct(string $nom, string $prenom, ?Address $address)
    {
        $this->nom = $nom;
        $this->prenom = $prenom;
        $this->address = $address;
    }
}
```

Maintenant:

```php
class Customer {
    public function __construct(public string $nom, public string $prenom, private ?Addres
}
```

On voit que c'est une syntaxe abrégée. Le fait de recevoir des variables dans le constructeur en

mettant leur visibilité (public, private, protected), php sait qu'il doit directement créer les propriétés en fonction de ce qui a été reçu en paramètres.

On peut aussi avoir un mixte:

```
class Customer {
    public int $age;
    public function __construct(public string $nom, public string $prenom, int $age, priva
        $this->age = $age;
    }
}
```

On peut directement utiliser ces variables dans le corps du constructeur:

```
class Customer {
    public function __construct(public string $nom, public string $prenom, public int $age
        if($age <18){
            throw new Exception("La personne doit être majeure !");
        }
    }
}
```

4. Opérateurs & Affectation

Nous allons donc vite voir comment votre code peut être plus court et tout aussi sûr avec les évolutions de PHP. Un langage de programmation vit et évolue pour le développeur. Ces évolutions lui permettent de lui simplifier son code.

> *Rendez les choses aussi simples que possible, mais pas plus simples.*
> Albert Einstein

4.1 L'opérateur Null coalescent

L'opérateur de fusion null (Null coalescing operator) est très intéressant et nous permet d'éviter de tester si un objet, propriété est null.

Imaginons que nous ayons une méthode hasChild qui indique via un booléen si le user a des enfants.

```
public function hasChild(): bool
{
    return count($this->Childs) > 0;
}
```

Vu que notre tableau peut être null, faire un count dessus n'est pas très propre... Les logs de PHP contiennent le message suivant:

> PHP Warning: count(): Parameter must be an array or an object that implements Countable

On peut régler le problème facilement:

```
public function hasChild(): bool
{
    return count(($this->Childs) ? $this->Childs: array()) > 0;
}
```

On peut aussi utiliser l'opérateur Null coalescent qui s'écrit ?? Si la propriété Childs est nulle alors on crée un array(). Comme ça, on est certain de toujours faire un count sur un tableau. Le count retournera soit la taille du tableau Childs s'il n'est pas vide ou retournera 0 (la taille du tableau vide qui a été créé).

```
public function hasChild(): bool
{
    return count($this->Childs ?? array()) > 0;
}
```

Un autre exemple parlant. On testait avec isset et on combinait avec un opérateur ternaire.

```
$site = isset($_GET['site']) ? $_GET['site'] : 'Pas de valeur';
```

Devient tout simplement:

```php
$site = $_GET['site'] ?? 'Pas de valeur';
```

Cool non ? 😀

4.2 L'affectation Null coalescent

Basé sur le même principe, on peut directement faire l'affectation.

Les deux affectations suivantes donnent le même résultat:

```php
$this->request->data['comments']['user_id'] = $this->request->data['comments']['user_id']
$this->request->data['comments']['user_id'] ??= 'value';
```

On constate que la seconde affectation est plus concise et donc plus propre/facile à lire que la première ligne qui était déjà une belle évolution de PHP.

4.3 L'opérateur Null Safe

Cet opérateur s'écrit ?-> Il est utilisé lorsque l'on essaie d'accéder à un objet/attribut/méthode. Si à gauche de ?-> l'objet/attribut/méthode en cours est null alors la valeur sera nulle et ça n'ira pas plus loin. Ca ne provoquera pas une erreur si on essaie de ?-> sur une méthode alors que précédemment l'objet était à null.
Normalement on doit faire une cascade de tests pour vérifier la valeur nulle.
Partons d'un exemple tout simple:

```php
class Customer {
    public function __construct(public string $nom, public string $prenom, private ?Addres

    public function getAddress(): ?Address {
        return $this->address;
    }
    public function setAddress(Address $address): void{
        $this->address = $address;
    }
}

class Address {
    public function __construct(public string $address, public string $numero, public ?st

    public function getCountry(): string
    {
        return $this->country;
    }
}

$customer1 = new Customer("Piette","johnny", new Address("Rue de la Longue Tige", "1", nu
$customer2 = new Customer("Dupont","Philip", null);

$country1 = $customer1->getAddress()->getCountry();
$country2 = $customer2->getAddress()->getCountry();
```

Ce code va planter pour $country2:

> Fatal error: Uncaught Error: Call to a member function getCountry() on null.

On peut éviter cette erreur avec un ensemble de tests:

```php
$country1 = null;
if($customer1)
    if($customer1->getAddress())
        $country1 = $customer1->getAddress()->getCountry();

$country2 = null;
if($customer2)
    if($customer2->getAddress())
        $country2 = $customer2->getAddress()->getCountry();
```

On peut dorénavant l'écrire de cette manière:

```php
$country1 = $customer1?->getAddress()?->getCountry();
$country2 = $customer2?->getAddress()?->getCountry();
```

Dans ce cas, $country2 prendra la valeur nulle dans deux cas:

- si $customer2 est null
- si getAddress() retourne null

Donc dès qu'il y a l'opérateur ?->, si la valeur à gauche de celui-ci est à null, il n'ira pas plus loin et retournera null pour ce qui est à sa gauche.
Ce code est à nouveau beaucoup plus simple à lire et à maintenir. 😄

5. Fonctions fléchées

A venir quand j'aurai le temps... 😉

⬅Revenir à la théorie.