formation.tech

PHP Objet

PHP Objet - Introduction

- Programmation orientée objet apparue dans les années 70 avec le langage Smalltalk
- Application de principe de tous les jours à la programmation
- Paradigme de programmation (style de programmation)
- Architecture l'application autour des données plutôt que des traitement (programmation procédurale)
- Facilite les tests, la réutilisation de code, le travail à plusieurs

PHP Objet - Classe vs Objet

- Classe
  - Un concept (Ordinateur)
  - Un type
  - Permet la création d’objets (un moule)
- Objet
  - Représentation en mémoire d’une classe (instance de classe)
  - Plusieurs objets possibles pour une classe

PHP Objet - Classe et Type

- Une classe permet la déclaration d'un nouveau type complexe
- Peut contenir plusieurs valeurs stockées dans des (nom au choix) :
  - propriétés
  - attributs
  - champs

```php
<?php

class Computer
{
    public $brand;
    public $model;
}

$macbook = new Computer;
$macbook->brand = 'Apple';

echo "This MacBook was built by $macbook->brand";
```

PHP Objet - Méthodes

- Une classe peut également contenir des fonctions appelées « méthodes »
- Le mot clé this est une référence interne à l'objet (permet d'accéder aux autres propriétés et méthodes de l'objet sur lequel la méthode d'origine a été appelée)

```php
<?php

class Computer
{
    public $brand;
    public $model;

    public function showInfos() {
        echo "This $this->model was built by $this->brand";
    }
}

$macbook = new Computer;
$macbook->brand = 'Apple';
$macbook->model = 'MacBook';
$macbook->showInfos();
```

PHP Objet - Visibilité

- Chaque membre (propriété, méthode) peut se déclarer avec une visibilité
  - public : accessible dès lors que l'objet est accessible
  - protected : accessible depuis la classe, les classe ascendante ou descendante (voir le chapitre sur l'Héritage)
  - private : uniquement à l'intérieur de la classe

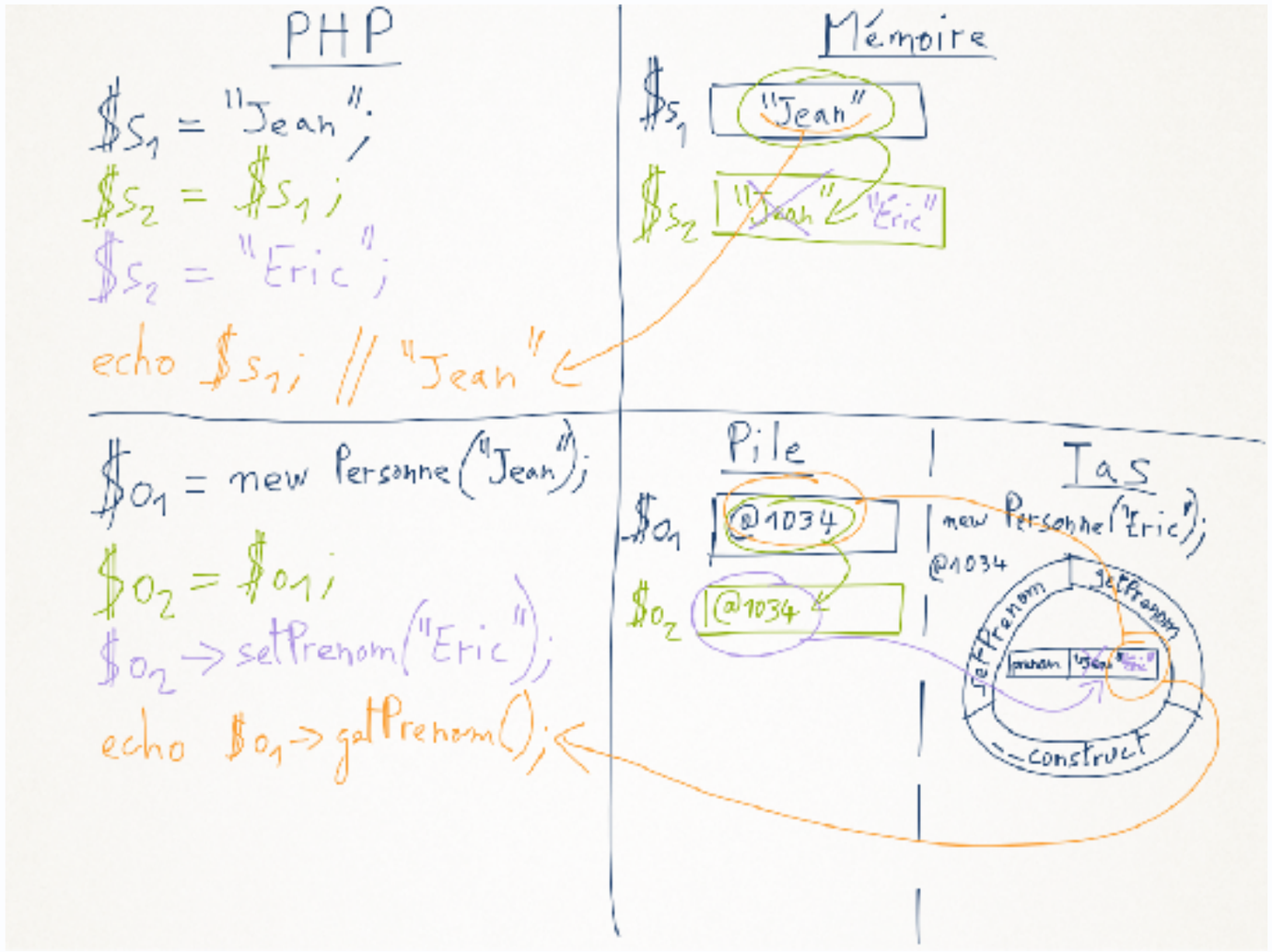
PHP Objet - Principe d’encapsulation

- Bonne pratique : ne jamais rendre les propriétés publiques
- Un objet peut représenter un concept complexe, il faut qu’à l’utilisation un autre développeur voit un API le plus simple possible
- Masquer les propriétés permet de choisir si elles sont accessibles en lecture et/ou écriture
- Ecrire via un setter permet d’exécuter du code de filtrage/validation à l’intérieur de la classe

```php
<?php

class Computer
{
    protected $brand;

    public function getBrand(): string
    {
        return $this->brand;
    }

    public function setBrand(string $brand): Computer
    {
        $this->brand = $brand;
        return $this;
    }
}
```

PHP Objet - Référence

- la différence des types scalaires (*int*, *boolean*, *float*, *string*, *array*…) les objets se manipulent au travers de références
- une référence permet de retrouver l'objet en mémoire
- seule les opérateurs *new* et *clone* permettent de construire de nouveau objets



PHP Objet - Namespace

- Pour éviter les conflits sur les noms de classe, PHP 5.3 introduit la notions de namespace
- Un namespace est un dossier virtuel qui vient préfixer le nom de la classe

```php
<?php

namespace FormationTech\Model;

class Contact
{

}
```

PHP Objet - Namespace

- A l'utilisation il faut utiliser le Fully Qualified Class Name (FQCN ou FQN)

```php
<?php

require_once './Contact.php';

$romain = new FormationTech\Model\Contact();
```

- On peut également raccourcir les lignes en utilisant au début de chaque fichier le mot clé use, suivi du FQCN

```php
<?php

use FormationTech\Model\Contact;

require_once './Contact.php';

$romain = new Contact();
```

PHP Objet - Namespace

- On peut également créer un Alias (nécessaire si 2 classes portent le même nom)

```php
<?php

use FormationTech\Model\Contact as ContactModel;

require_once './Contact.php';

$romain = new ContactModel();
```

- Les namespaces peuvent également être aliassés

```php
<?php

use FormationTech\Model as M;

require_once './Contact.php';

$romain = new M\Contact();
```

PHP Objet - Autochargement

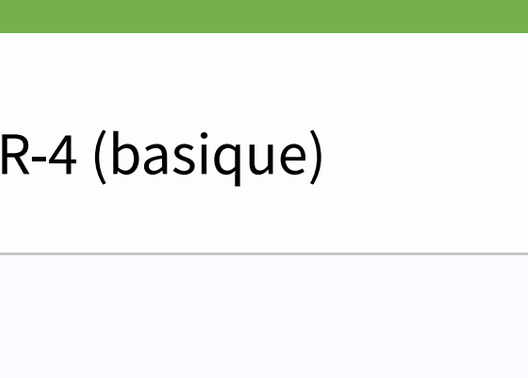
- Pour éviter d’inclure chaque classe avec un require, on peut créer un autoloader
- Il faut alors avoir une corrélation entre le nom la classe et le chemin sur le disque
- Historiquement chaque bibliothèque vient avec sa propre convention donc sont autoloader (problématique puisque certains s’exécuteront inutilement)
- Le PHP Framework Interop Group (PHP-FIG) a été créé pour que les développeurs de bibliothèques PHP s’accordent sur des normes
- PSR-0 (dépréciée) puis PSR-4 sont des normes d’autochargement
- Dans PSR-4 il faut associer un namespace préfixe à un répertoire sur le disque, à chaque namespace suivant doit correspondre une dossier, à chaque classe un fichier suffixé par .php
- PSR-4: Autoloader
  https://www.php-fig.org/psr/psr-4/

PHP Objet - Autochargement

- Exemple d'autoloader PSR-4 (basique)

```php
<?php

$prefixes = [
    'FormationTech\\' => __DIR__ . '/lib/',
];

spl_autoload_register(function ($fqcn) use ($prefixes) {

    $path = strtr($fqcn, $prefixes) . '.php';
    $path = strtr($path, '\\', DIRECTORY_SEPARATOR);

    @include_once $path;
});
```

- A l'utilisation




```php
<?php

use FormationTech\Model\Contact;

require_once 'autoload.php';

$romain = new Contact();
```

PHP Objet - Associations

- Pour lier des objets entre eux on utilise des associations
- Il suffit en PHP de stocker dans un objet la référence vers un autre objet
- Si plusieurs objets sont liés, les mettre dans un tableau, ou un classe de Collection comme *\SplObjectStorage*
- Si l'association n'est possible que dans un sens (classe A → classe B) on par d'association unidirectionnelle
- Si elle est possible dans les 2 sens (classe A ↔ classe B) on par d'association bidirectionnelle (ne le faire que si nécessaire)

```php
<?php

namespace FormationTech\Model;

class Contact
{
    /** @var string */
    protected $firstName;

    /** @var string */
    protected $lastName;

    /** @var Company */
    protected $company;
}
```

```php
<?php

namespace FormationTech\Model;


class Company
{
    /** @var string */
    protected $name;
}
```

PHP Objet - Associations

- Pour réaliser l'association vers une référence on utilise un setter

```php
<?php

namespace FormationTech\Model;

class Contact
{
    /** @var Company */
    protected $company;

    public function getCompany(): Company
    {
        return $this->company;
    }

    public function setCompany(Company $company): Contact
    {
        $this->company = $company;
        return $this;
    }
}
```

- TODO exemple utilisation

PHP Objet - Héritage

Héritage

Une classe réutilise les membres d'une autre classe.

```php
<?php

namespace Application\Entity;


class Contact
{
    protected $id;
    protected $prenom;
    protected $nom;
    protected $email;
    protected $telephone;

    /**
     * @var Societe
     */
    protected $societe;
}
```

```php
<?php

namespace Application\Entity;


class Salarie extends Contact
{
    protected $salaire;
}
```

PHP Objet - Interface

- ## Interface

  La définition d'une liste de méthodes, implémenter une interface oblige à implémenter ses méthodes.

  Dans ZF2 : leur nom se termine toujours par Interface

```php
<?php

namespace Application\Entity;

use Zend\Stdlib\ArraySerializableInterface;

class Contact implements ArraySerializableInterface
{
    // ...

    public function exchangeArray(array $array)
    {
        foreach ($array as $key => $value) {
            if (property_exists($this, $key)) {
                $this->$key = $value;
            }
        }
    }

    public function getArrayCopy()
    {
        return get_object_vars($this);
    }
}
```

```php
<?php

namespace Zend\Stdlib;

interface ArraySerializableInterface
{
    public function exchangeArray(array $array);

    public function getArrayCopy();
}
```

PHP Objet - Classe abstraite

- ## Classe abstraite

  Une classe qui n'a pour vocation à être utilisé qu'au travers d'un héritage. Peut également imposer l'implémentation de certaines méthodes comme une interface.
  Comme pour une interface, c'est la garantie que certaines méthodes seront bien présentes (programmation par contrat).
  Dans ZF2 leur nom commence toujours par Abstract.

```php
<?php

namespace Zend\Mvc\Controller;

abstract class AbstractActionController extends AbstractController
{
    // ...
}
```

```php
<?php

namespace AddressBook\Controller;

use Zend\Mvc\Controller\AbstractActionController;

class ContactController extends AbstractActionController
{
// ...
}
```

PHP Objet - Classe abstraite

- PHP FIG
  PHP Framework Interop Group, groupe de travail regroupant les principaux créateurs de frameworks/bibliothèques créant des normes PHP.
  http://www.php-fig.org/
- PSR-1
  PSR-0 + Basic Coding Standard
- PSR-2
  PSR-1 + Coding Style Guide
- PSR-3
  Logger Interface
- PSR-4
  Improved Autoloading
- PSR-7
  HTTP Message Interface

PHP Objet - Composition

Composition

Une composition est un type d'association forte entre 2 objet. La destruction d'un objet entrainerait la destruction de l'objet associé.
Exemple : Un objet Tasse est composée de Café

```php
<?php

namespace ExpressoComposition;

class Cafe
{
    protected $variete;
    protected $provenance;

    public function __construct($provenance, $variete)
    {
        $this->provenance = $provenance;
        $this->variete = $variete;
    }
}
```

```php
<?php

namespace ExpressoComposition;

class Tasse
{
    protected $contenu;

    public function __construct() {
        $this->contenu = new Cafe("Arabica", "Mexique");
    }
}
```

```php
<?php

require_once 'autoload.php';

$tasseDeCafe = new \ExpressoComposition\Tasse();
```

Mauvaise Pratique

La composition est désormais considéré comme une mauvaise pratique. Premièrement la classe Tasse n'est très réutilisable, elle ne peut contenir que du café. De plus il n'est pas possible d'écrire d'écrire un test unitaire de Tasse, puisqu'il faudrait en même temps tester Café.
Globalement il faut essayer de proscrire l'utilisation de new il l'intérieur d'une classe (à l'exception des Values Objects (DateTime, ArrayObject, etc…)

PHP Objet - Injection de Dépendance

- ## Solution

  La solution est simple, pour éviter le new dans cette classe nous allons injecter la dépendance.

```php
<?php

namespace ExpressoInjection;

interface Liquide {

}
```

```php
<?php

namespace ExpressoInjection;

class Cafe implements Liquide
{
    protected $variete;
    protected $provenance;

    public function __construct($provenance, $variete)
    {
        $this->provenance = $provenance;
        $this->variete = $variete;
    }
}
```

```php
<?php

namespace ExpressoInjection;

class Tasse
{
    protected $contenu;

    public function __construct(Liquide $contenu) {
        $this->contenu = $contenu;
    }
}
```

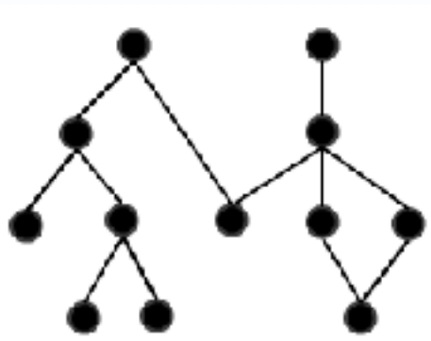
```php
<?php

require_once 'autoload.php';

$cafe = new \ExpressoInjection\Cafe();
$tasseDeCafe = new \ExpressoInjection\Tasse($cafe);
```

- La classe Tasse peut désormais recevoir n’importe quel contenu qui implémente l’interface Liquide.

PHP Objet - Injection de Dépendance

- ## Conteneur d'injection de dépendance (DIC)

  Le problème lorsqu'on injecte les dépendances est qu'on peut parfois se retrouver avec des dépendances complexes :



- Dans ce cas il devient utile d'utiliser un conteneur d'injection de dépendance qu'on aura configuré au préalable. En PHP il existe quelques DIC connus :
  - Pimple par Fabien Potencier
  - Dice par Tom Butler
  - PHP-DI par Matthieu Napoli
  - Symfony\Container intégré Symfony2
  - Zend\Di & Zend\ServiceManager intégrés ZF 2
- Zend\Di est très simple à configurer mais peu performant (utilise la Reflection), Zend\ServiceManager est le plus utilisé.
  Documentation : http://framework.zend.com/manual/current/en/modules/zend.service-manager.quick-start.html